Assessment of Electrical Thruster Configurations for Lunar CubeSat Attitude Control

Thesis Report

Master Thesis Aerospace Engineering

Pieter de Lange

Electronic Systems Laboratory

Assessment of Electrical Thruster Configurations for Lunar CubeSat Attitude Control

Thesis Report

by

Pieter de Lange

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday February 6, 2025 at 13:45.

Student number:	4678362	
Project duration:	March 4, 2024 – February 6, 2025	
Thesis committee:	Dr. ir. A. Cervone,	TU Delft, supervisor
	Dr. ir. J. Guo,	TU Delft, chair
	Dr. ir. E. van Kampen,	TU Delft, external supervisor

Cover: Illustration of the LUMIO mission, ESA

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Preface

After having spent the past eleven months on the research that lies before you, and now finally having put together all its individual pieces, one question popped into my head: why do people do this? What is the intrinsic motivation of humankind to constantly be looking for the unknown? The answer is quite simple, really. It is the same question of why WALL-E travelled into space with EVE, or why Paul Artreides devoted himself to studying the Fremen: curiosity. As long as people remain curious about the topics they are most fascinated about, we keep on innovating and reaching higher summits. As long as we remain curious, unthinkable events like human colonisation of Mars and intergalactic travel slowly come within grasp. In the world of curiosity, anything is possible.

The work before you could not have been written without the help of my supervisor, Angelo Cervone, whom I would like to thank for his full support throughout the entire process. Your critical view on the extensive information I kept sending your way was critical for the success of my thesis, and taught me a great deal about conducting research. In addition, I would like to thank Willem Jordaan for his great ideas, enthusiasm and further support during my time at the Electronic Systems Laboratory. Without him, and all the other great people I met there, my thesis would not have become what it is today, and would definitely not have been as much fun. Thank you all.

Pieter de Lange
Stellenbosch, January 2025

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

ADC	Analogue-to-Digital Convert
ADCS	Attitude Determination and Control System
API	Application Programming Interface
C&DH	Command and Data Handling
CAD	Computer Aided Design
CoM	Centre of Mass
COTS	Commercial-Off-The-Shelf
CPU	Central Processing Unit
DC	Duty Cycle
DSM	Deep Space Manoeuvres
EKF	Extended Kalman Filter
EPS	Electrical Power System
ESA	European Space Agency
GPIO	General-Purpose In/Out
HIL	Hardware-In-the-Loop
I^2C	Inter-Integrated Circuit
IMU	Inertial Measurement Unit
JAXA	Japan Aerospace Exploration Agency
LQ	Literature Question
LRO	Lunar Reconnaissance Orbiter
LVLH	Local Vertical Local Horizontal
MCU	Micro Controller Unit
MIB	Minimum Impulse Bit
NA	Not Applicable
NASA	National Aeronautics and Space Administration
OBC	On-Board Computer
ODE	Ordinary Differential Equation
PCB	Printed Circuit Board
PID	Proportional-Integral-Derivative

PPT	Pulsed Plasma Thruster
PWM	Pulse Width Modulation
PWPFM	Pulse-Width-Pulse-Frequency Modulation
RCS	Reaction Control System
RK4	Runge-Kutta 4
RO	Research Objective
ROB	Robustness
RQ	Research Question
RTOS	Real-Time Operating System
SE	Systems Engineering
SIM	Simulation
SLIP	Serial Line Internet Protocol
SLS	Space Launch System
SPI	Serial Peripheral Interface
SQ	Sub-Question
SRP	Solar Radiation Pressure
TRL	Technology Readiness Level
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
VAT	Vacuum Arc Thruster
WSB	Weak Stability Boundary

Symbols

α	Optimal solar array angle w.r.t. the Sun [rad]
β	Half-cone offset angle [rad]
Δ	Change in ...[-]
$\dot{m}$	Mass flow [kg/s]
ϵ	Integrator error
η_{ion}	Ion thruster efficiency [-]
ω	Angular velocity [rad/s]
ω_{rw}	Angular velocity of reaction wheel [rad/s]
ϕ	Roll angle [rad]
ψ	Yaw angle [rad]
ρ_d	Diffuse reflection coefficient [-]
ρ_s	Specular reflection coefficient [-]

θ	Pitch angle [rad]
θ_n	Euler angles [rad]
a_n	Orthogonal unit vector axes in the LVLH frame [-]
A_{panels}	Spacecraft panel areas, matrix [m^2]
A_{rw}	Reaction wheel configuration matrix [-]
A_{thrust}	Thruster mixing matrix [-]
B	Magnetic field [T]
b_n	Orthogonal unit vector axes in the body frame [-]
C	Direction cosine matrix [-]
c_p	Spacecraft panel centre of pressure locations, matrix [m]
D	Diameter of holes in ion thruster charged grids [m]
E	Energy [J]
e	State error
F	Force [N]
$f_{natural}$	Natural frequency [Hz]
F_{thrust}	Thrust force [N]
h	Angular momentum [Nms]
h_{rw}	Angular momentum of reaction wheel [Nms]
I	Mass moment of inertia matrix [kg m^2]
I_s	Total solar irradiance [W/m^2]
I_{rw}	Mass moment of inertia matrix of reaction wheel [kg m^2]
I_{sp}	Specific impulse [s]
J	Linear impulse [Ns]
K_d	Derivative gain [-]
K_i	Integral gain [-]
k_n	Runge-Kutta 4 intermediate terms
K_p	Proportional gain [-]
L	Distance between holes in ion thruster charged grids [m]
m	Mass [kg]
m_O	Initial spacecraft mass [kg]
m_P	Propellant mass [kg]
M_{res}	Spacecraft residual magnetic moment [A m^2]
n_n	Orthogonal unit vector axes in the Newtonian inertial frame [-]
n_s	Unit vectors pointing normal to panel, matrix [-]

P	Power [W]
p_a	Ambient pressure [Pa]
p_e	Pressure at nozzle exit [Pa]
q_e	Error quaternion vector [rad]
q_n	Quaternion vector [rad]
q_{ref}	Reference quaternion vector [rad]
r	Position vector [m]
$r_{\text{S/M}}$	Position of the Sun with respect to the Moon [m]
$r_{\text{SC/M}}$	Position of the spacecraft with respect to the Moon [m]
S	Unit vectors from Sun to spacecraft panel, matrix [-]
t	Time [s]
T_c	Control torque [Nm]
T_d	Disturbance torque [Nm]
T_{GG}	Gravity gradient torque [Nm]
T_{mag}	Magnetic disturbance torque [Nm]
T_{SRP}	Solar radiation pressure torque [Nm]
u	Output control variable
v	Velocity [m/s]
v_{eq}	Equivalent exhaust velocity [m/s]
v_e	Exhaust velocity [m/s]

Physics constants

μ_{Earth}	Gravitational parameter of Earth ($3.986004418 \cdot 10^{14}$ [m^3 s^{-2}])
μ_{Moon}	Gravitational parameter of Moon ($4.9048695 \cdot 10^{12}$ [m^3 s^{-2}])
μ_{Sun}	Gravitational parameter of Sun ($1.32712440018 \cdot 10^{20}$ [m^3 s^{-2}])
π	3.1415926535 [-]
ε_0	Permittivity of vacuum ($8.8542 \cdot 10^{-12}$ [F/m]
c	Speed of light ($2.99792458 \cdot 10^8$ [m/s])
G	Universal Gravitational Constant ($6.67428 \cdot 10^{-11}$ [m^3 kg^{-1} s^{-2}])
g_0	Standard acceleration of gravity (9.80665 [m/s^2]
P_{solar}	Power exerted by the Sun ($3.842 \cdot 10^{26}$ [W])
R_{Earth}	Earth mean radius ($6.3781 \cdot 10^6$ [m])
R_{Moon}	Moon mean radius ($1.7374 \cdot 10^6$ [m])
R_{Sun}	Sun mean radius ($6.957 \cdot 10^8$ [m])
AU	Astronomical unit ($1.495978707 \cdot 10^{11}$ [m])

p	Standard PocketQube Unit ($5 \times 5 \times 5$ [cm])
U	Standard CubeSat Unit ($10 \times 10 \times 10$ [cm])

Units

$^\circ$	degree
A	Ampère
F	Farad
Hz	Hertz
J	Joule
kg	kilogramme
m	meter
N	Newton
Pa	Pascal
rad	radian
s	second
T	Tesla
V	Volt
W	Watt
%	percent

Executive Summary

Over the past twenty years, miniaturisation in spacecraft technology has been a prominent topic of interest; a smaller object introduces less mass and volume to be taken aboard a launch vehicle compared to its larger counterpart, leading to significant cost benefits. In addition, rapid development is possible for standardised units. Standard form factors (U, p) have been developed, with the CubeSat and PocketQube models to be developed according to those sizes. The smaller sizes of overall spacecraft systems has led to equal miniaturisation of the subsystems, of which the propulsion system and Attitude Determination and Control System (ADCS) are critical for mission success. The research laid out in this report is looking at both of these aspects, and will attempt to introduce novelty by assessing the application of electrical thrusters within ADCS, an under-explored topic that should be investigated in order to take research on electrical micro-propulsion units a step further.

Before any technical research can commence, a literature study is carried out that tries to answer the following literature questions:

LQ-01 : What are the past, current and future developments in lunar CubeSat missions?

LQ-02 : What spacecraft dynamics relations need to be taken into account for the development of a CubeSat attitude control algorithm?

LQ-03 : For an electrical-thruster-based attitude control system applied to a CubeSat mission, in which way can thruster configurations be assessed on their feasibility in the required mission profile?

LQ-04 : What are the critical factors to consider when testing thruster hardware modules to ensure effective software-hardware integration for algorithm validation?

LQ-05 : What are the working principles of vacuum arc thrusters?

LQ-06 : What are the current vacuum arc thruster applications in CubeSat missions?

From the study, it was evident that a large number of lunar CubeSats have been launched in the past. On the Artemis 1 mission, for example, NASA launched five 6U spacecraft to the Moon or in a lunar orbit. Their applications ranged from collecting surface spectroscopy data to landing a small Moon lander. More recently, NASA's CAPSTONE mission was launched and operated in a near recti-linear halo orbit about the Earth-Moon L_2 Lagrange point. CAPSTONE's mission is notable, since it assesses the stability of this orbit for future use of NASA's lunar Gateway, a space station that will become the intermediate station for travel to the Moon. From recent data, the orbit is proven to be stable and the CAPSTONE mission was an overall success. With respect to the future, the European Space Agency's LUMIO mission will be launched in 2027, with its primary goal to detect meteoroid impacts on the far side of the Moon. The data collected by LUMIO will aid in developing meteoroid impact models that can predict dangerous situations for human colonisation on the Moon, but also impacts into the Earth's atmosphere.

Next, the literature study dives into the Euler's equation of rotational motion for a rigid body to describe a spacecraft's motion during attitude control. Disturbances in lunar orbit include the gravity gradient torque and solar radiation pressure, and quaternions can be used to express the attitude of an object without the singularities introduced by use of Euler angles. Different control laws are used within spacecraft applications, but simple proportional-integral-derivative laws pose benefits in terms of simplicity and accuracy. An integrated algorithm that includes the adequate dynamics and control laws should send signals to the appropriate actuators. Different types exist, such as magnetorquers, thrusters and reaction wheels. Thrusters can be sub-divided in the electrical types, with the Hall Effect thruster, ion thruster and vacuum arc thruster as examples. For non-electrical thrusters, mono-propellant, solid propellant and resistojet thrusters are examples.

Implementation of electrical thrusters as means for ADCS only can be assessed based on the pointing accuracy of the spacecraft, which can be measured by the angle between the desired pointing vector

(e.g. a camera pointing towards the Earth) and the actual pointing vector. Moreover, angular velocity limits can be tested against maximum value requirements, as well as the power and energy consumption of the system compared to the overall system usage. Whenever these hardware modules are tested in real-life, its accuracy of processing commands is of great importance and signal discrepancies should be tested. Upon reviewing electrical thruster modules, the Vacuum Arc Thruster (VAT) was further looked in to since their response to high-frequency signals allows for modification of the thrust output. VATs have an anode and cathode, between which an electric arc is created due to sudden discharges. This arc releases electrons from the cathode and are accelerated by the induced electric field, creating thrust. By altering the frequency or duration of these discharges, thrust output can be varied, which is convenient for usage in precise control algorithms. Such an application has been studied in the UWE-4 mission by the University of Würzburg.

From the literature study, a knowledge gap is identified with respect to the ADCS of CubeSat missions: most of the designs include reaction wheels as main actuators with non-electrical thrusters (monopropellant, cold gas, etc.) for angular momentum desaturation. Electrical micro-propulsion units should be investigated and applied more often to increase their technology readiness level, and the ADCS of ESA's LUMIO mission offers an opportunity to completely replace its current design by an electrical-thruster-only system. The lunar environment introduces an interesting test case. Based on this knowledge gap, the following main research question has been established:

What is the impact of adjusting the ADCS configurations, consisting of electrical thrusters only, on the LUMIO mission, a 12U lunar CubeSat, on its attitude control performance, robustness and connectivity?

In order to assess the *performance* aspect of the question, as well as overall feasibility within the LUMIO mission, an attitude control algorithm is developed. The basis for this is a PD controller, including Euler's equation for rotational dynamics with the gravity gradient and solar radiation pressure torques as disturbances. Since the orbital parameters for the LUMIO mission are not publicly available, the orbit attained by the CAPSTONE mission is used, which is similar but poses larger stress on the analysis due to its close fly-bys. Four different electrical thruster configurations are assessed, each containing at least three thruster pairs for attitude control. Differences between the configurations are based on whether each pair controls one primary spacecraft axis, or multiple at the same time. In addition, redundancy is introduced in configurations 3 and 4, to assess the effect of relaxing the performance limits in this way. The hardware module that is tested in these configurations, is the Pocket Rocket VAT developed by Solid State Propulsion from South Africa. Its maximum thrust output is 200 [μN], operating at power levels between 0.25 - 20 [W] (equal to 10 [mN/kW]).

The thruster configurations will be assessed in the CAPSTONE orbit over a simulation time of two weeks. During this period, the spacecraft will fly closely past the Moon twice, and attitude corrections due to this close fly-by will be analysed. In addition to the electrical thruster configurations, reaction wheels as presented in the original LUMIO mission design will also be tested as a base case, to make effective power and energy consumption comparisons. Four wheels are used in total, of which three are aligned perfectly with the spacecraft primary rotation axes, and one is aligned with all of these axes equally. In order to test extreme cases with respect to the control algorithm, single thruster failure, solar array deployment and de-tumbling manoeuvres will be introduced for the electrical thruster configurations, to assess whether they are capable of handling the scenarios adequately, and if so, what the induced accuracy, power and energy costs are. These *robustness* tests will further assess the thruster configuration feasibility and behaviour.

Code verification and validation to ascertain the proper functioning of the control algorithm while testing the reaction wheel and electrical thruster configurations is done based on multiple unit tests, integrator comparisons, orbit visualisation and algorithm response behaviour. In addition, an experimental set-up is created that validates the functioning of the code on existing hardware modules and testing their *connectivity*. An STM32 Nucleo development board is introduced that acts as the spacecraft OBC, along with a solenoid valve that acts as one of the thrusters within configuration 1. First of all, the calculation of the control torque, the reference quaternion and the thrust output values per thruster is ported from the simulation environment (Python) to the embedded environment (C). After proper verification of these functions on the OBC is completed, connection towards the solenoid valve is made, and required signals are pulse-width modulated so that the desired output thrust is achieved by adjusting the signal duty cycle. Finally, when proper functioning of the valve is observed, the number of output signals is extended to 6, so that signals for each thruster in the configuration are created. Throughout this experiment, numerical comparison between the ported functions is performed, as well as duty cycle values over time for hardware signalling. These duty cycle values stem directly from the signals sent from the Nucleo board and can be compared to the desired signals.

The results from the nominal simulation duration proved to be successful with respect to implementation of reaction wheels as well as for the vacuum arc thruster configurations. An important note to be taken here is that both their resolutions may not be suitable for the relatively low required control torques (order of magnitude of 10^{-7} $[Nm]$). With respect to the most important results, the following summary can be given on the attitude control simulation and robustness results:

- **Mission pointing requirement**: The quaternion components of the spacecraft closely adhered to the reference quaternion throughout the simulation, with relative quaternion errors remaining below 0.001% during close fly-bys. The half-cone angle offset consistently met the LUMIO mission requirement of 0.18°.
- **Reaction wheel performance**: Reaction wheels successfully controlled the spacecraft without saturating over the two-week simulation. The maximum angular momentum build-up of $6 \cdot 10^{-4}$ $[Nms]$ was well below the wheel capacity of 0.1 $[Nms]$. However, for extreme scenarios, such as de-tumbling or long-term operation, additional momentum dumping actuators will likely be required. The reaction wheel system operated efficiently within the power budget, consuming only $2.5 \cdot 10^{-4}$ $[W]$ during nominal operations.
- **Thruster configuration comparison**: In nominal scenarios, electrical thruster configurations successfully adhered to the control requirements, with configuration 4 showing the best performance, reducing total energy consumption by 29% and maximum power by 36% compared to configuration 1. All configurations remained within the 0.18° half-cone offset angle requirement set out by the LUMIO mission over the entire simulation time span. In general, it could be concluded that adding redundant thrusters enhances the energy consumption of a configuration compared to a determinate system, and that adding thruster pairs controlling multiple spacecraft primary axes simultaneously is advantageous compared to thruster pairs only responsible for one axis.
- **Mass budget**: Thruster-based configurations offer potential mass savings compared to the current LUMIO ADCS, with configurations 1 and 2 reducing the spacecraft's wet mass by 4.06%. However, these savings are valid only for nominal conditions and do not account for extreme scenarios, which would require additional hardware or modifications.
- **Single thruster failure**: Approach 1 (with system awareness of failures) allowed the system to maintain functionality, although redundancy (e.g., configuration 4) is essential to handle failures without compromising control. Approach 2 (without system awareness) induced critically high power and energy demands, rendering the system unfeasible. This highlights the necessity of real-time thruster health monitoring and communication with the onboard computer.
- **Disturbance torque reflection**: Disturbance torques from gravity gradient and solar radiation pressure were insignificant compared to the control torque. However, the disturbance torque from main engine firings, not included in this research, could significantly affect the ADCS performance and should be included in future analyses.
- **Solar array deployment**: Retracting solar arrays significantly reduced ADCS power and energy requirements, with average reductions of 59% in power and 62% in energy across all configurations. This scenario is highly beneficial for early mission phases or contingency cases.
- **De-tumbling**: The thruster-based ADCS struggled to control high initial angular velocities due to instantaneous torque limitations. This performance deficit severely restricts their practical application as a replacement for the reaction wheel system and would not be advised by this research. Enhancing the control algorithm to distribute corrections over longer time spans is recommended for future studies.

The practical experimentation carried out in this research successfully validated the simulation-based control algorithm on embedded hardware, demonstrating the feasibility of integrating electrical thruster modules into real-world systems. Using an STM32 Nucleo development board and a solenoid valve as a hardware proxy, the developed algorithm reliably generated control signals and modulated them to achieve the desired thrust output. Minor deviations between simulated and experimental results were attributed to solver precision and signal processing limitations but were within acceptable error margins. The same behaviour was observed for the output of six individual signals for the thrusters in configuration 1.

Based on the aforementioned conclusions, replacing the current LUMIO ADCS with electrical thruster configurations as the only means of attitude control is only feasible within the nominal mission scenarios. Within these, thruster configurations that are coupled to multiple axes simultaneously and redundant systems pose the best options. Still, these nominal scenarios require excessive power and energy from the system compared to the reaction wheel base case: total energy consumption was in range 1.5 to 2.0 $[kJ]$ for the thruster configurations, whereas they were only a couple of Joules for the reaction wheels. Unfortunately, realistic de-tumbling manoeuvres are not possible with any of the four configurations which makes them unsuitable for use on board the LUMIO mission. The hardware validation did, however, prove that the current attitude control algorithm is suitable for real-life application. Future research should focus on including additional disturbances such as the main engine parasitic torque, adjusting the VAT hardware modules to be more suitable within the mission limits, and focus on reaction wheel systems in combination with electrical thrusters, for desaturation.

Ultimately, this work contributes to the broader goal of advancing CubeSat technology, emphasizing the need for continued research into electrical propulsion systems. By addressing the identified gaps and refining the approaches presented here, future missions may unlock the full potential of these systems, enabling precise and efficient attitude control in lunar and deep-space environments.

Abstract

This research investigated the application of electrical thrusters within the Attitude Determination and Control System (ADCS) of the European Space Agency's LUMIO mission, a 12U CubeSat set to operate in a quasi-periodic halo orbit around the Earth-Moon L_2 point. While CubeSats have become pivotal tools for lunar exploration, the use of electrical thrusters for ADCS, particularly in the lunar environment, remains under-explored. This study aimed to evaluate the impact of replacing LUMIO's current ADCS design by electrical thruster configurations on attitude control performance, robustness, and connectivity, bridging a critical gap in current literature.

A simulation framework was developed to assess spacecraft performance over a two-week period, examining the LUMIO mission reaction wheel set-up, as well as four distinct thruster configurations. Key metrics included thrust output, angular velocity, and pointing accuracy, represented by the half-cone offset angle. In addition, power and energy requirements were investigated. Robustness tests evaluated system adaptability to single thruster failures, solar array deployment, and high initial angular velocities. Physical validation was achieved by porting the control algorithm to an embedded STM32 Nucleo board, which was connected to a solenoid valve representing a dummy thruster. This setup allowed for real-time testing of execution accuracy, signal fidelity, and system integration.

The results demonstrated that the control algorithm consistently maintained high pointing accuracy, keeping the half-cone offset angle within the mission requirement of 0.18 [$^\circ$]. Reaction wheels and electrical thrusters effectively generated the required control torques, with negligible angular momentum build-up in the nominal simulation. Overdetermined thruster configurations showcased enhanced energy efficiency, reducing total energy consumption by approximately 29% compared to determinate setups. However, the study also highlighted challenges, such as the minimum impulse bit constraints of the Pocket Rocket thrusters, which limited precise low-thrust operations.

Robustness tests revealed that determinate configurations failed to accommodate single thruster failures, while overdetermined systems adapted effectively, albeit with increased power demands. From the results, it is recommended for any space mission to be certain that the on-board computer is aware of any thruster fail occurring. The undeployed solar array configuration significantly reduced energy requirements, supporting its feasibility for early mission phases. The thruster configurations showed not to be able to realistically counteract de-tumbling manoeuvres, significantly degrading their feasibility in the actual LUMIO mission. Additionally, the embedded system validation confirmed accurate command signal generation and execution, bridging the gap between simulation and real-world implementation.

Although the electrical thruster configurations adhered to pointing accuracy requirements in nominal simulation conditions, they impose significantly higher power and energy demands and are unable to counteract high de-tumbling manoeuvrers compared to the current reaction wheel configuration in the LUMIO ADCS design. Addressing main engine parasitic torques and exploring alternative, higher-performance thrusters may offer a feasible solution for integrating electrical propulsion into lunar CubeSat ADCS. While the reaction wheel and non-electrical thruster combination remains the preferable option for now, this research demonstrates the potential for electrical thrusters in advancing CubeSat capabilities. Future studies should build upon this foundation by addressing current limitations, extending simulation durations, and exploring innovative propulsion technologies to unlock their full potential in deep-space missions.

1 Introduction and Problem Description

1.1. Background

Over the past two decades, miniaturisation has been a defining trend in the rapidly evolving field of satellite technology. CubeSats, small satellites with a standardised form factor of a 10 [cm] cube, were introduced in 1998 to provide university students with the opportunity to design, test, and operate spacecraft with capabilities similar to those of the Russian Sputnik, using primarily commercial-off-the-shelf (COTS) components. Since their introduction, CubeSats have gained widespread adoption across universities worldwide, becoming a standard platform for a variety of space applications. The 2010s saw a surge in CubeSat launches fuelled by the rise of commercial spacecraft, and national space agencies are now incorporating CubeSats into numerous missions. To date, over 2,000 CubeSats have been launched and other form factors, such as the PocketQube with standard form of 5 [cm] cube, have spun off from its development.[1] The relevance of CubeSats has grown far beyond their educational origins, particularly in the context of lunar exploration. Recent advancements in technology have positioned CubeSats as pivotal tools for a wide range of lunar applications, including observation, communication relays, and scientific investigation. National space agencies and private entities alike are leveraging CubeSats to explore lunar environments, search for resources, and test innovative space technologies in the demanding conditions of deep space.

Subsystem miniaturisation remains a critical topic in the development of CubeSats, especially for propulsion systems. These systems, essential for orbital manoeuvrers, rendezvous operations, and attitude control, typically employ different types of thruster modules. Non-electrical thrusters, which generate linear impulse by expelling mass, have been extensively tested and widely used in space missions, including CubeSat applications. More recently, advances in micro-propulsion have brought electrical thrusters into the spotlight. While these thrusters have been less frequently deployed in space missions, their development shows significant promise for future CubeSat applications.

Another crucial subsystem in spacecraft development is the Attitude Determination and Control System (ADCS). Its primary function is to control the spacecraft's orientation to meet the requirements of scientific observation, power generation, or communication. The CubeSat industry has seen significant advancements in miniaturised ADCS components, including reaction wheels and magnetorquers. While the use of non-electrical thrusters within ADCS has been extensively studied and applied, the potential of electrical thrusters in this domain remains under-explored. This research focuses on bridging this gap by investigating the application of electrical thrusters within CubeSat ADCS design.

1.2. Problem Definition

The primary goal of this thesis is the assessment of different electrical thruster configurations, applied to an existing space mission, for its attitude control performance, robustness and connectivity. The steps that are taken to carry out this analysis, are listed below.

[1]URL: `https://www.jpl.nasa.gov/topics/cubesats` [Accessed 11 March 2024]

- Identify the current trends in lunar CubeSat and electrical propulsion development.
- Construct the main dynamic and kinematic relations necessary for spacecraft attitude control.
- Convert all the necessary relations into a functional attitude control algorithm.
- Include different thruster set-ups within the control algorithm.
- Perform separate robustness tests and assess the system's behaviour.
- Port the simulation to embedded software and connect with existing hardware modules as a validation method.

The context of this research will be an existing lunar CubeSat mission, which will be the European Space Agency's LUMIO mission, set to launch in 2027 and analyse meteoroid impacts on the far-side of the Moon. Its entire mission geometry will be analysed and adapted for this research, based on the most recent information available.

1.3. Thesis Layout

The thesis research will be explained throughout this report in a structured manner. First of all, in Chapter 2, a literature study will be presented, including recent studies into spacecraft attitude dynamics, lunar CubeSats, embedded systems and electrical thrusters. Next, in Chapter 3, the research objective will be formulated, as well as the main research question. Based on this main question, several sub-questions and hypotheses will be proposed as well. Chapter 4 will elaborate on all the specific steps taken in this research, explaining the context, the methods used and the resources required. This chapter will solely focus on the simulation aspects within the research. Then, in Chapter 5, the experimental validation part of the research will be explained, highlighting the steps taken towards a practical set-up and indicating what results will be focused on. Chapter 6 shall present all the results from the simulation as well as from the experiments, and provide brief observations and preliminary conclusions from these results. After all these have been examined, a verification and validation section is included, emphasising the methods used throughout simulation development, practical set-up and result analysis. Based on the results, Chapter 7 will present an elaborate discussion and further clarification of the results, after which the research is concluded in Chapter 8. Recommendations for future work and assessment of the research questions and its hypotheses will also take place in these two chapters, respectively.

1.4. Novelty

The literature review in Chapter 2 will demonstrate that no CubeSat ADCS currently exists that extensively integrates electrical micro-thrusters for precision attitude control and manoeuvrability, whilst also comparing different configurations. While non-electrical thrusters have been widely implemented for attitude control in CubeSats, they pose challenges such as higher mass, propellant limitations, and reduced flexibility for prolonged missions. Reaction wheels and magnetorquers are well-established solutions for CubeSat attitude control, but their performance is limited in high-disturbance or reaction wheel desaturation scenarios. The integration of electrical micro-thrusters within CubeSat ADCS systems presents a promising but under-explored opportunity to address these limitations. This thesis aims to demonstrate novelty by:

- Investigating the integration of electrical micro-thrusters into CubeSat ADCS for enhanced precision and redundancy in attitude control.
- Developing a comprehensive control algorithm that combines traditional ADCS components with electrical thrusters to meet demanding mission requirements.
- Designing and implementing simulation configurations to assess the performance of different thruster setups under realistic mission conditions.
- Validating the feasibility of using electrical micro-thrusters for long-duration CubeSat missions by analysing power, energy, and mass requirements.
- Validating the feasibility of using electrical micro-thrusters for long-duration CubeSat missions by connecting existing hardware modules to the developed control algorithm.

2 Literature Review

2.1. Introduction

As documented in the Planetary Exploration Horizon 2061 (Lasue et al. [31] and Grande et al. [20]), an increasing interest in Moon missions has been observed among space agencies all over the globe. Notable is, of course, NASA's Artemis missions, that will attempt human exploration of the Moon for the first time since 1972. Other missions are driven by sample return, cartography, radiation, in-situ resource utilisation and many more scientific goals. On the long term, for missions executed after 2035, developments will take place concerning robotic and human infrastructures, so that a permanent Moon Village may become a reality. Moon-based Earth observatories, astrobiological research centres and astronomical observatories will be built so that the Moon may serve as the gateway for further deep space exploration. [31]

As presented in Grande et al. [20], the miniaturisation of spacecraft introduce disruptive technologies for the next four decades of space exploration. Future small satellites may be able to achieve the same performance as what today's satellites do that have approximately 5 to 10 times as much mass. It will become increasingly more standard to add CubeSat and NanoSat missions to the main mission of a launch, in order for risky close approximation measurements to be done, land on the surface of planetary bodies and explore the subsurfaces of planetary bodies. These small satellites require low-thrust, low-power propulsion systems, for which electrical system offer the ideal outcome; not only can they be mass-produced via standardised CubeSat technology, inducing extreme cost reduction, but they also offer quality and extensive lifetimes.

This chapter will pose as a basis of knowledge containing important developments within topics such as lunar exploration, spacecraft miniaturisation, CubeSat technology and many more. For any research, it is of importance to have the most up-to-date information as possible, so that overlaps and repetitions are omitted.

For this research, it is crucial to thoroughly understand what spacecraft attitude control is, what its main principles are and in which way it can be applied to existing or future space missions. Moreover, since the context of this research will include a lunar CubeSat mission, it is necessary to gain knowledge on past, current and future missions with a similar profile. Next, in order to understand how actual attitude determination and control system function, embedded systems need to be examined, outlining their basic principles and applications in existing spacecraft. Finally, a closer look should be given to vacuum arc thrusters, since this type of electrical propulsion system will be the main focus of this research.

This chapter will examine the topics explained above, so that no crucial information will be overlooked before the start of the research. The information in this chapter will be presented in a general form, so that it can easily be applied to the research at hand.

2.2. Research Questions

In order to effectively approach the literature review, a number of research questions specifically for the review were developed. These are all indicated by the abbreviation ”**LQ**” (Literature Question) along with a number to distinguish them for future reference. They form the basis for this chapter and clarify the structure of information provided. The literature research questions are presented below.

LQ-01 : What are the past, current and future developments in lunar CubeSat missions?

LQ-02 : What spacecraft dynamics relations need to be taken into account for the development of a CubeSat attitude control algorithm?

LQ-03 : For an electrical-thruster-based attitude control system applied to a CubeSat mission, in which way can thruster configurations be assessed on their feasibility in the required mission profile?

LQ-04 : What are the critical factors to consider when testing thruster hardware modules to ensure effective software-hardware integration for algorithm validation?

LQ-05 : What are the working principles of vacuum arc thrusters?

LQ-06 : What are the current vacuum arc thruster applications in CubeSat missions?

In Section 2.7, these research questions will be reviewed again in order to ascertain that they have been fulfilled.

2.3. Spacecraft Attitude Control

In space, objects experience forces directly without the mitigating effects of an atmosphere or other medium. In the vacuum of space, even small forces exerted on an object can result in significant changes to its orientation, position, and velocity over time. Examples of external forces that influence spacecraft include gravitational forces, solar radiation pressure, atmospheric drag (in low Earth orbit), interactions with the Earth’s magnetic field, and collisions with micrometeoroids or other particles.

For most spacecraft, maintaining their orientation, or attitude, within specific bounds is essential. For some missions, such as Earth observation, precise attitude control is critical to ensure that the satellite observes the correct regions of the Earth and transmits meaningful data to ground stations. For other missions, the requirement may simply be to prevent the spacecraft from spinning uncontrollably. The subsystem responsible for maintaining the spacecraft’s attitude within these boundaries is called the Attitude Determination and Control System (ADCS). This section will discuss the fundamentals of ADCS, including its dynamics, actuators, controllers, and methods of integration. Since this research revolves around CubeSat applications, special notice will be given to the main differences with regularly-sized spacecraft, in terms of miniaturisation and other considerations.

2.3.1. Basics

A standard attitude determination and control system (ADCS) for a spacecraft consists of hardware and software. Hardware components can be subdivided into sensors, actuators, and an on-board computer. Software should be installed on the on-board computer (OBC) so that signals are received from the sensors, and adequate commands are forwarded to the actuators. [54] A summary of regularly used sensors is given below:

- **Sun sensors**: Provide data on the spacecraft’s orientation relative to the Sun, crucial for basic attitude determination and especially useful in LEO.
- **Magnetometers**: Measure the nearby planet’s magnetic field strength and orientation, assisting in determining the spacecraft’s attitude relative to the planet’s magnetic field.
- **Star Trackers**: Use images of star patterns to accurately determine the spacecraft’s orientation in space. While they offer high accuracy, their size, power consumption, and cost might not always be suitable for very small spacecraft.
- **Inertial Measurement Units (IMUs)**: Include accelerometers and gyroscopes to measure linear acceleration and angular velocity, respectively, providing data on the spacecraft’s motion and rotation.

- **Horizon Sensors**: Detect the edge of a planet against the backdrop of space, providing data on the spacecraft's orientation relative to the planet. These are particularly useful for Earth-orbiting spacecraft.
- **GPS Sensors**: Use signals from navigation satellites to determine the spacecraft's position and velocity. This information can also indirectly support attitude determination when combined with other data.

The actuators that can be used for attitude control within spacecraft are:

- **Reaction wheels**: Wheels that spin at controlled speeds to adjust the spacecraft's orientation through the conservation of angular momentum.
- **Momentum wheels**: Similar to reaction wheels but primarily used to maintain a steady orientation rather than making frequent adjustments by storing angular momentum. They are simpler in design and suited for long-duration missions requiring stable attitude control.
- **Magnetic torquers**: Utilise the interaction between an onboard electromagnetic coil and a planet's magnetic field to exert control forces that can adjust the spacecraft's attitude.
- **Thrusters**: Propulsion units that can provide precise control, using a non-electrical, electrical system or hybrid system. Next to independent control, thrusters also provide a solution to momentum wheel saturation.
- **Solar sails**: Relatively small sails can be used for minor corrections in attitude. This does not require any fuel and only power to unfold.
- **Control moment gyroscopes**: These are gimbal-mounted rotors that can be used to control two spin axes. They are complex components that are especially suited for large spacecraft.

An on-board computer will make sure all sub-components of the ADCS are connected and commands are properly sent through. It uses a control strategy, or control law, in order to attain the desired attitude for the spacecraft. Modern-day OBCs are designed to process and transmit digital signals with high reliability, ensuring that commands are accurately executed by actuators and data from sensors are processed efficiently. They operate with robust fault tolerance to handle the harsh environment of space and typically include radiation-hardened components to prevent failures caused by high-energy particles. Within spacecraft, the ADCS can be controlled by a central OBC, which is also responsible for managing the other subsystems such as the thermal control system or the payload. Another option is implementing an OBC that is specifically dedicated to the ADCS, in addition to the central OBC. Examples of these two configurations include the following:

- **Central OBC for all subsystems**: The CubeSat mission ”Dove” by Planet Labs uses a central OBC for all subsystems, including ADCS. This approach simplifies the system architecture and reduces hardware costs but may impose limitations on processing capacity.[1]
- **Dedicated OBC for ADCS**: The OPS-SAT mission by the European Space Agency, a 3U CubeSat designed to test and validate new tehcniques in satellite control, featured an experimental platform with a powerful on-board computer, separate from the main bus systems, to manage ADCS functions among other experimental tasks.[2]

In the context of CubeSats, OBC development has focused on miniaturisation, power efficiency, and modularity. A notable example is the ISISpace On-Board Computer developed by Innovative Solutions In Space (ISISpace).[3] This OBC is compact, lightweight, and specifically tailored for small satellite missions. It includes features like a low-power processor, ample storage, and multiple communication interfaces, making it ideal for the tight power and volume constraints of CubeSats.

2.3.2. Dynamics

This section outlines the fundamental dynamic equations that govern the attitude behaviour of a rigid object. The equations will be inspected, after which definitions regarding reference frames are given. Next, the attitude state expressed in Euler angles and quaternions will be elaborated upon. Finally, relevant disturbance torques will be explained.

[1]URL: `https://www.eoportal.org/satellite-missions/dove` [Accessed 06 January 2025]
[2]URL: `https://www.esa.int/Enabling_Support/Operations/OPS-SAT` [Accessed 06 January 2025]
[3]URL: `https://www.isispace.nl/product/on-board-computer/` [Accessed 25 November 2024]

2.3.2.1 Rigid Body Dynamics

In order to understand the problem of spacecraft attitude dynamics, the reader must understand the underlying equations. From Wertz [54], Euler's dynamic equation of motion for a rigid-body dynamics model is formulated as presented in Equation 2.1, in vector form. Throughout this report, bold-faced symbols will indicate vectors.

$$\boldsymbol{T} = \dot{\boldsymbol{h}} + \boldsymbol{\omega} \times \boldsymbol{h} \tag{2.1}$$

$$\boldsymbol{T} = \boldsymbol{T}_c + \boldsymbol{T}_d \tag{2.2}$$

$$\boldsymbol{h} = I\boldsymbol{\omega} \tag{2.3}$$

First of all, it should be noted that this Euler equation is valid within a rotating reference frame (e.g. spacecraft-centred frame, axes fixed to the body) with respect to an inertial reference frame, such as the Earth-centred inertial (ECI) frame. Secondly, from the equation, the vector $\boldsymbol{T}$ is presented, which consists of two individual torque vectors: the control torque vector ($\boldsymbol{T_c}$) and the disturbance torque vector ($\boldsymbol{T_d}$, Equation 2.2). The control torque vector is the imposed torque or moment by the attitude control system (e.g. momentum wheels, thrusters, magnetorquers). The disturbance torque is caused by external factors and used as an input to the system. Next, $\dot{\boldsymbol{h}}$ represents the time derivative of the angular momentum, in units of $[kg \cdot m^2 \cdot rad \cdot s^{-2}]$ or $[Nm]$. It can be determined by Equation 2.3, and taking the time derivative of the angular velocity $\dot{\boldsymbol{\omega}}$ instead. The angular velocity vector $\boldsymbol{\omega}$ is expressed in $[rad \cdot s^{-1}]$ and represents the angular velocity with which the rotating coordinate frame is rotating relative to an inertial frame. Finally, as presented in Equation 2.3, the angular momentum vector $\boldsymbol{h}$ is defined, in which I represents the spacecraft rotational inertia matrix around the centre of mass (CoM) with units $[kg \cdot m^2]$.

In order to further understand the Euler dynamic equation of motion, it should be noted that for a spacecraft, the change in angular momentum of the body is equal to the torques applied to the body: $\boldsymbol{T} = \dot{\boldsymbol{h}}$. The cross product of the angular velocity and the angular momentum in Equation 2.1 is a term that represents the gyroscopic torque, which is the internal torque that arises when a rotating body (which already has angular momentum) is subjected to a change in the direction of its angular velocity vector. This is a result of the conservation of angular momentum and it does not change the magnitude of the angular momentum, only its direction. Substituting Equation 2.3 in Equation 2.1, the equation can be written as presented in Equation 2.4. This form of the equation can easily be used for satellite attitude integration over time, for which different techniques will be elaborated upon in subsection 2.3.5.

$$\dot{\boldsymbol{\omega}} = I^{-1}\left(\boldsymbol{T}_c + \boldsymbol{T}_d - \boldsymbol{\omega} \times I\boldsymbol{\omega}\right) \tag{2.4}$$

2.3.2.2 Reference Frames

As mentioned previously, the equations above are defined in a spacecraft-centred frame, of which the axes are fixed to the body of the spacecraft. For this reason, this reference frame shall be called the body frame, with B as its indicator. The origin of this frame is at the centre of mass of the rigid body, and the three axes should complete a right-hand system as displayed in Figure 2.1. Here, the axis are denoted by $\boldsymbol{b_1}$ to $\boldsymbol{b_3}$, which are unit vectors in the directions indicated. In attitude control problems, the orientation of these unit vectors with respect to an additional, inertial frame, is the issue to be solved. A local vertical local horizontal (LVLH) reference frame A, with its axes $\boldsymbol{a_1}$ to $\boldsymbol{a_3}$ can now be defined relative to the body frame. This is depicted in Figure 2.2. The axes are not fixed to the body of the satellite, but rather fixed to the orbit around the Earth in this case. $\boldsymbol{a_1}$ is pointed in the orbit direction, $\boldsymbol{a_2}$ is pointed perpendicular to the orbital plane and $\boldsymbol{a_3}$ is directed towards the centre of the Earth. In addition to the LVLH frame, a Newtonian inertial frame is presented, depicted by $\boldsymbol{n_1}$ to $\boldsymbol{n_3}$, which can be attached to the centres of any orbital element or desired location. While the inertial frame provides a stable reference for long-term planning, the LVLH frame is particularly useful for describing satellite motion relative to its orbit. For instance, the LVLH frame simplifies tasks such as Earth observation, where instruments must point toward specific surface regions. The body frame, on the other hand, is critical for on-board computations, such as actuator commands or gyroscopic measurements.

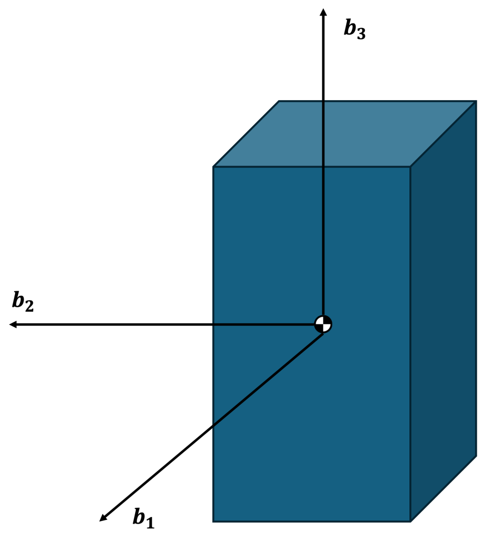


Figure 2.1: Body-centred reference frame depicted on an arbitrary rigid body, with the origin in the body's centre of mass and three orthogonal axes.

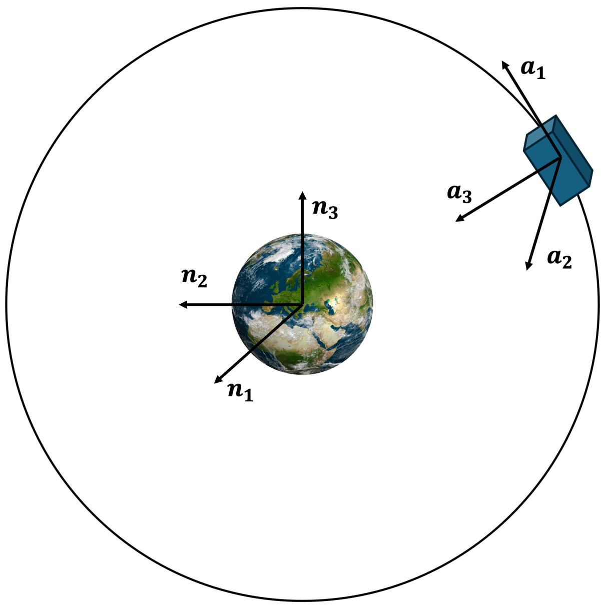


Figure 2.2: Newtonian inertial reference frame N and local horizontal local vertical (LVLH) reference frames, depicted with orthogonal axes and origins in the centre of mass of the celestial body and the spacecraft for both frames, respectively.

For all these references frames, the unit vectors that represent the axes are orthogonal and can be expressed in terms of each other. This is shown in Equation 2.5 to Equation 2.7. Each of the terms C_{ij} in these equations are called the direction cosines and represent the cosine of the angle between the two axes $\boldsymbol{b_i}$ and $\boldsymbol{a_j}$, which can be seen in Equation 2.8. When writing Equation 2.5 to Equation 2.7 in matrix form, the matrix $C^{B/A}$ represents the direction cosine matrix or coordinate transformation matrix from A to B. These matrices can be used to transform the components of a vector from any coordinate frame to another, and are therefore practical to use situations where different coordinate systems are of interest. The matrix representation is shown in Equation 2.9. [25]

$$\boldsymbol{b}_1 = C_{11}\boldsymbol{a}_1 + C_{12}\boldsymbol{a}_2 + C_{13}\boldsymbol{a}_3 \tag{2.5}$$

$$\boldsymbol{b}_2 = C_{21}\boldsymbol{a}_1 + C_{22}\boldsymbol{a}_2 + C_{23}\boldsymbol{a}_3 \tag{2.6}$$

$$\boldsymbol{b}_3 = C_{31}\boldsymbol{a}_1 + C_{32}\boldsymbol{a}_2 + C_{33}\boldsymbol{a}_3 \tag{2.7}$$

$$C_{ij} = \boldsymbol{b_i} \cdot \boldsymbol{a_j} \tag{2.8}$$

$$\begin{bmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \\ \boldsymbol{b}_3 \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_1 \\ \boldsymbol{a}_2 \\ \boldsymbol{a}_3 \end{bmatrix} = C^{B/A} \begin{bmatrix} \boldsymbol{a}_1 \\ \boldsymbol{a}_2 \\ \boldsymbol{a}_3 \end{bmatrix} \tag{2.9}$$

With this in mind, the definition of the Euler angles can be established. These are the angles between a reference frame attached to a rigid body, with respect to a pre-defined inertial reference frame. The Euler angles can be used to map an inertial reference frame (the Newtonian inertial frame in Figure 2.2) to a body-fixed frame (the body frame in Figure 2.1). Three distinct rotations over the axes of the inertial reference frame need to be carried out to achieve this, and this sequence is called the 3-2-1 Euler rotation. [4] In order to properly understand how this relates to the Euler angles, it will be explained step by step. Equation 2.10 shows the direction cosine matrix for rotation over axis $\boldsymbol{n_1}$, by angle θ_1. Therefore, applying this direction cosine matrix C_1 in a similar fashion as $C^{B/A}$ in Equation 2.9 to the inertial frame $\boldsymbol{n}$, results in a coordinate frame $\boldsymbol{c}$. This is shown in Equation 2.11. The new coordinate frame $\boldsymbol{c}$ has its $\boldsymbol{c_2}$ and $\boldsymbol{c_3}$ axes oriented in a different direction compared to the previous $\boldsymbol{n_2}$ and $\boldsymbol{n_3}$ axes. Note that the $\boldsymbol{c_1}$ and $\boldsymbol{n_1}$ axes are in fact equal.

$$C_1(\theta_1) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_1 & \sin\theta_1 \\ 0 & -\sin\theta_1 & \cos\theta_1 \end{bmatrix} \tag{2.10}$$

$$\begin{bmatrix} \boldsymbol{c}_1 \\ \boldsymbol{c}_2 \\ \boldsymbol{c}_3 \end{bmatrix} = C_1(\theta_1) \begin{bmatrix} \boldsymbol{n}_1 \\ \boldsymbol{n}_2 \\ \boldsymbol{n}_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_1 & \sin\theta_1 \\ 0 & -\sin\theta_1 & \cos\theta_1 \end{bmatrix} \begin{bmatrix} \boldsymbol{n}_1 \\ \boldsymbol{n}_2 \\ \boldsymbol{n}_3 \end{bmatrix} \tag{2.11}$$

Additional direction cosine matrices exist for the remaining two axes, which are displayed in Equation 2.12 for the $\boldsymbol{n_2}$ axis and displayed in Equation 2.13 for the $\boldsymbol{n_3}$ axis. Different rotations of axes can also be combined, in order to obtain a desired, changed reference frame for pre-determined angles. In the 3-2-1 Euler rotation sequence, this combination of rotations is applied to transform from any given inertial frame to the desired body frame. This combination of rotations is displayed in Equation 2.14, where it can be seen that the first rotation of the initial frame is placed closest to the frame vector. The reason why this sequence exactly represents such a rotation, is because the rotation angles (Euler angles) are defined as the angles between inertial and body frames.

$$C_2(\theta_2) = \begin{bmatrix} \cos\theta_2 & 0 & -\sin\theta_2 \\ 0 & 1 & 0 \\ \sin\theta_2 & 0 & \cos\theta_2 \end{bmatrix} \tag{2.12}$$

$$C_3(\theta_3) = \begin{bmatrix} \cos\theta_3 & \sin\theta_3 & 0 \\ -\sin\theta_3 & \cos\theta_3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.13}$$

$$\begin{bmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \\ \boldsymbol{b}_3 \end{bmatrix} = C_1(\theta_1)C_2(\theta_2)C_3(\theta_3) \begin{bmatrix} \boldsymbol{n}_1 \\ \boldsymbol{n}_2 \\ \boldsymbol{n}_3 \end{bmatrix} \tag{2.14}$$

The sequence is non-commutative, which means it does not yield the same result when the direction cosine matrices are applied in a reverse order. This sequence is often also referred to as the yaw-pitch-roll rotation, in which:

- The first rotation (yaw) rotates the inertial frame around the z-axis, reorienting the x- and y-axes in the horizontal plane.
- The second rotation (pitch) tilts the frame around the new y'-axis, bringing the z-axis into alignment with the body frame's intended orientation.
- The third rotation (roll) spins the body about the new x''-axis, fully aligning the body frame with the desired orientation.

The yaw angle is often displayed by the Greek "psi" (ψ), the pitch angle by the Greek "theta" (θ) and the roll angle by the Greek "phi" (ϕ). Euler angles are commonly used in spacecraft simulations to describe and predict orientation changes. For instance, the yaw angle (ψ) might indicate how a satellite orients its solar arrays toward the Sun, while pitch (θ) and roll (ϕ) may describe manoeuvrers required for Earth observation or docking with another spacecraft.

2.3.2.3 Quaternions

While Euler angles provide an intuitive representation of orientation, they are prone to singularities, such as gimbal lock, where two axes align, leading to a loss of a degree of freedom. To overcome this, quaternions are often used in spacecraft attitude control due to their ability to represent rotations without singularities and with lower computational cost for real-time applications. Quaternions are a mathematical notation that extends complex numbers, consisting of one real part and three imaginary parts, often written as $q = q_w + q_1\boldsymbol{i} + q_2\boldsymbol{j} + q_3\boldsymbol{k}$, where q_w is the real or scalar part. q_1, q_2 and q_3 represent the imaginary or vector part. They provide a robust way to represent spatial rotations without singularities associated with Euler angles. In satellite attitude determination, quaternions are used to accurately and efficiently describe the satellite's orientation in three-dimensional space. They are particularly useful because they can represent large rotations with small, continuous changes in their values, and the quaternion multiplication operation directly corresponds to the composition of rotations.

This section will provide an extensive overview of quaternion mathematical operations, so that no mistakes can be made with respect to their definitions throughout this report. The operations are key for the development of an attitude control algorithm, since applying quaternions to this end will provide the most robust solution. The mathematical expressions will be used as seen in Fresk and Nikolakopoulos [17]. First of all, as seen previously, quaternions can be noted as Equation 2.15 or Equation 2.16. The distinction between the scalar and vector parts should be evident.

$$\boldsymbol{q} = q_w + q_1\boldsymbol{i} + q_2\boldsymbol{j} + q_3\boldsymbol{k} \tag{2.15}$$

$$\boldsymbol{q} = \begin{bmatrix} q_w & q_1 & q_2 & q_3 \end{bmatrix}^T \tag{2.16}$$

Since quaternions also represent rotations between reference frames, these rotations can also be combined as was seen in the Euler 3-2-1 sequence. In order to achieve this combination of rotations, the Kronecker product can be applied ($\otimes$) between quaternions $\boldsymbol{p}$ and $\boldsymbol{q}$. Similar to combined Euler rotations, quaternion rotations are also non-commutative. The Kronecker product can be seen in Equation 2.17.

$$\boldsymbol{p} \otimes \boldsymbol{q} = Q(\boldsymbol{p})\boldsymbol{q} = \begin{bmatrix} p_w & -p_1 & -p_2 & -p_3 \\ p_1 & p_w & -p_3 & p_2 \\ p_2 & p_3 & p_w & -p_1 \\ p_3 & -p_2 & p_1 & p_w \end{bmatrix} \begin{bmatrix} q_w \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} p_wq_w - p_1q_1 - p_2q_2 - p_3q_3 \\ p_wq_1 + p_1q_w + p_2q_3 - p_3q_2 \\ p_wq_2 - p_1q_3 + p_2q_w + p_3q_1 \\ p_wq_3 + p_1q_2 - p_2q_1 + p_3q_w \end{bmatrix} \tag{2.17}$$

The norm of a quaternion is always equal to 1 exactly. This is displayed in Equation 2.18. In addition, the complex conjugate of a quaternion is equal to the conjugate of regular complex numbers, as seen in Equation 2.19.

$$\text{Norm}(\boldsymbol{q}) = \|\boldsymbol{q}\| = \sqrt{q_w^2 + q_1^2 + q_2^2 + q_3^2} = 1 \tag{2.18}$$

$$\text{Conj}(\boldsymbol{q}) = \boldsymbol{q}^* = \begin{bmatrix} q_w & -q_1 & -q_2 & -q_3 \end{bmatrix}^T \tag{2.19}$$

The time derivative of a quaternion vector can be computed by using the angular velocity vector $\boldsymbol{\omega}$ or $\begin{bmatrix} \omega_1 & \omega_2 & \omega_3 \end{bmatrix}^T$. This time derivative can be used for numerical propagation / integration of the spacecraft attitude and is therefore a crucial parameter in attitude determination simulations. Equation 2.20 displays the equation, where $Q(\boldsymbol{q})$ is equal to the notation used in Equation 2.17.

$$\dot{\boldsymbol{q}}_\omega(\boldsymbol{q}, \boldsymbol{\omega}) = \frac{1}{2}\boldsymbol{q} \otimes \begin{bmatrix} 0 \\ \boldsymbol{\omega} \end{bmatrix} = \frac{1}{2}Q(\boldsymbol{q}) \begin{bmatrix} 0 \\ \boldsymbol{\omega} \end{bmatrix} \tag{2.20}$$

Next, the Euler 3-2-1 rotation can be expressed in quaternion form, to convert an inertial Newtonian frame $\boldsymbol{n}$ to a body-fixed frame $\boldsymbol{b}$. This is represented in Equation 2.21. The full derivation can be examined in Fresk and Nikolakopoulos [17].

$$\begin{bmatrix} \boldsymbol{b}_1 \\ \boldsymbol{b}_2 \\ \boldsymbol{b}_3 \end{bmatrix} = C_1(\boldsymbol{q})C_2(\boldsymbol{q})C_3(\boldsymbol{q}) \begin{bmatrix} \boldsymbol{n}_1 \\ \boldsymbol{n}_2 \\ \boldsymbol{n}_3 \end{bmatrix} = \begin{bmatrix} 1 - 2(q_2^2 + q_3^2) & 2(q_1q_2 + q_3q_w) & 2(q_1q_3 - q_2q_w) \\ 2(q_1q_2 - q_3q_w) & 1 - 2(q_1^2 + q_3^2) & 2(q_2q_3 + q_1q_w) \\ 2(q_1q_3 + q_2q_w) & 2(q_2q_3 - q_1q_w) & 1 - 2(q_1^2 + q_2^2) \end{bmatrix} \begin{bmatrix} \boldsymbol{n}_1 \\ \boldsymbol{n}_2 \\ \boldsymbol{n}_3 \end{bmatrix} \tag{2.21}$$

A given direction cosine matrix can directly be converted to the desired quaternion attitude. Each instance of the direction cosine matrix is denoted by C_{11}, C_{12}, C_{13}, etc. Note that the conversion is based on the value of the trace of the matrix, defined as $C_{11} + C_{22} + C_{33}$. The conversion formulae are displayed in Equation 2.22, Equation 2.23, Equation 2.24 and Equation 2.25.

If trace $= C_{11} + C_{22} + C_{33} > 0$:

$$q_w = \frac{1}{2}\sqrt{1 + C_{11} + C_{22} + C_{33}} \tag{2.22}$$
$$q_1 = \frac{C_{32} - C_{23}}{4q_w}, \quad q_2 = \frac{C_{13} - C_{31}}{4q_w}, \quad q_3 = \frac{C_{21} - C_{12}}{4q_w}$$

If $C_{11} > C_{22}$ and $C_{11} > C_{33}$:

$$q_1 = \frac{1}{2}\sqrt{1 + C_{11} - C_{22} - C_{33}} \tag{2.23}$$
$$q_w = \frac{C_{32} - C_{23}}{4q_1}, \quad q_2 = \frac{C_{12} + C_{21}}{4q_1}, \quad q_3 = \frac{C_{13} + C_{31}}{4q_1}$$

If $C_{22} > C_{33}$:

$$q_2 = \frac{1}{2}\sqrt{1 + C_{22} - C_{11} - C_{33}} \tag{2.24}$$
$$q_w = \frac{C_{13} - C_{31}}{4q_2}, \quad q_1 = \frac{C_{12} + C_{21}}{4q_2}, \quad q_3 = \frac{C_{23} + C_{32}}{4q_2}$$

Otherwise (if C_{33} is largest):

$$q_3 = \frac{1}{2}\sqrt{1 + C_{33} - C_{11} - C_{22}} \tag{2.25}$$
$$q_w = \frac{C_{21} - C_{12}}{4q_3}, \quad q_1 = \frac{C_{13} + C_{31}}{4q_3}, \quad q_2 = \frac{C_{23} + C_{32}}{4q_3}$$

Euler angles can easily be converted to the associated quaternion values using Equation 2.26, Equation 2.27, Equation 2.28 and Equation 2.29. In this equation, strict adherence need to be followed regarding the definition of θ_1, θ_2 and θ_3, which should remain equal to the angles described previously.

$$q_w = \cos\left(\frac{\theta_1}{2}\right)\cos\left(\frac{\theta_2}{2}\right)\cos\left(\frac{\theta_3}{2}\right) + \sin\left(\frac{\theta_1}{2}\right)\sin\left(\frac{\theta_2}{2}\right)\sin\left(\frac{\theta_3}{2}\right) \tag{2.26}$$

$$q_1 = \sin\left(\frac{\theta_1}{2}\right)\cos\left(\frac{\theta_2}{2}\right)\cos\left(\frac{\theta_3}{2}\right) - \cos\left(\frac{\theta_1}{2}\right)\sin\left(\frac{\theta_2}{2}\right)\sin\left(\frac{\theta_3}{2}\right) \tag{2.27}$$

$$q_2 = \cos\left(\frac{\theta_1}{2}\right)\sin\left(\frac{\theta_2}{2}\right)\cos\left(\frac{\theta_3}{2}\right) + \sin\left(\frac{\theta_1}{2}\right)\cos\left(\frac{\theta_2}{2}\right)\sin\left(\frac{\theta_3}{2}\right) \tag{2.28}$$

$$q_3 = \cos\left(\frac{\theta_1}{2}\right)\cos\left(\frac{\theta_2}{2}\right)\sin\left(\frac{\theta_3}{2}\right) - \sin\left(\frac{\theta_1}{2}\right)\sin\left(\frac{\theta_2}{2}\right)\cos\left(\frac{\theta_3}{2}\right) \tag{2.29}$$

The conversion can also be carried out in a reverse fashion, displayed in Equation 2.30 in matrix form.

$$\begin{bmatrix}\phi \\ \theta \\ \psi\end{bmatrix} = \begin{bmatrix}\text{atan2}\left(2(q_w q_1 + q_2 q_3), q_w^2 - q_1^2 - q_2^2 + q_3^2\right) \\ \arcsin\left(2(q_w q_2 - q_3 q_1)\right) \\ \text{atan2}\left(2(q_w q_3 + q_1 q_2), q_w^2 + q_1^2 - q_2^2 - q_3^2\right)\end{bmatrix} \tag{2.30}$$

Finally, Equation 2.31 shows the calculation of the error quaternion $\boldsymbol{q}_e$, which is the offset between a reference quaternion $\boldsymbol{q}_{ref}$ and the actual attitude quaternion $\boldsymbol{q}$.

$$\boldsymbol{q}_e = \boldsymbol{q}_{ref} \otimes \boldsymbol{q}^* \tag{2.31}$$

2.3.2.4 Disturbances

For any spacecraft, the distribution of its mass in combination with the presence of a gravitational field will induce disturbance torques around its centre of mass. These torques are called gravity gradient torques and should be incorporated in the $\boldsymbol{T_d}$ term along with additional disturbance torques. From Gottlieb [19], an equation is obtained for the gravity gradient torque at any point in time around any celestial body. This equation is presented in Equation 2.32. $\boldsymbol{r_{SC}}$ represents the position vector from the centre of mass of the satellite to the centre of mass of the central celestial body. μ is the gravitational parameter in $[m^3 s^{-2}]$ of the celestial body, and $\boldsymbol{I}$ is the inertia matrix of the satellite. Note that this equation is valid in the body reference frame and any vectors expressed in an inertial frame (for example, the position of a satellite with respect to the centre of the Earth) should be converted by use of the Euler or quaternion 3-2-1 sequence.

$$\boldsymbol{T}_{GG} = 3 \cdot \frac{\mu}{\|\boldsymbol{r}_{\text{SC}}\|^3} \cdot (\hat{\boldsymbol{r}}_{\text{SC}} \times (\boldsymbol{I} \cdot \hat{\boldsymbol{r}}_{\text{SC}})) \tag{2.32}$$

$$\hat{\boldsymbol{r}}_{\text{SC}} = \frac{\boldsymbol{r}_{\text{SC}}}{\|\boldsymbol{r}_{\text{SC}}\|} \tag{2.33}$$

In addition to the gravity gradient torque, the solar radiation pressure is a disturbance input that should be considered in spacecraft control algorithm development. The solar radiation pressure is caused by photons, particles that constitute the propagation of light, that physically interact with surfaces they are travelling in to. These interactions or impacts cause a relatively small force to be exerted, which can cause notable effect in space. A relation is obtained from Wertz [54], which is also applied in Romero-Calvo, Biggs, and Topputo [43], that is presented in Equation 2.34 to Equation 2.36. Equation 2.34 shows the inverse power law for computing the solar radiation intensity at a given distance r from the Sun. P_{solar} is the solar constant, equal to approximately $3.842 \cdot 10^{26}$ $[W]$. The solar intensity in an Earth orbit is approximately 1366.1 $[W/m^2]$.

For a spacecraft with a total number of k surface plates, Equation 2.35 shows the force exerted by the solar radiation pressure on a flat surface, for the i_{th} plate. I_s is the solar intensity, c is the speed of light, A_i is the surface area of the i_{th} plate, $\boldsymbol{S}$ is a $3 \times k$ matrix with unit vectors pointing from the Sun to the surface in its columns. $\boldsymbol{n_s}$ is a $3 \times k$ matrix with unit vectors normal to the surface of each plate and directed towards the interior of the spacecraft. Finally, ρ_s and ρ_d are the specularly and diffusely reflected radiation. Their values are determined from experiments. All vectors are expressed in the body frame, similar to the gravity gradient torque definition. With the calculated force vectors, a disturbance torque can be calculated with Equation 2.36, which is a simplified model neglecting interactions between the surfaces such as shadows and other reflections. In here, $\boldsymbol{c}_{pi}$ denotes the position vectors between the centre of mass of the spacecraft and the centre of pressure of the surfaces, which are assumed to be equal to their geometrical centres for this research. The dot product between $\boldsymbol{S}_i$ and $\boldsymbol{n}_{si}$ checks whether this specific panel is in eclipse, and it will set the solar radiation pressure torque to zero if it is.

$$I_s = \frac{P_{\text{solar}}}{4\pi r^2} \tag{2.34}$$

$$\boldsymbol{F}_i = \frac{I_s}{c} A_i \left(\boldsymbol{S}_i \cdot \boldsymbol{n}_{si}\right) \left\{ (1 - \rho_s)\, \boldsymbol{S}_i + \left[2\rho_s \left(\boldsymbol{S} \cdot \boldsymbol{n}_{si}\right) + \frac{2}{3}\rho_d \right] \boldsymbol{n}_{si} \right\} \tag{2.35}$$

$$\boldsymbol{T}_{SRP} = \begin{cases} \sum_{i=1}^{n} \boldsymbol{c}_{pi} \times \boldsymbol{F}_i & \text{if } \boldsymbol{S}_i \cdot \boldsymbol{n}_{si} > 0, \\ \boldsymbol{0} & \text{otherwise.} \end{cases} \tag{2.36}$$

In addition to the aforementioned disturbance torques, the magnetic disturbance torque is also a significant perturbation to be dealt with in attitude control systems. It is caused by the interaction of a spacecraft's residual magnetic moments with the magnetic field of a celestial body. Residual magnetic moments are caused by onboard electronics, materials or magnetised components. A simple expression for the calculation of the magnetic disturbance torque is presented in Equation 2.37, in which $\boldsymbol{M}_{res}$ denotes the spacecraft's residual magnetic moment and $\boldsymbol{B}$ denotes the external magnetic field at the spacecraft's location. [54]

$$\boldsymbol{T}_{mag} = \boldsymbol{M}_{res} \times \boldsymbol{B} \tag{2.37}$$

Finally, disturbance torques that could influence the attitude control strategy of a spacecraft are listed below.

- **Aerodynamic drag**: For spacecraft in LEO, the thin atmosphere that is still present at their altitudes can cause a significant disturbance torque and decelerating force due to drag.

- **Thruster misalignment**: When thrusters are not aligned in their desired directions, disturbance torques are created upon firing.
- **Third-body gravity**: Each celestial body has a gravitational influence on a spacecraft, regardless of its position. In most simplified simulations, only the close celestial bodies are considered.
- **Thermal radiation**: The uneven thermal radiation emitted by spacecraft can cause an additional, relatively small disturbance torque.
- **Micro-meteoroid impact**: High-velocity impact of micro-meteoroids can cause disturbance torques (and significant damage) at the spacecraft's surface.

2.3.3. Actuators

The on-board computer or specific ADCS computer of a spacecraft sends signals to actuators in order for the spacecraft attitude to correct itself adequately. Actuators can be sub-divided in different types, as was explained in subsection 2.3.1, and two distinct types that are relevant to this research will be examined: reaction wheels and thrusters. In the context of this research, magnetic torquers will not be possible for usage, since a magnetic field is required and the Moon is not in possession of one. Additionally, control moment gyroscopes have similar functionality to reaction wheels, so no additional chapter will be dedicated to this. Solar sails will not be considered for this research either.

2.3.3.1 Reaction Wheels

Reaction wheels are a type of flywheel commonly used in spacecraft for attitude control and maintaining stability. These electrically powered devices can be commanded to spin when external disturbances affect the spacecraft's angular motion. The fundamental principle behind reaction wheel operation is the conservation of angular momentum. When a spacecraft is undisturbed in space and a reaction wheel is spun in a clockwise direction (with its rotational axis aligned with the spacecraft's x-axis), the spacecraft will rotate in the opposite direction (counter-clockwise) around its x-axis to conserve angular momentum. This counteracting motion ensures precise attitude adjustments and stability without the need for propellant.

There is an important difference between a reaction wheel and a momentum wheel, and this paragraph serves as clarification. As described in Wertz [54], a momentum wheel differs from a reaction wheel in its operational design. A momentum wheel typically operates with a constant spin rate (known as a bias), providing a gyroscopic stability effect to the spacecraft. In contrast, a reaction wheel operates with a variable spin rate, allowing it to produce torque for attitude adjustments, and is generally not designed to maintain a constant bias.

When an external disturbance force alters the angular motion of a spacecraft, the reverse principle can be applied. The increase in angular momentum the spacecraft experiences can be stored in the reaction wheel by letting it rotate in the opposite direction. In this way, the spacecraft change in attitude is compensated. Reaction wheels have a high pointing accuracy but can only make the spacecraft rotate about its centre of mass. Also, translational motion is not possible by use of reaction wheels. Full three-axis attitude control is only possible when at least three reaction wheels are placed on the spacecraft. Stability around each of the axes is created due to the gyroscopic effect, which counter-acts any inclination of the axis that is rotating. Advantages of the usage of reaction wheels in spacecraft are that they do not require fuel and therefore an extra fuel tank, they are relatively power efficient and they store angular momentum for stability. Disadvantages are the fact that they are mechanisms with vibrational limits and individual components. Moreover, they can become saturated when the maximum achievable angular momentum has been stored in them. Desaturation strategies using different sets of actuators are possible and should be considered in the design of the ADCS.

In order to gain a better understanding of reaction wheel performances, an example from industry will be shown. The CubeWheel, developed by the South African company CubeSpace, and described in Leibbrandt and Miller [32], is a balanced reaction or momentum wheel (bias excluded or included) and can, at the time of writing, be acquired in three different sizes. It is specifically designed for application in CubeSats up to 12U, and can be applied in a three-wheel set-up or a four-wheel pyramid set-up, with a redundant fourth wheel. From small to large, the CubeWheel reference IDs are CW0017, CW0057 and CW0162. In Figure 2.3, the CubeWheel (seond generation) is shown in four different sizes. Figure 2.4

shows the pyramid set-up of four CubeWheels to be directly applied in spacecraft as ADCS subsystem.

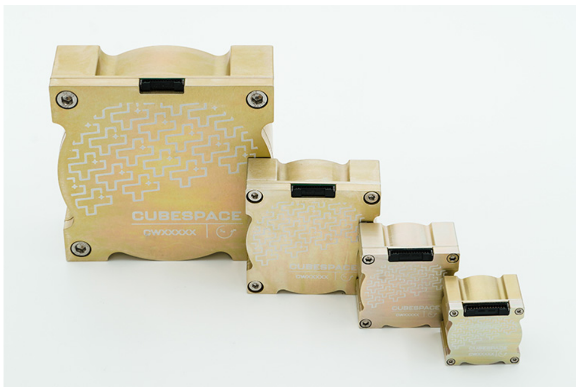


Figure 2.3: CubeWheel hardware in four different sizes. The three wheels on the right side are the CW0162, CW0057 and the CW0017, seen from right to left. [32]

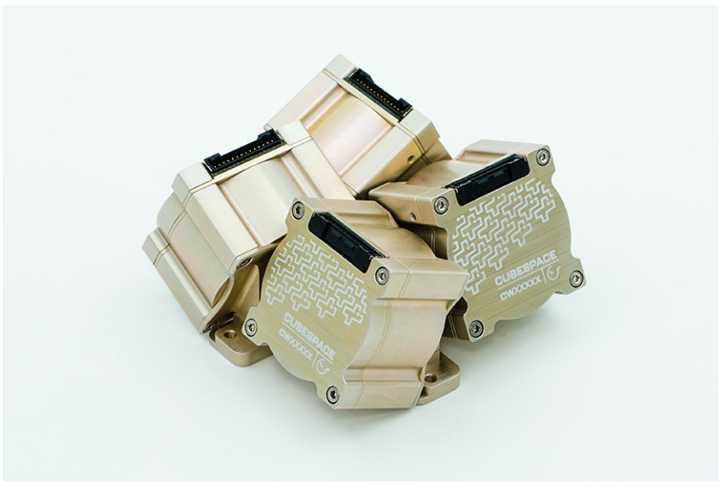


Figure 2.4: CubeWheel pyramid configuration, consisting of a four-wheel set-up that can directly be acquired to act as part of the ADCS subsystem. [32]

Table 2.1 provides an overview of the characteristics of the aforementioned CubeWheel sizes. A number of characteristics stand out. First of all, it can be seen that for each of the reaction wheels, the supply voltage for the maximum speed is lower than that of the nominal motor supply. The reason for this lies in the back electromotive force, which is an induced voltage by the spinning motor that opposes and therefore reduces the input voltage necessary for operation. The momentum at 6000 revolutions per minute defines the angular momentum that is stored in the wheels at this rate. Moreover, the saturation torque indicates the maximum torque possible of the wheels, within its thermal, mechanical and electronic limits. Dynamic imbalance occurs when the mass or inertia axis does not coincide with the rotational axis. Since a mechanical flywheel is never perfect in shape, an imbalance will always be present. Imbalance has effects for the vibrational analysis of the structure. Finally, the mass, dimensions and power characteristics are there to obtain an idea of the value magnitudes for CubeSat applications.

Performance	**CW0017**	**CW0057**	**CW0162**
Nominal Motor Supply Voltage [V]	8	12	12
Supply Voltage for Max Speed [V]	6.4	11	11
Max Speed [RPM]	10000	10000	10000
Momentum @ 6000 RPM [mNms]	1.77	5.7	16.2
Saturation Torque [mNm]	0.23	2	7
Dynamic Imbalance [g.cm^2]	$<$0.005	$<$0.014	$<$0.014
Physical			
Mass [g]	60	115	144
Dimensions [WxHxL] [mm]	28x26x28	35x24x35	46x24x46
Power			
Average Power @ 2000 RPM [mW]	180	336	480
Peak Power [Max Torque] [W]	0.85	2.7	7.2

Table 2.1: Specifications of the CW017, CW057, and CW162 CubeWheels.

The addition of reaction wheels to the ADCS of a spacecraft has consequences for the dynamic equations that govern its attitude. Euler's dynamic equation of rotational motion should be adjusted so that an additional angular momentum term due to the reaction wheel is also considered. This is described in Ismail and Varatharajoo [23] and can be written similar to Equation 2.1 and Equation 2.4, which is displayed in Equation 2.38 and Equation 2.39 below. It should be noted that the total angular momentum is now equal to the angular momentum of the spacecraft ($\boldsymbol{h}$) and the reaction wheels ($\boldsymbol{h}_{rw}$) combined. The control torque $\boldsymbol{T}_c$ is the torque necessary on the spacecraft body to counter-act the disturbance torques; adhering to Newton's third law of action-reaction systems, this means that the exerted torque by the reaction wheels is equal in magnitude but opposite in direction to the control torque applied to the spacecraft body. Ergo: $\boldsymbol{T}_{rw} = -\boldsymbol{T}_c$.

$$\boldsymbol{T} = \dot{\boldsymbol{h}} + \boldsymbol{\omega} \times (\boldsymbol{h} + \boldsymbol{h}_{rw}) \tag{2.38}$$

$$\dot{\boldsymbol{\omega}} = I^{-1} \left(\boldsymbol{T}_c + \boldsymbol{T}_d - \boldsymbol{\omega} \times (I\boldsymbol{\omega} + I_{rw}\boldsymbol{\omega}_{rw})\right) \tag{2.39}$$

Now, $\boldsymbol{T}_{rw}$ itself does not say much about the torque exerted by only one reaction wheel; this will completely depend on the number of wheels present, and their respective configurations. A link must therefore be made between the required torque and the individual torque that was actually exerted by each wheel. The necessary parameter for this link is called the configuration matrix, and is denoted by A_{rw} as seen in Equation 2.40. The instances $a_{i,rw}$ are unit vectors in the direction of the central spin axis of the i_{th} reaction wheel, relative to the body axes of the satellite. As an example, Equation 2.41 shows the configuration matrix for the usage of three reaction wheels, with their spin axis exactly aligned with the primary or body axes of the satellite.

$$A_{rw} = \begin{bmatrix} a_{1,rw} & a_{2,rw} & a_{3,rw} & \dots & a_{k,rw} \end{bmatrix} \tag{2.40}$$

$$A_{rw} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.41}$$

If another reaction wheel is added, for redundancy reasons, and it is tilted from the other three axis with an equal angle, the configuration matrix becomes as shown in Equation 2.42.

$$A_{rw} = \begin{bmatrix} 1 & 0 & 0 & \frac{1}{\sqrt{3}} \\ 0 & 1 & 0 & \frac{1}{\sqrt{3}} \\ 0 & 0 & 1 & \frac{1}{\sqrt{3}} \end{bmatrix} \tag{2.42}$$

The configuration matrix is linking the required overall spacecraft torque to the required torque per reaction wheel as shown in Equation 2.43, taking the four-wheel configuration from Equation 2.42 as example.

$$\begin{bmatrix} T_{rw,x} \\ T_{rw,y} \\ T_{rw,z} \end{bmatrix} = A_{rw} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \frac{1}{\sqrt{3}} \\ 0 & 1 & 0 & \frac{1}{\sqrt{3}} \\ 0 & 0 & 1 & \frac{1}{\sqrt{3}} \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{bmatrix} \tag{2.43}$$

In most cases, the required combined control torque by the wheels T_{rw} is known, and the control torque per wheel is requested. Since A_{rw} is not a square matrix, it cannot be inverted and the right pseudo-inverse transformation as shown in Equation 2.44 should be taken instead. The calculation of individual reaction wheel torques is then performed as shown in Equation 2.45. [48]

$$A_{rw}^{+} = A_{rw}^{T} \left(A_{rw} A_{rw}^{T}\right)^{-1} \tag{2.44}$$

$$\begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{bmatrix} = A_{rw}^{+} \begin{bmatrix} T_{rw,x} \\ T_{rw,y} \\ T_{rw,z} \end{bmatrix} \tag{2.45}$$

In order to conclude the theory on reaction wheels and their application, the reader should understand that from an integrated control algorithm, based on Euler's dynamic equations of rotational motion, a required control torque is computed for the proper control of the spacecraft's attitude. From this required control torque, using the configuration matrix applicable to this specific spacecraft, the required torques from each of the reaction wheels is determined. With these values, reaction wheels can be chosen that fit within the torque limits, along with their mass and volume constraints. Consequently, a power budget can be constructed that provides an overview of the total power needed for the entire spacecraft. In this way, reaction wheel sizing becomes an integral part of the overall spacecraft design.

2.3.3.2 Thrusters

The second type of actuators that is relevant for a lunar spacecraft are thrusters. As mentioned previously, when using reaction wheels, at least one other type of actuators is necessary for reaction wheel de-saturation. In general, thrusters can be sub-divided in electrical, non-electrical and hybrid variations. Non-electrical thrusters are pure reaction systems that expel an amount of a substance, called the propellant, in the opposite direction of the flight path. Newton's third law, which states that any action should yield an opposite but equal in magnitude reaction, lies on the basis of this. Performance parameters that should always be considered in the design of thrusters are the thrust exerted, the specific impulse and the obtained change in velocity, Δv. [10]

The thrust produced by a rocket or a non-electrical thruster is calculated using Equation 2.46. The thrust is expressed in Newtons and F_T indicates the thrust as a scalar. $\dot{m}$ is the mass flow rate of the propellant, expressed in $[kg/s]$. v_e is the exhaust velocity of the propellant in $[m/s]$, which is the velocity that the propellant leaves the nozzle of the thruster with, relative to the rest of the spacecraft. p_e is the pressure of the propellant at the exit of the nozzle, and p_a is the pressure of the surroundings. In space, where there is vacuum, p_a will be equal to 0. Pressures are expressed in $[Pa]$ or, equivalently, $[N/m^2]$. Finally, A_e is the area over which the propellant leaves the nozzle, or the thruster exit area in $[m^2]$. As can be seen from the equation, the total thrust consists of two separate parts, the momentum term and the pressure term. The entire equation can be written in a more concise manner, combining the momentum and pressure effects in one equivalent jet velocity term, v_{eq}.

$$F_T = \dot{m} \cdot v_e + (p_e - p_a) \cdot A_e = \dot{m} \cdot v_{eq} \tag{2.46}$$

The specific impulse of a thruster, defined as the ratio of the total impulse generated by a thruster over the propellant weight (not mass) for this impulse, is generally computed using Equation 2.47. In case the effective exhaust velocity over time is constant, and substituting the right-hand side of Equation 2.46 in the equation, results in Equation 2.48. In here, g_0 is the gravitational acceleration of the Earth at sea level, which is equal to approximately 9.80665 $[m/s^2]$. The specific impulse gives an indication of the efficiency with which propellant is turned into velocity or into force. A higher specific impulse implies that less mass is needed for the same thrust level.

$$F_T = \dot{m} \cdot I_{sp} \cdot g_0 \tag{2.47}$$

$$I_{sp} = \frac{v_{eq}}{g_0} \tag{2.48}$$

Finally, using the Tsiolkovsky equation as shown in Equation 2.49, the increase in velocity (Δv) due to the burning of a specific amount of propellant mass can be calculated. This equation is also called the rocket equation and depends on the initial mass of the entire spacecraft m_0 and the mass that was burned during firing, m_P. This equation is only valid under a number of assumptions: there are no additional forces influencing the spacecraft, the thrust direction is opposite the direction of flight and v_e remains equal in magnitude over time.

$$\Delta v = v_{eq} \cdot \ln \left(\frac{m_0}{m_0 - m_P} \right) \tag{2.49}$$

In order to obtain an overview of the available options within non-electrical thrusters, the list below shows the most common types. Note that the mono-propellant, bi-propellant and solid propellant systems belong to the category of chemical thrusters.

- **Cold gas thruster**: In this type of thruster, the propellant is stored in a dedicated tank maintained at a specific pressure. A valve controls the release of the propellant, generating thrust when opened. No additional heating or pressurisation is required for operation. As the propellant is consumed, the pressure in the tank gradually decreases, leading to a corresponding reduction in thrust over time.
- **Mono-propellant thruster**: Propellant is stored in a propellant tank, which is pressurised with the use of an additional pressurant. Upon opening the thrust valve, propellant will flow to the decomposition chamber of the nozzle, where decomposition takes place and extra heat is generated. In most cases, pre-heating occurs as well.
- **Bi-propellant thruster**: Bi-propellant systems are similar to mono-propellant system, although two propellant types are now present: a fuel and an oxidiser. Both propellants chemically react with each other in the combustion chamber, after having been released from their individual storage tanks by activation of the valves. High thrust levels are often observed compared to mono-propellant or cold gas systems.
- **Solid propellant thruster**: The working principles for this type is identical to that of bi-propellant system, although the propellants are now stored in a solid phase (compared to the liquid or gaseous state of the bi-propellant system). One major disadvantage of these systems is that, after they have been ignited, they cannot be stopped until the propellant is depleted. Thrust levels can be altered by changing the geometric shape of the grains.

Electrical propulsion is based on the acceleration of particles or propellants by means of electrostatic or electromagnetic forces. Multiple different types of electrical thrusters exist and they all have different working principles, which is why general equations for parameters such as the specific impulse and thrust are not possible, but should rather be examined based on the type of thruster selected for a mission. An overview of current and future electrical propulsion systems for satellites has been given in Esho et al. [15]. Common types of electrical thrusters are:

- **Ion thruster**: Due to ionisation of the propellant, charged particles are created. Ionisation is generally performed by shooting electrons at the propellant. These particles are then accelerated by an electrostatic field, that is created by a voltage drop between two grids. This acceleration of ions creates thrust.
- **Radio-frequency thruster**: The working principle of a RF thruster is the same to that of an ion thruster, with the major difference the ionisation technique. In RF thrusters, electric coils create an oscillating electric field, which in its turn creates ions from the propellant.
- **Hall effect thruster**: The working principle of Hall effect thrusters is also similar to that of ion thrusters. The difference lies in the way of ion acceleration; mutually perpendicular magnetic and electric fields in the ionisation chamber induce the required acceleration.
- **Electrospray thruster**: The acceleration of ions in this thruster happens from a liquid propellant. Ionic liquids are used as propellant, and a strong electric field releases the ions from the liquid and accelerates them.
- **Magnetoplasmadynamic thrusters**: The working principle in this type of thruster is the acceleration of gaseous ions. Acceleration happens due to the Lorentz force exerted on them, induced by a current that is passing through the gas or plasma, and a magnetic field. The magnetic field can be created externally or by the current itself. Magnetoplasmadynamic thrusters distinguish themselves from other electric thruster systems by their superior specific impulse values.
- **Pulsed plasma thruster**: PPTs work by using a high-current, pulsed arc discharge to ablate a solid propellant, creating ionised gas. The ions are accelerated by a Lorentz force generated by the interaction of the current and a self-induced magnetic field, producing thrust. The main

advantage of using such thrusters is the fact that they rely on pulses; this creates precision in manoeuvrers, and lets the user create a varying thrust level by varying its pulse frequency. A specific type of PPT is the Vacuum Arc Thruster (VAT), for which the ablation takes place by use of a vacuum arc, instead of a spark plug in regular PPTs. Also, the cathode material itself ablates for VATs, instead of the additional material added in regular PPTs. More specific information on VATs will be provided in Section 2.6.

From the above list, ion, RF and electrospray thrusters accelerate charged particles using an electrostatic field. For these types, simplified equations can be generated for their thrust and specific impulse, as also presented in Cervone [10]. These equations will give an idea on what parameters are important for these performance values. First of all, in Equation 2.50, the thrust of an ion thruster is presented, with ϵ_0 the permittivity of vacuum (which is equal to $8.8542 \cdot 10^{-12}$ $[F/m]$), V the voltage difference between the two grids in $[V]$ (Volts), n the number of holes in the charged grids, L the distance between these holes and D the diameter of these holes.

$$F_T = \frac{2\pi\varepsilon_0}{9} \cdot V^2 \cdot n \left(\frac{D}{L}\right)^2 \tag{2.50}$$

The specific impulse of the electrical thruster can be calculated similar to the specific impulse of non-electrical thrusters. This is shown in Equation 2.51, in which η_{ion} represents the thruster efficiency. The jet velocity of the ion thruster is dependent on the voltage between the two grids and the molecular mass of the charged ions. The larger the molecular mass, the lower the jet velocity, and the higher the potential between the grids, the higher the jet velocity.

$$I_{sp} = \eta_{\text{ion}} \frac{v_{eq}}{g_0} \tag{2.51}$$

A final type of thruster that is worth mentioning is the resistojet. It is a hybrid thruster, using electrical as well as non-electrical thruster aspects. Propellant is electrically heated in the combustion chamber and is then accelerated through a convergent-divergent nozzle. Liquid propellants are most widely used for this application. Research is currently being performed for the usage of liquid water as propellant, and the University of Tokyo has successfully executed an orbit transfer using their AQUARIUS propulsion system on board the EQUULEUS mission, as described in Sekine et al. [45]. The specific application of this resistojet was a 6U CubeSat, and it achieved a total thrust by two thruster heads of 6.0 $[mN]$, with a specific impulse of 91.0 $[s]$.

When comparing the various thruster options discussed in this section for application within the ADCS of a spacecraft, it becomes evident that each type has distinct advantages and limitations, influencing their suitability as stand-alone systems or in combination with reaction wheels. Non-electrical thrusters, such as cold gas, mono-propellant, bi-propellant, and solid propellant systems, generally offer simplicity in design and rapid thrust response, which can be advantageous for attitude correction. Among these, cold gas thrusters stand out for their straightforward operation and precise control, making them viable for small attitude adjustments. However, their thrust levels are often insufficient for significant manoeuvres, especially as the propellant pressure decreases over time. Mono-propellant and bi-propellant systems deliver higher thrust levels due to chemical reactions, but their complexity, need for additional pressurisation systems, and potential thermal challenges may complicate integration within a CubeSat's limited space and thermal budget. Solid propellant thrusters provide high thrust but lack controllability once ignited, which make them significantly inconvenient for ADCS applications. Throttle ability is also limited in these systems, which means the output thrust levels cannot be controlled as smoothly as might be desired.

Electrical thrusters, such as ion, Hall effect, radio-frequency, and electrospray thrusters, provide precise and efficient thrust, with high specific impulses ideal for long-duration missions. However, their relatively low thrust levels and dependency on electrical power make them less suitable for standalone ADCS use, particularly in missions requiring rapid or frequent attitude adjustments. Magnetoplasmadynamic and pulsed plasma thrusters can deliver higher thrust compared to other electric options, but their power requirements can become limiting. Pulsed plasma thrusters may offer a unique advantage in ADCS applications by enabling precise, short bursts of thrust, which could complement reaction wheels or serve as a standalone system in some cases. Hybrid options like resistojets could bridge the gap by combining non-electrical and electrical propulsion principles but their reliance on liquid propellants and additional heating systems may pose integration challenges.

As has been introduced at the beginning of this report, miniaturisation is a common theme within spacecraft development nowadays. This miniaturisation automatically leads to advancements in the development of micro-propulsion systems. The thruster types mentioned above are all also examples of micro-thrusters and can be applied to micro-satellites, ranging from PocketQube sizes (5$\times$5$\times$5 $[cm]$) to 12U CubeSat sizes, dependent on specific mission requirements. In general, non-electrical propulsion units have a higher thrust level but lower specific impulse level compared to electrical thrusters. Electrical thrusters only require a specific amount of power from the electrical power system, whereas non-electrical thrusters require an infrastructure including tank volume. In CubeSat design, the latter may not be beneficial. Also, toxic chemicals used in non-electrical thrusters should be replaced by green propulsion types. Nowadays, these green propulsion types have not yet reached the Technology Readiness Level (TRL) that is required for novel space missions to launch, which is why more sustainable thrusters for attitude control of CubeSat applications are desired.

In order to provide a general overview of micro-propulsion system performance, Table 2.2 shows an overview of different existing modules. From left to right, the columns of the table represent the module name, the type of thrusters used, the average thrust level of each thruster, the specific impulse of the thrusters and the system size. Note that the S-iEPS from MIT is an propulsion system consisting of eight different thrusters. The other modules shown are single-thruster systems. Thrust-over-power ratios for the electrical thrusters differ significantly per type as well, being in the 10 to 30 $[mN/kW]$ range for pulsed plasma thrusters, to the 50 to 70 $[mN/kW]$ range for Hall Effect thrusters.

Name	Thruster type	Thrust level [mN]	Vacuum specific impulse [s]	Size
Busek Co. BGT-X5	Chemical, mono-propellant	500	220	1U
NASA JPL, MiXI	Ion	3.0	3050	3 $[cm]$ diameter
Busek Co. BHT-200	Hall effect	13	1390	Not specified
MIT, S-iEPS	Electrospray	8 $\times$ 0.074	950	0.2U
Mars Space Ltd. PPTCUP	Pulsed plasma	0.040	600	0.3U

Table 2.2: Existing micro-propulsion systems with their thruster type, thrust level, specific impulse and sizes. This table serves as an overview of the system performance parameters. [7] [56] [8] [28] [33]

Past research into CubeSat attitude control using electrical thrusters was done by Kronhaus et al. [29], in which vacuum arc thrusters were used for the attitude control of 1U CubeSats. Their proposed micro-propulsion module, consisting of four separate thrusters each with a thrust level of 1 $[\mu N]$ and integrable with 1U LEO CubeSats, was capable of achieving a pointing accuracy of 0.5°. Additional research was conducted by King et al. [27], in which attitude control using VATs was analysed theoretically for a 3U LEO CubeSat. Results included pointing accuracy in the order of 0.007°, as well as significant ADCS mass reduction from 300 to 1500 $[g]$ to a maximum of 250 $[g]$ (under 72% of the original wheel/torquer system). Further improvements were gained with respect to volume and power requirement. Finally, the research in Lian, Xiang, and Zhao [34] shows a novel attitude tracking control specifically designed for the tracking and six-degrees-of-freedom control of CubeSats and larger spacecraft. Control accuracy was traded off with number of thrusters fired (in order to reduce power requirement) by tuning the width and frequency of the pulse signals, a method that will be examined in subsection 2.6.2 of this report as well.

Without using gimballing thrusters, any spacecraft needs a minimum number of thrusters in order to maintain three-axis attitude control. In order to obtain a perfectly determinate system, a total of three thruster pairs are required as a minimum, in order to provide for three-axis control without any resultant net thrust values. Thrusters in such pairs should be pointed opposite in direction of each other, and create a torque about the primary axes of the spacecraft. Including fewer thrusters will result in an under-determined system, whereas including more thrusters introduces redundancy, within an over-determined system. [54]

Regarding the actual torque exerted by a specific thruster configuration, the placement and the angular offset of the thrusters will contribute to the torque that is produced, as well as the thrust level of the thruster. Equation 2.52 below shows the total torque produced by a system of an arbitrary number of thrusters. $\boldsymbol{r_i}$ is the position vector of thruster i in an arbitrary reference frame. $\boldsymbol{r}_M$ denotes the position vector of the centre of mass, in the same reference frame. $\boldsymbol{F_i}$ represents the force vector of each thruster i, taking into account the thruster angular offset in the reference frame as indicates before. The thrust is a function of time t and control signal $f_i(t)$, meaning that it will be exerted due to a specific control signal from the controller, which in itself is based on the current time step. For this reason, the total control torque $\boldsymbol{T}_c$ is a function of time in this equation.

$$\boldsymbol{T}_c(t) = \sum_i (\boldsymbol{r}_i - \boldsymbol{r}_M) \times \boldsymbol{F}_i(f_i(t), t) \tag{2.52}$$

Within attitude control algorithms, a required spacecraft torque stems from the Euler equations for rotational motion in order to gain attitude stability, as was seen in, among others, Equation 2.2. It was previously found that this required torque could be converted to individual reaction wheel torques for any reaction wheel configuration given. The same can be done for different thruster configurations, as was described in Dennehy et al. [14]. This method is called the non-negative least-squares method. The configuration of the thrusters on a specific spacecraft can be transformed into a thruster mixing matrix. A thruster mixing matrix contains in its columns the unit torque vectors of each thruster in a specific configuration. For example, the thruster mixing matrix of three thrusters that solely produce a positive torque around the x-axis, would look like Equation 2.53.

$$A_{thrust} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{2.53}$$

In order to construct a thruster mixing matrix for an arbitrary number of thrusters, with different positions and angular offsets, an adaptation of Equation 2.52 can be used, which is shown in Equation 2.54. In this equation, $\hat{\boldsymbol{T}}_i$ is the unit torque produced by thruster i, $\boldsymbol{r}_i$ is the position vector of the thruster with respect to the centre of mass of the spacecraft (note that the $\boldsymbol{r}_M$ vector is in this way omitted) and $\hat{\boldsymbol{F}}_i$ is the unit vector of the thrust exerted by the thruster. This equation is formulated in the spacecraft body frame (origin is the centre of mass). The unit torque vectors of thruster 1 through n are appended to the thruster mixing matrix as displayed in Equation 2.55.

$$\hat{\boldsymbol{T}}_i = \boldsymbol{r}_i \times \hat{\boldsymbol{F}}_i \tag{2.54}$$

$$A_{thrust} = \begin{bmatrix} \hat{\boldsymbol{T}}_1 & \hat{\boldsymbol{T}}_2 & ... & \hat{\boldsymbol{T}}_n \end{bmatrix} \tag{2.55}$$

In order to determine the force or thrust necessary from each of the thrusters to fulfil the control torque requirement, the linear equation shown in Equation 2.56 has to be solved. In this equation, F_{thrust} is an array of length n containing the thrust values of each thruster. A solution is guaranteed if the thruster mixing matrix is full row rank, meaning that the matrix has linearly independent rows.

$$\begin{aligned} A_{thrust} F_{thrust} &= \boldsymbol{T}_c, \\ F_{thrust} &\geq 0 \end{aligned} \tag{2.56}$$

In addition, the total thrust required from the thrusters should be minimised, so that a power- or fuel-optimal approach is adhered to. Therefore, the sum of the instances in F_{thrust} should be minimised. The above leads to the formalised linear programming model as displayed in Equation 2.57. c_i is the efficiency factor for thruster i, also note the symbol replacements for this generic form of the equation. A number of algorithms exist to efficiently solve this problem.

$$\begin{aligned} \text{min} \sum_{i=1}^{n} c_i X_i & \\ \text{s.t.} \qquad AX = B & \\ X \geq 0 & \end{aligned} \tag{2.57}$$

2.3.4. Controllers

Attitude information obtained from the Euler angles or quaternions as explained previously can be used to construct a spacecraft attitude control system. The basics of control system theory can be examined in literature such as Franklin, Powell, and Emami-Naeini [16] and has been adapted to this specific spacecraft application. Spacecraft attitude control systems can be visualised using block diagrams. Each of the blocks in these diagrams represents an operation that is performed, for which an input is taken and an output is produced. Figure 2.5 represents a general form of such a block diagram, with a closed-loop architecture.

In the figure, it can be seen that a commanded spacecraft state $\boldsymbol{x_{cmd}}$ is fed as input. Before continuing, it should be clear what a spacecraft state entails for orbit keeping or attitude control. For spacecraft orbit propagation, its position and velocity are of primary importance. For this reason, the spacecraft state consists of three Cartesian position coordinates, as well as the change of these coordinates over time, ergo: the velocities in those directions. Such a state can be observed in Equation 2.58.

$$\boldsymbol{x}_{orbit} = \begin{bmatrix} x & y & z & v_x & v_y & v_z \end{bmatrix}^T \tag{2.58}$$

For attitude control, not the position and velocity but rather the attitude and angular rate are of importance for the propagation of the state. In this way, the angle of the spacecraft body and the change of this angle with respect to an inertial reference frame can be observed. The form of the specific state depends on whether Euler angle analysis or quaternion analysis is carried out. They are displayed in Equation 2.59 and Equation 2.60, respectively.

$$\boldsymbol{x}_{Euler} = \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 & \omega_1 & \omega_2 & \omega_3 \end{bmatrix}^T \tag{2.59}$$

$$\boldsymbol{x}_{quat} = \begin{bmatrix} q_w & q_1 & q_2 & q_3 & \omega_1 & \omega_2 & \omega_3 \end{bmatrix}^T \tag{2.60}$$

For attitude control, the input commanded state is then compared to a measured spacecraft state $\boldsymbol{x_{meas}}$, which is the result of sensors or an inertial measurement unit. Note that the plus and minus indicate addition and subtraction, respectively. This results in a state error, $\boldsymbol{x_{err}}$ that is fed to the system controller. This controller can take many different forms, such as a PID (proportional, integral and differential) controller or a bang-bang (on/off) controller. This controller will use the state error to produce a signal f_i that is directly fed to the actuators, generating a change in the spacecraft state $\boldsymbol{\delta x_{act}}$. This change is then added to the change due to external disturbances, such as the solar radiation pressure or the gravity gradient torque, $\boldsymbol{\delta x_{dist}}$. The actuators and the external disturbance will actually impose forces and moments on the spacecraft, which is now denoted by a change in state. The spacecraft dynamics block represents the spacecraft itself, which will behave according to the laws of physics under the influence of forces and moments. As a result, a spacecraft state $\boldsymbol{x_{sc}}$ will be the final output of the system, which is then again measured by the sensors and the IMU, so that the error can be calculated again. This loop repeats itself until the error is within pre-defined boundaries and remains within these boundaries.

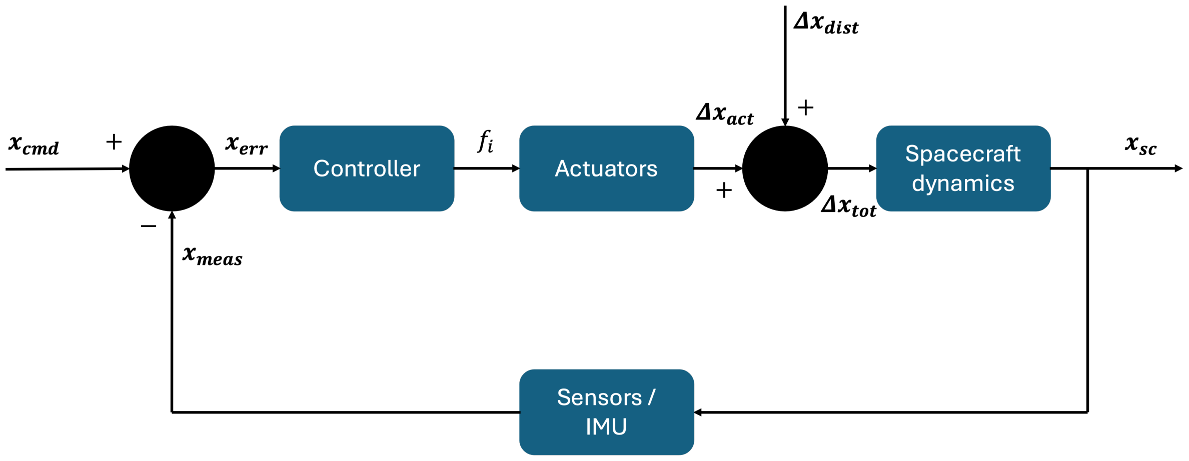


Figure 2.5: General spacecraft ACDS block diagram.

A closer look to the different controller options will now be given. Bang-bang control or on/off control is a simple controller that abruptly switches between two states (on and off) when a specific error value is passed. The two states are extremes: a vacuum cleaner, for example, is either completely on or completely off. In most bang-bang controller applications, time-optimal control is desired, achieving the desired state within a minimum time frame. Problems that arise with these types of controllers is usually an undesired state just before the switching happens. Also, when the hardware does not only provide two extreme values (for example, an intensity-varying light bulb), this type of control is not optimal in terms of energy efficiency, overshoot and oscillatory patterns. Bang-bang control is applied in spacecraft attitude control systems to activate thruster firing, in which non-electrical thrusters can only be switched on or off.

2.3.4.1 PID

As mentioned previously, the Proportional-Integral-Derivative (PID) control is a control loop structure that uses the error of the measured state to apply corrections. These corrections are calculated using proportional, integral, and derivative relations with respect to the error, using gains to balance the control output. Each component of the PID controller—proportional (P), integral (I), and derivative (D)—plays a specific role in the control mechanism:

- **Proportional (P)**: The proportional term generates an output that is proportional to the current error value. The proportional gain (K_p) modulates the magnitude of the proportional response. A high proportional gain results in a large change in the output for a given change in the error. If the proportional gain is too high, the system can become unstable and oscillate, while a low gain results in a sluggish response.
- **Integral (I)**: The integral term focuses on the accumulation of past errors, providing a necessary adjustment to the system that helps eliminate residual steady-state errors. The integral gain (K_i) determines how strongly the accumulated sum of past errors influences the controller output. An excessively high integral gain can lead to overshooting and oscillation, whereas too low a gain might cause a slow corrective response to a bias or sustained error.
- **Derivative (D)**: The derivative term predicts future error trends based on its current rate of change, acting as a form of damping mechanism. This helps reduce the system's overshoot and settling time. The derivative gain (K_d) controls the extent to which this prediction influences the control action. Too high a gain can make the control output too sensitive to rapid changes in error, potentially leading to instability due to noise amplification; too low a gain may not adequately dampen the response.

A block diagram of a general PID controller is presented in Figure 2.6. It can be seen that the state error x_{err} is inserted into each of the three blocks, and that operations take place within these blocks that are then summed to generate an output control variable $u(t)$. The variables are formulated in the time domain and can be transformed into the Laplace domain to analyse the system's behavior, particularly its stability and frequency response. Equation 2.61 shows the summed control variable output equation, including the previously mentioned gains and the proportional, integral and derivative terms, respectively. Equation 2.62 shows this same equation in the Laplace domain, which is the transfer function of the PID controller.

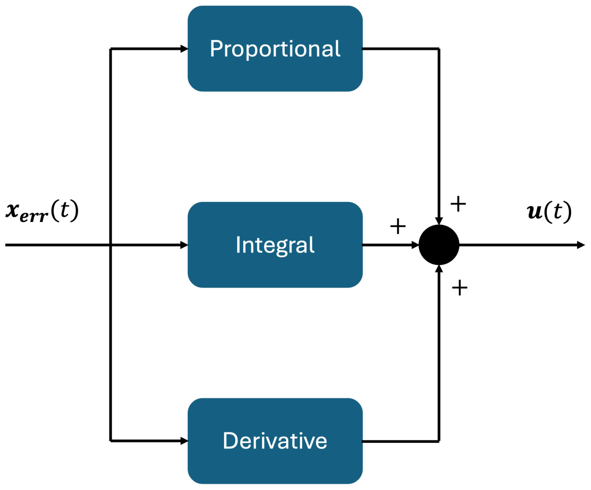


Figure 2.6: Block diagram of a PID controller block. $\boldsymbol{x_{err}}$ is the state error, which is inserted in the proportional, integral and derivative control blocks, after which they are summed to produce an output control variable $\boldsymbol{u}(t)$.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de(t)}{dt} \tag{2.61}$$

$$L(s) = K_p + \frac{K_i}{s} + K_d s \tag{2.62}$$

For the control algorithm to work properly, the gains need to be adjusted to the specific situation. For each situation, gains vary and a multitude of gain tuning techniques can be used to reach the best system performance. Three of such methods are:

- **Manual Tuning**: Involves manually adjusting the gains based on trial and error while observing the system's response.
- **Ziegler-Nichols Method**: A popular heuristic tuning method that provides a systematic procedure for setting the gains.
- **Software and Computational Tools**: Utilise algorithms to determine optimal gain values that conform to specific performance criteria.

Typically, tuning starts by setting the integral and derivative gains to zero and increasing the proportional gain until the output of the loop oscillates, then the integral gain is increased until any offset is corrected in the right amount of time, and finally, the derivative gain is adjusted to minimise overshoot. Effective tuning of PID gains is crucial as it directly affects the stability, responsiveness, and performance of the control system. Properly tuned PID parameters help ensure that the controlled system behaves in a desirable manner, responding quickly to changes without significant overshoot or oscillations.

In the Ziegler-Nichols method [57], the integral and derivative gains are also set to zero at first, and the proportional gain is increased and tested. When a specific ultimate gain K_u has been reached, in which sustained oscillations (stable and consistent) are present, an oscillation period T_u can then be determined. With these two parameters, tuning of the other gains is then carried out.

As a final note on PID controllers, Bello et al. [6] show experimental verification and comparison of two distinct PID controller, with an additional control method called a fuzzy controller, for application on nano-satellites. In this paper, two PID controllers are presented: one regular version with the three distinct gains, taking as input the error angle over the z-axis of the spacecraft and the angular velocity over the z-axis of the spacecraft. It is mentioned that the error in the angular velocity is not taken, due to low sampling frequency of this component. A required z-torque is then computed as output ($\boldsymbol{u}(t)$) which is passed on to the actuators. The second PID controller, the adaptive PID, includes a logic block that sets the integral function to zero, every time the error value over the z-axis changes sign. The reason this was implemented were two conditions regarding the size of the integral gain and the response overshoot. The integral gain would otherwise be able to induce undesired windup effects.

For both PID controllers, the gain calibration was performed in an experimental way. The yaw angle of the spacecraft was increased with a rotation of 179°, after which the convergence times for different gains were observed. These times should be minimised for the experiment. The process was divided in a coarse and fine calibration part. The coarse part included using the previously explained Ziegler-Nichols method to adjust the gains. The fine calibration part included three steps: first of all, the gains were adjusted with $\pm 1\%$ from their value obtained from the coarse calibration. Then, the calibration maneuver (179° yaw) was performed again to analyse its response. This process is finally repeated for a large number of times and the gains that minimised the response times were chosen as the final gains. More research into PID controllers for CubeSats has been published in recent years, such as Alkatheeri et al. [3] and Kannan, Anitha, and Kumarasamy [26].

2.3.4.2 Phase plane analysis

In Lim [35], a spacecraft attitude controller for the Tactical Satellite 4 was proposed based on a phase plane method. This method includes visualising certain characteristics of specific types of differential equations. In general, the phase plane will be a coordinate plane with two axes, each representing one state variable. As an example, the angular velocity in the x-direction (ω_x) can be presented on the x-axis of the phase plane, and the time derivative of the angular velocity in x-direction ($\dot{\omega}_x$) can be displayed on the y-axis of the phase plane. Creating these figures for all three Cartesian coordinates, a set-up is formed for three-axis stability control. In the analysis of differential equations, limitations to the solutions of these equations can be identified using the phase plane visualisation.

Within spacecraft attitude control, the phase plane is used to identify so-called ”switching lines”. Looking at Figure 2.7, which was adapted from Jang, Plummer, and Jackson [24], for each axis, a certain starting point can be identified that is different from the desired attitude. The attitude for the starting point is here defined by the angle θ_x and the angular velocity ω_x. When this starting point is outside of the switching lines, a thruster firing is started. In the case of this figure, while being on the bottom left of the solid lines around the origin, a positive thruster firing is started. This will increase the angular velocity and also increase the angle. For each thruster firing time step, the current location within the phase plane is re-evaluated, and a new thruster command is created again. Between the solid lines, two different regions can be distinguished, where no thruster firing is initiated. One of these is called the drift zone, in which the angular velocity remains constant and the attitude angle changes linearly over time. The other region, the attitude hold channel, indicates a safe zone in which convergence towards the solution takes place. As can be seen from the trajectory in the equation, a linear change over time of the attitude angle is observed towards the right side of the graph. This continues until it reaches the right boundary of the no-firing zone, after which a negative thruster fire is induced again. After a number of iterations on the right side of the attitude hold channel, when the spacecraft state has reached the negative y-axis zone of the diagram, the state will move back to the left border of the channel again. Here, it will now enter an oscillatory trajectory around the origin and will convergence towards the solution. Note that, as an example, the actual attitude angle and angular velocity are presented here, whereas the error for these parameters (as seen in Figure 2.5) is the desired value to convergence towards the origin of the phase plane.

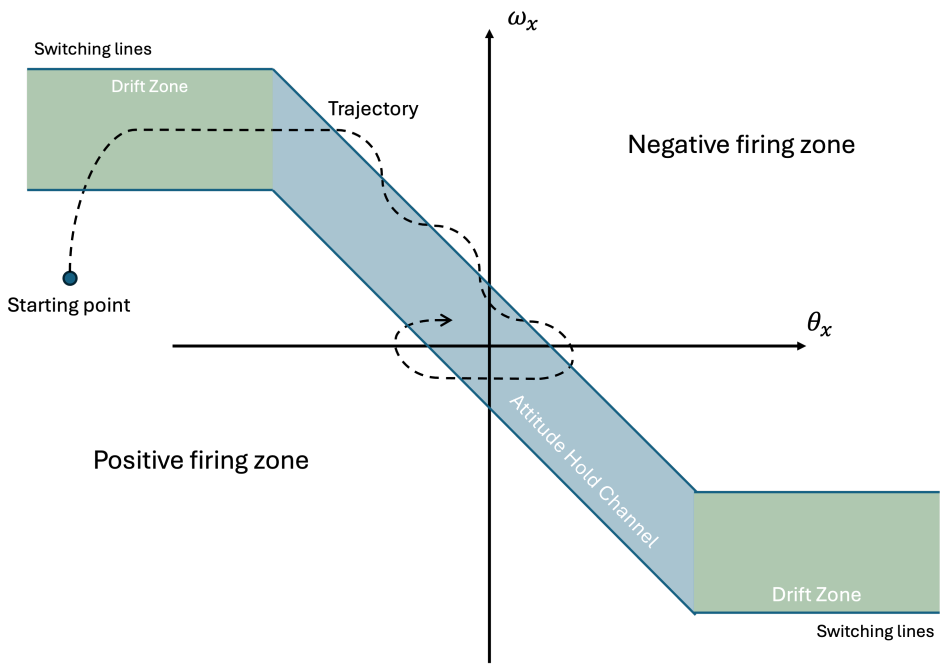


Figure 2.7: Phase plane controller method visualised, adapted from Jang, Plummer, and Jackson [24].

The major improvement made by Lim [35] is that the switching lines in its phase plane diagrams are supplemented with an additional switching line each. This is depicted in Figure 2.8. When the spacecraft state is between the solid and dashed lines, also called the hysteresis region, the previous state is maintained. This means that, coming from the drift zone, the non-firing thruster setting will be maintained, but coming from outside of the drift zone, the thruster firing setting will be maintained. The width and height of the hysteresis region on the phase plane are defined by the attitude hysteresis (θ_{hys}) and rate hysteresis (ω_{hys}). The main reason for the inclusion of such a hysteresis region is the event that flexible-body responses induce small state changes during firing. By doing so, the thruster switching lines might be passed and unnecessary switching is activated.

Cilliers, Steyn, and Jordaan [12] present an improved version of the phase plane controller by Lim [35]. The research recognises the problem of the usage of commercial-off-the-shelf parts with regard to thruster configurations on CubeSats. Control strategies for single axes, such as explained before with the phase plane method, are not effective nor efficient in these cases, and a multi-axis control strategy is therefore proposed. The primary idea of the research is that change in angular velocity around only one axis can be obtained by firing multiple thrusters, of which some thrust components then cancel each other out.

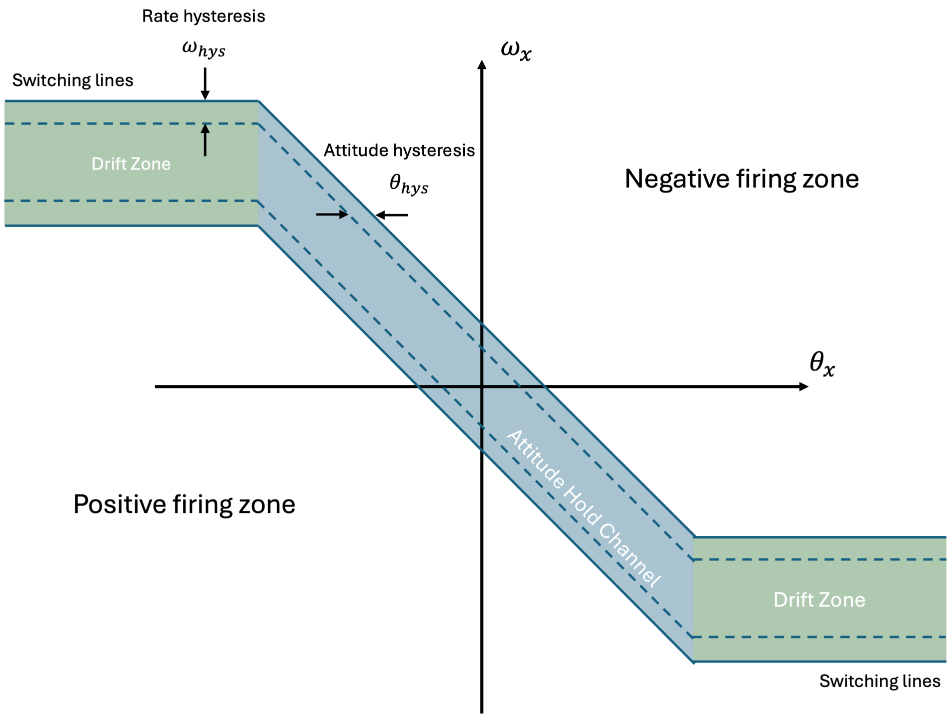


Figure 2.8: Phase plane controller method visualised with hysteresis regions.

2.3.5. Integration

For any dynamics problem, setting up the equations of motion is not the end of the story. They need to be used and propagated over time, in order to assess an object's behaviour over that time span. In some cases, analytical solutions exist that provide the exact answer to the problem. In most realistic cases, however, the equations of motion are so detailed that propagation over time is only possible with a numerical solution, which is the result of the integration of a differential equation over time.

In its most basic form, a differential equation can be written as shown in Equation 2.63, in which $\dot{y}$ denotes the time derivative of the state y, and $f(y,t)$ is a function dependent on both y and t. In addition, an initial state of the function has been given, namely y_0. Since the exact solution cannot be found, an approximation solution of the differential equation should be found for discrete time steps, as shown in Equation 2.64. Note that the previously continuous function y has now been discretised in a total of N time steps.

$$\begin{aligned}\dot{y}(t) &= f(y,t)\\ y(t_0) &= y_0\end{aligned} \tag{2.63}$$

$$\begin{aligned}\bar{y}(t_i) &\approx y(t_i)\\ i &= 0...N\end{aligned} \tag{2.64}$$

From the approximation solution, an error is found, as displayed in Equation 2.65. This error can be evaluated at each time step t_i. In numerical integration, an integrator or solver for a differential equation is used, that creates this approximation solution and with which the state on each time step can be evaluated. A large number of integrators exist, and each have their own advantages and disadvantages with respect to integration speed, error magnitude and adaptability to the problems at hand.

$$\epsilon(t_i) = \bar{y}(t_i) - y(t_i) \tag{2.65}$$

Within spacecraft attitude control, different integrators can also be used for the integration of Euler's equations of rotational motion. The most common and computationally inexpensive integrator is the forward Euler method. A more advanced and also widely-used integrator is the Runge-Kutta-4 integrator. In this section, both integrators will be examined as solvers for spacecraft attitude control problems.

2.3.5.1 Forward Euler method

As described in "The Forward Euler Method" [51], the forward Euler method is a simple and widely-used numerical integrator for solving ordinary differential equations (ODEs). It approximates the solution by using the derivative of the state at the current time step to estimate the state at the next time step. The method is defined mathematically in Equation 2.66.

In words, the state at the next time step, $y(t_{n+1})$, is calculated as the state at the current time step, $y(t_n)$, plus the product of the time step size $h = \Delta t$ and the derivative of the state, $\dot{y}(t_n)$, at the current step. Here, $\dot{y}(t_n)$ is derived from the governing differential equation, as shown in Equation 2.67.

$$y(t_{n+1}) = y(t_n) + hf(t_n, y(t_n)) \tag{2.66}$$

$$\dot{y}(t_n) = f(t_n, y(t_n)) \tag{2.67}$$

The forward Euler method is computationally inexpensive and easy to implement, making it suitable for simple problems or as a starting point for understanding numerical integration. However, it has limitations: it is only first-order accurate (the global truncation error is $\mathcal{O}(h)$) and may become unstable when applied to stiff systems or when large time steps are used. The global truncation error consists of the errors made for each time step, and accumulated over the entire time span. As can be expected, the higher the order of the local and global truncation errors, the more accurate the solver is.

2.3.5.2 Runge-Kutta 4 method

The Runge-Kutta 4 (RK4) method is another popular numerical integrator, as described in Butcher [9]. It is a fourth-order method, meaning its global truncation error (GTE) is $\mathcal{O}(h^4)$, where h is the time step size. RK4 achieves this high accuracy by using intermediate steps to estimate the derivative at multiple points within each time interval.

The numerical scheme for the RK4 method is derived from Taylor series expansion and is presented in Equation 2.68.

$$k_1 = hf\big(y(t), t\big), \tag{2.68}$$

$$k_2 = hf\big(y(t) + \frac{1}{2}k_1, t + \frac{1}{2}h\big), \tag{2.69}$$

$$k_3 = hf\big(y(t) + \frac{1}{2}k_2, t + \frac{1}{2}h\big), \tag{2.70}$$

$$k_4 = hf\big(y(t) + k_3, t + h\big). \tag{2.71}$$

The next state, $y(t + h)$, is then computed as:

$$y(t + h) = y(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{2.72}$$

This method works by combining weighted contributions from the slope ($f(y, t)$) at the beginning, mid-point, and end of the time interval. These contributions are calculated using four intermediate evaluations (k_1, k_2, k_3, k_4), and their weighted average is used to advance the solution. The RK4 method balances computational efficiency and accuracy, making it suitable for a wide range of problems. Its fourth-order accuracy means it can achieve good results with relatively large step sizes compared to lower-order methods like the forward Euler method.

2.4. Lunar CubeSat Missions

This section serves as a summary of lunar CubeSat developments over the past years, and an outlook on missions that will take place in the future. A number of scientific feats will be highlighted, specifically for the CAPSTONE and LUMIO missions from NASA and ESA, respectively. The reason these two missions are highlighted, is because the CAPSTONE mission has attained a novel halo orbit, investigating its stability and potential for a future lunar space station, and the LUMIO scientific papers contain extensive information on its ADCS, including its main components.

2.4.1. Overview of Existing Missions

This section shall provide a brief overview of past, current and future lunar CubeSat missions, in order to provide more detailed insights into the context of the research at hand and the most recent developments. The missions' applications and scientific goals will be elaborated upon.

The first notable lunar mission performed by a nano-satellite is NASA's CAPSTONE (Cislunar Autonomous Positioning System Technology Operations and Navigation Experiment) mission. It was launched on 28 June 2022 by a Rocket Lab Electron booster and is currently performing tests and verification for the stability of the lunar orbit chosen for the Lunar Gateway space station, that is planned to be launched by NASA as part of their Artemis program. [18]

In this way, CAPSTONE is an exploratory mission within the first phases of human travel to the Moon. The CubeSat has a 12U volume and is inserted in a near-rectilinear halo orbit that was the result of various computer simulations. It is the first spacecraft to ever attain such an orbit. Its positioning will rely on the presence of a different NASA satellite, the Lunar Reconnaissance Orbiter (LRO), so that no ground stations are needed for accurate navigation and future space missions can also rely on this method of navigation. CAPSTONE's initial mission lifetime of six months has already been expired, and it is still operational in orbit. More information on the CAPSTONE mission and specifically its orbit are provided in subsection 2.4.2.

On 16 November 2022, the Artemis 1 mission launched, containing as primary payload the Orion spacecraft for in-flight testing. Next to the primary payload, ten low-cost 6U CubeSats were launched as secondary payload [38], of which five were designed to investigate the Moon and attain lunar orbits. The lunar CubeSat missions were the EQUULEUS (JAXA), OMOTENASHI (JAXA), LunIR (Lockheed Martin), LunaH-Map (NASA) and Lunar IceCube (NASA). EQUULEUS was deployed to measure the distribution of plasma surrounding the Earth, as well as perform multiple lunar flyby trajectories within the Earth-Moon Lagrange points. Another Japanese CubeSat is OMOTENASHI, with its primary goal to land on the Moon and show that low-cost lunar exploration is possible. The CubeSat was supposed to measure the radiation environment near the Moon and on the lunar surface, but after separation from the Space Launch System (SLS), communication was lost and the mission was terminated by JAXA.

LunIR is a CubeSat mission developed by Lockheed Martin with its primary goal to collect surface spectroscopy and thermography data.[4] As of December 2022, the mission has successfully been completed. Notable about LunIR is the demonstration of an electrospray thruster in lunar orbit. The LunaH-Map [21], or Lunar Polar Hydrogen Mapper, is a CubeSat in a lunar polar orbit around the south pole in order to map the presence of hydrogen up to one meter beneath the surface. A neutron detector was present as payload and it successfully completed its 96-day mission duration. Finally, the Lunar IceCube mission [36] was designed to inspect the amount and composition of water ice deposits on the lunar surface. It uses an infrared spectrometer instrument and was supposed to demonstrate a miniature electric radio-frequency ion engine system as its propulsion system. Since February 2023, contact with the Lunar IceCube mission was lost.

A CubeSat that was supposed to have launched with the Artemis 1 mission, was the Lunar Flashlight mission designed by NASA.[5] Having missed the launch integration mission, it launched on 11 December on a Falcon 9 Block 5 rocket along with the Japanese Hakuto-R Mission (commercial lunar landing mission). Its primary objective was to determine the presence of water ice on the surface and its exact physical state, along with the mapping of its concentration near the lunar south pole, that is located in a permanent shadow. Using near infrared lasers within a spectrometer (infrared spectroscopy) as its payload, it was supposed to measure surface reflection and composition. Due to a failure in the green mono-propellant propulsion system, the mission was abandoned in May 2023, without having reached the desired lunar orbit.

Another mission intended to be present within the Artemis 1 CubeSat batch is the Cislunar Explorers[6],

[4]URL: `https://terranorbital.com/missions/lunir/` [Accessed 11 March 2024]
[5]URL: `https://www.jpl.nasa.gov/missions/lunar-flashlight` [Accessed 11 March 2024]
[6]URL: `https://nssdc.gsfc.nasa.gov/nmc/spacecraft/display.action?id=CISLUNEXP` [Accessed 11 March 2024]

a pair of CubeSat spacecraft each of size 3U and L-shaped, that has its primary goal to assess the viability of water electrolysis propulsion as well as interplanetary optical navigation to attain a lunar orbit. The latter scientific goal is achieved by optical cameras, that measure the sizes of the Earth, the Moon and the Sun whilst they are in an arbitrary lunar orbit. With these optical measurements, the spacecraft are supposed to determine their locations. The launch date for this mission has not been established yet. The CubeSat-based HiveR rover is proposed by Unwerth et al. [53], with the ability to carry a wide array of scientific payloads, such as a 1U robotic arm.

Finally, the LUMIO mission as proposed by Topputo et al. [52] in collaboration with numerous European universities and research institutions, is a 12U CubeSat mission designed to observe, quantify and characterise the meteoroid impacts on the far side of the Moon by measuring optical flashes. The CubeSat shall be deployed with a detector with a sensitivity toward the near-infrared spectrum and it will attain a halo orbit about the Earth-Moon L_2 (second Lagrange) point. The mission is expected to launch in 2027. A more detailed mission description for the LUMIO mission is given in subsection 2.4.3.

2.4.2. CAPSTONE

As mentioned previously, the novelty of the orbit for CubeSat application in the CAPSTONE mission is the reason this is further investigated in this literature review. From Gardner et al. [18], the CAPSTONE mission had its primary goal to aid future participants in the cislunar (between the Earth and the Moon) ecosystem. Its first mission objective was to validate and demonstrate the Near-Rectilinear Halo Orbit (NRHO), a halo orbit that is close to the smaller of two bodies, in this case the Moon in the Earth-Moon system. A halo orbit itself is a periodic orbit associated with one of the first three Lagrange points, L_1, L_2 or L_3, which are depicted in Figure 2.9 for clarification. Note that this is a top view, looking at the orbital plane of the Moon around the Earth. In a two-body system, such as the Earth-Moon or Earth-Sun system, Lagrange points are positions in space where the gravitational forces of the two large bodies, combined with the centrifugal force felt by a smaller object (the spacecraft), create a situation where the smaller object, despite being in motion, maintains a stable position relative to the two larger bodies. For the halo orbit, near-rectilinear refers to the large curvatures seen in comparison to a standard elliptical orbit with the same semi-major axis. The requirement for an NRHO to exist is the presence of at least two other bodies with a third body of negligible mass. It poses a solution to the three-body-problem. [46]

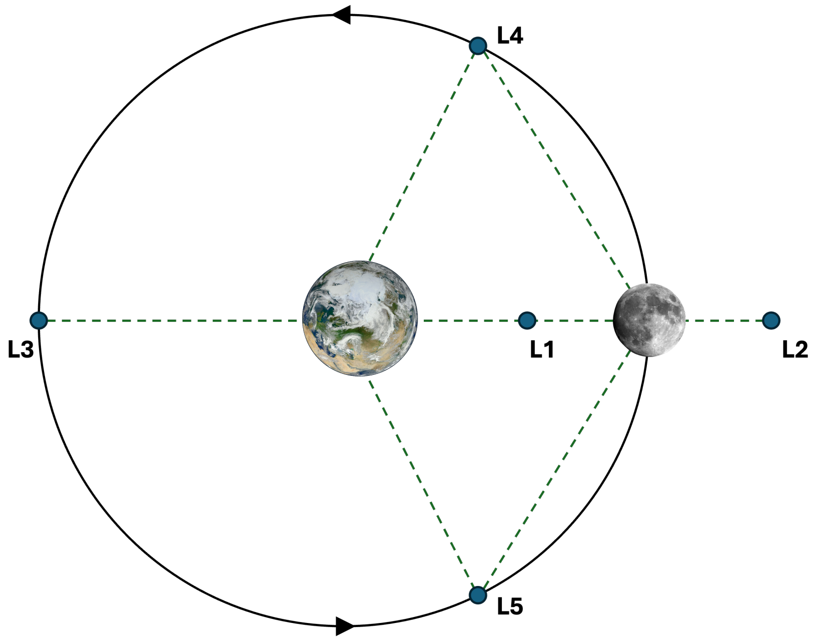


Figure 2.9: Lagrange points in the Earth-Moon system. Top view of the Earth, looking at the orbital plane of the Moon and Earth's North Pole.

A number of issues and uncertainties arise from NRHO physics. First of all, the dynamics of NRHOs are highly sensitive to perturbations due to their location in the gravitational potential of the Earth-Moon system. Small changes in spacecraft velocity or position can lead to significant deviations from the intended path over time. This sensitivity necessitates precise navigation and control strategies to detect and correct for any diverging behavior in real-time. Mission constraints related to eclipsing and communication requirements also complicate the dynamics of NRHOs. Designing trajectories that minimise shadow events due to eclipses by the Earth and Moon is critical for maintaining continuous solar power and communication links. Spreen, Howell, and Davis [46] highlight the use of orbital resonance and careful epoch selection to align the spacecraft's path with periods of minimal eclipse risk. Finally, NRHOs can be reached from LEO with reasonable propellant costs and flight times. However, designing effective transfers requires a deep understanding of the multi-body dynamics and the ability to navigate through the gravitational influences of the Earth, Moon, and other celestial bodies.

NRHOs can have four distinct shapes: they can either be centred around the L_1 or L_2 points, and can attain southern or northern orbits. Two of these types are depicted in Figure 2.10, namely the northern L_1 orbit and the southern L_2 orbit. The remaining two options follow logically from observing the figure. Note that the north and south pole of the Earth are indicated, and the view is directed at the orbital plane of the Moon. The NRHO attained by CAPSTONE was a southern L_2 version with a 9:2 resonant period with the lunar synodic period, meaning that for each two lunar synodic periods of approximately 30 days, CAPSTONE completed nine revolutions in NRHO. Southern refers to the apogee being on the southern side of the lunar orbital plane. The orbit was specifically designed to not experience Earth-induced solar eclipses during the mission and it provides the advantage of an unobstructed view of Earth in addition to coverage of the lunar South Pole. In order to counter-act drift and stay in the correct orbit, a number of orbit maintenance manoeuvrers were executed, in the order of 0.1 $[m/s]$. For attitude control, the main disturbance factors are solar radiation pressure (SRP) and gravity gradient torques near the Moon. SRP can exert forces and torques on the spacecraft due to the pressure of sunlight on its surfaces. This effect can alter the spacecraft's attitude and necessitates the use of attitude control mechanisms to counteract these disturbances. The impact of SRP on the spacecraft's attitude depends on the spacecraft's configuration, surface properties, and orientation relative to the Sun. As the spacecraft passes close to the Moon (at perilune), the differential gravitational pull on the spacecraft's components can induce torques due to the gravity gradient. These torques can change the spacecraft's orientation and must be managed by the attitude control system. The magnitude of the gravity gradient torques depends on the spacecraft's altitude, configuration, and mass distribution. Both disturbances can accurately be modelled using dedicated software. [39] [18]

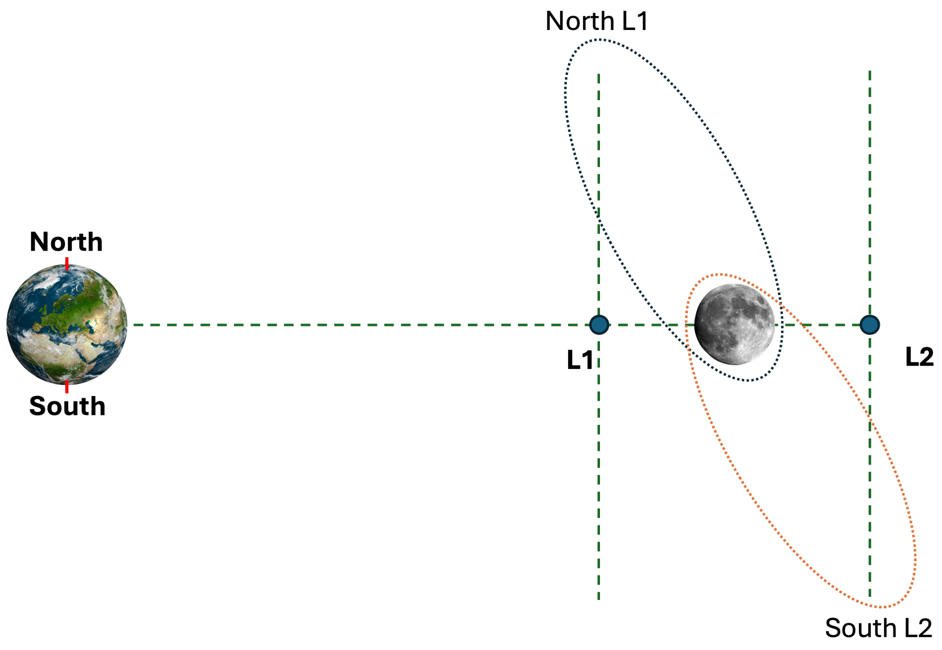


Figure 2.10: The northern L_1 and southern L_2 near-rectilinear halo orbits, in the Earth-Moon system.

2.4.3. LUMIO

The Lunar Meteoroid Impacts Observer (LUMIO) mission, as described in Topputo et al. [52], is a 12U CubeSat mission with the goal to observe and analyse the meteoroid impacts on the lunar far-side. It is the result of ESA's LUCE SYSNOVA competition and has successfully the Preliminary Design Review, currently awaiting for Phase C design. Meteoroid impacts are detected by the uniquely developed LUMIO-Cam, which is able to detect impact flashes in the visible and near infrared spectrum. The mission is set to launch in 2027, according to the latest ESA publication in June 2024[7]. This section will show an overview of the entire mission, including a brief description on its scientific goals and payload, followed by a description of its target orbit. Next, the CubeSat platform will be examined, paying specific attention to the ADCS and its latest requirements.

The meteoroid flux of the Earth-Moon system is the frequency with which meteoroid showers occur in this environment. On Earth, humans are generally safeguarded for these showers due to the ablating effect of the atmosphere. On the Moon, in the absence of an atmosphere, meteoroids impact the surface at full speed. Looking at future lunar exploration by humankind, as was also described in Lasue et al. [31], it is important to have a thorough understanding of this meteoroid flux; its particle sizes, speed of impact and affecting area should be well understood. Models have been built that describe the fluxes, which are similar for the Earth and the Moon. The information provided by the LUMIO mission will aid in improving these models, so that more accurate predictions can be made in the future. The information will help answering questions regarding equipment damage on the lunar surface, which defensive strategies would be needed for human habitats to be built, what the meteoroid distribution over the lunar surface will be and how the lunar situational awareness program should be constructed.

A number of trends within the meteoroid impacts have already been established in literature and summarised in Topputo et al. [52]. For the flux of meteoroids larger than 1 [kg], the number of impacts vary from 1290 to 4000 per year. An updated figure states that approximately 23,000 impacts occur annually, for meteoroids of 30 [g] or larger. A theory has also been established that states that the lunar nearside receives 0.1% more impacts compared to the far-side due to the Earth's gravitational field. Moreover, the equatorial regions receive 10 to 20% more impacts due to the presence of more meteoroids in the orbital plane or at low orbital inclinations, compared to the polar regions. Finally, the leading side of the

[7]URL: `https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Shaping_the_Future/LUMIO_New_CubeSat_Illuminating_Lunar_Impacts` [Accessed 4 December 2024]

Moon receives 37 to 80% more impacts, due to the tidally locked rotation of the Moon around the Earth.

When an object hits the surface of the Moon, its energy is distributed over four different activities: seismic waves, crater creation, particle ejection and radiation. From these phenomena, particle ejection and radiation will be the most beneficial to observe, since they only require remote observation of visible and near infrared light, can be observed over a large surface area at the same time and contain the most complete information about the meteoroid. Observations can only be performed while the lunar surface is illuminated for less than 50%, which significantly obstructs observation from Earth. The absence of an atmosphere, weather and day/night fluctuations are incentives to create space-based payloads for impact observations. Moreover, lower background noise and being able to observe the fully dark lunar surface on the far-side of the Moon makes observation from this far-side the most appealing for such a mission. The combination of these space-based and readily-available ground-based observations complement each other. The space observations performed by LUMIO are done within the 14-day science phase, which is then followed by a 14-day engineering and navigation phase. These phases are dependent on the lighting conditions on the lunar surface, as shown in Figure 2.11.

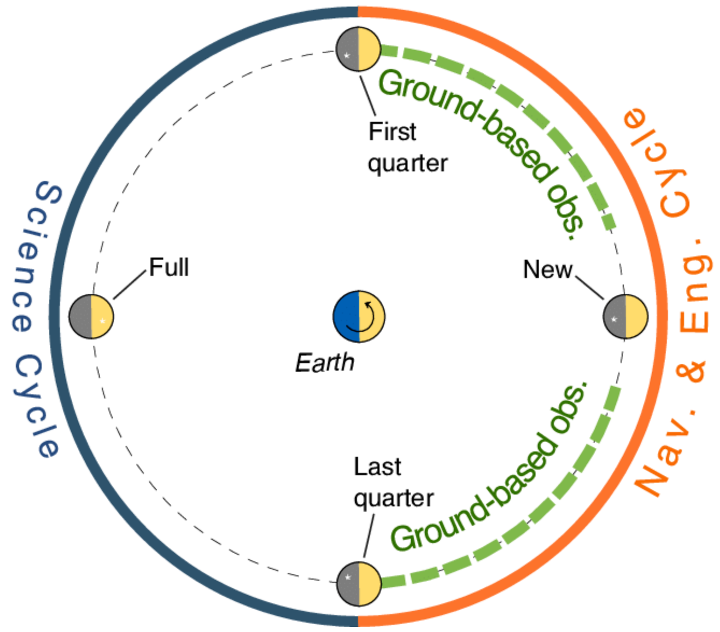
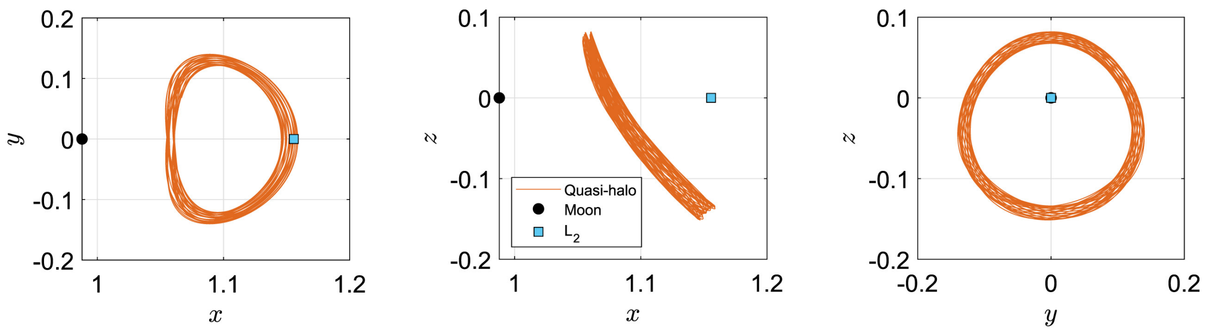


Figure 2.11: Illustration of the Moon phases and the primary trajectories of incoming meteoroids within the Earth-Moon system. The dashed green line marks the segment of the Moon's orbit where Earth-based observations of the nearside are possible. The solid blue line highlights the time frame for space-based observations of the lunar far-side, while the solid orange line denotes periods designated for other operations. Figure taken from Topputo et al. [52].

The LUMIO-Cam, the payload designed to observe the impact flashes of meteoroids, receives incoming electromagnetic waves with wavelengths between 450 and 950 $[nm]$. The total mass of the payload is 3.85 $[kg]$ and its volume is confined within a 3U format. The maximum power required by the payload, including its heaters, is 27.8 $[W]$. Impact flashes are received both in the visible band and the near infrared band, making it possible to analyse the temperatures of the impacts. Furthermore, the detector in the LUMIO-Cam has been adjusted so that it is ideal for imagery where lighting is limited and the frame rate is high. A minimum field of view of 5.68° is required to be able to fully capture the Moon at all times during the science phase, which resulted in a 6.0° field of view including margin.

Regarding the more broad mission geometry of LUMIO, an extensive trade-off for its operational orbit selection has been performed and described in Cipriano, Tos, and Topputo [13]. As a result, the L_2 halo family has been selected for the mission, offering the only realistic option in terms of technical application and economical aspects for CubeSat deployment. Before reaching the halo orbit, LUMIO will be launched, then guided towards the Moon into a Weak Stability Boundary (WSB) transfer, applying multiple Deep Space Manoeuvrers (DSM) and inserted into the halo orbit. This process, up until insertion in the target orbit, takes approximately 150 days. After 1 year of operation in orbit, it will continue into its end-of-life procedure by crashing into the surface of the Moon, poetically becoming what it was meant to observe.

The halo orbit selected for the mission is a quasi-periodic halo orbit around the Earth-Moon L_2 point. The specifics of a halo orbit have been described in subsection 2.4.2, but there are crucial differences between NRHOs and quasi-periodic halo orbits. Quasi-periodic means that there is a certain level of periodicity, but the orbit will not precisely return back to its original position. The quasi-periodic orbit is characterised by symmetry and has sinusoidal motions in multiple planes. It is centred around the Lagrange point and are generally less stable than NRHOs, requiring more station-keeping manoeuvrers. As can be seen from Figure 2.12, the orbit is significantly more symmetric compared to the elongated NRHO. In addition, the perigee of the orbit is not passing close to the lunar surface, but rather remains at a distance from the Moon. For this reason, this type of orbit is more suited for Moon observations and experiences fewer accelerating and decelerating motions. Finally, the Jacobi constant for this orbit, detailing the sum of kinetic and potential energy of the spacecraft in orbit, is equal to $C_j = 3.09$. With this constant, the so-called ”forbidden regions” for the spacecraft can be assessed, without the application of additional forces.



(a) Top view, xy-plane. (b) Lateral view, xz-plane. (c) Lateral view, yz-plane.

Figure 2.12: Three views on the LUMIO operative orbit, in all three Cartesian planes. The Moon and Earth-Moon L_2 point are clearly visible. Multiple orbits have been simulated. Note that the axes are normalised. Figure taken from Topputo et al. [52].

In addition to the main scientific goals of LUMIO, it will execute an autonomous navigation experiment using its images taken of the lunar surface. These images are processed so that the spacecraft position with respect to the Moon is computed, in a Moon-centred reference frame. After this, an Extended Kalman Filter (EKF) is used to estimate the current state of the spacecraft. From Monte Carlo simulations, position accuracy in the z-axis lies below 100 $[km]$ and below 10 $[km]$ in the other two axes, which is regarded sufficient for vision-based navigation.

The overall LUMIO system design will now be examined. A CAD image has been included in Figure 2.13a. An approach in which zero redundancy is tolerated was adhered to, and multiple scenarios involving COTS have been approached. In Table 2.3 below, the most important system requirements for this mission have been displayed. Also, the system requirements will provide insight into the size and volume of the entire system. Two subsystems will be analysed in particular: the propulsion system and the ADCS. Finally, a closer look will be given to the power and mass budgets of the system.

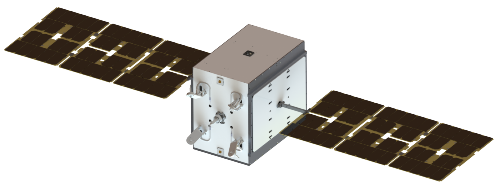
ID	Requirement
SYS-01	The mass of the spacecraft shall not be greater than 28 kg [52].
SYS-02	The spacecraft volume shall not exceed that of a 12U CubeSat [52].
SYS-03	The satellite shall be able to operate in the Lunar environment for at least 1 year [52].
PROP-04	The RCS propulsion system shall provide a Total Impulse for all RCS tasks of 110 Ns [52].
ADCS-01	The spacecraft shall provide an absolute performance error of better than 0.18 deg half-cone during Moon pointing for scientific acquisitions [52].
ADCS-02	The spacecraft shall provide a relative performance error of better than 5 arcsec over 66.7 ms during Moon pointing for scientific acquisitions [52].
ADCS-03	The ADCS shall provide a maximum slew rate of 0.5 deg/s [52].
EPS-01	The EPS shall have a power generation larger than 53.8 W average and a peak power capability of 68 W [52].

Table 2.3: Subsystem requirements for the LUMIO mission, adapted from [52].

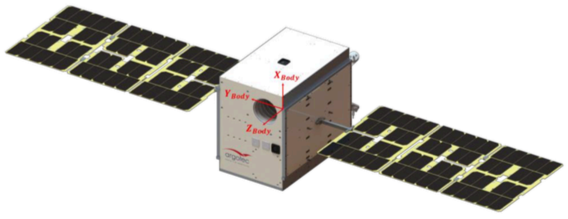
LUMIO's propulsion system consists of two distinct subsystems, namely the main propulsion system for orbital manoeuvrers, and the reaction control system, which is part of the ADCS for torque compensation during manoeuvrers and reaction wheel desaturation. The main thruster system has been designed during the Phase B design but will most likely change during the following design phases. A 1 [N] green mono-propellant designed by ECAPS was chosen after an extensive trade-off (see Cervone et al. [11]). More importantly for the research at hand, the Reaction Control System (RCS) is a cold gas system that uses a refrigerant, R134a as propellant, in order to save tank volume compared to completely gaseous propellants. R134a is present in liquid and gas state at the same time, which makes it beneficial for this purpose. The RCS is equipped with four different thrusters, so that 3-axis control of the spacecraft is possible.

In addition to the thruster control system, an ADCS system designed by Blue Canyon Technologies, the XACT-100, has been chosen for the LUMIO mission. The LUMIO ADCS has been extensively described in Rizza et al. [41]. A closer look to the XACT-100 system can be given in Technologies [50]. The system consists of an electronic circuit board, star trackers and gyroscopes. In addition, two external star trackers, three external sun sensors and four reaction wheels have been added. These external components have also been chosen from Blue Canyon Technologies.[8] Within the XACT-100, the gyroscopes serve to compute the angular rate of the spacecraft and be able to assess the attitude of the spacecraft in combination with the added star trackers. When the angular rate of the spacecraft exceeds 2.0 [deg/s], however, the star trackers do not work adequately any more, and the gyroscopes need to assess the angular rate on their own. Three of the external reaction wheels, the RWp100, are aligned with the spacecraft body axes as shown in Figure 2.13b. They have a angular momentum capacity of 100 [$mNms$]. The fourth reaction wheel, the RWp050, has a momentum capacity of 50 [$mNms$] and is aligned equally with all axes. The entire system is designed to adhere to the limits set by the ADCS requirements as presented in Table 2.3.

[8]URL: `https://www.bluecanyontech.com/components` [Accessed 5 December 2024]



(a) Computer-aided design image of the LUMIO spacecraft after Preliminary Design Review, as shown in Topputo et al. [52]. Main thruster system and the reaction control system are shown, indicating four distinct thrusters. Solar arrays are deployed.



(b) LUMIO spacecraft body axes, as shown in Topputo et al. [52].

Figure 2.13: LUMIO system overview.

The specific pointing strategy of LUMIO consists of two steps, which include Moon-pointing of the LUMIO-Cam and power generation by the solar arrays. The z-axis of the spacecraft body frame is assumed to be aligned with the optical instrument, the y-axis is aligned with the solar arrays and the x-axis complements the coordinate system by use of the right-hand rule. Now, the z-axis has to be pointed towards the Moon, using its own optical navigation-based capabilities for feedback of its attitude. Next, sun sensors are used to let the solar arrays point perpendicularly to the Sun, whilst they can also turn around their own axes. In this way, optimal attitude is attained by LUMIO.

Finally, the mass budget as shown in Table 2.4 and the power budget as shown in Table 2.5 will be given a closer look to. Each subsystem is displayed for its absolute contribution towards to the total, as well as its relative contribution to the total in %, taking the total wet mass as the final total for the mass budget. The power budget is displayed for the science mode, during which impact flashes are detected.

Subsystem	Mass [kg]	[%]
Payload	4.80	16.8
Data processing unit	0.25	0.9
Communication	1.43	5.0
Electrical power system	4.27	15.0
On-board computer	0.56	2.0
ADCS	2.13	7.5
Propulsion	5.34	18.7
Thermal control	0.21	0.7
Structure	4.29	15.0
Total dry mass	**23.28**	**81.5**
Margins	3.51	12.3
Propellant	1.77	6.2
Total wet mass	**28.56**	**100**

Table 2.4: LUMIO mission mass budget, adapted from Topputo et al. [52].

Subsystem	Power [W]	[%]
Payload	19.0	21.4
Data processing unit	5.78	6.5
Communication	14.19	16.0
Electrical power system	10.9	12.3
On-board computer	5.78	6.5
ADCS	2.91	3.3
Propulsion	1.83	2.1
Thermal control	3.15	3.5
Required power w/o primary loss	**66.71**	**75.0**
Margins	**22.23**	**25.0**
Total (margined)	**88.94**	**100.0**
Max. available	98.76	–

Table 2.5: LUMIO mission power budget, adapted from Topputo et al. [52].

2.5. Embedded Systems

Throughout a broad range of industries, such as the automotive, aerospace, health equipment and energy, devices or systems are used that contain computing devices and specialised software, but are not externally observable and, in general, inaccessible to the user. Within spacecraft, for example, data handling and navigation systems are embedded systems that are programmed to perform a specific task within a given amount of time, using low power and having low size and cost. An important aspect concerning embedded systems is the interface between hardware and software. Specific considerations in the design of software need to be taken into account in order to successfully interact with the hardware. In this section, a brief overview of these basic principles will be provided. In addition, a number of spacecraft on-board computer systems will be presented, to provide the reader with current developments in this field. Finally, a widely-used development platform, the STM32 Nucleo board, will be elaborated upon.

2.5.1. Overview of Basic Principles

When creating software for a software-hardware interface, three main aspects need to be taken into consideration: concurrency, scheduling and real-time operations. Co-design between the software and hardware should always be applied. Moreover, compared to traditional programming, limited resources in terms of, for example, power and memory size impose difficulties, as well as the fact that the code should be event-driven, with interruptions and initiations.

Concurrency is a characteristic of systems in which several operations or computations are executed at the same time, and could interact with each other. Each computation is called a thread, and multiple threads can make up a process. Only one process is executed at the same time, but in case multiple threads (multi-threading) are present within this process, multiple computations are being performed and, therefore, concurrency is present in a Central Processing Unit (CPU). In order to handle this concurrency, hardware also has multiple tools such as timers, power management devices and communication devices. In this way, hardware and software work together to be able to process all threads.

Scheduling means that certain pieces of code have to be executed before other pieces of code are executed. A CPU can manage this by using event-triggered interrupts, which means that certain pieces of code are activated when a specific hardware event occurs. An example of such an event-triggered interrupt is a bike computer, which gives as output your velocity (activated code), after your wheel has started turning (hardware event). Different techniques for scheduling are available: static scheduling makes the tasks run in the same order each time, whereas dynamics scheduling mixes up the schedule based on external factors. Also, decisions can be made about the requirement of completing one task before the other. Different schedulers are common to use, such as the run-to-completion and round-Robin schedulers.

As a final consideration in the design of the software-hardware interface, the usage of real-time operating systems (RTOS) needs to be explained. These operating systems generally have three requirements: their execution times should be predictable, they should manage timing and scheduling and they should be working fast. As examples, RTOS have to be able to efficiently manage tasks, handle interrupts and create priority-based scheduling formats. Also, error handling, power management and communications and data exchange between tasks are primary functions of real-time operating systems. All in all, an RTOS is a perfectly stream-lined operating system that, in contrast to regular phones and computers, executes its tasks quickly and effectively. It can be seen as an ”efficient manager of the embedded system”. [37]

To work properly, embedded devices need embedded software to function properly. An RTOS can aid in the handling of all tasks, and could potentially be included in the software requirements. Typical software languages include C or C++, but over the years a variety of high-level languages such as Python or Java have also been included in this range. After the software has been created, the communication line between the processor (on which the software is executed) and the other hardware components should be established. These lines are called protocols and many different options exist, such as Universal Serial Bus (USB), Ethernets, serial ports or Serial Peripheral Interface (SPI).

Looking at serial ports specifically, modern-day implementations of these are established using a Universal Asynchronous Receiver-Transmitter (UART), a form of asynchronous serial communication. Serial communication means that data is sent one bit at a time, sequentially. Asynchronous means that the end points of the communication lines (for example, a computer and a thermometer) are not continuously synchronised by a specific predetermined signal. Instead, the data stream or the bits contain information on the synchronisation by using start and stop signals. The data that is being sent and received, should be formatted for transmission, so that signals such as these start and stop signals are added to the original message. There exist different methods for this formatting, called data framing protocols. Examples of these data framing protocols are the Point-to-Point Protocol (PPP) and the Serial Line Internet Protocol (SLIP). A different form of serial communication is I^2C (Inter-Integrated Circuit), which is synchronous in contrast to UART and is generally faster. It is typically used for communication through different devices on the same bus.

2.5.2. CubeSat OBC

Each year, NASA publishes its State-of-the-Art of Small Spacecraft Technology report ([22]), highlighting annual trends in small satellite developments. In the command and data handling system (C&DH) of CubeSats, two general trends can be identified. First of all, reliability and performance of these systems need to be significantly increased for the scientific goals they serve nowadays; retrieving large amounts of data and handling complex commands requests this of the systems. On the other hand, affordable and easily developed systems leveraging open-source software and hardware are offering a straightforward pathway into space system development, particularly for student teams or individuals without specialised spacecraft knowledge.

Over the last years, a number of commercial vendors of small on-board computers have emerged, offering the processors, memory storage devices, the electrical power system (EPS) and a number of input/output (I/O) ports. In addition, these components are designed for radiation protection for missions of longer duration, in LEO or to deep space. A standard form factor has been adopted throughout the board designs, called the PC/104 form factor, with dimensions 9$\times$9.6 [cm] so that it fits within the standard CubeSat form factor of 10$\times$10 [cm]. In addition, these printed circuit boards (PCB) are often stackable with other subsystem boards, allowing for easy integration and a modular design. An example of stacked PC/104 form factor PCBs is shown in Figure 2.14.

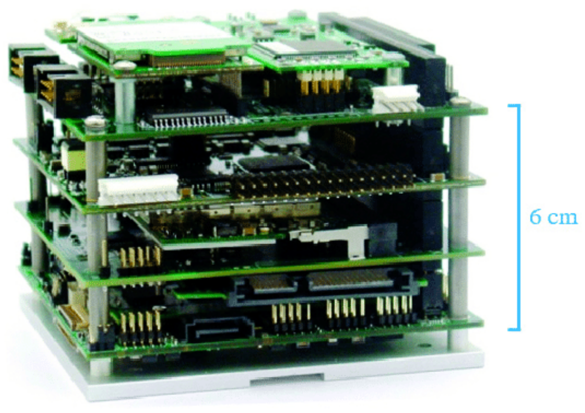


Figure 2.14: Stacked PC/104 form factor PCBs for CubeSat application, as shown in Nieto and Emami [40].

The CPU of an OBC is a micro-controller, which has to be connected to power, memory and other ports externally. Over the past years, commercial vendors are also offering micro-controllers (MCU), which are compact, all-in-one systems that integrates a microprocessor, memory, and peripherals (e.g., GPIO, timers, ADCs, UART, I^2C, SPI) on a single chip. Note that GPIO stands for General-Purpose In/Out and ADC stands for Analog-to-Digital Converter. These MCUs are therefore ideal for the development of embedded systems. For space system developers, the use of ready-made hardware and software platforms that can be implemented without significant integration efforts with other systems is highly preferred. Popular micro-processors used in spacecraft applications are the ARM processors. Micro-controllers typically used in modern-day CubeSat applications are the STM32 controllers from STMicrocontrollers and the Texas Instruments MSP430 series.

As a final example, the on-board computer of the LUMIO mission will be examined. The specific OBC used is the Argotec Fermi OBC, as described in Argotec [5]. It contains a dual-core processor that is radiation-hardened, has 256MB of random access memory, requires 5 $[V]$ of input voltage and has a typical power consumption of 5 $[W]$. It has a large number of digital peripherals, among which are Serial Peripheral Interfaces, UART and GPIO ports. The TRL is 9 out of a total of 9 and the entire OBC fits in a 0.4U volume.

2.5.3. STM32 Nucleo

The STM32 Nucleo boards[9] are development platforms created by STMicroelectronics, designed to accelerate the development of applications based on STM32 microcontrollers. These boards provide an accessible way to experiment with STM32 microcontrollers, allowing developers to quickly prototype and test their applications. They feature an affordable and user-friendly design, making them suitable for beginners, students, and professional engineers alike.

Figure 2.15 shows the STM32 Nucleo-F303RE board, a versatile development platform featuring an ARM Cortex-M4 microprocessor. Nucleo boards are widely used for various applications, including embedded system development, educational projects, and debugging and prototyping. They provide an accessible way to test external sensors, actuators, and communication modules, making them ideal for beginners and professionals alike. The STM32 Nucleo boards integrate seamlessly with STM32CubeIDE (Integrated Development Environment), a software suite tailored for STM32 development. STM32CubeIDE supports code development in C or C++, peripheral configuration using the STM32CubeMX graphical interface, and firmware uploads via the onboard ST-LINK programmer/debugger. The USB interface simplifies both power supply and communication, allowing efficient prototyping and real-time debugging.

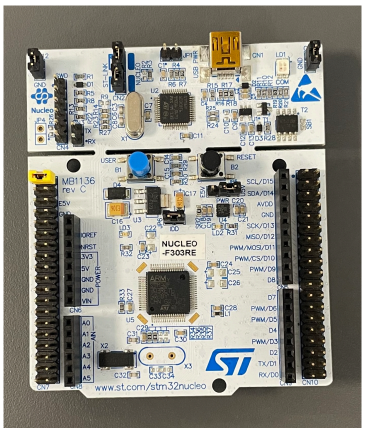


Figure 2.15: Photo of the STM32 Nucleo RE303RE development board, including an ST-LINK on the top for communication via USB.

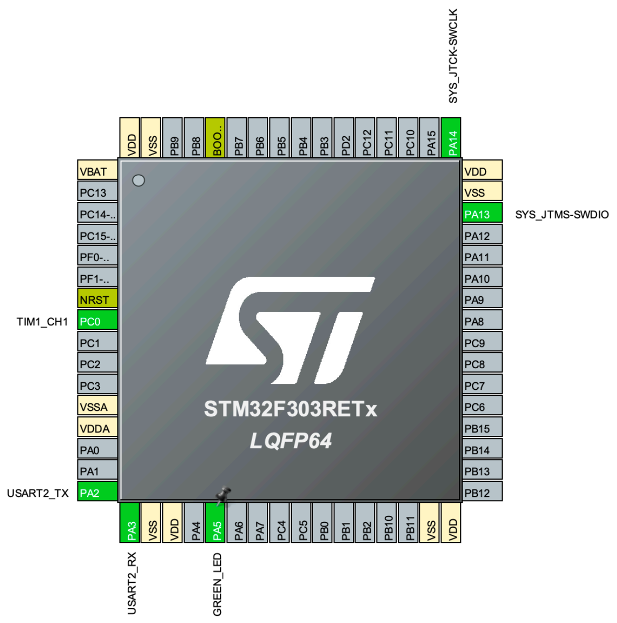


Figure 2.16: STM32IDE view of the microcontroller on the STM32F303RE board, including .

The schematic view of the microcontroller shown in Figure 2.16 demonstrates the configuration of peripherals within the STM32IDE environment. Multiple peripherals, such as UART for serial communication and timers for generating signals, have been configured. In this example, the timers are set to output pulse-width modulated (PWM) signals, which can be used to control external devices, such as electrical thrusters or motors, by modulating their response based on the received signal. The labelled

[9]URL: `https://www.st.com/en/evaluation-tools/stm32-nucleo-boards.html` [Accessed 5 December 2024]

connections surrounding the processor in the schematic are referred to as GPIO pins, grouped into ports (e.g., PA, PB, PC). Each pin can be assigned to specific peripheral functions, such as UART communication or PWM signal generation, depending on the application requirements. These GPIO pins correspond to physical connections on the Nucleo board, allowing developers to interface with external hardware like sensors, actuators, and communication modules.

2.6. Vacuum Arc Thrusters

This section can be viewed as an addition to subsubsection 2.3.3.2 and focuses on the specific type of pulsed plasma thrusters formerly introduced as vacuum arc thrusters. First, the working principle behind these thrusters will be examined in-depth, after which the variation in thrust within control systems for VATs will be elaborated upon. Here, most aforementioned concepts come together and form a solid basis for the research in this report. Finally, an overview of existing vacuum arc thruster modules is provided to assess the state-of-the-art.

2.6.1. Working Principle

As mentioned previously, a specific type of electrical micro-propulsion system is the vacuum arc thruster (VAT), which operates with the components as shown in Figure 2.17. This thruster design includes an anode and cathode separated by a thin layer of insulating material, all contained within a vacuum chamber. When a sufficiently high voltage is applied between the anode and cathode, it initiates a vacuum arc. This arc generates free electrons from the electrode surfaces, either through thermionic emission due to heating or field emission driven by the strong electrical field. These electrons gain kinetic energy from the electric field and, upon colliding with the metal surfaces, cause localised heating and the formation of so-called cathode spots. Material from these spots is ejected as a vapour, which quickly becomes fully ionised to form a plasma. This plasma, conducting the current between the cathode and anode, is accelerated by the electric field to velocities up to approximately 10^4 $[m/s]$, thereby generating thrust. This acceleration of ions not only facilitates the expulsion of plasma but directly results in the creation of thrust. By maintaining the voltage between the anode and cathode at a sufficient level, and by modulating operational parameters, the thruster can continuously generate and control thrust, making vacuum arc thrusters a versatile choice for micro-satellite propulsion. [29]

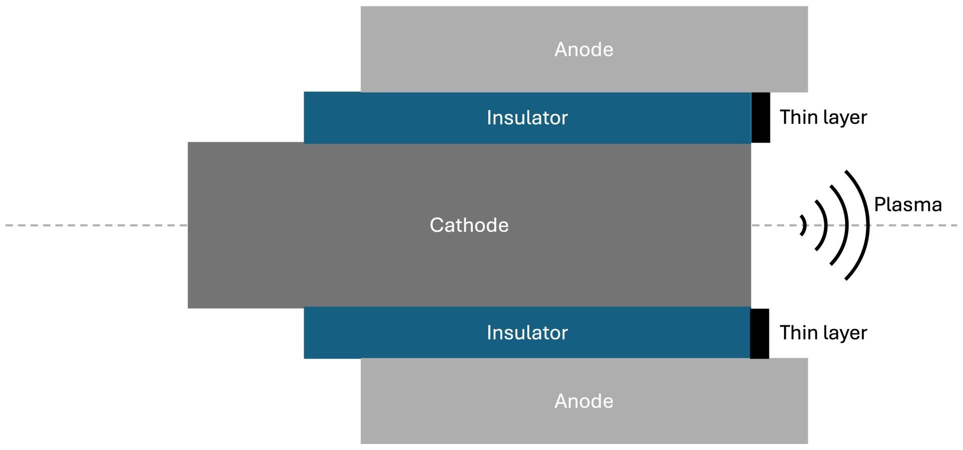


Figure 2.17: Schematic overview of the components of a vacuum arc thruster.

Miniaturisation of vacuum arc thrusters was only possible after the innovation of the triggerless operation. This included the addition of a layer of a specific material, for example graphite, on the outer surface of the anode, cathode and insulation layer. The coating made sure that lower voltage levels were needed for the creation of the vacuum arc, which could be reached in CubeSat designs. In recent years, research has been dedicated to the development of more optimal VAT systems, as their demonstrations in space are limited. One example of this is the Micro Cathode Arc Thruster (μCAT) developed by the George Washington University [10], launched on board a 1.5U CubeSat in order to de-tumble the CubeSat after deployment. Moreover, a vacuum arc thruster developed by Hypernova

Space Technologies was launched on board Endurosat's Platform-2 mission, in order to demonstrate in-orbit manoeuvrers. Research such as presented in Kühn and Schein [30] shows that insufficient reliability is achieved in low-power modes below 1 $[W]$, and that solutions exist by using a combination of specific geometry and power supply.

In order to obtain a preliminary idea of the performance ranges of vacuum arc thrusters, Table 2.6 shows an overview of existing modules, along with its maximum thrust level, specific impulse, size and mass. Further specification on these modules will be presented in subsection 2.6.2 and subsection 2.6.3.

Name	Thrust-to-power ratio [mN/kW]	Max. thrust level [mN]	Vacuum specific impulse [s]	Size
Pocket Rocket	10	0.2	700	45×45×25 $[mm]$ 85 $[g]$
UWE-4 VAT	1.5	0.002	1000	50 $[g]$
AIS-VAT1-PQ	5	0.026	87	42×42×21 $[mm]$ 56 $[g]$

Table 2.6: Existing vacuum arc thruster characteristics overview. [1] [29]

2.6.2. Pulse-Width Modulation

Since vacuum arc thrusters are electrical modules, they receive electrical signals from the OBC or other control board for their activation. The thrusters themselves are connected to the EPS for their basic power input, which is set to a standard voltage (usually 5 $[V]$ or 3.3 $[V]$ and power level. The final power level determines the output thrust of the VAT, and is not able to vary throughout the mission. This means that each vacuum arc thruster on board a spacecraft can only be turned on or off, implying that a control algorithm based on, for example, PID control cannot be implemented. This can be compared to the working of cold gas thrusters, which use valves that usually have no intermediary positions they can deflect to, but only fully close or fully open. Continuous thrust outputs seem not to be possible in this case. Luckily, a technique called pulse width modulation (PWM) can ascertain a varying thruster output. [12]

VATs are ideal modules to use for PWM since their thrust comes from pulsed vacuum arc discharges. The explanation can best be complemented with an example. (SSP) [1] shows the Pocket Rocket data sheet from the company Solid State Propulsion. The maximum thrust output is 200 $[\mu N]$ for an input power of 20 $[W]$, and the pulse repetition rate varies between 0.1 to 10 $[Hz]$. For this example, it is assumed the maximum thrust is obtained by the power input and the repetition rate is set to 10 $[Hz]$, corresponding to 10 discharges every second. The discharges are activated by the electrical signals going into the thruster, which should correspond to the desired thruster activations. The 10 repetitions per second imply that the period of a signal is now set to 0.1 $[s]$, and the signal does not necessarily have to output its maximum value during that period. The duty cycle, given in percentages, indicates for which part of that period the signal is on, and for which part the signal is off. In the exemplary Figure 2.18 and Figure 2.19 below, the signals are shown with a 50% duty cycle and a 25% duty cycle, respectively.

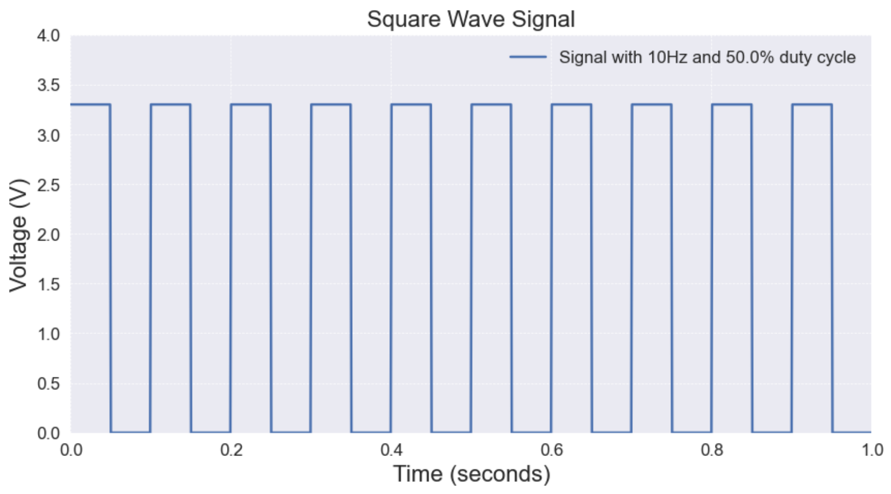


Figure 2.18: Square wave signal of 10 $[Hz]$ with a 50% duty cycle over 1 second.

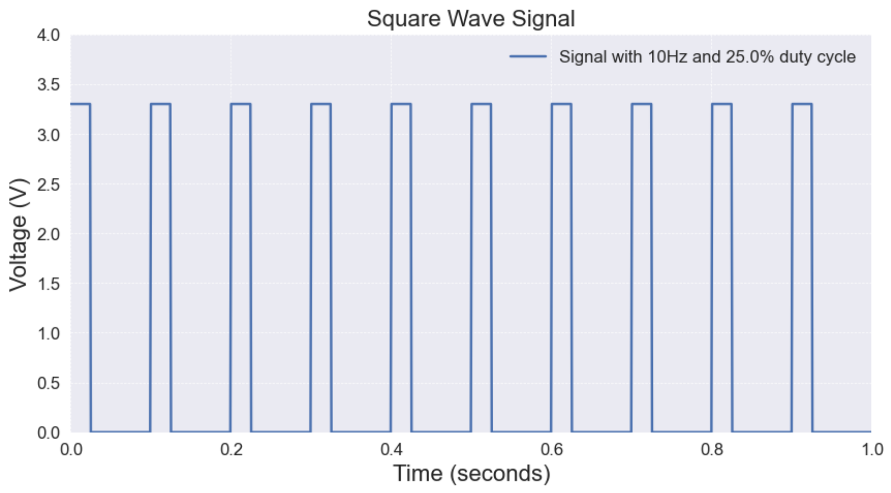


Figure 2.19: Square wave signal of 10 $[Hz]$ with a 25% duty cycle over 1 second.

In order to vary the thrust output of the vacuum arc thruster as mentioned above, the duty cycle of the signals need to be varied. A duty cycle of 100% would in this case result in the maximum thrust, so 200 $[\mu N]$, whereas a duty cycle of 50% as indicated in the figure will result in a thrust output of 100 $[\mu N]$. The signals provided by the on-board computer or other control board can be modulated in their width by adjusting the duty cycle, which in their turn results in varying thrust level outputs. For this reason, the technique is called pulse-width modulation. In control problems, the desired control torque for keeping the spacecraft at the correct (reference) attitude is computed at each time step. This computation is done by the on-board or ADCS computer and results in the computation of desired thrust outputs per thruster, based on their configuration, as was shown in subsubsection 2.3.3.2. Since the thrust level at 100% duty cycle for each thruster is known, the desired thrust can be compared to the maximum thrust, and a duty cycle is immediately computed based on their ratio. This duty cycle is converted into the signals coming from the CPU, and directly converted to vacuum arc discharges at the desired rate, resulting in the desired thrust output per thruster.

An important hardware parameter to take into account is the minimum impulse bit (MIB), which indicates the minimum signal length the thruster responds to. A signal that has a duration smaller than the MIB will not result in any thruster output, and should therefore always be considered in control algorithm development. Finally, as was also analysed in Cilliers, Steyn, and Jordaan [12], a more advanced method of varying the thrust level from thrusters is the use of Pulse-Width-Pulse-Frequency Modulation

(PWPFM), in which the frequency is also varied during the creation of signals from the control torque. In this way, the MIB of the thrusters is less constraining, since it allows for smaller thrust outputs.

2.6.3. Overview of Existing Modules

The work presented by Kronhaus et al. [29] shows a suggested attitude control system for small satellites that are flying in formation with the use of vacuum arc thrusters. In this research, small satellites are expected to replace current large satellites for the space segment of a communications network or Earth observation sensor network. Large satellites that can currently not keep a constant line of sight with a ground network are then replaced by smaller satellites flying in a swarm, so that constant communication is possible. Due to limitations on the resources available to such satellites, their relative distances should be kept below a minimum value, for which orbital manoeuvrers as well as attitude control are of importance. For these actions, electrical propulsion shall be used, so that thrust and power limitations are a prioritised consideration. The specific CubeSat investigated in this research was the University of Würzburg experimental 4 satellite (UWE-4), which include a number of identical CubeSats.

The orbital manoeuvrers to maintain relative proximity between the satellites are done by applying thrust in the in-track and anti-in-track directions. Drift due to the Earth Gravitational Model and drag effects was taken as the test for the orbital control algorithm. Based on two-line element data of four existing CubeSats, a worst-case and regular-case scenario was examined, in which a delta-V requirement of 15 $[m/s]$ and 7.5 $[m/s]$ was estimated. In addition to the orbital manoeuvrers, the thrust vector of the UWE-4 CubeSat had to be pointed in the in-track direction. In order to achieve this, a two-axis attitude control system was implemented based on four vacuum arc thrusters, all on the edges of one side of the CubeSat and pointing in the same direction. It was assumed that one of the axes is allowed to be uncontrollable. Reaction wheels were discarded in this analysis, since power, mass and volume demands were too high in case electrical thrusters were also present. On-off control without modulation was used for the thrusters, so that the highest possible thrust was applied continuously, and because this led to a time-optimal manoeuvrer for attitude control based on thrusters. Due to energy limitations on the CubeSat, only one thruster was allowed to fire at the same time with a maximum thrust level of 2 $[\mu N]$.

A vacuum arc thruster developed by the University of Würzburg was tested for this research, and it was found that using four of these thrusters would achieve the mission requirements. The required delta-V in the general-case scenario could be reached using a tungsten cathode, with an I_{sp} of approximately 1000 $[s]$, and a total propellant / cathode mass of 1 $[g]$ and 2 $[W]$ of power available for the propulsion system. The thrust-to-power ratio was found to be approximately 1.5 $[\mu N/W]$. Next to the successful completion of formation flying proximity requirements, a pointing accuracy of 0.5° was achieved, using single thrust values of 1 $[\mu N]$. The thrusters were used as part of the CubeSat structure and would only require 1/4 of the volume and 1/5 of the mass of the entire CubeSat. With the research, CubeSat formation flying using a vacuum arc thruster system was proved to be possible.

The company Applied Ion Systems, focused on research and development in advanced electrical propulsion systems for small satellites, has developed the AIS-VAT1-PQ thruster, a micro vacuum arc thruster that can be used in 1U CubeSats. Its size is 42×42×21 $[mm]$, its maximum thrust is 26 $[\mu N]$ for a total power input of 5 $[W]$, resulting in a thrust-to-power ratio of approximately 5 $[mN/kW]$. Its total mass is 56 $[g]$ and it has a specific impulse of 87 $[s]$. Note that this is a relatively small example, but its modularity introduces great advantages for especially PocketQube design.[10]

On a final note, Saddul et al. [44] present the CubeSat De-orbiting All-Printed Propulsion System (Cube-de-ALPS), a novel thin-film VAT developed by the University of Southampton in collaboration with the European Space Agency. Designed to enhance the de-orbiting capabilities of CubeSats smaller than 3U, Cube-de-ALPS features a flexible base with coplanar arrays of vacuum arc micro-thrusters (micro-VATs) and supporting electronic subsystems printed onto it. A key innovation highlighted in the paper is the introduction of a novel 5x5 thruster setup, optimising the distribution and operation of micro-VATs to effectively manage the CubeSat's de-orbiting process. Next to this, a gyroscope and Faraday cup were

[10]URL: `https://appliedionsystems.com/products/` [Accessed 09 January 2025]

present in the new propulsion system for attitude determination. The system's performance, particularly its de-orbiting efficiency from a 1400 km orbit, was validated through a series of high-fidelity numerical simulations that incorporated six degrees of freedom for coupled attitude and orbital dynamics. The results confirmed Cube-de-ALPS's viability for ensuring the de-orbit of CubeSats from altitudes as high as 1400 km, significantly enhancing space debris mitigation efforts for small satellites. This advancement in micro-propulsion technology represents a significant step forward in the sustainable management of CubeSats, offering a practical solution for their end-of-life disposal.

2.7. Conclusion

This literature review provided the reader with information about developments within the field of six distinct topics, based on six pre-determined literature review research questions. A broad overview was given on the past, current and future lunar CubeSat missions in order to give a solid foundation of the current developments and achievements and give an elaborate answer to **LQ-01**. These topics were primarily discussed in Section 2.4 but also mentioned throughout the rest of the literature review as exemplary insights. Next, an overview of spacecraft attitude dynamics and control was provided in Section 2.3, highlighting physics, control algorithm solutions and the lunar control environment, effectively addressing **LQ-02**. Actuators for effective spacecraft control and specifically CubeSat control were discussed in much detail, as well as the integration methods for attitude state propagation. In addition, in line with **LQ-03**, possibilities within thruster configurations were mentioned, as well as conversion from required control torques to thruster output values. This information was provided in subsubsection 2.3.3.2. Next, providing an answer to **LQ-04**, an introduction to embedded systems was provided in Section 2.5 including all aspects to take into account when converting a developed algorithm to a software-hardware interface. In addition, a current overview of on-board CubeSat computers was provided with the specification of the widely-used STM32 Nucleo board. Finally, as a specific example within electrical thrusters, vacuum arc thrusters were explained along with current developments in their field. The modulation of signals for effective usage of VATs in control systems was explained as well, all included in Section 2.6 of this report and giving answer to **LQ-05** and **LQ-06**. Based on all the information gathered from research and summarised in this chapter, it can be concluded that all literature research questions have been answered sufficiently and that a firm basis for the start of the research in the rest of this report has been established.

The final goal of this literature review is to identify knowledge gaps and therefore fields of interest for research to provide answers for. From the information provided in this chapter, a number of topics were selected in which current research does not suffice yet, or in which additional research is interesting for future space missions. First of all, electrical thruster applications on CubeSats have been investigated in the past but research to specific applications in attitude control lacks. In addition, the papers that do analyse this do not present a wide range of configuration options to be tested, which can lead to a more profound verification of algorithm robustness. Moreover, the Technology Readiness Levels of the advantageous electrical micro-propulsion units is far behind that of conventional non-electrical modules, calling for an increase in its application in missions, which starts in the inclusion in research. The lunar environment, and specifically the near rectilinear halo orbit that is to be attained by the NASA Lunar Gateway spacecraft, also poses an interesting application for CubeSat deployment in the future due to its stable nature. Also, the future LUMIO mission is relevant for further investigation with respect to its attitude control system, since accurate pointing is desired for this mission and research in this area can add to the verification of the final attitude control system. Moreover, LUMIO's final mass, volume and power requirements for the system as a whole can be re-iterated upon using new ADC strategies. Finally, the application of an ADCS simulation environment to existing hardware modules can add to the validation of existing and new research and introduce educational opportunities in this area. These topics all contribute to the research questions that have been established in Chapter 3.

3 Research Questions & Hypotheses

3.1. Introduction

In this chapter, the main research question and sub-questions are presented, stemming from the literature review in Chapter 2. These questions aim to address the identified knowledge gaps and contribute to the broader body of knowledge. Each question and the research objective will have a unique identifier for consistency throughout the report, with the research centered entirely around answering these questions.

The knowledge gap identified in the previous chapter highlights opportunities in CubeSat micro-propulsion research, driven by advancements in miniaturization, cost reduction, and rapid deployment. While non-electrical thrusters are widely used, electrical thrusters lag behind in Technology Readiness Level and have rarely been studied for standalone ADCS applications. This gap offers potential benefits in cost, mass, and volume for CubeSats. The ESA LUMIO mission, launching in 2027, provides an opportunity to explore replacing its ADCS with an all-electrical thruster system, advancing both simulation and real-world feasibility studies.

3.2. Research Questions

Based on the aforementioned research gap, the research objective can be described as written below.

Research Objective [RO-01]: *Assess various electrical micro-thruster configurations on the LUMIO mission, a meteoroid-detecting 12U CubeSat in cis-lunar environment, for their attitude control performance, robustness and connectivity.*

This main research objective automatically leads to the primary research question as described below.

Research Question [RQ-01]: *What is the impact of adjusting the ADCS configurations, consisting of electrical thrusters only, on the attitude control performance, robustness and connectivity of the LUMIO mission, a 12U lunar CubeSat?*

The last part of **RQ-01** states three important terms: *performance*, *robustness* and *connectivity*. Each of these three should be sub-divided in additional research questions, to obtain a sufficient idea of what is meant with these terms and guide the research into the desired direction. With *performance*, the research will attempt to assess whether replacing the current LUMIO ADCS with electrical thrusters will work, and if so, what its response will be for different electrical thruster configurations. Replacement means, removing all the reaction wheels and current thrusters, and placing electrical thrusters in their place. Direct comparison between the old and new systems can then also be conducted, and feasibility can be assessed. If the system works, it can be tested for *robustness* by introducing extreme situations, so that the attitude response to these can be assessed. Finally, *connectivity* addresses the practical application of the entire research; is it possible to adapt an ADCS simulation to existing hardware modules and assess their performance in real life?

Sub-question 1 [SQ-01]: *How do different electrical thruster configurations influence the pointing accuracy capabilities and power requirements of a 12U meteoroid-detecting CubeSat in a near-rectilinear halo orbit in the Earth-Moon L_2 point using a PD-based control algorithm?*

- **[SQ-011]**: Which orbital characteristics are relevant for the development of the attitude control algorithm?
- **[SQ-012]**: Which and how many different electrical thruster configurations shall be taken into account?
- **[SQ-013]**: Will the electrical-thruster-based attitude control system be able to adhere to the LUMIO ADCS requirements?
- **[SQ-014]**: What are the power requirements of using electrical thrusters for the ADCS?
- **[SQ-015]**: How does the usage of electrical thrusters for the ADCS compare to using reaction wheels with a specific desaturation strategy in terms of power usage?
- **[SQ-016]**: How does the usage of electrical thrusters for the ADCS compare to using reaction wheels with a specific desaturation strategy in terms of pointing accuracy?

Sub-question 2 [SQ-02]: *What influence do single thruster failure, de-tumbling manoeuvrers and solar array deployment have on the pointing accuracy of an ADCS consisting of electrical thrusters?*

- **[SQ-021]**: What are the thruster failure modelling options?
- **[SQ-022]**: Are all previously developed configurations feasible within the robustness tests?
- **[SQ-023]**: What is the maximum angular rate for which accurate de-tumbling is possible?
- **[SQ-024]**: What are the driving factors behind the difference in pointing accuracy due to solar array deployment?

Sub-question 3 [SQ-03]: *How can the control algorithm be effectively transformed in embedded code so that functional connectivity is established with an existing vacuum arc thruster hardware module?*

- **[SQ-031]**: What is the influence of the hardware specifications on the software adaptation?
- **[SQ-032]**: To which extent can the software adaption be generalised for a broader range of hardware modules?
- **[SQ-033]**: Which hardware module is suitable to serve as a satellite OBC?
- **[SQ-034]**: What is the difference in execution times between a hardware-integrated simulation and a purely mathematical simulation?
- **[SQ-035]**: Which aspect within the hardware-integrated simulation causes the largest difference in execution time?

3.3. Hypotheses

Based on the previous research questions and sub-questions, a number of hypotheses can be created that are tested in this research work. Hypotheses adhere to rules set out by, for example, Williamson [55], which are: good hypotheses should be stated in correct terminology, should be as brief and clear as possible, describe a predicted association or distinction among two or more variables, can be tested and are rooted in previous understanding, derived from a review of literature or theoretical frameworks.

The approach adhered to in hypothesis generation is firstly identifying the most critical sub-sub-questions, and formulating informed hypotheses regarding them. From there, more general hypotheses can be constructed for the sub-questions, which naturally leads to a hypothesis for the main research question. In Table 3.1 below, each row represents a hypothesis for this research and the first column indicates the unique identifier for the hypothesis. The second column indicates the research question it attempts to make a prediction for, and the third column states the hypothesis.

ID	RQ	Hypothesis
H-01	SQ-011	The position of the spacecraft with respect to the Moon is important for determining its reference attitude as well as disturbance due to the gravity gradient torque. The position of the spacecraft with respect to the Moon, Sun and Earth is relevant for the solar radiation pressure exerted on its panels.
H-02	SQ-013	As long as three-axis stabilisation of the thruster configuration is possible, the LUMIO ADCS requirements are adhered to using an electrical-thruster-based ADCS.
H-03	SQ-014/015	It is expected that the average maximum power input for electrical thrusters in an ADCS is 20 $[W]$, since this was the maximum value observed within the existing vacuum arc thruster modules. For thruster configurations between 6 and 12 thrusters, the total could amount to 240 $[W]$ of input power. Compared to the several Watts required by a reaction wheel system, it is expected that this ADCS proposal will require significantly more power.
H-04	SQ-01	The overall pointing accuracy of a 12U meteoroid-detecting CubeSat in cis-lunar environment will be increased for an increased number of thrusters present for the ADCS, as long as three-axis stabilisation of the spacecraft is possible.
H-05	SQ-022	Configurations with three thruster pairs, or equivalently six thrusters, or fewer are not suitable for single thruster failure tests, since three-axis stabilisation will not be possible with the malfunctioning of one of the thrusters.
H-06	SQ-023	Within the operational limits of the hardware components, any angular rate is feasible for balancing as long as three-axis spacecraft stabilisation is possible.
H-07	SQ-024	The most significant factor in performance difference between deployed and undeployed solar arrays is the change in mass moment of inertia. Solar radiation pressure increase will not be as significant.
H-08	SQ-02	All configurations as assessed for **SQ-01** will be able to counteract de-tumbling and show the required performance with deployed solar arrays, but configurations that rely on the geometrical position of one of their thrusters for three-axis stabilisation will not be able to counteract single thruster failure.
H-09	SQ-035	The most critical execution time lengthening will occur due to the communication line, since messages need to be encoded, sent, received, and decoded again.
H-10	SQ-03	Transformation of on-board calculations as present in an ADCS simulation have to be converted to code in C and uploaded to a micro-controller unit, which in its turn is connected to thruster modules and has been generate and receive thruster signals.
H-11	RQ-01	Adjusting the electrical thruster configuration on the LUMIO mission's 12U CubeSat will enhance its attitude control performance, robustness, and connectivity by improving pointing accuracy with an optimised number of thrusters, ensuring resilience against de-tumbling and solar array deployment, and enabling functional integration with hardware through embedded code adaptation.

Table 3.1: Hypothesis table containing hypothesis ID, related research questions and hypothesis.

4 Attitude Control Simulation

4.1. Introduction

Within the scientific framework laid out in Chapter 2, a number of research topics to further investigate could be introduced. In order to effectively scope this research within acceptable bounds, Chapter 3 proposed a main research question, with three sub-questions as guiding tools within this work. In order to find answers to these, the research should also be divided in three distinct areas. This chapter will cover the entire first and second sub-questions, which will dive into the different electrical thruster configurations, their application to LUMIO and assess their performance and robustness to extreme situations. This is all done by means of an attitude control algorithm, that will be developed from scratch and explained in this chapter.

First of all, in Section 4.2, the context of this research will be detailed, including explanation on which important aspects of the LUMIO mission will be used, what the operational orbit will include and how it is retrieved, and which thruster configurations will be assessed. Next, Section 4.3 will specify the assumptions used within the numerical simulation, including the simplifications made and conventions adhered to throughout the research. Next, Section 4.4 will present a step-by-step walk-through of the developed attitude control code and highlight its individual programming blocks. Then, Section 4.5 will introduce the robustness methods used within the research, explain the relevant parameters for the result analysis, and explain how the original simulation code was modified to achieve this goal.

4.2. Context

For any simulation development research, the context of the simulation is of utmost important for creating a basis with which all different blocks are constructed. In view of the current developments in CubeSat technology and the opportunity to analyse future missions, the ESA LUMIO mission has been selected as the mission of context in this research, as elaborately detailed in subsection 2.4.3. Arguments for this selection are threefold:

1. LUMIO is set to launch in 2027, and multiple research institutions are currently designing its mission architecture and subsystems. Because of this, relevance is created by the current parallel investigation done by this research and the LUMIO team.
2. LUMIO is a 12U CubeSat that will be in a halo orbit around the Earth-Moon L_2 Lagrange point, posing it as a unique research opportunity in a deep space environment. Electrical thruster applications have not been studied to a great extent in similar environments.
3. The current design of LUMIO's ADCS is a conventional reaction wheel system with thrusters for momentum wheel desaturation. Effective comparison with infrequently-studied full electrical thruster ADCS is therefore easily possible, with its previously designed ADCS as benchmark.

In view of the near-future launch of the LUMIO mission, and its current rapid developments, paper publications regarding the mission have actively been monitored, with the help of the supervising professor Mr A. Cervone, who was involved with the mission himself. Further contact was also established with Mr F. Topputo from the Politecnico di Milano university.

4.2.1. Orbit

Although the future LUMIO's orbit has been described in papers and simulated in recent research such as Topputo et al. [52], direct ephemeris data for the orbit was not openly available. This research is not focused on orbit propagation or determination, which is why no effort has been made to reconstruct the exact orbit using simulation software. Instead, investigation was done to orbits that could easily be retrieved from stored ephemeris data. Before detailing the exact orbit that was chosen, the method of data retrieval will be elaborated upon.

The online solar system database for ephemeris retrieval created by the Jet Propulsion Laboratory (NASA) is called the Horizons System.[1] It includes more than 290 planetary satellites, all 8 planets, the Sun, selected spacecraft and more. Access is possible via their website, through the built-in command-line on a personal computer, through email or via an Application Programming Interface (API). In addition, a dedicated Python package is available called `astroquery` with a sub-class named `jplhorizons`.[2] A crucial functionality that is used from this package, is the retrieval of vectors for specific objects.

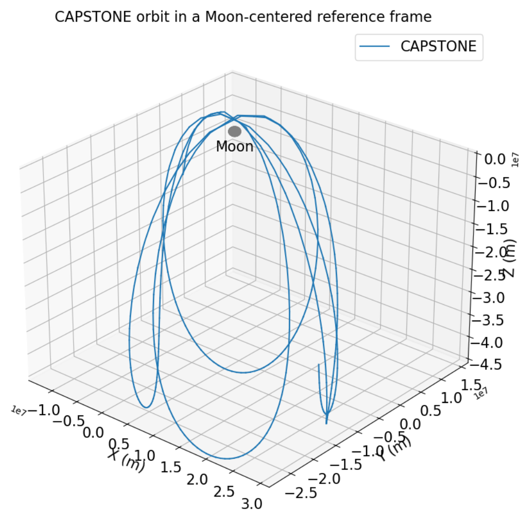
The relevant objects for this research are the Moon, the Sun and the CAPSTONE spacecraft. Each of these bodies has a specific ID within the JPL Horizons library, specified in Table 4.1. For each of the bodies, the vectors include the Cartesian position coordinates (x, y and z) and their velocities (v_x, v_y and v_z), with respect to a selected celestial body or other location. In this way, Moon-centred ephemeris can automatically be collected for the CAPSTONE spacecraft, for example. Moreover, a range of epochs can be indicated, along with a pre-determined time step for data collection. The minimum time step available is 1 minute, meaning that Cartesian positions and velocities for the bodies can be retrieved at every single minute. The valid and accurate epochs for which data can be retrieved for each body is also indicated in Table 4.1. Note that the CAPSTONE mission is still active, and its ephemeris is propagated two weeks into the future (time of writing is 7 December 2024 14:15:00).

Body	Sun	Moon	CAPSTONE
ID	10	301	-1176
Validity	Infinite	Infinite	[10:06:06.277 28-06-2022] to [00:01:09.184 31-12-2024]

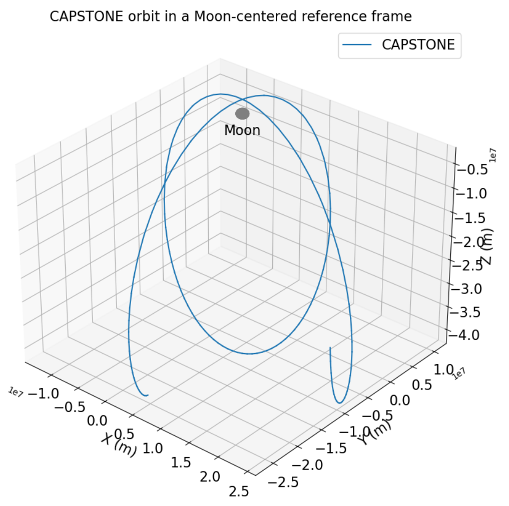
Table 4.1: Overview of three target bodies for this research, including their JPL Horizons ID and validity.

As introduced, the NASA's CAPSTONE historic ephemeris data will be used as the context orbit for the attitude control simulation in this chapter. As observed in Chapter 2, the near-rectilinear halo orbit attained by CAPSTONE is different in various aspects to the desired halo orbit to be attained by LUMIO (see Figure 2.12). The CAPSTONE orbit is plotted in Figure 4.1a based on its ephemeris data, between [00:00:00 01-01-2023] and [00:00:00 01-02-2023]. It can be seen that CAPSTONE performs approximately four distinct orbits in this time span, which are characterised by a close fly-by of the Moon and then a southern movement towards the Earth-Moon L_2 point. Since this L_2 point is fixed with respect to an Earth-Moon reference frame, and the figure is produced in a Moon-centred reference frame (non-rotating), it was to be expected that the southern apogee passes around the L_2 point would shift around the Moon in a period of one month, since the orbital period of the Moon around the Earth is also approximately one month.

[1]URL: `https://ssd.jpl.nasa.gov/horizons/` [Accessed 7 December 2024]
[2]URL: `https://astroquery.readthedocs.io/en/latest/jplhorizons/jplhorizons.html` [Accessed 7 December 2024]



(a) Moon-centred CAPSTONE orbit, plotted using JPL Horizons ephemeris data from [00:00:00 01-01-2023] to [00:00:00 01-02-2023].



(b) Moon-centred CAPSTONE orbit, plotted using JPL Horizons ephemeris data from [00:00:00 01-01-2023] to [00:00:00 15-01-2023].

Figure 4.1: Comparison of Moon-centred CAPSTONE orbits for two different timespans.

For the attitude control simulation at hand, CAPSTONE's orbit between [00:00:00 01-01-2023] and [00:00:00 15-01-2023] will be used. The orbit during this time period is presented in Figure 4.1b. The chosen timespan is ideal because it encompasses two complete orbits of CAPSTONE, during which two close flybys of the Moon occur. Since the distances to the Moon's centre during these flybys are nearly identical, the attitude control simulation can be analysed under similar conditions for both events, allowing for consistent and comparable responses. The inclusion of two fly-bys allows for the close observation of either one of them for quaternion response behaviour, for example. A second reason for choosing these dates is the fact that the CAPSTONE mission officially started on 14 November 2022 and ended on 18 May 2023, which means this orbit is within its operational timespan.

As mentioned, the CAPSTONE southern L_2 NRHO is different from the quasi-periodic halo orbit outlined in LUMIO research. The six reasons that this specific orbit is used throughout this research, are detailed in the list below, addressing the important differences with LUMIO's orbit and explaining why the selected orbit is still relevant for this research.

1. Both the LUMIO and CAPSTONE orbits are located in cislunar environment, at a sufficient distance from to Earth to not be affected by its magnetic field or gravitational potential for significant attitude disturbance.
2. The CAPSTONE orbit is proven to be stable (see Gardner et al. [18]) and therefore suitable for CubeSat usage.
3. From its definition, the CAPSTONE orbit will always be able to observe the lunar far-side. During close fly-bys, the far-side will always be in view as well. For LUMIO, the same holds, although its disk area is larger due to its larger distance from the Moon.
4. CAPSTONE's period is approximately 7 days, compared to the approximate 14-day period of LUMIO. The common divider allows for easier extrapolation of conclusions.
5. CAPSTONE's orbit provides a more extreme environment for attitude control testing, since its close fly-by of the Moon requires its attitude to change significantly further than would be required in LUMIO's orbit, at an approximate constant distance from the Moon. Moreover, this close fly-by induces an increase in the gravity gradient torque, which has an inverse relation with the cube of the distance to the body (see Equation 2.32). For these two reasons, it is on a scientific level more interesting to analyse the CAPSTONE orbit compared to the LUMIO orbit.
6. Due to the approximate same distance from the Sun, the solar radiation pressure will be equal. Both orbits have been designed to be eclipse free ([18], [13]).

From this list, the most important conclusions to be drawn are: the environments of both orbits do not differ significantly, since they are both situated in cis-lunar environment and will, on average, experience the same degrees of incoming solar radiation and experience the same gravitational disturbances. The clear benefit of analysing the CAPSTONE orbit, is the fact that it shows more extreme behaviour compared to the LUMIO orbit, since it passes closely by the Moon. This will induce an increase in gravity gradient torque compared to the regular LUMIO mission, and will also require more extreme camera pointing manoeuvres. Now, one may conclude that the design of the LUMIO spacecraft was not intended for this orbit, but the CAPSTONE orbit poses a great limit-case testing, since conditions will be more extreme. If the system behaves well in the CAPSTONE orbit, it will also be suitable for the LUMIO orbit.

As a numerical comparison, Table 4.2 shows average maximum values for the solar radiation pressure and gravity gradient torque, as a comparison between the two orbits. In addition, the Moon-centred position coordinates along the orbit have been posted for comparison as well. For these calculations, the following conditions have been included:

- For the CAPSTONE orbit, an analysis time frame between [00:00:00 01-01-2023] and [00:00:00 15-01-2023] will be used.
- For the solar radiation pressure, the extreme case will be selected in which the Earth, Moon and Sun are aligned, and the spacecraft is situated around the Earth-Moon L_2 Lagrange point. In this situation, the spacecraft is as close to the Sun as possible whilst also present in either the CAPSTONE or LUMIO orbit. It is assumed that, in this Moon-centred reference frame, all the bodies are aligned along the x-axis and the spacecraft coordinates are extrapolated for this situation. For the CAPSTONE orbit, the spacecraft will be in apogee on the southern region of the Lagrange point. For the LUMIO orbit, the most extreme case for the solar radiation pressure would in this case be furthest from the Moon. The coordinates for this are extracted from Figure 2.12, using a distance between the Moon centre of mass and the L_2 Lagrange point of 64,500 [km].
- For the gravity gradient torque, the most extreme values stem from the closest approach to the Moon, based on Equation 2.32. Therefore, the perigee coordinates for CAPSTONE will be used as the spacecraft location. For the LUMIO orbit, coordinates will again be selected from Figure 2.12, closest to the Moon, which results in a position that is approximately on the y-axis of the frame.
- Note that the calculations serve mainly to give an idea of the difference in magnitude for both orbits and therefore verify the statement that the CAPSTONE orbit is more extreme compared to the LUMIO orbit. These results will not be used for further analysis since they only serve as tests. The CAPSTONE spacecraft position for analysis of the solar radiation pressure is also not one from the actual ephemeris data, but rather interpolated to be exactly on the y-axis. It is, however, equally far from the Moon as the actual ephemeris data state, with the same distance magnitude as found in the orbit apolune

Orbit	**Position vectors [km]**	$\lvert T_{GG}\rvert_{\text{max}}$ [N]	$\lvert T_{SRP}\rvert_{\text{max}}$ [N]
LUMIO	GG: [30,100; 0; 21,500] SRP: [64,500; 0; -55,900]	$9.872 \cdot 10^{-11}$	$7.836 \cdot 10^{-21}$
CAPSTONE	GG: [437; -348; 3348] SRP: [12,438; 0; -70,465]	$6.168 \cdot 10^{-8}$	$7.855 \cdot 10^{-21}$

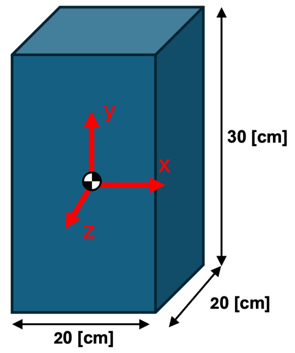
Table 4.2: Maximum disturbance torque magnitude values for the LUMIO spacecraft located in extreme-case positions, for the solar radiation pressure and the gravity gradient torque, along with position coordinates in a Moon-centred reference frame.

From Table 4.2, it can be concluded that the CAPSTONE orbit does indeed approach the Moon closer than the LUMIO orbit, which is directly translated in the disturbance torque due to the gravity gradient. For this reason, the CAPSTONE orbit can be considered as a more extreme test case. This is further strengthened by the fact that close approaches require quicker attitude adjustments compared to orbits that remain at further distance. In addition, the solar radiation pressure values at their predicted maxima are only slightly different and their order of magnitude will not have an effect on the control simulation results.

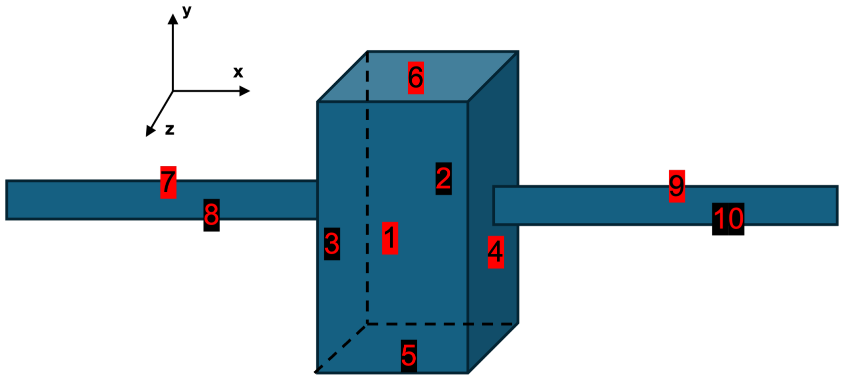
Finally, as mentioned previously, the CAPSTONE orbit allows for easy retrieval of ephemeris data without the need of extensive orbit propagation. In this way, the main focus of the research can be retained.

4.2.2. Spacecraft

The LUMIO spacecraft design has been shown in Figure 2.13a previously. In order to simplify the analysis in the attitude control simulation, a number of simplifications will be made. First of all, the LUMIO spacecraft will be a 12U CubeSat with sides of L$\times$W$\times$H equal to 20$\times$20$\times$30 $[cm]$. Moreover, the physical positioning solar arrays will not be considered in the control system design of the spacecraft, with the simple reason that this allows for more freedom in the thruster configuration design. In contrast to the body axis definition as was proposed in Figure 2.13b, this research will adhere to the body axis definition as shown in Figure 4.2a. The origin of the body reference frame coincides with the centre of mass of the spacecraft, which is assumed to coincide with the geometrical centre as well. Furthermore, the Cartesian axes as displayed in the figure coincide with the primary axes of rotation, assigning the roll axis to be equal to rotation over the x-axis, the pitch angle to be equal to the y-axis and the yaw angle to be equal to the z-axis. Roll angle will be denoted by ϕ, pitch by θ and yaw by ψ. The solar arrays will be positioned along the negative and positive x-axes, attached at locations [10, 0, 0] $[cm]$ and [-10, 0, 0] $[cm]$. The LUMIO-Cam will be positioned in the centre of the negative y-panel (location [0, -15, 0] $[cm]$), which means the attitude control algorithm should in any case point the negative y-panel towards the Moon.



(a) Simplified spacecraft bus, including its dimensions and body-centred reference frame to be used in the research.



(b) Panel numbering as used throughout the attitude control simulation.

Figure 4.2: (a) Simplified spacecraft bus with body-centred reference frame. (b) Panel numbering for attitude control simulation.

An important characteristic for the spacecraft bus to be used within the attitude control simulation, is its inertia matrix (mass moment of inertia). Two different matrices can be distinguished, the deployed solar array matrix and the undeployed solar array matrix, denoted $I_{deployed}$ and $I_{undeployed}$, respectively. Their values, as taken from Romero-Calvo, Biggs, and Topputo [43], are represented in Equation 4.1. The larger values in the deployed state were to be expected and are the result of the definition of the mass moment of inertia; the distribution of mass with respect to the centre of mass of the object.

$$
\begin{aligned}
I_{\text{deployed}} &= \begin{bmatrix} 100.9 & 0 & 0 \\ 0 & 25.1 & 0 \\ 0 & 0 & 91.6 \end{bmatrix} \cdot 10^{-2}\,[\text{kg m}^2] \\
I_{\text{undeployed}} &= \begin{bmatrix} 30.5 & 0 & 0 \\ 0 & 20.9 & 0 \\ 0 & 0 & 27.1 \end{bmatrix} \cdot 10^{-2}\,[\text{kg m}^2]
\end{aligned} \tag{4.1}
$$

For the analysis of the solar radiation pressure throughout the control simulation, the centre of pressure and centre of mass of all the panels of the spacecraft are assumed to be located in their geometrical centres as well. In order to be able to compute the solar radiation pressure on the spacecraft at any given moment in time, the matrices $\boldsymbol{S}$, $\boldsymbol{n}_s$, $\boldsymbol{A}_{panels}$ and $\boldsymbol{c}_p$ should be constructed for the simplified spacecraft architecture. Before construction of these matrices is possible, it should be clear which spacecraft panel is which, and a dedicated numbering system should be put in place. This numbering system is shown in Figure 4.2b, in which the black numbers with a red background are associated with the front panels, and the red numbers with a black background are associated with the backward panels, which one cannot directly see from this angle. The solar panels are associated with panel 7 through 10.

With these panels defined, the position vectors of the centres of pressure for the spacecraft panels can be constructed and column stacked in $\boldsymbol{c}_p$. Note that these vectors are all defined in the body frame, as is required for calculation of the solar radiation pressure. The matrix is shown in Equation 4.2. From the matrix, it can be concluded that the solar panels have a total length of 70 $[cm]$. Each column of the matrix adheres to the location of the geometric centre of the panel equal to the column number in Figure 4.2b.

$$
\boldsymbol{c}_p = \begin{bmatrix} 0 & 0 & -0.1 & 0.1 & 0 & 0 & -0.45 & -0.45 & 0.45 & 0.45 \\ 0 & 0 & 0 & 0 & -0.15 & 0.15 & 0 & 0 & 0 & 0 \\ 0.1 & -0.1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad [\text{m}] \tag{4.2}
$$

Next, the matrix containing all the panel surface areas can be constructed as seen in Equation 4.3. Also, the matrix containing the unit vectors normal to the surfaces of the panel and pointing towards the interior is presented in Equation 4.4. Note that α is the optimal angle for the solar arrays to be pointed towards the Sun.

$$
A_{panels} = \begin{bmatrix} 6 & 6 & 6 & 6 & 4 & 4 & 12 & 12 & 12 & 12 \end{bmatrix} \cdot 10^{-2}\,[\text{m}^2] \tag{4.3}
$$

$$
\boldsymbol{n}_s(\alpha) = \begin{bmatrix} 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & \sin(\alpha) & -\sin(\alpha) & \sin(\alpha) & -\sin(\alpha) \\ -1 & 1 & 0 & 0 & 0 & 0 & \cos(\alpha) & -\cos(\alpha) & \cos(\alpha) & -\cos(\alpha) \end{bmatrix} \tag{4.4}
$$

The matrix $\boldsymbol{S}$ can be constructed to represent the unit vectors pointing from the Sun to the surface of each panel in the spacecraft body frame. This matrix is derived by subtracting the vector $\boldsymbol{r}_{\text{S/SC}}$, which is the coordinate vector of the Sun with respect to the spacecraft, from the position vectors of the centres of pressure in $\boldsymbol{c}_p$. Each column of the resulting matrix is normalised to ensure that it is a unit vector, as shown in Equation 4.5. Note that all vectors are defined in the body frame again.

$$
\boldsymbol{S} = \begin{bmatrix} \frac{\boldsymbol{c}_{p1} - \boldsymbol{r}_{\text{S/SC}}}{\|\boldsymbol{c}_{p1} - \boldsymbol{r}_{\text{S/SC}}\|} & \frac{\boldsymbol{c}_{p2} - \boldsymbol{r}_{\text{S/SC}}}{\|\boldsymbol{c}_{p2} - \boldsymbol{r}_{\text{S/SC}}\|} & \cdots & \frac{\boldsymbol{c}_{p10} - \boldsymbol{r}_{\text{S/SC}}}{\|\boldsymbol{c}_{p10} - \boldsymbol{r}_{\text{S/SC}}\|} \end{bmatrix} \tag{4.5}
$$

Here, $\boldsymbol{c}_{p_i}$ represents the i-th column of $\boldsymbol{c}_p$, corresponding to the centre of pressure for the i-th panel. The resulting S_{inertial} matrix is a 3×10 matrix, where each column is a unit vector pointing from the Sun to the corresponding panel in the body frame. This matrix is crucial for determining the direction of solar radiation pressure on each panel. Finally, the values for the specularly and diffusely reflected radiation, ρ_s and ρ_d, will be equal to 0.6 and 0.1, similar to the values used in Romero-Calvo, Biggs, and Topputo [43].

In order to create a benchmark case within the research to electrical thruster configurations, the original reaction wheel set-up as proposed in the LUMIO ADCS will be used. In this set-up, three reaction wheels (RWp100) with a total angular momentum capacity of 100 $[mNms]$ and a maximum torque of 0.007 $[Nm]$ are aligned with the spacecraft primary axes, so that each of them is responsible for either the roll, pitch or yaw angle. In addition, a slightly smaller reaction wheel (RWp050) with the same maximum torque but an angular momentum capacity of 50 $[mNms]$ is placed so that it contributes towards the torque over all three axes equally. The power consumption of both types of reaction wheels is similar and equal to $9[W]$ at peak operation, meaning at maximum torque. [49] The configuration matrix for the reaction wheel set-up in the original LUMIO ADCS is the same as seen in Equation 2.42 and will be used for the reaction wheel analysis. It is displayed in Equation 4.6 again for the reader's convenience. Moreover, due to the addition of reaction wheels, the increase in total angular momentum induces a gyroscopic term that should be included in the analysis, and creates an adjustment in the Euler equations for rotational motion of a rigid body. This adjustment in the equations of motion was previously shown in Equation 2.39.

$$A_{rw,LUMIO} = \begin{bmatrix} 1 & 0 & 0 & \frac{1}{\sqrt{3}} \\ 0 & 1 & 0 & \frac{1}{\sqrt{3}} \\ 0 & 0 & 1 & \frac{1}{\sqrt{3}} \end{bmatrix} \tag{4.6}$$

4.2.3. Thrusters

The main component of this research is the assessment of the electrical thruster configurations. An infinite number of configurations is possible, adjusting the number of thrusters, their locations, and their angular offset with respect to one of the primary axes. In order to keep the scope of this research within justifiable limits, only four different set-ups will be analysed throughout this work. In addition, since the aforementioned linear programming approach (see Equation 2.57) minimises the total thrust output subject to constraints ensuring the required control torque is met, it inherently seeks solutions that optimise efficiency.

However, for configurations with exactly four thrusters, the system becomes underdetermined when attempting to produce torques along all three axes simultaneously. This is due to the following key constraints:

1. **Geometric placement constraints:** The positions and orientations of the thrusters relative to the spacecraft centre of mass determine the torques they can generate. With only four thrusters, the geometric arrangement often leads to linear dependence in the generated torque vectors, making it impossible to create independent torques along all three axes.
2. **Thrust magnitude constraints:** Each thruster has physical limits on the minimum and maximum thrust it can produce. These limits restrict the solver's ability to find feasible combinations of thrust magnitudes to satisfy the desired torques.
3. **Operational constraints:** Thrusters typically provide unidirectional thrust, meaning they can only push in one direction. This reduces the feasible solution space since reverse thrust would require repositioning or additional thrusters.
4. **Coupled torque and force constraints:** In a 4-thruster system, any thrust configuration designed to produce torque also generates residual forces. These forces can conflict with the requirement to maintain translational equilibrium (no net force), creating additional constraints that may be impossible to satisfy simultaneously.

These constraints collectively restrict the feasible solution space required for the linear programming solver. With only four control inputs (thrusters), the system lacks sufficient degrees of freedom to resolve these conflicts while independently controlling torque in three dimensions. As a result, the solver often fails to find valid solutions for certain required torque combinations.

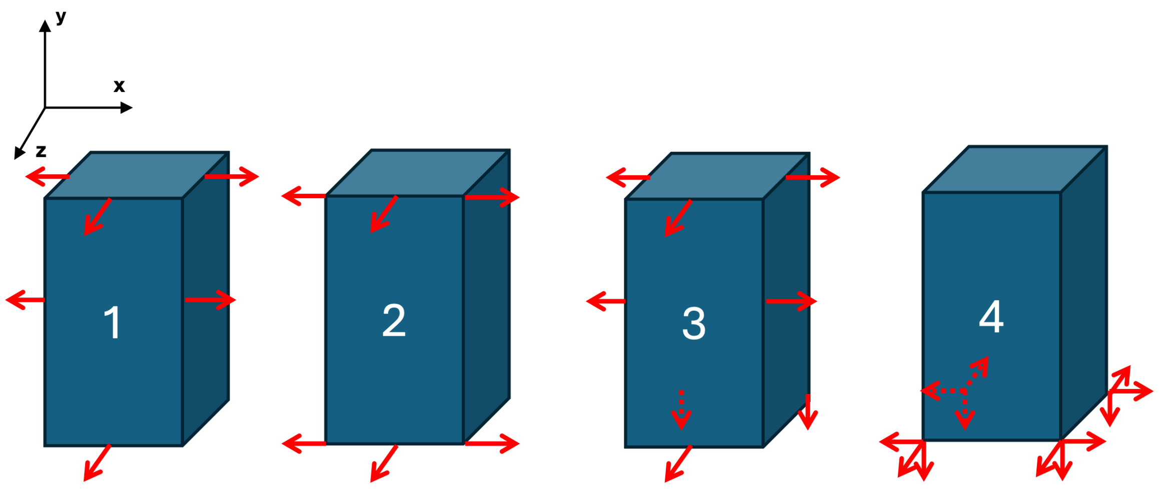


Figure 4.3: Four different thruster configurations used throughout this research, within the body axes as defined previously.

For this reason, the minimum number of thrusters used is six in this research. In Figure 4.3, the four configurations are shown, and a brief explanation about each of them follows below, along with the reasoning behind choosing this specific configuration.

1. For three-axis spacecraft control, this configuration is determinate. As can be seen from the figure, each pair of thrusters is responsible for rotation about only one axis. Since the spacecraft has six degrees of freedom (translation in any x, y and z direction, and rotation about any of the primary axes), and each thruster gives a single scalar control variable (thrust magnitude), this problem is determinate.
2. This configuration also employs six thrusters in total, although each pair does not contribute to the rotation of only one axis. The two thrusters pointed into the positive z-direction will give rotation over the roll angle, whereas the other four thrusters are responsible for the pitch angle as well as the yaw angle. For the analysis at hand, a performance comparison with the first set-up is interesting to assess, since the effect of axis coupling can be quantified by this. Similar to the first set-up, this problem is determinate.
3. The third set-up is overdetermined, since it employs the exact same set-up as configuration 1 but has added two thrusters on the bottom, expelling particles in the negative y-direction. These thrusters can create torque around the x-axis as well as z-axis. The reason for selecting this configuration is also the comparison with configuration 1; it can be assessed whether the addition of more thrusters leads to improvement in pointing accuracy and could lead to a reduction in required power.
4. Configuration 4 is also overdetermined and employs 12 thrusters in total. Each of the bottom (negative y) corners consists of a thruster module with three orthogonal thrusters. The reason for inclusion of this configuration is a redundancy in the number of hardware modules used and the modularity of the thruster systems. Also, the power requirement and pointing accuracy with such an overdetermined system is relevant to analyse.

The mixing matrix for each of these thrusters, $A_{thrust,1}$ to $A_{thrust,4}$, can now be constructed using the geometries presented in Figure 2.13b. Adhering to the definition of Equation 2.55, the unit torque vector of each thruster has to be computed first. In Table 4.3 to Table 4.6, the components for the cross product of each thruster in the four configurations are presented. In addition, these tables serve to present the thruster numbering scheme used. Appending these cross products to the columns of the mixing matrices results in Equation 4.7 to Equation 4.10.

Thruster $\hat{T_i}$	Location Vector [x, y, z] [m]	Thrust Direction [x, y, z]
$\hat{T_1}$	[0.1, 0.15, 0]	[-1, 0, 0]
$\hat{T_2}$	[-0.1, 0.15, 0]	[1, 0, 0]
$\hat{T_3}$	[0, 0.15, 0.1]	[0, 0, -1]
$\hat{T_4}$	[0, -0.15, 0.1]	[0, 0, -1]
$\hat{T_5}$	[0.1, 0, 0.1]	[-1, 0, 0]
$\hat{T_6}$	[-0.1, 0, 0.1]	[1, 0, 0]

Table 4.3: Unit torque cross product components for thruster configuration 1.

Thruster $\hat{T_i}$	Location Vector [x, y, z] [m]	Thrust Direction [x, y, z]
$\hat{T_1}$	[0.1, 0.15, 0.1]	[-1, 0, 0]
$\hat{T_2}$	[-0.1, 0.15, 0.1]	[1, 0, 0]
$\hat{T_3}$	[0, 0.15, 0.1]	[0, 0, -1]
$\hat{T_4}$	[0, -0.15, 0.1]	[0, 0, -1]
$\hat{T_5}$	[0.1, -0.15, 0.1]	[-1, 0, 0]
$\hat{T_6}$	[-0.1, -0.15, 0.1]	[1, 0, 0]

Table 4.4: Unit torque cross product components for thruster configuration 2.

Thruster $\hat{T_i}$	Location Vector [x, y, z] [m]	Thrust Direction [x, y, z]
$\hat{T_1}$	[0.1, 0.15, 0]	[-1, 0, 0]
$\hat{T_2}$	[-0.1, 0.15, 0]	[1, 0, 0]
$\hat{T_3}$	[0, 0.15, 0.1]	[0, 0, -1]
$\hat{T_4}$	[0, -0.15, 0.1]	[0, 0, -1]
$\hat{T_5}$	[0.1, 0, 0.1]	[-1, 0, 0]
$\hat{T_6}$	[-0.1, 0, 0.1]	[1, 0, 0]
$\hat{T_7}$	[0.1, -0.15, -0.1]	[0, 1, 0]
$\hat{T_8}$	[-0.1, -0.15, -0.1]	[0, 1, 0]

Table 4.5: Unit torque cross product components for thruster configuration 3.

Thruster $\hat{T}_i$	Location Vector [x, y, z] [m]	Thrust Direction [x, y, z]
$\hat{T}_1$	[0.1, -0.15, 0.1]	[0, 1, 0]
$\hat{T}_2$	[0.1, -0.15, 0.1]	[-1, 0, 0]
$\hat{T}_3$	[0.1, -0.15, 0.1]	[0, 0, -1]
$\hat{T}_4$	[-0.1, -0.15, 0.1]	[1, 0, 0]
$\hat{T}_5$	[-0.1, -0.15, 0.1]	[0, 1, 0]
$\hat{T}_6$	[-0.1, -0.15, 0.1]	[0, 0, -1]
$\hat{T}_7$	[0.1, -0.15, -0.1]	[-1, 0, 0]
$\hat{T}_8$	[0.1, -0.15, -0.1]	[0, 1, 0]
$\hat{T}_9$	[0.1, -0.15, -0.1]	[0, 0, 1]
$\hat{T}_{10}$	[-0.1, -0.15, -0.1]	[0, 0, 1]
$\hat{T}_{11}$	[-0.1, -0.15, -0.1]	[0, 1, 0]
$\hat{T}_{12}$	[-0.1, -0.15, -0.1]	[1, 0, 0]

Table 4.6: Unit torque cross product components for thruster configuration 4.

$$A_{thrust,1} = \begin{bmatrix} 0 & 0 & -0.15 & 0.15 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.1 & 0.1 \\ 0.15 & -0.15 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.7}$$

$$A_{thrust,2} = \begin{bmatrix} 0 & 0 & -0.15 & 0.15 & -0 & -0 \\ -0.1 & 0.1 & 0 & 0 & -0.1 & 0.1 \\ 0.15 & -0.15 & 0 & 0 & -0.15 & 0.15 \end{bmatrix} \tag{4.8}$$

$$A_{thrust,3} = \begin{bmatrix} 0 & 0 & -0.15 & 0.15 & 0 & 0 & 0.1 & 0.1 \\ 0 & 0 & 0 & 0 & -0.1 & 0.1 & 0 & 0 \\ 0.15 & -0.15 & 0 & 0 & 0 & 0 & 0.1 & -0.1 \end{bmatrix} \tag{4.9}$$

$$A_{thrust,4} = \begin{bmatrix} -0.1 & 0 & 0.15 & 0 & -0.1 & 0.15 & 0 & 0.1 & -0.15 & -0.15 & 0.1 & 0 \\ 0 & -0.1 & 0.1 & 0.1 & 0 & -0.1 & 0.1 & 0 & -0.1 & 0.1 & 0 & -0.1 \\ 0.1 & -0.15 & 0 & 0.15 & -0.1 & 0 & -0.15 & 0.1 & 0 & 0 & -0.1 & 0.15 \end{bmatrix} \tag{4.10}$$

In addition to the minimum thrust requirement of 0 [N] for the linear programming solution, a maximum thrust value should also be in place in order to effectively perform the analysis for each of the thruster configurations. This maximum thrust is determined by the hardware module used, and it was decided to use the Pocket Rocket as developed by the company Solid State Propulsion from Pretoria, South Africa. The reason for this decision is the collaboration established during the execution of this research and the fact that the company produces vacuum arc thrusters, which are the ideal thrusters to use for attitude control problems. As was discussed in Section 2.6, the frequent pulses given by VATs can easily be commanded by signals at the same frequency, and the addition of pulse-width modulation makes the thrust variation for precise control possible.

As observed in (SSP) [1], the maximum output thrust of the Pocket Rocket varies with the input power supplied to the unit, and there is a linear relation between these two parameters as observed in Figure 4.4. Other specifications for the Pocket Rocket can be found in Table 4.7. Each thruster that is used in the aforementioned configurations will be a Pocket Rocket, and the power setting for each configuration will be determined based on the results from the attitude control simulation so that a power-optimal approach is adhered to. At first, however, the maximum thrust output will be allowed to assess whether this module is possible for usage within the attitude control simulation. Moreover, the Minimum Impulse Bit as shown will not be included at first, in order to give a level of flexibility on the problem boundaries. After the first experiments, adjustments in the code based on the power input and MIB will be made.

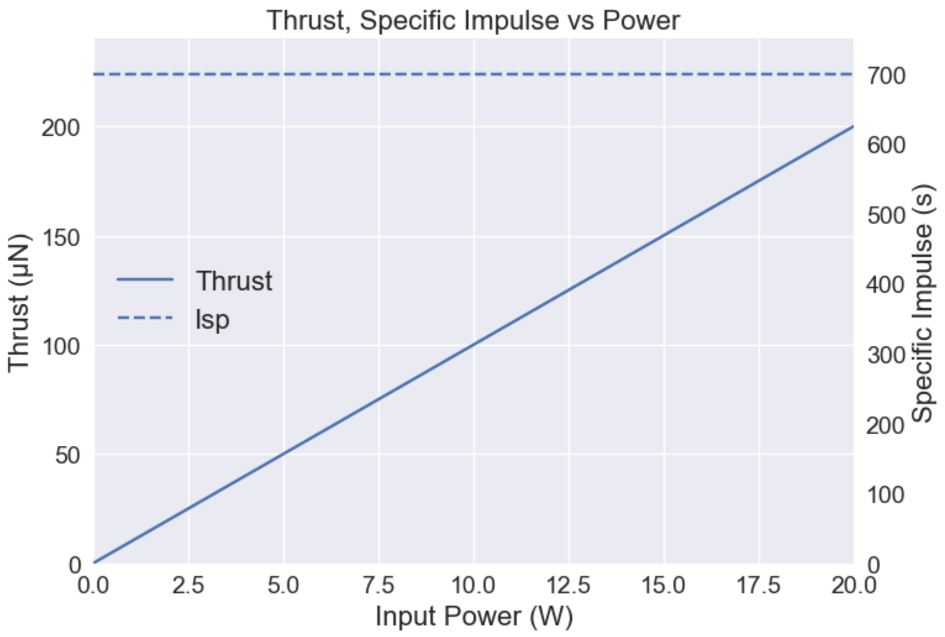


Figure 4.4: SSL Pocket Rocket thrust versus input power relation, including the invariant specific impulse of the vacuum arc thruster module. Adapted from (SSP) [1].

Specification	Value
Voltage input	5V Power / 3.3V Logic
Size	4.5×4.5×2.5 $[cm]$
Pulse frequency	0.1 - 10 $[Hz]$
Input power	0.5 - 20 $[W]$
Thrust/Power ratio	10 $[\mu N/W]$
Maximum thrust	200 $[\mu N]$
Wet mass	85 $[g]$
Total impulse	8 $[Ns]$
Minimum Impulse Bit	1.18 $[\mu Ns]$

Table 4.7: SSL Pocket Rocket specifications, adapted from (SSP) [1].

4.3. Assumptions & Considerations

As mentioned in the introduction of this chapter, the general approach towards the assessment of the electrical thruster configurations is the creation of an attitude control algorithm from scratch. While professional software such as the Ansys Systems Tool Kit[3] exists to accomplish similar goals, this study requires a deeper understanding of the underlying dynamics and the ability to tailor the control system to specific research needs. Developing the simulation from the ground up provides the following advantages:

[3]URL: `https://www.ansys.com/products/missions/ansys-stk` [Accessed 8 December 2024]

- **Comprehensive understanding**: By implementing the dynamics and control equations manually, the relationships between the system's physical parameters and its behavior can be fully explored, providing a deeper understanding of the fundamental principles of attitude control.
- **Customisation**: Existing software often comes with predefined frameworks that may not accommodate unconventional or experimental configurations. A custom-built simulation allows for full flexibility in testing various thruster configurations and exploring unique scenarios.
- **Optimisation and Integration**: Tailoring the simulation in Python enables seamless integration with other tools, such as optimisation libraries or specific numerical solvers. This flexibility is critical for iterating designs and fine-tuning performance metrics.
- **Cost-Effectiveness**: Utilising Python and open-source packages avoids the licensing costs associated with professional tools, making this approach more accessible and budget-friendly.
- **Documentation and Reproducibility**: A self-developed simulation ensures clear documentation of all implemented steps, enhancing the reproducibility and transparency of the research.

Within the attitude control algorithm, use will be made of quaternions to propagate the spacecraft state, using the state as described in Equation 4.11 which will be propagated over the desired time span. Note that the angular rates for each axis is now denoted by the Cartesian axis, x, y or z. A number of reasons can be given for using quaternions over Euler angles, but the main reasons are the avoidance of gimbal lock in Euler angles (singularity due to alignment of two of the three rotation axes) and the numerical stability of quaternions. Moreover, the convention used for expressing quaternions in this research is [q_w, q_1, q_2, q_3], in which the first term denotes the scalar part and the other three the vector part.

$$\boldsymbol{x}_{LUMIO} = \begin{bmatrix} q_w & q_1 & q_2 & q_3 & \omega_x & \omega_y & \omega_z \end{bmatrix}^T \tag{4.11}$$

The propagation of the spacecraft state is achieved by integrating the spacecraft's dynamic equations of rotational motion. These equations of motion were observed in subsection 2.3.2 and will be adapted for this research as well. It is crucial to understand its underlying assumptions, before applying them to the LUMIO mission. These are:

1. The spacecraft bus is rigid, meaning that all particles belonging to the body are fixed relative to each other and no elasticity takes place.
2. A body-fixed reference frame is adhered to, with its origin in the centre of mass of the spacecraft. This body-fixed coordinate system rotates with the spacecraft body and therefore has a rotational rate with respect to an inertial Newtonian frame.
3. The moment of inertia matrix of the spacecraft bus remains constant over the simulation timespan.
4. Torques, either internal or external, are summed around the spacecraft centre of mass, which coincides with the origin of the body-fixed reference frame.
5. The inertial frame within this analysis has its origin in the centre of mass of the Moon. The frame's axes are aligned with the International Celestial Reference Frame, which is a standard frame defined by the positions of distant quasars.

The torques that are part of Euler's equation are the control torque exerted by the ADCS, the gravity gradient torque and the solar radiation pressure exerted on the spacecraft. Other disturbance torques do not have a significant influence to be included in this research. The gravity gradient torque is modelled as Equation 2.32 and the solar radiation pressure as Equation 2.34 to Equation 2.36. The control torque at each time step is determined using a PD control algorithm. The choice for a PD controller was made primarily for the simplicity of calculation; it only requires the quaternion error of the current state and the current angular rate of the body. Gain tuning needs to be performed to obtain the optimal response, applied to the proportional (k_p), derivative (k_d) and speed (k_s) gains. The integral term of the control logic was left out, since this gave instability of the control algorithm upon testing due to integral wind-up, meaning that errors were accumulated excessively over time.

For each axis (x, y, z), the control torque is given as shown in Equation 4.12. As can be seen, the proportional gain amplifies each of the vector parts of the quaternion error (ergo: q_{e1}, q_{e2} and q_{e3}). The quaternion errors are defined as seen in Equation 2.31. Moreover, the derivative gain amplifies the current angular rate around each of the spacecraft body axes (ω_x, ω_y or ω_z). Finally, the speed gain amplifies the entire control logic for faster convergence. Note that each gain is defined to have components along a specific axis.

$$T_{ci} = k_{si}\left(k_{pi} q_{ei} - k_{di}\omega_i\right) \tag{4.12}$$

Although manual gain tuning will be executed to find the desired gains, a first estimation for k_{pi} and k_{di} can be made based on the angular natural rate (ω_n) and the damping ratio (ζ) selected. Note that the damping ratio will create an under-damped controller with oscillations for 0 < ζ < 1, and an over-damped controller for $\zeta >$ 0. The critically damped system ($\zeta = 1$) would be the best option considering overshoot and rise time of the response. These first gain estimations can be calculated as shown in Equation 4.13. The gain values will be equal for each axis and therefore $\boldsymbol{K}_d = k_{di}\boldsymbol{I}$ and $\boldsymbol{K}_p = k_{pi}\boldsymbol{I}$.

$$\begin{aligned} k_{di} &= 2\zeta\omega_n \\ k_{pi} &= 2\omega_n^2 \\ \omega_n &= \sqrt{\frac{T_{ci}}{I_{ii}}} \end{aligned} \tag{4.13}$$

The desired state of LUMIO should be included in the simulation in order to compute the quaternion errors at each time step throughout the integration. As observed in Romero-Calvo, Biggs, and Topputo [43], the desired direction cosine matrix is dependent on the position of the Sun and the Moon with respect to the LUMIO spacecraft. More specifically, the normalised Moon pointing vector as shown in Equation 4.14 and the normalised Sun pointing vector as shown in Equation 4.15 are retrieved from the position of the Sun with respect to the Moon ($\boldsymbol{r}_{\text{S/M}}$) and the position of the spacecraft with respect to the Moon ($\boldsymbol{r}_{\text{SC/M}}$).

$$\hat{\boldsymbol{r}}_{\text{M/SC}} = \frac{-\boldsymbol{r}_{\text{SC/M}}}{\| \boldsymbol{r}_{\text{SC/M}} \|} \tag{4.14}$$

$$\hat{\boldsymbol{r}}_{\text{S/SC}} = \frac{\boldsymbol{r}_{\text{S/M}} - \boldsymbol{r}_{\text{SC/M}}}{\| \boldsymbol{r}_{\text{S/M}} - \boldsymbol{r}_{\text{SC/M}} \|} \tag{4.15}$$

The desired DCM can be constructed as shown in Equation 4.16, which is a rotation matrix between the inertial Moon frame and the spacecraft body-fixed reference frame. This DCM optimises the power generation by the solar arrays and also makes sure the LUMIO-Cam is always pointed towards the Moon. The first column ensures the power generation criterion, the second column ensures the Moon pointing with the negative y-panel, the third column completes the desired attitude by complementing a right-hand coordinate system. This direction cosine matrix can now be converted to the required quaternion vector, $\boldsymbol{q}_{ref}$, by means of Equation 2.22, Equation 2.23, Equation 2.24 and Equation 2.25.

$$C_{desired} = \left[\boldsymbol{c}_1 = \frac{\hat{\boldsymbol{r}}_{\text{S/SC}} \times \hat{\boldsymbol{r}}_{\text{M/SC}}}{\| \hat{\boldsymbol{r}}_{\text{S/SC}} \times \hat{\boldsymbol{r}}_{\text{M/SC}} \|}, \boldsymbol{c}_2 = -\hat{\boldsymbol{r}}_{\text{M/SC}}, \boldsymbol{c}_3 = \frac{\boldsymbol{c}_2 \times \boldsymbol{c}_1}{\| \boldsymbol{c}_2 \times \boldsymbol{c}_1 \|}\right] \tag{4.16}$$

The integration time step within the control simulation has been set to 1 [s], meaning the frequency is equal to 1 [Hz]. The Nyquist criterion suggests that reconstruction in an attitude control problem is accurate when the sampling frequency is at least twice the highest frequency in the dynamics to be sampled. To ensure the Nyquist criterion is adhered to, the highest frequency in the spacecraft's rotational dynamics must be analysed. For this system, the highest frequency is determined by the natural frequencies of the rigid body equations of motion, which depend on the applied torques and the spacecraft's moments of inertia. The relation shown in Equation 4.17 can be used to assess the natural frequency of each rotational axis, obtained from Franklin, Powell, and Emami-Naeini [16].

$$f_{natural,i} = \frac{1}{2\pi}\sqrt{\frac{T_{ci}}{I_{ii}}}, \tag{4.17}$$

In the equation, T_{ci} represents the maximum applied torque around one the principal axes and I_{ii} the moment of inertia about one of the principal axes. Since the order of magnitude of the total control torque is approximately 10^{-7} [Nm] and the moment of inertia values are in the order of magnitude of 10^{-2} [kgm^2], the natural frequencies for this spacecraft are estimated to be below 0.1 Hz. This is far below the Nyquist limit of $f_s/2 = 0.5$ Hz, ensuring that the 1-second time step provides accurate reconstruction of the dynamics. This low natural frequency arises from the large moments of inertia typical of spacecraft and the damping effects of the control system, which suppress high-frequency dynamics. Furthermore, external disturbance torques, such as those from solar radiation pressure and the gravity gradient, are generally small, contributing to slower rotational dynamics. Therefore, the chosen 1-second time step is sufficient for accurately capturing the spacecraft's attitude behaviour while adhering to the Nyquist criterion.

The integrator used for the simulation will be the Runge-Kutta-4 integrator as described in subsection 2.3.5. The reason for this selection is its relative simplicity compared to other more advanced integrators like variable step-size and multi-step methods. Moreover, it is more accurate than the standard Euler integrator, which is also included in the simulation for comparison. The updates of the quaternion part of the spacecraft state depends on the time derivative of the quaternion state at each moment in time, denoted by $\dot{\boldsymbol{q}}$, and shown in Equation 2.20. The time derivative of the angular rate has been presented in Equation 2.4 and is used in the RK4 integrator for integrating the angular velocity around each axis of the spacecraft body. The integrator logic for the quaternion update becomes as presented in Equation 4.18, with the updated quaternion state as computed in Equation 4.19.

$$\begin{aligned} k_{q1} &= \Delta t\, \dot{\boldsymbol{q}}\big(\boldsymbol{q}(t), \boldsymbol{\omega}(t)\big), \\ k_{q2} &= \Delta t\, \dot{\boldsymbol{q}}\big(\boldsymbol{q}(t) + \frac{1}{2}k_{q1}, \boldsymbol{\omega}(t + \frac{1}{2}\Delta t)\big), \\ k_{q3} &= \Delta t\, \dot{\boldsymbol{q}}\big(\boldsymbol{q}(t) + \frac{1}{2}k_{q2}, \boldsymbol{\omega}(t + \frac{1}{2}\Delta t)\big), \\ k_{q4} &= \Delta t\, \dot{\boldsymbol{q}}\big(\boldsymbol{q}(t) + k_{q3}, \boldsymbol{\omega}(t + \Delta t)\big). \end{aligned} \tag{4.18}$$

$$\boldsymbol{q}(t+\Delta t) = \boldsymbol{q}(t) + \frac{1}{6}(k_{q1} + 2k_{q2} + 2k_{q3} + k_{q4}). \tag{4.19}$$

For the angular rate, the same logic is presented in Equation 4.20 and Equation 4.21.

$$\begin{aligned} k_{\omega1} &= \Delta t\, \dot{\boldsymbol{\omega}}\big(\boldsymbol{\omega}(t), \boldsymbol{T}_c(t)\big), \\ k_{\omega2} &= \Delta t\, \dot{\boldsymbol{\omega}}\big(\boldsymbol{\omega}(t) + \frac{1}{2}k_{\omega1}, \boldsymbol{T}_c(t + \frac{1}{2}\Delta t)\big), \\ k_{\omega3} &= \Delta t\, \dot{\boldsymbol{\omega}}\big(\boldsymbol{\omega}(t) + \frac{1}{2}k_{\omega2}, \boldsymbol{T}_c(t + \frac{1}{2}\Delta t)\big), \\ k_{\omega4} &= \Delta t\, \dot{\boldsymbol{\omega}}\big(\boldsymbol{\omega}(t) + k_{\omega3}, \boldsymbol{T}_c(t + \Delta t)\big). \end{aligned} \tag{4.20}$$

$$\boldsymbol{\omega}(t+\Delta t) = \boldsymbol{\omega}(t) + \frac{1}{6}(k_{\omega1} + 2k_{\omega2} + 2k_{\omega3} + k_{\omega4}). \tag{4.21}$$

Now that the integrator set-up has been elaborated upon, more detailed simulation considerations can be outlined. First of all, Table 4.8 shows a table with constants used throughout the simulation, along with their symbols in this research and units. In addition, it should be clear that the base case for the simulation is the deployed solar array state of the spacecraft, and the inertia matrix $I_{deployed}$ shown in Equation 4.1 will be adhered to for the simulation. Additionally, it is assumed that the reaction wheels work perfectly (100% efficiency) and are able to store all required angular momentum at any point in time. Wheel saturation will be studied in the benchmark scenario.

Description	Symbol	Value	Units
Speed of light	c	$2.99792458 \cdot 10^{8}$	m/s
Universal Gravitational Constant	G	$6.67428 \cdot 10^{-11}$	$\text{m}^3\text{s}^{-2}\text{kg}^{-1}$
Gravitational parameter of Earth	μ_{Earth}	$3.986004418 \cdot 10^{14}$	m^3s^{-2}
Gravitational parameter of Moon	μ_{Moon}	$4.9048695 \cdot 10^{12}$	m^3s^{-2}
Gravitational parameter of Sun	μ_{Sun}	$1.32712440018 \cdot 10^{20}$	m^3s^{-2}
Astronomical unit	AU	$1.495978707 \cdot 10^{11}$	m
Earth mean radius	R_{Earth}	$6.3781 \cdot 10^{6}$	m
Moon mean radius	R_{Moon}	$1.7374 \cdot 10^{6}$	m
Sun mean radius	R_{Sun}	$6.957 \cdot 10^{8}$	m
Power exerted by the Sun	P_{solar}	$3.842 \cdot 10^{26}$	W

Table 4.8: Physical constants used in the simulation

The output thrust of the thrusters in this analysis will not be modelled as ideal, as real hardware systems inherently exhibit uncertainty in their outputs. This uncertainty is modelled using a Gaussian (normal) distribution, characterised by its mean, representing the desired thrust value, and its standard deviation, σ. For this research, a standard deviation of 5% of the mean thrust value will be assumed, meaning that thrust outputs are expected to deviate within one σ of the mean approximately 68% of the time, as per the properties of a normal distribution. With the calculated thruster outputs, the linear impulse exerted by them can be calculated as observed in Equation 4.22. This linear impulse will be used to give an initial indication of the power and energy requirements of the configurations.

$$J = \int F_{thrust}(t)\, dt \tag{4.22}$$

The power and energy consumption of the hardware will be based on the assumed linear relation between the power input and the output thrust or torque. For thrusters, this is validated by Figure 4.4. For reaction wheels, a mathematical approach can be followed. Torque is defined as the change of angular momentum over time. Angular momentum is defined as the mass moment of inertia of an object times its angular velocity. Power is needed for a change in angular velocity, inducing a change in angular momentum and therefore torque. Following this line of thought, it is assumed there exists a linear relation between power input and torque. For the energy consumed, it will simply be seen as the power function integrated over time for the simulation duration, and therefore accumulate over time.

As a final consideration, and looking at LUMIO requirement **ADCS-03**, the half-cone offset angle of the spacecraft should be assessed throughout the simulation. This offset angle is defined as shown in Equation 4.23, which computes the angle between the unit vector pointing from the spacecraft body frame origin towards the panel located at [0, -0.15, 0] and the Moon pointing vector, which points from the spacecraft body frame origin towards the centre of the Moon.

$$\beta = \cos^{-1}\left(\hat{\boldsymbol{x}}_{y,panel} \cdot \hat{\boldsymbol{r}}_{\text{Moon/SC}}\right) \tag{4.23}$$

4.4. Code

With the research context, its underlying assumptions and further considerations for simulation development, the code for the spacecraft attitude control simulation could be written. The main building blocks of the code have all been created in a object-oriented programming approach; classes were defined for the categories present in the code, which are:

- `Constants`: Includes all constants used throughout the simulation, as shown in Table 4.8.
- `Rotation`: Mathematical operations for rotations between reference frames.
- `DisturbanceTorques`: SRP and GG disturbance torques.
- `EphemerisData`: Retrieval of ephemeris data from JPL Horizons.

- `PID`: Relevant control law functions, among which $\dot{\omega}$ and $\dot{q}$ calculations.
- `Visualisation`: Visualisation tools based on `matplotlib` library.

These classes were all written in a Python file named `classes.py`. In addition, Jupyter notebook files were created for code testing and simulation execution. In the file `simulation.ipynb`, a main simulation function was created that executes the simulation with a number of input parameters. In this way, the code has been written in a modular and logical way. The code is presented in Appendix B and can be viewed in the online repository on `https://github.com/Pieter1999/lunar_CubeSat`.

This section provides an overview of the entire code for this research. It will give a step-by-step walk-through, highlighting important sections in the code. Moreover, its data production, storage and manipulation conventions will be addressed, as well as the visualisation techniques for result generation. This will give a thorough understanding of how the results from this research were created and used to draw conclusions from.

4.4.1. Overview

A block diagram that shows the entire functioning of the code is presented in Figure 4.5 and contains all relevant information for the control simulation. It should be noted that the blue blocks represent functions, that typically consist of one of the functions in `classes.py`. The lines and arrows represent values that are output from one function, and used as input again for the next function. The orange blocks contain additional information for each function block, indicating for example which package is used or which solver is used. Relevant variables are shown in the diagram as well.

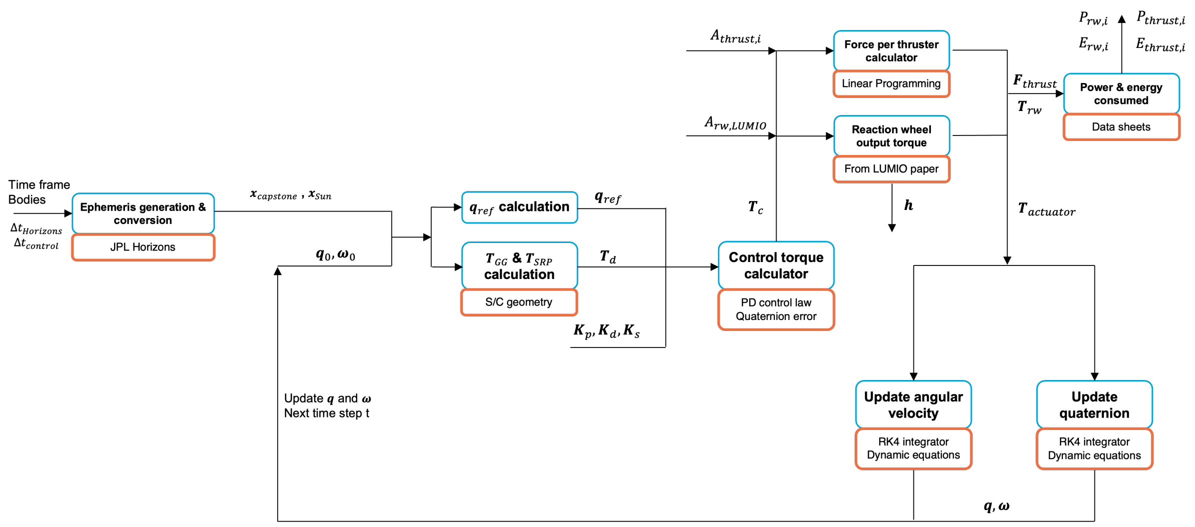


Figure 4.5: Attitude control simulation code block diagram. Blue blocks indicate functions that are executed, the orange blocks contain additional information about the underlying principles or relations of the function blocks. Arrows indicate the order of operations.

In order to go through the code in an organised manner, each function block will be discussed in detail. Before looking at these more closely, the general structure of the diagram should be understood. It represents a closed-loop system, in which each time a loop is performed, one time step in the control simulation is performed. This means that in the simulation at hand, each loop counts as one second in the spacecraft mission, and each loop therefore propagates the attitude of the spacecraft one second further in time. As a starting point for the state vector of the spacecraft, q_0 and ω_0 are fed to the system as initialisation. In the simulation, the first quaternion vector is equal to the desired reference quaternion and the first angular rate is equal to $\mathbf{0}$. After this, the state is updated each loop. It should be noted that a choice can be made in the analysis that is performed: either the thruster analysis can be executed, which is done by activating the *Force per thruster calculator* function, or the reaction wheel analysis is executed, including only four reaction wheels, by the function *Reaction wheel output torque*. From left to right, the functions are presented in the following list.

- **Ephemeris generation & conversion**: As discussed previously, the JPL Horizons database will be queried from within the Python code to obtain the ephemeris data for the CAPSTONE mission. In addition, the ephemeris data of the Sun will be collected as well, and both datasets are retrieved with respect to the inertial frame defined through the centre of mass of the Moon. In subsection 4.4.2, a more elaborate explanation will be given on the method of data collection, manipulation and storage. As input to this function, the required bodies with the desired observer location need to be given, which were specified in Table 4.1. In addition, the time frame for retrieval needs to be given, in this case [00:00:00 01-01-2023] to [00:00:00 15-01-2023]. Also, the time step for retrieval from JPL Horizons ($\Delta t_{Horizons}$) needs to be provided in hours, with the minimum value 1/60 for one minute. Finally, the desired control time step is also provided to this function, which is 1 [s] for this control simulation. As output, two data sets with seven columns are produced: the first column contains the time in seconds or the epochs since [00:00:00 01-01-2023], the other six columns contain the Cartesian coordinates and velocities [x, y, z, v_x, v_y, v_z] in [m] and [m/s].
- $\boldsymbol{q_{ref}}$ **calculation**: Having obtained the required ephemeris data for CAPSTONE and the Sun, each row in the dataset can be iterated over. In this way, propagation of the attitude of the spacecraft can take place while the pre-determined orbit is being followed. Now, the reference quaternion is calculated as shown in Equation 4.16 and takes the current positions of CAPSTONE and the Sun as input. As output, a reference quaternion vector for that epoch is computed.
- $\boldsymbol{T_{GG}}$ **&** $\boldsymbol{T_{SRP}}$ **calculation**: Based on the spacecraft geometry factors and current position of the spacecraft and the Sun, the gravity gradient torque and solar radiation pressure are calculated. The output is a disturbance torque vector for both of them which can be combined in the total disturbance torque vector $\boldsymbol{T}_d$. Special attention should be given to the determination of the angle α, which was described in Equation 4.4 as the optimal angle for the solar arrays to be pointed towards the Sun. To find this optimal angle, an optimisation problem has been implemented in the code. Specifically, the objective is to maximise the sum of the dot products between the columns of $\boldsymbol{n}_s$ (the surface normal vectors) and $\boldsymbol{S}$ (the incident vectors). The value of α that maximises this sum represents the solution to the optimisation problem.
- **Control torque calculation**: The control torque function takes as input a number of variables: the current quaternion vector $\boldsymbol{q}$, the reference quaternion vector $\boldsymbol{q}_{ref}$, the current angular velocity $\boldsymbol{\omega}$, and the vectors containing the gain values. First of all, the quaternion error is calculated, after which the PD logic is applied and the control torque vector at that moment in time is computed. From this point, the diagram is followed based on the reaction wheel or thruster analysis.
- **Reaction wheel output torque**: The reaction wheel configuration matrix as defined beforehand will be fed into the function, as well as the desired control torque. Based on the reaction wheel physical parameters, it is assessed whether the reaction wheels are able to adhere to the desired torque or not based on the calculation of individual wheel torques. At the same time, the angular momentum increase per momentum wheel is calculated and added to the total momentum wheel stored in the wheels. This is denoted by the vector $\boldsymbol{h}$.
- **Force per thruster calculation**: The thruster configuration matrix for the desired electrical thruster configuration (one of the four aforementioned options) is fed into the function, as well as the required control torque. The linear programming solution is then applied, yielding an array of all the output thrust values for each thruster.
- **Power & energy consumed**: Either the reaction wheel torques per reaction wheel are inserted, or the thrust values per thruster, and combined with the hardware specifics to compute the power and energy consumption. Note that the power is a value in Watts at that specific timestep, whereas the energy builds up for each computation loop.
- **Update angular velocity**: With the actual control torque exerted by the actuators, the integration of the dynamic equations can take place. This integrator function takes as input the current angular velocity, the time step, the disturbance torque vector and the control torque vector. In combination with the relations for the current time derivative of the angular velocity, the current state is updated using the RK4 integrator.
- **Update quaternion**: Similar to the previous function, the quaternions are updated with an RK4 integrator function. Having updated the angular velocity as well as the quaternion, the next time

step can begin and the loop is finished. As starting point for the next iteration, the new ephemeris data are retrieved and the new quaternion and angular velocity values are inserted.

It should be noted that the Euler dynamic equations of rotational motion are different for the reaction wheel analysis compared to the thruster analysis. Therefore, the angular velocity and quaternion update functions differ for each of those analyses. Furthermore, at the end of each control loop, all relevant values are appended to arrays for later visualisation. The exact values will be discussed in the next section.

An additional class that does not clearly come forward from the diagram is the `Rotation` class, in which all important operations concerning direction cosine matrices, Euler angles and quaternions are posted. It allows for reference frame rotations, changing vectors expressed in the Moon-centred inertial frame to the spacecraft-fixed body reference frame, for example. The functions from this class are used throughout the other classes in the code.

4.4.2. Data

As mentioned previously, the smallest time step between two adherent points in the ephemeris data from JPL Horizons is one minute. In the attitude control simulation, the desired control time step is one second. This means that the retrieved data from JPL should be manipulated in order to be useful for the research. The unaltered data is first of all stored locally, stating the body name, the observer (e.g. Moon-centred), the start date and time, the end date and time, and the retrieval time step. An example of such a file name is:

```
ephemeris\_data/cartesian\_coordinates\_CAPSTONE\_Moon-centered\_2023-01-01 00:00\_to\_2023
-01-01 00:30\_1m.dat
```

These files contain the Julian date in the first column, and the Cartesian ephemeris coordinates in the other six columns in units $[AU]$ for position and $[AU/day]$ for velocities. In the conversion, the AU units are converted to $[m]$ and $[m/s]$ and the Julian date is converted to seconds from the start time, and each minute that has been collected as the minimum possible time step is complemented with the full 60 seconds that it consists of. Next, from the first second to the 60th second, the Cartesian ephemeris coordinates are linearly interpolated. This is considered to be sufficiently accurate within the scope of this research, since the entire length of the analysis is two weeks. To briefly explain the linear interpolation functioning, an example is presented in Equation 4.24. It can be seen that for the first minute in the data file, the gradient of the x-position is determined with respect to time. In this way, the change for each time step of x is determined and can be used to calculate all the intermediate values of x in the first minute.

$$\begin{aligned} \frac{\delta x}{\delta t} &= \frac{x_{60} - x_0}{60} \\ x_n &= x_0 + \frac{\delta x}{\delta t} * (n - 0) \end{aligned} \tag{4.24}$$

These newly created data files are then stored under a name similar to the one below (`1s` is the control loop time step), and used within the control loop simulation.

```
converted_ephemeris_data/converted_1s_cartesian_coordinates_Sun_Moon-centered_2023-01-05
08:00_to_2023-01-05 15:00_1m.dat
```

A number of variables are stored in lists during each iteration of the control loop for visualisation at the end of the simulation. These variables are $\boldsymbol{q}_{\text{ref}}$, $\boldsymbol{q}$, $\boldsymbol{T}_d$, $\boldsymbol{T}_{GG}$, $\boldsymbol{T}_{SRP}$, $\boldsymbol{T}_c$, $\boldsymbol{T}_{rw}$, P_{rw}, E_{rw}, $\boldsymbol{\omega}$, $\boldsymbol{h}$, the half-cone angle, $\boldsymbol{F}_{thrust}$, P_{thrust} and E_{thrust}. These lists are finally stored as data files as well, so that they are easily accessible and usable for the visualisation code. In addition, references for maximum power consumption, momentum wheel saturation and half-cone angle requirements are added for visual proof of compliance or proper functioning.

4.4.3. Visualisation

In the code, a visualisation class has also been added that contains multiple functions for quick and clear result visualisation. Since the result data files will all be stored, time frame selection within the two-week research period is also possible. First of all, the quaternions over time will be plotted against their reference value to assess their compliance to the required attitude. Moreover, the quaternion error will be plotted over time, to investigate the offset further. These quaternion values can then also be converted to Euler angles for a different visualisation approach. In addition, the control torque and disturbance torque values over time, for each of the rotation axes, can be shown. Power and energy of the reaction wheels over time is also possible to visualise, as well as the specific thrust and torque values of the components. The half cone offset can be plotted versus its requirement. Crucial to understand is that the four different thruster configurations can be plotted simultaneously for comparison. In this way, all relevant performance parameters can be visualised and assessed.

4.5. Robustness

The testing of the control algorithm for robustness and extreme situations is an essential aspect of its validation process. In real-world applications, spacecraft often encounter conditions that deviate significantly from nominal assumptions. These could include unexpected external disturbances, actuator failures, sensor inaccuracies, or deviations in system dynamics due to environmental factors. Ensuring the algorithm can maintain stability and performance under such conditions is crucial for mission success. Robustness testing allows for the identification of vulnerabilities in the control logic, ensuring the algorithm performs reliably across a wide range of scenarios. For instance, the algorithm may be subjected to extreme conditions, such as high disturbance torques due to solar radiation pressure or temporary loss of actuation in one or more thrusters. Testing these situations ensures the control system can handle edge cases that, while rare, could otherwise jeopardise the spacecraft's mission.

Moreover, testing in extreme situations aids in verifying the algorithm's ability to recover from critical scenarios, such as overshooting attitude targets, encountering high angular velocities, or dealing with hardware degradation. This process ensures that the system is not only functional under nominal conditions but also capable of responding effectively to unforeseen challenges. In addition, robustness testing builds confidence in the control algorithm, especially for high-stakes missions where system failure could lead to mission loss. By proactively identifying and mitigating potential points of failure, such testing enhances the reliability, safety, and operational longevity of the spacecraft.

A wide range of robustness tests can be performed, but given the limited time of this research, these could unfortunately not all be executed. After the main simulation code had been created and the first results were produced, a table was created with ideas for improvement and addition to the code. Each of these ideas was given a unique identifier and a priority from 1 to 5. The table is presented in Table 4.9. Note that ROB stands for *robustness*, SIM to *simulation* and SE for *systems engineering*.

Identifier	Addition	Priority (1-5)
ROB1	Simulate and assess single thruster failure behaviour	5
ROB2	Simulate and assess the systems' reaction to imperfect sensors or sensor noise	3
ROB3	Simulate and assess the systems' reaction to initial momentum wheel bias	3
ROB4	Simulate micro-meteoroid impacts as additional disturbance torque	2
ROB5	Simulate an extreme de-tumbling manoeuvrer	4
SIM1	Perform a direct performance comparison between two distinct fly-bys	4
SIM2	Make a direct comparison between deployed solar arrays versus undeployed solar arrays	5
SE1	Systems engineering power budget	4
SE2	Systems engineering mass budget	2
SE3	Cost analysis	2

Table 4.9: Summary of robustness tests and simulations with priority levels.

From the table, it is clear that the first five rows represent actual robustness tests, whereas the last five rows represent additions that have also been identified as options for further completion of this research. The SIM rows show options for the simulation results itself. **SIM1** includes the analysis of two distinct fly-bys, therefore comparing two perilune passings for the ADCS. Parameters such as the total energy required for the fly-by and the half-cone angle offset can be compared between the two cases. **SIM2** has been explained clearly in the table and includes adjusting certain parameters in the simulation so that solar arrays become undeployed. After this, the total maximum power consumption of both cases can be compared, as well as the tot energy required for both cases, during the full simulation. In addition, a comparison for the half-cone angle offset for both cases can be made. The SE rows are connected to further systems engineering or overall spacecraft design tasks, such as creating an adjusted power and mass budget based on the proposed ADCS in this report, as well as creating an extensive cost analysis for the mission in case the electrical thruster configurations would replace the current LUMIO ADCS. Direct comparisons between all previously mentioned electrical-thruster-only configurations can be made, for all the SIM and SE tests described.

In this section, three of the additions to the code as presented in the table will be elaborated upon: the single thruster failure behaviour (**ROB1**), the reaction to an extreme de-tumbling manoeuvrer (**ROB5**) and the comparison between the deployed and undeployed solar arrays (**SIM2**). This table will be revisited throughout the remainder of this report. Throughout these robustness tests, the thruster-only configurations as were seen during the regular simulations will be used as the test cases, and the reaction wheel analysis (with only the reaction wheels) will be used as a base case comparison. Therefore, there is no simulation in which reaction wheels as well as thrusters are used.

4.5.1. Single Thruster Failure

Single thruster failure can be added to the simulation. Before the control loop starts iterating over the epochs, one thruster should randomly be picked from the desired configuration and its thrust output should become equal to zero. Two distinct approaches can now be followed:

1. The system is aware of the malfunction and corrects itself using the remaining thrusters. This essentially means that the linear programming solution is found with $n-1$ number of thrusters instead of n. This approach will not give any valid solutions for thruster configuration 1 and 2, since the system becomes underdetermined by the exclusion of one of the thrusters. The attitude will be maintained similar to the original simulation for configuration 3 and 4.

2. The system is unaware of the malfunction of one of the thrusters. In this approach, the thrust outputs of the thrusters are computed, after which the randomly chosen failed thruster output is set to zero. With this new thrust matrix, the new control torque is computed and fed back into the control loop. Therefore, each control loop iteration does not meet the requirements for attaining the correct attitude. The reference attitude, however, is computed again at each time step. It is expected that the attitude error will remain approximately constant over time but will never return to zero. It is expected that configurations 1 and 2 will again not produce any results for this approach; this will be tested for confirmation.

In the code, for both approaches, a random thruster number needs to be selected based on the available number of thrusters from the selected configuration. Next, for approach 1, the linear programming solution is given an extra constraint stating that this selected thruster has an output of zero. After this the code is run similar to the original simulation. For approach 2, the linear programming solution is found first, and after the thrust values have been found, the designated thruster output is set to zero. Then, a new control torque is computed and fed to the rest of the control loop.

For either approach, different parameters are of interest to investigate. In approach 1, the performance in terms of quaternion error over time will be exactly equal to the original simulation. Therefore, only the power and energy consumption of the thrusters is of interest in this test. An increase in power and energy is expected, and the increase can easily be quantified compared to the base case. Approach 2 will yield differences in terms of performance and pointing stability. The quaternion error over time, along with the half-cone offset angle, can be compared to the base case. In addition, the power and energy consumptions can be observed. The main question to answer for approach 2 is: will the system be able to adequately control the attitude for the science pointing requirements as laid out in Table 2.3, when it is unaware of one of the thrusters failing?

4.5.2. Solar Array Deployment

As mentioned previously, the solar arrays of the LUMIO spacecraft can be in a deployed state or in an undeployed state. The differences between these two configurations is the mass moment of inertia matrix of the spacecraft and the solar radiation pressure disturbance torque magnitude. The power generation difference is not considered for this research. A quantitative comparison can be made for both configurations. The deployed solar array case has been the base case for this research, since this poses the largest strain on the attitude control system.

The undeployed case creates a simpler approach towards the solar radiation pressure calculations. The matrices $\boldsymbol{S}$, $\boldsymbol{n}_s$, $\boldsymbol{A}_{panels}$ and $\boldsymbol{c}_p$ as seen in Equation 4.2 Equation 4.3 Equation 4.4 and Equation 4.5 can now be reduced by excluding the last 4 columns for each. Then, the exact same equations for calculation of the solar radiation pressure can be used. In addition, throughout the code, $I_{undeployed}$ should be used instead of $I_{deployed}$.

Performance comparison for the quaternion error, half-cone offset angle, power and energy consumption between the four thruster set-ups can now be performed. This will be shown as a numerical comparison, from which it becomes evident how the undeployed case compares to the deployed case for each of the thruster-only set-ups. Although the undeployed configuration will not generally be the spacecraft situation, it adds to the completeness of the research to analyse its response in this case as well.

4.5.3. De-tumbling Manoeuvrer

A de-tumbling manoeuvrer involves initialising the spacecraft with a high angular velocity to simulate a scenario where it is spinning uncontrollably, often as a result of deployment from the launch vehicle or an external disturbance. This simulation assesses whether the thruster-only-based control system is capable of counteracting the angular velocity and stabilising the spacecraft to achieve the desired attitude. The configuration that will be tested in this case, is configuration 1 as previously seen for the regular simulation. The analysis helps to determine the system's ability to handle extreme initial conditions and recover to a stable state. The manoeuvrer aims to identify the boundaries for a completely uncontrollable satellite, such as the maximum initial angular velocity beyond which the control system fails to stabilise the spacecraft. Additionally, the time required for the spacecraft to settle within a predefined tolerance of the desired state is of interest, as it is a critical metric for operational readiness and mission planning.

In the code, the only change that has to be made is initiating the angular velocity at a high rate. For this analysis, a number of initial angular velocities will be tested, which are presented in Table 4.10 below. For CubeSats, in general, angular velocities exceeding 50 $[°/s]$ or 0.87 $[rad/s]$ is considered to be high due to their limited de-tumbling capabilities. As can be seen from the table, tests 1, 2 and 3 will assess the de-tumbling of one axis only, with a 180°initial spin angular velocity. The other four tests will assess the same angular rate over all axes, and increase from 50°until 360°per second. The last is clearly an exaggerated scenario, but nonetheless valuable for assessment.

Test number	$[\omega_{0,x}, \omega_{0,y}, \omega_{0,z}]$ $[deg/s]$
1	[180, 0, 0]
2	[0, 180, 0]
3	[0, 0, 180]
4	[50, 50, 50]
5	[100, 100, 100]
6	[200, 200, 200]
7	[360, 360, 360]

Table 4.10: Initial angular velocities in degrees per second for the de-tumbling robustness test of the attitude control algorithm.

Results that are important from this research are again the quaternion error and half-cone offset over time, as well as the required control torque for de-tumbling and, with this, the power and energy consumption of the actuators. Reaction wheel as well as thruster analysis can be carried out again and the settling times from the excessive angular rates can be analysed and compared to each other.

In order to summarise this chapter and gain a clear overview of the analyses, their time frames, the configurations tested in them and the results for comparison, Table 4.11 is provided for convenient reference.

Analysis	Time frame	Configurations	Results
Regular simulation	[00:00:00 01-01-2023] to [00:00:00 15-01-2023]	Reaction wheels Thruster conf. 1 – 4	$\boldsymbol{q}_e(t)$, $\boldsymbol{q}(t)$, $\boldsymbol{\omega}(t)$, $\boldsymbol{T}_{GG/SRP}(t)$, $\boldsymbol{T}_c(t)$, $\boldsymbol{T}_{rw}(t)$, $P_{rw}(t)$, $E_{rw}(t)$, $J_{thrust}(t)$, $\boldsymbol{h}_{rw}(t)$, $\boldsymbol{F}_{thrust}(t)$, $P_{thrust}(t)$, $E_{thrust}(t)$, $\beta(t)$
STF1	[05:33:20 05-01-2023] to [14:46:40 05-01-2023]	Thruster conf. 3 & 4	$\Delta\boldsymbol{F}_{thrust}(t)$, $P_{tot,max}$, E_{tot}
STF2	[05:33:20 05-01-2023] to [14:46:40 05-01-2023]	Thruster conf. 1 – 4[4]	$\boldsymbol{q}_e(t)$, $\beta(t)$, $\Delta\boldsymbol{F}_{thrust}(t)$, $P_{tot,max}$, E_{tot}
Undeployed arrays	[00:00:00 01-01-2023] to [00:00:00 15-01-2023]	Thruster conf. 1 – 4	$\beta(t)$, $P_{tot,max}$, E_{tot}
De-tumbling	[00:00:00 01-01-2023] to [00:02:00 01-01-2023]	Reaction wheels Thruster conf. 1 – 4	$\boldsymbol{q}_e(t)$, $\boldsymbol{\omega}(t)$, $\boldsymbol{F}_{thrust}(t)$, , $P_{rw}(t)$, $E_{rw}(t)$, $P_{thrust}(t)$, $E_{thrust}(t)$

Table 4.11: Overview of the analyses performed, along with its time frames, configurations of interest and expected results.

[4]It is expected that configurations 1 and 2 will not yield any results.

5 Experimental Characterisation

5.1. Introduction

In any engineering industry, simulations are created to imitate the real world. For example, the simulation of the acceleration in a race car should give the race car manufacturers an idea of the performance of the actual car. The question that arises, is whether the simulation that is created, actually represents the real system, if it were to be built. The link between the simulation and the real world is called validation and is an important part of any research performed in engineering practices.

For the attitude control simulation as presented in Chapter 4, a number of validation techniques could be applied. The first option is creating the actual ADCS as it is proposed, and testing it in the same orbit on the same spacecraft bus with the same specifications. If the results from the actual system would be positive, the research can ascertain the developers of the LUMIO mission that it will work on their mission as well. However, this is a very unpractical method of validation. First of all, one would build the exact same spacecraft, which is a waste of time and an exaggeration for the level of validation necessary. Furthermore, there are only a few options to launch spacecraft in their desired orbits, certainly for missions to the Moon. Finally, it is very costly to build the entire system and have it launched. For all of these reasons, this method of validation is not selected for this research.

An additional method of validation for such algorithms in space application is performing a Hardware-In-the-Loop (HIL) simulation. Generally, a HIL simulation tests real-time embedded systems (which contain the simulation software) by connecting to the actuators, or to the physical plant. In this way, the hardware is tested for its response to the software, and real performance parameters can be evaluated. HIL simulations include accurate force gauges that measure the applied forces of the thrusters, in this case. Moreover, the rest of the environment (vacuum, disturbance torques, spacecraft bus) should also be recreated, and the bus should be able to rotate freely along its primary axes. All in all, recreating the space environment on Earth is a possibility, but since a lot of hardware and other practicalities need to be considered, it will not be the validation method for this research either. Cost and the professionalism of the required equipment are too excessive for this research.

What validation method is then possible for this research to somehow prove its relevance in the real world? Connecting the simulation software with a micro-controller hardware module is possible; this would show the possibility of adapting the control logic to an on-board spacecraft computer. Moreover, micro-controllers are able to generate signals, based on their pre-programmed rules, which means actuators could be commanded according to the requirements imposed by the attitude control algorithm. Although the vacuum arc thrusters from SSL are not physically available for this research, their functioning can be mimicked by other hardware modules such as valves or LEDs. These hardware modules can in their turn communicate the signal received, which paves the way for a comparison between desired and actual signals. In this way, a simplified engineering model of the ADCS computer along with the actuators is possible to construct and use for validation of the attitude control simulation.

Therefore, this chapter will completely focus on the validation of the attitude control simulation by means of the recreation of an OBC with an MCU that generates and sends signals to the relevant actuators, based on computations performed within the OBC itself. First of all, the MCU to be used will be elaborated upon in Section 5.2. The way in which software is written for the MCU and uploaded towards the MCU will be explained, along with debugging conventions. Moreover, this section will make a distinction between the code blocks of the original simulation that would need to be part of the OBC in a real space mission, and the ones that would be part of the environment. Then, a code overview will be given, in which the approach to porting all relevant code to the MCU will be explained as well as the interaction between the Python simulation environment and the embedded environment. Next, in Section 5.3, the connection with the ”dummy” actuators will be explained, including the signal creation and signal feedback reception. The different set-ups will be discussed in detail and the other expected results will be shared.

5.2. Code Porting

5.2.1. Experimental Framework

The micro-controller unit to be used in this research is the STM32 Nucleo-F303RE development board, shown in Figure 2.15. It features an ARM Cortex-M4 microprocessor and includes a module called the ST-LINK, enabling USB connectivity for code uploading and debugging. The board has a total of 76 pins that support various functionalities, such as sending electrical signals (PWM), transmitting data (e.g., through UART or I^{2}C), or providing power connections. These capabilities are collectively referred to as peripherals. Detailed specifications of the STM32 can be found in STMicroelectronics [47]. The pin ports are labeled PA, PB, and PC, numbered 1 to 15, as shown in Figure 5.2. Each port can be assigned to a specific peripheral, such as a UART connection or a PWM timer. Important abbreviations from the figure are clarified below.

- **GND**: Ground, where one of the connectors to the power supply should always be connected to in order to raise a desired potential.
- **+3V3** or **+5V**: Other connector to the power supply, providing either 3.3[V] or 5.0[V].
- **NC**: Not Connected, often included on the PCB for compatibility, future use, or mechanical stability.

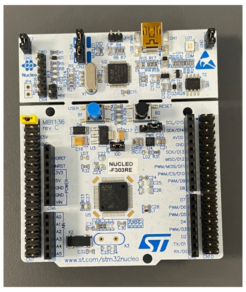


Figure 5.1: Photo of the STM32 Nucleo RE303RE development board, including an ST-LINK on the top for communication via USB.

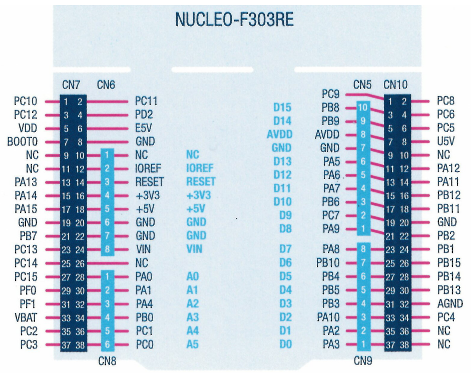


Figure 5.2: Systematic layout of the pin functionalities of the Nucleo-F303RE board. These pins coincide with the pins seen in Figure 5.1.

While the STM32 Nucleo-F303RE is designed for prototyping and laboratory use, it is not flight-qualified for spacecraft applications due to the lack of radiation hardening and long-term reliability in extreme environments. However, its 72 MHz ARM Cortex-M4 processor, equipped with a floating-point unit (FPU), provides sufficient computational performance for real-time control tasks in this research. This makes it well-suited for validating and testing ADCS algorithms under simulated conditions.

Although the STM32 Nucleo-F303RE differs from the microcontroller typically used in a flight-ready CubeSat, such as a radiation-hardened processor like the LEON series or a space-qualified ARM Cortex variant, the research focuses on developing and validating hardware-agnostic algorithms. Once validated, these algorithms can be optimised for the specific processor used in the spacecraft. The use of the STM32 board ensures a practical and efficient development process while maintaining compatibility with the eventual transition to flight-ready hardware.

After connection of the Nucleo board to a personal computer, the STM32IDE program can be run and a project can be created for this specific board. Before any code can be written or can be generated, it should be clear what the goal is of working with the STM and which parts of the original simulation need to be added to the board. Also, a brief overview of the differences in C code compared to Python should be given. First of all, the STM32 board should be programmed so that, when connected to power, it automatically and autonomously executes the required parts of the simulation. It should be noted that this practical will only include the commanding of thrusters, and considers thruster configuration 1 as its set-up. In real spacecraft OBCs, the calculations that would be done are:

- **q_{ref} calculation**: Normally, the spacecraft would now its own location based on star sensors and Sun sensors (in cislunar environment). Potentially, communication with an Earth-based ground station could be used, and a Kalman filter can be added to the measurements to create an accurate estimation of the spacecraft state. Based on this estimation, it would determine its own position with respect to the Moon and also have the position of the Sun with respect to the Moon. Using these values as input, it calculates the desired attitude as was also performed in the original simulation.
- **Control torque calculation**: Using the reference attitude q_{ref}, its current attitude (measured with the help of a.o. gyroscopes) and angular velocity, and pre-determined gain values, the OBC is able to compute the required control torque at each moment of time by using the relation as described in Equation 4.12. This desired control torque can then be used to create signals the thrusters.
- **Force per thruster calculation**: Based on the control torque from the previous step, the OBC should be able to compute the desired torque per thruster. It has been pre-programmed with the exact thruster configuration in order to properly do so and will be able to convert these values into useful signals, as will be seen in the next section.

In addition, a number of functions from the `Rotation` class will also be programmed in the STM32 board, since these are necessary for execution of above functions. The other mathematical operations that have been executed in the original simulation are all part of the environment on a space mission.

Code on an embedded system is typically designed to initialize various peripherals, such as timers (for time-sensitive operations), GPIOs (for general input and output control), and UART connections (for serial communication). After initialization, the system enters an infinite while-loop, which continuously monitors and manages the system's operations when the PCB is powered. To enhance responsiveness and efficiency, interrupts are implemented. Interrupts are special mechanisms that pause the main program flow to execute specific code in response to predefined events, such as receiving a message, a timer overflow, or a sensor signal. This event-driven approach ensures that the system reacts promptly to critical inputs without continuously polling for changes, thereby avoiding unnecessary computational overhead. By enabling the system to execute tasks only when needed, interrupts significantly reduce power consumption and optimize memory usage. This is particularly important in resource-constrained environments, such as battery-operated or low-power devices.

A `main.c` file is uploaded, which contains initialization functions, timer configuration settings, an error handler, interrupt rules, and the infinite while-loop. To maintain clarity and organization, not all functions are stored in this file. Instead, specific functions are placed in separate `.c` files and are called from the main file when needed. In C programming, each variable is assigned a specific memory location, a process managed using pointers. A pointer is essentially a variable that stores the memory address of another variable, allowing direct access to the data stored there. Each variable has a unique pointer that can be used throughout the code. Variables in C can be categorized as global or local. Global variables are accessible across the entire program, while local variables are confined to specific functions. Local variables are especially useful for iterative processes, as their values can be overwritten with each iteration, ensuring they serve only their intended scope. Moreover, variables are assigned the data type `double`, which is a floating point number capable of containing 64 bits of decimals. This accuracy is equal to the `float` data type used in Python. Each `.c` file has an associated `.h` file, which serves as its header file. In STM32IDE (or any C-based development environment), the `.h` file is used to declare the functions, macros, constants, and sometimes global variables that are defined and implemented in the corresponding `.c` file. The header files for the script are all included in the main C file.

An adjusted version of the Python file `simulation.ipynb` has been stored called `embedded.ipynb`, which uses the `serial` package to send values to and receive values from the STM32 board. Necessary values are all concatenated in one array and then converted to a string, which allows for quick UART data transmission. In addition, code has been added that allows for receiving data via UART and converts it back to floating point numbers again.

5.2.2. Porting

Now, a step-by-step code walkthrough can be provided. The entire code in C can be viewed in Appendix C. First of all, the peripheral allocation should be clear. Three different peripherals were initialised:

1. **GPIO**: Assigned to PA5 and used for the toggling of an LED on the board itself. This LED is identified by the abbreviation LD3, which is connected to pin PA5. As mentioned previously, the GPIO simply sends a 3.3V signal through, and can therefore turn the LED on and off. This functionality will be used to assess the proper code functioning in development; a visual confirmation of code running is often used in embedded software engineering.
2. **UART**: Assigned to PA2 for transmission (Tx) and assigned to PA3 for reception (Rx). In order to communicate with the personal computer, these pins need to be connected to, for example, a USB-to-UART module. The USB connection established through the ST-LINK can also be used for UART communication. UART is used in this research instead of I^2C due to its simplicity and existing Python libraries for connection.
3. **PWM timer**: The timer peripheral is connected to pin PC0 and will generate the required signals for actuator activation. At first, only one PWM signal will be created, to ascertain its functioning. After this has been proven, the set-up will be extended to the six thrusters present in thruster configuration 1. The PWM timer settings will be as follows: the assumed Pocket Rocket operational frequency is 10 [Hz]. Therefore, the period of the PWM signal is set to 100,000 [μs] and the duty cycle is initially set to 25%. As will be seen in Section 5.3, the duty cycle will be adjusted based on the internal OBC computations.

5.2.2.1 1/0 Test

Having assigned the peripherals, a first test of the connection between PC and MCU had to be implemented. This was done with the 1/0 test and included a slight adaptation of the original Python simulation code. The main idea behind this test is running the Python simulation as was done before, but make the code stop at the beginning of each iteration. It will only continue after receiving a ”1” in string format from the MCU. Then, at the end of the iteration, a ”0” is sent back to the MCU as confirmation of finishing the iteration. After this message has been received by the MCU again, it will re-send the ”1” and the iterations continue. In the C code, after initialisation of the peripherals, the first infinite while-loop is started. It instantly transmits a ”1” to Python using the `HAL_UART_Transmit` function. HAL is a library containing functions to be used on the embedded system. Next, another infinite while-loop is entered that calls the `HAL_UART_Receive` function, and will continue to do so until the value of the received data is a ”0”. Upon reception of this ”0”, the LED is toggled as visual confirmation. Then the initial while-loop starts again. Next to the visual confirmation of the LED, the time for each iteration in the Python code (so: time for the execution of one time step in the control simulation) is measured and compared to the base case in which no connection with the MCU is made.

5.2.2.2 PD Control

The next step in the code porting process is the conversion of one of the functions that need to be present in the MCU. The PD control function was implemented first, since this is the computationally least expensive of the three functions. The function does require a number of inputs, as was discussed previously, and these are for the moment all coming from the Python code. In the dedicated simulation function in `embedded.ipynb`, the required values are concatenated into the same array and then converted to a string with a comma as their separator and `\n` at the end of the string. Using the Python `serial` package, these values are then transmitted to the C code and used for the calculation of $\boldsymbol{T}_c$. The Python code is now paused until a new reception via UART is established. The torque vector is then received in string format, and converted to a floating number vector again. On the MCU, the string is first received bit by bit, meaning character by character. Then, a dedicated function converts the string to floating point numbers (doubles) again, using the commas and `\n` signs as indicators. These values are then used in a dedicated function for computing the proportional-derivative control law as was seen in Equation 4.12 and uses the an additional function for computing the quaternion error, as was seen in Equation 2.31. When the control torque vector comes out of this function as a result, it is converted to string again to be sent back to the personal computer via UART. In contrast to the functioning of the code for the 1/0 test, the infinite while-loop is now empty, but the reception of data via UART will cause it to interrupt and do the computations. When the control torque has been sent back, the LED will toggle again for visual confirmation. The calculations from the C code are compared to the calculations in Python, and the accuracy is determined with this. Furthermore, the iteration computation time is compared to the all-Python simulation case.

5.2.2.3 Reference Attitude

The next step in the porting process is migrating the $\boldsymbol{q}_{ref}$ function. This function takes as input the position of the spacecraft with respect to the Moon and the position of the Sun with respect to the Moon. For this reason, these position vectors (retrieved from CAPSTONE and Sun ephemeris in the Python code) should be sent through UART to the MCU as well, and an extension of the function that converts the values to string and sends them needs to take place. Also, the reference quaternion does not need to be computed any more in the Python simulation and also does not need to be sent any more. No other adjustments are needed on the Python part of the code at this moment. In the MCU, the C code should receive the new variables and feed them to a function that computes the reference quaternion. An additional function that converts the resultant direction cosine matrix to quaternions is created as well. The reference quaternion value is now directly fed into the C-based control torque calculation and the simulation runs the same as seen in the pervious porting step. Again, the values to be measured and compared here are the floating point number accuracy and the iteration computation time.

5.2.2.4 Thruster Allocation

Next, the thruster allocation algorithm as shown previously needs to be ported to `main.c`. On the Python side of the code, not many changes are observed except the omission of the linear programming solution itself. The thruster output values for each time step are received again via UART for comparison to the base case. In the C code, after the control torque has been computed, the mixing matrix, number of thrusters, control torque vector and maximum allowable thrust are fed to the thrust solution function. As mentioned previously, thruster configuration 1 will be included for analysis, but the thruster mixing matrices for all four configurations will be included as global variables in case a different configuration is tested. The thrust output solution is found using the ECOS solver, which is similar to the one used in the Python code and was installed for this C environment. The thrust outputs are then sent back to Python using UART, in string format again, for further power and energy calculation and for comparison with the base case. Again, for this function in C, the accuracy of the calculation and computation times are compared. It is expected that the accuracy will remain similar as was seen previously, since the same solver is used in the optimization problem. The computation time will significantly increase compared to previous ported functions.

Now, all the relevant functions that an OBC should be able to execute have been ported to the STM32 Nucleo-F303RE development board and should be executed when plugged into powered, based on the data received from the personal computer. A final code overview diagram, similar to the one seen in Figure 4.5, is included in Figure 5.3, in which the distinction between Python code on the PC and C code on the MCU is made clear. The next step in this practical is the conversion of the required thrust values for the desired control torque to signals to be sent to hardware modules, a process that is examined in the following section.

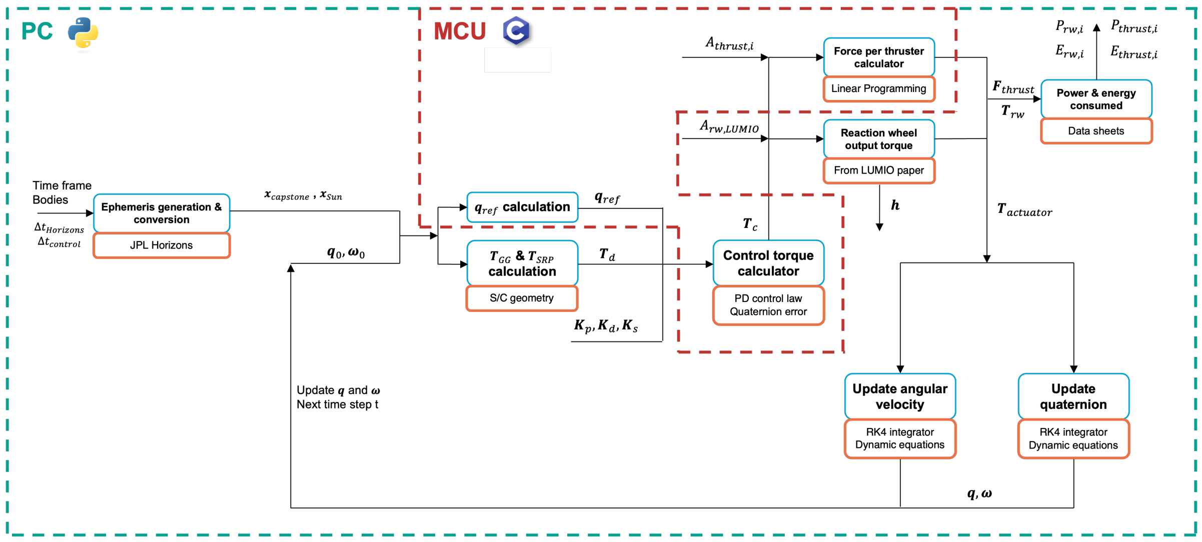


Figure 5.3: Attitude control simulation code block diagram, similar to Figure 4.5 with the separation of PC-based and MCU-based computations. Logos taken from open source, copyright free internet sources.

5.3. Actuator Connection

The next step in this practical process is the connection to the hardware modules. The designated hardware module for this experiment is a solenoid valve. Before diving into the hardware layout and specifics, it should be understood why this specific scenario was selected for testing. First of all, in order to validate the functioning of the OBC output signals, any hardware that responds to it would suffice, and the solenoid valve used in this experiment was available. Also, valves are used within chemical thrusters in order to control the thrust, and are therefore somewhat related to the actual mission geometry. It should be noted, however, that the minimum impulse bit (MIB) of solenoid valves is typically much larger than that of vacuum arc thrusters (VATs). The impulse bit, which is measured in Newton-seconds (Ns), represents the smallest achievable impulse for a device and is a constant characteristic. For solenoid valves, the MIB is in the milli-Newton-second range, while for VATs, it is in the micro-Newton-second range. This significant difference highlights that the experimental setup does not physically emulate the connection used in VATs. Instead, the solenoid valve is solely used to verify the proper functioning of the microcontroller unit (MCU) and assess its signal output.

Vacuum arc thrusters (VATs) are typically connected to the OBC through electrodes that transmit electrical pulses. As explained in subsection 2.6.2, these pulses are often square waves derived from pulse-width-modulated signals. These signals are delivered to the VATs at a pre-determined frequency, with the width of each pulse modulated by the duty cycle. The duty cycle directly controls the pulse width, determining the on-time of the thruster and the resulting impulse per firing. At the minimum achievable duty cycle, the MIB is established, representing the smallest impulse the thruster can reliably deliver over time. Attempting to further reduce the duty cycle beyond this minimum will cause the thruster's performance to fall below its resolution limit, leading to unreliable or improper operation.

The solenoid valve used in the first experimental test setup is the AirTac Model 2V025-08 from AirTac International. Its key characteristics can be viewed in AirTAC and Trimantec [2]. The valve operates with a 12 $[V]$ supply voltage and consumes 3.0 $[W]$ of input power from the power supply to actuate. Additionally, the control signals that trigger the valve are logic-level signals requiring a 3.3 $[V]$ input. The physical valve characteristics can be seen in Figure 5.4 below; its total dimensions are approximately W$\times$L$\times$H = 2.0$\times$6.0$\times$6.0 $[cm]$.

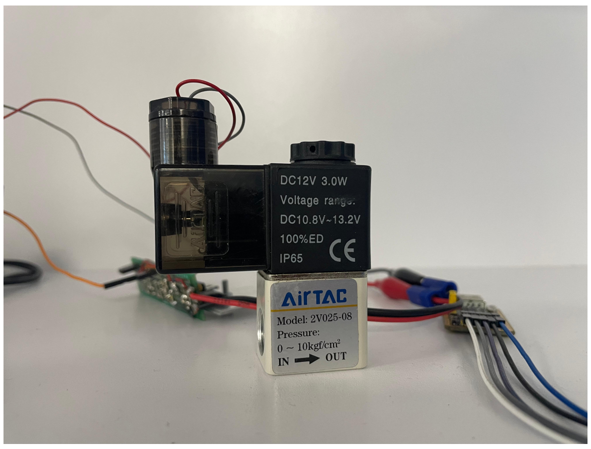


Figure 5.4: AirTac Model 2V025-08 side view.

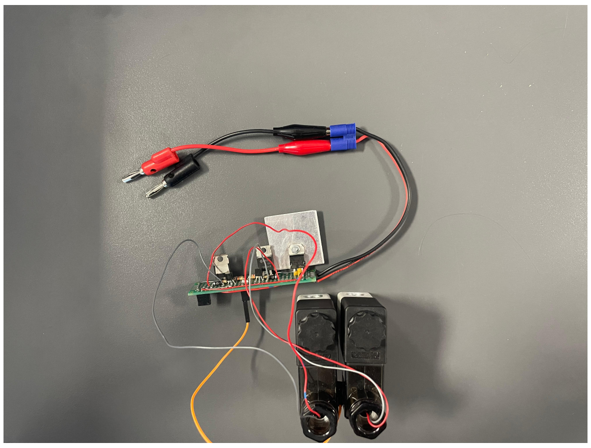


Figure 5.5: Two AirTac Model 2V025-08 attached to a power distribution board with a heat sink, main input power cables and signal input cable for connection to the MCU.

The solenoid valve was attached to a power distribution board along with an additional valve. The second valve was disregarded for the experiment. A top view of this set-up is shown in Figure 5.5, in which the two valves are seen on the the bottom, both connected to the power distribution board. The large metallic plate acts as a heat sink for excess heat. The two cables on the top will be connected to the main power supply, supplying 12[V] and 3.0[W]. The orange cable on the bottom of the power distribution board will be connect to the MCU PWM signal output pin, in order for the signals to be transmitted to the valve. The exact lay-out of the power distribution board is not relevant to understand for this research, and it should be understood that the second valve was not commanded with an additional PWM signal.

Regarding the required software adaptation of this experiment, a minor addition has to be done in the final code as seen in the previous chapter. Thruster configuration 1 will be used as a software input. After the thruster allocation problem was solved using the linear programming solution, the thrust values for each thruster were output. Since only one valve is available in this experiment, only one thruster output will be taken to create signals from, which for convenience will be the first thruster in the array. Upon examination of the simulation results, it was seen that this thruster was indeed the most critical thruster for configuration 1. Its thrust value is fed to a function that updates the PWM signal duty cycle (DC), in which the duty cycle is computed as seen in Equation 5.1. In STM32IDE, the duty cycle needs to be provided in micro-seconds, which is why it is multiplied with the timer period again. Furthermore, when the thrust output of thruster 1 is smaller than the MIB, the duty cycle will be set to zero. In the code, the duty cycle will be sent through UART to Python in order to obtain the settings for the commanded PWM signal. Then, using the function `HAL_TIM_SET_COMPARE`, the timer duty cycle can be adjusted to the value that has just been calculated. In this way, during each iteration time step, the duty cycle for the valve is adjusted based on the required thrust output. In a real-world application, this would be the exact same way as the thrust output would be varied. [42]

$$DC = \frac{F_{thrust,1}}{F_{thrust,max}} \tag{5.1}$$

The final set-up can now be constructed and is shown in Figure 5.6. In the middle, the PC used for this research can be seen, with the `embedded.ipynb` script open to run. It is connected to the STM32 Nucleo board, via USB. A red light is toggled, indicating that the board is activated. The board is connected to the valve power distribution board on pin PC0, which generates the PWM signal. A blue wire is connecting the negative pole of the power supply to one of the ground pins on the Nucleo board, in order to close the electrical circuit. The power distribution board is plugged into the power supply, which is set to the correct settings (12[V] and 3.0[W]) and turned on. The Nucleo board will feed the 3.3[V] logic signals. Upon activation, an LED within the valve was toggled to confirm its proper connection.

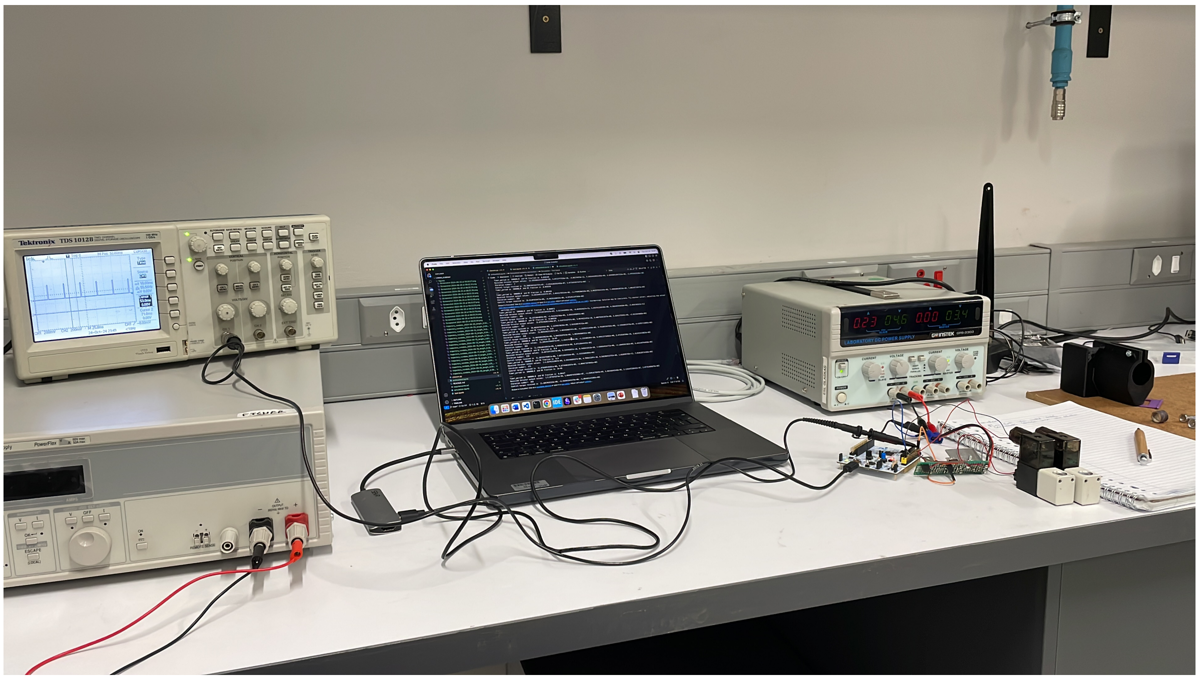


Figure 5.6: Complete experimental valve set-up, including an oscilloscope, a personal computer, a power supply, the STM32 Nucleo developer board, two valves and the power distribution board they are connected to.

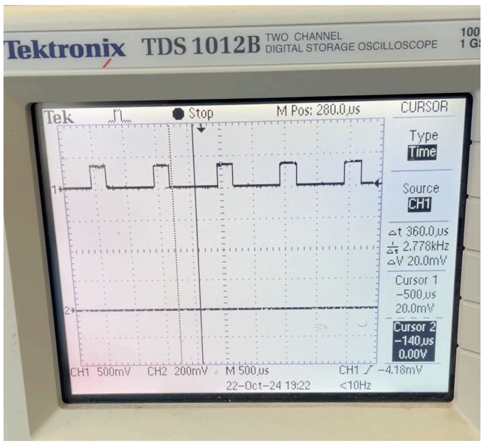


Figure 5.7: Example of the screen of the oscilloscope used in this research to analyse the characteristics of the measured signals.

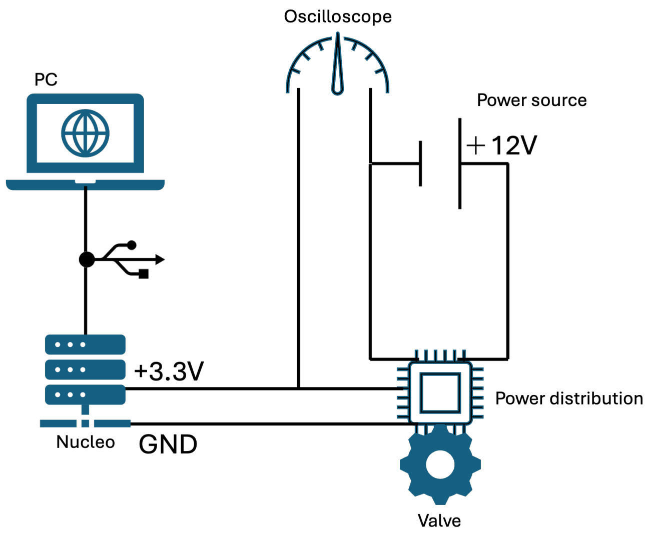


Figure 5.8: Schematic overview of Figure 5.5.

Finally, on the top left side of the image, an oscilloscope can be seen, that is connected to the PC0 pin and the negative pole of the power supply, in order to measure the electrical signals provided. The oscilloscope has multiple buttons and switches that can be used to analyse the incoming signals; a snapshot at a specific moment in time can be made, which can then be enlarged and played back over time. Moreover, the frequency and period of the signal are estimated by the device, as well as the amplitude of the signal. Two measurement lines are present, for dual signal measurements. An example of an arbitrary square wave signal measured by this oscilloscope is shown in Figure 5.7. The oscilloscope is used in this research to observe the incoming signals when the thrust output is larger than the Minimum Impulse Bit, and therefore confirm visually that the valve activation occurs when the signal are being sent. A schematic overview of the entire set-up can be seen in Figure 5.8, in which all relevant components and voltages have been indicated. Note that the valve and power distribution board can be seen as one component here. Also, the GND connection on the LUMIO board is, as mentioned previously, connected to the negative pole of the power source.

The experiment is started by running the Python script that was also run for the last step during code porting. After the required variables from the PC side have been transmitted and received by the MCU, it will perform its calculation and send a pulse-width modulated signal to the valve. If this signal is non-zero and therefore above the MIB, the thruster will respond and a clicking sound will be heard. The iteration does not stop before the thruster is fired, which means a new signal can immediately be created after the first duty cycle adjustment has taken place. The time span around which this experiment is performed will be shorter than was examined during the Python-based simulation: the close fly-bys will be especially of interest in this experiment, since these are the more extreme situations in which the thrusters can be used. The time frame used is therefore between [05:33:20 05-01-2023] and [14:46:40 05-01-2023]. This flyby time frame was found from the preliminary results

From this experiment, a number of results can be analysed. First of all, as was also seen in the code-only approach, the time for each iteration can be measured and stored. Since the only addition to the code length is the adjustment of the PWM duty cycle, it is expected to have similar iteration times as the last test in subsection 5.2.2. Next, the oscilloscope attached to the set-up can be visually inspected and a visual confirmation of the proper valve working can be given. Since the level of accuracy in this method is very low, an additional signal creation will be incorporated. The output pin PC0 will internally be connected to an Analogue-to-Digital Converter (ADC) peripheral, that samples the analogue PWM signal and saves them. These values are then used to determine the actual duty cycle sent out by the pin as well as the actual frequency. Comparing these values to the desired values gives an indication of the accuracy of the system and the noise present. In addition, the duty cycle adjustments are compared to the actual thruster firing and confirm its adherence to the MIB.

An important note needs to be taken on the Minimum Impulse Bit and the maximum thrust output. From the preliminary results of the main simulation in this research, it was found that the maximum thrust levels only lie in the order of magnitude of 10^{-7} $[N]$, which is well below the maximum thrust threshold of 200 $[\mu N]$. In addition, considering the imposed MIB of 1.18 $[\mu Ns]$ for the Pocket Rocket, this would mean that these values would only be possible to continuously exert over a longer time period, which cannot be implemented due to the 1-second accuracy of the control algorithm. For this reason, for the hardware connection practical, the MIB value that is used will be 0.01 $[\mu Ns]$, and the maximum thrust value $F_{thrust,max}$ is set to $8.0 \cdot 10^{-7}$ $[N]$, so that duty cycle values are used that are realistic and that can be visualised.

After completion of the experiment with one valve, additional PWM ports were created in STM32IDE, so that signals for all 6 thrusters could be created. Although these could not be attached to physical thruster hardware modules (only a number of LEDS were present), their outputs could be measured and analysed. From these outputs, the duty cycles and frequencies over time could be obtained, along with the adherent thrust values from the thrusters. As will be seen in the next chapter, these outputs can be visualised and compared to each other, to ensure the proper functioning of the set-up.

6 Results

Having established the required simulation environment and robustness tests, the simulations were conducted for each thruster configuration and the reaction wheel analysis. For the predetermined analysis period of two weeks, as explained in Chapter 4, a substantial amount of data was collected and stored for visualisation. Additionally, the measured results from the experiment described in Chapter 5 have also been gathered. This comprehensive dataset allows for an in-depth inspection, further discussion, and the formulation of conclusions.

This chapter presents the visual and numerical results from the simulations and experiments in a chronological sequence. First, the results of the reaction wheel simulation are shown, in order to determine the benchmark for the thruster research, as well as elaborate more clearly on the visualisation methods used and what can be seen in them. Next, the results for the thruster analysis are presented in a similar format, followed by a comparative study between the reaction wheel and thruster systems. This concludes Section 6.1, which focuses on the standard simulation results.

In Section 6.2, the robustness test results are discussed. Only relevant parameters critical to the algorithm's robustness are presented, omitting less significant simulation parameters. Then, Section 6.3 elaborates on the numerical and visual results from the practical experiment. Finally, Section 6.4 outlines the verification and validation methods applied to all simulations and experiments, as discussed in the previous chapter. These verification and validation methods were executed after the initial results were obtained.

It is important to note that this chapter serves primarily as a repository for all results. Discussions and interpretations of the findings will follow in Chapter 7, with the relevant conclusions for the research questions drawn in Chapter 8.

6.1. Simulation

As mentioned previously, the main simulation results can be divided in the reaction wheel analysis, which serves as a benchmark, the thruster analysis, and the comparative study between them. Before diving into the reaction wheel system, it should be clear that different gain values need to be used for the two analyses. The reason for this difference is the fact that the dynamic equations are slightly different for both analyses, and the required control is therefore also different for each iteration time step.

For this reason, the control gain vectors $\boldsymbol{K}_s$, $\boldsymbol{K}_p$ and $\boldsymbol{K}_d$ should be determined independently for the two analyses. No distinction is made between the gains for the different Cartesian axes; in recent research ([12] and [26]) this distinction was also not made since it did not improve the algorithm to any significant amount. If the three axes were to be governed by completely different equations or systems, this distinction would be relevant.

6.1.1. Reaction Wheel Analysis

The analysis will start with the determination of the gains for the simulation. For the proportional and derivative gains, an estimation of their order of magnitude could be made based on Equation 4.13. The natural angular velocity of the system, for the highest torques predicted (order of magnitude of 10^{-6} $[Nm]$ based on disturbance torque predictions) and considering the order of magnitude of the inertia matrices (10^{-2} $[kgm^2]$), is approximately 0.01 $[rad/s]$. Therefore, the first approximation towards the proportional gain will be 0.0002 and for the derivative gain 0.02. Note that a critically-damped system is considered, implying ζ to be equal to 1. The quaternion error, for a small time period during the first fly-by in the simulation, is presented in Figure 6.1 in which these first gain values were used, with a speed gain equal to 1.

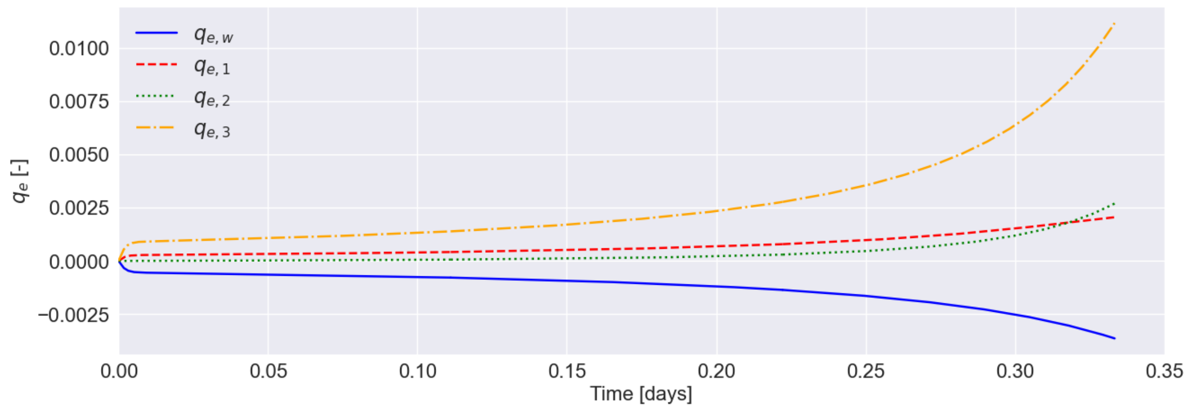


Figure 6.1: q_e development over time for the four quaternion components, with proportional gain set to 0.0002, derivative gain set to 0.02 and speed gain set to 1. Simulation over time period [00:00:00 05-01-2023] to [08:00:00 05-01-2023].

From this point onward, slight adjustments of the proportional gain and derivative gain, in combination with setting different values for the speed gain, resulted in different convergence performance and quaternion error magnitudes. The gain tuning method used was therefore the manual tuning, with the help of visual inspection of the quaternion error over time. It is understood that for real spacecraft, gains may be computed and uploaded before mission deployment, so that manual tuning is still possible. In most cases, however, a spacecraft employs adaptive control and updates its control gains based on real-time measurements of the system's behaviour and changes in operating conditions. Sophisticated algorithms have been developed for this purpose. The final gains chosen for the reaction wheel analysis, were:

$$\begin{aligned}\boldsymbol{K}_p &= \begin{bmatrix}5\cdot10^{-2} & 5\cdot10^{-2} & 5\cdot10^{-2}\end{bmatrix}\\ \boldsymbol{K}_d &= \begin{bmatrix}5\cdot10^{-2} & 5\cdot10^{-2} & 5\cdot10^{-2}\end{bmatrix}\\ \boldsymbol{K}_s &= \begin{bmatrix}12 & 12 & 12\end{bmatrix}\end{aligned} \tag{6.1}$$

With these new gains found, the quaternion error over time, for the same time span as was seen previously, is presented in Figure 6.3. It should be noted that the magnitude of the error has significantly improved (100×) and that further increase of the gains only led to unstable system response. To determine whether these errors are acceptable, a benchmark was established based on requirement **ADCS-01** as seen in Table 2.3 from the LUMIO mission. Figure 6.2 shows the half-cone angle β between the LUMIO-Cam and the Moon pointing vector for the entire simulation as will be presented in the rest of this section. As mentioned previously, the LUMIO-Cam was assumed to be pointing out of the negative y-panel of the spacecraft bus. It can be seen that the maximum offset is approximately 0.055[°] and therefore well within the limit of 0.18[°]. The final results indicate that the system meets this benchmark, confirming the adequacy of the selected gains.

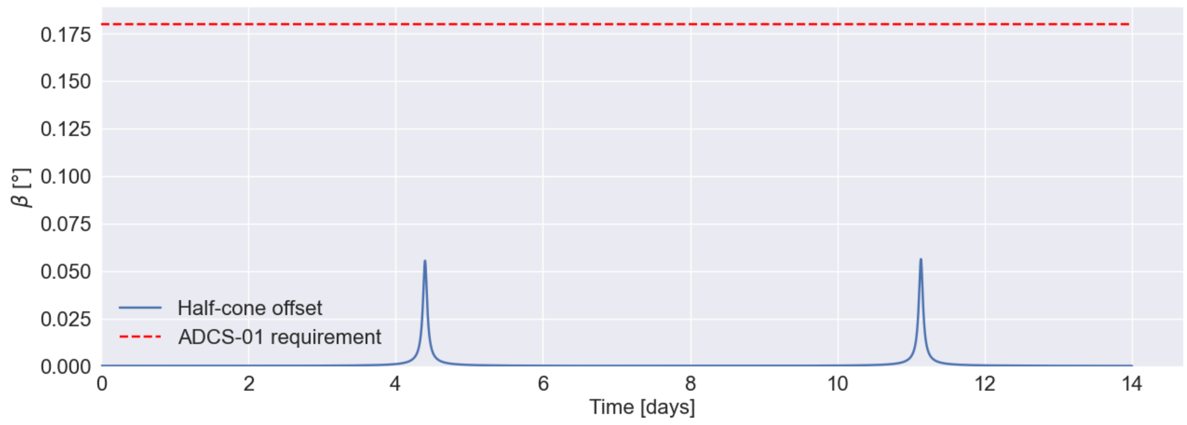


Figure 6.2: Reaction wheels: β, measured between the negative y-panel normal vector and the Moon pointing vector and presented in absolute numbers, over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

It should also be noted that further increases in the gains led to an unstable system response. For this study, "unstable system response" is defined as oscillations in the half-cone offset angle β that exceed the 0.18[°] threshold for longer than 60 seconds. Such behaviour would compromise the stability and reliability of the system, particularly during critical mission phases, and is therefore deemed unacceptable. The gains selected achieve a balance between error reduction and system stability, making them the most suitable choice for the simulation.

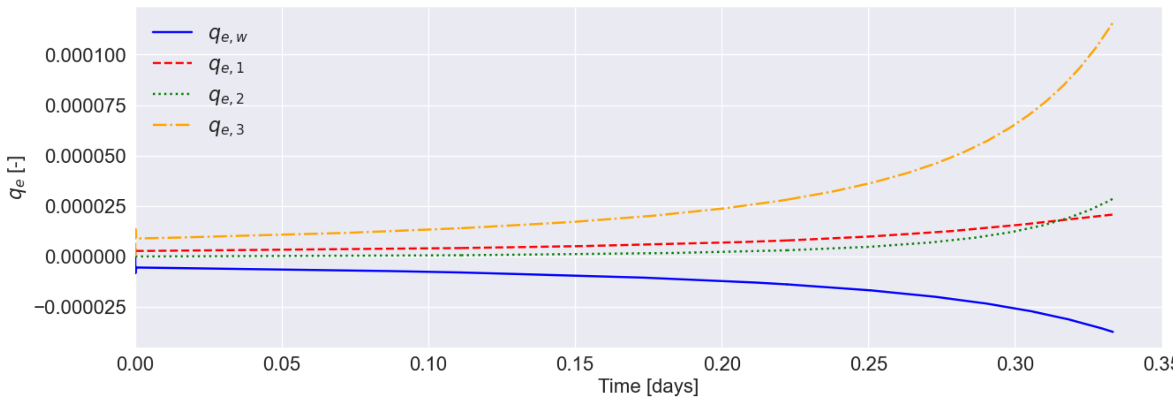


Figure 6.3: q_e development over time for the four quaternion components, with proportional gain set to 0.05, derivative gain set to 0.05 and speed gain set to 10. Simulation over time period [00:00:00 05-01-2023] to [08:00:00 05-01-2023].

Now, the simulation results can be presented. In the reaction wheel section, all relevant results will be shown in order to obtain an idea of the order of magnitude of the variables. First of all, q_w, q_1, q_2 and q_3 are presented over time in Figure 6.4 to Figure 6.7. In these graphs, the commanded quaternion for each component is also presented and it can be seen that due to the adequate PD control settings, the attitude is followed accurately.

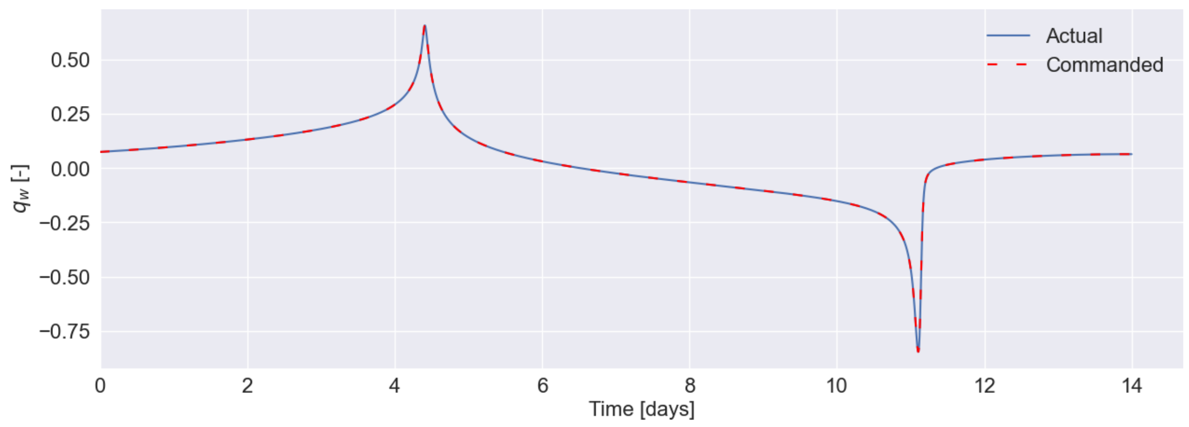


Figure 6.4: Reaction wheels: q_w versus the reference quaternion over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

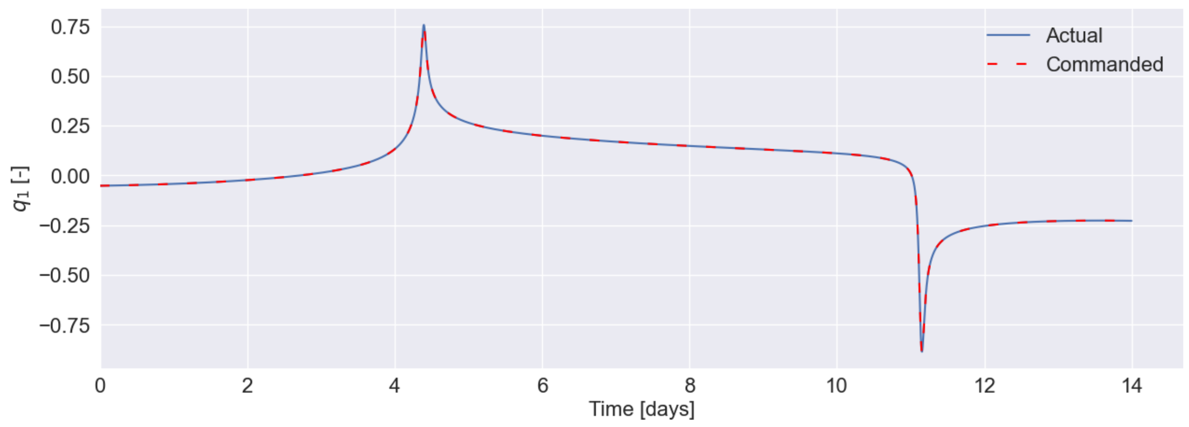


Figure 6.5: Reaction wheels: q_1 versus the reference quaternion over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

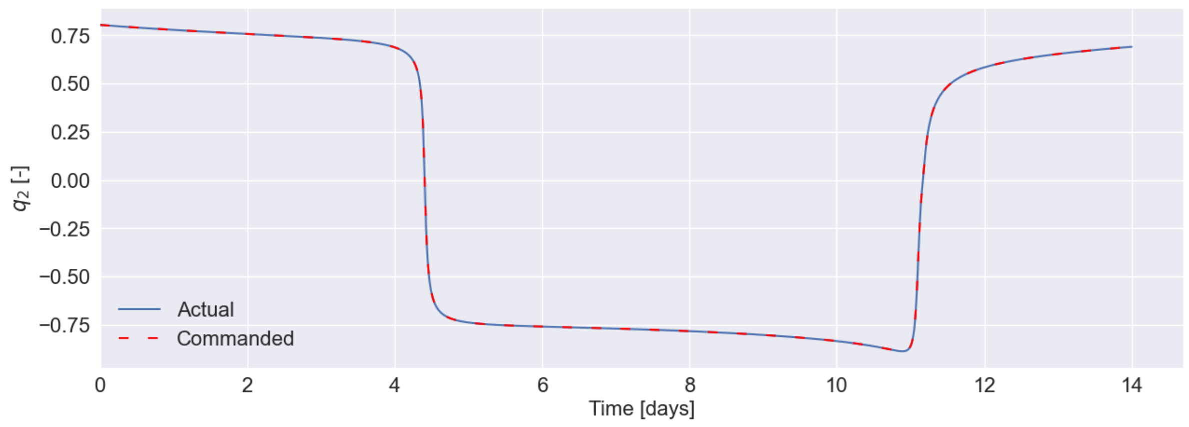


Figure 6.6: Reaction wheels: q_2 versus the reference quaternion over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

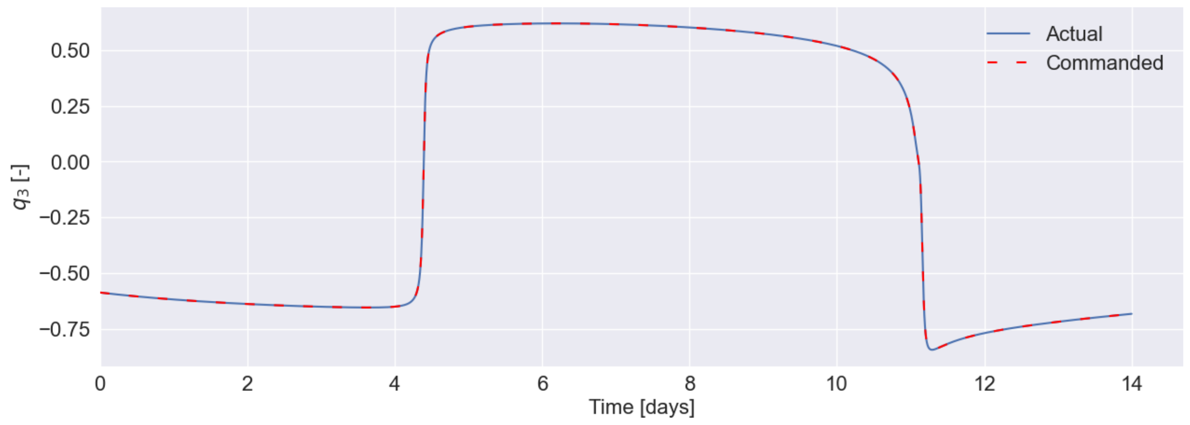


Figure 6.7: Reaction wheels: q_3 versus the reference quaternion over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

In order to have a closer look to the performance of the quaternions over time, Figure 6.8 shows the absolute quaternion error over time for the reaction wheel analysis, for each component individually. The relative difference with respect to the current value of each of the components is shown in Figure 6.9, in which the y-axis shows the absolute percentual offsets.

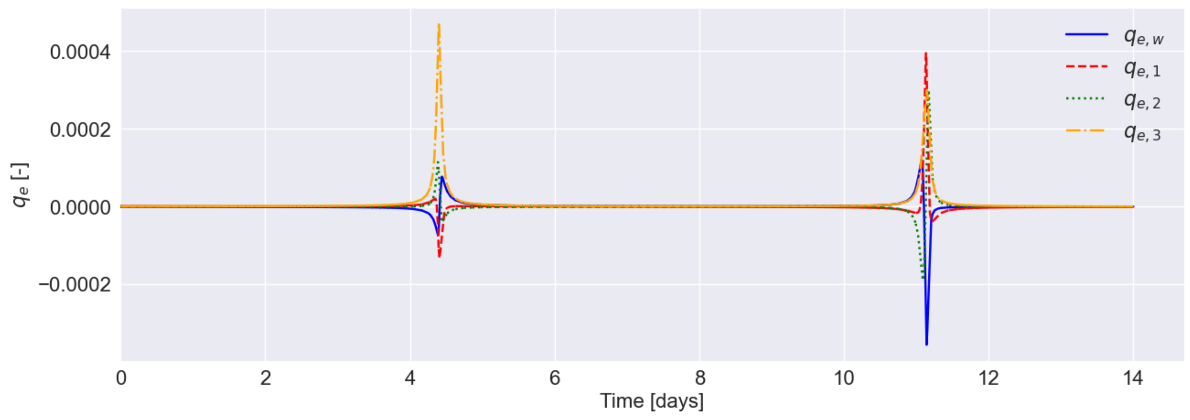


Figure 6.8: Reaction wheels: q_e (absolute) over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

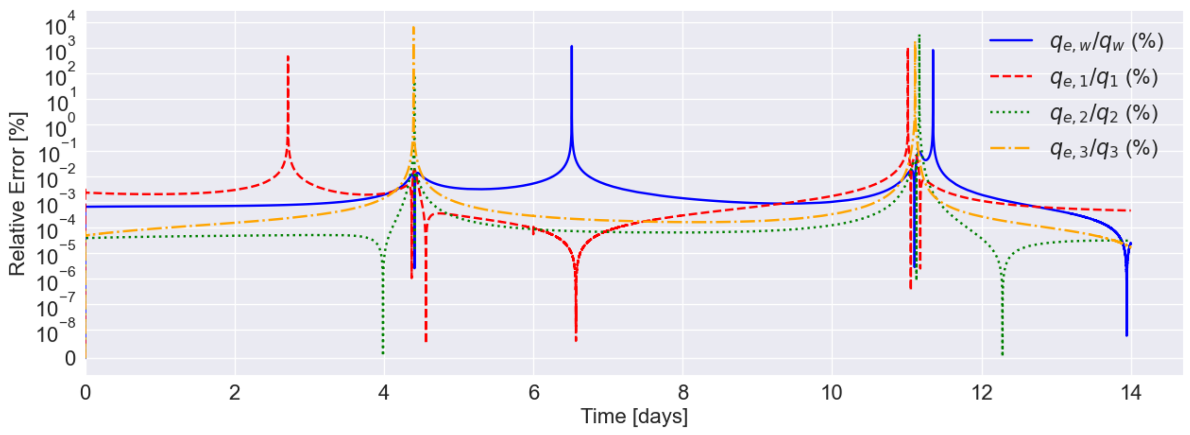


Figure 6.9: Reaction wheels: q_e (relative in %) over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

It can be observed that the relative error in the quaternions remains below 0.1% throughout the simulation, and shows a number of peaks in the areas where the quaternions themselves change rapidly. These rapid changes are attributable to the aforementioned close fly-bys of the Moon, seen around 4.25 days and 11 days. Moreover, relative offset peaks for q_1 and q_w can be observed around 3 days and 6.5 days as seen from epoch. This relative peak has to do with the switching of signs of both quaternions, as can be seen in Figure 6.4 and Figure 6.5. Although one may suspect these quaternion error peaks to last for a duration longer than a minute, it can be seen from Figure 6.10 that this only covers several seconds, during which the sign changes or rapid attitude corrections take place. Only the evident peak for the relative error of q_3 has been shown here, but the same behaviour was found for the other quaternions. For this reason, the peaks are considered to be insignificant.

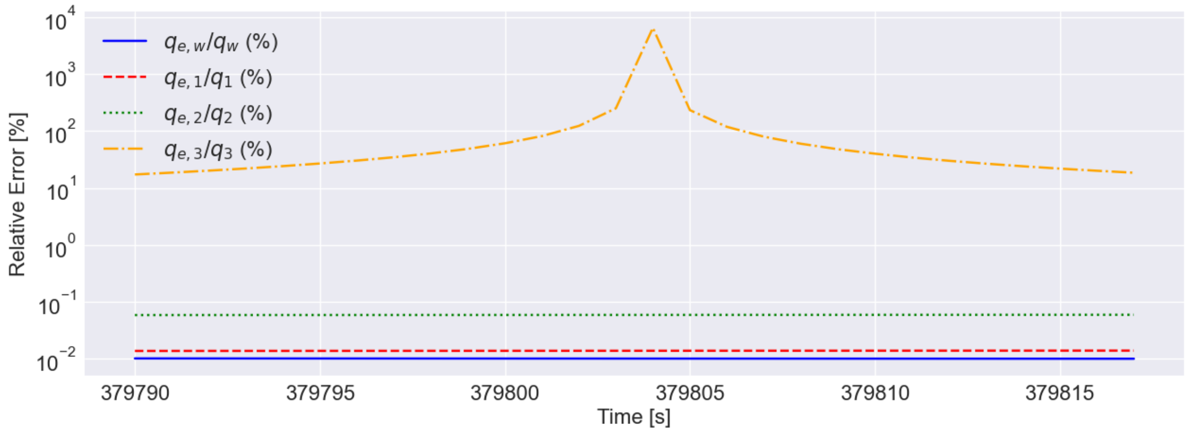


Figure 6.10: Reaction wheels: q_e (relative in %) over time in days, analysed from [05:32:10 05-01-2023] until [05:32:38 05-01-2023].

The next reaction wheel result to be observed is the angular velocity of the spacecraft, divided over its three primary axes, as shown in Figure 6.11. The maximum angular velocity experienced is approximately 0.00045 [rad/s], equal to approximately 0.0258 [$°/s$]. This was also experienced during the close fly-by manoeuvrers, and for the rest of the simulation, the angular velocity remains approximately zero, which already indicates good stability. The peaks indicated in the figure are necessary for the adjustment of the attitude during the close fly-bys, and therefore do not pose a risk for the spacecraft. In addition, these values remain below the slew rate given in requirement **ADCS-03** from the LUMIO mission, which is equal to 0.5 [$°/s$].

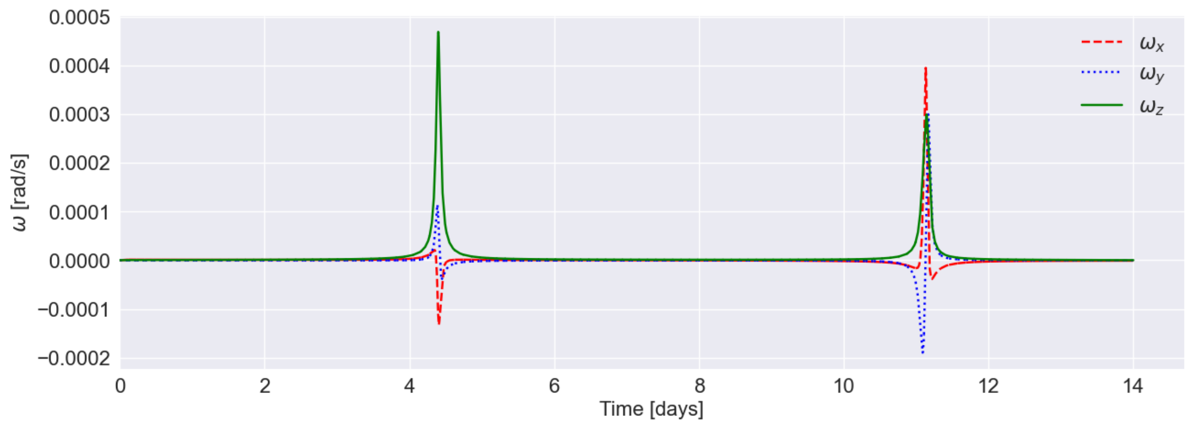


Figure 6.11: Reaction wheels: ω for each of the spacecraft primary axes over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

The external disturbance torques, as well as the required control torque at each moment in time, can also be visualised for each of the primary axes. In Figure 6.12, the gravity gradient torque exerted on the spacecraft is shown for the full two-week simulation. It can be seen that the order of magnitude of this torque is 10^{-10} and it becomes only significant with the close fly-bys again.

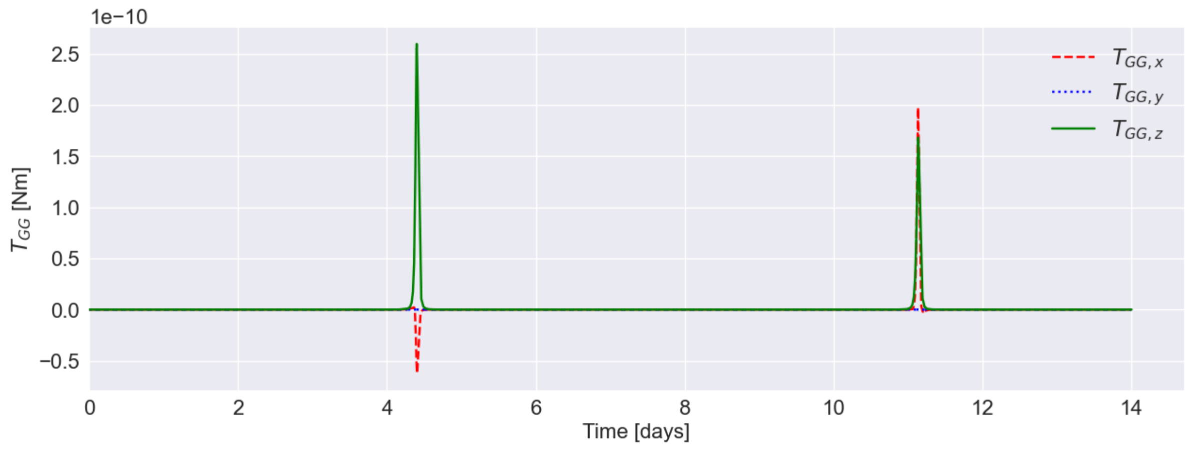


Figure 6.12: Reaction wheels: T_{GG} for each of the spacecraft primary axes over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

In Figure 6.13, the solar radiation pressure torque exerted on the spacecraft is shown. It can be seen that its order of magnitude is only 10^{-18}, making it considerably insignificant with respect to the gravity gradient torque and to the attitude control system as a whole. For this reason, a combined disturbance torque graph is omitted since the solar radiation pressure torque will not give significant rise to the gravity gradient torque.

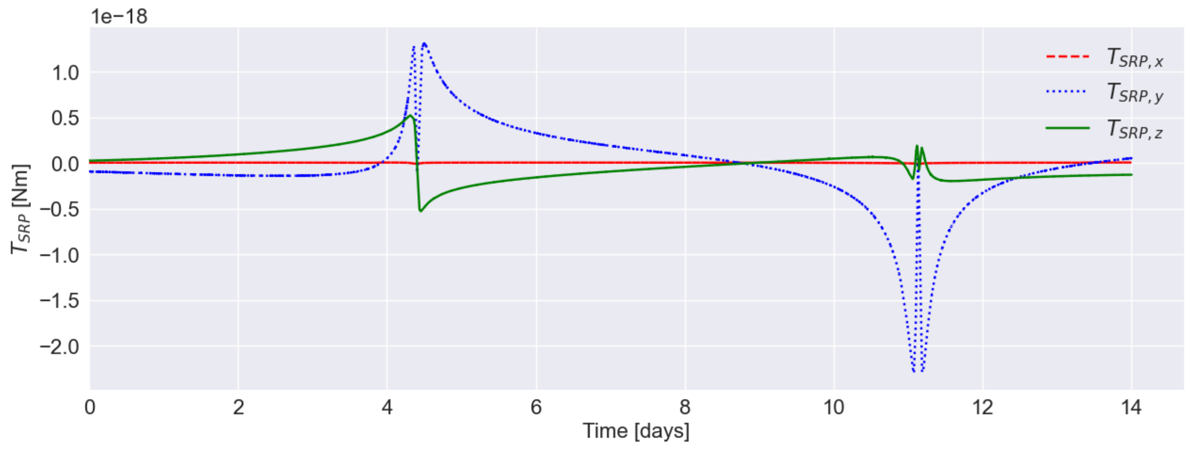


Figure 6.13: Reaction wheels: T_{SRP} for each of the spacecraft primary axes over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

Next, the control torque necessary to attain the reference as seen in the quaternion performance is displayed in Figure 6.14 and was well within the limits of 0.007 $[Nm]$ possible by the reaction wheels. The maximum required torque by one of the primary axis is experienced over the x-axis (roll axis) and has a magnitude of approximately $1.3 \cdot 10^{-7}$ $[Nm]$. It can be observed that this is about an order 10^3 larger than the maximum experienced disturbance torque. Moreover, a required torque in the first few iterations of the simulation is observed.

An important observation that should already be mentioned, is the fact that the required control torques are an order 10^4 smaller than the capabilities of the used reaction wheels. Since no direct information on the resolution of the Blue Canyon reaction wheels as used in this research is publicly available, no conclusion can yet be drawn on whether they can be used for the analysis in this research. As a first approximation, usual reaction wheel resolution can go up to 1% of the torque range, which would mean a minimum torque of $7.0 \cdot 10^{-5}$ $[Nm]$ would be possible, disqualifying the reaction wheels for this analysis. In order to achieve the goals of this research, the reaction wheel analysis is assumed to be feasible. Further discussion will be provided in Chapter 7.

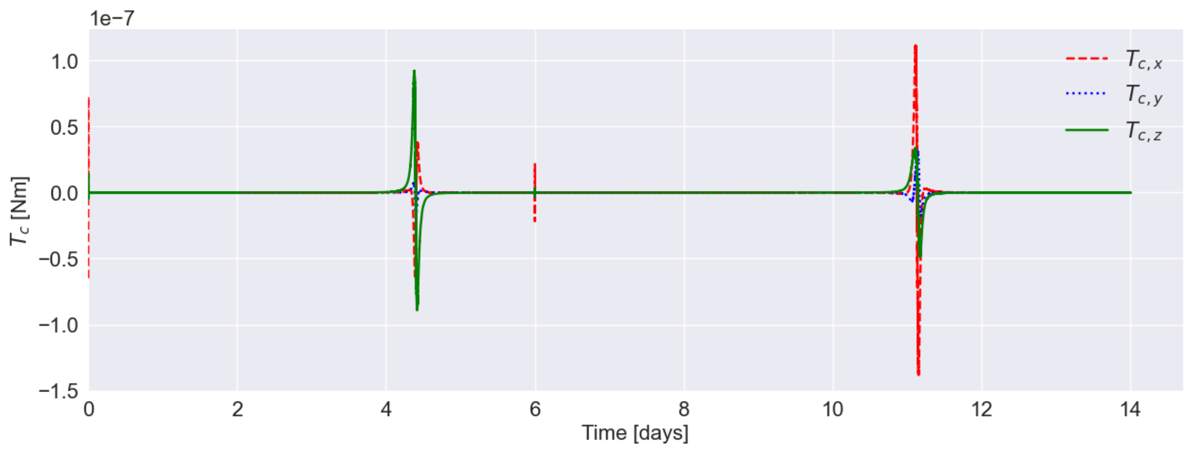


Figure 6.14: Reaction wheels: T_c for each of the spacecraft primary axes over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

In order to have a closer inspection of the close fly-bys and their effect on the control torque needed for the system, Figure 6.15 and Figure 6.16 are displayed below. The variation of the torque per axis can be observed more clearly in this way.

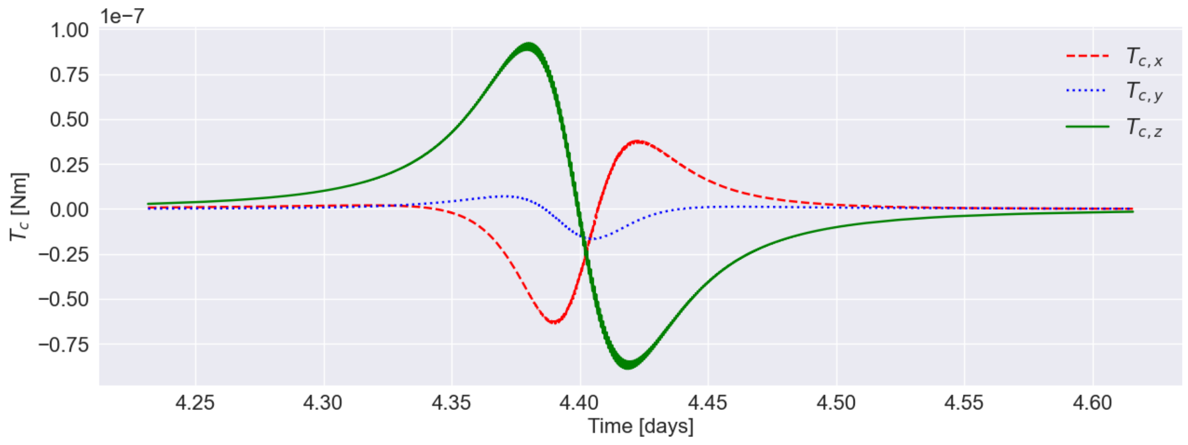


Figure 6.15: Reaction wheels: T_c for each of the spacecraft primary axes over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

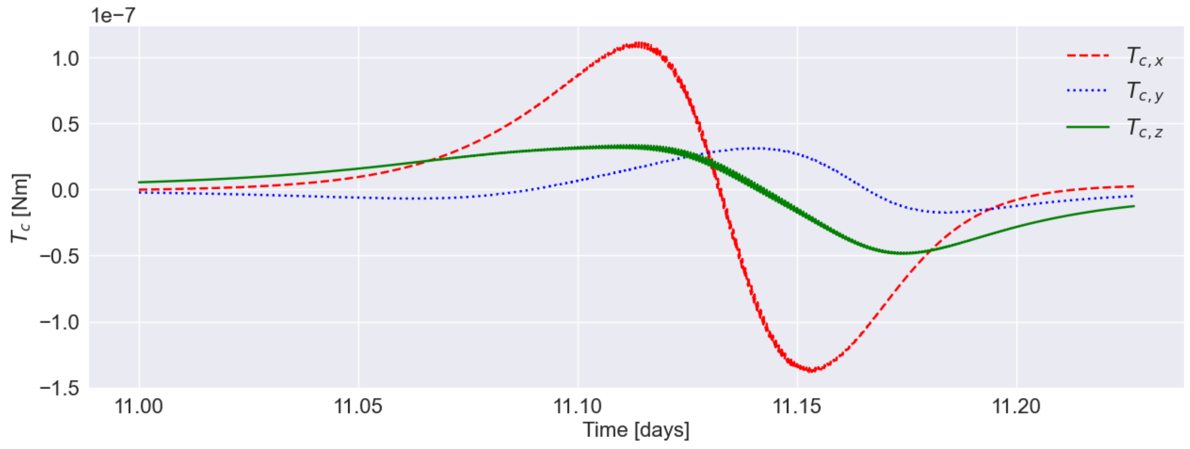


Figure 6.16: Reaction wheels: T_c for each of the spacecraft primary axes over time in days, analysed from [00:00:00 12-01-2023] until [05:26:40 12-01-2023].

All the results as shown above will be approximately similar for the thruster analysis. As was mentioned before, the dynamics that constitute these results are primarily based on the control algorithm with its underlying dynamic equations, and these equations only differ in the fact that the total angular momentum of the spacecraft is supplemented by the reaction wheels. Therefore, for the reaction-wheel-specific results below, a closer look will also be given to the angular momentum term the wheels introduce, and a conclusion shall be drawn on whether this has a significant influence for the control algorithm results as shown above. With this conclusion drawn, the thruster analysis results can efficiently be pre-selected.

From the required control torque, the necessary torque per wheel could be computed and their results are presented in Figure 6.17. It can be seen that reaction wheel 3, aligned with the z-axis, exerts the highest torques around the first fly-by, and this goes for reaction wheel 1 during the second fly-by. In order to more thoroughly inspect the graph, the zoomed-in versions are shown in Figure 6.18 and Figure 6.19, for the same time period as shown previously. Note that the control torque, as well as logically the reaction wheel torques, show required values around day 6. This is due to the switching of signs again, as was seen previously in the quaternion error as well.

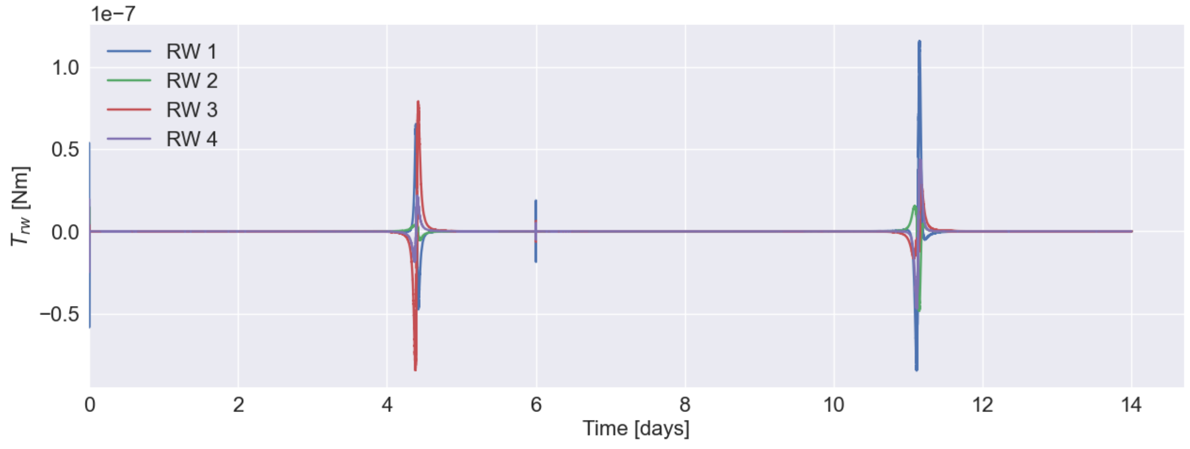


Figure 6.17: Reaction wheels: T_{rw} for each of the four reaction wheels over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

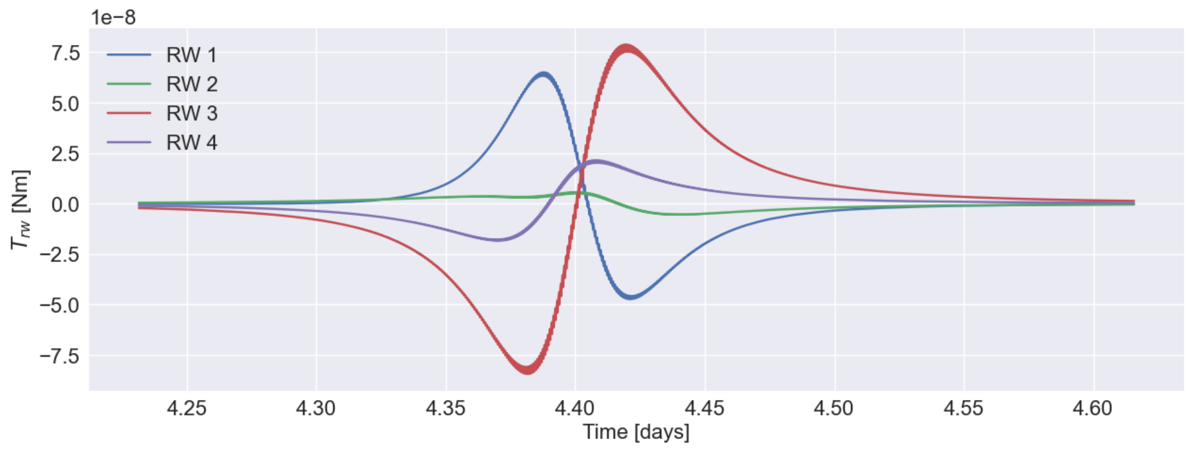


Figure 6.18: Reaction wheels: T_{rw} for each of the four reaction wheels over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

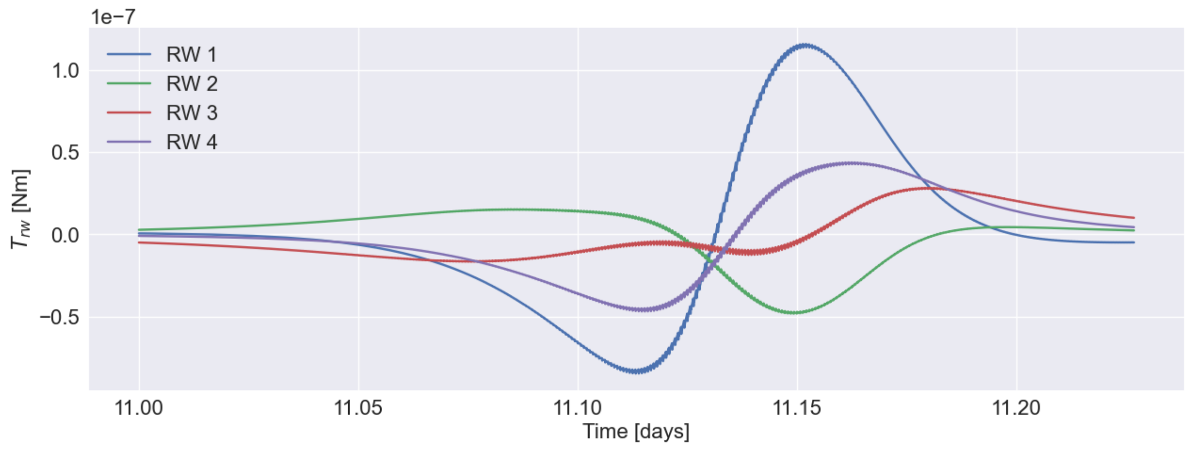


Figure 6.19: Reaction wheels: T_{rw} for each of the four reaction wheels over time in days, analysed from [00:00:00 12-01-2023] until [05:26:40 12-01-2023].

Based on the torques exerted per reaction wheel, the power required per wheel could also be analysed. A linear relationship between the power required by the reaction wheels and the torque exerted was assumed in this research. In addition, the integration of power over time gives the energy in Joules that is consumed by the wheels in total. Note that this energy increases over time, which is expected. In addition, the total power and energy consumption of the wheel is shown. This information is presented in Figure 6.20 and Figure 6.21. Finally, the power graphs have been posted more closely in Figure 6.22 and Figure 6.23 as well, from which it is again evident that reaction wheel 3 requires most power during the first fly-by and reaction wheel 1 during the second fly-by.

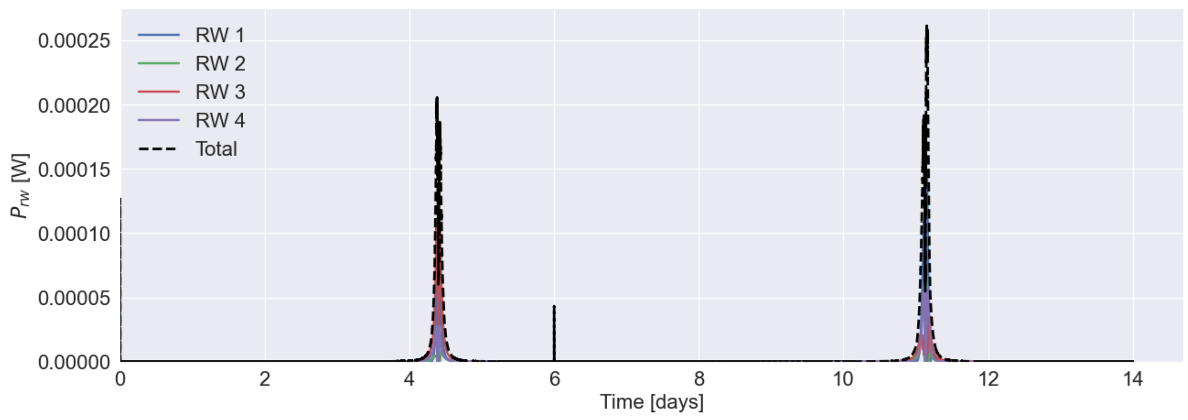


Figure 6.20: Reaction wheels: P_{rw} for each of the four reaction wheels over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

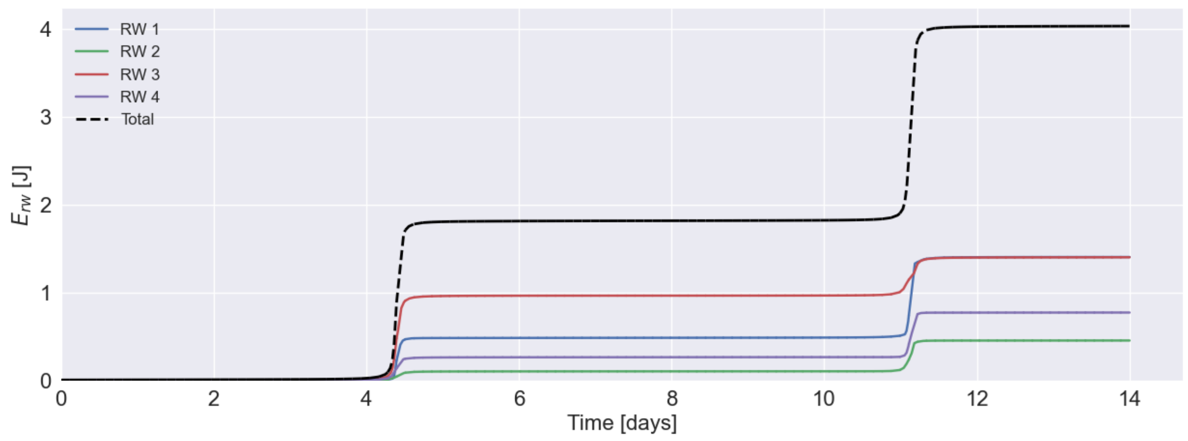


Figure 6.21: Reaction wheels: E_{rw} for each of the four reaction wheels over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

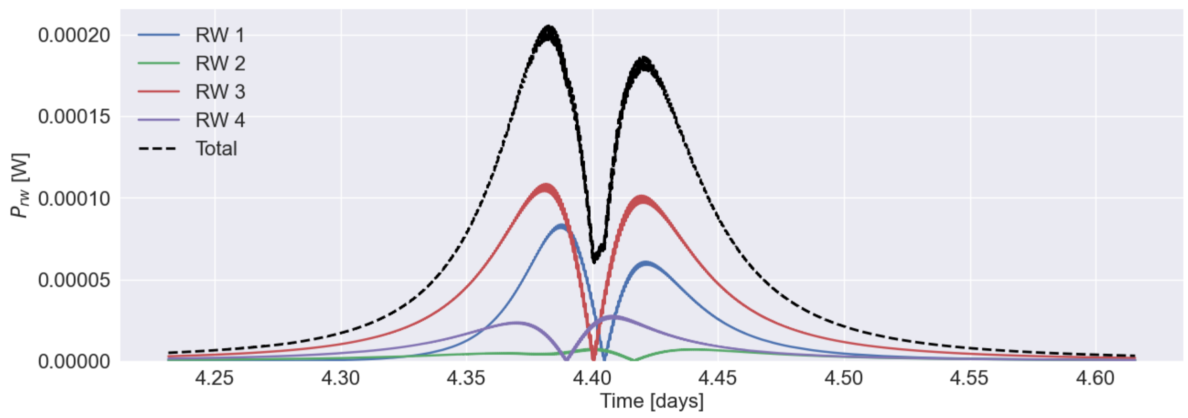


Figure 6.22: Reaction wheels: P_{rw} for each of the four reaction wheels over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

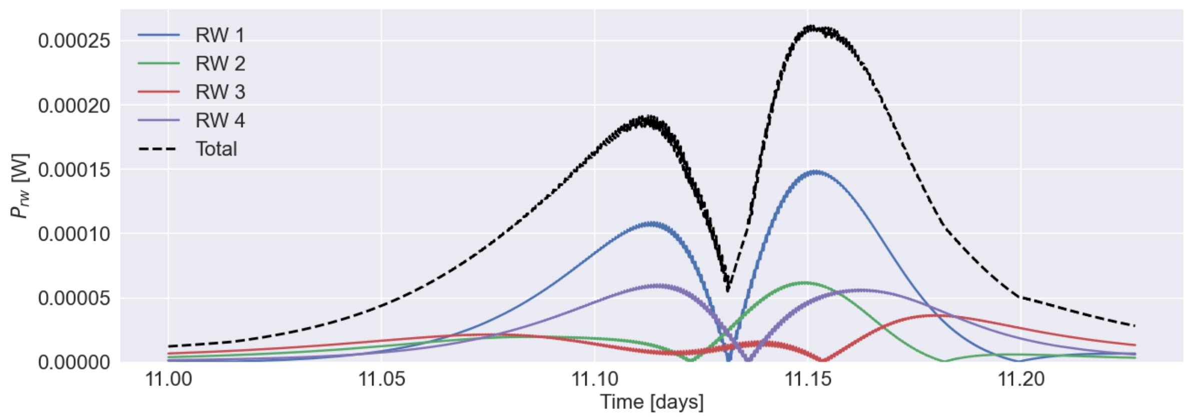


Figure 6.23: Reaction wheels: P_{rw} for each of the four reaction wheels over time in days, analysed from [00:00:00 12-01-2023] until [05:26:40 12-01-2023].

From above figures, it can already be seen that the total power requirement for this reaction wheel system is approximately 0.00025 $[W]$ during the critical close fly-by maneuver. The question now arises whether additional actuators (thrusters) are needed for momentum wheel desaturation. In order to assess this, the angular momentum build-up in each of the wheels needs to be visualised and checked with the angular momentum limits of the wheels, which was set to be 0.1 $[Nms]$ for the first three reaction wheels and 0.05 $[Nms]$ for the fourth reaction wheel. The angular momentum build-ups are visible in Figure 6.24 and Figure 6.25 below, for each reaction wheel individually. It can be seen that the angular momentum for the wheels builds up around the close fly-by, but does not necessarily increase towards the end of the simulation. Under nominal conditions and without any external actuation, the angular momentum does not reach the limits of the wheels. However, non-nominal scenarios, such as emergency manoeuvrers or unexpected high reaction wheel usage, could lead to rapid saturation of the wheels, necessitating external momentum dumping.

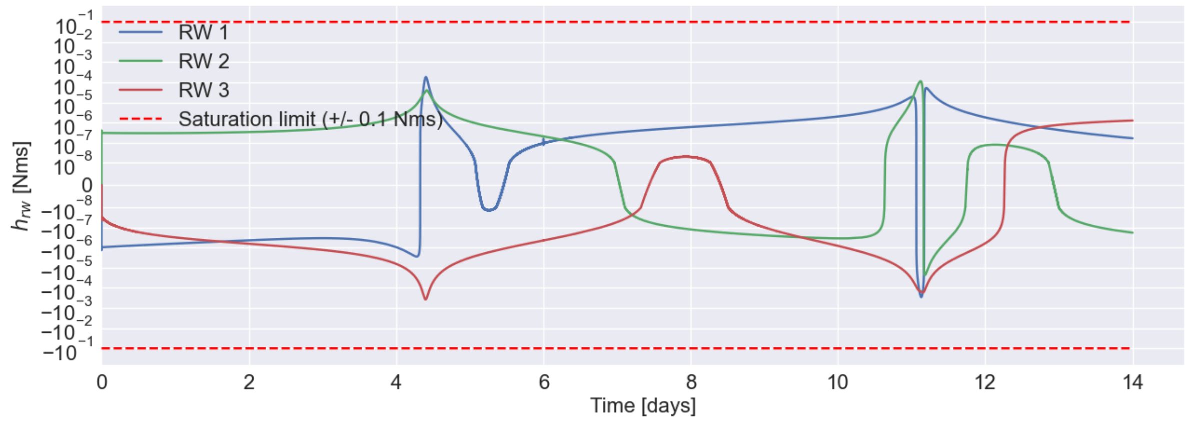


Figure 6.24: Reaction wheels: h_{rw} for reaction wheels 1, 2 and 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023]. Saturation limits indicated.

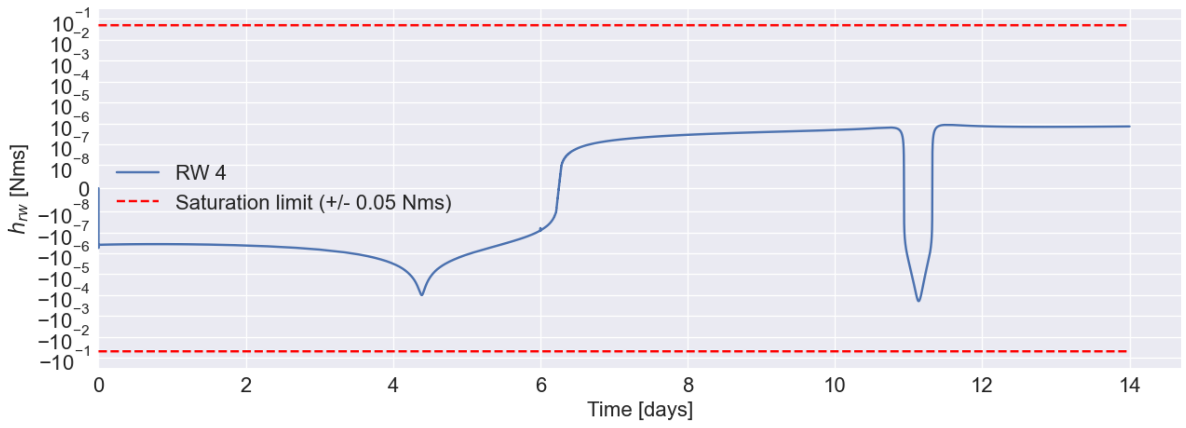


Figure 6.25: Reaction wheels: h_{rw} for reaction wheel 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023]. Saturation limits indicated.

In order to assess the significance of the reaction wheels on the overall spacecraft angular momentum throughout the simulation, the angular momentum vectors of the wheels can be summed to create the total angular momentum from the wheels $\boldsymbol{h}_{rw}$. Its evolution over time, for each of the primary spacecraft axes, can be seen in Figure 6.26 below. Evidently, the induced angular momentum on the spacecraft by the disturbance torques is fully compensated for by the reaction wheels, resulting in the desired zero angular momentum value for the spacecraft itself. This is also directly observed from the angular velocity results as seen in Figure 6.11, since angular momentum equal $I\omega$.

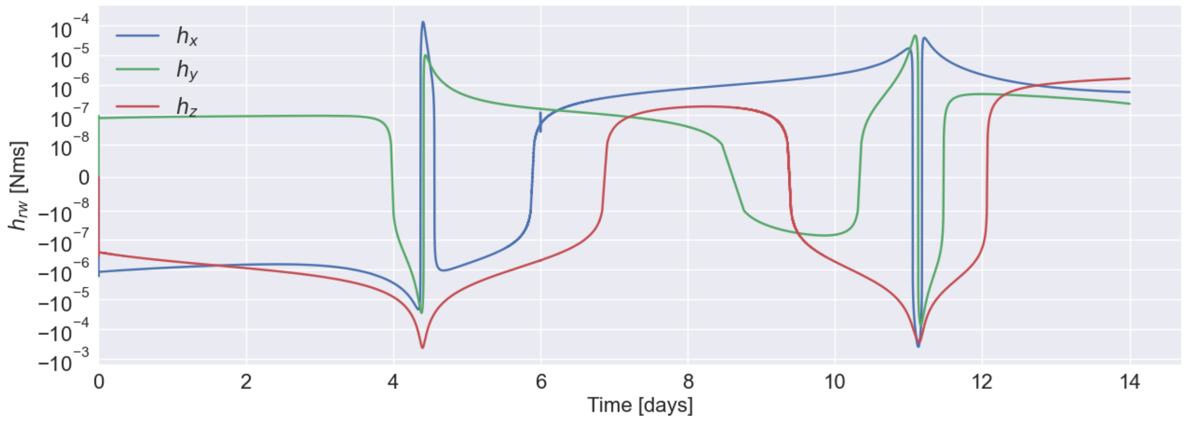


Figure 6.26: Reaction wheels: h_{rw} for each of the spacecraft primary axes over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

6.1.2. Thruster Analysis

The gain determination in the thruster analysis was conducted in the exact same way as was done for the reaction wheel analysis. Also, the first estimations for the gains were in the same order of magnitude, and the same estimations were therefore used. Based on the preliminary results obtained from the thruster analysis, in combination with the knowledge that only the angular momentum terms were different in the reaction wheel analysis, the same gain values were in the end used for both the reaction wheel and thruster analyses, as displayed in Equation 6.2.

$$\begin{aligned}\boldsymbol{K}_p &= \begin{bmatrix}5\cdot10^{-2} & 5\cdot10^{-2} & 5\cdot10^{-2}\end{bmatrix}\\ \boldsymbol{K}_d &= \begin{bmatrix}5\cdot10^{-2} & 5\cdot10^{-2} & 5\cdot10^{-2}\end{bmatrix}\\ \boldsymbol{K}_s &= \begin{bmatrix}12 & 12 & 12\end{bmatrix}\end{aligned} \tag{6.2}$$

This section will show the results for the thruster analysis per thruster configuration as proposed in

Chapter 4. Since the basic lay-out of the results has already been shown in subsection 6.1.1, this section will only include visualisations that are inherently different from the ones shown previously. In subsection 6.1.3, a comparison between the reaction wheels and thrusters will be given, and differences between, for example, the quaternion evolution over time will also be provided there. All other results not shown in this chapter are present in Appendix A.

6.1.2.1 Configuration 1

The quaternion behaviour for the systems is equal to the ones seen for the reaction wheel analysis. In Figure 6.27, the error quaternion evolution over time has been shown for visual confirmation of its equality to the reaction wheel analysis.

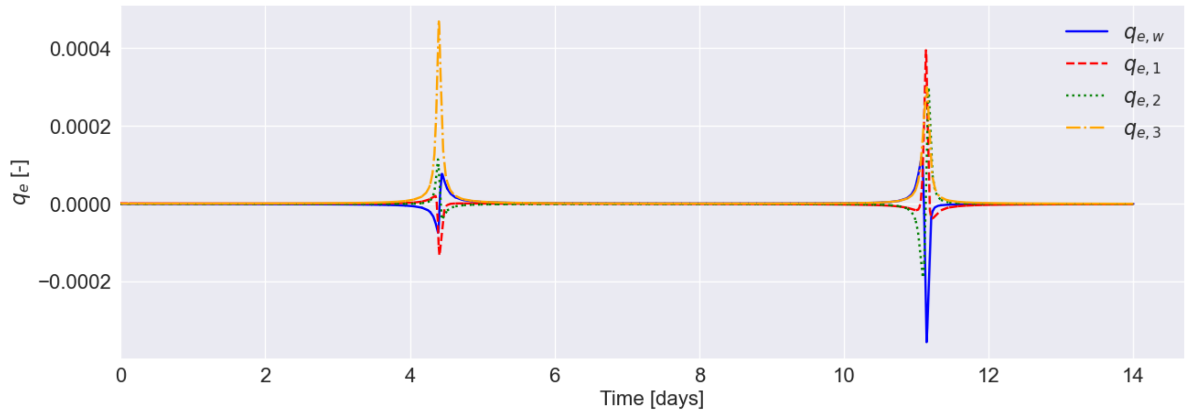


Figure 6.27: Thrusters: q_e (absolute) over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

Moreover, the environmental parameters in the analysis are equal to the reaction wheel case, because of which the visualisations for $\boldsymbol{T}_{GG}$ and $\boldsymbol{T}_{SRP}$ will not be shown here. Next, the control torque values display the exact same behaviour as well, and a comparative analysis between reaction wheel and thruster analysis will be given in subsection 6.1.3. Now, the thruster-specific visualisations can be examined. For configuration 1, six thrusters were present that all exhibit thruster firing during the simulation in order to comply with the required control torque values. Their thrust values over time have been displayed in Figure 6.28 to Figure 6.33 below.

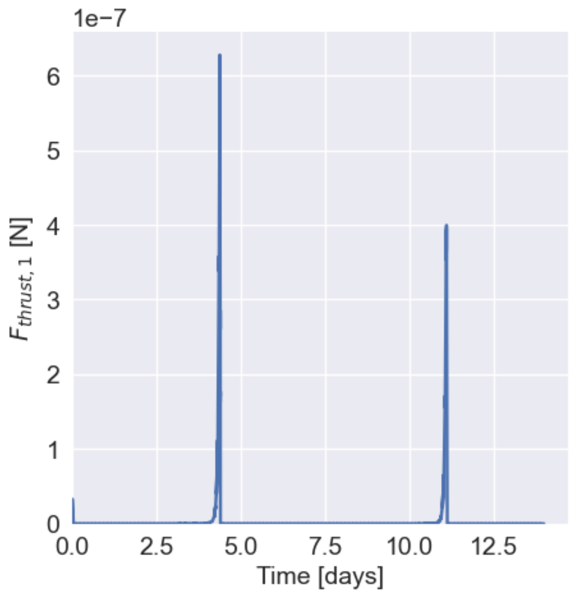


Figure 6.28: Thrusters: F_1 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

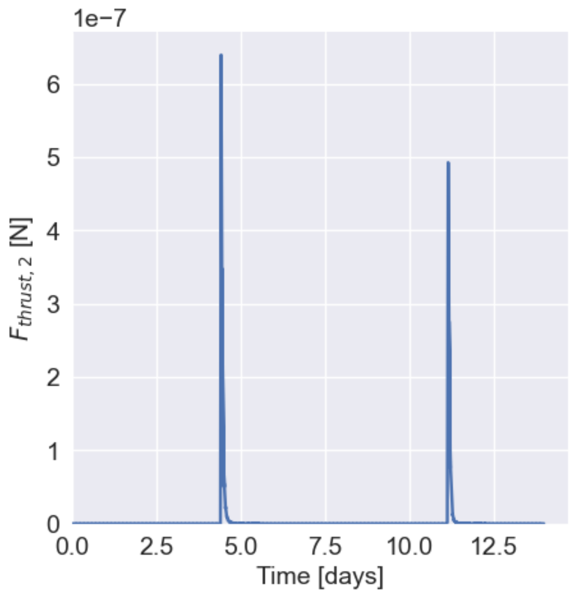


Figure 6.29: Thrusters: F_2 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

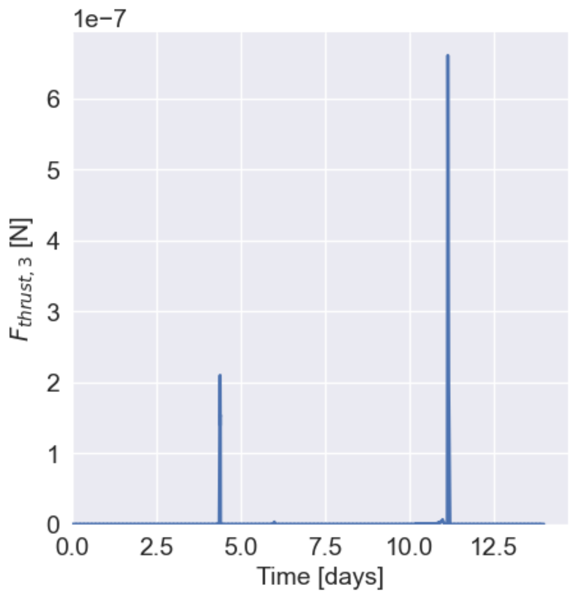


Figure 6.30: Thrusters: F_3 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

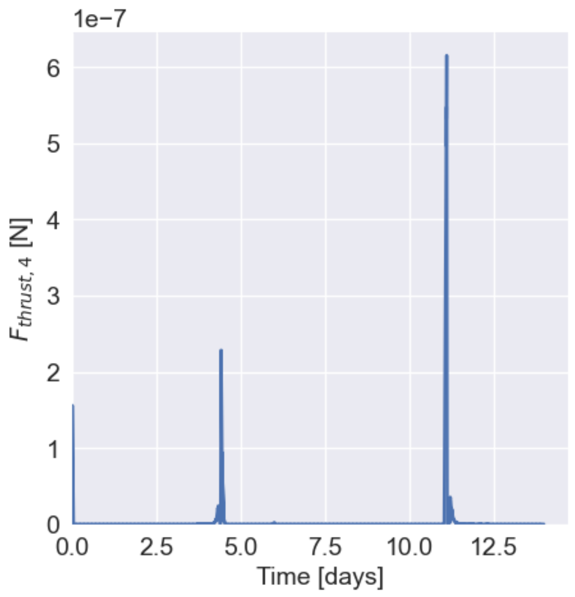


Figure 6.31: Thrusters: F_4 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

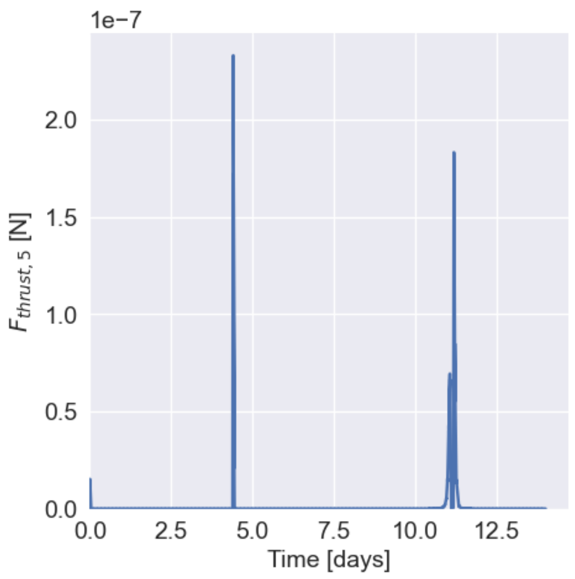


Figure 6.32: Thrusters: F_5 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

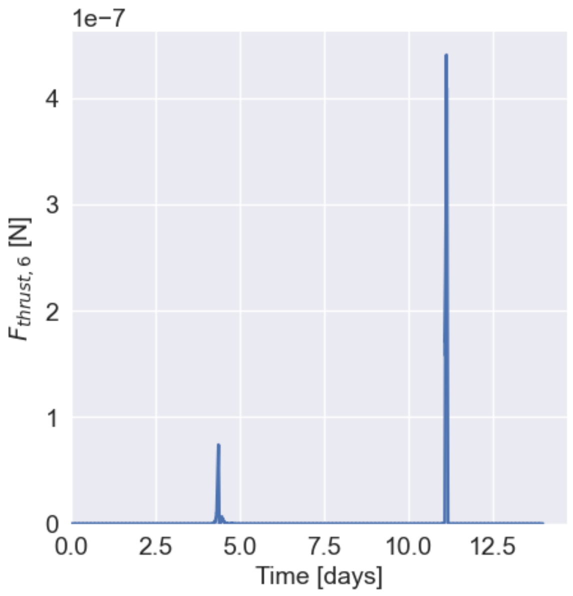


Figure 6.33: Thrusters: F_6 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

These plots can all be combined into one graph, shown in Figure 6.34. Closer examination of these plots is also shown, for the exact same time frames as was seen for the reaction wheel analysis. These can be seen in Figure 6.35 and Figure 6.36. From the graphs, it can be seen that the maximum thrust that is required for any of the thrusters is approximately $6.5 \cdot 10^{-7}$ $[N]$ or 0.65 $[\mu N]$, which is well within the limits of the maximum thrust delivered by the Pocket Rocket module in this research. Discussion with respect to the Minimum Impulse Bit will follow in Section 6.3. Also, it can be seen that the thruster fire behavior differs for each of the two fly-bys.

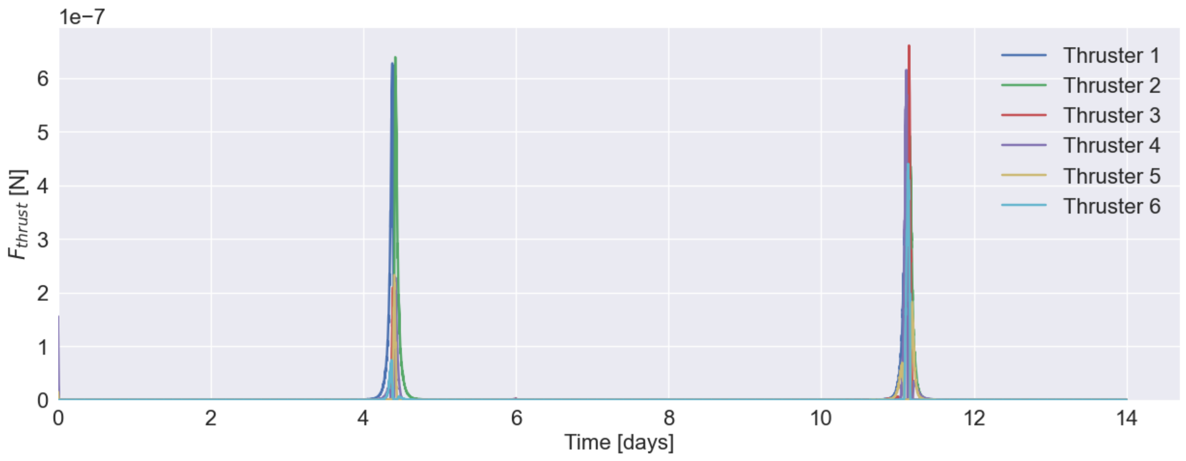


Figure 6.34: Thrusters: F output for all thrusters in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

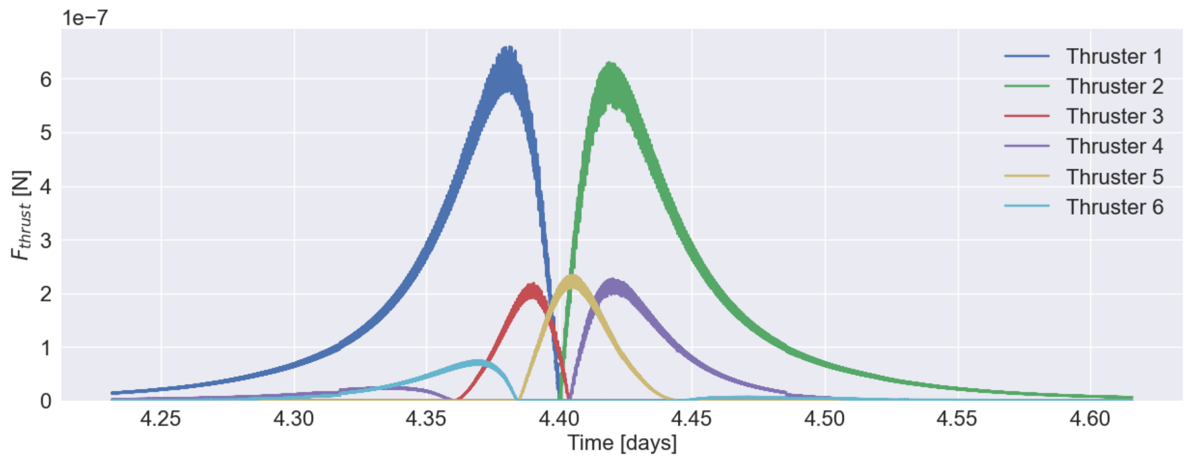


Figure 6.35: Thrusters: F output for all thrusters in configuration 1 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

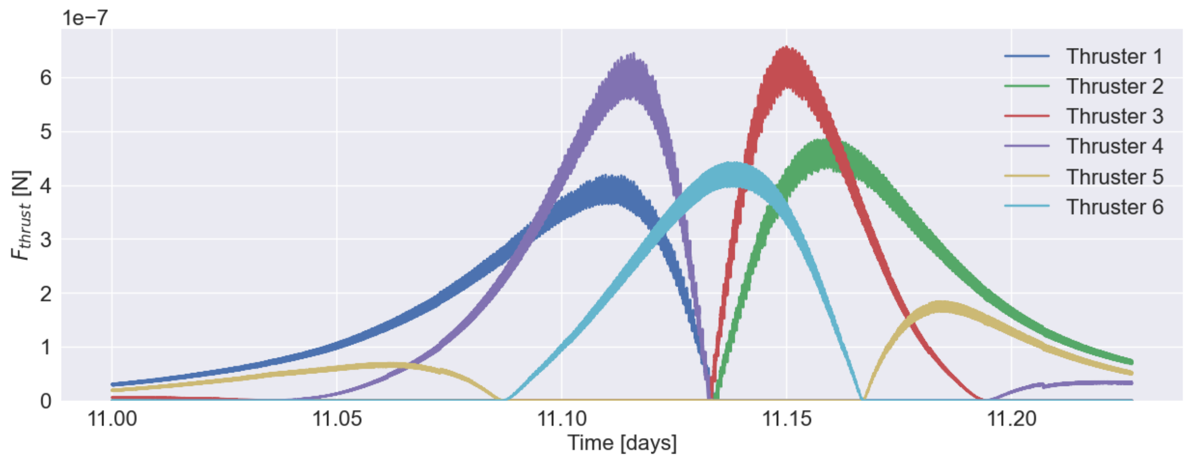


Figure 6.36: Thrusters: F output for all thrusters in configuration 1 over time in days, analysed from [00:00:00 12-01-2023] until [05:26:40 12-01-2023].

In addition to the thruster output values, the linear impulse of the thrusters (J_{thrust}) can be plotted over time and has been displayed in Figure 6.37. This graph provides a direct overview of the thrusters that have been fired for the longest duration.

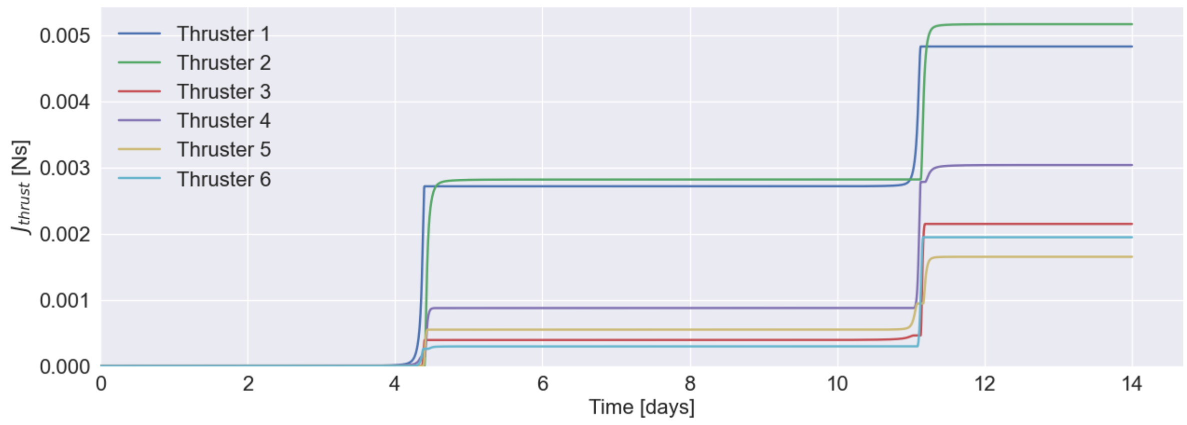


Figure 6.37: Thrusters: J_{thrust} output for all thrusters in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

Based on the thruster output graphs, the power and energy consumption per thruster could be computed and combined, as seen in Figure 6.38 and Figure 6.41. In addition, the closer examination plots are again shown in Figure 6.39 and Figure 6.40. It can be seen that the maximum total power necessary for the thruster setup is approximately 0.14$[W]$. The total energy consumed by the thrusters builds up towards approximately 1900 $[J]$ over the time frame analysed.

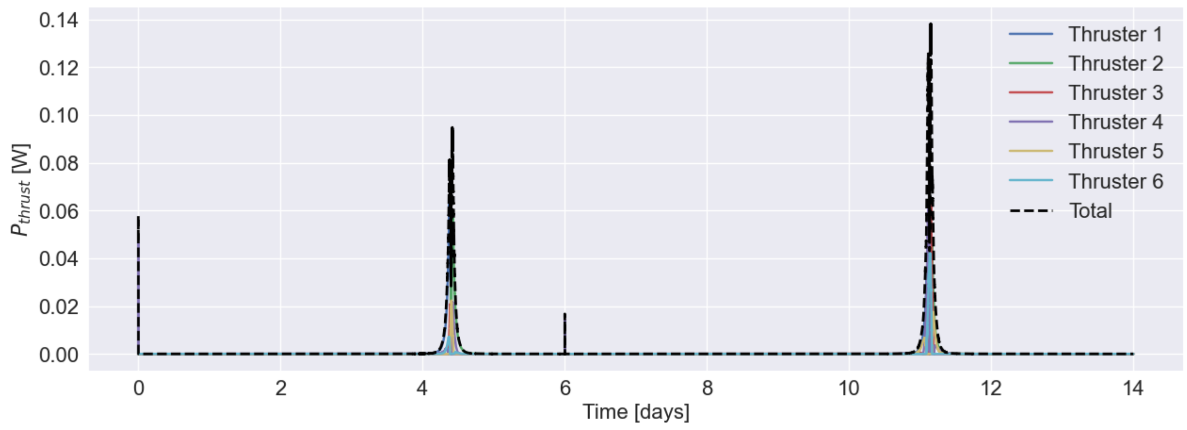


Figure 6.38: Thrusters: P required for all thrusters in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

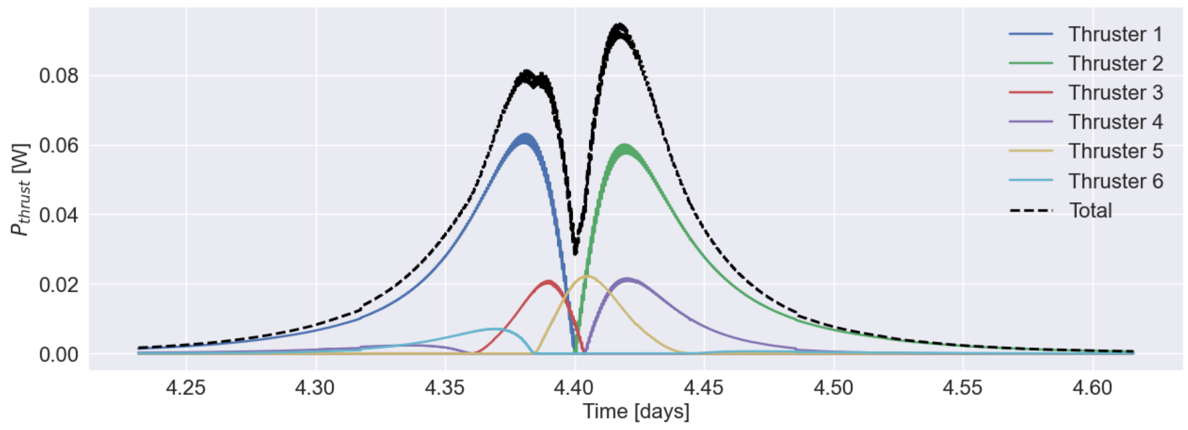


Figure 6.39: Thrusters: P required for all thrusters in configuration 1 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

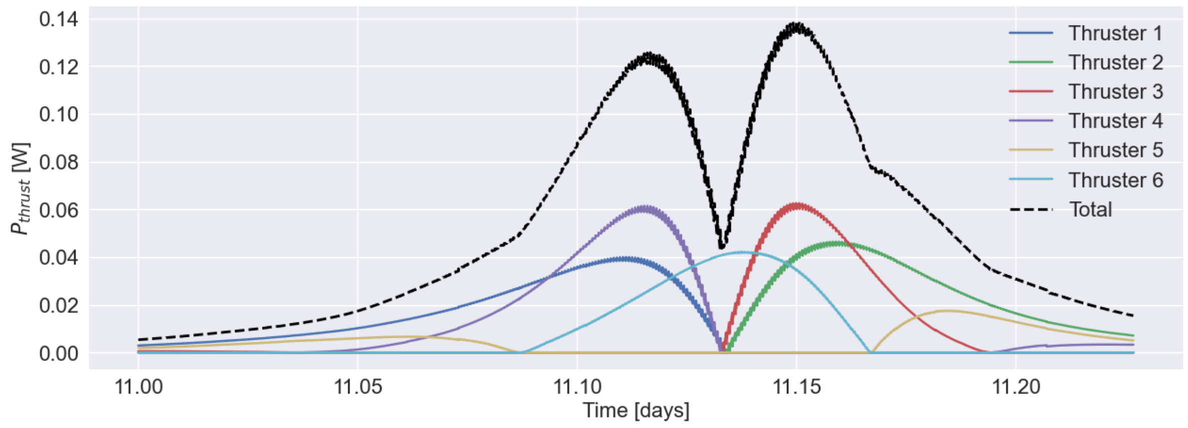


Figure 6.40: Thrusters: P required for all thrusters in configuration 1 over time in days, analysed from [00:00:00 12-01-2023] until [05:26:40 12-01-2023].

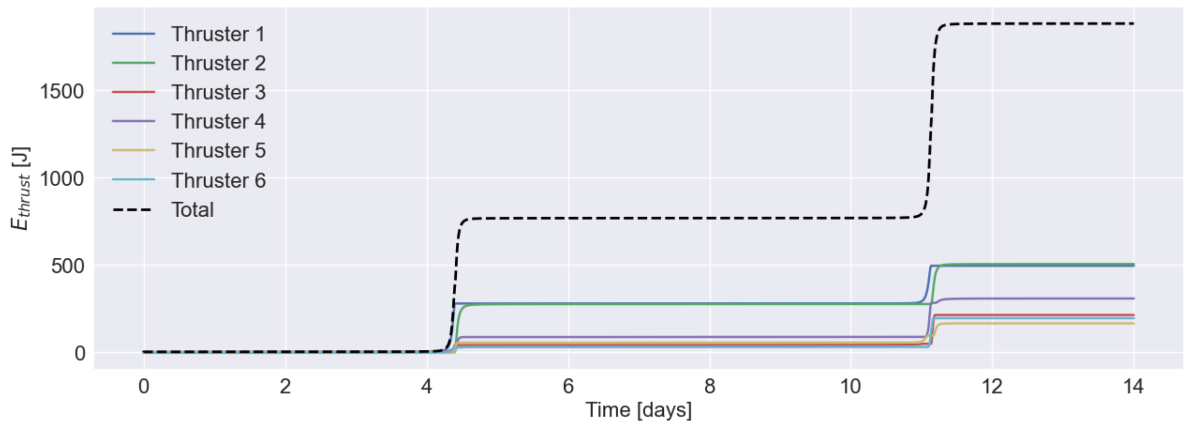


Figure 6.41: Thrusters: E required for all thrusters in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

The half-cone angle offset β will be equal to the offset shown in Figure 6.2 as the thruster configurations all perfectly match the control torques required, and because of the insignificance of the reaction wheel dynamics as seen before.

6.1.2.2 Configuration 2

For configuration 2, the thruster required thrust values are shown in Figure 6.42 and Figure 6.43 for the two fly-by maneuvers. At first glance, it can already be seen that overall, less thrust is needed for the attitude control maneuvers and that different thrusters are activated for this compared to configuration 1.

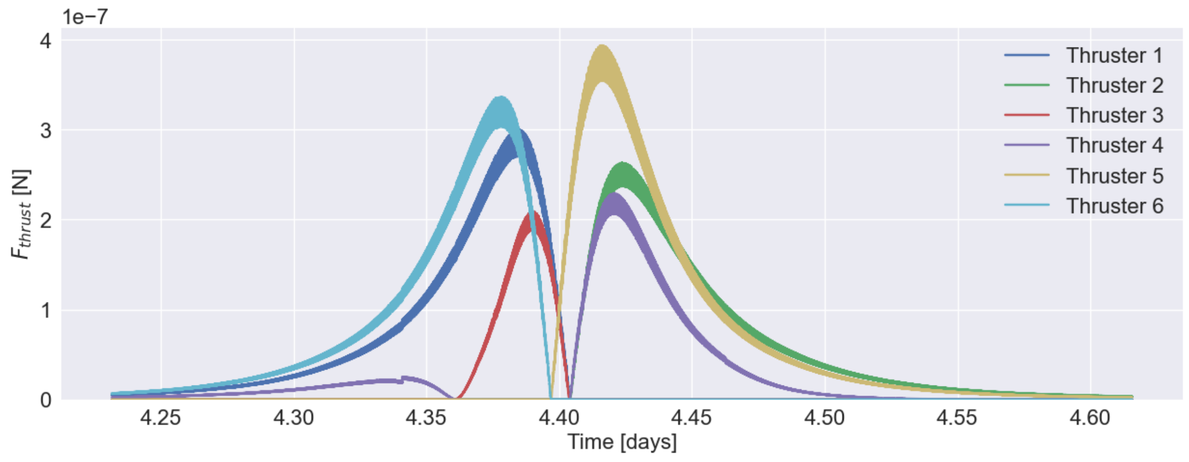


Figure 6.42: Thrusters: F output for all thrusters in configuration 2 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

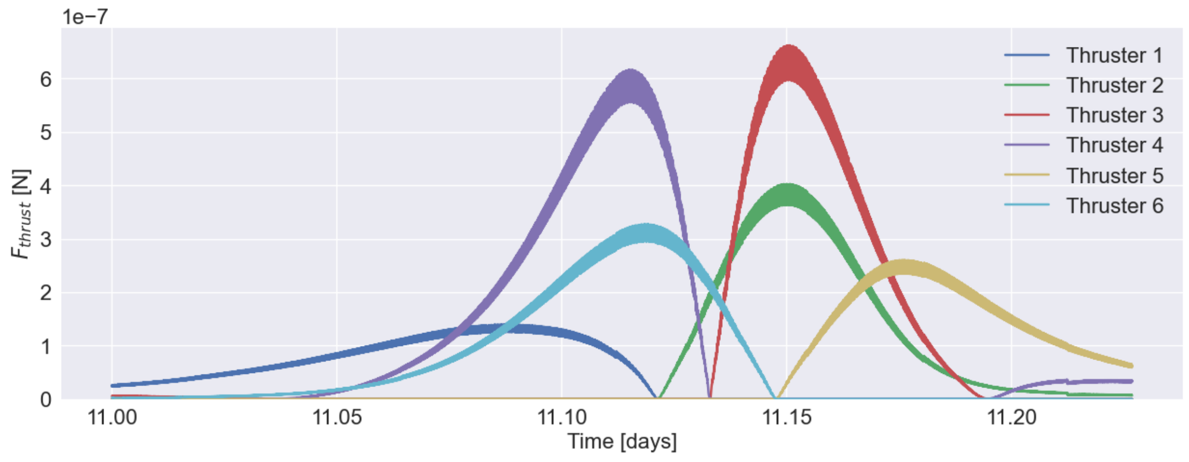


Figure 6.43: Thrusters: F output for all thrusters in configuration 2 over time in days, analysed from [00:00:00 12-01-2023] until [05:26:40 12-01-2023].

The total linear impulse of the thrusters for configuration 2 is shown in Figure 6.44. As seen before, these values increase over time and give a representation of the usage of each of the thrusters during the simulation period.

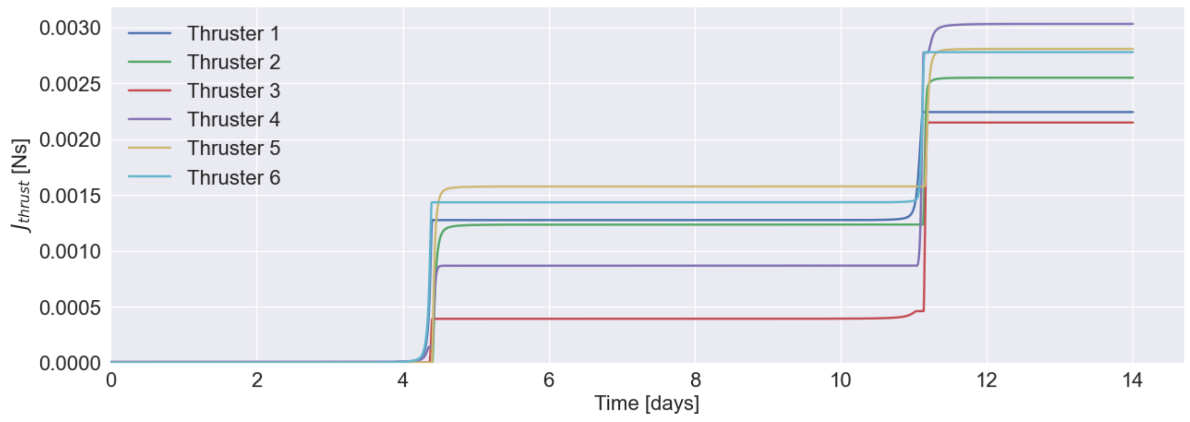


Figure 6.44: Thrusters: J_{thrust} output for all thrusters in configuration 2 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

In line with the thrust output values, the power consumption of each thruster for the two fly-bys are shown in Figure 6.45 and Figure 6.46, from which it can already be seen that the power requirements of the system with configuration 2 is lower than that of configuration 1. The final total energy required for configuration 2 is 1556 [J], whereas this value is equal to 1882 [J] for configuration 1.

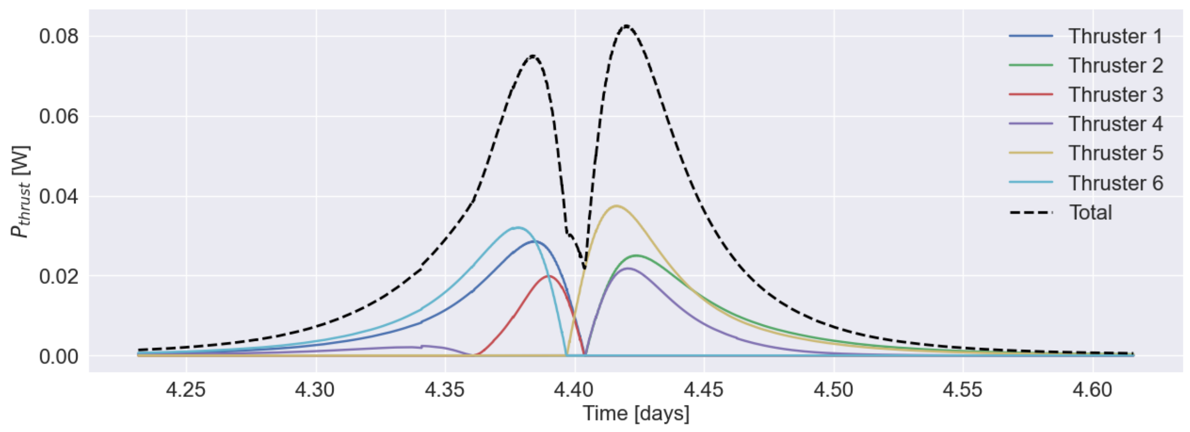


Figure 6.45: Thrusters: P required for all thrusters in configuration 2 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

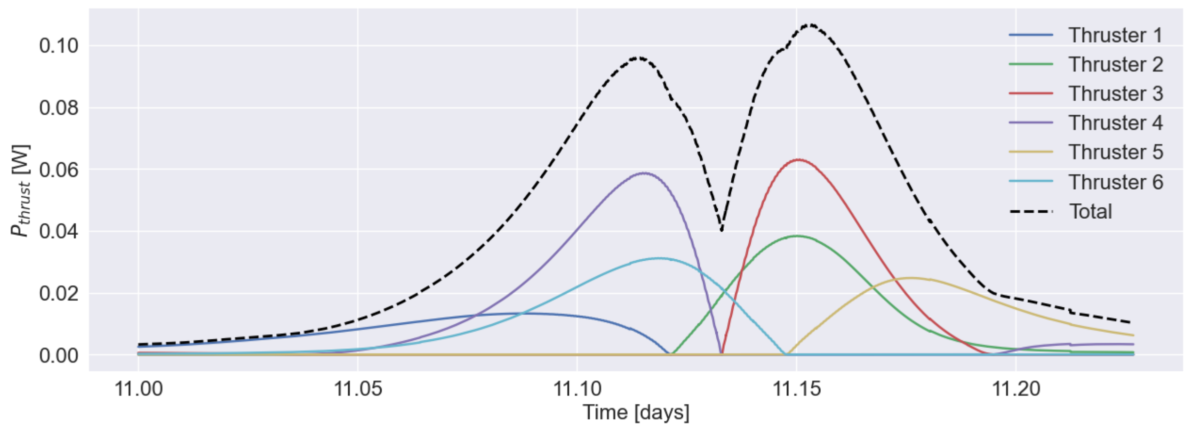


Figure 6.46: Thrusters: P required for all thrusters in configuration 2 over time in days, analysed from [00:00:00 12-01-2023] until [05:26:40 12-01-2023].

6.1.2.3 Configuration 3

For configuration 3, the thrust outputs for the flybys have been shown in Figure 6.47 to Figure 6.50. The addition of 2 extra thrusters called for an extra graph for a better view on the thrust required.

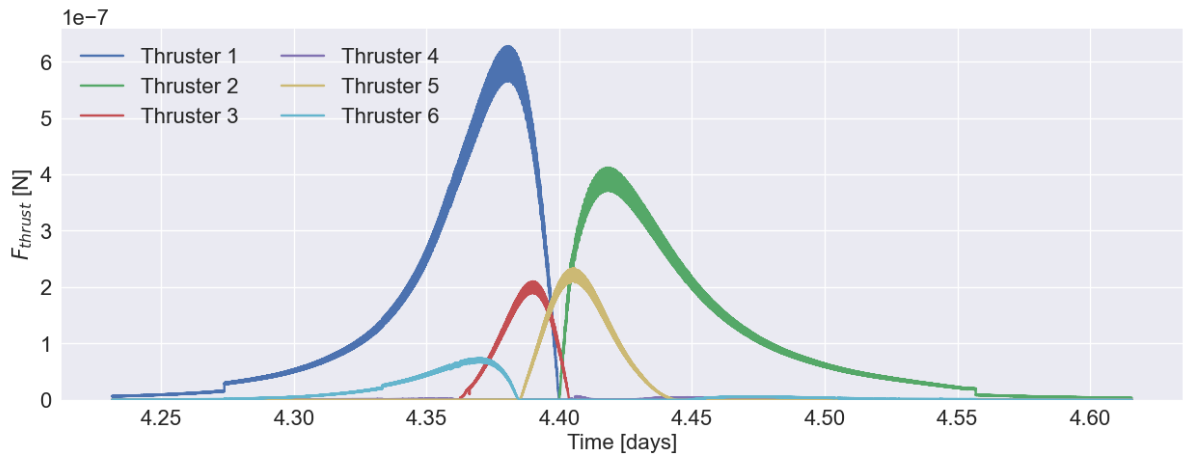


Figure 6.47: Thrusters: F output for thrusters 1, 2, 3, 4, 5 and 6 in configuration 3 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

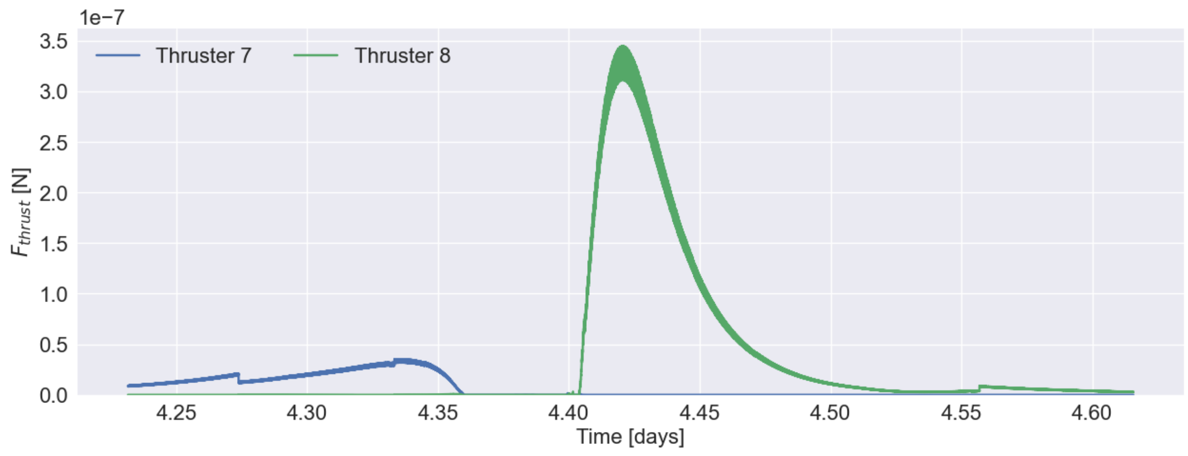


Figure 6.48: Thrusters: F output for thrusters 7 and 8 in configuration 3 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

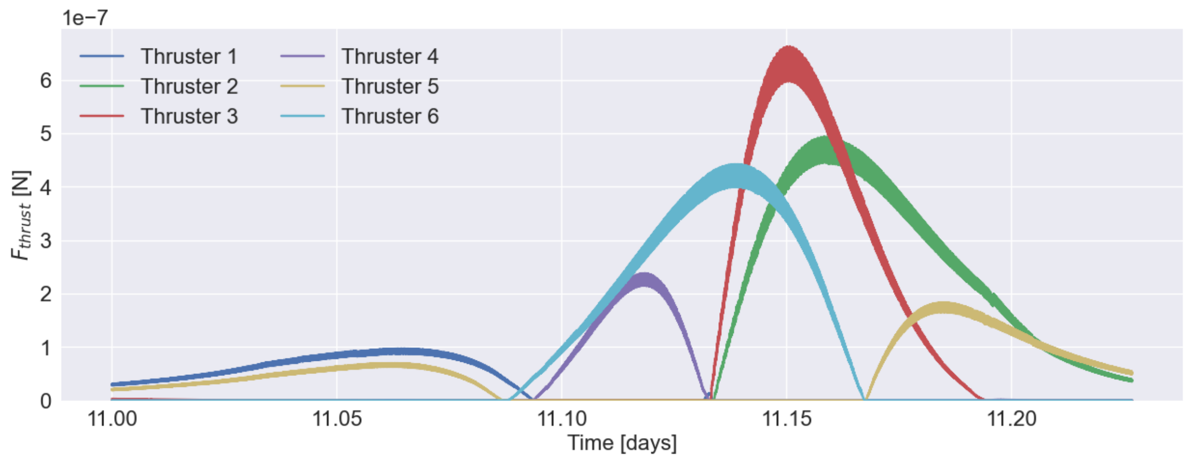


Figure 6.49: Thrusters: F output for thrusters 1, 2, 3, 4, 5 and 6 in configuration 3 over time in days, analysed from [00:00:00 12-01-2023] until [05:26:40 12-01-2023].

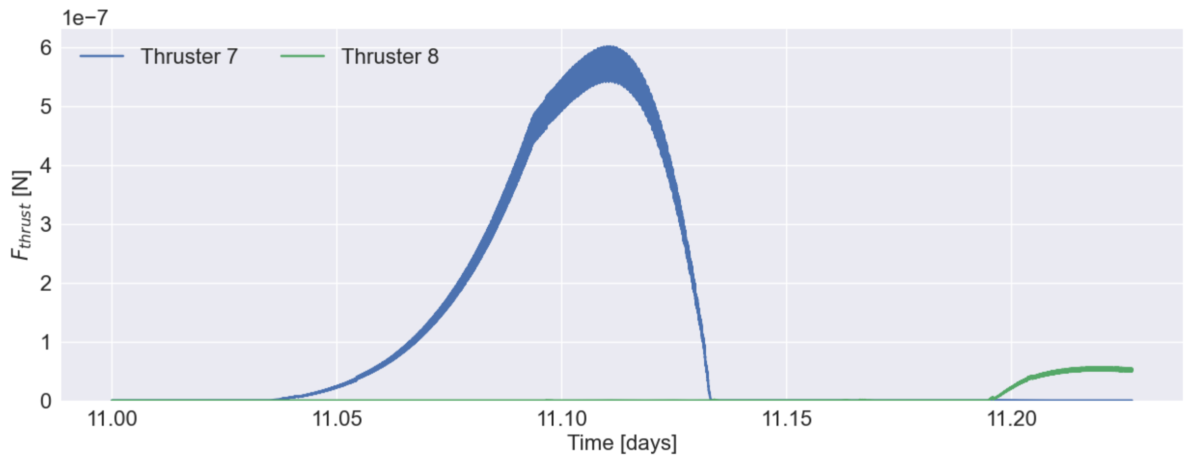


Figure 6.50: Thrusters: F output for thrusters 7 and 8 in configuration 3 over time in days, analysed from [00:00:00 12-01-2023] until [05:26:40 12-01-2023].

In Figure 6.51, the linear impulse of the thrusters in configuration 3 is shown. The power graph has been omitted for this thruster configuration since this will be shown in subsection 6.1.3. The final total energy required for this configuration amounted to 1753 [J].

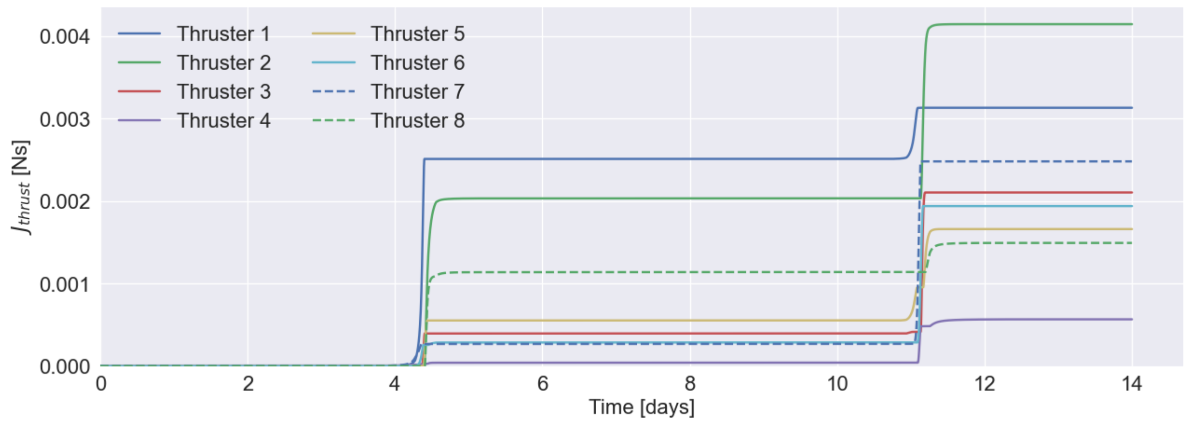


Figure 6.51: Thrusters: J_{thrust} output for all thrusters in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

6.1.2.4 Configuration 4

Figure 6.52 to Figure 6.55 shows the thrust requirements for the twelve thrusters in configuration 4.

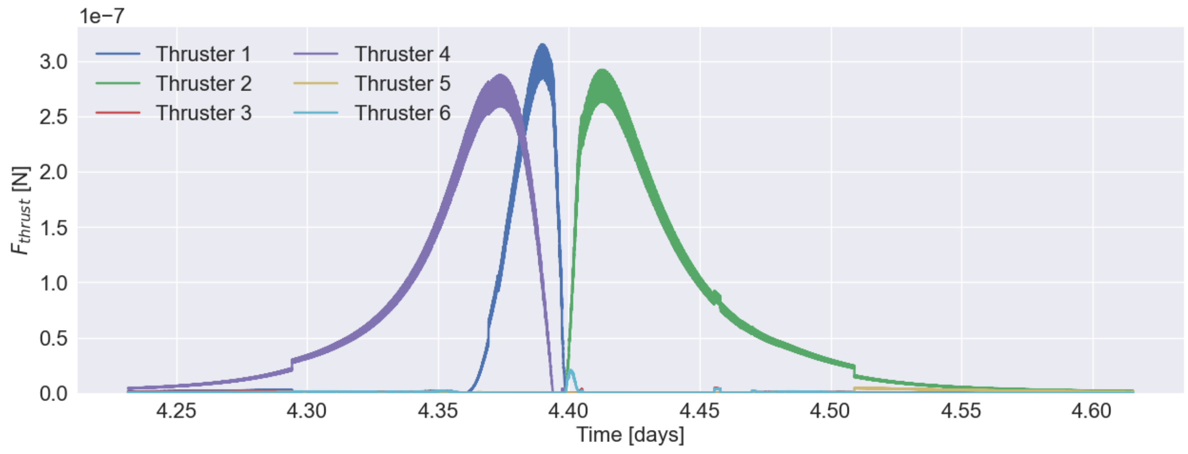


Figure 6.52: Thrusters: F output for thrusters 1, 2, 3, 4, 5 and 6 in configuration 4 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

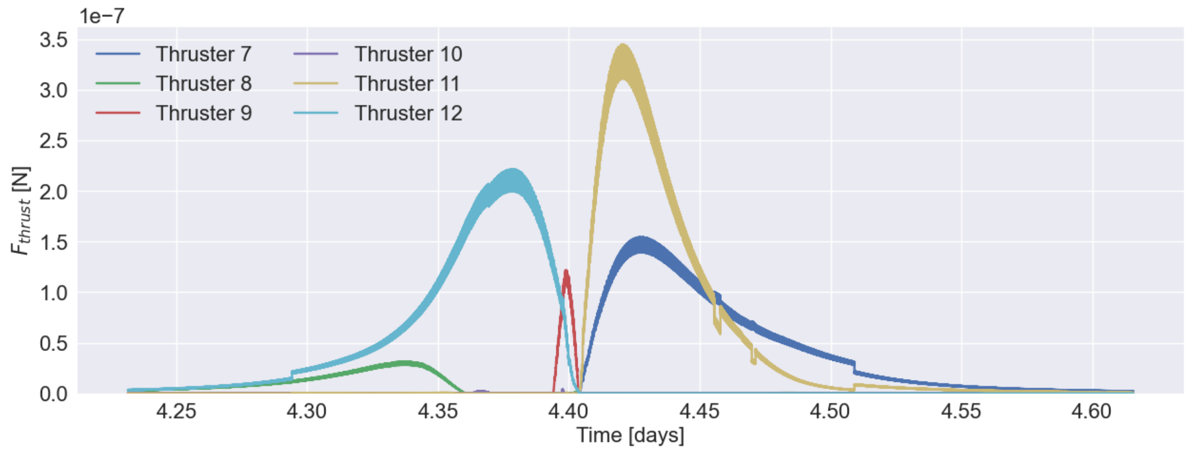


Figure 6.53: Thrusters: F output for thrusters 7, 8, 9, 10, 11 and 12 in configuration 4 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

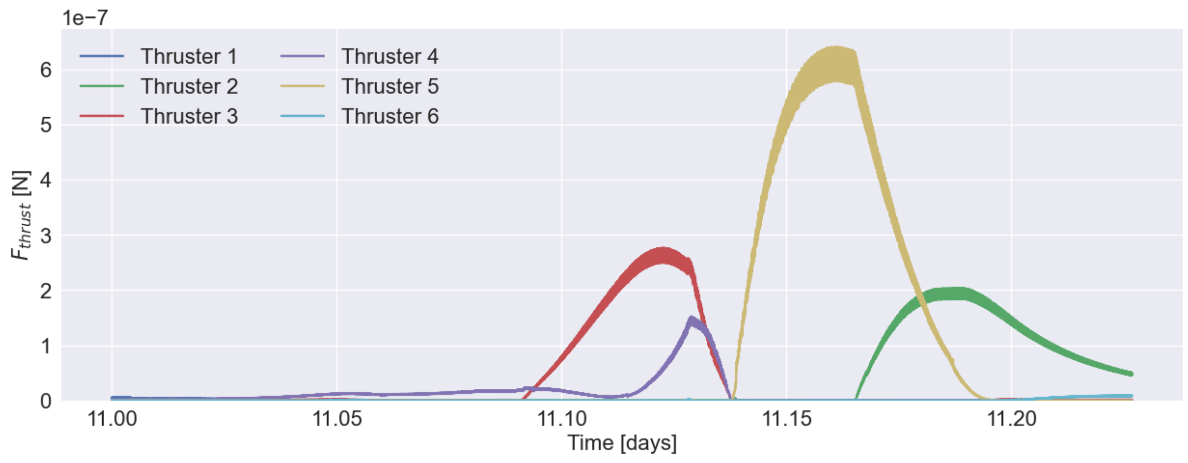


Figure 6.54: Thrusters: F output for thrusters 1, 2, 3, 4, 5 and 6 in configuration 4 over time in days, analysed from [00:00:00 12-01-2023] until [05:26:40 12-01-2023].

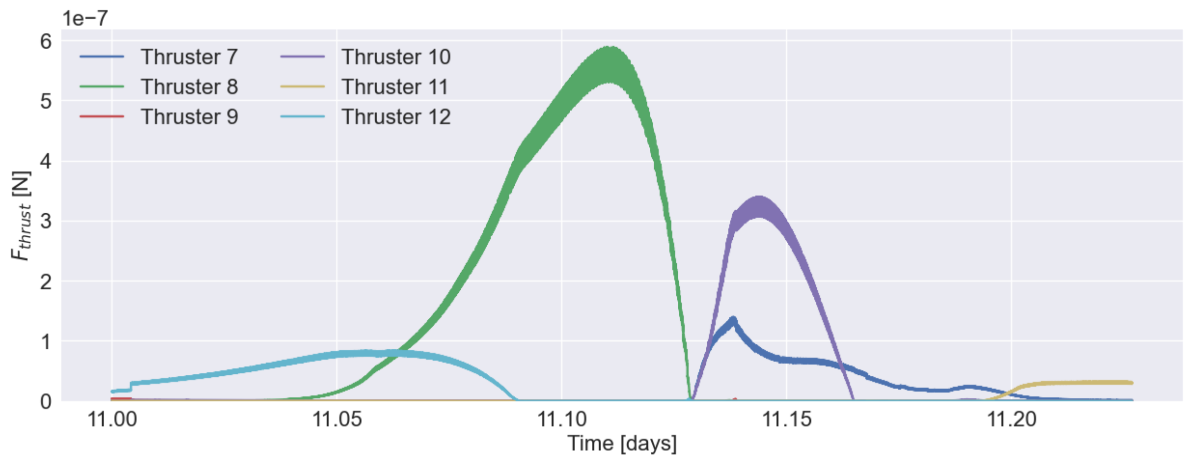


Figure 6.55: Thrusters: F output for thrusters 7, 8, 9, 10, 11 and 12 in configuration 4 over time in days, analysed from [00:00:00 12-01-2023] until [05:26:40 12-01-2023].

Finally, the linear increase over time for the thrusters in configuration 4 is shown in Figure 6.56 and Figure 6.57. The final total energy required for this configuration amounted to 1334 [J], making it the least energy-consuming thruster configuration in this research.

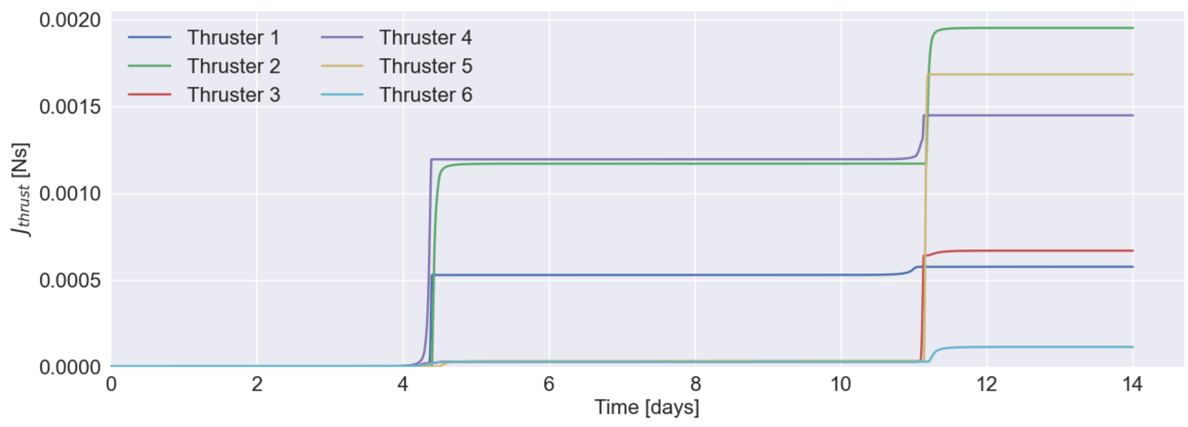


Figure 6.56: Thrusters: J_{thrust} output for the first six thrusters in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

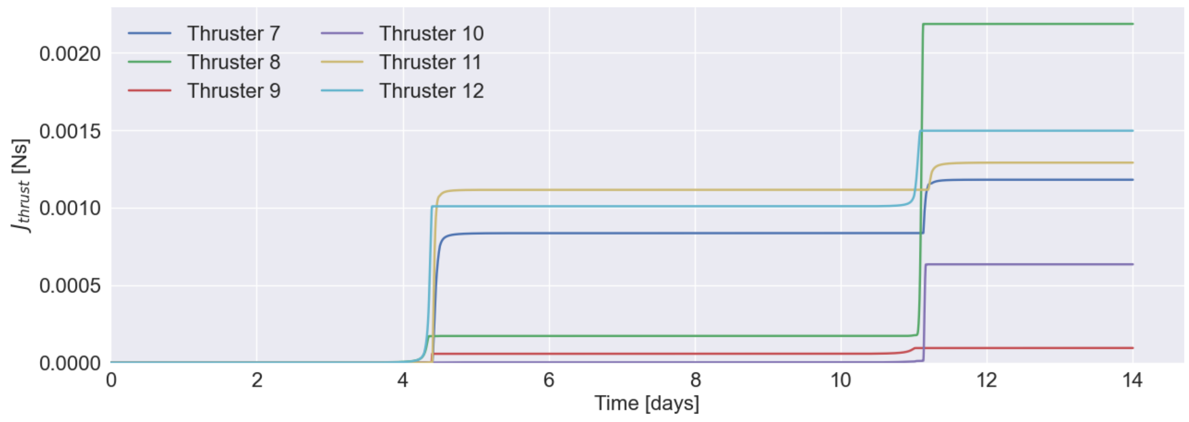


Figure 6.57: Thrusters: J_{thrust} output for the last six thrusters in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

6.1.3. Comparison

This section will conclude all the results from subsection 6.1.1 and subsection 6.1.2 and will visually compare the total power consumption of the thrusters and reaction wheels, as well as the total energy consumption. In addition, the absolute difference in the half-cone angle between the reaction wheel and thruster analysis will be shown, which proves the assumption made previously regarding the influence of the total angular momentum.

The power graphs for the thruster configurations can be seen in Figure 6.58 to Figure 6.60. In addition, Figure 6.61 shows the energy consumption of all thruster configurations over time. From this graph, it can be seen that configuration 1 requires the most amount of power, after which configuration 3, 2 and finally configuration 4 follow.

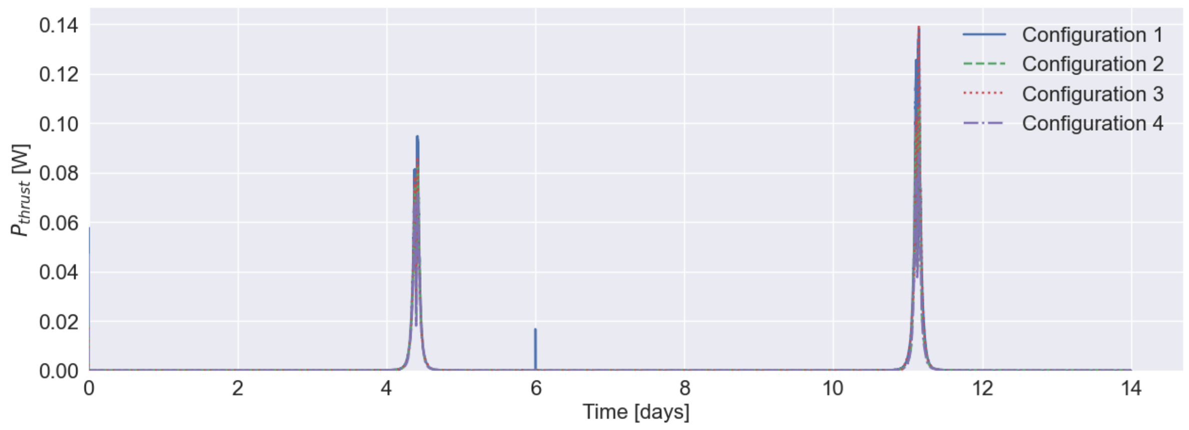


Figure 6.58: Total power required for each thruster configuration over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

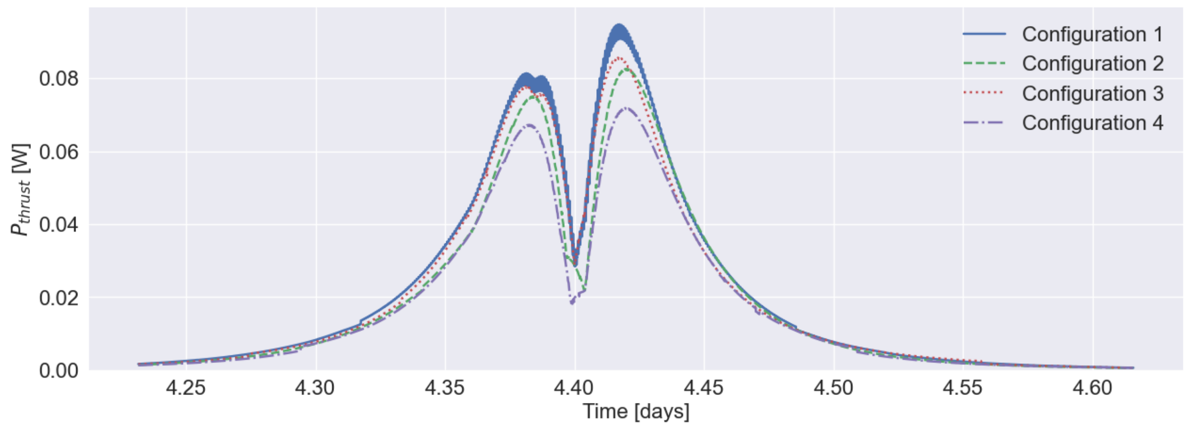


Figure 6.59: Total power required for each thruster configuration over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

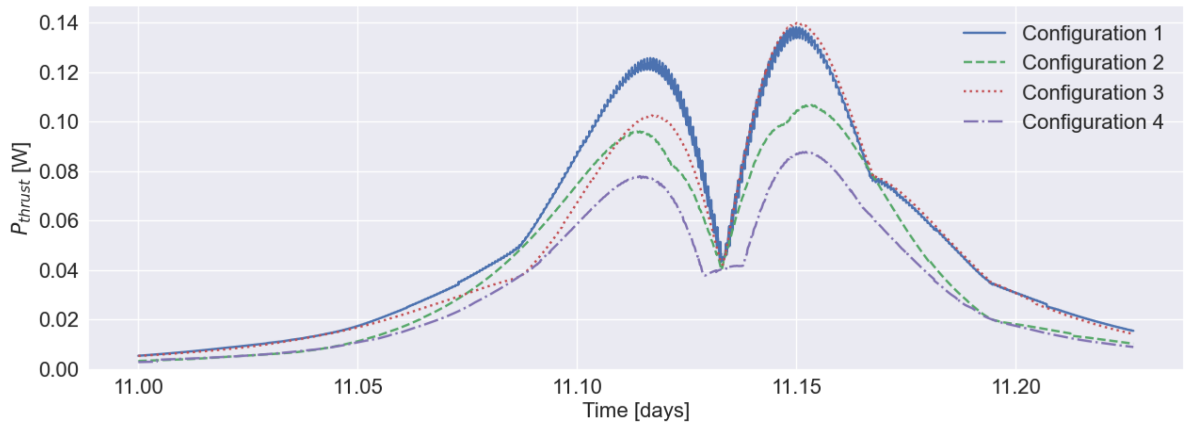


Figure 6.60: Total power required for each thruster configuration over time in days, analysed from [00:00:00 12-01-2023] until [05:26:40 12-01-2023].

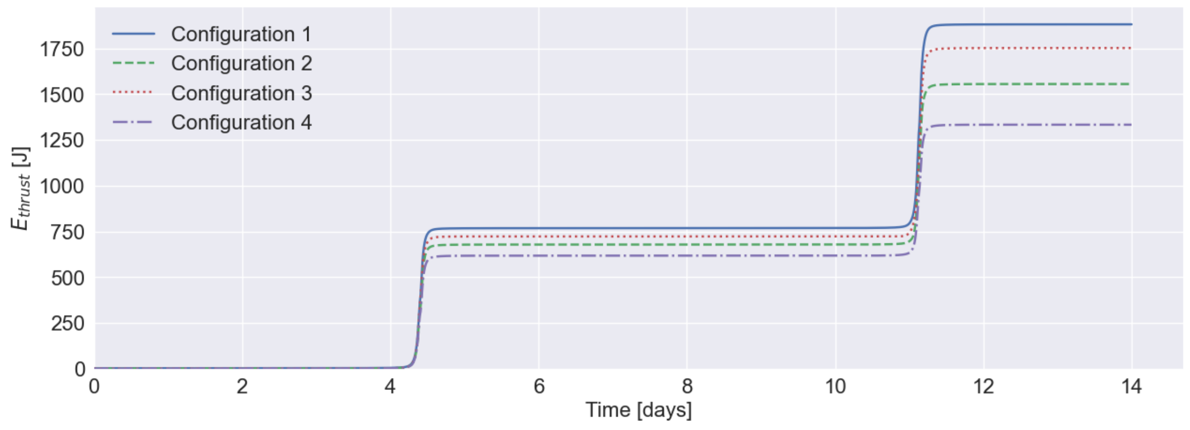


Figure 6.61: Total energy consumed by each thruster configuration over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

In Figure 6.62 below, the reaction wheel results as seen previously are combined and show the total power requirement and total energy consumption again.

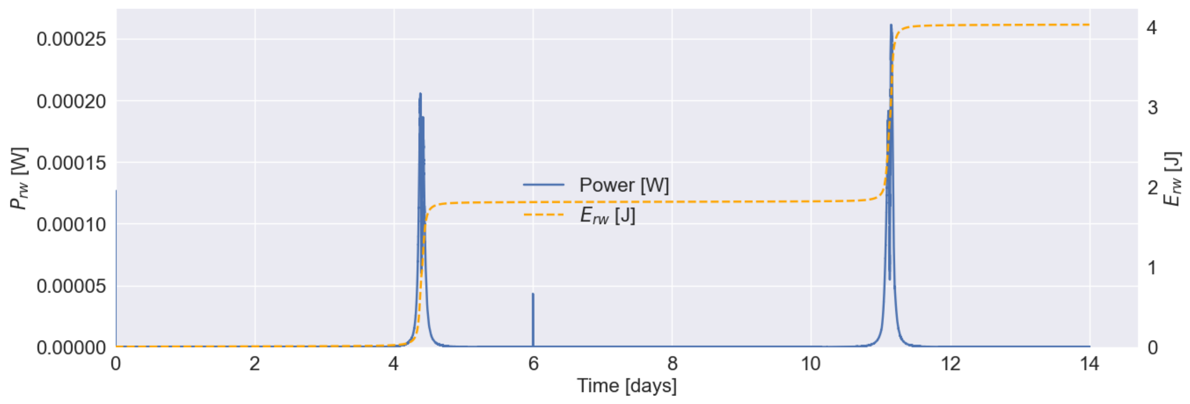


Figure 6.62: Total power and energy consumed by each reaction wheels over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

Finally, Figure 6.63 shows the absolute difference in half-cone angle offset between the reaction wheel analysis and thruster analysis. It can be seen that this difference is in the order of $10^{-7\circ}$, which is approximately 10^{-4}% of the half-cone offset angles shown in Figure 6.2. For this reason, it can finally be concluded that the difference in attitude dynamics for the reaction wheels is sufficiently small for the gain values in both analyses to be equal.

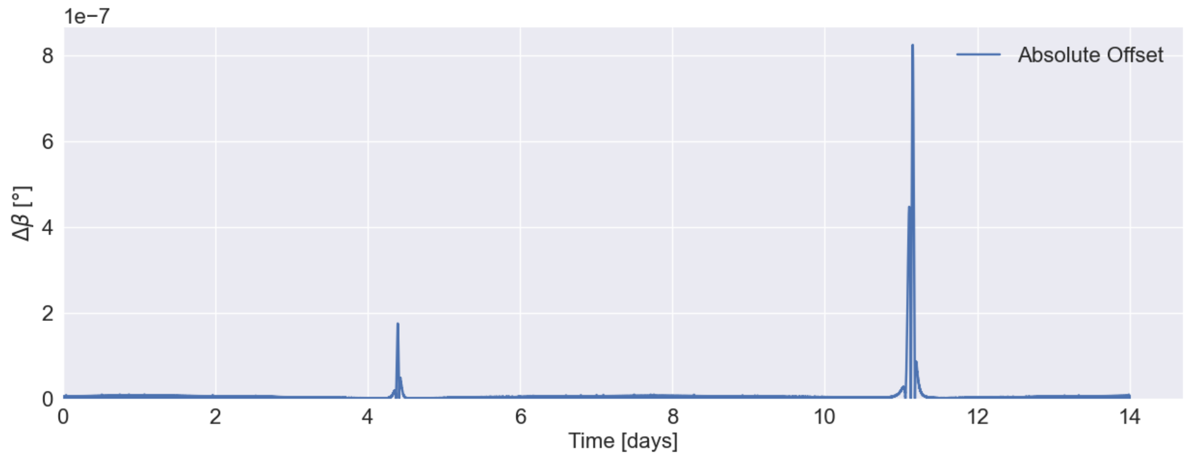


Figure 6.63: Absolute difference in degrees between the reaction wheel and thruster analysis half-cone angle offset, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

From the above results, specifically looking at the power and energy requirements for the thruster-only ADCS, it can already be concluded that the reaction wheel system is beneficial. The total energy required for the reaction wheel set-up only amounts to several Joules over the total simulation period, whereas the thrusters require several kilo-Joules for the same time span. Since the reaction wheel set-up still requires thrusters, and these were chosen to be chemical mono-propellant types, extra tank volume and mass is required in this ADCS. For this reason, further analysis of the mass and volume of each system is still to be done to draw final conclusions on this, which will be performed in Chapter 7.

Specifically looking at the comparison between the electrical thruster configurations, it can be seen that configuration 1 has the highest energy demand, followed by configuration 3, 2 and 4. From this, as will also be discussed more extensively in the next chapter, it can be concluded that coupling multiple axes for attitude control introduces power and energy consumption benefits. Adding redundant thrusters to an existing configuration, as was done in configuration 3, also introduces slight benefits with respect to power and energy. A fully redundant system as was introduced in configuration 4 performed best.

6.2. Robustness

This section provides an overview of the aforementioned three robustness tests: the single thruster failure, the solar array deployment and, finally, the de-tumbling manoeuvrer behaviour. All relevant results have been collected in tables and clear graph in order to provide an easy overview.

6.2.1. Single Thruster Failure

The single thruster failure robustness test is divided in two distinct parts: in the first approach, in which the system is aware of thruster failure and compensates the malfunctioning thruster with the remaining ones, only configurations 3 and 4 are analysed due to the under-determined state of the first two configurations when one of the thrusters is not taken into account any more. The second approach assumes that the system is not aware of the malfunctioning thruster, and attempts to correct the incorrect attitude attained at every time step. For both approaches, the difference in thrust levels for each configuration is assessed, as well as the power and energy requirement. In order to scope this robustness test within relevant bounds, only the first fly-by as was seen in previous results will be investigated (between [05:33:20 05-01-2023] and [14:46:40 05-01-2023]). Moreover, the thruster that was required most in terms of total thrust delivered in the base case scenario, will be turned off for these tests. For configuration 1, thruster 1 was most critical. For configuration 2, this was thruster 5. For configuration 3, this was thruster 1 and, finally, for configuration 4, thruster 11 was most critical during the first fly-by. These conclusions could easily be drawn from the results in the previous section.

6.2.1.1 Approach 1

For the first approach, Figure 6.64 and Figure 6.65 show the differences in thrust output for each of the thrusters over time, compared to the original simulation as shown in the previous section, for configuration 3. Configuration 4 is shown in Figure 6.66 and Figure 6.67. For configuration 3, it can be seen that the malfunctioning of thruster 1 induces the largest increase in the output thrust of thruster 7, as well as a slightly lower increase in thruster 3. This increase in thruster 3 is approximately equal in magnitude to original thrust output of thruster 1. For configuration 4, net thruster activity increase in almost all thrusters can be seen, except thruster 5, 8, 9 and 10.

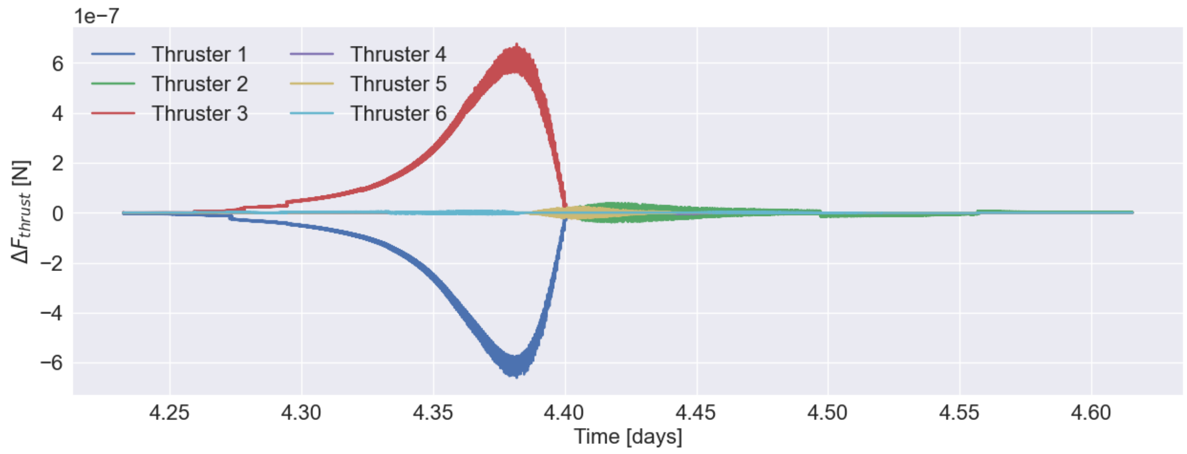


Figure 6.64: Single thruster failure 1: difference in F output, compared to the original simulation with all working thruster, for thrusters 1, 2, 3, 4, 5 and 6 in configuration 3 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

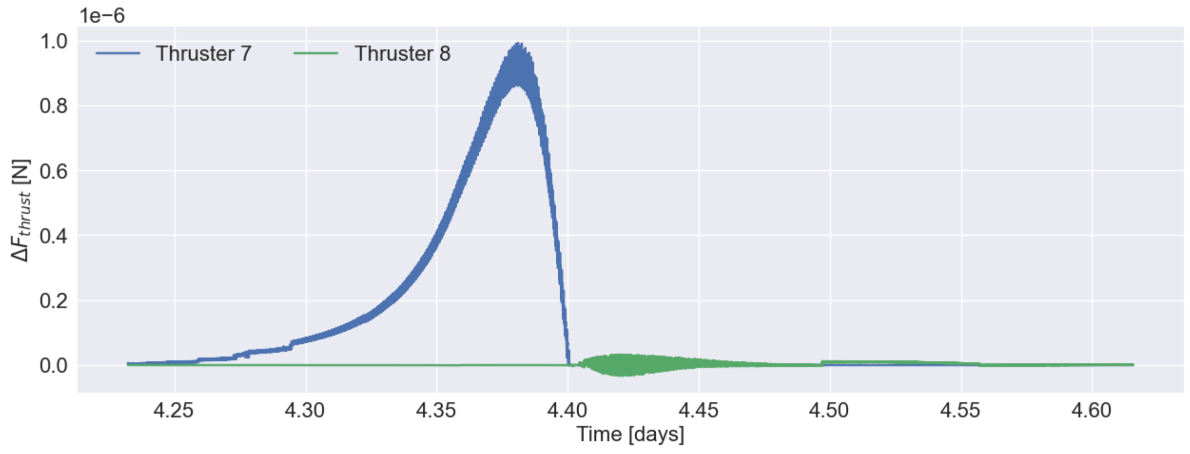


Figure 6.65: Single thruster failure 1: difference in F output, compared to the original simulation with all working thruster, for thrusters 7 and 8 in configuration 3 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

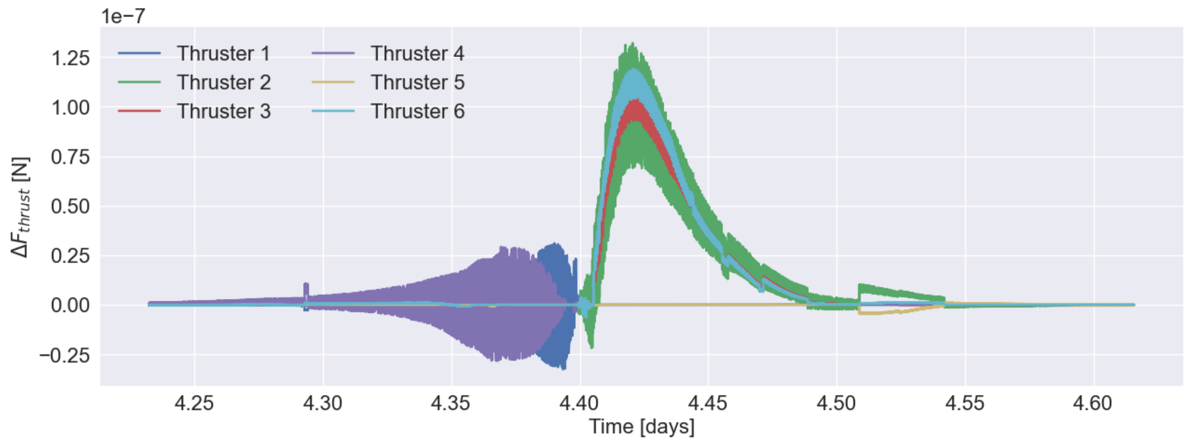


Figure 6.66: Single thruster failure 1: difference in F output, compared to the original simulation with all working thruster, for thrusters 1, 2, 3, 4, 5 and 6 in configuration 4 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

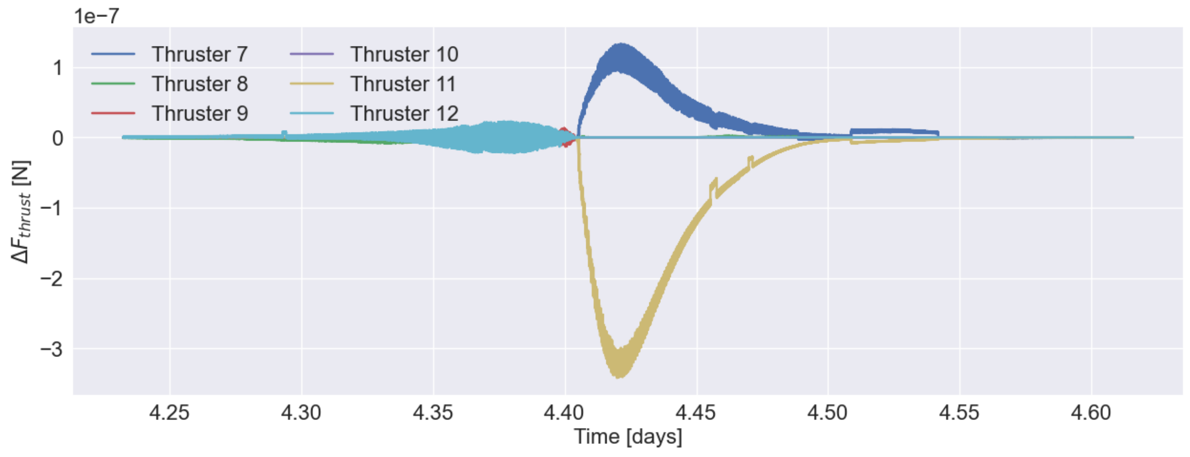


Figure 6.67: Single thruster failure 1: difference in F output, compared to the original simulation with all working thruster, for thrusters 7, 8, 9, 10, 11 and 12 in configuration 4 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

Table 6.1 shows the results for the first and second approaches in single thruster failure, for configuration 3 and 4. Its columns represent the simulation type, the maximum total power needed throughout the simulation, and finally the total energy consumed by the system. In addition, the base cases without thruster failure, for the simulation period of [05:33:20 05-01-2023] to [14:46:40 05-01-2023], are presented in their total power requirement and energy consumption. It is important to realise that this does not represent the full simulation, only the close fly-by period, which is why these results are different from the ones observed before.

Simulation type	Configuration	$P_{tot,max}$ $[W]$	E_{tot} $[J]$
Base	3	0.085	760
STF 1	3	0.3005	1165
STF 2	3	20.18	291,582
Base	4	0.0711	649
STF 1	4	0.1567	683
STF 2	4	20.53	362,486

Table 6.1: Results for single thruster failure simulations, approach 1 and 2, for both configuration 3 and 4. The maximum power requirement ($P_{tot,max}$) and total energy consumption (E_{tot}) for the first fly-by are shown, and compared to the base case.

6.2.1.2 Approach 2

Upon execution of the simulations for approach 2, it was evident that no stable simulation could be obtained for configurations 1 and 2. This is shown in Figure 6.68 and Figure 6.69, in which the quaternion error for all components start oscillating excessively after a certain time step. For this reason, it could be concluded that the thruster failure case cannot successfully be applied to these configurations in the present simulation set-up. Although this was already expected during the research generation phase, it could not be readily be confirmed for this failure approach from past literature or research sources.

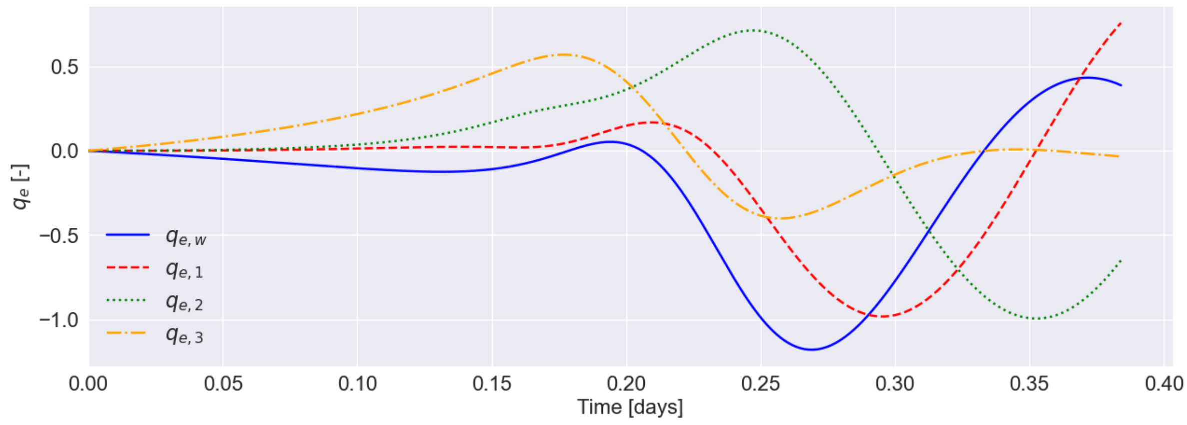


Figure 6.68: Single thruster failure 2: q_e (absolute) over time in days, for configuration 1, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

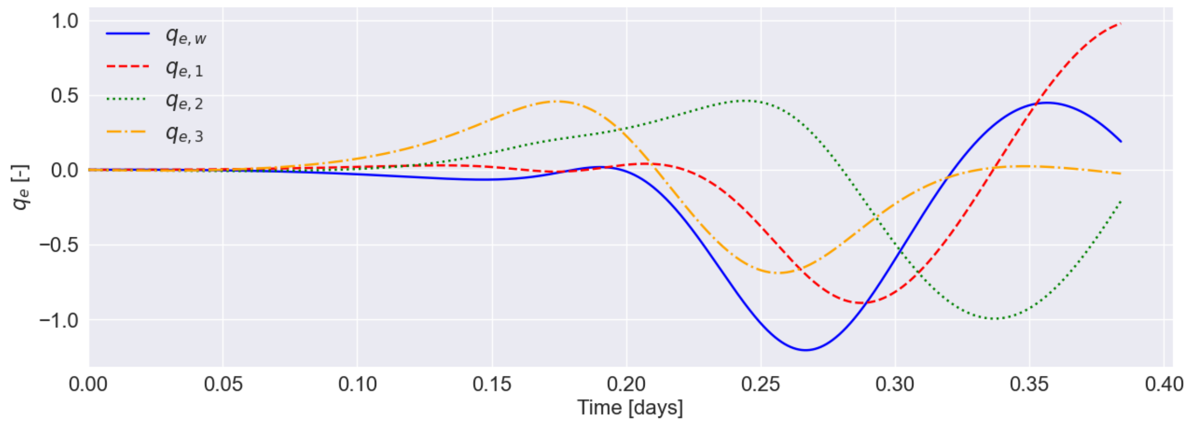


Figure 6.69: Single thruster failure 2: q_e (absolute) over time in days, for configuration 2, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

For configuration 3 and 4, a stable solution was found and its maximum total power requirement and total energy consumption can be seen in Table 6.1. Moreover, the change in output thrust required from each thruster can be observed in Figure 6.70 to Figure 6.73 below. It can be seen that large thrust correction spikes are present, in the same thrusters that were activated during the first single thruster failure test. Thruster 6 en 7 especially show a large increase in thrust, around the closest approach point. These spikes pose the explanation for the high maximum total power and energy consumed seen from the results table and it is evident that more effort was needed for attitude control in the second thruster failure case compared to the first one.

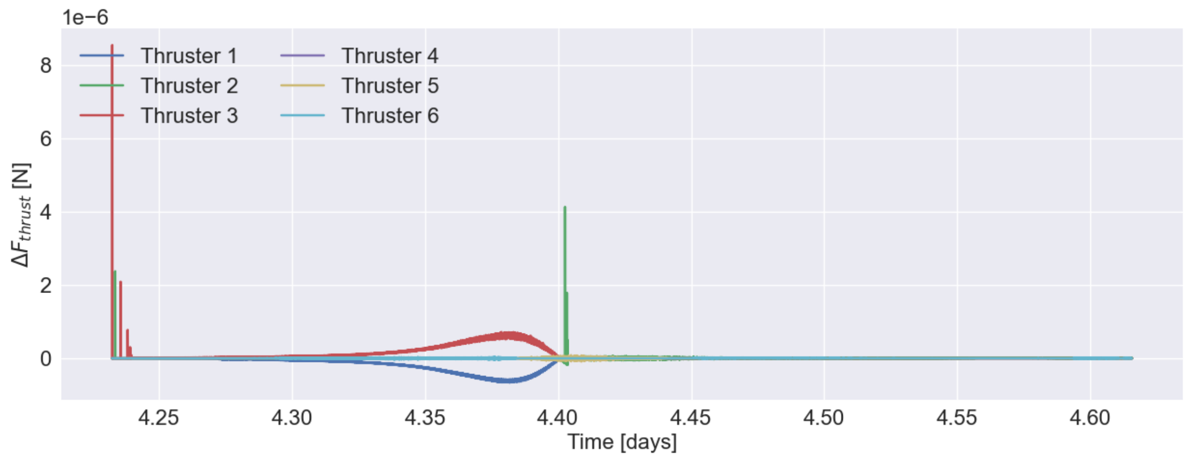


Figure 6.70: Single thruster failure 2: difference in F output, compared to the original simulation with all working thruster, for thrusters 1, 2, 3, 4, 5 and 6 in configuration 3 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

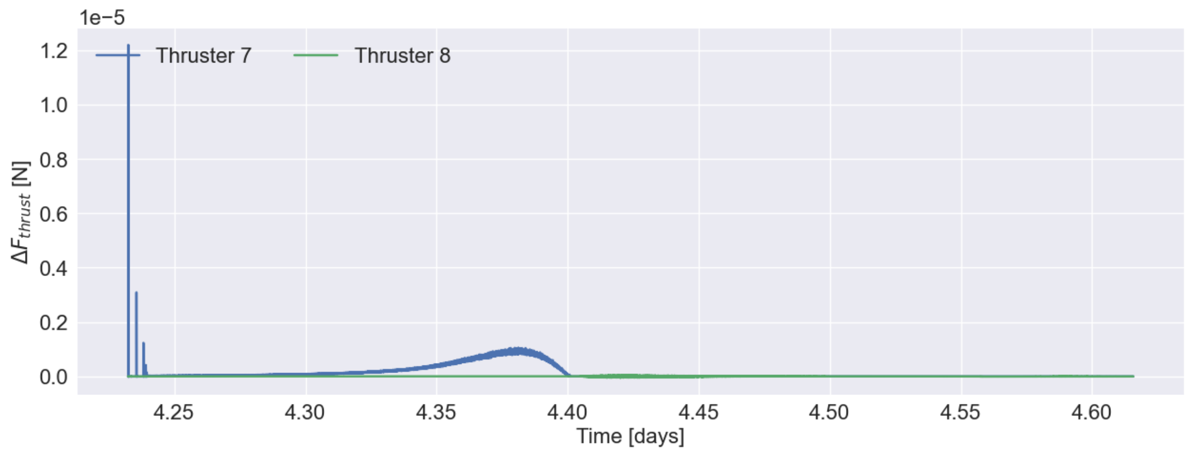


Figure 6.71: Single thruster failure 2: difference in F output, compared to the original simulation with all working thruster, for thrusters 7 and 8 in configuration 3 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

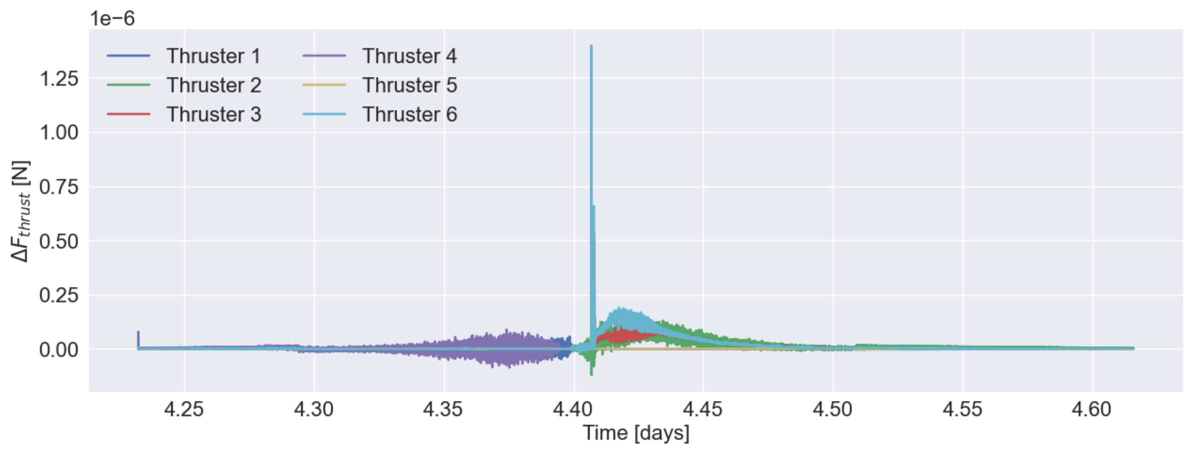


Figure 6.72: Single thruster failure 2: difference in F output, compared to the original simulation with all working thruster, for thrusters 1, 2, 3, 4, 5 and 6 in configuration 4 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

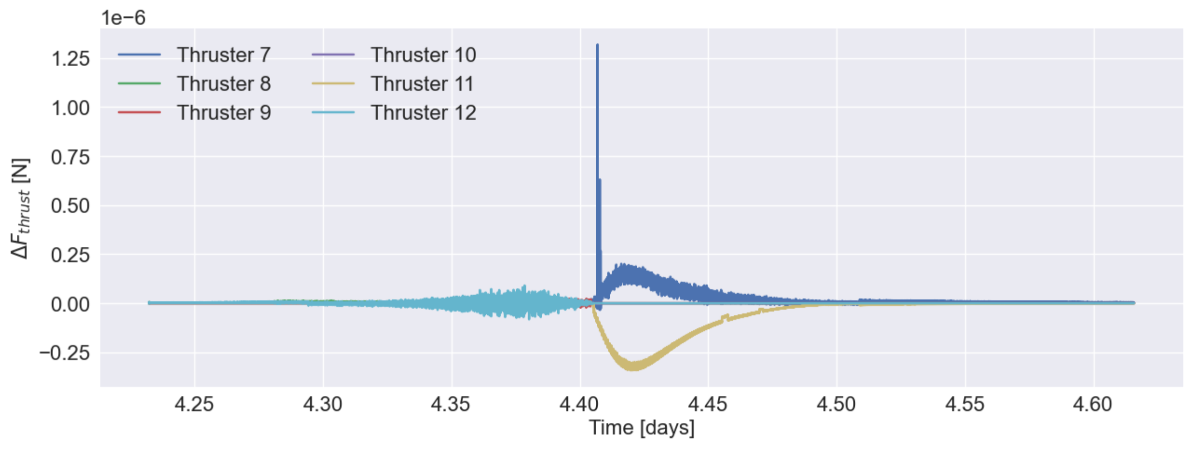


Figure 6.73: Single thruster failure 2: difference in F output, compared to the original simulation with all working thruster, for thrusters 7, 8, 9, 10, 11 and 12 in configuration 4 over time in days, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

Finally, the half-cone offset angle for both configuration 3 and 4 are shown in Figure 6.74 below, and compared to the base case as was seen in Section 6.1. It can be seen, that the all half-cone angles remain within the required limits again, and that the offset for configuration 3 is larger than the base case, until the closest approach point during the fly-by. Moreover, the offset angle for configuration 4 is actually smaller than the base case after this closest approach. Therefore, in terms of pointing error, configuration 4 performed better with one thruster malfunctioning, while the system was not aware of this happening.

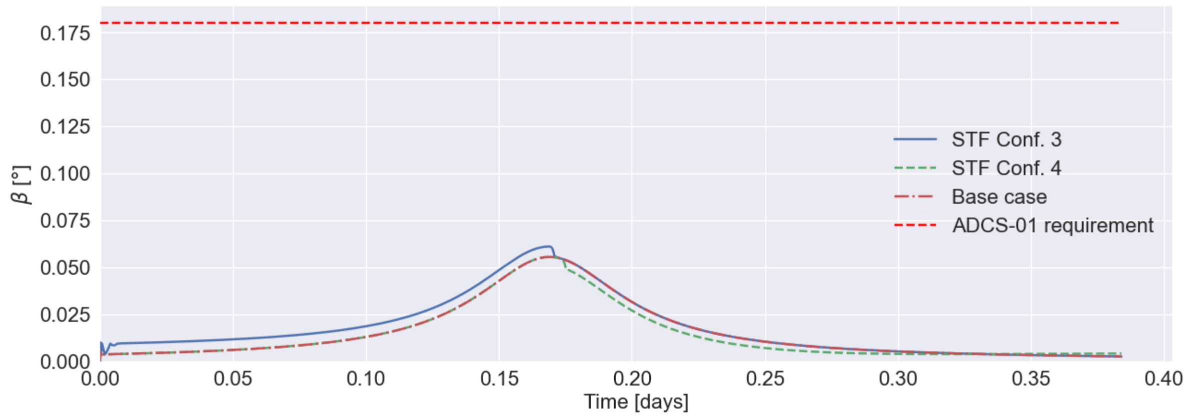


Figure 6.74: Single thruster failure 2: half-cone offset angle for configuration 3, 4 and the base case, compared to the system requirement, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023.

From the results in Table 6.1, it can readily be concluded that approach 1 requires a total energy increase of +53% for configuration 3 and +5.2% for configuration 4. For configuration 2, however, the energy increases and total maximum power required for the thrusters become exceptionally high (380× total energy for configuration 3,559× total energy for configuration 4) compared to the base case, due to the incorrectly-attained attitude at every time step. These power increases would most likely cause the system and mission to fail under real-life circumstances, which is why sensors should always be present for the OBC to assess thruster failure and adhere to approach 1.

6.2.2. Solar Array Deployment

Un-deploying the solar arrays during the full two-week simulation does not impose many differences; the solar radiation pressure, which was already found to be insignificantly small for the control torque computations, and the spacecraft inertia matrix need to be changed. The inertia matrix will show a reduction, as seen in Equation 6.3. The reduction of the inertia matrix called for a decrease of the speed gain from 12 to 10, for proper functioning of the algorithm.

$$\Delta I_{deployed \rightarrow undeployed} = \begin{bmatrix} -70\% & 0 & 0 \\ 0 & -17\% & 0 \\ 0 & 0 & -70\% \end{bmatrix} \quad (6.3)$$

As mentioned previously, the full two-week simulation has been run for the undeployed solar array configuration. Upon inspection of the solar radiation pressure values, the disturbance torques were equal in both cases. The reason for this lies in the simplified model used for this research, as will be discussed in more detail in Chapter 7. Table 6.2 shows the maximum power and energy required for the full two-week simulation, for each configuration deployed as well as undeployed.

Configuration	$P_{tot,max}$ $[W]$	E_{tot} $[J]$
1, undeployed	0.0575	733
2, undeployed	0.0506	582
3, undeployed	0.0574	703
4, undeployed	0.0305	467
1, deployed	0.1382	1882
2, deployed	0.1067	1556
3, deployed	0.1398	1753
4, deployed	0.0877	1334

Table 6.2: Results for thruster configurations, showing maximum power requirement ($P_{tot,max}$) and total energy consumption (E_{tot}) for the two-week simulation analysis.

Finally, Figure 6.75 shows the half cone angle offset over time for the undeployed case. Similar as before, this graph is equal for all configurations. It can be seen that the half-cone angle offset is similar to the one observed for the original deployed solar array case. In fact, the maximum difference in half-cone angle offset between the two cases over the entire simulation duration only amounted to $6.35 \cdot 10^{-5\circ}$.

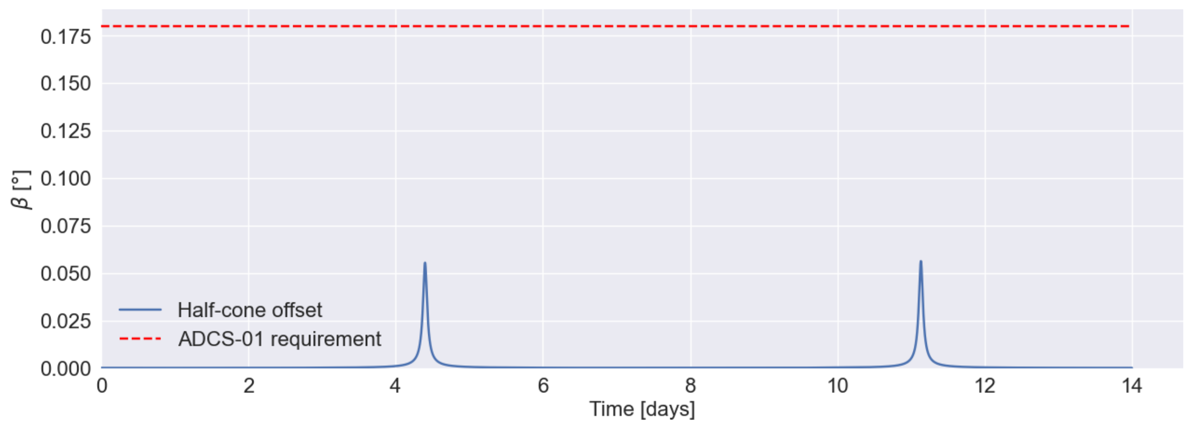


Figure 6.75: Single thruster failure: β, measured between the negative y-panel normal vector and the Moon pointing vector and presented in absolute numbers, over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

6.2.3. De-tumbling Maneuver

Upon the start of the de-tumbling robustness test, it was quickly found that no thruster output solutions existed for any initial angular velocities above 0.00005 $[rad/s]$ (approximately 0.003$[^\circ/s]$). An explanation and further discussion on this, related to the method with which the attitude control algorithm was created, will be given in Chapter 7. Although the feasible angular velocities do not reflect real-life mission situations, in which de-tumbling should occur at far higher rates (above 10 $[^\circ/s]$), the behaviour of the system was in any case assessed for the initial angular velocity vectors as shown below in Table 6.3.

Test number	$[\omega_{0,x}, \omega_{0,y}, \omega_{0,z}]$ $[rad/s]$
1	$[5 \cdot 10^{-5}$, 0, 0]
2	[0, $3 \cdot 10^{-5}$, 0]
3	[0, 0, $5 \cdot 10^{-5}$]
4	$[5 \cdot 10^{-5}, 3 \cdot 10^{-5}$, 0]
5	$[5 \cdot 10^{-5}$, 0, $5 \cdot 10^{-5}]$
6	[0, $3 \cdot 10^{-5}, 5 \cdot 10^{-5}]$
7	$[5 \cdot 10^{-5}, 3 \cdot 10^{-5}, 5 \cdot 10^{-5}]$

Table 6.3: Initial angular velocities in degrees per second for the de-tumbling robustness test of the attitude control algorithm. Values found based on tests for feasibility within the thruster allocation algorithm.

As mentioned previously, the time frame during which these de-tumbling manoeuvrers were tested is [00:00:00 01-01-2023] until [00:02:00 01-01-2023], since the disturbance torques are insignificant in this frame as was seen in the results for the main simulation. Each initial angular velocity from the table is fed to the control algorithm as initial state. The reference quaternion was used as the quaternion vector for the initial state. For the first test, the angular velocity and error quaternions over time can be seen in Figure 6.77 and Figure 6.76, in which a clear oscillatory pattern around the equilibrium is seen for the first time steps. As was also seen before, the relative quaternion error with respect to each component's absolute value is shown in Figure 6.78. It can be seen that, after approximately 28 seconds, the error components stop oscillating significantly and remain within a 5% error band, which will be defined as the settling time in this analysis.

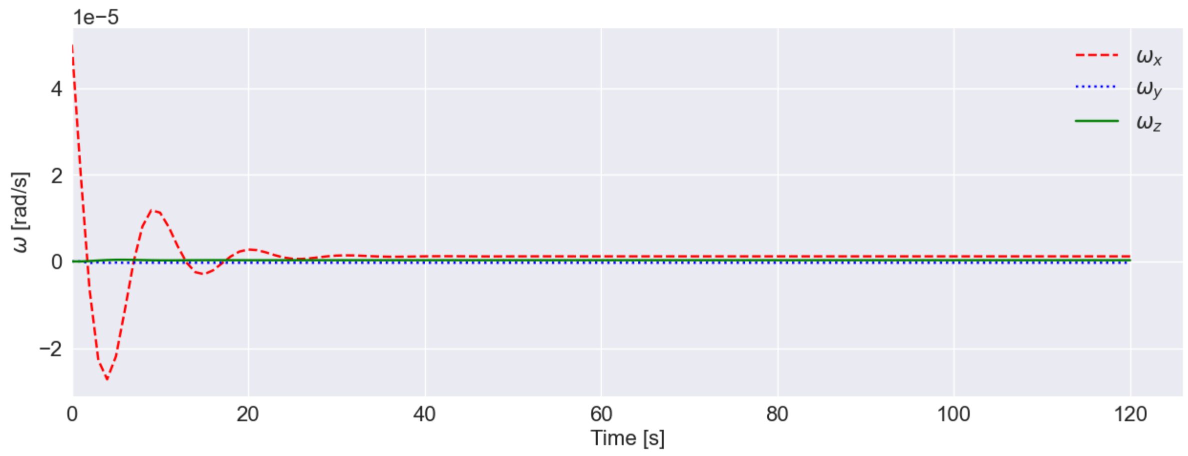


Figure 6.76: De-tumble test 1: ω for each of the spacecraft primary axes over time in days, analysed from [00:00:00 01-01-2023] until [00:02:00 01-01-2023].

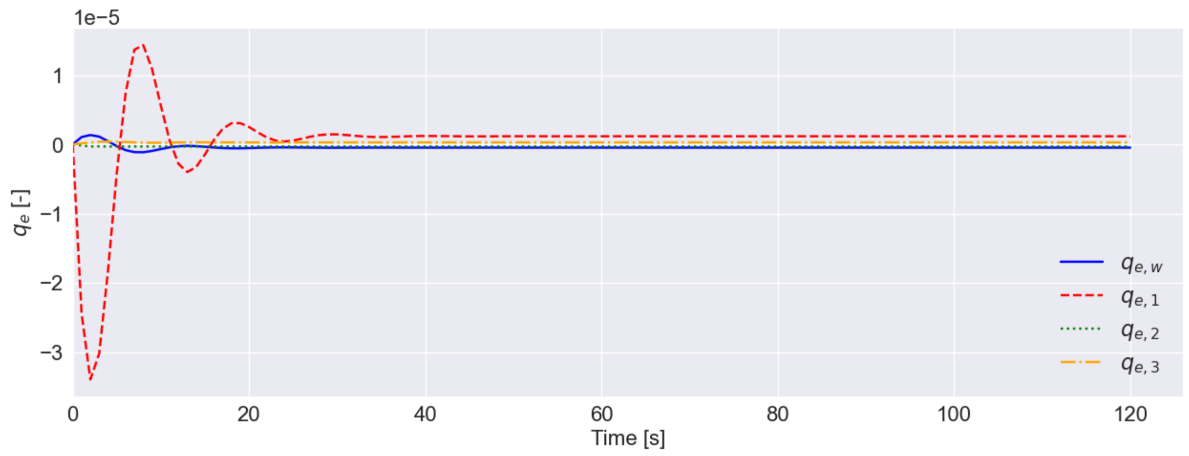


Figure 6.77: De-tumble test 1: q_e (absolute) for each quaternion error component over time in days, analysed from [00:00:00 01-01-2023] until [00:02:00 01-01-2023].

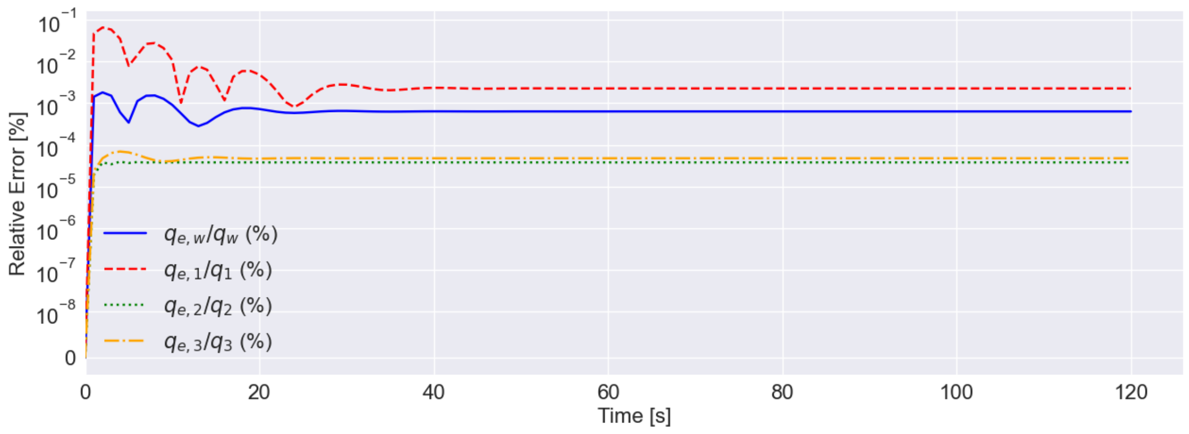


Figure 6.78: De-tumble: q_e (relative) for each quaternion error component over time in days, analysed from [00:00:00 01-01-2023] until [00:02:00 01-01-2023].

In addition, the thrust output values of all thrusters can be visualised, as seen in Figure 6.79. The activation of thrusters 3 and 4 is observed and could also be expected based on their location and thrust directions; these two thrusters control the x-axis of the spacecraft in the simulation.

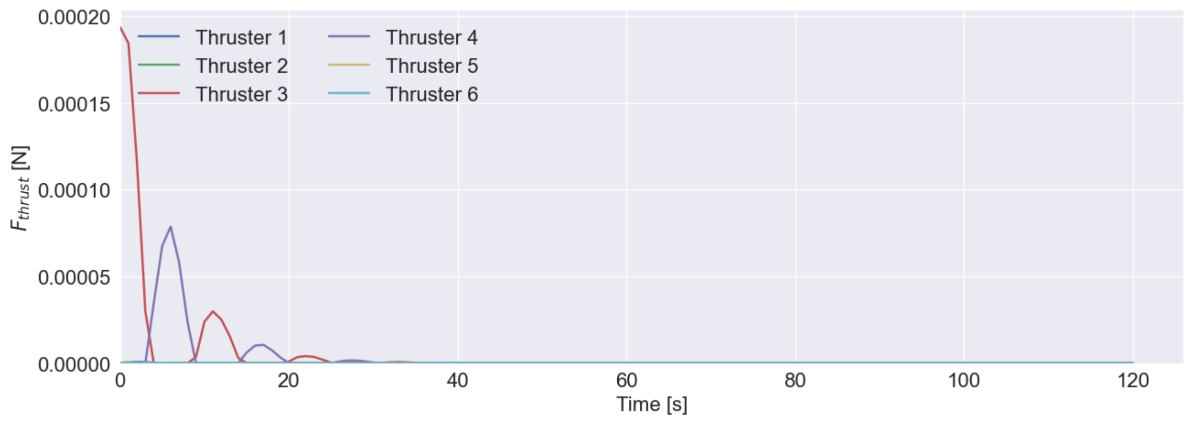


Figure 6.79: De-tumble test 1: F output for all thrusters over time in days, analysed from [00:00:00 01-01-2023] until [00:02:00 01-01-2023].

Finally, the total power and energy consumed by the thrusters is observed in Figure 6.80 and Figure 6.81, in order to provide an idea of the additional resources needed for such manoeuvrers. Having observed this de-tumbling behaviour over a time span of two minutes, all tests could be carried out.

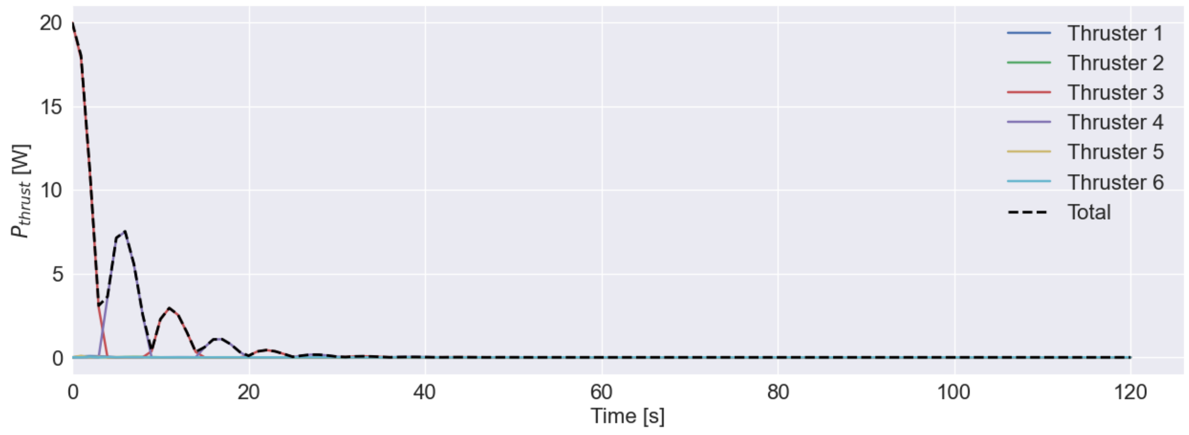


Figure 6.80: De-tumble test 1: P required for all thrusters over time in days, analysed from [00:00:00 01-01-2023] until [00:02:00 01-01-2023].

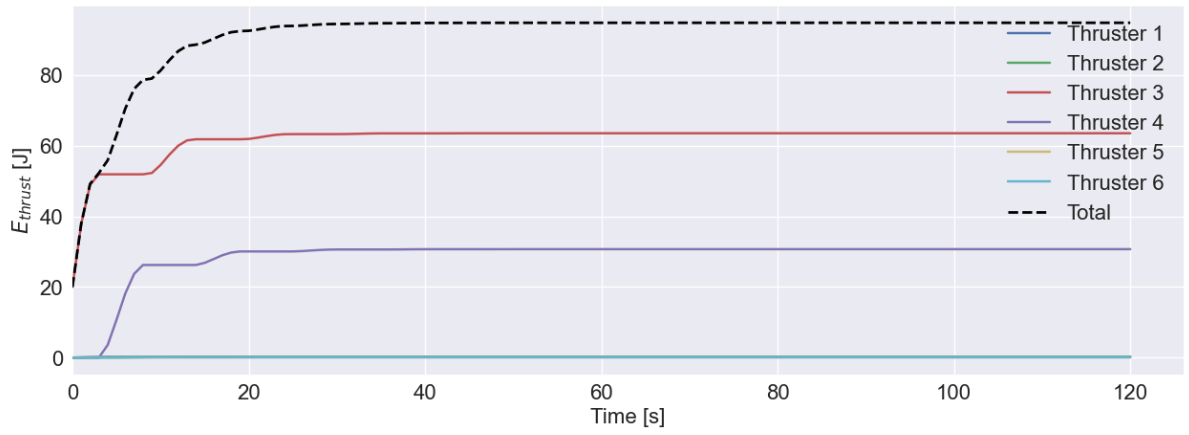


Figure 6.81: De-tumble test 1: E required for all thrusters over time in days, analysed from [00:00:00 01-01-2023] until [00:02:00 01-01-2023].

The results from the de-tumbling tests are observed in Table 6.4, in which five columns are present: the first column gives the test number, the second column shows the thruster numbers that were activated during the manoeuvrer, the third column shows the settling times as defined before, the fourth column shows the maximum power required in Watts, and the last column shows the total energy consumed by the system during the manoeuvrer.

Test number	Thrusters activated	Settling time $[s]$	$P_{tot,max}$ $[W]$	E_{tot} $[J]$
1	3, 4	28	20.00	94.86
2	5, 6	18	18.00	67.10
3	1, 2	29	20.00	85.01
4	3, 4, 5, 6	33	38.00	159.58
5	1, 2, 3, 4	32	40.00	177.49
6	1, 2, 5, 6	25	38.00	149.73
7	1, 2, 3, 4, 5, 6	30	58.00	242.21

Table 6.4: Results for the de-tumbling manoeuvre tests, indicating active thrusters, settling times, maximum power requirement, and total energy consumption. Results are shown for configuration 1.

Finally, the last test, in which an initial de-tumbling manoeuvrer was performed for correction of all three spacecraft primary axes, has been repeated for the other three configurations as well. Their results are shown in Table 6.5 and show a similar pattern in performance as observed in the regular configuration simulations, in which the fourth configuration shows the best results in terms of power and energy. Note that the settling times have been omitted, since these are independent on the configurations as they exactly adhere to the control torque required.

Conf. number	Thrusters activated	$P_{tot,max}$ $[W]$	E_{tot} $[J]$
2	All	40.00	183.84
3	All, except 8	58.00	230.02
4	All, except 11 and 1	34.50	152.94

Table 6.5: Results for de-tumbling manoeuvre test, indicating active thrusters, settling times, maximum power requirement, and total energy consumption, for the last test as shown in Table 6.3. Results are shown for configuration 2, 3 and 4.

Regarding the de-tumbling manoeuvrer results, it was found that a higher de-tumbling rate could be imposed for configurations 3 and 4 (approximately 2 times the rate used for configuration 1), which was to be expected since the addition of more thrusters gives more control capacity. Results for this were omitted because no comparison with other configurations could be made.

In order to assess the attitude control algorithm itself for de-tumbling correction behaviour, irrespective of the actuators used, code was run in order to find the highest initial angular velocity values for 1, 2 and 3 axes at the same time that could still be corrected by the system. The results for these algorithm-only tests are shown in Table 6.6, in which the test initial angular velocity is shown in the first column, the settling time is shown in the second column and the maximum control torque vector is shown in the last column. The angular velocity values seen in the table have been found by iterating over a large number of combinations, assessing the maximum attainable angular velocity for 1 axis (first three tests), two axes (test four, five and six) and, finally, three axes.

ω_0 [$°/s$]	Settling Time [s]	$T_{c,max}$ [Nm]
[10132, 0, 0]	50	[-106.10, 0.00, 0.00]
[0, 222, 0]	26	[0.00, 2.68, 0.00]
[0, 0, 15275]	52	[0.00, 0.00, -160.96]
[115, 115, 0]	35	[-0.52, 1.94, 0.10]
[113, 0, 113]	40	[0.10, 1.93 -0.57]
[0, 123, 123]	37	[-0.01, 1.94, -0.56]
[85, 85, 85]	34	[-0.35, 1.93, -0.42]

Table 6.6: Results for algorithm-only de-tumbling tests. The table shows the initial angular velocity, the settling time, and the maximum control torque vector.

From the table, it can be seen that one-axis stabilisation is possible for excessively high angular rates over the x- and z-axes, totalling to almost 43 revolutions per second for the z-axis. It is understood that this is not a realistic angular rate at which the spacecraft will remain structurally intact, but has been tested in this way for looking at the system response to such extreme situations. The maximum initial angular velocity over the y-axis is significantly lower due to the mass moment of inertia over this primary axis. It can finally be seen that the control torque vectors are indeed larger than can be achieved using the Pocket Rocket thruster configurations as were used in this research. These results can for this reason be used to size appropriate ADCS actuators for such extreme situations.

As a final analysis within de-tumbling manoeuvres, the reaction wheel configuration is assessed. The same approach will be used as shown for the thruster configurations previously, in which the maximum angular rates are tested that can still be corrected by the reaction wheels. The feasibility of correction is based on whether the algorithm still provides a solution for adjusted control torque values based on the reaction wheel capabilities. More specifically, when a certain initial angular velocity needs to be corrected that is within the values shown in Table 6.5, the algorithm will provide a solution that will contain high control torque values. When these values are converted into the required torque per reaction wheel (using Equation 2.43), it is assessed whether one or more reaction wheels require an absolute torque magnitude of 0.007 [Nm] or larger. If this is the case, their values are corrected for this maximum value, and a new control torque is calculated. Within this approach, the maximum initial angular velocities are therefore the ones for which the simulation still gives a solution based on the corrected control torque values.

Having retrieved the maximum initial angular velocities, it could be seen that the inverse pattern is observed compared to Table 6.5: the x- and z-axes now have the smallest possible angular rates, whereas the y-axis shows the largest. When multiple axes require de-tumbling, the maximum rates again show that the values are "spread out" over the axes. With the initial conditions in place, the settling times, maximum total power consumption and total energy consumption of the tests are assessed. Results for this analysis are shown in Table 6.7 below.

Test number	ω_0 [$°/s$]	Settling time [s]	$P_{tot,max}$ [W]	E_{tot} [J]
1	[70, 0, 0]	1710	36	55,476
2	[0, 252, 0]	1580	36	74,920
3	[0, 0, 85]	3460	36	109,444
4	[70, 70, 0]	3190	36	114,925
5	[53, 0, 53]	1705	36	55,307
6	[0, 80, 80]	2640	36	92,391
7	[55, 55, 55]	3220	36	108,027

Table 6.7: Results for the de-tumbling manoeuvre tests, maximum initial angular rate, settling times, maximum power requirement, and total energy consumption. Results are shown for the reaction wheel configuration, making use of the four reaction wheels only.

From these results, it can readily be concluded that regardless of which axis is initially rotated, all reaction wheels are needed for attitude correction. This can be observed from the maximum total power required, which equals the maximum power usage of 9.0 [W] for each of the four reaction wheels. Moreover, the settling times are significantly longer than was observed in Table 6.4. The reason for this is that the reaction wheel capabilities allow for a larger range of solutions, which result in solutions of longer settling time compared to the thruster configurations. With these longer settling times, significantly larger energy consumptions are paired, which is a natural result of the longer activation of the actuators. From the results, it can be concluded that relatively large initial angular velocities can be corrected for using the reaction wheels from the LUMIO ADCS design.

6.3. Experimental Characterisation

This section will show the results for the practical discussed in Chapter 5, which consists of two distinct parts: results for the code porting activities, in which the original Python-based simulation was migrated to the C-based embedded environment, and the actuator connection set-up, in which the embedded system was tested with actual hardware. These results are displayed in subsection 6.3.1 and subsection 6.3.2, respectively.

6.3.1. Code Porting

The results from this experiment include the iteration time per simulation time step, set to one second, and the accuracy of the computed values ($\boldsymbol{T}_c$, $\boldsymbol{q}_{ref}$, and $\boldsymbol{F}_{thrust}$) compared to the original Python simulation. Before presenting these numerical results, a summary of the challenges encountered during the porting process is provided:

- **Data Transmission**: Data sent from Python to the hardware was transmitted as strings, which the C code had to receive bit by bit. This required sufficiently large buffers to store the data and convert it into double-precision floating-point numbers. Similarly, sending computed results back to Python required adaptive functions to handle varying vector lengths, as quaternions, control torque vectors, and thrust values have different dimensions.
- **Memory Management**: Variables such as gain values were reused across iterations, while reference quaternions and angular velocities changed per iteration. To manage these efficiently, a systematic memory allocation strategy was implemented.
- **Thruster Allocation Algorithm**: The thruster allocation algorithm, based on the ECOS solver for linear programming, posed significant challenges. The ECOS library had to be separately downloaded and installed, and modifications to its Makefile were necessary to ensure compatibility with the STM32IDE environment. Integration with the existing project code required further adjustments during the compilation process.

The porting process was successfully completed, starting with the ”1/0” test, after which the control torque PD, reference quaternion and thruster allocation equations were migrated. Aforementioned numerical results are displayed in Table 6.8. It should be noted that the simulation time period for these tests was only the first close fly-by, since this poses a good representation of the rest of the simulation.

The iteration time stands for the average time, over all time steps, the computation was performed in. This time was measured in the Python environment, and data sending and receiving takes place within this time measurement. The base case stands for the regular simulation, without any connection with the MCU. Note that the time values are presented in milliseconds and the absolute difference units are given in the table itself.

In the set-up where only the control torque has been added to the embedded environment, the control torque values were compared to the Python-based values to assess their absolute and relative difference. Then, these values were averaged over the three components of the vector, and averaged over all time iterations. For the quaternion reference, the reference values were compared to the Python case and the absolute and relative differences were computed in the same manner. For the final row in the table, this was done in the same way for the six thruster output values per iteration time step.

Set-up	Iteration time $[ms]$	Absolute difference	Relative difference
Base case	1.5	NA	NA
1/0 Test	4.2	NA	NA
$\boldsymbol{T}_c$	35	$1.460 \cdot 10^{-19}\ [Nm]$	$-2.17 \cdot 10^{-9}\%$
$\boldsymbol{T}_c$ and $\boldsymbol{q}_{ref}$	53	$6.685 \cdot 10^{-19}$	$1.25 \cdot 10^{-15}\%$
$\boldsymbol{T}_c$, $\boldsymbol{q}_{ref}$, and $\boldsymbol{F}_{thrust}$	1050	$6.783 \cdot 10^{-11}\ [N]$	$9.99 \cdot 10^{-3}\%$

Table 6.8: Numerical results for the code porting process in the practical experiment.

As can be seen from the table, the order of magnitude of the errors for the control torque vector and the reference quaternion vector are due to round-off errors, since double precision floating point numbers remain accurate for 15 to 17 decimals. Therefore, it could be concluded that the values for these vectors were equal to the Python-based simulation. The propagation of the round-off errors resulted in the numerical offset seen for the thrust values: since the exact same linear programming solver was used as in Python, and intermediate results were not possible to verify within the solver, this has been concluded.

6.3.2. Actuator Connection

Having performed the experiment with one valve, it could be said that a successful connection between the Nucleo board and the valve power distribution board was assessed. A high-frequency clicking sound (opening and closing) was heard from the valve upon activation of the code, and a square wave signal could be observed from the oscilloscope at the moment of this sound. This confirmed the increase and decrease of the duty cycle values internally, since the activation as also seen from the oscilloscope did not have the same length over the entire simulation. Externally, it confirmed that the board was producing the correct 3.3$[V]$ signal from the designated pin and the timer was therefore working correctly. From the physical behaviour of the valve, it could be concluded that this ”dummy” thruster did indeed respond to the created signals correctly.

It should now be assessed whether the commanded signal, or whether the commanded duty cycle adjustment, was correct. In order to prove this, the actual duty cycle output over time, in percentages, is compared to the thrust output over time, compared to the maximum thrust achievable. The thrust is presented both in absolute numbers and percentages. It should again be clear that the maximum thrust used in this experiment was set to $8.0 \cdot 10^{-7}$ and the MIB to 0.01 $[\mu Ns]$. For the MIB, considering a frequency of 10 $[Hz]$, the minimum thrust that the system will respond to is equal to:

$$F_{thrust,min} = \frac{MIB}{period} = 1 \cdot 10^{-7}[N] \tag{6.4}$$

In this equation, the period is equal to $\frac{1}{f}$ in seconds. The average computational time for one iteration for one valve was found to be equal to the value seen in Table 6.8 since the exact same code was used with the inclusion of the PWM signal adaptation, which did not cause any significant increase in time. The computed thrust outputs from the MCU are presented in Figure 6.82, as seen previously and comparable to Figure 6.34.

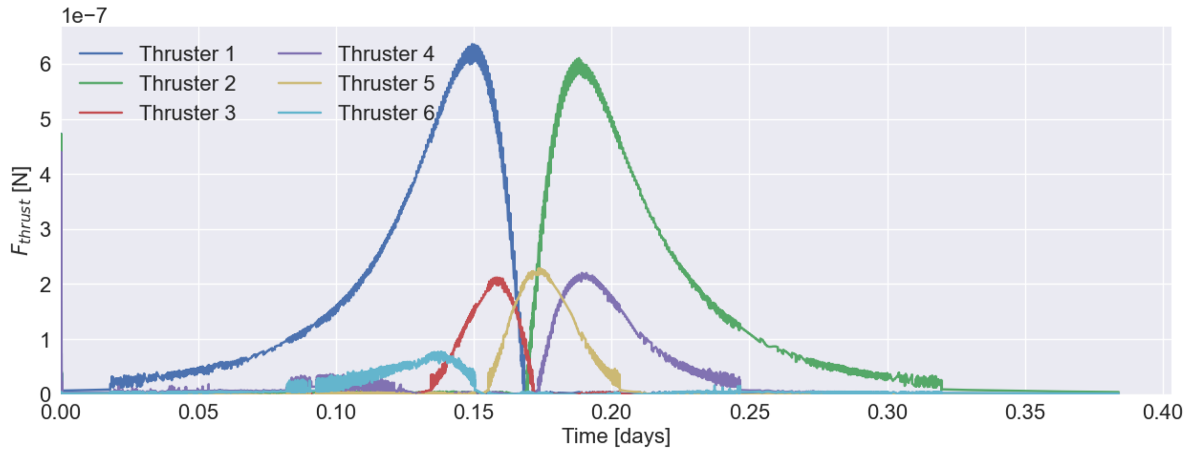


Figure 6.82: Connectivity: F output for all thrusters in configuration 1 over time in days, calculated by the Nucleo MCU, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

Looking at the output thrust for thruster 1, the output that has been connected to the valve and for which PWM signals have been created, the duty cycle for this thruster over time can be analysed, as seen in Figure 6.83. It can be observed that the increase and decrease in duty cycle neatly adheres to the increase and decrease of thrust values, and that the threshold of $1 \cdot 10^{-7}$ $[N]$ is accurately adhered to as well. A more thorough comparison can now be made, in which the percentage the desired output thrust is compared to the duty cycle percentage. This has been shown in Figure 6.84. Note that the large values on either end of the oscillatory pattern can be attributed to the exclusion of the MIB requirement for the desired signal. This is obviously a physical limitation, so is not regarded in the theoretical desired output.

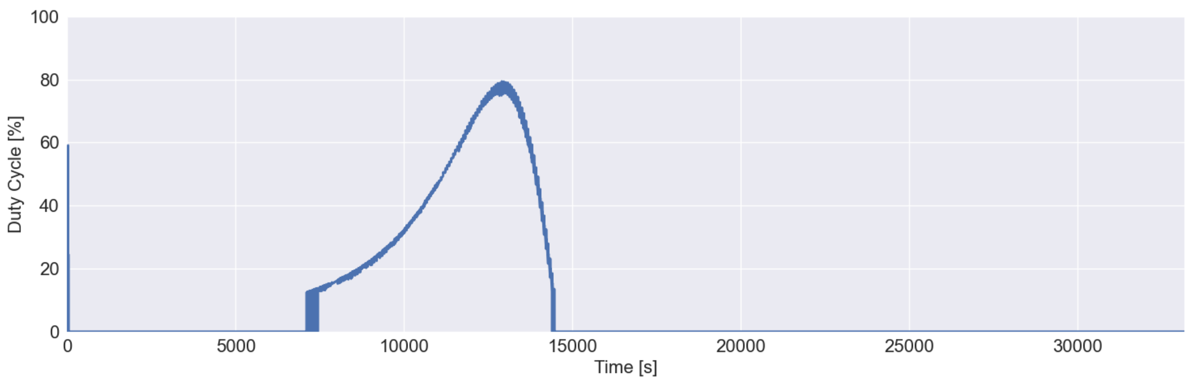


Figure 6.83: Connectivity: duty cycle in % for thruster 1 in configuration 1, as measured from the output signal to the valve, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

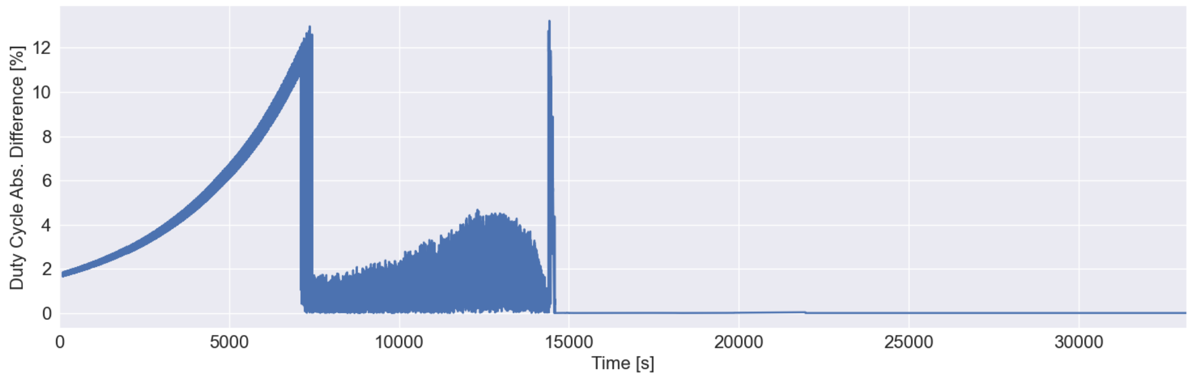


Figure 6.84: Connectivity: difference between the commanded duty cycle in % and the actual duty cycle in %, as measured from the output signal to the valve, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

Finally, the duty cycles of all the thrusters, using six separate PWM channels, can be shown in Figure 6.85. It can be seen that the thrust output for thruster 6 does not yield any duty cycle over the simulation period. This is due to the MIB requirement set to the simulation and will lead to a loss of accuracy in control torque generation. Using different settings for MIB (so, using different hardware) may solve this. The maximum percentual offset per thruster has been presented in Table 6.9, disregarding the large spikes shown for the Minimum Impulse Bit requirement.

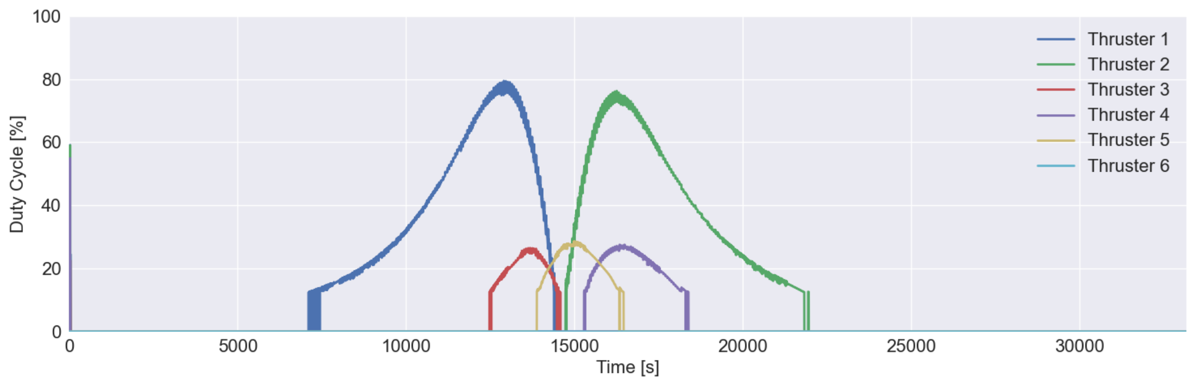


Figure 6.85: Connectivity: duty cycle in % for all thrusters in configuration 1, as measured from the output signals of the PWM channels, analysed from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

Configuration	1	2	3	4	5	6
Maximum Δ DC	4.67%	4.91%	1.28%	1.49%	2.06%	0.0%

Table 6.9: Maximum difference in duty cycle between commanded, theoretical, and output from PWM, disregarding MIB requirement differences.

6.4. Verification & Validation

The results presented in this chapter require both verification and validation to ensure mathematical correctness and adherence to real-world systems. This section explores these aspects, focusing on the methods used for the main attitude control simulation, robustness tests, and the practical experiment conducted as part of this research. The practical experiment serves as a validation case for the ADCS simulation, applying its methods to existing hardware, while the robustness tests verify the algorithm's performance under extreme conditions. These techniques complement the approaches discussed earlier and are divided into verification techniques (subsection 6.4.1) and validation techniques (subsection 6.4.2).

6.4.1. Verification

Algorithms are composed of various building blocks, each of which may include smaller blocks or functions, creating a hierarchical and clear structure. Unit testing is a widely used technique for verifying the correctness of these individual components, ensuring that underlying mathematical operations are accurate. During the development process, unit tests were extensively applied to validate nearly every mathematical operation, comparing results against established libraries or manual calculations. While detailing all the unit tests would be exhaustive, several key tests that were particularly important in ensuring the algorithm's correctness are highlighted below, for the adherent parts of the code.

- `Rotation` **class**: The functions in this class were used for applying specific Euler angle and quaternion rotations, creating direction cosine matrices and converting quaternions to Euler angles and vice-versa. All functions were mathematically verified using the `scipy.spatial.transform` library, which contained the exact same functions. In addition, rotations were tested by rotating an arbitrary 3D vector over a specific set of quaternions, and performing the reverse rotation in order to verify the exact same 3D vector was obtained (or: verify that RR^{-1} is equal to the identity matrix). Additional tests for these rotations include verifying that $det(R) = 1$ and $R^T R$ is also equal to the identity matrix.
- **Quaternion operations**: Different functions were created for the calculation of the quaternion error, the quaternion product and the time derivative of the quaternion. All these functions have also been developed in the dedicated `pyquaternion` library, and each of these was tested against the same functions from the library.
- $\boldsymbol{T_{GG}}$ **and** $\boldsymbol{T_{SRP}}$: Although no direct libraries were available for the verification of these disturbance torques, which are essential in the attitude control simulation, verification was performed in two ways. First of all, the equations used for the disturbances were taken from Rizza et al. [41] and reproduced. Next, a number of hand calculations were performed in order to assess their magnitude and therefore validity throughout the algorithm. As will be seen after the unit tests, the functions were tested for their behaviour with closer proximity to the Moon and the Sun, respectively.
- **Linear Programming Solver**: The ECOS solver for the linear programming solution was part of the `cvxpy` library for Python. This library is extensively used throughout research and therefore acts as the verification of the solution for the thrust outputs itself. In addition, the ECOS library installed for the embedded environment served as additional proof of its proper functioning.
- **Quaternion normalisation**: The magnitude of each quaternion vector should be equal to one, as per their definition. Throughout the code, normalisation checks (`np.linalg.norm()`) have been inserted to assure this.
- **Ephemeris data conversion**: The linear interpolation applied to the retrieved ephemeris data was verified by ensuring that the interpolated data points remained within the range defined by the original data points. Specifically, the interpolated points between full-minute intervals were checked to confirm they fell between the values of the nearest surrounding full-minute data points.
- $\boldsymbol{T_c}$ **order of magnitude**: The preliminary results of the control torque computed by the proportional-derivative controller have been checked for their order of magnitude. This is relevant, since these values should lie in the approximate same order of magnitude as the disturbance torques, with an allowed offset of a thousand times larger, primarily due to reference quaternion adjustments. This factor thousand was also observed and allowed in recent research (Cilliers, Steyn, and Jordaan [12]). As could be seen from Figure 6.12 and Figure 6.14, this was verified as well.
- **RK-4 integrator**: To enhance the accuracy of numerical integration within the simulation, the RK-4 integrator replaced the simpler Euler integrator. While the Euler method was initially implemented due to its simplicity, its accuracy was quickly assessed using quaternion results. Upon introducing the RK-4 integrator, two verification techniques were employed to validate its implementation: First, the RK-4 integrator was tested on a standard ordinary differential equation (ODE) with a known analytical solution: $\frac{\delta y}{\delta t} = -y$. The global truncation error was calculated and confirmed to align with the theoretical order of accuracy, $\mathcal{O}(\Delta t^4)$. For an integration time step of 0.1 [s], the error was found to be in the range of 10^{-8}, demonstrating the expected high accuracy of the method. Second, the results of the RK-4 integrator were compared against those of the Euler integrator. The Euler method, as expected, exhibited a larger error, on the order of 10^{-3},

for the same time step and simulation conditions. This confirmed the superior accuracy of the RK-4 integrator. The visual comparison of the integrated values using the Euler method, RK-4 method, and the analytical solution can be seen in Figure 6.86. The figure clearly illustrates how RK-4 more closely matches the analytical solution over the total time span of 5 [s].

- **Single thruster failure**: Output thrust values were printed in order to ascertain that, during the single thruster failure experiment, the correct thruster did not output any thrust.

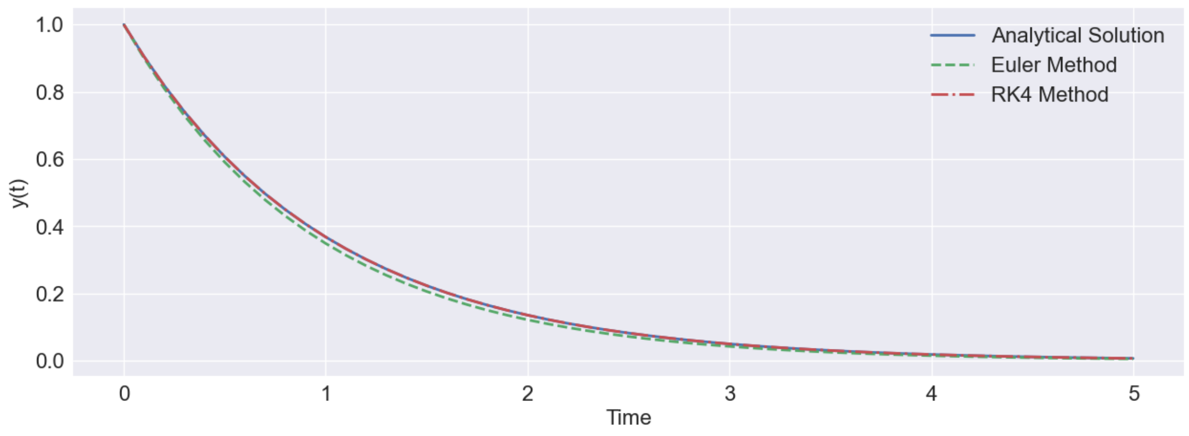


Figure 6.86: Results for the integrated values of the Euler method, the RK-4 method and the analytical solution of the ordinary differential equation $\frac{\delta y}{\delta t} = -y$. Total time span was 5 [s], integration time step was 0.1 [s].

After all the unit tests had been done and verified, it could be concluded that the underlying functions were correct. This did not, however, directly imply the overall system was computing what it should be computing, which is why an additional number of tests were introduced. Firstly, the orbit retrieved from JPL Horizons needed to be checked for adherence to the orbit laid out by NASA in their research towards the mission. This could be done by visualising the orbit as was done in Figure 4.1a and Figure 4.1b. In order to verify that the aforementioned time periods of [05:33:20 05-01-2023] until [14:46:40 05-01-2023] and [00:00:00 12-01-2023] until [05:26:40 12-01-2023] did indeed coincide with the close fly-by manoeuvrers during the entire simulation, Figure 6.87 and Figure 6.88 show these periods of time plotted.

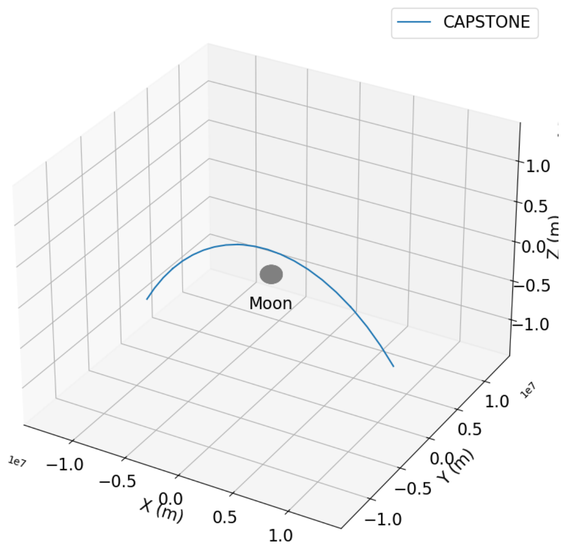


Figure 6.87: CAPSTONE orbit used for this research based on JPL Horizons retrieved ephemeris data, plotted from [05:33:20 05-01-2023] until [14:46:40 05-01-2023].

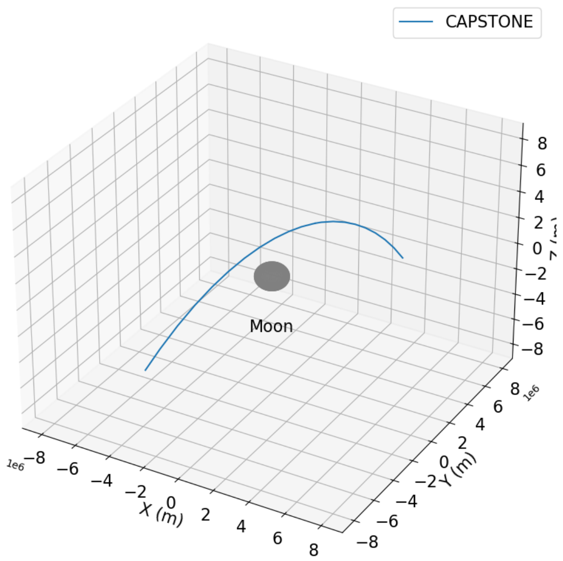


Figure 6.88: CAPSTONE orbit used for this research based on JPL Horizons retrieved ephemeris data, plotted from [00:00:00 12-01-2023] until [05:26:40 12-01-2023].

The second test to assess the functioning of the overall system was testing the response of the attitude control algorithm to an arbitrary reference quaternion value. This reference quaternion was generated using the `np.random.rand()` function and normalised to fulfil the quaternion definition. Then, it was kept constant over a dummy simulation time span. The same disturbance torques were not included in this analysis and the gains were kept equal compared to the gains used in simulation. The result of this test can be seen in Figure 6.89 to Figure 6.92.

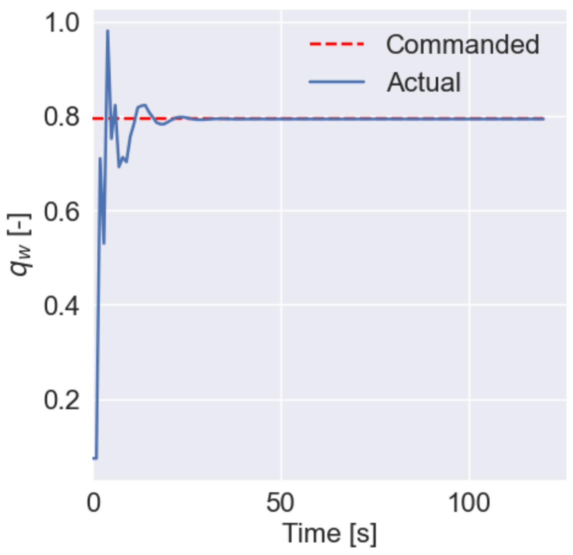


Figure 6.89: q_w versus $q_{w,ref}$ over time.

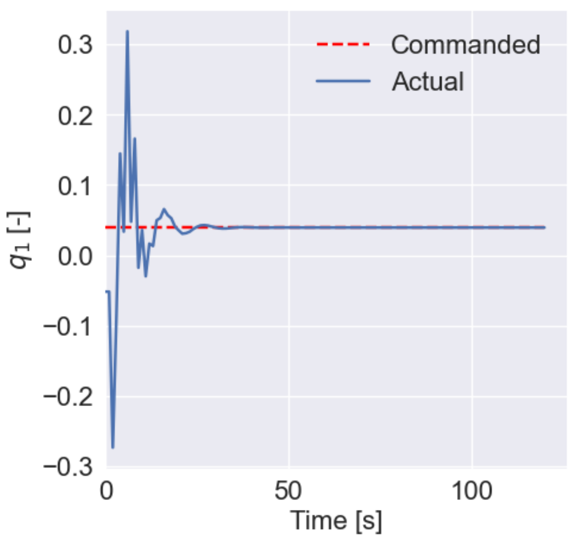


Figure 6.90: q_1 versus $q_{1,ref}$ over time.

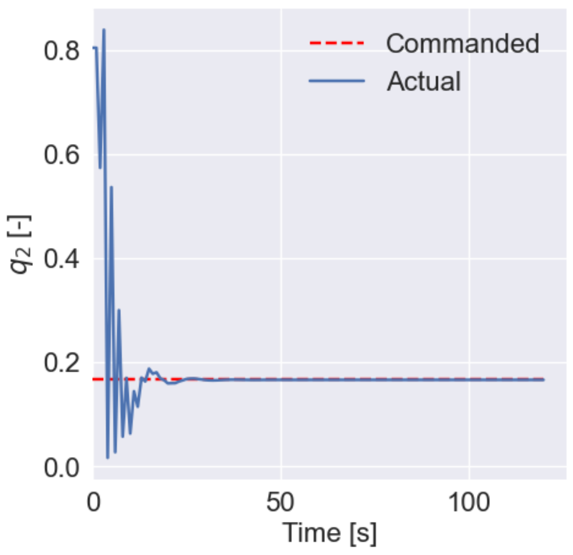


Figure 6.91: q_2 versus $q_{2,ref}$ over time.

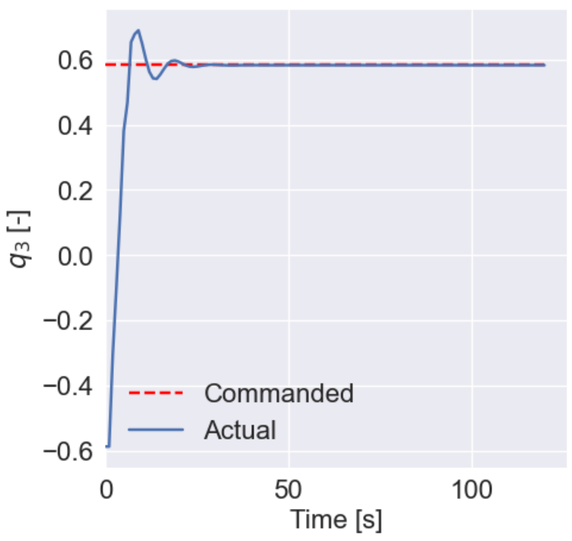


Figure 6.92: q_3 versus $q_{3,ref}$ over time.

As can be seen, the functioning of the control system was hereby verified and the reference quaternion for the research could be introduced, and tested again without any disturbance torques present. After the proper functioning with a varying reference quaternion was also confirmed, the disturbance torques were introduced and results of a short time period were assessed (close fly-by) to see whether the actual attitude state of the spacecraft adhered to the desired state. After this was verified as well, the question arose whether the reference quaternion from the paper was actually correct. In order to assess this, the spacecraft location at each time step was converted into a body-centred Moon pointing vector, pointing exactly at the centre of the Moon. This pointing vector was then compared in its orientation with the location of the spacecraft negative y-panel (coordinates in the body frame again), which resulted in the half-cone angle as displayed in previous sections. For this reason, the computation and analysis of the half-cone angle serves in itself as a verification method for the correctness of the reference quaternion.

The gravity gradient torque and solar radiation pressure torque values were then computed for extreme scenarios. In the closer proximity of the Moon, the gravity gradient torque should increase with the cube of the decrease in distance to the centre of the Moon. For the solar radiation pressure, this increase scales with the square of the distance to the centre of the Sun. For the gravity gradient torque, the value was computed at the surface of the Moon ($r_{\text{SC/Moon}} = [R_{Moon}, 0, 0]$), and for the solar radiation pressure, the value was computed at the location of Mercury, the closest planet to the Sun in the solar system. The absolute distance between the Sun and Mercury is approximately 0.4 $[AU]$, so the spacecraft position with respect to the Sun is taken to be $r_{\text{SC/Sun}} = [0.4AU, 0, 0]$ for this test. The results are displayed in Table 6.10. The values for the CAPSTONE orbit as shown in the table have been computed for the first close fly-by manoeuvrer, at the time where the gravity gradient torque was largest. The results correctly show increase in the magnitudes of both disturbance torques, with the expected increase scale.

Case	T_{GG} $[Nm]$	T_{SRP} $[Nm]$
CAPSTONE	$[-1.17 \cdot 10^{-9}, 3.08 \cdot 10^{-12}, 5.19 \cdot 10^{-9}]$	$[-1.0 \cdot 10^{-20}, 2.0 \cdot 10^{-20}, 2.0 \cdot 10^{-20}]$
Extreme	$[-1.12 \cdot 10^{-7}, -1.25 \cdot 10^{-7}, -9.67 \cdot 10^{-8}]$	$[-6.32 \cdot 10^{-20}, -7.33 \cdot 10^{-20}, 9.54 \cdot 10^{-20}]$

Table 6.10: Gravity gradient torque and solar radiation pressure torque values for regular CAPSTONE orbit and extreme situations.

The actuator implementation also had to be verified. Having performed unit tests on the linear programming solution, as well as on the reaction wheel configuration matrix computations, the resultant thrust and torque outputs for the actuators needed to be checked to lie within the constraints imposed on the system (maximum thrust of 200 $[\mu N]$ and maximum torque of 0.007 $[Nm]$. Moreover, the thrust outputs could never be below 0 and the reaction wheels should remain within their angular momentum limits. This was all verified by analysing the results from the previous sections, both visually and numerically. With this verification, it was also proved that the system would be able to properly function with electrical thrusters, irrespective of power or other systems engineering constraints. Finally, the timing of the thrust outputs, as was observed in many of the graphs in the previous sections, was compared to the timing of the control torque required and based on that, it could be seen that the thrusters were indeed activated when control torque was necessary.

In order to assess the control algorithm's functioning for different accuracy levels, the time steps were decreased and compared to each other. Four different simulations were run, with time steps of 1, 0.1, 0.01 and 0.001 $[s]$. The resultant quaternions all converged towards stability with the same quaternion error values as a result. Simulation execution times increased with every reduced time step, and were analysed on a 300-second total simulation run. For each decreased step, the simulation time increased by a factor 10 approximately (1.5, 8.7, 77 and 772 $[s]$. Choice for the 1-second time step throughout the regular simulation was based on this difference in computation time.

The verification results for the code porting part of the practical have already been shown in Table 6.8 and confirmed the proper functioning of the code in C.

6.4.2. Validation

The main question that needs to be answered by validation is "Does the system that is created, actually represent a real-life system?" and all its aspects should therefore be assessed for whether the results make any sense. As mentioned previously, the practical results serve as a main validation tool and will therefore not be repeated here. Furthermore, a number of verification tests also included validation: the fact that the reference quaternion direction cosine matrix was taken from the LUMIO ADCS paper, validated its relevance to the real-life scenario. The main validation techniques that are left for this section, are the validation of the dynamic attitude control system as a whole, as well as the mission requirement and budget checks for the thruster and reaction wheel analyses.

In order to assess the control system as a whole, a simplified case will be studied. The reference quaternion will be kept constant, so that no control torque should be required for attitude control. Then, three dummy input disturbance torques will be exerted on the system: one disturbance torque purely over each of its primary axes, with a magnitude of $1 \cdot 10^{-2}$ $[Nm]$, in order to be able to visually assess its effect. The system state response will then be analysed as a sanity check on what a real-life system would exert. Other disturbance torques will be turned off.

Within a 300-second simulation, the dummy disturbance torques were activated at 80, 160 and 240 seconds, respectively, for a total duration of 10 seconds. They were executed in the order x, y and z axis. The results in the control torque and the angular velocity response of the system can be seen in Figure 6.93 and Figure 6.94 below. From the torque graph, it can be seen that for all axes, a positive disturbance gives rise to a negative control torque with a similar magnitude. This is in any real case to be expected for control of the attitude of a rigid body. Moreover, after the ten-second disturbance interval is over, extra torque in the opposite direction is needed to counter-act the initial correction and come back to the equilibrium state. The control about the y-axis shows a more intense correction mechanism, which is due to its lower mass moment of inertia and therefore lower resistance against changes in angular velocity, for the same PD gains. The mass moments of inertia about the x- and z-axes are similar and their response therefore show similar results.

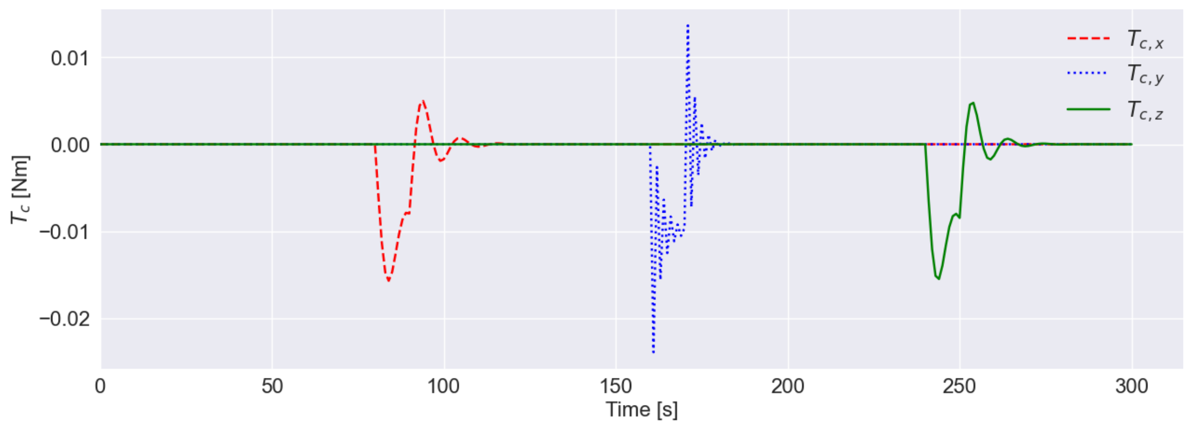


Figure 6.93: Control torque over time for a 600-second simulation with disturbance torques exerted over the x-, y- and z-axes, at 80, 160 and 240 seconds, respectively.

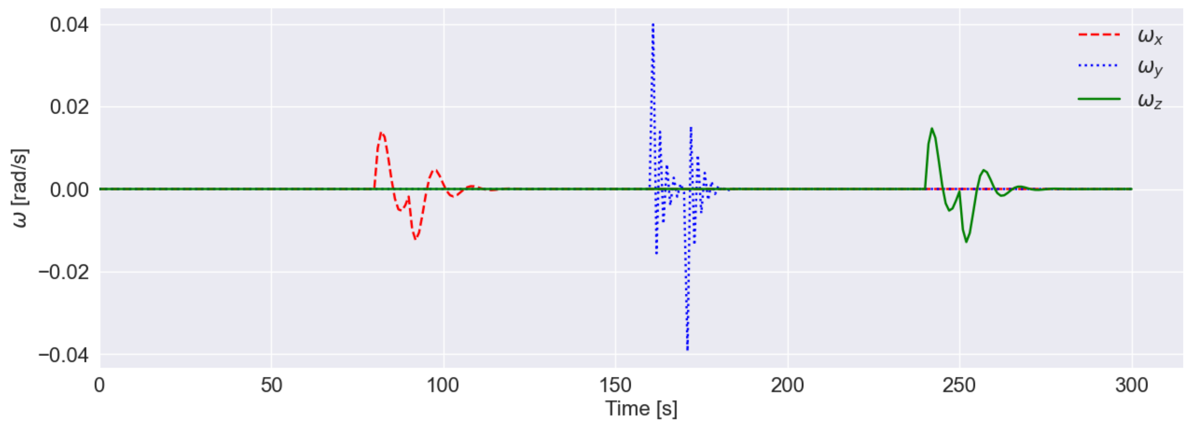


Figure 6.94: Angular velocity over time for a 600-second simulation with disturbance torques exerted over the x-, y- and z-axes, at 80, 160 and 240 seconds, respectively.

In the angular velocity graph, it is clear to observe the positive angular velocity initially attained due to the positive disturbance torque for the x and z axes. When the control torque has attained its maximum value, the angular velocity has decreased towards zero again, after which it keeps oscillating until convergence to the equilibrium point. The negative angular velocity is due to the overshoot of the control torque. Again, for the y-axis, strong oscillatory behaviour is seen due to its lower inertia. Using half of the original gains for the y-axis only solved this oscillatory behaviour, as can be seen in Figure 6.95. All in all, the results for this validation test confirmed that the system indeed reacts to external disturbances as one would expect any rigid object to respond. For this reason, validation of the dynamic system can be confirmed.

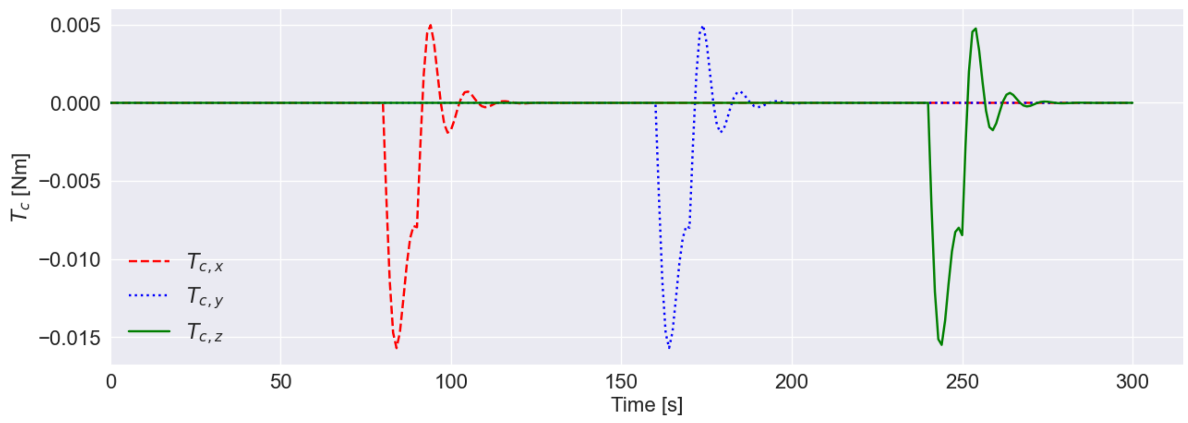


Figure 6.95: Figure 6.93 with adjusted gains for the y-axis.

It should be noted that, in order to perform a more complete validation of the developed algorithm, results from comparable research should be discussed and compared. It was unfortunately concluded that this research was not openly available to be used in this report. The results for the LUMIO mission, and particularly its ADCS, were available as seen in, for example, Romero-Calvo, Biggs, and Topputo [43] and Rizza et al. [41], but included the disturbance caused by the main engine for orbit keeping. Since this was not included in this research, direct validation was not possible with these results. Moreover, the ADCS analysis for the CAPSTONE mission was not sufficiently detailed in literature to be useful for validation. For this reason, a validation gap is still present in this report and will be added to the final recommendations for future study.

7 Discussion

7.1. Introduction

With the results of the previous chapter, a large amount of information has been collected. By means of sanity checks, verification and validation tests, and pure observations, a number of conclusions could already be drawn from these results. It should also not be forgotten that preliminary results were used to improve the attitude control algorithm developed in this research, so that the final results as displayed previously could be found.

Naturally, the information provided by the obtained results should be discussed, so that potential problems are identified, or new insights may be found. This chapter will focus on the discussion of these results, and will draw the main conclusions of this research from them, in order to approach the research questions presented in Chapter 3. This will be done in a structured way. First of all, all results from the simulation part of the research, including the reaction wheel, thruster and robustness analyses, will be elaborated upon in Section 7.2, and the result adherence to the LUMIO mission requirements will be assessed. Then, a closer look will be given to the results from the experimental validation and its main conclusions will drawn. Finally, based on all the methods used, results obtained and conclusions drawn, recommendations for future work will be presented, so that any reader of interest may continue on this intriguing topic.

The final conclusions will be used to answer the research questions in Chapter 8. In this chapter, the hypotheses will also be tested.

7.2. Simulation

This section will analyse the results for the entire algorithm in general first, after which the reaction wheel, thrust and robustness analyses will follow. Therefore, all conclusions that follow from the results of the Python simulation are presented here and elaborated upon. Reference will be made to figures in the previous chapter.

7.2.1. General

In the general results of the algorithm, Figure 6.4 to Figure 6.7 showed good adherence of the spacecraft state to the desired state. The varying reference quaternion pattern can be explained based on the reference DCM, which points the negative y-axis of the spacecraft towards the centre of the Moon, whilst keeping the positive x-axis perpendicular to the pointing vectors to the Sun and the Moon, maximising the power yield of the solar arrays. A close fly-by with these requirements set out, requires the spacecraft to quickly adjust its attitude in order to follow, which is clearly seen from the spikes observed in q_w and q_1, and the sudden sign changes in q_2 and q_3.

The absolute quaternion error shown in Figure 6.8 confirms the previous statement on good adherence: the only significant error occurs during the fly-bys. In order to make this more insightful, Figure 6.9 was created. A number of large error spikes are present that can be ignored since they are related to quaternion sign changes. In fact, the overall relative error is only in the range of 0.001% during the fly-bys, which again confirms the good adherence of the quaternion components to the reference. In order to put this in a more understandable perspective and compare these results to a clearer benchmark, Figure 6.2 was presented to be able to assess the offset of the camera towards the desired centre of the Moon location. The error in the angle, which is well below the requirement from the LUMIO mission, only occurs during the fly-bys as well.

The maximum angular rate observed in Figure 6.11 also occurred during the fly-bys, was equal to 0.0248 [$^\circ/s$] and therefore below the requirement for the maximum slew rate (**ADCS-03**). From this angular velocity analysis throughout the two-week simulation, it could be concluded that no uncontrollable angular velocities will be present within the spacecraft. The non-zero angular velocity values observed were required for correct attitude control and therefore desired.

The magnitudes of the gravity gradient torque and solar radiation pressure torques (Figure 6.12 & Figure 6.13) were significantly low compared to the required control torque. The order of magnitude of the solar radiation pressure was insignificant to the control torque. Its magnitude varied for two fly-by scenarios due to its attitude with respect to the Sun: it was exposed to the Sun more significantly during these fly-bys. The gravity gradient torque itself varies with the distance to the centre of the Moon, as explained and tested before. Its magnitude was 1000 times smaller than that of the control torque, which also makes it have a minor influence on the required torque, and no realistic influence on the required power from either actuator system. For these reasons, it could be concluded that the external disturbances used in this research for the NRHO are insignificant for the calculation of control torque and power requirements for actuators. The difference with the quasi-periodic orbit for the actual LUMIO mission will be that the gravity gradient torque is even smaller. The SRP difference is negligible since the distance towards the Sun will not vary, as was proven in Table 4.2. Over a longer time span, of e.g. a year, these disturbance torques may cause significant angular momentum to build up in reaction wheels, and equally require linear impulse from non-electrical thrusters as reaction control system. For an ADCS with electrical thrusters only, however, this should not call for any additional design considerations, since no saturation of any case can be present. The only limitation observed for the configurations would be the power required for firing, so that the total necessary impulse can be achieved.

Because of this conclusion with respect to the disturbance torques, it could be concluded that the control torque as observed in Figure 6.14 is mainly due to the change in reference quaternion, especially during the close fly-by manoeuvrers, where the control torque values are largest. The time steps during which the control torque values are most prominent, the quick changes in state vector also occur (Figure 6.4 to Figure 6.7). It should again be noticed, that the quasi-periodic halo orbit designed for the LUMIO mission will not show this quick reference quaternion behaviour; since it will not have close fly-by manoeuvrers, its reference quaternion will remain relatively equal throughout its mission. Based on the results in this research, it can be concluded that the final LUMIO mission will also be able to attain its correct attitude using the control algorithm explained here, whilst adhering to the mission pointing requirements.

Looking at the control torque necessary during the close fly-bys in Figure 6.15 and Figure 6.16, the behaviour of the attitude control system can be explained. During the first close fly-by, the control torque is primarily focused on corrections around the z-axis, with minimal adjustments around the y-axis. This indicates that the spacecraft is performing a rotation about its z-axis, likely to ensure that the onboard camera remains oriented towards the Moon during the manoeuvrer. In the second close fly-by, a similar behaviour is observed, but this time the corrections are predominantly around the x-axis. This difference arises due to the different approach angles of the spacecraft relative to the Moon during the two fly-bys, as seen when visualizing the spacecraft's orbit in the Moon-centred frame (Figure 6.87 and Figure 6.88). Despite the changes in orientation relative to the Moon, the solar panels continue to point towards the Sun, which remains approximately in the same position relative to the spacecraft after 7 days. This consistency occurs because, while the Moon has moved about one-quarter of its orbit around the Earth in that time, the Sun's position relative to the spacecraft remains largely unchanged.

Consequently, the attitude control system needs to balance the requirements of both targeting the Moon during the fly-bys and maintaining solar panel alignment towards the Sun. This dual objective explains the observed torque corrections and highlights the effectiveness of the control system in achieving the desired attitude state.

Before continuing with the specific reaction wheel analyses, it should be noted that in the original LUMIO ADCS analysis, the disturbance torques due to the main engine firing (used for orbital insertions and orbit keeping) were also included in the attitude control algorithm. For this reason, the results in the upcoming sections will not match the expected results from the LUMIO papers. The reason for not including these disturbance torques was the fact that the NRHO was assumed to be a stable orbit in which analysis could be done without the need for orbit keeping. Upon reviewing Rizza et al. [41], using the information on the parasitic torque due to the main engine firing within the Monte Carlo simulations performed, the maximum torque that has to be corrected for has an order of magnitude of $1 \cdot 10^{-2}$ $[Nm]$. The remainder of this analysis will be left as a future recommendation.

7.2.2. Reaction Wheel Analysis

From the required torques per reaction wheel, displayed in Figure 6.17, Figure 6.18 and Figure 6.19, it can be observed that the reaction wheel torques correctly align with the required control torques during the simulation. Reaction wheel 1, for example, shows a positive and then negative torque exerted during the first fly-by, which is the negative of the control torque over the x-axis during that time period. This can be seen for all the wheels in both fly-bys. In addition, reaction wheel 4, which is aligned with all the axes, takes up part of the required control torque and therefore carries part of the load for the other wheels. This is best seen for the negative control torque needed in the second fly-by by over the x-axis, which translates into a positive torque for reaction wheel 1 that is opposite in sign, and slightly lower than the control torque value in magnitude. In this way, it can be concluded that adding additional reaction wheels to a determinate system not only gives provides redundancy, but also decreases the power required per wheel.

It should be noted that the reaction wheels will most probably not have the required resolution for the considerably small control torques necessary in this analysis. Due to the lack of information on the reaction wheel resolution, and considering a standard resolution of 1%, the minimum available torque by these reaction wheels will be too large for the required torques in this research. For this reason, the simplified disturbance environment that was presented in this research is not suitable for extensive reaction wheel analysis. For the sake of comparison to the thruster cases, however, the results have still been used.

In the total power and energy graphs, observed in Figure 6.20 to Figure 6.23, it can be seen that the reaction wheels aligned with the x- and z-axes demand most power and energy from the spacecraft. This aligns again with the attitude observations during the fly-by, which also showed significant rotations over these axes for pointing the y-axis towards the Moon and was also expected beforehand. As was mentioned in subsection 6.4.2, the power required will fall well within the power budget of the LUMIO mission, since the required system power values are small compared to existing other ADCS analyses. Since the exact results from the research in the paper and this report do not align, this research cannot add any information to the currently known ADCS research done within the LUMIO mission ([41]).

In line with the difference in power required, the angular momentum analysis as shown in Figure 6.25 does not show the expected angular momentum build-up as was seen in Rizza et al. [41]. Again, the reason for this is the exclusion of the parasitic torque induced by the main engines, which is not included in this research. What can be concluded from the angular momentum build-up analysis, however, is that over a span of 14 days, the maximum magnitude of angular momentum stored in wheels 1, 2 or 3 is equal to $6 \cdot 10^{-4}$ $[Nms]$ which is well below the saturation limit of 0.1 $[Nms]$. In addition, after this value has been attained, the magnitude reduces again. None of the momentum wheels reached saturation limit over a 14-day time span, from which it can be concluded that no momentum wheel desaturation is needed within the nominal simulation bounds of this research for a two-week time span. For more extreme situations, however, in which de-tumbling manoeuvres or other disturbances are present, the reaction wheels will most likely require additional actuators for desaturation. Over a longer simulation time span, such as a full year, it may also be necessary to include actuators for desaturation. Therefore, for realistic mission scenarios, these actuators will definitely be needed. This will be added as a recommendation for future work.

The reaction wheel analysis demonstrates that the torques produced by the wheels closely align with the required control torques during both fly-bys. The inclusion of a fourth reaction wheel provides redundancy and reduces the load on individual wheels, improving overall power distribution. The power and energy consumption analysis confirms that the reaction wheel system operates well within the LUMIO mission's power budget. It is not completely certain whether the actual reaction wheel resolution will be sufficient for the correction of such small disturbances, and from a preliminary estimation it is assumed that they will not be able to. Regardless, this analysis is used as a comparable case to the electrical thruster configurations ADCS. Additionally, the angular momentum build-up in the wheels remains minimal, with no wheel saturation observed over the two-week simulation period, eliminating the immediate need for desaturation manoeuvrers. However, it is recommended to assess long-term behaviour over the mission's full duration, as momentum limits may be reached over a year-long time frame. Moreover, extreme mission situations will most definitely require additional actuators for desaturation, which is why the inclusion of these is recommended.

7.2.3. Thruster Analysis

The thrusters configurations will be analysed one by one, after which conclusions on the set-ups and application in the simulation will be given. After this, the discussion on the robustness tests will follow.

For **configuration 1**, it could be observed from Figure 6.28 to Figure 6.36 that the first fly-by required the most amount of thrust from thrusters 1 and 2, with values of $6.60 \cdot 10^{-7}$ and $6.31 \cdot 10^{-7}$ $[N]$, respectively. Looking at the thruster set-up from Figure 4.3, in which it can be seen that these two thrusters are located on top of the CubeSat (+y) and pointed at both positive and negative x directions, a pure rotation about the z-axis is caused by firing either one of them. Linking this to the control torque required from Figure 6.14, the firing of thruster one for the first half of the close fly-by does indeed cause the required positive control torque over the z-axis, and the firing of thruster 2 for the second half of the fly-by does indeed cause the negative torque over the z-axis. The other thrusters are fired at a lower thrust level, and will cause the torques necessary about the remaining axes. Since configuration 1 is constructed so that each thruster will generate either a positive or negative torque about 1 axis only, one will see that from each thruster pair (1&2 or 3&4 or 5&6), only one thruster is activated at the same time. For the second close fly-by of configuration 1, the same can be observed, with maximum thrust values for thrusters 3 and 4, with a magnitude of $6.57 \cdot 10^{-7}$ $[N]$ and $6.45 \cdot 10^{-7}$ $[N]$, respectively. These thrusters cause pure rotation over the x-axis, which explains their firing for the second fly-by, again based on the required control torque from Figure 6.14. The other thrusters also exactly adhere to their pure rotations, and when one thruster of a pair is firing, the other is disabled.

The oscillations observed in all thruster output graphs stem directly from the control torque output, as shown in Figure 6.14. These oscillations arise due to the high precision of the control algorithm and the discrete nature of the simulated system. In the current simulation, the controller rapidly adjusts the torque, resulting in fine variations in the commanded thrust. However, in real-world applications, these small oscillations will not translate directly to the thruster outputs. In practice, the on-board computer (OBC) would average or smooth the control torque over short time intervals to provide a more stable thrust command. This averaging process reduces the impact of rapid fluctuations in control torque and ensures that the physical thrusters do not oscillate as sharply. The averaged thrust values would instead reflect the desired control torque over an extended duration, as opposed to instantaneous changes seen in the simulation results.

Figure 6.37 provides a first indication of the thruster that requires the most power and energy over time. The impulse over time is an important parameter, usually for non-electrical thrusters, since the amount of propellant stored directly determines its maximum allowable linear momentum. For electrical thrusters, however, the impulse is not directly constrained by propellant storage but instead by the available power onboard the spacecraft. The ability of electrical thrusters to generate continuous low thrust over prolonged durations allows them to achieve significant total impulse, despite lower instantaneous thrust compared to non-electrical systems. In this analysis, it can be observed that thrusters 1 and 2 exhibit the highest accumulated impulse over the 14-day period. This result corresponds well with the significant control effort required during the two close fly-bys, as also observed in the torque demands on the spacecraft.

Based on the thrust outputs, the power requirement over this simulation period could be assessed and examined more closely during the two fly-bys, which has been presented in Figure 6.38 to Figure 6.41. The required power values for one thruster, scaled between 0 and 20 [W] based on the total available thrust of 200 [μN], have a maximum of 0.0631 [W] for thruster 1 during the first fly-by and a maximum of 0.0626 [W] for thruster 3 during the second fly-by. Over the entire simulation, as was also seen in the impulse graph, it is thruster 1 and 2 that consume most energy and add approximately 26% of the total energy consumption over the entire simulation, each. This means that more than half of the energy consumed by the thrusters is due to these two thrusters, for the control of the z-axis.

From this analysis of thruster configuration 1, a number of conclusions can readily be drawn. First of all, it can be concluded that the theoretical implementation of the configuration can adhere to the nominal control torque requirements of the attitude control algorithm and showed positive thrust outputs (as the linear programming solution demanded) over the entire simulation. In addition, all thrust values are well within the limits of the 200 [μN] set by the Pocket Rocket specifications and would therefore be feasible to exert. The values are, however, all lower than the Minimum Impulse Bit imposed by the Pocket Rocket, which was equal to 1.18 [μNs]. This means that, given the thruster resolution, the values would all not be feasible to output and the final thruster outputs would be zero. A further discussion on this will follow at the end of this section. Looking at the power requirements for the system, which totals to a maximum of 0.09 [W] during the first fly-by and a total of 0.14 [W] during the second fly-by, it can be concluded that these values are small compared to the total power budget given in the LUMIO mission (3.1% and 4.8%, respectively) and the power should therefore be available for the thrusters to control the spacecraft's attitude.

For **configuration 2**, the adjusted set-up did not show pure rotations over one axis for all thruster pairs any more: thrusters 1, 2, 5 and 6 were responsible for rotations over the z- and y-axes, simultaneously. Therefore, looking at Figure 6.42 and Figure 6.43, the required torque over the z-axis for the first fly-by is now covered by all four of these thrusters. What is interesting to observe, is that the maximum thrust required for this, during the first fly-by, is only $3.94 \cdot 10^{-7}$ [N], exerted by thruster 5 and approximately 40% lower than the required thrust for configuration 1. For the second fly-by, the maximum thrust is exerted by thruster 3 and equal to $6.62 \cdot 10^{-7}$ [N]. Since the largest required torque for this fly-by was over the x-axis, which is not covered by the four thruster 1, 2, 5 and 6, approximately the same thrust value as observed for configuration 1 is required, from thruster 3 again.

The impulse graph in Figure 6.44 provides the first indication of thruster 4 being the most demanding thruster over the entire simulation period. This shows a clear difference with configuration 1: thrusters 1 and 2 are not so demanding over time any more, and a benefit is already observed. This impulse graph directly translates into the total energy consumed over the thruster and the same conclusion can therefore be drawn for this. The maximum power required by thruster 5 during the first fly-by was 0.0375 [W], with a maximum total power consumption of 0.083 [W], and the maximum power required by thruster 3 during the second fly-by was 0.0631 [W] and the maximum total power of the system amounted to 0.107 [W]. From this, it can concluded that configuration 2, having an equal number of thrusters but with different locations, required less power per thruster, and in total, during both fly-bys. From this, it can be concluded that having thrusters coupled for torques about multiple axes at the same time, introduces benefits in terms of thrust needed, power consumption and total energy consumption.

For **configuration 3**, in which configuration 1 was copied and two extra thrusters for torque over the x- and z-axes were added, it can be seen from Figure 6.47 to Figure 6.50 that introducing these redundant thrusters provides benefits. For the first fly-by, it can be seen that thruster 7 takes over a part of the required thrust for thruster 1 and thruster 4, compared to configuration 1. The same can be said for thruster 8, which takes away the required thrust from thruster 2 and thruster 4. Again, thruster 7 and 8 are never fired simultaneously. Above observations make sense: thruster 7 induces positive torque over the x-axis and z-axis, which is exactly what thruster 4 and thruster 1 cause as well, respectively. For thruster 8, which induces positive torque over x-axis and negative torque over the z-axis, it is therefore also logical that it takes away the load of thruster 4 and 2. For fly-by two, the exact same behaviour can be observed, and the increase in thruster 7 is now larger than that of thruster 8, which is exactly in line with the large required control torque over the x-axis observed during this fly-by compared to the one over the z-axis seen in the first fly-by. The largest thrust value from the thrusters during the first close fly-by was $6.28 \cdot 10^{-7}$ $[N]$ for thruster 1, which is a 5% decrease, and for the second fly-by this was $6.62 \cdot 10^{-7}$ $[N]$, which is a slight increase compared to the configuration 1 case.

Looking at the impulse graph for configuration 3 in Figure 6.51, it can be seen that, in contrast to configuration 2, thrusters 1 and 2 pose the largest strain on the system again. Thruster 4, in contrast, is relieved of its required thrust significantly. This again directly translates in the required power and energy consumption of the thrusters. It appears that adding thrusters, coupled to two axes for attitude control, does not provide benefit for all the thrusters in the original system. Further conclusions on the power consumed over time will be given at the end of this section.

For **configuration 4**, the adjusted approach compared to the other configurations shows benefits with respect to the required thrust outputs to adhere to the control torque. The maximum thrust required over the first close fly-by was equal to $3.45 \cdot 10^{-7}$ $[N]$ for thruster 11, and for the second fly-by equal to $6.41 \cdot 10^{-7}$ $[N]$ for thruster 5. Although the maximum thrust output for the second fly-by is in the same range as was seen for the previous configurations, the benefit can be observed from Figure 6.56 and Figure 6.57, when the total linear impulse magnitude for each of the thrusters is compared to the previous set-up. The total accumulated impulse of thruster 8, which has the highest demand of all the thrusters over the entire simulation, is equal to 0.00219 $[Ns]$, while the maximum total impulse for the other configurations all lay above 0.004 $[Ns]$. The behaviour of the thruster firing themselves is more complicated to verify: each thruster is responsible for the torque over two axes simultaneously. For the first fly-by, for example, thrusters 1, 2, 4, 11 and 12 are most prominently activated, which induce torques over the x- and z-axes or y- and z-axes, simultaneously.

Looking at the comparative graphs of Figure 6.58 to Figure 6.62, a number of conclusions can be drawn on the different configurations. First, Table 7.1 shows the maximum total power over all thrusters and the total energy consumed over the entire simulation, compared to the worst-performing configuration, which is configuration 1. The values are shown in percentages that have been saved due to these configurations. From the graphs and this table, it can be concluded that, in terms of power and energy consumption, which directly relates to the thrust requirements of each of the thrusters, the best performance was observed by configuration 4, then configuration 2, configuration 3, and finally configuration 1. It can therefore be concluded that, having a redundant system with multiple redundant thrusters will provide the best system in terms of power and energy required. Having thrusters that are coupled to two axes simultaneously will also greatly enhance performance compared to thrusters that control one axis only. Furthermore, as was seen in configuration 3, a system with thrusters that control one axis only will dominate in case extra thrusters are added that control multiple axes. This is the reason that configuration 3 only performs slightly better than configuration 1. Finally, in terms of pointing angle accuracy, as measured by the half-cone angle offset, the configurations all performed equally well, since the control torque required was perfectly adhered to by all of them. Therefore, all of these configurations are realistic for usage within the LUMIO ADCS as explained in this research, within the boundaries of its assumptions.

Configuration	$\Delta P_{tot,max}$ [%]	ΔE_{tot} [%]
1	0	0
2	-22.79	-17.33
3	+1.15	-6.88
4	-36.51	-29.14

Table 7.1: Relative differences of the maximum total power and total energy consumed per configuration, compared to configuration 1.

At first glance, the thruster configurations do not seem beneficial at all compared to a reaction wheel set-up, with respect to the power consumption differences (kJ range for thruster, J range for wheels). In order to further analyse this, another look is give to Table 4.9, specifically to IDs **SE1** and **SE2**, the power, mass and cost budgets of the entire system can be examined to give extra relevance to this research. Since the correction of the disturbance caused by main engine firing is not included in this research, and the magnitudes of the extra load imposed on the reaction wheels and thrusters could only be guessed, no accurate power budget conclusions can be drawn from this research. It is only clear what the required power is from the results of the research, and that these values lie well within the constraints given by the LUMIO mission. For the mass budget, however, an attempt has been made to review what the influence of using the thruster system would be. The mass of the currently used reaction wheels of type RWp100 is 0.33 [kg] and of type RWp050 is 0.24 [kg]. Looking at the total mass budget from Table 2.4, the ADCS mass without the reaction wheels would become 0.9 [kg]. Using the Pocket Rocket mass of 0.085 [kg], the total mass of the ADCS and the LUMIO spacecraft as a whole can be observed in Table 7.2. Since the RCS thrusters in the original LUMIO ADCS design require propellant themselves, and are not considered in case the entire ADCS is replaced with electrical thrusters, only part of the propellant mass of the original budget should be included: 25% was taken as assumption for this. Note that the bottom row indicates the wet mass difference with the originally calculated LUMIO wet mass of 28.56 [kg] and the new wet mass includes the same absolute margins as was used previously, equal to 3.51 [kg]. In addition, it is assumed that the original thrusters for ADCS, including its two tanks for the two-phase propellant storage, are part of the ADCS mass budget.

Component	Conf. 1	Conf. 2	Conf. 3	Conf. 4
ADCS [kg]	1.41	1.41	1.58	1.92
ADCS [%]	5.15	5.15	5.73	6.88
LUMIO wet mass	27.40	27.40	27.57	27.91
Absolute difference [kg]	-1.16	-1.16	-0.99	-0.65
Difference [%]	-4.06	-4.06	-3.47	-2.28

Table 7.2: Adjusted mass budget for the four different vacuum arc thruster set-ups.

For the nominal operations as were analysed in this report, the electrical thruster configurations can save mass compared to the reaction wheel system. It is recognised, however, that extreme cases such as the de-tumbling of the spacecraft, and the correction for the main engine firing (not included in this research), cannot effectively be covered by the electrical thrusters. For this reason, this conclusion from Table 7.2 can only be made based on the nominal simulation conditions. Using configurations with improved electrical thruster performance (e.g. maximum thrust, minimum impulse bit) should be tested for extreme spacecraft situations, and when successful, another mass budget should be created to assess whether mass saving can actually be realistic by this thruster-only ADCS.

Configuration 1 and 2 both pose a 4.06% mass saving on the overall mass budget. To put this mass saving in perspective, the total launch cost of the Space Launch System Block 1 by NASA amounts to US$2.5 Billion per launch, with a total mass of 27,000 [kg] that can be taken to the Moon, every gram saved equals US$92.59. This means that configuration 1 and 2 could save more than US$100,000. In the above, simplified, mass budget estimations, the effect of extra wires, or structural adjustments has not been taken into account.

Now, the robustness test results can be examined and conclusions can be drawn from them. First of all, during the single thruster failure in approach 1, where the system is aware of a malfunctioning thruster and will compensate using the other thrusters, it could already be concluded that the determinate configurations 1 and 2 would not produce any results for these test. Thrusters 3 and 4 showed that the lack of the thrust output of the most prominent thruster in the configuration was adequately covered by the other thrusters, inducing a total energy consumption increase of approximately +53% for configuration 3 and +5.2% for configuration 4. This was only analysed for the first fly-by. From these results, it can be concluded that an overdetermined system is only able to cover for a single thruster failure, within the current simulation. Moreover, having an increased number of redundant thrusters will greatly enhance the power requirement and energy consumption requirements when one of the thrusters fail.

From the results of approach 2, different conclusions could be drawn. For configuration 3, and looking at Figure 6.70 and Figure 6.71, it is clear that a large increase in the thrust values per thruster was needed to compensate for the incorrectly attained attitude during the first time steps. Thrust values up to 12.1 $[\mu N]$ were necessary for this correction. After this was achieved, the system operated well, until the fly-by manoeuvrer. There, similar increases in required thrust was seen as compared to the first approach, with the main difference being a thrust spike in thruster 2. This spike can be attributed to a sign switch of reference quaternion values, which the malfunctioning system did give as the reference attitude, but did not manage to attain at that time step. Therefore, a large adjustment over a few time steps was necessary to compensate, inducing a large increase in power and energy consumed.

For configuration 4, as seen in Figure 6.72 and Figure 6.73, the exact same could be observed, although different thrusters were now responsible for the thrust spikes. Moreover, since more thrusters were available compared to configuration 3, two thrusters show this spike behaviour, with thrust values reaching 1.35 $[\mu N]$. These results can directly be attributed to the increase in power and energy required as seen in Table 6.1. Finally, as seen from Figure 6.74, the half-cone angle offset compared to the base case even increased for configuration 4, after the closest approach to the Moon was reached. The reason for this lies in the large thrust spikes; after this, the pointing accuracy of the spacecraft increased compared to what the original system was capable of. For configuration 3, this angle offset slightly increased.

From the results of single thruster failure, approach 2, it could be concluded that a determinate system again fails to cover for a malfunctioning thruster, and its attitude propagates out of bounds without a stable solution. Moreover, having more thrusters allows the system to compensate for the incorrect attitude more severely, resulting in a better pointing angle than was present in the original simulation, but inducing a large increase in power and energy required. These power and energy increases would make the system and entire mission fail in real-life situations, compared to the total power budget of the LUMIO mission. For this reason, sensors should always be present within the thrusters so that the OBC is always aware of any potential thruster failures and approach 1 can be adhered to. The non-feasibility of configurations 1 and 2 also show that redundancy is essential within space mission design.

Comparing both approach 1 and approach 2, it is evident that approach 1 would realistically be possible to handle for the LUMIO spacecraft in terms of the required energy. For approach 2, in which the system is unaware of the thruster malfunction, the thrusters would require excessive energy compared to the base case (as was seen in Table 6.1) and is therefore considered unfeasible. Luckily, in real-life spacecraft applications, the system will be aware of any malfunctioning occurring in the system electronics, using built-in sensors and warnings.

From the results for the undeployed solar array configuration as shown in Table 6.2, it could be seen that the power and energy required for attitude control significantly reduced due to the retraction of the solar arrays. This was to be expected, since the mass moment of inertia showed a decrease in all its values along the primary spacecraft axes as well. The average maximum total power decrease amounted to -59% (averaged over all configurations) and the total energy consumed decreased by -62%. The largest decrease was observed for configuration 4, which was to be expected based on the previous results. From this, it can be concluded that the undeployed solar array case relieves significant demands on the system and is most definitely realistic to also be solved by the attitude control algorithm from this research.

Finally, the initial angular velocity test has to be examined. It was, unfortunately, noticed that high angular velocities above very small values ($5 \cdot 10^{-5}$ $[rad/s]$) were not feasible to control using the thruster set-ups in this simulation. Two distinct reasons have been identified. First of all, the thruster maximum force limits do not allow for the creation of the instantaneous control torque necessary for correcting this attitude. Also with lowering the PD gains, uncontrolled motion was still observed and no thruster solution was found. Secondly, the attitude control algorithm is structured in such a way, that it calculates the torque immediately necessary at every time step for control of the rigid object. In real life, whenever this instantaneous correction is not possible, the system will try to solve this over a longer time span, allowing for more time for the attitude to be corrected. The total angular momentum introduced by the high initial angular velocity, would then be counter-acted by the thrusters over a longer period, which is perfectly possible. This simulation has not included this approach to solving the control problem, and can be seen as a recommendation for improving the algorithm.

For the initial angular velocity values that were possible, on the other hand, the thruster activations for correction over the simulation time were in line with the expected thrusters for firing. Since the maximum attainable rates were tested, the maximum thrust output (as concluded from the maximum power required in Table 6.4) was observed from these thrusters. In addition, the settling times and total energy required for control of the system, for initial velocities over one axis only, scaled inversely with the mass moment of inertia about each axis. For three-axis initial tumble, the energy needed per configuration showed the same results again as was seen for the regular simulation: configuration 1 required the most total energy, after which configuration 3, 2 and 4 placed themselves. These results serve as independent verification technique for the results of the regular simulation. Finally, the de-tumbling manoeuvrers experienced by the control system only, without taking into account any actuators, showed that a maximum spin rate of 85 [$^\circ/s$] was feasible to control, and that any spin rate beyond that or the other rates as shown in Table 6.6 would cause uncontrolled spin within the current attitude control system. Based on the control torque results in this table, the ADCS of the spacecraft can be re-sized based on the expected angular rates during the actual mission.

Finally, the reaction wheel configuration was also tested against high initial angular velocities. Due to the superior capabilities of the reaction wheels in terms of maximum torque achievable, higher angular rates were possible to induce and they could be corrected over a longer time span, as was seen in Table 6.7, looking at the settling times. The power requirements for de-tumbling were significant and would, similar to the thrusters in single thruster failure approach 2, most likely lead to system and mission failure. Comparing the reaction wheel results to the electrical-thruster-only set-ups, however, leads to the conclusion that the thrusters are not suitable for de-tumbling manoeuvres, whereas the reaction wheel system is. For this reason, the currently proposed thruster set-ups would not be suitable for implementation in the LUMIO mission, since the ADCS should be able to handle these extreme situations.

7.2.4. Mission Requirements

For both the thruster and reaction wheel cases, it was seen that the half-cone offset angle fell well within the limits of the requirement of 0.18° set by the mission's **ADCS-01** requirement. For requirement **ADCS-02**, the attitude control simulation performed in this research was not sufficiently accurate to assess the requirement's fulfilment. For **ADCS-03**, it was observed from Figure 6.11 that the maximum slew rate throughout the regular simulation was approximately 0.00045 [rad/s], which is equal to 0.026 [$^\circ/s$]. This falls well within the limit of 0.5 [$^\circ/s$] set out by the requirement. For this reason, the performance of the system adheres to the requirements.

The total power that was initially budgeted for the ADCS of the LUMIO mission is 2.91 [W] for all scenarios except de-tumbling, for which 7.97 [W] was budgeted. From the reaction wheel analysis in this report, it became evident that the necessary power for the reaction wheel functioning throughout the entire nominal simulation is equal to $2.5 \cdot 10^{-4}$ [W], which is also well within the limits of the power available. This does not yet take into account the power necessary for the other ADCS components. Unfortunately, it was not possible to examine the power consumption of each of the components for the simulation time individually, but it is given, for example, that the power consumption of the star trackers is <1.5 [W], and since activation of reaction wheels requires generally more power than these trackers, it can be concluded that the nominal power requirements that have come from this research will fall well within the power budget given by the LUMIO mission. For the de-tumbling manoeuvres, however, the required power of the ADCS could become significantly more than the budgeted value. Further research should examine what the maximum expected de-tumbling rates may become and adjust the power budget accordingly.

7.2.5. Summary

To conclude the discussion on the results of this research, the following summary highlights the main findings and conclusions drawn from the analyses:

- **Mission pointing requirement**: The quaternion components of the spacecraft closely adhered to the reference quaternion throughout the simulation, with relative quaternion errors remaining below 0.001% during close fly-bys. The half-cone angle offset consistently met the LUMIO mission requirement of 0.18°.
- **Reaction wheel performance**: Reaction wheels successfully controlled the spacecraft without saturating over the two-week simulation. The maximum angular momentum build-up of $6 \cdot 10^{-4}$ $[Nms]$ was well below the wheel capacity of 0.1 $[Nms]$. However, for extreme scenarios, such as de-tumbling or long-term operation, additional momentum dumping actuators will likely be required. The reaction wheel system operated efficiently within the power budget, consuming only $2.5 \cdot 10^{-4}$ $[W]$ during nominal operations.
- **Thruster configuration comparison**: In nominal scenarios, electrical thruster configurations successfully adhered to the control requirements, with configuration 4 showing the best performance, reducing total energy consumption by 29% and maximum power by 36% compared to configuration 1. In general, it could be concluded that adding redundant thrusters enhances the energy consumption of a configuration compared to a determinate system, and that adding thruster pairs controlling multiple spacecraft primary axes simultaneously is advantageous compared to thruster pairs only responsible for one axis.
- **Mass budget**: Thruster-based configurations offer potential mass savings compared to the current LUMIO ADCS, with configurations 1 and 2 reducing the spacecraft's wet mass by 4.06%. However, these savings are valid only for nominal conditions and do not account for extreme scenarios, which would require additional hardware or modifications.
- **Single thruster failure**: Approach 1 (with system awareness of failures) allowed the system to maintain functionality, although redundancy (e.g., configuration 4) is essential to handle failures without compromising control. Approach 2 (without system awareness) induced critically high power and energy demands, rendering the system unfeasible. This highlights the necessity of real-time thruster health monitoring and communication with the onboard computer.
- **Disturbance torque reflection**: Disturbance torques from gravity gradient and solar radiation pressure were insignificant compared to the control torque. However, the disturbance torque from main engine firings, not included in this research, could significantly affect the ADCS performance and should be included in future analyses.
- **Solar array deployment**: Retracting solar arrays significantly reduced ADCS power and energy requirements, with average reductions of 59% in power and 62% in energy across all configurations. This scenario is highly beneficial for early mission phases or contingency cases.
- **De-tumbling**: The thruster-based ADCS struggled to control high initial angular velocities due to instantaneous torque limitations. This performance deficit severely restricts their practical application as a replacement for the reaction wheel system and would not be advised by this research. Enhancing the control algorithm to distribute corrections over longer time spans is recommended for future studies.
- **Final conclusion**: Replacing the current LUMIO ADCS with electrical thruster configurations could be feasible for nominal scenarios but is unsuitable for extreme situations such as de-tumbling or handling main engine firing disturbances. Therefore, while theoretical mass savings and energy performance gains are achievable, no clear benefits arise from replacing the reaction wheel system in practice.

7.3. Experimental Characterisation

The practical experiment performed in this research can, based on its results, be used as a validation case for the simulation. It was first of all shown, as presented in Table 6.8, that the ported code showed the exact same results for the control torque and reference attitude vector compared to the Python-based simulation, with slight deviation due to non-significant round-off errors due to the double-precision of the numbers. The thrust values showed slight deviation, which can be attributed to the usage of the linear programming solution with the ECOS solver. As mentioned previously, the extensive calculations performed in this solver are the reason the magnitude of the error rises to the order of 10^{-11} and cannot be attributed to any computation error. These results show that the Python-based code is reproducible and translatable in other languages, which makes future research in line with this work a possibility.

Although reproducibility is an important aspect of any research, the main validation value came from the proper functioning of the valve. With this positive result observed, it was proven that any OBC running on the attitude control algorithm as developed throughout this research, would be able to successfully send its signals to electrical thrusters. It is understood that a research gap is still present due to the usage of a solenoid valve instead of an actual electrical thruster, but connection to hardware has been proven.

Upon analysis of the single thruster 1 from configuration 1, it could already be concluded that the imposed duty cycle on the valve coincided with the required thrust from the thruster, including the MIB requirement cut-off. Upon closer inspection, it was seen that the offset between the commanded and actual signal was in the range 4-5%, which is equal to the assumed thrust uncertainty modelled throughout the simulation. For this reason, it can be concluded that the real-life output of the system accurately adheres to what is desired from internal calculations. With respect to the duty cycles of the other thrusters, the same can be concluded.

One issue that can be observed is the lack of the thrust from thruster 6 in the final output. This will influence the control torque of the spacecraft, and can be compared to the single thruster failure approach 2 as was seen in the robustness results. Unfortunately, the simulation would not yield a stable state for the failure of one thruster in configuration 1, so its exact effects over this same time span could not be assessed. The adaptation of the control algorithm for allowing configuration 1 and 2 to have thruster failure although being a determinate system, will be left for future research and feasibility analysis.

7.4. Recommendations for Future Work

Based on the elaborate discussion laid out in this chapter, a number of focused recommendations for future research and potential continuation of this work can be presented. These recommendations are based on ideas during the execution of the research, gaps that were identified during the discussion of the research or general ideas that were thought of afterwards. Again, a clear distinction can be made between the simulation part of the research and the practical part of the research.

First of all, in order to analyse the total angular momentum build-up in the reaction wheels and to assess the total impulse needed for the entire mission duration, the two-week simulation as was shown can be extended to the full-year simulation. In order to do this with the same accuracy as was done in this report, one of these simulations would take approximately 26 hours to complete. It can then also be assessed whether the CAPSTONE orbit shows any irregularities on the long term. Moreover, with respect to the reaction wheels, the physical limitations of its resolution should be known, so that a final conclusion can be given on their usability in this research.

One of the primary disturbance sources not accounted for in this research is the torque induced by main engine firings. To align more closely with the LUMIO mission requirements, this disturbance torque should be modelled and incorporated into the simulation. This would require knowledge of the firing instances, including the exact duration, magnitude, and orientation of the main engine thrust at each moment. By introducing these parameters and accounting for uncertainties in both direction and magnitude, the disturbance torque can be accurately computed for each firing sequence. Incorporating these effects into the analysis will provide a stronger foundation for validating the feasibility of the electrical thruster configurations proposed in this research within the LUMIO mission framework.

Based on the Monte Carlo simulations performed in the research from Rizza et al. [41], it was found that the maximum disturbance torque generated by the main engine firing would be in the order of magnitude of 0.01 [Nm]. In the electrical thruster configurations from this research, the maximum torque induced would be equal to $4 \cdot 200$ $[\mu N]$ $\cdot 0.1$ $[m]$ = $8 \cdot 10^{-5}$ $[Nm]$, only around one axis (configuration 4). From these high-level estimates, it is therefore already expected that the instantaneous maximum disturbance from the main engine firing would also not be feasible to counteract using these thrusters. For future research, different actuator options with higher thrust capabilities should be investigated, such as electrospray thrusters or regular ion thrusters, to ensure the ADCS system can handle such disturbances effectively.

To enhance the realism of the simulation, the introduction of noise to parameters sensed externally by the spacecraft's components could be considered. Incorporating noise in the determination of the spacecraft's position, attitude, and measured disturbance torques would introduce offsets between expected and actual values. Computational tools, such as the extended Kalman filter, could then be employed to mitigate these offsets and smooth out the noise. This approach is particularly relevant because no real-world system operates perfectly; incorporating such imperfections into the simulation would provide an additional layer of validation for its robustness and reliability.

As outlined in the assumptions section of this research, the orbital environment analysed here is based on the CAPSTONE orbit, rather than the actual quasi-periodic halo orbit planned for the LUMIO mission. The CAPSTONE orbit introduces more disturbances due to its close lunar fly-bys, which require the reference quaternion to adapt more rapidly, and due to a more pronounced gravity gradient torque. In contrast, the actual LUMIO orbit is expected to provide a more stable environment with fewer rapid attitude adjustments and reduced disturbance torques. The effects of using the true LUMIO orbit should be investigated in future work to quantify the differences in control torque requirements and assess how much lower the strain on the ADCS would be. This investigation would not only provide a more realistic operational scenario but also serve as an additional test case to further validate the successful implementation of the thruster system under mission-specific conditions.

Additionally, the control torque magnitudes observed in this research primarily arise from the rapid changes in the reference quaternion during the close lunar fly-bys. This issue could potentially be mitigated by refining the PD control algorithm. Specifically, introducing a desired angular velocity term into the control law could help smooth out the rapid attitude adjustments and reduce torque spikes. Although this approach was explored during this research, the results did not yield a feasible or realistic solution within the current system framework. For this reason, it is recommended to further investigate enhancements to the PD controller, including tuning its gains more systematically or incorporating additional control terms, such as feed-forward components to anticipate the rapid changes in the reference quaternion.

Alternatively, exploring other advanced control strategies could also prove beneficial. For example, model predictive control could be considered, as it enables the controller to account for future states and disturbances over a prediction horizon, potentially improving performance during dynamic phases like close fly-bys. Similarly, adaptive control techniques may provide a solution by dynamically adjusting controller parameters based on the system's behaviour, allowing the ADCS to better respond to varying disturbance environments and reference changes. Finally, optimal control methods, such as Linear Quadratic Regulators, could be applied to minimize control effort while maintaining pointing accuracy, further reducing strain on the actuators.

Within the nominal simulation, the thruster hardware specifications currently used show performance parameters that are over-qualified for the research results. For this reason, a different vacuum arc thruster module should be included, with lower maximum torque values and also lower MIB limits. A good candidate for this research application is the VAT developed by the University of Würzburg, with a maximum thrust level of 2 $[\mu N]$ (Section 2.6).

Currently, the results of the algorithm have not been extensively validated against comparable studies or real-world data. As a future recommendation, similar studies should be made available or conducted to facilitate effective validation. This could involve benchmarking the algorithm against previously published work or collaborating with other teams working on comparable systems. The LUMIO ADCS research that was published in the past, could be used as a benchmark when the analysis of the main engine parasitic torque is included, and when the actual LUMIO orbit would be used.

Regarding the practical aspects of this research, establishing a direct connection between the STM32 Nucleo board and the actual thruster module, or engineering models thereof, is recommended to enable a more in-depth study of connectivity and system integration. Utilizing a setup that closely mimics the final onboard configuration of the spacecraft would allow for the validation of critical aspects, including signal communication between the control system and the thrusters, the acceptance and execution of control commands, and the physical output performance of the thruster modules.

In the current research, the STM32 Nucleo board was programmed and connected to a valve that served as a ”dummy thruster,” providing a basic proof of concept for the communication and control loop. However, a more comprehensive approach would involve replacing the dummy thruster with a set of engineering models of the actual thrusters, matching the configurations used in the simulations. Conducting hardware-in-the-loop tests with these engineering models would enable real-time thrust measurements, facilitating an accurate assessment of the control system's performance in driving the thrusters.

This setup would also allow for the study of the dynamic behaviour of the thrusters, including response times and any potential discrepancies between the commanded and actual thrust values. Additionally, testing under various operational conditions, such as different firing sequences or prolonged operation, would provide insights into potential wear, thermal effects, or performance degradation over time. For this to be possible, additional data should be transmitted again from thruster module to the OBC, requiring an extension of the data handling system. These enhancements would bridge the gap between simulation and real-world application, strengthening the validity of the control algorithm and the feasibility of the proposed thruster configurations.

All in all, since the complete replacement of the current LUMIO ADCS by electrical thruster configurations does not seem to be feasible for the entire mission, alternatives can be tested in future research. As mentioned previously, other electrical thruster modules can be incorporated, and additional configurations can be tested. A more promising analysis that can be carried out is keeping the reaction wheels from the current ADCS design, and replacing the mono-propellant thrusters that serve as the reaction control system with electrical thrusters. In this way, the novel hardware is only necessary for wheel desaturation, and the excessive tank volume that is currently included in the design is omitted.

8 Conclusion

This report has shown an extensive examination of the application of electrical thrusters to an existing 12U CubeSat mission, ESA's LUMIO mission. This research found its origin in the study of lunar CubeSat missions: a significant number of these small spacecraft have already been sent to the lunar environment for a diverse range of scientific objectives, and there are many more to come. Within CubeSat development, miniaturisation and specifically the development of micro-propulsion units has been a popular topic over the past few years, which has led to the rise of small electrical and non-electrical thruster units. Electrical units pose the benefit of not having to include a propellant tank and the simplicity of connection with the spacecraft on-board computer and power system. CubeSats also contain an ADCS, with which it can attain its correct attitude for its scientific, communication or power generation purposes. Miniaturised ADCS components, such as reaction wheels and magnetorquers, have been widely applied in existing CubeSat missions, and therefore extensively tested.

The above observations clearly showed that a research gap was present in current literature: electrical thrusters were not being applied for the ADCS of CubeSats, especially not for the lunar environment. With this in mind, the main research question could be constructed:

What is the impact of adjusting the ADCS configurations, consisting of electrical thrusters only, on the LUMIO mission, a 12U lunar CubeSat, on its attitude control performance, robustness and connectivity?

In order to properly approach this research question, it was sub-divided in three main components: assessment of the spacecraft performance with an attitude control simulation, the assessment of the control algorithm using multiple robustness tests, and finally validating the algorithm by connecting it to existing hardware. The simulation context consisted of ESA's LUMIO mission, a 12U CubeSat to be launched in 2027 to attain a quasi-periodic halo orbit, around the Earth-Moon L_2 point, and detect meteoroid impact flashes in the far-side of the Moon. Since its orbital parameters were not publicly available, a similar orbit as attained during the CAPSTONE mission was taken for this research. With the context described, a control algorithm was developed that could be run for four different thruster configurations over a simulation period of two weeks. As output of this algorithm, several parameters such as the thrust per thruster, angular velocity of the spacecraft body and half-cone offset angle over time were computed. With these results, feasibility of the thruster configurations could be assessed based on mission requirements and physical limitations.

Within the spacecraft robustness tests, single thruster failure was adopted and split into two approaches: the first let the system be aware of a malfunctioning thruster, the second did not make the system aware of this. For the second approach, the correct control torque would therefore never be generated. Deployment of the solar arrays was taken as the base case in the general simulations, but as additional test its undeployed state was evaluated as well. Finally, initial angular velocities were fed to the system to assess its response to these extreme situations. It was found that, due to the limitation of the maximum thrust output per thruster, significant initial rotations could not be implemented. The system itself, without assessment of any actuator, could be tested for its maximum allowed initial rotation.

For the nominal two-week simulation scenario, as well as for the initial angular velocity robustness test, a reaction wheel set-up was assessed in addition to the electrical-thruster-only configurations. This analysis consisted of four reaction wheels only, without any reaction control thrusters for attitude control, copied from the current design of the LUMIO mission. In this design, three reaction wheels are aligned with one spacecraft primary axis, and a fourth reaction wheel is equally aligned with all three of these axes. The results of this reaction wheel configuration served as a base case to compare the electrical thruster configurations with, and finally to assess whether replacing the entire ADCS by these thrusters would be beneficial based on performance, volume and mass.

Regarding the physical validation of the system, the Python-based algorithm was partly ported to an embedded system, in which reference quaternion, control torque and thruster allocation calculations were performed. After the results were verified, the embedded system, an STM32 Nucleo development board, was connected to a solenoid valve and pulse-width modulated command signals were sent from the Nucleo board to this hardware module, based on the control algorithm. In this way, a physical set-up containing a PC, micro-controller and an actuator was created that could be tested for execution time, accuracy and for the correctness of the commands.

From results of the main simulation of this research, a number of conclusions could be drawn. It was observed that the spacecraft attitude correctly and accurately adhered to the reference attitude as desired throughout a two-week simulation period. In addition, the control torques required were successfully produced by the reaction wheel configuration, as well as by all the thruster configurations. In the reaction wheel analysis, it was seen that no momentum desaturation would be necessary over the simulated time span, since angular momentum build-up was insignificant due to the barely perturbed environment in which the spacecraft was operational. It was realised, however, that for longer simulation runs and for extreme situations, reaction wheel desaturation would definitely be necessary, which is why it was concluded that additional thrusters would in any case be necessary for this configuration. To add to the reaction wheel results, it was observed that the resolution of the currently implemented wheels was too small for accurate control of the spacecraft. The wheels were included regardless of this, in order to adhere to the ADCS design laid out in the LUMIO mission research.

For the thruster configurations, the maximum thrust output value was in the range of $6.0 \cdot 10^{-7}$ $[N]$, which is well below the maximum Pocket Rocket thrust output of $2.0 \cdot 10^{-4}$ $[N]$. Also, to be able to fire considering the Minimum Impulse Bit of the Pocket Rocket, firings of longer than 1 second with low thrust values should be applied. Considering the 1 $[Hz]$ control simulation frequency, this would cause major inaccuracies in the attitude results. Therefore, it was concluded that the resolution of the thrusters was not accurate enough for application in this simulation. The major deficit in this simulation compared to the LUMIO mission analyses from literature was the disturbance caused by the main engine firing. Were this included, the angular momentum build-up and thrust values would adhere better to previous research values. As a preliminary estimation, it is expected that the thruster configurations would in any case not be able to correct for the parasitic torque introduced by the main engine firing, since its maximum disturbance value has an order of magnitude of 0.01 $[Nm]$. Apart from this lacking disturbance, the thruster configurations are suitable for application in the nominal mission scenario as was tested in this simulation. Concerns were, however, raised by the significant increase of power and energy compared to the reaction wheel configuration, as well as the results from robustness tests, which portray extreme situations.

All aspects of the simulation, from unit tests to larger integration tests, have been verified in order to ascertain its correct computations. Moreover, as means of validation, the algorithm was tested against a ”simple” input and its results were compared to what would be expected in the real world. Moreover, the experimental set-up from this research serves as a main validation technique. In order to further complement the validation aspects of this research, a comparable study should be included to ascertain the validity of results. It was, unfortunately, not possible to retrieve such research results.

From the single thruster failure approaches, it was concluded that the determinate configuration 1 and

2 did not allow for any thruster failure, as was expected. No solutions were simply possible within the limitations set out and using the linear programming solver. For the other two configurations, the remaining thrusters would cover for the thruster loss in approach 1 which was clearly seen in the increase in thrust output over all thrusters. An increase in power required and total energy consumption was also observed. Moreover, in approach 2, large thrust spikes were observed for correcting the incorrectly attained attitude, which led to large increase in required power and energy over the simulated time span. These increases will most likely lead to system and mission failure in real-life situations, which is why approach 2 should be avoided in actual missions. Sensors should always be present that can inform the OBC on any potential thruster failure. Moreover, determinate systems should be avoided as well, since complete loss of mission takes place when any of the thrusters in configuration 1 or 2 fails.

For the undeployed solar array case, it was observed that less effort from the ADCS was required for its attitude control, as was also expected based on the change in mass moment of inertia. It could be concluded that the undeployed case would most definitely also be feasible within the simulation limits to deploy electrical thrusters on. For the initial angular velocities, the thrusters responded as expected, in the sense that the thrusters causing rotation over a specific axis were activated for an initial angular velocity over that same axis. Energy requirement for the de-tumble manoeuvrers, with the thruster limitations in place and also solely examining the control algorithm, showed the same hierarchy as was observed during the regular simulation. The initial possible rates for thrusters were, however, unrealistically low. Upon examining the maximum initial rates and results for the reaction wheels, it was evident that their superior performance range compared to the thruster configurations allows for realistic angular velocities. For this reason, it was concluded that the thrusters will not be able to be applied in real-life extreme situations, adding to the disadvantages of using these compared to the current reaction wheel set-up used in the LUMIO mission. From these results, it is evident that the robustness tests have added valuable information on the limitations of this research, in addition to the nominal simulation conditions.

Finally, the reproducibility demonstrated by the code porting in the practical implementation highlights the potential for further development of this research, as well as its adaptability to multiple programming languages and embedded systems. The successful valve firing tests validated that the simulation framework could be seamlessly integrated into an existing OBC setup with real actuator modules. Furthermore, the results confirmed that the control signals generated by the ”dummy” OBC were accurately transmitted to the ”dummy” thrusters, ensuring correct interfacing and functionality. The only discrepancy in accuracy observed could be attributed to the readily expected thruster uncertainty. This validation step represents a critical milestone, bridging the gap between simulation and real-world implementation.

Having summarised and concluded the research performed in this report, the research questions should be assessed for their fulfilment. In addition, the hypotheses have to be checked for their correctness. For the first sub-question (**SQ-01**), it can be concluded that the replacement of the reaction-wheel-based ADCS by electrical thrusters only is not beneficial in terms of power requirements and energy consumption. The total energy required over a time span of two weeks was in the range of several Joules for the reaction wheel set-up, whereas the thruster results showed 1 to 2 kilo Joules. The pointing accuracy between different thruster configurations is equal, since the accuracy stems directly from the control torque computed in attitude control algorithm. The same goes for the reaction wheel set-up, which was also equal in pointing accuracy. As a main recommendation for elaboration on this research question, additional configurations should be assessed and the main engine parasitic torque should be included in the simulation. In this way, comparable results might be retrieved with respect to past LUMIO mission research.

Regarding sub-question 2 (**SQ-02**), three separate answers can be formulated. First of all, single thruster failure is only possible in the current algorithm for determinate thruster systems and will generally induce an increase of power and energy consumed by any configuration. Moreover, the pointing accuracy will be equal if the system is aware of the problem, and can increase and decrease if the system is unaware, dependent on the configuration. In real-life scenarios, the spacecraft data system should always be aware of failure using sensors or other data feeding. De-tumbling manoeuvrers are limited by the physical limits of the thrusters, but are generally solved by the thrusters that control the tumbling axis. Moreover, they require high power and energy levels for their solution, especially in the limit cases, and will give equal pointing accuracy as seen previously when stabilised. The initial angular rates were unrealistic for real-life mission application of the thrusters and is one of the reasons the thruster configurations are discarded as they have been analysed in this research. For the reaction wheel analysis, more relaxed limits and behaviour were observed, as also expected. Realistic initial rates with realistic power and energy requirements were reached for this. Finally, the deployment of solar arrays generally increases the mass moment of inertia, requiring more power and energy from the ADCS than was originally needed. The pointing accuracy remains equal.

The answer to **SQ-03** has been treated elaborately in this research, and can be summarised by stating that effective code porting, with unit tests and debugging, needs to take place, after which signal modulation needs to take place to make sure correct output signals are produced. These outputs signals are fed to the actuator hardware modules, and can in this way represent a real-life attitude control system.

Although the correctness of the hypotheses is clear from this report, the major differences can be highlighted. First of all, **H-03** posed that significantly more energy would be required from this system than in the original ADCS. From the research performed here, this was not evidently true; no actual power budget analysis could be performed, but the results shown in this report are below the results from reaction wheel analysis in the original LUMIO mission analysis. Compared to the reaction wheel analysis in this research, however, the thrusters required significantly more power and energy. For this reason, **H-03** is true. **H-04** stated that the pointing accuracy for more thrusters would increase, although it was seen that this does not necessarily have any influence; the configuration responds to the required control torque, and needs more or less energy to solve this dependent on the number of thrusters present. The hypothesis for the main research question, **H-11**, was also not completely valid; pointing accuracy is only improved during the robustness tests and remains equal for the rest of the simulations, and resilience against de-tumbling was not proven. The optimised number of thrusters was shown to be configuration 4 in this research, in terms of power and energy requirements.

A number of recommendations for future work include: the analysis of the main engine firing as a disturbance; the usage of the actual orbit LUMIO will attain; an extended simulation duration, to a full year timespan; including a different vacuum arc thruster module with lower performance parameters for the nominal simulation; including a different vacuum arc thruster module with higher performance parameters for de-tumbling; include a vacuum arc thruster engineering model in the practical set-up; perform a hardware-in-the-loop test that analyses thrust output values compared to their desired value. All these suggestions can be taken to future research in the domain of the LUMIO spacecraft and its ADCS.

In conclusion, this research demonstrated the partial feasibility of replacing the current reaction-wheel-based ADCS of ESA's LUMIO mission with different electrical thruster configurations. Including the gravity gradient torque and solar radiation pressure torque, which proved to be insignificant for direct attitude control, a two-week simulation showed that all Pocket Rocket vacuum arc thruster configurations could effectively keep the half-cone angular offset below the limit of 0.18° throughout the entire simulation, whilst keeping the thrust output below their maximum values as well. Better performance in terms of energy consumption was achieved by including more thruster pairs, and allowing them to control multiple axes simultaneously. Compared to the reaction wheels as used in the current LUMIO ADCS design, however, total energy consumption was on average a factor 10^3 larger for the thruster configurations and showed no direct benefits in those terms. Thruster redundancy should in any case be present to counteract potential thruster failure, and the control algorithm showed to be adaptable to solar array deployment. The limited thrust outputs from the VATs did not allow for realistic initial angular velocities to be imposed on the system, whilst the algorithm did show proper response to such behaviour. Apart from the disadvantages laid out here, mass could be saved on the overall LUMIO system by incorporating the electrical thruster configurations, but it is realised that this does not pose a realistic mission scenario. The successful validation of the control algorithm on embedded hardware further bridged the gap between simulation and real-world application, emphasizing the adaptability of the developed system for future CubeSat missions.

Although the approach adhered to in this research did not provide a realistically feasible alternative for the reaction wheel configuration currently employed within the LUMIO mission, future developments might yield more promising results. Incorporating parasitic torque effects due to main engine firing and replacing the current thrusters with higher-performance modules could significantly enhance the feasibility of such systems. Additionally, extending simulations to account for longer mission durations would offer deeper insights. Research into the use of electrical thrusters for CubeSat ADCS should continue to foster innovation, exploring alternative configurations, thruster technologies, and algorithms. By expanding the boundaries of current knowledge, future missions may unlock the full potential of miniaturised propulsion systems, contributing to the advancement of CubeSat capabilities in lunar and other deep-space environments.

References

[1] Solid State Propulsion (SSP). *Pocket Rocket: Advanced Propulsion System*. Version 2.5. Compact propulsion system with solid-state design, optimized for collision avoidance and minor orbital adjustments. First flight scheduled for Q4 2024. Solid State Propulsion. South Africa, May 2024. URL: `https://www.solidstatepropulsion.co.za`.

[2] AirTAC and Trimantec. *2V Series Fluid Control Valve Datasheet*. Includes specifications for 2V025, 2V130, and 2V250 solenoid valves. 2016. URL: `https://trimantec.com/product-category/airtac-2v-fluid-control-valve-2-2-way/`.

[3] A. Alkatheeri et al. "Design and Implementation of Attitude Control System for Gnssas 6U Cubesat". In: *IEEE International Geoscience and Remote Sensing Symposium* (2023), pp. 384–387. DOI: `10.1109/IGARSS52108.2023.10281925`.

[4] H. Alemi Ardakani and T.J. Bridges. "Review of the 3-2-1 Euler Angles: a yaw–pitch–roll sequence". In: *Department of Mathematics, University of Surrey* (Apr. 2010).

[5] Argotec. *FERMI OBC&DH: Miniaturized Rad-Hard On-Board Computer & Data Handling Unit*. FERMI enables deep-space operations for CubeSat-class platforms, offering high-performance processing, edge computing, and radiation-hardened components for spacecraft management and autonomous missions. Argotec. 2024. URL: `https://www.argotec.it`.

[6] A. Bello et al. "Experimental verification and comparison of fuzzy and PID controllers for attitude control of nanosatellites". In: *Advances in Space Research 71* (2023), pp. 3613–3630. DOI: `https://doi.org/10.1016/j.asr.2022.05.055`.

[7] Inc. Busek Co. *BGT-X5 Green Monopropellant Thruster*. Version 1.0. Busek Co., Inc. 11 Tech Circle, Natick, MA 01760, USA, 2021. URL: `https://www.busek.com`.

[8] Inc. Busek Co. *BHT-200 Hall Effect Thruster: Outsized Performance in a Compact Package*. Version 1.0. Busek Co., Inc. 11 Tech Circle, Natick, MA 01760, USA, 2021. URL: `https://www.busek.com`.

[9] J.C. Butcher. *Numerical Methods for Ordinary Differential Equations*. New York: John Wiley Sons, 2008. ISBN: 978-0-470-72335-7.

[10] A. Cervone. *AE4S07 - Course Reader - Micro-Propulsion*. 4th ed. Delft, The Netherlands: Delft University of Technology, 2022.

[11] A. Cervone et al. "Selection of the Propulsion System for the LUMIO Mission: an Intricate Trade-Off Between Cost, Reliability and Performance". In: 2021.

[12] T. Cilliers, W.H. Steyn, and H.W. Jordaan. "Multi-axis thruster-only fine pointing control strategies for nanosatellites". In: *Elsevier* (2024), pp. 1–10.

[13] A. Cipriano, D.A. Dei Tos, and F. Topputo. "Orbit Design for LUMIO: The Lunar Meteoroid Impacts Observer". In: *Frontiers in Astronomy and Space Sciences* 5 (Sept. 2018). DOI: `10.3389/fspas.2018.00029`.

[14] C.J. Dennehy et al. *Application of micro-thruster technology for space observatory pointing stability*. Tech. rep. NASA Engineering Safety Center, 2020.

[15] A.O. Esho et al. "Electrical propulsion systems for satellites: a review of current technologies and future prospects". In: *International Journal of Frontiers in Engineering and Technology Research* 6 (2024), pp. 35–44. DOI: `10.53294/ijfetr.2024.6.2.0034`. URL: `https://doi.org/10.53294/ijfetr.2024.6.2.0034`.

[16] G.F. Franklin, D.J. Powell, and A. Emami-Naeini. *Feedback Control of Dynamic Systems*. 4th. USA: Prentice Hall PTR, 2001. ISBN: 0130323934.

[17] E. Fresk and G. Nikolakopoulos. "Full quaternion based attitude control for a quadrotor". In: *2013 European Control Conference (ECC)*. 2013, pp. 3864–3869. DOI: `10.23919/ECC.2013.6669617`.

[18] T. Gardner et al. "CAPSTONE: A CubeSat Pathfinder for the Lunar Gateway Ecosystem". In: *Small Satellite Conference* (2021), pp. 1–7. DOI: https://digitalcommons.usu.edu/smallsat/2021/all2021/142/.

[19] R. Gottlieb. "Fast gravity, gravity partials, normalized gravity, gravity gradient torque and magnetic field: Derivation, code and data". In: *NASA Contractor Report 188243* (1993), pp. 1–64.

[20] M. Grande et al. *Planetary Exploration Horizon 2061 Report Chapter 5: Enabling technologies for planetary exploration*. 2023. URL: https://arxiv.org/abs/2302.14832.

[21] C. Hardgrove et al. "The Lunar Polar Hydrogen Mapper (LunaH-Map) Mission". In: *33rd Annual AIAA/USU* (2019), pp. 1–7. DOI: https://digitalcommons.usu.edu/smallsat/2019/all2019/97/.

[22] Small Spacecraft Systems Virtual Institute. *State-of-the-Art Small Spacecraft Technology*. NASA Technical Publication NASA/TP—2024–10001462. Moffett Field, CA: NASA Ames Research Center, Feb. 2024. URL: http://www.nasa.gov/smallsat-institute/sst-soa.

[23] Z. Ismail and R. Varatharajoo. "A study of reaction wheel configurations for a 3-axis satellite attitude control". In: *Advances in Space Research* 45.6 (2010), pp. 750–759. DOI: https://doi.org/10.1016/j.asr.2009.11.004.

[24] J. Jang, M. Plummer, and M. Jackson. "Absolute Stability Analysis of a Phase Plane Controlled Spacecraft". In: *AIAA Space Flight Mechanics Meeting* (2010), pp. 1–13. DOI: 10.13140/2.1.2472.0002.

[25] E. van Kampen. *Lecture 2: Eigenaxis, quaternions, MRP*. AE4313-20: Spacecraft Attitude Dynamics and Control. 2024. URL: https://studiegids.tudelft.nl/a101_displayCourse.do?course_id=65819&_NotifyTextSearch_.

[26] M. Kannan, U. Anitha, and A. Kumarasamy. "CubeSat attitude control by implementation of PID controller using Python". In: *12th International Conference on Advanced Computing* (2023), pp. 1–5. DOI: 10.1109/ICoAC59537.2023.10249887.

[27] J.T. King et al. "Performance analysis of nano-sat scale μCAT electric propulsion for 3U CubeSat attitude control". In: *Acta Astronautica* 178 (2021), pp. 722–732. DOI: https://doi.org/10.1016/j.actaastro.2020.10.006. URL: https://www.sciencedirect.com/science/article/pii/S0094576520306020.

[28] D. Krejci and P. Lozano. "Space Propulsion Technology for Small Spacecraft". In: *Proceedings of the IEEE* 106.3 (2018), pp. 362–378.

[29] I. Kronhaus et al. "Simple Orbit and Attitude Control Using Vacuum Arc Thrusters for Picosatellites". In: *Journal of Spacecraft and Rockets* (2014), pp. 2008–2015. DOI: https://doi.org/10.2514/1.A32796.

[30] M. Kühn and J. Schein. "Development of a High-Reliability Vacuum Arc Thruster System". In: *Journal of Propulsion and Power* (2022), pp. 752–758. DOI: https://doi.org/10.2514/1.B38202.

[31] J. Lasue et al. "From planetary exploration goals to technology requirements". In: *Planetary Exploration Horizon 2061*. Elsevier, 2023, pp. 177–248. DOI: 10.1016/b978-0-323-90226-7.00005-2. URL: http://dx.doi.org/10.1016/B978-0-323-90226-7.00005-2.

[32] C. Leibbrandt and J. Miller. *CubeWheel Gen 2 Product Description*. Commercial in Confidence. CubeSpace Satellite Systems RF (Pty) Ltd. The LaunchLab, Hammanshand Rd, 7600, RSA, 2023. URL: http://www.cubespace.co.za/.

[33] C. Lemmer. "Propulsion for CubeSats". In: *Acta Astronautica* 134 (2017), pp. 231–243.

[34] Y. Lian, J. Xiang, and Y. Zhao. "Modulated multi-sliding-surface attitude tracking control for all-electric propulsion satellites". In: *Advances in Space Research* 72.8 (2023), pp. 3297–3307. ISSN: 0273-1177. DOI: https://doi.org/10.1016/j.asr.2023.06.012. URL: https://www.sciencedirect.com/science/article/pii/S0273117723004428.

[35] T.W. Lim. "Thruster Attitude Control System Design and Performance for Tactical Satellite 4 Maneuvers". In: *Journal of Guidance, Control and Dynamics* (2014), pp. 403–412. DOI: https://doi.org/10.2514/1.61727.

[36] B. Malphrus et al. “The lunar IceCube EM-1 mission: Prospecting the Moon for water ice”. In: *IEEE Aerospace and Electronic Systems Magazine* (2019), pp. 6–14. DOI: `https://doi.org/10.1109/MAES.2019.2909384`.

[37] P. Marwedel. *Embedded System Design: Embedded systems, Foundations of Cyber-Physical Systems and Internet of Things*. 3rd ed. New York, USA: Springer, 2018.

[38] D. McIntosh, J. Baker, and J. Matus. “The NASA Cubesat Missions Flying on Artemis-1”. In: *Small Satellite Conference* (2020), pp. 1–11. DOI: `https://digitalcommons.usu.edu/smallsat/2020/all2020/44/`.

[39] C.P. Newman et al. “Stationkeeping, orbit determination and attitude control for spacecraft in Near-Rectilinear Halo Orbits”. In: *AAS Astrodynamics Specialists Conference* (2018), pp. 1–20.

[40] C. Nieto and R. Emami. “CubeSat Mission: From Design to Operation”. In: *Applied Sciences* 9 (Aug. 2019), p. 3110. DOI: `10.3390/app9153110`.

[41] A. Rizza et al. “Design, Analysis and Validation of the ADCS for the LUMIO mission”. In: *Proceedings of the 74th International Astronautical Congress (IAC)*. International Astronautical Federation (IAF). Oct. 2023.

[42] H. Rom and A. Gany. “Thrust control of hydrazine rocket motors by means of pulse width modulation”. In: *Acta Astronautica* 26.5 (1992), pp. 313–316. ISSN: 0094-5765. DOI: `https://doi.org/10.1016/0094-5765(92)90077-V`. URL: `https://www.sciencedirect.com/science/article/pii/009457659290077V`.

[43] Á. Romero-Calvo, J. Biggs, and F. Topputo. “Attitude Control for the LUMIO CubeSat in Deep Space”. In: *International Astronautical Congress* (2019), pp. 1–13. DOI: `http://hdl.handle.net/11311/1117585.1880`.

[44] K. Saddul et al. “Mission analysis of a 1U CubeSat post-mission disposal using a thin-film vacuum arc thruster”. In: *Acta Astronautica 219* (2024), pp. 318–328. DOI: `https://doi.org/10.1016/j.actaastro.2024.03.019`.

[45] H. Sekine et al. “On-orbit Performance Evaluation of AQUARIUS: a Water Resistojet Propulsion System during Initial Flight Operation of a 6U CubeSat EQUULEUS”. In: *Transactions of the Japan Society for Aeronautical and Space Sciences* 67.5 (2024), pp. 274–284. DOI: `10.2322/tjsass.67.274`.

[46] E. Spreen, K. Howell, and D. Davis. “Near Rectilinear Halo Orbits and their application in cis-lunar space”. In: *3rd IAA Conference on Dynamics and Controls of Space Systems* (2017), pp. 1–20.

[47] STMicroelectronics. *STM32F303xD, STM32F303xE*. Rev 5. Datasheet for STM32F303xD/E microcontrollers. 2016. URL: `https://www.st.com`.

[48] A.H. Tavakkoli, M. Kabganian, and M. Shahravi. “Modeling of attitude control actuator for a flexible spacecraft using an extended simulation environment”. In: *2005 International Conference on Control and Automation*. Vol. 1. 2005, 147–152 Vol. 1. DOI: `10.1109/ICCA.2005.1528107`.

[49] Blue Canyon Technologies. *Reaction Wheels - Small Wheels 2024*. Lafayette, CO, USA: Blue Canyon Technologies, Jan. 2024.

[50] Blue Canyon Technologies. *XACT and FleXcore Attitude Control Systems: Product Description*. REV 1/2024. Specifications for XACT and FleXcore ACS systems, including design and performance parameters for various configurations. Blue Canyon Technologies. 2550 Crescent Drive, Lafayette, CO 80026, USA, 2024. URL: `https://www.bluecanyontech.com`.

[51] “The Forward Euler Method”. In: *Practical Analysis in One Variable*. New York, NY: Springer New York, 2002, pp. 583–604. DOI: `10.1007/0-387-22644-3_43`. URL: `https://doi.org/10.1007/0-387-22644-3_43`.

[52] F. Topputo et al. “LUMIO CubeSat: Current Status and Lessons Learnt (So Far)”. In: *4S Symposium 2024*. 2024.

[53] M. von Unwerth et al. “Application of CubeSat Technologies for Research and Exploration on the Lunar Surface”. In: *Advances in Astronautics Science and Technology* (2023), pp. 57–72. DOI: `https://doi.org/10.1007/s42423-023-00144-w`.

[54] J.R. Wertz. *Spacecraft Attitude Determination and Control*. 1st ed. Dordrecht, The Netherlands: Kluwer Academic Publishers, 1978.

[55] K. Williamson. *Research Methods for Students, Academics and Professionals*. 2nd ed. Kingston upon Hull, UK: Chandos Publishing, 2002.

[56] R. Wirz et al. “Hollow Cathode and Low-Thrust Extraction Grid Analysis for a Miniature Ion Thruster”. In: *International Journal of Plasma Science and Engineering* 2008 (2008), 11 pages. DOI: `10.1155/2008/693825`. URL: `https://doi.org/10.1155/2008/693825`.

[57] J. Ziegler and N. Nichols. “Optimum settings for automatic controllers”. In: *Journal of Dynamic Systems, Measurument and Control* (1993), pp. 220–222. DOI: `https://doi.org/10.1115/1.2899060`.

A Simulation Results

Below, the results for the thrust, power and energy per thruster, for the full two-week simulation between [00:00:00 01-01-2023] until [00:00:00 15-01-2023], are displayed. Each configuration is presented.

A.1. Configuration 1

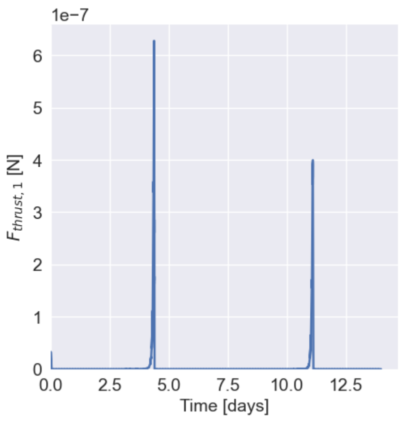


Figure A.1: Thrusters: F_1 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

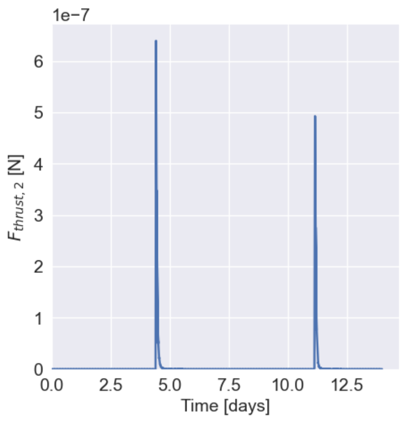


Figure A.2: Thrusters: F_2 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

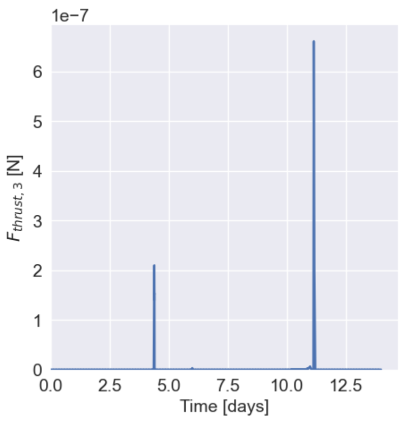


Figure A.3: Thrusters: F_3 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

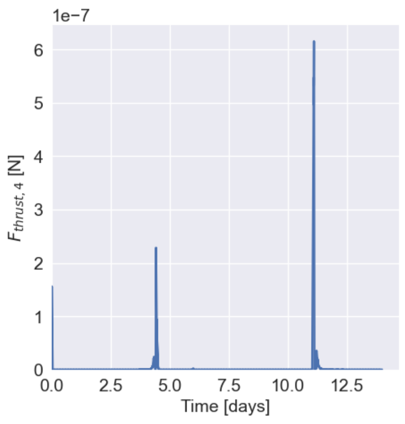


Figure A.4: Thrusters: F_4 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

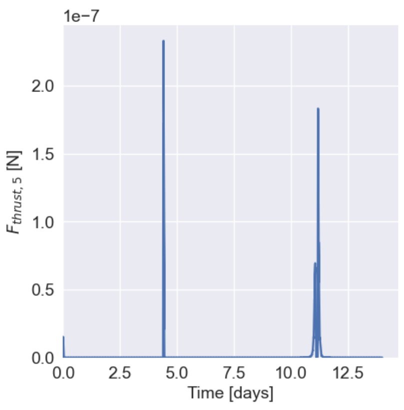


Figure A.5: Thrusters: F_5 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

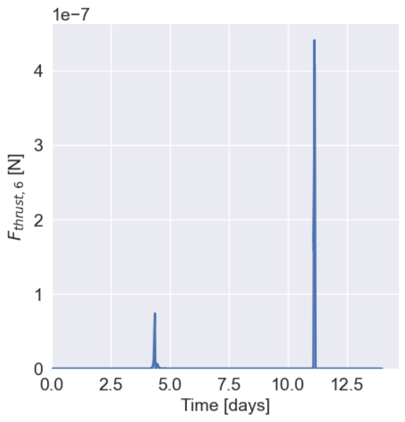


Figure A.6: Thrusters: F_6 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

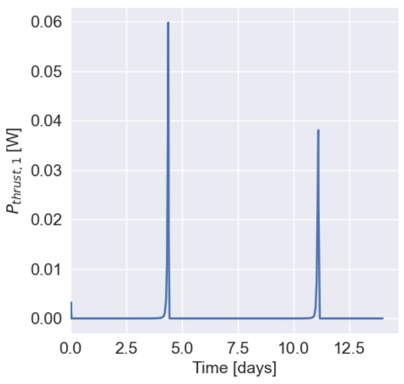


Figure A.7: Thrusters: P_1 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

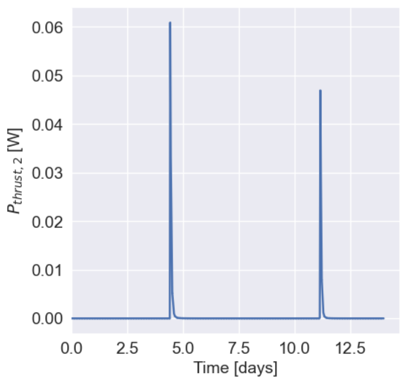


Figure A.8: Thrusters: P_2 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

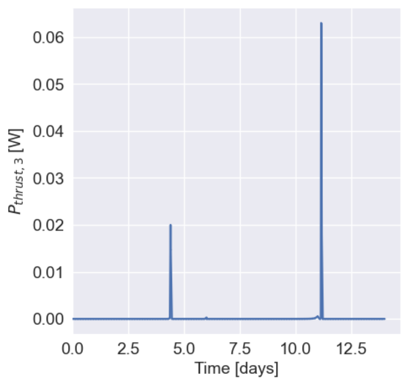


Figure A.9: Thrusters: P_3 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

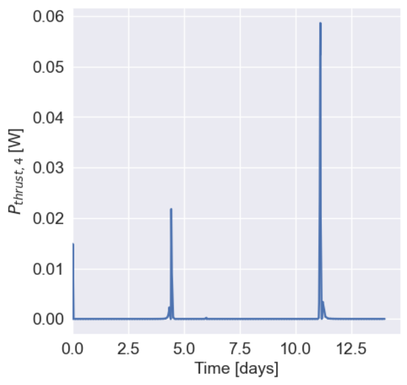


Figure A.10: Thrusters: P_4 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

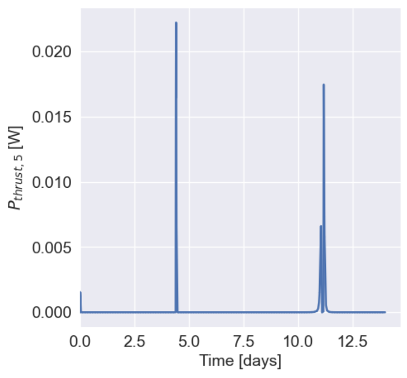


Figure A.11: Thrusters: P_5 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

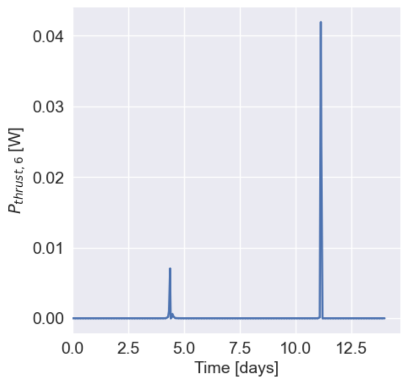


Figure A.12: Thrusters: P_6 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

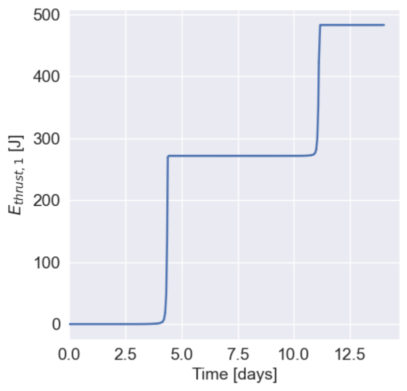


Figure A.13: Thrusters: E_1 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

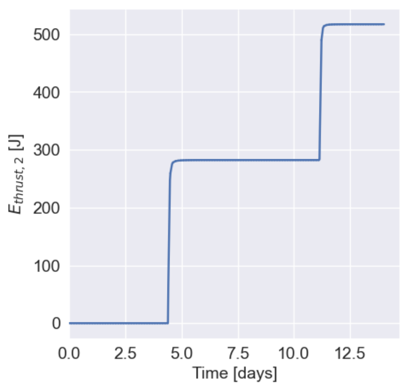


Figure A.14: Thrusters: E_2 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

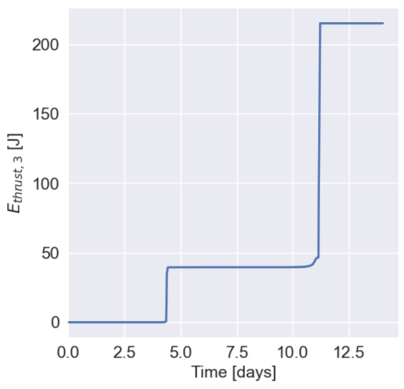


Figure A.15: Thrusters: E_3 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

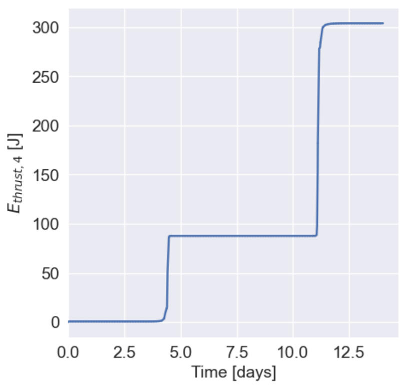


Figure A.16: Thrusters: E_4 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

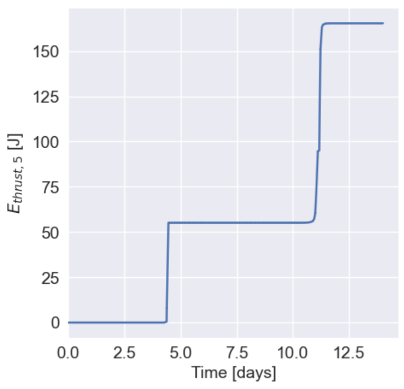


Figure A.17: Thrusters: E_5 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

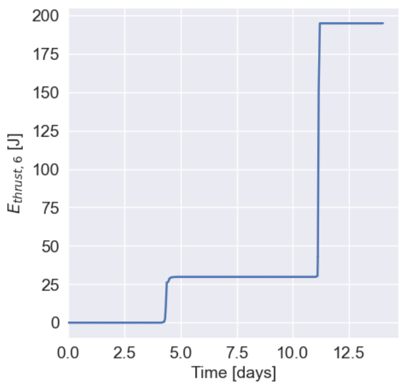


Figure A.18: Thrusters: E_6 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

A.2. Configuration 2

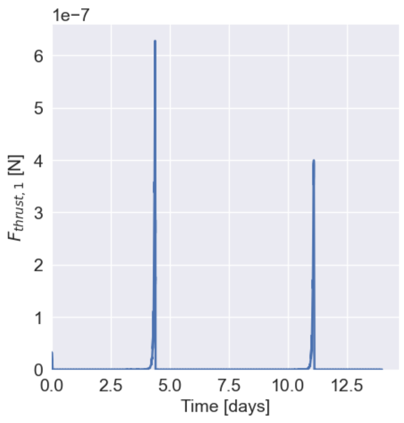


Figure A.19: Thrusters: F_1 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

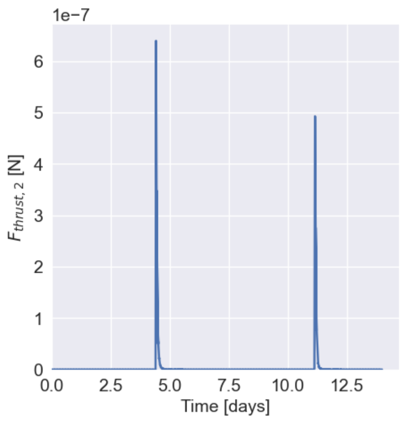


Figure A.20: Thrusters: F_2 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

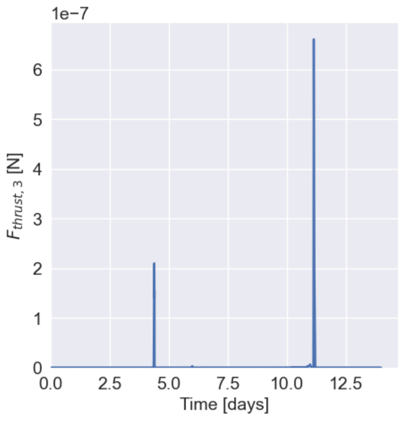


Figure A.21: Thrusters: F_3 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

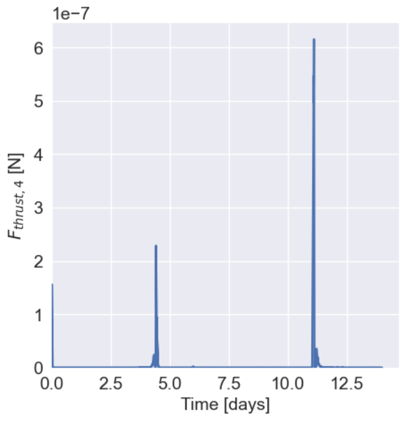


Figure A.22: Thrusters: F_4 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

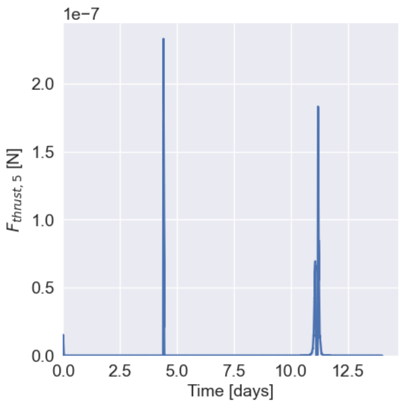


Figure A.23: Thrusters: F_5 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

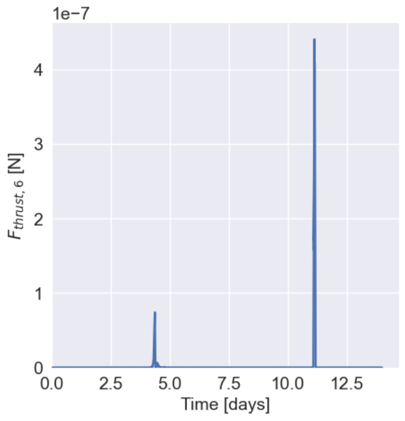


Figure A.24: Thrusters: F_6 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

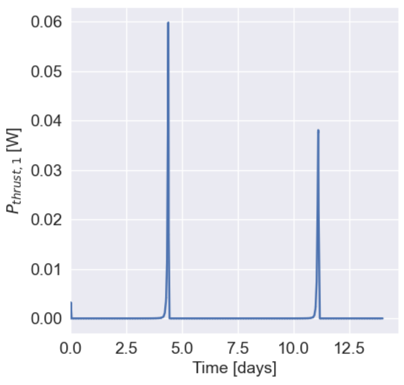


Figure A.25: Thrusters: P_1 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

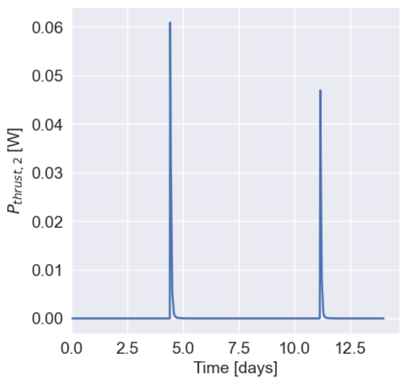


Figure A.26: Thrusters: P_2 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

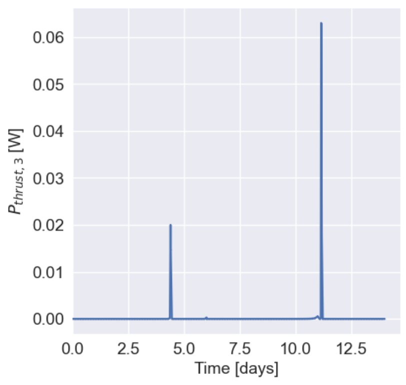


Figure A.27: Thrusters: P_3 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

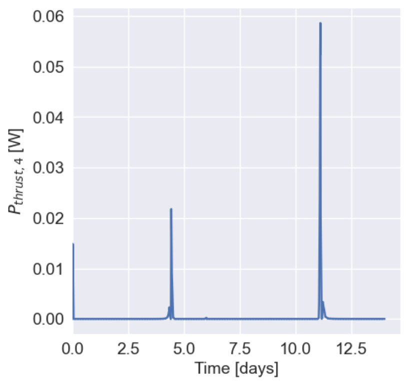


Figure A.28: Thrusters: P_4 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

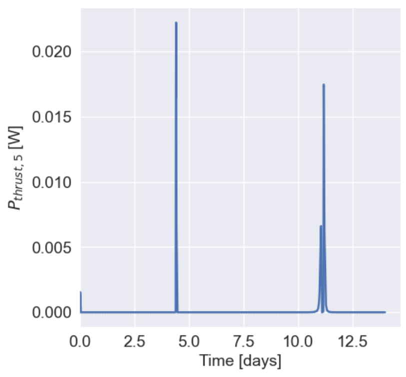


Figure A.29: Thrusters: P_5 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

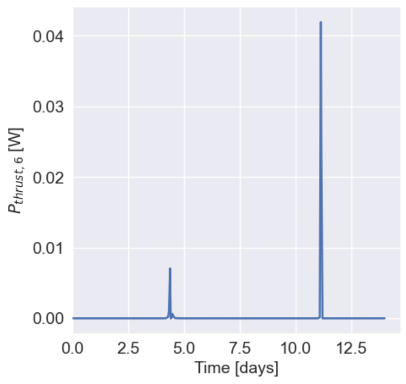


Figure A.30: Thrusters: P_6 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

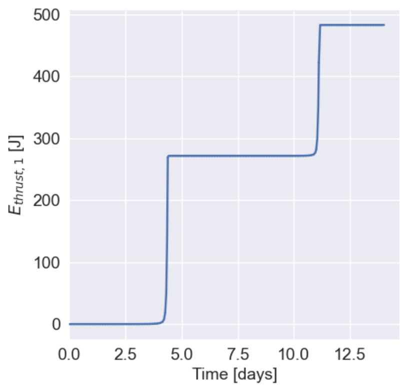


Figure A.31: Thrusters: E_1 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

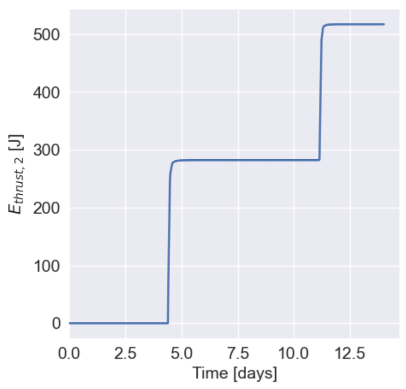


Figure A.32: Thrusters: E_2 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

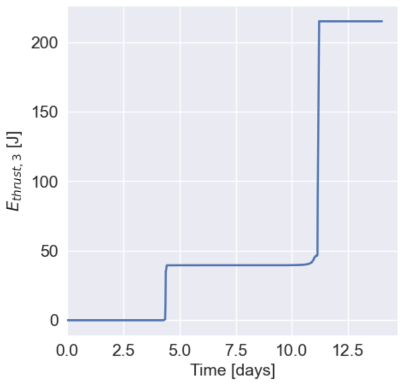


Figure A.33: Thrusters: E_3 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

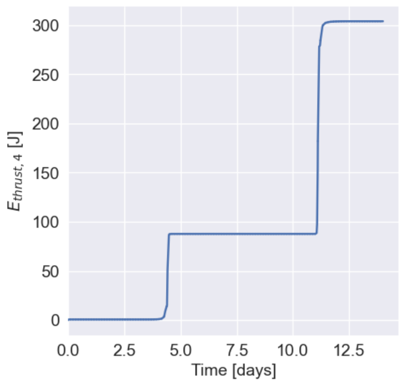


Figure A.34: Thrusters: E_4 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

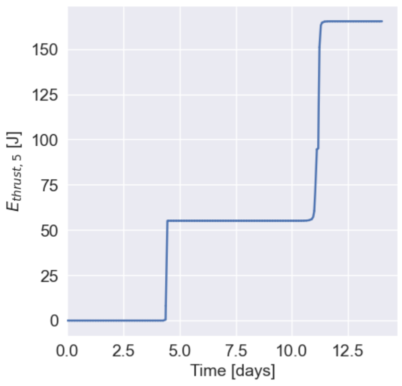


Figure A.35: Thrusters: E_5 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

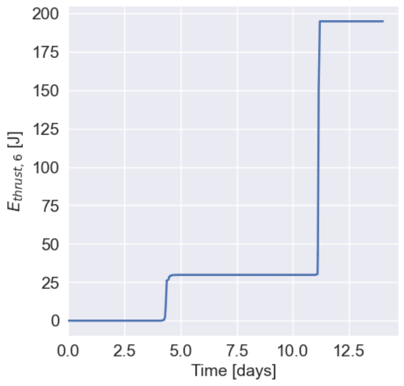


Figure A.36: Thrusters: E_6 output in configuration 1 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

A.3. Configuration 3

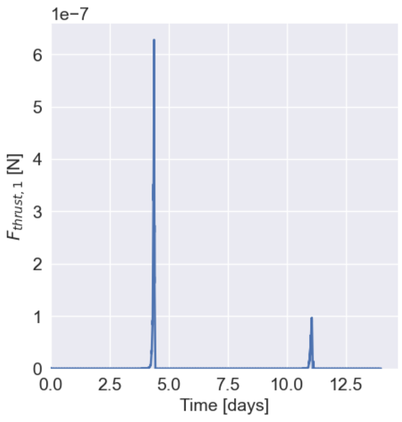


Figure A.37: Thrusters: F_1 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

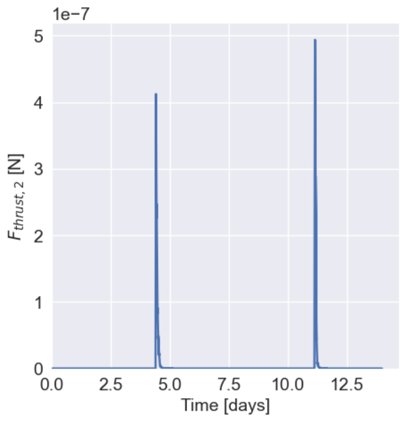


Figure A.38: Thrusters: F_2 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

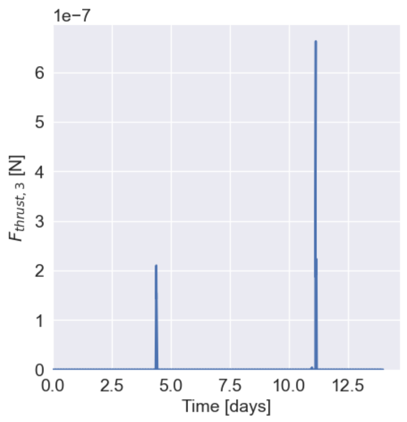


Figure A.39: Thrusters: F_3 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

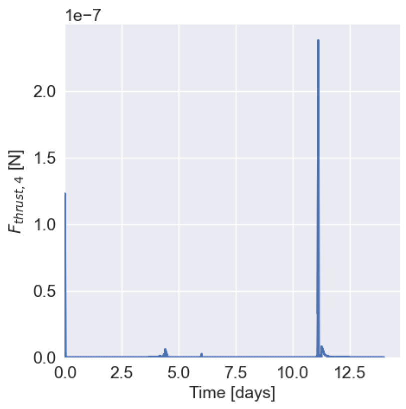


Figure A.40: Thrusters: F_4 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

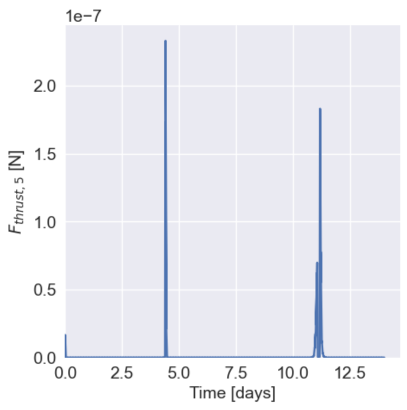


Figure A.41: Thrusters: F_5 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

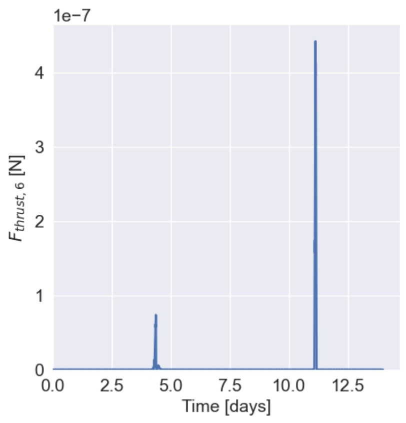


Figure A.42: Thrusters: F_6 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

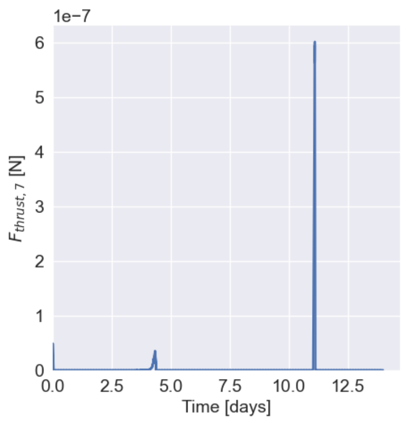


Figure A.43: Thrusters: F_7 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

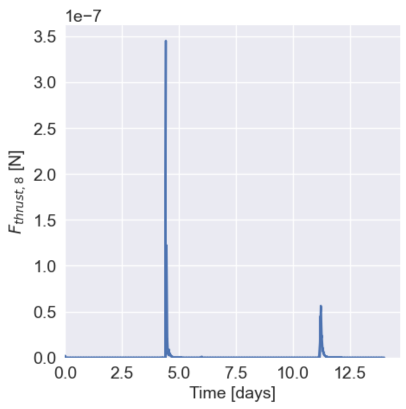


Figure A.44: Thrusters: F_8 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

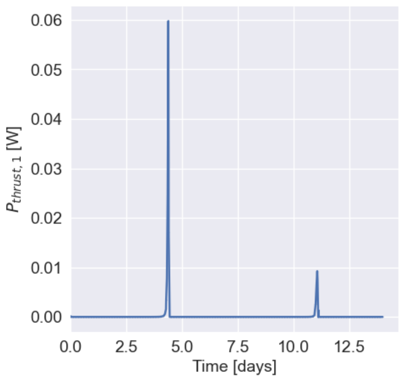


Figure A.45: Thrusters: P_1 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

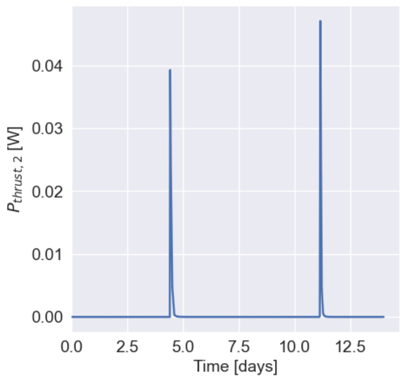


Figure A.46: Thrusters: P_2 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

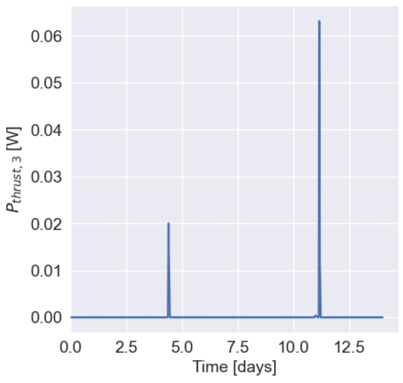


Figure A.47: Thrusters: P_3 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

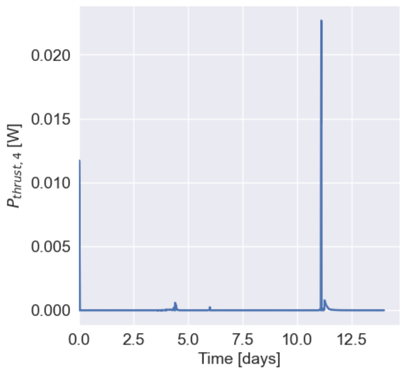


Figure A.48: Thrusters: P_4 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

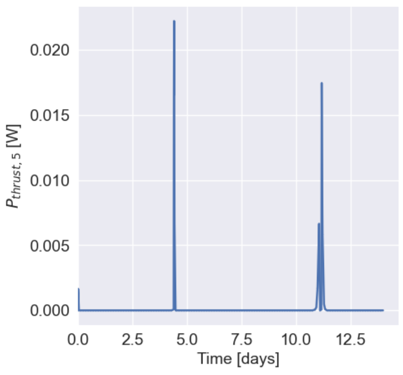


Figure A.49: Thrusters: P_5 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

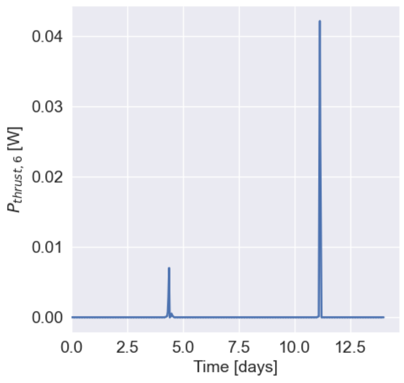


Figure A.50: Thrusters: P_6 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

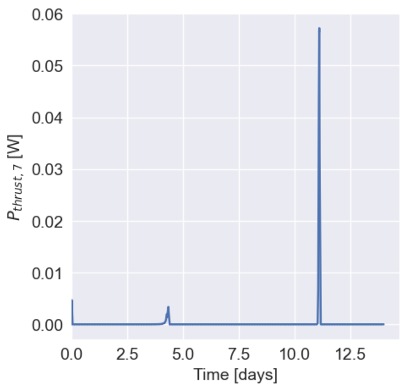


Figure A.51: Thrusters: P_7 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

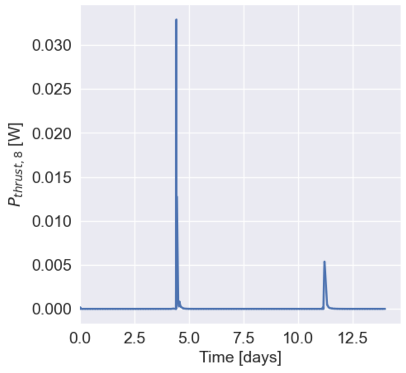


Figure A.52: Thrusters: P_8 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

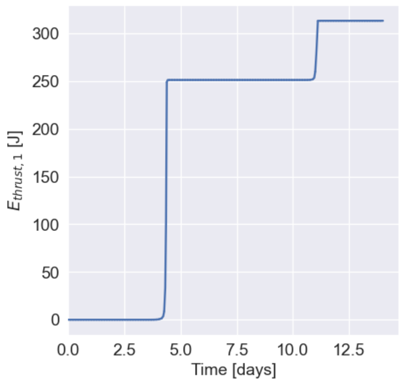


Figure A.53: Thrusters: E_1 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

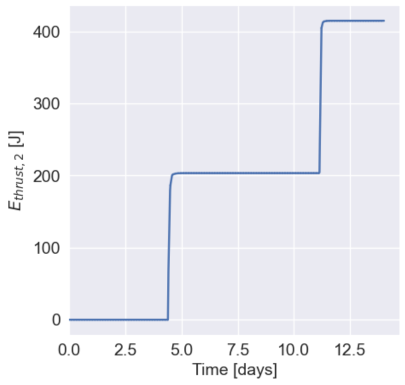


Figure A.54: Thrusters: E_2 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

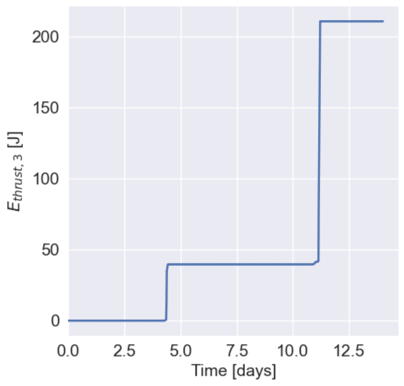


Figure A.55: Thrusters: E_3 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

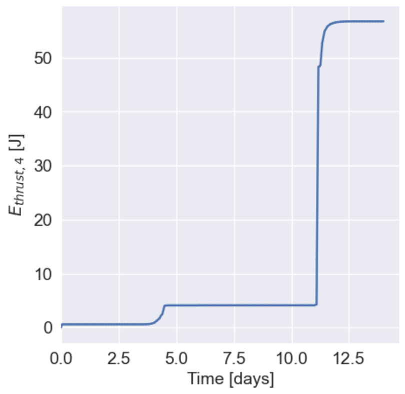


Figure A.56: Thrusters: E_4 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

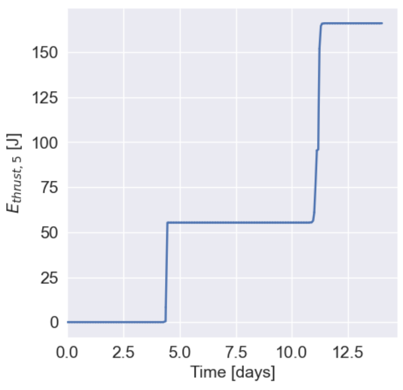


Figure A.57: Thrusters: E_5 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

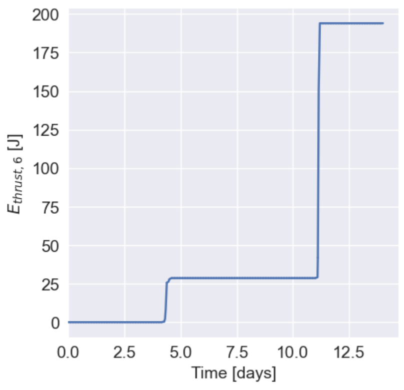


Figure A.58: Thrusters: E_6 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

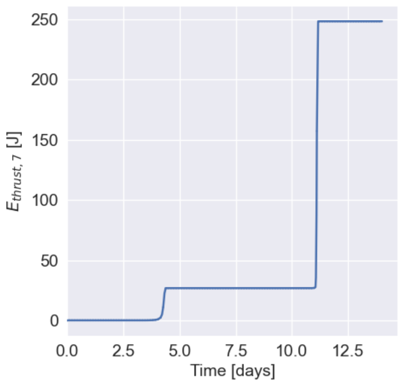


Figure A.59: Thrusters: E_7 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

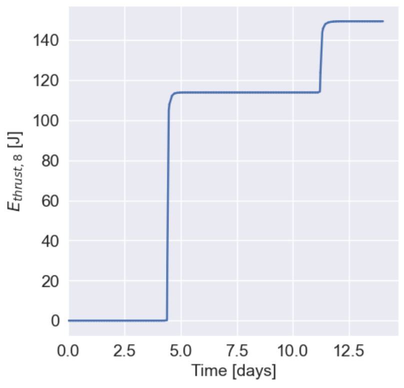


Figure A.60: Thrusters: E_8 output in configuration 3 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

A.4. Configuration 4

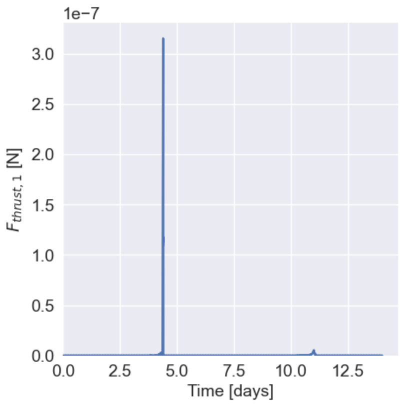


Figure A.61: Thrusters: F_1 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

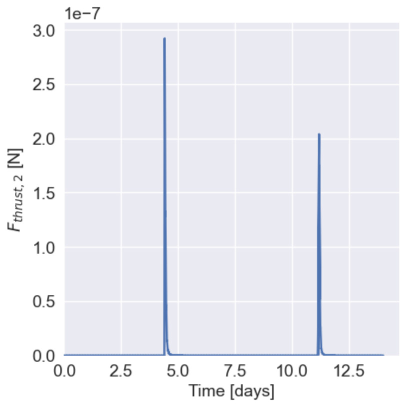


Figure A.62: Thrusters: F_2 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

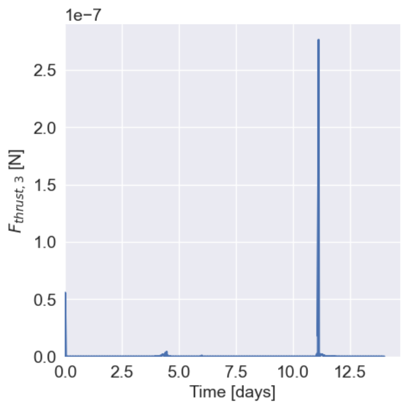


Figure A.63: Thrusters: F_3 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

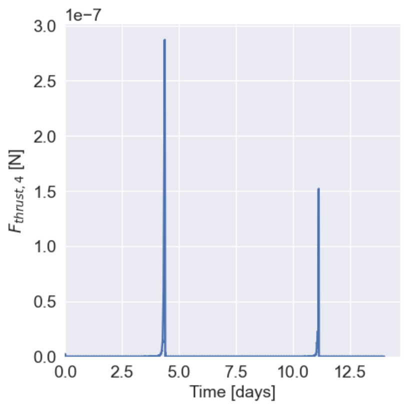


Figure A.64: Thrusters: F_4 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

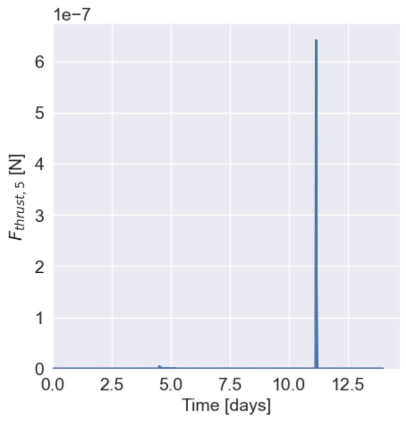


Figure A.65: Thrusters: F_5 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

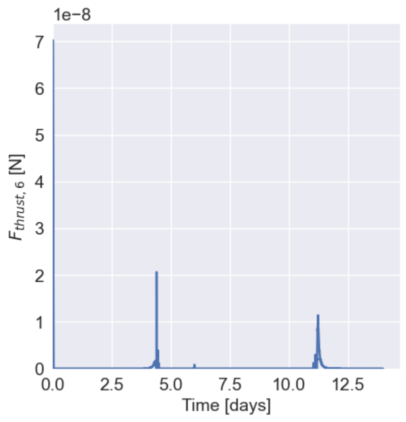


Figure A.66: Thrusters: F_6 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

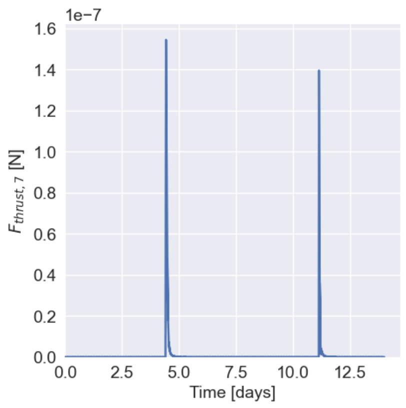


Figure A.67: Thrusters: F_7 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

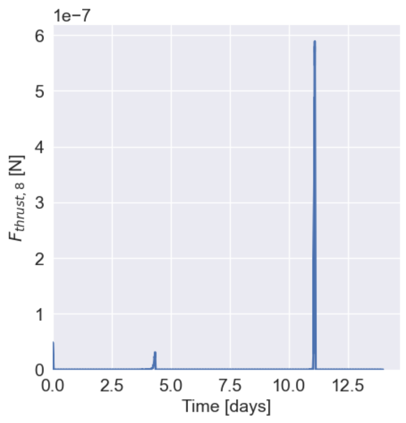


Figure A.68: Thrusters: F_8 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

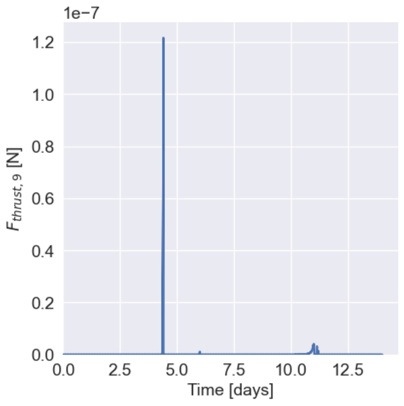


Figure A.69: Thrusters: F_9 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

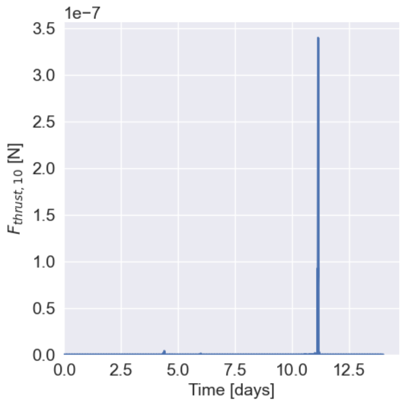


Figure A.70: Thrusters: F_{10} output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

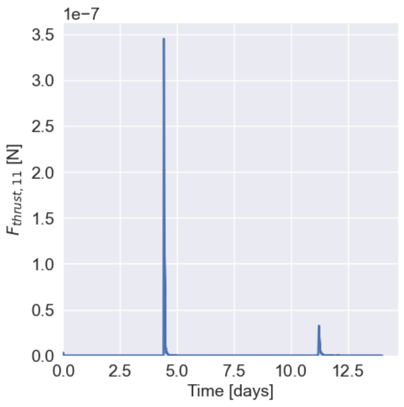


Figure A.71: Thrusters: F_{11} output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

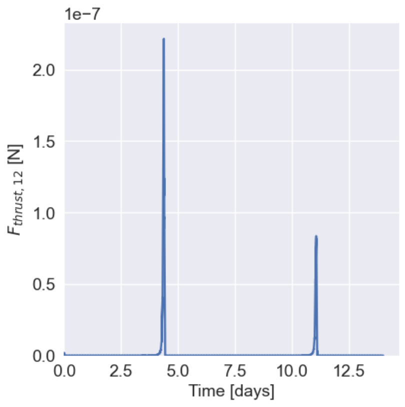


Figure A.72: Thrusters: F_{12} output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

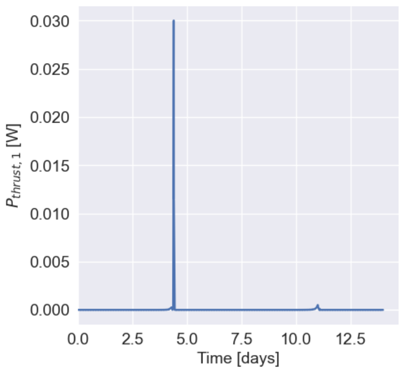


Figure A.73: Thrusters: P_1 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

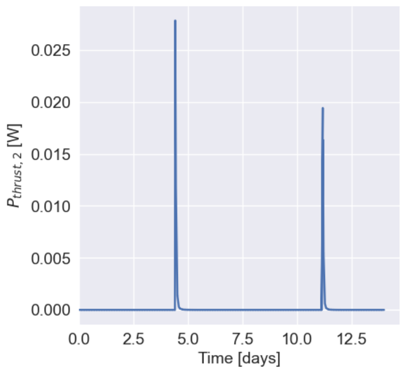


Figure A.74: Thrusters: P_2 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

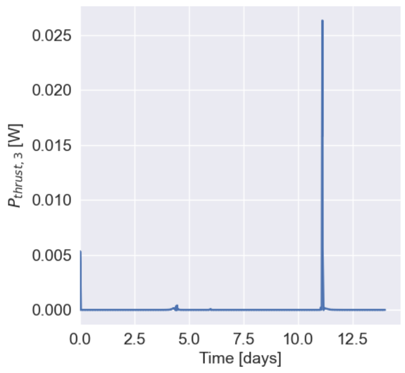


Figure A.75: Thrusters: P_3 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

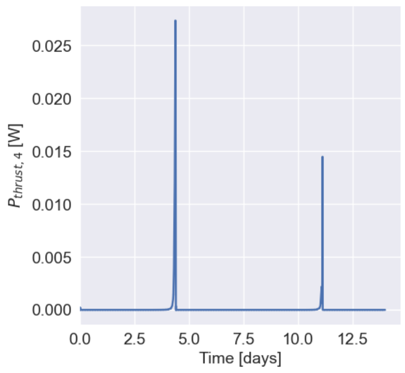


Figure A.76: Thrusters: P_4 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

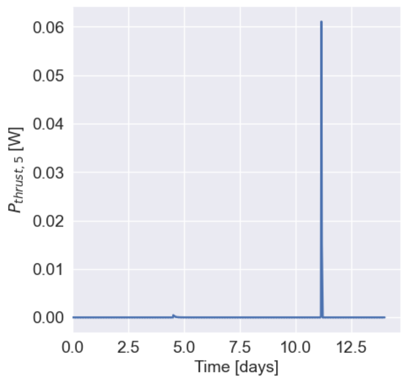


Figure A.77: Thrusters: P_5 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

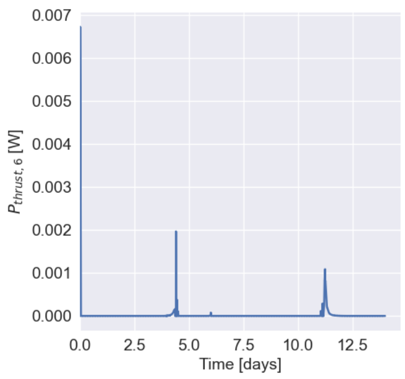


Figure A.78: Thrusters: P_6 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

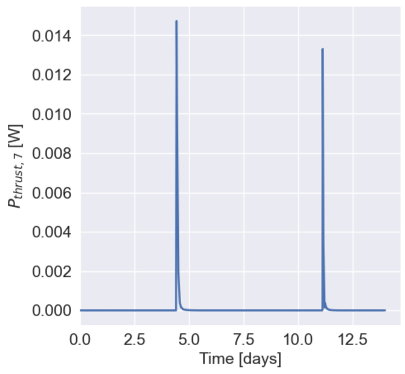


Figure A.79: Thrusters: P_7 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

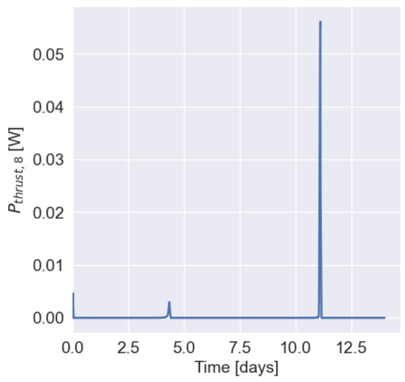


Figure A.80: Thrusters: P_8 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

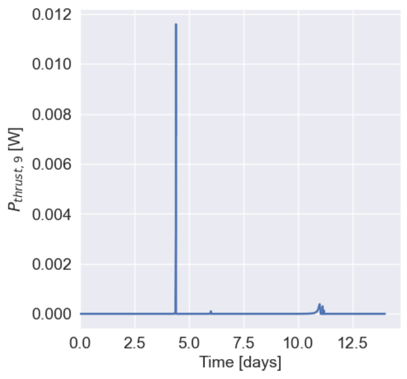


Figure A.81: Thrusters: P_9 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

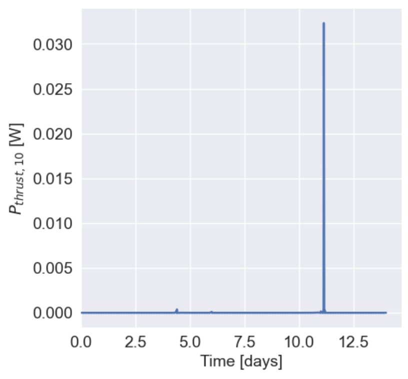


Figure A.82: Thrusters: P_{10} output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

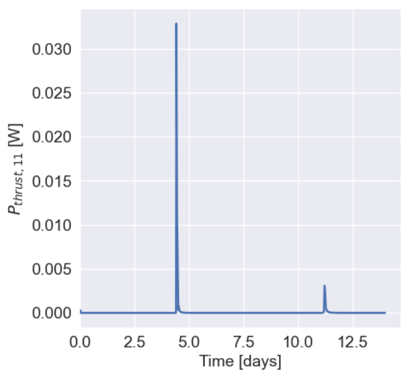


Figure A.83: Thrusters: P_{11} output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

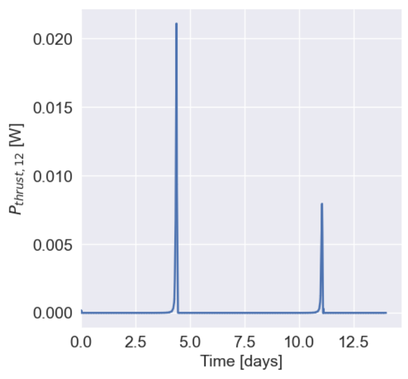


Figure A.84: Thrusters: P_{12} output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

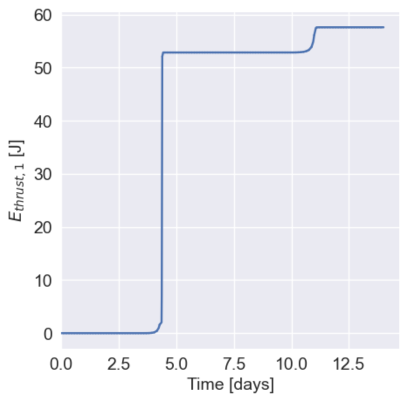


Figure A.85: Thrusters: E_1 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

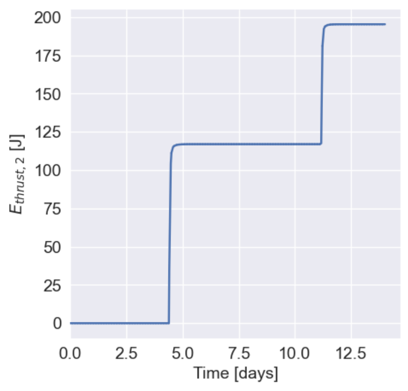


Figure A.86: Thrusters: E_2 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

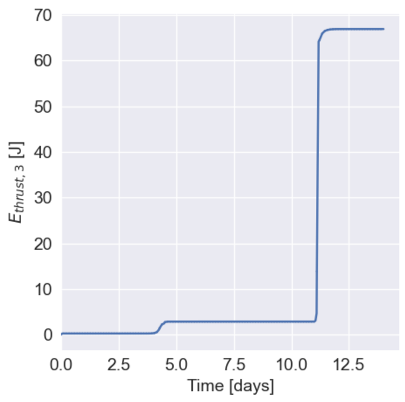


Figure A.87: Thrusters: E_3 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

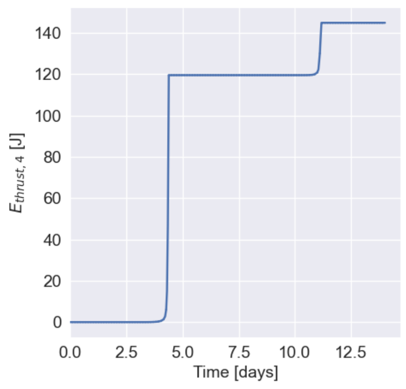


Figure A.88: Thrusters: E_4 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

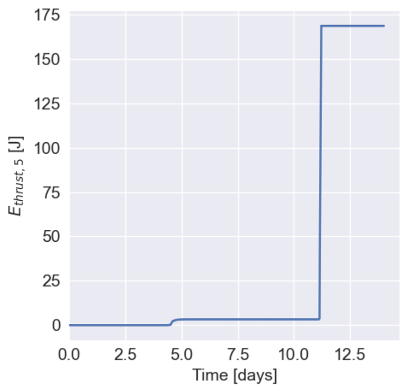


Figure A.89: Thrusters: E_5 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

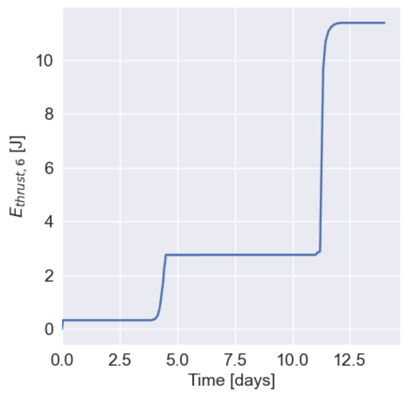


Figure A.90: Thrusters: E_6 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

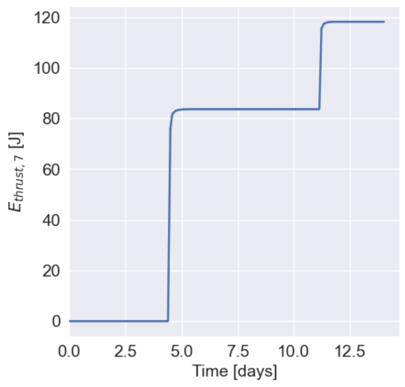


Figure A.91: Thrusters: E_7 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

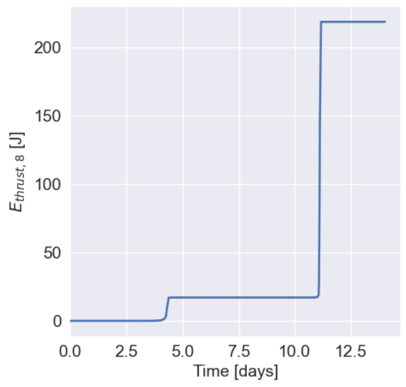


Figure A.92: Thrusters: E_8 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

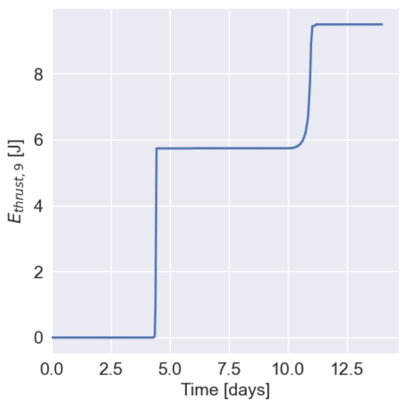


Figure A.93: Thrusters: E_9 output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

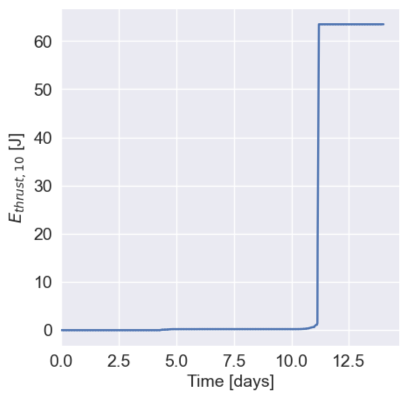


Figure A.94: Thrusters: E_{10} output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

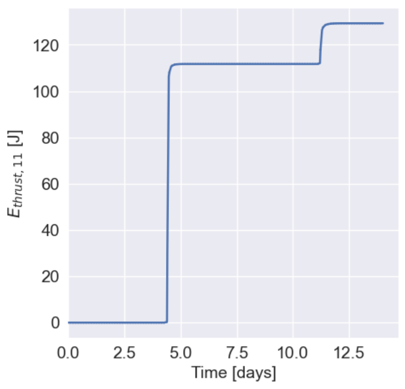


Figure A.95: Thrusters: E_{11} output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

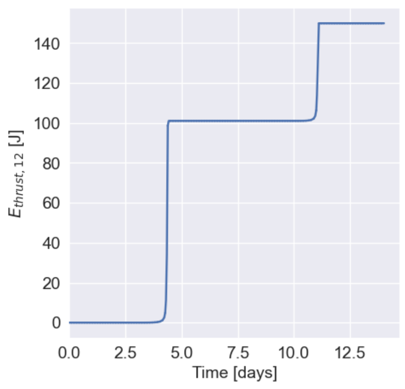


Figure A.96: Thrusters: E_{12} output in configuration 4 over time in days, analysed from [00:00:00 01-01-2023] until [00:00:00 15-01-2023].

B
Python Source Code

Below, a raw copy of the code for this thesis research is presented. Note that this is all included in the `classes.py` file. The full code, inlcuding Jupyter notebooks for execution and testing, can be viewed in:

https://github.com/Pieter1999/lunar_CubeSat

```python
from astroquery.jplhorizons import Horizons
from astropy.time import Time
from datetime import date
from datetime import datetime
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from scipy.linalg import null_space
import pandas as pd
import os
import re
from scipy.optimize import minimize

class Constants:
    """
    This class contains all the constants that are used throughout the research. Constants
        can in this way
    easily be called and their values are stored in one central location. Further in-line
        comments will
    clarify the meaning and values of the constants present.
    """

    # Physical constants
    c = 299792458  # Speed of light [ms^-1]
    G = 6.67428e-11  # Universal Gravitational Constant [m^3s^-2kg^-1]
    mu_Earth = 398600441800000.0  # Gravitational parameter Earth [m^3s^-2]
    mu_Moon = 4.9048695e12  # Gravitational parameter Moon [m^3s^-2]
    mu_Sun = 1.32712440018e20  # Gravitational parameter Sun [m^3s^-2]
    AU = 149597870700.0  # Astronomical unit [m]
    R_Earth = 6.3781e6  # Earth mean radius [m]
    R_Moon = 1737.4e3  # Moon mean radius [m]
    R_Sun = 695700e3  # Sun mean radius [m]
    P_solar = 3.842e26  # Power exerted by the Sun, [W]

    # LUMIO spacecraft data
    inertia_matrix = np.array([[100.9, 0, 0], [0, 25.1, 0], [0, 0, 91.6]]) * 10 ** (
        -2
    )  # deployed inertia matrix, [kg m^2]
    inertia_matrix_undeployed = np.array(
        [[30.5, 0, 0], [0, 20.9, 0], [0, 0, 27.1]]
    ) * 10 ** (-2)

    # Payload panel location
    LUMIO_loc_pp = np.array([0, 0.15, 0])  # m

    # Hyperion RW400 data
    T_RW_max = 0.007 #Nm
    P_peak_max = 9 #W

    # Solid State Propulsion Pocket Rocket data
    F_SSP_max = 0.0002 #N
    P_SSP_max = 20 #W
    MIB = 1.18e-6 #Ns

    # Astroquery data
    # CAPSTONE data
    id_CAPSTONE = "-1176"  # JPL Horizons
    start_date_CAPSTONE = (
        "2022-11-14"  # Official nominal mission start date, insertion in NRHO
    )
    end_date_CAPSTONE = "2023-05-18"  # Official end date nominal mission

    # Location data
    id_Moon = "301"
    id_Sun = "10"
    id_Earth = "500"
    location_Moon_centre = "500@301"  # 500 indicates the body-centric location
    location_Sun_centre = "500@10"
    location_Earth_centre = "500"
    location_CAPSTONE_centre = "500@-1176"


class Rotation:
    """
    Placeholdere name. This class will hold several rotational dynamics functions for easy
        use in reference frame conversion practices.
    At the moment of writing, only the 3-2-1 Euler rotation method and Euler angle -
        quaternion conversions are considered for this class.
    Take three Euler angles as input, potentially angular velocity as well. NOTE: roll =
        theta_1, pitch = theta_2, yaw = theta_3
    """

    def euler_rotation(
        self, theta_1, theta_2, theta_3
    ):  # 3-2-1 rotation, using Euler angles, from INERTIAL TO BODY FRAME, is R_x @ R_y @ R_z
        . Other way around is the inverse of this.
        # Define rotation matrices
        R_z = np.array(
            [
                [np.cos(theta_3), np.sin(theta_3), 0],
                [-np.sin(theta_3), np.cos(theta_3), 0],
                [0, 0, 1],
            ]
        )
        R_y = np.array(
            [
                [np.cos(theta_2), 0, -np.sin(theta_2)],
                [0, 1, 0],
                [np.sin(theta_2), 0, np.cos(theta_2)],
            ]
        )
        R_x = np.array(
            [
                [1, 0, 0],
                [0, np.cos(theta_1), np.sin(theta_1)],
                [0, -np.sin(theta_1), np.cos(theta_1)],
            ]
        )

        return R_x @ R_y @ R_z

    def quaternion_321_rotation(
        self, qw, qx, qy, qz
    ):  # 3-2-1 rotation, using quaternions, is the inverse of the resultant matrix, so np.
        linalg.inv(R) should be used to match Euler 3-2-1 rotation from inertial to body
        frame.
        # This regular matrix is other way around.

        R = np.array(
            [
                [
                    1 - 2 * (qy**2 + qz**2),
                    2 * (qx * qy - qz * qw),
                    2 * (qx * qz + qy * qw),
                ],
                [
                    2 * (qx * qy + qz * qw),
                    1 - 2 * (qx**2 + qz**2),
                    2 * (qy * qz - qx * qw),
                ],
                [
                    2 * (qx * qz - qy * qw),
                    2 * (qy * qz + qx * qw),
                    1 - 2 * (qx**2 + qy**2),
                ],
            ]
        )

        return np.linalg.inv(R)

    def euler_to_quaternion(self, theta_1, theta_2, theta_3):

        # Extract individual angles
        cr, sr = np.cos(theta_1 * 0.5), np.sin(theta_1 * 0.5)
        cp, sp = np.cos(theta_2 * 0.5), np.sin(theta_2 * 0.5)
        cy, sy = np.cos(theta_3 * 0.5), np.sin(theta_3 * 0.5)

        # Calculate quaternion components, based on the 3-2-1 Euler sequence
        qw = cr * cp * cy + sr * sp * sy  # Real component
        qx = sr * cp * cy - cr * sp * sy  # Imaginary, x
        qy = cr * sp * cy + sr * cp * sy  # Imaginary, y
        qz = cr * cp * sy - sr * sp * cy  # Imaginary, z

        return np.array([qw, qx, qy, qz])  # Sequence w, x, y, z

    def quaternion_product(self, q1, q2):  # Kronecker definition
        """
        Compute the product of two quaternions.
        Each quaternion is represented as an array [qw, qx, qy, qz]

        Args:
        q1 (array): The first quaternion as (w, x, y, z).
        q2 (array): The second quaternion as (w, x, y, z).

        Returns:
        array: The resulting quaternion product as (w, x, y, z).
        """
        w1, x1, y1, z1 = q1
        w2, x2, y2, z2 = q2

        # Calculate the product components
        qw = w1 * w2 - x1 * x2 - y1 * y2 - z1 * z2
        qx = w1 * x2 + x1 * w2 + y1 * z2 - z1 * y2
        qy = w1 * y2 - x1 * z2 + y1 * w2 + z1 * x2
        qz = w1 * z2 + x1 * y2 - y1 * x2 + z1 * w2

        return np.array([qw, qx, qy, qz])  # Sequence w, x, y, z

    def quaternion_to_euler(
        self, qw, qx, qy, qz
    ):  # Verified, using euler_to_quat first, then quat_to_euler

        # Compute Euler angles
        t0 = 2.0 * (qw * qx + qy * qz)
        t1 = 1.0 - 2.0 * (qx**2 + qy**2)
        roll = np.arctan2(t0, t1)

        t2 = 2.0 * (qw * qy - qz * qx)
        t2 = np.clip(t2, -1.0, 1.0)  # Clamp to avoid numerical errors on the boundaries
        pitch = np.arcsin(t2)

        t3 = 2.0 * (qw * qz + qx * qy)
        t4 = 1.0 - 2.0 * (qy**2 + qz**2)
        yaw = np.arctan2(t3, t4)

        return np.array([roll, pitch, yaw])

    # Below definition of DCM to quaternion conversion is incorrect; it assumes a DCM
        consisting of quaternions
    # This raises instability. Below code can be verified with the PyQuaternion module.
    # New DCM_to_quaternion function settles this
    def DCM_to_quaternion_old(self, DCM_matrix):
        a11 = DCM_matrix[0,0]
        a22 = DCM_matrix[1,1]
        a33 = DCM_matrix[2,2]

        qw = 1/2 * np.sqrt(1 + a11 + a22 + a33)
        q1 = 1/2 * np.sqrt(1 + a11 - a22 - a33)
        q2 = 1/2 * np.sqrt(1 - a11 + a22 - a33)
        q3 = 1/2 * np.sqrt(1 - a11 - a22 + a33)

        return np.array([qw, q1, q2, q3])

    def DCM_to_quaternion(self, DCM_matrix):
        a11 = DCM_matrix[0, 0]
        a22 = DCM_matrix[1, 1]
        a33 = DCM_matrix[2, 2]
        trace = a11 + a22 + a33

        if trace > 0:
            qw = 0.5 * np.sqrt(1 + trace)
            qx = (DCM_matrix[2, 1] - DCM_matrix[1, 2]) / (4 * qw)
            qy = (DCM_matrix[0, 2] - DCM_matrix[2, 0]) / (4 * qw)
            qz = (DCM_matrix[1, 0] - DCM_matrix[0, 1]) / (4 * qw)
        elif a11 > a22 and a11 > a33:
            qx = 0.5 * np.sqrt(1 + a11 - a22 - a33)
            qw = (DCM_matrix[2, 1] - DCM_matrix[1, 2]) / (4 * qx)
            qy = (DCM_matrix[0, 1] + DCM_matrix[1, 0]) / (4 * qx)
            qz = (DCM_matrix[0, 2] + DCM_matrix[2, 0]) / (4 * qx)
        elif a22 > a33:
            qy = 0.5 * np.sqrt(1 + a22 - a11 - a33)
            qw = (DCM_matrix[0, 2] - DCM_matrix[2, 0]) / (4 * qy)
            qx = (DCM_matrix[0, 1] + DCM_matrix[1, 0]) / (4 * qy)
            qz = (DCM_matrix[1, 2] + DCM_matrix[2, 1]) / (4 * qy)
        else:
            qz = 0.5 * np.sqrt(1 + a33 - a11 - a22)
            qw = (DCM_matrix[1, 0] - DCM_matrix[0, 1]) / (4 * qz)
            qx = (DCM_matrix[0, 2] + DCM_matrix[2, 0]) / (4 * qz)
            qy = (DCM_matrix[1, 2] + DCM_matrix[2, 1]) / (4 * qz)

        return np.array([qw, qx, qy, qz])



class DisturbanceTorques:
    """
    This class contains the disturbance torques used for the ADCS modelling. These will be
        the gravity gradient torque and
    solar radiation pressure torque described from literature. The Gravity Gradient torque
        function returns a torque vector...
    """

    def __init__(self, inertia_matrix):

        self.inertia_matrix = inertia_matrix
```

```python

    def GGMoon(self, q, position_SC_Moon):

        # Generate the rotation matrix to express the position vector in the body frame
            instead of the inertial Moon-centered frame (retrieve wrt this frame)
        rot = Rotation()
        rot_matrix = rot.quaternion_321_rotation(q[0], q[1], q[2], q[3])

        # Generate the unit vector of the spacecraft position vector in the body frame
        # First approximation -> needs to be verified, in the body frame as well
        R_sc = rot_matrix @ position_SC_Moon  # 3x1 vector
        R_sc_hat = R_sc / np.linalg.norm(R_sc)

        # Calculate gravity gradient torque in the body frame
        T_GG = (
            3
            * Constants.mu_Moon
            / (np.linalg.norm(R_sc) ** 3)
            * (np.cross(R_sc_hat, np.dot(self.inertia_matrix, R_sc_hat)))
        )

        return T_GG

    # DEFAULT: undeployed scenario. For deployed scenario, see next definition
    def SRP(self, q,  position_Sun_Moon, position_SC_Moon):

        rot = Rotation()

        rho_s = 0.6  # These values have been taken from the LUMIO ADCS paper for now; to be
            verified or adjusted
        rho_d = 0.1

        # First, define surface CoM locations wrt satellite CoM [0,0,0] in body frame
        # Numbering according to ADCS LUMIO paper
        # Different reference frame, same as thruster config. Surface 1 is in positive z-axis
            , surface 4 is positive x-axis, surface 6 is positive y-axis
        # All values in meter, sides are either 20cm or 30cm (locations 5 and 6 are on the
            far sides)

        S_loc_1 = np.array([0, 0, 0.1])
        S_loc_2 = np.array([0, 0, -0.1])
        S_loc_3 = np.array([-0.1, 0, 0])
        S_loc_4 = np.array([0.1, 0, 0])
        S_loc_5 = np.array([0, -0.15, 0])
        S_loc_6 = np.array([0, 0.15, 0])

        c_p = np.column_stack(
            [S_loc_1, S_loc_2, S_loc_3, S_loc_4, S_loc_5, S_loc_6]
        )  # Centre of pressure locations for calculation
        n_s = np.array([[0, 0, 1, -1, 0, 0], [0, 0, 0, 0, 1, -1], [-1, 1, 0, 0, 0, 0]])
        r_S_SC = position_Sun_Moon - position_SC_Moon
        S_inertial = np.column_stack(
            [
                (S_loc_1 - r_S_SC) / np.linalg.norm(S_loc_1 - r_S_SC),
                (S_loc_2 - r_S_SC) / np.linalg.norm(S_loc_2 - r_S_SC),
                (S_loc_3 - r_S_SC) / np.linalg.norm(S_loc_3 - r_S_SC),
                (S_loc_4 - r_S_SC) / np.linalg.norm(S_loc_4 - r_S_SC),
                (S_loc_5 - r_S_SC) / np.linalg.norm(S_loc_5 - r_S_SC),
                (S_loc_6 - r_S_SC) / np.linalg.norm(S_loc_6 - r_S_SC),
            ]
        )

        # Rotation matrix to body frame
        rot_matrix = rot.quaternion_321_rotation(q[0], q[1], q[2], q[3])
        # Convert S-matrix to the body frame
        S = rot_matrix @ S_inertial

        A = np.array([6, 6, 6, 6, 4, 4]) * 10 ** (
            -2
        )  # Surface areas of all labeled surfaces, in m2
        F = np.empty((3, len(A)))

        # Calculate the solar intensity at the spacecraft location, as seen in LUMIO ADCS
            paper

        I = Constants.P_solar / (4 * np.pi * np.linalg.norm(r_S_SC) ** 2)

        for i in range(len(A)):
            F_SRP = (
                I
                / Constants.c
                * A[i]
                * np.dot(S.T[i], n_s.T[i])
                * (
                    (1 - rho_s) * S.T[i]
                    + (2 * rho_s * np.dot(S.T[i], n_s.T[i]) + 2 / 3 * rho_d) * n_s.T[i]
                )
            )

            if np.dot(S.T[i], n_s.T[i]) > 0:
                F[0, i], F[1, i], F[2, i] = np.cross(c_p[:, i], F_SRP)

            else:
                F[:, i] = np.zeros(3)

        return F.sum(axis=1)

    def SRP_deployed(self, q, position_Sun_Moon, position_SC_Moon):

        rot = Rotation()

        rho_s = 0.6  # These values have been taken from the LUMIO ADCS paper for now; to be
            verified or adjusted
        rho_d = 0.1

        # First, define surface CoM locations wrt satellite CoM [0,0,0] in body frame
        # Numbering according to ADCS LUMIO paper
        # Different reference frame, same as thruster config. Surface 1 is in positive z-axis
            , surface 4 is positive x-axis, surface 6 is positive y-axis
        # All values in meter, sides are either 20cm or 30cm (locations 5 and 6 are on the
            far sides)
        # Solar arrays lie

        S_loc_1 = np.array([0, 0, 0.1])
        S_loc_2 = np.array([0, 0, -0.1])
        S_loc_3 = np.array([-0.1, 0, 0])
        S_loc_4 = np.array([0.1, 0, 0])
        S_loc_5 = np.array([0, -0.15, 0])
        S_loc_6 = np.array([0, 0.15, 0])
        S_loc_7 = np.array([-0.45, 0, 0])
        S_loc_8 = np.array([-0.45, 0, 0])
        S_loc_9 = np.array([0.45, 0, 0])
        S_loc_10 = np.array([0.45, 0, 0])

        c_p = np.column_stack(
            [S_loc_1, S_loc_2, S_loc_3, S_loc_4, S_loc_5, S_loc_6, S_loc_7, S_loc_8, S_loc_9,
                 S_loc_10]
        ) # Centre of pressure locations for calculation

        r_S_SC = position_Sun_Moon - position_SC_Moon

        S_inertial = np.column_stack(
            [
                (S_loc_1 - r_S_SC) / np.linalg.norm(S_loc_1 - r_S_SC),
                (S_loc_2 - r_S_SC) / np.linalg.norm(S_loc_2 - r_S_SC),
                (S_loc_3 - r_S_SC) / np.linalg.norm(S_loc_3 - r_S_SC),
                (S_loc_4 - r_S_SC) / np.linalg.norm(S_loc_4 - r_S_SC),
                (S_loc_5 - r_S_SC) / np.linalg.norm(S_loc_5 - r_S_SC),
                (S_loc_6 - r_S_SC) / np.linalg.norm(S_loc_6 - r_S_SC),
                (S_loc_7 - r_S_SC) / np.linalg.norm(S_loc_7 - r_S_SC),

                (S_loc_8 - r_S_SC) / np.linalg.norm(S_loc_8 - r_S_SC),
                (S_loc_9 - r_S_SC) / np.linalg.norm(S_loc_9 - r_S_SC),
                (S_loc_10 - r_S_SC) / np.linalg.norm(S_loc_10 - r_S_SC),
            ]
        )

        # Rotation matrix to body frame
        rot_matrix = rot.quaternion_321_rotation(q[0], q[1], q[2], q[3])
        # Convert S-matrix to the body frame
        S = rot_matrix @ S_inertial

        A = np.array([6, 6, 6, 6, 4, 4, 12, 12, 12, 12]) * 10 ** (
            -2
        )  # Surface areas of all labeled surfaces, in m2

        # Define the n_s array as a function of alpha
        def normal_s(alpha):
            return np.array([
                [0, 0, 1, -1, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 1, -1, np.sin(alpha), -np.sin(alpha), np.sin(alpha), -np.sin(
                    alpha)],
                [-1, 1, 0, 0, 0, 0, np.cos(alpha), -np.cos(alpha), np.cos(alpha), -np.cos(
                    alpha)]
            ])

        # Define the objective function to minimize (negative total dot product to maximize
            it)
        def objective(alpha_array):
            alpha = alpha_array[0]
            n_s_alpha = normal_s(alpha)  # Get n_s for this alpha
            total_dot_product = np.sum([np.dot(n_s_alpha[:, i], S[:, i]) for i in range(10)])
            return -total_dot_product  # Minimize the negative to maximize the positive

        # Use scipy.optimize to find the alpha that maximizes the total dot product
        result = minimize(objective, x0=0)  # x0 is the initial guess for alpha
        alpha_optimal = result.x[0]  # Optimal alpha value

        n_s = normal_s(alpha_optimal)

        F = np.empty((3, len(A)))

        # Calculate the solar intensity at the spacecraft location, as seen in LUMIO ADCS
            paper

        I = Constants.P_solar / (4 * np.pi * np.linalg.norm(r_S_SC) ** 2)

        for i in range(len(A)):
            F_SRP = (
                I
                / Constants.c
                * A[i]
                * np.dot(S.T[i], n_s.T[i])
                * (
                    (1 - rho_s) * S.T[i]
                    + (2 * rho_s * np.dot(S.T[i], n_s.T[i]) + 2 / 3 * rho_d) * n_s.T[i]
                )
            )

            if np.dot(S.T[i], n_s.T[i]) > 0:
                F[0, i], F[1, i], F[2, i] = np.cross(c_p[:, i], F_SRP)

            else:
                F[:, i] = np.zeros(3)

        return F.sum(axis=1)

class EphemerisData:
    """
    This class contains the functions for the retrieval of ephemeris data. This data is
        collected from the JPL Horizons module
    and can be retrieved in two separate ways. The function "vectors" retrieves Cartesian
        coordinates and velocities over time, whereas the
    function "keplerian" retrieves the Keplerian elements (semi-major axis, eccentricity, etc
        .) over time.

    Input:
    id: string, for Horizons query
    location: string, for Horizons query
    time_step: number, hours desired for time step of epoch query. Fractionals possible to
        indicate minutes / seconds
    start_date: custom start date, default set to beginning of nominal CAPSTONE mission,
        convention: YEAR-MONTH-DAY HOUR:MINUTE:SECOND
    end_date: custom end date, default set to end of nominal CAPSTONE mission
    """

    def __init__(self, id, location, time_step, start_date=None, end_date=None):

        # If no input start or end dates are given, input the CAPSTONE mission start and end
            dates

        if start_date is None:
            start_date = Constants.start_date_CAPSTONE
        if end_date is None:
            end_date = Constants.end_date_CAPSTONE

        # Validate input start and end date, to verify they fall within the nominal mission
            duration, see definition
        self.validate_dates(start_date, end_date)

        # Convert scalar dates (input) to Julian dates for Horizons query input
        self.t0 = start_date
        self.t1 = end_date
        self.id = id
        self.location = location
        self.time_step = f"{int(time_step␣*␣60)}m"  # Convert fractional time_step to minutes
            , input for class remains hours

    # Validation of correct date usage, simple if-statement for range description, in jd
    def validate_dates(self, start, end):
        mission_start = Time(
            Constants.start_date_CAPSTONE, format="iso", scale="utc"
        ).jd
        mission_end = Time(Constants.end_date_CAPSTONE, format="iso", scale="utc").jd
        start_jd = Time(start, format="iso", scale="utc").jd
        end_jd = Time(end, format="iso", scale="utc").jd

        if not (mission_start <= start_jd <= end_jd <= mission_end):
            raise ValueError(
                "Provided␣dates␣must␣be␣within␣the␣official␣mission␣dates."
            )

    def vectors(self):
        # Retrieve desired object
        object = Horizons(
            id=self.id,
            location=self.location,
            epochs={"start": self.t0, "stop": self.t1, "step": self.time_step},
        )

        # Query Cartesian coordinate vectors, full table
        ephemeris = object.vectors()

        # Query specific columns from list
        cartesian = ephemeris["datetime_jd", "x", "y", "z", "vx", "vy", "vz"]

        # Data storage process
        if self.id == "301":
            body = "Moon"
        elif self.id == "-1176":
            body = "CAPSTONE"
        elif self.id == "10":
            body = "Sun"
        elif self.id == "500":
            body = "Earth"
        else:
            body = "other"

        if self.location == "500@301":
            center = "Moon-centered"
        elif self.location == "500@10":
            center = "Sun-centered"
        elif self.location == "500":
            center = "Earth-centered"
        elif self.location == "500@-1176":
            center = "CAPSTONE-centered"

        name = (
            "/Users/pieter/Library/Mobile␣Documents/com~apple~CloudDocs/Thesis/Research␣Phase
                /lunar_CubeSat/ephemeris_data/cartesian_coordinates_"
            + body
            + "_"
            + center
            + "_"
            + self.t0
            + "_to_"
            + self.t1
            + "_"
            + str(self.time_step)
            + ".dat"
        )

        np.savetxt(
            name,
            cartesian,
        )

        return name, cartesian

    def keplerian(self):
        object = Horizons(
            id=self.id,
            location=self.location,
            epochs={"start": self.t0, "stop": self.t1, "step": self.time_step},
        )

        # Query complete elements table
        ephemeris = object.elements()

        # Query desired Keplerian elements
        keplerian = ephemeris["datetime_jd", "a", "e", "incl", "Omega", "w", "nu"]

        # Data storage process
        if self.id == "301":
            body = "Moon"
        elif self.id == "-1176":
            body = "CAPSTONE"
        elif self.id == "10":
            body = "Sun"
        elif self.id == "500":
            body = "Earth"
        else:
            body = "other"

        if self.location == "500@301":
            center = "Moon-centered"
        elif self.location == "500@10":
            center = "Sun-centered"
        elif self.location == "500":
            center = "Earth-centered"
        elif self.location == "500@-1176":
            center = "CAPSTONE-centered"

        np.savetxt(
            "/Users/pieter/Library/Mobile␣Documents/com~apple~CloudDocs/Thesis/Research␣Phase
```

```python
            /lunar_CubeSat/ephemeris_data/keplerian_elements_"
            + body
            + "_"
            + center
            + "_"
            + self.t0
            + "_to_"
            + self.t1
            + "_"
            + str(self.time_step)
            + ".dat",
            keplerian,
        )

        return keplerian

    def convert_data(self, data_file, control_time_step):

        original_data = np.loadtxt(data_file)

        # Determine time step from original data
        original_time_step = round(
            (original_data[1, 0] - original_data[0, 0]) * 24 * 3600
        )

        # Convert first original data column to seconds since epoch
        for i in range(len(original_data[:, 0])):
            original_data[i, 0] = i * original_time_step
            original_data[i, 1:4] *= Constants.AU
            original_data[i, 4:] *= Constants.AU / 24 / 3600

        # From this point onwards, data becomes converted to the desired number of time steps
             for the control algorithm
        min_number_intervals = int(original_time_step / control_time_step)

        converted_data = np.repeat(original_data, min_number_intervals, axis=0)

        time_list = []

        for j in range(len(original_data[:, 0])):
            for k in range(min_number_intervals):
                new_time = (
                    converted_data[j * min_number_intervals, 0] + control_time_step * k
                )
                time_list.append(new_time)

        converted_data[:, 0] = time_list

        # Save using adjusted naming in different repository
        base_name = os.path.basename(data_file)
        np.savetxt(
            "/Users/pieter/Library/Mobile␣Documents/com~apple~CloudDocs/Thesis/Research␣Phase
                /lunar_CubeSat/converted_ephemeris_data/"
            + "converted_"
            + str(control_time_step)
            + "s_"
            + base_name,
            converted_data,
        )

    def convert_data_interpolated(self, data_file, control_time_step):

        original_data = np.loadtxt(data_file)

        # Determine time step from original data
        original_time_step = round(
            (original_data[1, 0] - original_data[0, 0]) * 24 * 3600
        )

        # Convert first original data column to seconds since epoch
        for i in range(len(original_data[:, 0])):
            original_data[i, 0] = i * original_time_step
            original_data[i, 1:4] *= Constants.AU
            original_data[i, 4:] *= Constants.AU / 24 / 3600

        # Prepare arrays for the new data set
        new_time_steps = np.arange(original_data[0, 0], original_data[-1, 0] + control_time_step, control_time_step)
        converted_data = np.zeros((len(new_time_steps), original_data.shape[1]))

        # Interpolate each column
        for col in range(1, original_data.shape[1]):  # Skip time column for interpolation
            converted_data[:, col] = np.interp(new_time_steps, original_data[:, 0], original_data[:, col])

        # Fill in the new time column
        converted_data[:, 0] = new_time_steps

        # Save using adjusted naming in different repository
        base_name = os.path.basename(data_file)
        np.savetxt(
            "/Users/pieter/Library/Mobile␣Documents/com~apple~CloudDocs/Thesis/Research␣Phase/lunar_CubeSat/converted_ephemeris_data/"
            + "converted_"
            + str(control_time_step)
            + "s_"
            + base_name,
            converted_data,
        )


class PID:
    """
    Functions:
    """

    def __init__(
        self,
        inertia_matrix,
    ):
        """
        Initialize the PID controller with the required inertia matrix.

        Args:
            inertia_matrix (array): A numpy array representing the inertia matrix of the system.
        """
        self.I = inertia_matrix
        self.integral_term = np.zeros(3)  # Integral term is initiated at zero

    def quaternion_error(self, quaternion_vector, quaternion_ref_vector):
        """
        Compute the error between two quaternions.
        Each quaternion is represented as an array np.array([qw, qx, qy, qz]).

        Args:
        quaternion_vector (array): The actual quaternion vector as np.array([qw, qx, qy, qz])
        quaternion_ref_vector (array): The reference quaternion vector as np.array([qw_ref, qx_ref, qy_ref, qz_ref]).

        Returns:
        array: quaternion error vector
        """

        qrw, qr1, qr2, qr3 = quaternion_ref_vector
        qw, q1, q2, q3 = quaternion_vector

        adj_quaternion_vector = np.array([-q1, -q2, -q3, qw])
        adj_matrix = np.array([[qrw, qr3, -qr2, qr1], [-qr3, qrw, qr1, qr2], [qr2, -qr1, qrw, qr3], [-qr1, -qr2, qr3, qrw]])

        qe1, qe2, qe3, qew = adj_matrix @ adj_quaternion_vector

        return np.array([qew, qe1, qe2, qe3])

    def derivative_omega(self, omega_vector, T_d, T_c):
        """
        Calculate the derivative of the angular velocity vector (omega_dot) based on the
            current state and external torques.

        Args:
            T_d (array): Disturbance torque vector.
            T_c (array): Control torque vector.
            omega_vector (array): Current angular velocity vector.

        Returns:
            array: The derivative of the angular velocity (angular acceleration).
        """

        omega_dot_vector = np.linalg.inv(self.I) @ (
            T_c + T_d - np.cross(omega_vector, (self.I @ omega_vector))
        )

        return omega_dot_vector

    def derivative_omega_reactionwheel(self, omega_vector, T_d, T_c, h_rw):
        """
        Specifically for reaction wheel analysis; includes h_rw term in the equations of
            motion
        """

        omega_dot_vector = np.linalg.inv(self.I) @ (
            T_c + T_d - np.cross(omega_vector, (self.I @ omega_vector + h_rw))
        )

        return omega_dot_vector


    def derivative_quaternion(self, quaternion_vector, omega_vector):
        """
        Compute the time derivative of a quaternion based on the current angular velocity.

        Args:
            quaternion_vector (array): The current quaternion vector as np.array([qw, qx, qy,
                 qz]).
            omega_vector (array): The angular velocity vector.

        Returns:
            array: The time derivative of the quaternion.
        """

        # Again, order qw qx qy qz
        omega_q_vector = np.append(0, omega_vector)
        rot = Rotation()

        q_dot = 0.5 * rot.quaternion_product(quaternion_vector, omega_q_vector)

        return q_dot

    def derivative_quaternion_error(
        self, quaternion_vector, quaternion_ref_vector, omega_vector
    ):  # See notes in notebook
        """
        Calculate the derivative of the quaternion error between the reference and current
            quaternion.

        Args:
            quaternion_vector (array): The current quaternion vector as np.array([qw, qx, qy,
                 qz]).
            quaternion_ref_vector (array): The reference quaternion vector as np.array([qw,
                qx, qy, qz]).
            omega_vector (array): The angular velocity vector.

        Returns:
            array: The derivative of the quaternion error.
        """

        qcw, qcx, qcy, qcz = quaternion_ref_vector
        q_dot = self.derivative_quaternion(quaternion_vector, omega_vector)

        ref_matrix = np.array(
            [
                [qcw, qcx, qcy, qcz],
                [qcx, -qcw, qcz, -qcy],
                [qcy, -qcz, -qcw, qcx],
                [qcz, qcy, -qcx, -qcw],
            ]
        )

        return ref_matrix @ q_dot

    def control_torque(
        self, quaternion_vector, quaternion_ref_vector, omega_vector, k_p, k_i, k_d, k_s, dt
    ):
        """
        Compute the control torque based on PID control laws using the quaternion error and
            its derivative.

        Args:
            quaternion_vector (array): The current quaternion vector.
            quaternion_ref_vector (array): The reference quaternion vector.
            omega_vector (array): The angular velocity vector.
            k_p (array): Proportional gain coefficients (k_p_x, k_p_y, k_p_z).
            k_i (array): Integral gain coefficients (k_i_x, k_i_y, k_i_z).
            k_d (array): Derivative gain coefficients (k_d_x, k_d_y, k_d_z).
            dt (float): Time step for the integral calculation.

        Returns:
            array: Control torque vector [T_c_x, T_c_y, T_c_z].
        """
        qew, qex, qey, qez = self.quaternion_error(
            quaternion_vector, quaternion_ref_vector
        )  # quaternion error components

        omega_x, omega_y, omega_z = omega_vector

        k_p_1, k_p_2, k_p_3 = k_p  # proportional gain
        k_i_1, k_i_2, k_i_3 = k_i  # integral gain
        k_d_1, k_d_2, k_d_3 = k_d  # derivative gain
        k_s_1, k_s_2, k_s_3 = k_s  # speed gain


        int_x, int_y, int_z = self.integral_term

        # control torque in the x, y and z directions
        T_c_x = k_s_1 * (
            k_p_1 * qex + k_i_1 * int_x - k_d_1 * (omega_x)
        )
        T_c_y = k_s_2 * (k_p_2 * qey + k_i_2 * int_y - k_d_2 * (omega_y))
        T_c_z = k_s_3 * (k_p_3 * qez + k_i_3 * int_z - k_d_3 * (omega_z))

        self.integral_term += np.array([qex, qey, qez]) * dt

        return np.array([T_c_x, T_c_y, T_c_z])

    def rk4_integrator(self, func, y, dt, *args):
        """
        General RK4 integrator.
        Args:
            func: The function to calculate derivatives, signature func(y, *args)
            y: Current state variable (np.array)
            dt: Time step (float)
            *args: Additional arguments required by `func`
        Returns:
            np.array: Updated state after dt
        """
        k1 = func(y, *args) * dt
        k2 = func(y + 0.5 * k1, *args) * dt
        k3 = func(y + 0.5 * k2, *args) * dt
        k4 = func(y + k3, *args) * dt
        return y + (k1 + 2 * k2 + 2 * k3 + k4) / 6

    def euler_integrator(self, func, y, dt, *args):
        """
        Basic Euler integrator.
        Args:
            func: The function to calculate derivatives, with signature func(y, *args)
            y: Current state variable (np.array)
            dt: Time step (float)
            *args: Additional arguments required by `func`
        Returns:
            np.array: Updated state after dt
        """
        dy = func(y, *args)  # Calculate the derivative
        return y + dy * dt  # Update state

    def reference_omega(self, q_ref, omega_vector):
        q_ref_dot = self.derivative_quaternion(q_ref, omega_vector)
        q_ref_dot_adj = np.array([q_ref_dot[1], q_ref_dot[2], q_ref_dot[3], q_ref_dot[0]])

        mat = np.array([[q_ref[0], q_ref[3], -q_ref[2], -q_ref[0]], [q_ref[0], q_ref[3], -
            q_ref[2], -q_ref[0]], [q_ref[0], q_ref[3], -q_ref[2], -q_ref[0]]])

        return 2 * mat @ q_ref_dot_adj

    def reference_quaternion(self, position_SC_Moon, quaternion_vector):
        """
        This function takes the spacecraft position vector with respect to the Moon and the
            current spacecraft
        attitude defined in quaternions to calculate the reference quaternion at one instance
             of time. The
        reference is defined to have the upper panel (y+ direction) of the LUMIO spacecraft
            pointed towards
        the Moon and this should at all times be adhered to. The output is a vector [qw, qx,
            qy, qz] with the
        reference quaternion vectors to be used within the PID algorithm for error
            calculation.
        """

        qw, qx, qy, qz = quaternion_vector

        rot = Rotation()
        direction_SC_Moon = (
            -position_SC_Moon
        )  # Define the direction vector from the spacecraft to the Moon, inertial (Moon-
            centered) frame
        R = rot.quaternion_321_rotation(
            qw, qx, qy, qz
        )  # Rotation matrix for conversion from inertial to body frame
        direction_SC_Moon_body = np.dot(
            R, direction_SC_Moon
        )  # Convert to the body frame
        panel_body = Constants.LUMIO_loc_pp  # Retrieve desired panel CoM location

        n_current = panel_body / np.linalg.norm(
            panel_body
        )  # compare current orientation in the body frame to desired orientation in the body
             frame
        n_desired = direction_SC_Moon_body / np.linalg.norm(direction_SC_Moon_body)
        q_rot = np.array(
            [
                np.sqrt(np.linalg.norm(n_current) ** 2 * np.linalg.norm(n_desired) ** 2)
                + np.dot(n_current, n_desired)
            ]
            + list(np.cross(n_current, n_desired))
        )  # Rotation quaternion so that the panel is oriented towards the Moon
        q_rot /= np.linalg.norm(q_rot)  # Normalize the quaternion

        # Calculate the reference quaternion to adhere to, combination of current quaternion
            (rotation representation) and newly calculated quaternion q_rot
        return rot.quaternion_product(q_rot, quaternion_vector)


    def reference_quaternion_paper(self, position_SC_Moon, position_Sun_Moon):
        """
        Based on the paper "ATTITUDE CONTROL FOR THE LUMIO CUBESAT IN DEEP SPACE", outputting
        the desired attitude for the LUMIO spacecraft based on power maximisation. It can be
        shown that this reference frame maximizes power generation by allowing the solar
            arrays
        to be always normal to the Sun vector.

        ADJUSTMENT: in this case, considering the usage of a different reference frame
        , the negative y-plane should always be pointed towards the Moon,
        """
        rot = Rotation()

        Sun_pointing_vector = (position_Sun_Moon - position_SC_Moon) / np.linalg.norm(
            position_Sun_Moon - position_SC_Moon)
        Moon_pointing_vector = - position_SC_Moon / np.linalg.norm(position_SC_Moon)

        x1 = Moon_pointing_vector
        x2 = np.cross(Sun_pointing_vector, x1) / np.linalg.norm(np.cross(Sun_pointing_vector,
             x1))
        x3 = np.cross(x1, x2) / np.linalg.norm(np.cross(x1, x2))

        A_d = np.column_stack((x2, -x1, x3))

        return rot.DCM_to_quaternion(A_d)


class Visualization:
    """
    Visualization class for result from attitude control.
    """

    def __init__(self, time_array):
        self.t = time_array

    def extract_dates_from_filename(self, filename):
        # Define the regular expression pattern to match the dates
        pattern = r'_(\d{4}-\d{2}-\d{2}␣\d{2}:\d{2})_to_(\d{4}-\d{2}-\d{2}␣\d{2}:\d{2})_'

        # Search the pattern in the filename
        match = re.search(pattern, filename)

        if match:
            # Extract the start and end date strings
            start_date_str = match.group(1)
            end_date_str = match.group(2)

            # Convert the strings to datetime objects
            start_date = datetime.strptime(start_date_str, '%Y-%m-%d␣%H:%M')
            end_date = datetime.strptime(end_date_str, '%Y-%m-%d␣%H:%M')

            return start_date, end_date
        else:
            raise ValueError("The␣filename␣does␣not␣match␣the␣expected␣pattern")

    def plot_trajectories(self, *data_files):
        plt.style.use('fast')
        fig = plt.figure(figsize=(10, 10))  # Set the figure size to 10x10
        ax = fig.add_subplot(111, projection="3d")

        # Define reference frames with adjusted Moon radius for a smaller sphere
        reference_frames = {
            "Sun": Constants.R_Sun,
            "Moon": Constants.R_Moon * 0.5,  # Reduced radius for Moon
            "Earth": Constants.R_Earth,
            "CAPSTONE": 0.2,
        }  # Radii in meters
        used_centers = set()

        first_file = True
        all_epochs = None

        for data_file in data_files:
            # Extracting information from file name
            parts = data_file.split("_")
            body_name = parts[6]
            center_part = parts[7]
            center = center_part.split("-")[0]

            print(center)

            # Check if center has been used consistently
            if used_centers and center not in used_centers:
                raise ValueError(
                    f"Inconsistent␣reference␣systems:␣{used_centers.pop()}␣vs␣{center}.␣All␣
                        data␣must␣be␣in␣the␣same␣reference␣frame."
                )
            used_centers.add(center)

            # Reading data
            data = np.loadtxt(data_file)
            epochs, x, y, z, vx, vy, vz = data.T
            x, y, z = x, y, z  # AU to meters

            # Plotting trajectories
            ax.plot(x, y, z, label=body_name)

            # Check epochs
            if all_epochs is None:
                all_epochs = epochs
            elif not np.array_equal(all_epochs, epochs):
                raise ValueError(
                    "Epochs␣do␣not␣match␣across␣data␣files.␣Ensure␣all␣input␣files␣cover␣the␣
                        same␣time␣periods."
                )

            # Centering reference body
            if center in reference_frames:
                # Plotting central body as a sphere
                radius = reference_frames[center]
                u, v = np.mgrid[0:2 * np.pi:100j, 0:np.pi:50j]
                sphere_x = radius * np.cos(u) * np.sin(v)
                sphere_y = radius * np.sin(u) * np.sin(v)
                sphere_z = radius * np.cos(v)
                ax.plot_wireframe(sphere_x, sphere_y, sphere_z, color="grey", alpha=0.5)
                ax.text(0, 0, -5 * radius, center, color="black", fontsize=16, ha="center")

        # Setting limits for equal aspect ratio
        max_radius = max(
            abs(ax.get_xlim()[0]), abs(ax.get_ylim()[0]), abs(ax.get_zlim()[0])
        )
        ax.set_xlim([-max_radius, max_radius])
        ax.set_ylim([-max_radius, max_radius])
        ax.set_zlim([-max_radius, max_radius])

        # Add labels, title, and legend with adjusted font sizes
        ax.set_xlabel("X␣(m)", fontsize=16)
        ax.set_ylabel("Y␣(m)", fontsize=16)
        ax.set_zlabel("Z␣(m)", fontsize=16)
        ax.legend(fontsize=16)

        # Adjust tick parameters for axis font sizes
        ax.tick_params(axis='both', which='major', labelsize=16)

        plt.show()


        # # Save plot
        # today = date.today()
        # plt.savefig("/Users/pieter/Library/Mobile Documents/com~apple~CloudDocs/Thesis/
            Research Phase/lunar_CubeSat/figures/trajectory_rundate_" + str(today) + "
            _length_" + str(round(float(len(self.t)) / 60 / 60 / 24, 3)) + "_days_timestep_"
            + str(self.t[1] - self.t[0]) + "_seconds.png")

    # def euler_versus_time(self, quaternion_ref_array, quaternion_array):

    #     """
    #     This function consists of two distinct parts: one shows the actual Euler angles and
    #     the commanded both plotted in the same graph. The next shows the offset of the
        Euler
    #     angles over time, compared to the maximum allowed offset of 0.1degrees during the
    #     Science & Navigation phase.
    #     """

    #     rot = Rotation()

    #     euler_ref_vis = np.empty((3, len(self.t)))
    #     euler_vis = np.empty((3, len(self.t)))

    #     for index in range(len(self.t)):

    #         # Extract quaternion value for each time step
    #         qwr, qxr, qyr, qzr = quaternion_ref_array[:, index]
    #         qw, qx, qy, qz = quaternion_array[:, index]

    #         # Append do visualisation array
    #         euler_ref_vis[:, index] = np.rad2deg(rot.quaternion_to_euler(qwr, qxr, qyr, qzr
        ))
    #         euler_vis[:, index] = np.rad2deg(rot.quaternion_to_euler(qw, qx, qy, qz))

    #     plt.figure(1)
    #     plt.figure(figsize=(15, 5))
    #     plt.rcParams.update({"font.size": 16})
    #     plt.plot(
    #         self.t,
    #         euler_ref_vis[0, :],
    #         color="red",
    #         linestyle="dashed",
    #         label="Commanded",
    #     )

    #     plt.plot(self.t, euler_vis[0, :], label="Actual")
    #     plt.xlabel("Time since epoch [s]")
    #     plt.ylabel("$\phi$ [deg]")
    #     plt.title("Roll angle commanded and actual signal versus time")
    #     plt.legend()
    #     plt.show()

    #     plt.figure(2)
    #     plt.figure(figsize=(15, 5))
    #     plt.rcParams.update({"font.size": 16})
    #     plt.plot(
    #         self.t,
    #         euler_ref_vis[1, :],
    #         color="red",
    #         linestyle="dashed",
    #         label="Commanded",
    #     )

    #     plt.plot(self.t, euler_vis[1, :], label="Actual")
    #     plt.xlabel("Time since epoch [s]")
    #     plt.ylabel("$\\theta$ [deg]")
    #     plt.title("Pitch angle commanded and actual signal versus time")
    #     plt.legend()
    #     plt.show()

    #     plt.figure(3)
    #     plt.figure(figsize=(15, 5))
    #     plt.rcParams.update({"font.size": 16})
    #     plt.plot(
    #         self.t,
    #         euler_ref_vis[2, :],
    #         color="red",
    #         linestyle="dashed",
    #         label="Commanded",
    #     )

    #     plt.plot(self.t, euler_vis[2, :], label="Actual")
    #     plt.xlabel("Time since epoch [s]")
    #     plt.ylabel("$\psi$ [deg]")
    #     plt.title("Yaw angle commanded and actual signal versus time")
    #     plt.legend()
    #     plt.show()

    #     # Offset

    #     euler_offset_vis = np.abs(euler_vis - euler_ref_vis)
    #     acc_req  = np.full(len(self.t), 0.1)

    #     plt.figure(4)
    #     plt.figure(figsize=(15,5))
    #     plt.rcParams.update({"font.size": 16})
    #     plt.plot(
    #         self.t,
    #         acc_req,
    #         color="red",
    #         linestyle="dashed",
    #         label="ADCS requirement",
    #     )
    #     plt.plot(self.t, euler_offset_vis[0, :], label="Actual offset")
    #     plt.xlabel("Time since epoch [s]")
    #     plt.ylabel("$\Delta$$\phi$ [deg]")
    #     plt.title("Roll angle offset versus time")
    #     plt.legend()
    #     plt.show()

    #     plt.figure(5)
    #     plt.figure(figsize=(15,5))
    #     plt.rcParams.update({"font.size": 16})
    #     plt.plot(
    #         self.t,
    #         acc_req,
    #         color="red",
    #         linestyle="dashed",
    #         label="ADCS requirement",
    #     )
    #     plt.plot(self.t, euler_offset_vis[1, :], label="Actual offset")
    #     plt.xlabel("Time since epoch [s]")
    #     plt.ylabel("$\Delta$$\\theta$ [deg]")
    #     plt.title("Pitch angle offset versus time")
    #     plt.legend()
    #     plt.show()

    #     plt.figure(6)
    #     plt.figure(figsize=(15,5))
    #     plt.rcParams.update({"font.size": 16})
    #     plt.plot(
    #         self.t,
    #         acc_req,
    #         color="red",
    #         linestyle="dashed",
    #         label="ADCS requirement",
    #     )
    #     plt.plot(self.t, euler_offset_vis[2, :], label="Actual offset")
    #     plt.xlabel("Time since epoch [s]")
    #     plt.ylabel("$\Delta$$\psi$ [deg]")
    #     plt.title("Yaw angle offset versus time")
    #     plt.legend()
    #     plt.show()

    #     plt.figure(8)
    #     plt.figure(figsize=(15, 5))
    #     plt.rcParams.update({"font.size": 16})
    #     plt.plot(self.t, acc_req, color="red", linestyle="dashed", label="ADCS requirement
        ")
    #     plt.plot(self.t, euler_offset_vis[0, :], label="Roll offset ($\Delta\phi$)")
    #     plt.plot(self.t, euler_offset_vis[1, :], label="Pitch offset ($\Delta\\theta$)")
    #     plt.plot(self.t, euler_offset_vis[2, :], label="Yaw offset ($\Delta\psi$)")
    #     plt.xlabel("Time since epoch [s]")
    #     plt.ylabel("Angle Offset [deg]")
    #     plt.title("Angle Offset versus Time for Roll, Pitch, and Yaw")
    #     plt.legend()
    #     plt.show()

    #     # Accuracy
    #     plt.figure(7)
    #     plt.figure(figsize=(15,5))
    #     plt.rcParams.update({"font.size": 16})
    #     plt.plot(
    #         self.t,
    #         np.full(len(self.t), 1),
    #         color="red",
    #         linestyle="dashed",
    #         label="ADCS requirement",
    #     )
    #     plt.plot(self.t, acc_req / euler_offset_vis[0, :], label="$\phi_{max}$ / $\phi_{err
        }$")
    #     plt.plot(self.t, acc_req / euler_offset_vis[1, :], label="$\\theta_{max}$ / $\\
        theta_{err}$")
    #     plt.plot(self.t, acc_req / euler_offset_vis[2, :], label="$\\psi_{max}$ / $\\psi_{
        err}$")
    #     plt.ylim(0,10)
    #     plt.xlabel("Time since epoch [s]")
    #     plt.ylabel("Accuracy [-]")
    #     plt.title("Accuracy of all Euler angles over time")
    #     plt.legend()
    #     plt.show()

    def quaternion_versus_time(self, quaternion_ref_array, quaternion_array):

        # Setup for quaternion error array

        pd = PID(Constants.inertia_matrix)
        # Setup for quaternion error array
        qe = np.zeros_like(quaternion_array)  # Initialize error quaternion array with the
            same shape as quaternion_array

        # Iterate over each time step to compute the error quaternion
        for i in range(quaternion_array.shape[1]):  # Loop over the time array's length
            qe[:, i] = pd.quaternion_error(quaternion_array[:, i], quaternion_ref_array[:, i
                ])  # Compute quaternion error for each time step
            qe[0,i] = quaternion_array[0,i] - quaternion_ref_array[0,i]

        plt.figure(1)
        plt.style.use('seaborn-v0_8')
        plt.figure(figsize=(5, 5))
        plt.plot(
            self.t,
            quaternion_ref_array[0, :],
            color="red",
            linestyle="dashed",
            label="Commanded",
        )
        plt.plot(self.t, quaternion_array[0, :], label="Actual")
        plt.xlabel("Time␣[s]", fontsize = 16)
        plt.ylabel("$q_w$␣[-]", fontsize = 16)
        plt.xlim(left=0)
        plt.tick_params(axis='both', which='major', labelsize=16)
        plt.legend(fontsize=16)
        # Increase the size of the scientific notation offset
        plt.show()

        # Plot for quaternion q1
        plt.figure(2)
        plt.figure(figsize=(5, 5))
        plt.rcParams.update({"font.size": 16})
        plt.plot(
            self.t,
            quaternion_ref_array[1, :],
            color="red",
            linestyle="dashed",
            label="Commanded",
        )
        plt.plot(self.t, quaternion_array[1, :], label="Actual")
        plt.xlabel("Time␣[s]", fontsize=16)
        plt.ylabel("$q_1$␣[-]", fontsize=16)
        plt.xlim(left=0)
        plt.tick_params(axis='both', which='major', labelsize=16)
        plt.legend(fontsize=16)  # Legend font size
        plt.show()

        # Plot for quaternion q2
        plt.figure(3)
        plt.figure(figsize=(5, 5))
        plt.rcParams.update({"font.size": 16})
        plt.plot(
            self.t,
            quaternion_ref_array[2, :],
            color="red",
            linestyle="dashed",
            label="Commanded",
        )
        plt.plot(self.t, quaternion_array[2, :], label="Actual")
        plt.xlabel("Time␣[s]", fontsize=16)
        plt.ylabel("$q_2$␣[-]", fontsize=16)
        plt.xlim(left=0)
        plt.tick_params(axis='both', which='major', labelsize=16)
        plt.legend(fontsize=16)  # Legend font size
        plt.show()

        # Plot for quaternion q3
        plt.figure(4)
        plt.figure(figsize=(5, 5))
        plt.rcParams.update({"font.size": 16})
        plt.plot(
            self.t,
            quaternion_ref_array[3, :],
            color="red",
            linestyle="dashed",
            label="Commanded",
        )
        plt.plot(self.t, quaternion_array[3, :], label="Actual")
        plt.xlabel("Time␣[s]", fontsize=16)
        plt.ylabel("$q_3$␣[-]", fontsize=16)
        plt.xlim(left=0)
        plt.tick_params(axis='both', which='major', labelsize=16)
        plt.legend(fontsize=16)  # Legend font size
        plt.show()

        # Plot for quaternion error components
        plt.figure(figsize=(5, 5))
        # Plot all components of qe
        plt.plot(self.t, qe[0, :], label="$q_{e,w}$", linestyle="solid", color="blue")
        plt.plot(self.t, qe[1, :], label="$q_{e,1}$", linestyle="dashed", color="red")
        plt.plot(self.t, qe[2, :], label="$q_{e,2}$", linestyle="dotted", color="green")
        plt.plot(self.t, qe[3, :], label="$q_{e,3}$", linestyle="dashdot", color="orange")
        # Axis labels
        plt.xlabel("Time␣[s]", fontsize=16)
        plt.ylabel("$q_{e}$␣[-]", fontsize=16)
        # Set x-axis limits
        plt.xlim(left=0)
        # Customize ticks
        plt.tick_params(axis='both', which='major', labelsize=16)
        ax = plt.gca()  # Get the current axis
        ax.yaxis.get_offset_text().set_fontsize(16)  # Adjust the font size of the 1e-6
        # Add legend
        plt.legend(fontsize=16)
        # Show the plot
        plt.show()

        # Plot for quaternion relative error components (in percentages)
        plt.figure(figsize=(15, 5))

        # Compute relative error in percentages
        relative_error_0 = np.abs(qe[0, :] / quaternion_array[0, :]) * 100
        relative_error_1 = np.abs(qe[1, :] / quaternion_array[1, :]) * 100
        relative_error_2 = np.abs(qe[2, :] / quaternion_array[2, :]) * 100
        relative_error_3 = np.abs(qe[3, :] / quaternion_array[3, :]) * 100


        # Plot all relative error components
        plt.plot(self.t/ (24*3600), relative_error_0, label="$q_{e,w}␣/␣q_{w}$␣(%)",
            linestyle="solid", color="blue")
        plt.plot(self.t/ (24*3600), relative_error_1, label="$q_{e,1}␣/␣q_{1}$␣(%)",
            linestyle="dashed", color="red")
        plt.plot(self.t/ (24*3600), relative_error_2, label="$q_{e,2}␣/␣q_{2}$␣(%)",
            linestyle="dotted", color="green")
        plt.plot(self.t/ (24*3600), relative_error_3, label="$q_{e,3}␣/␣q_{3}$␣(%)",
            linestyle="dashdot", color="orange")

        # Axis labels
        plt.xlabel("Time␣[days]", fontsize=16)
        plt.ylabel("Relative␣Error␣[%]", fontsize=16)
        plt.yscale('symlog', linthresh=1e-8)
        # Set x-axis limits
        plt.xlim(left=0)

        # Customize ticks
        plt.tick_params(axis='both', which='major', labelsize=16)

        # Add legend
        plt.legend(fontsize=16)

        # Show the plot
        plt.show()



    def disturbance_torque_versus_time(self, T_d, T_GG, T_SRP):

        # Apply seaborn style
        plt.style.use('seaborn-v0_8')

        # Disturbance torques
        plt.figure(figsize=(15, 5))
        plt.plot(self.t / (24*3600), T_d[0, :], label="$T_{d,x}$", linestyle="dashed", color=
            "red")
        plt.plot(self.t / (24*3600), T_d[1, :], label="$T_{d,y}$", linestyle="dotted", color=
            "blue")
        plt.plot(self.t / (24*3600), T_d[2, :], label="$T_{d,z}$", linestyle="solid", color="
            green")
        plt.xlabel("Time␣[days]", fontsize=16)
        plt.ylabel("$T_d$␣[Nm]", fontsize=16)
        plt.xlim(left = 0)
        plt.tick_params(axis='both', which='major', labelsize=16)
        ax = plt.gca()  # Get the current axis
        ax.yaxis.get_offset_text().set_fontsize(16)  # Adjust the font size of the 1e-6
        plt.legend(fontsize=16)
        plt.show()

        # Gravity gradient torques
        plt.figure(figsize=(15, 5))
        plt.plot(self.t / (24*3600), T_GG[0, :], label="$T_{GG,x}$", linestyle="dashed",
            color="red")
        plt.plot(self.t / (24*3600), T_GG[1, :], label="$T_{GG,y}$", linestyle="dotted",
            color="blue")
        plt.plot(self.t / (24*3600), T_GG[2, :], label="$T_{GG,z}$", linestyle="solid", color
            ="green")
        plt.xlabel("Time␣[days]", fontsize=16)
        plt.ylabel("$T_{GG}$␣[Nm]", fontsize=16)
        plt.xlim(left = 0)
        plt.tick_params(axis='both', which='major', labelsize=16)
        ax = plt.gca()  # Get the current axis
        ax.yaxis.get_offset_text().set_fontsize(16)  # Adjust the font size of the 1e-6
        plt.legend(fontsize=16)
        plt.show()

        # Solar radiation pressure torques
        plt.figure(figsize=(15, 5))
        plt.plot(self.t / (24*3600), T_SRP[0, :], label="$T_{SRP,x}$", linestyle="dashed",
            color="red")
        plt.plot(self.t / (24*3600), T_SRP[1, :], label="$T_{SRP,y}$", linestyle="dotted",
            color="blue")
        plt.plot(self.t / (24*3600), T_SRP[2, :], label="$T_{SRP,z}$", linestyle="solid",
            color="green")
        plt.xlabel("Time␣[days]", fontsize=16)
        plt.ylabel("$T_{SRP}$␣[Nm]", fontsize=16)
        plt.xlim(left = 0)
        plt.tick_params(axis='both', which='major', labelsize=16)
        ax = plt.gca()  # Get the current axis
        ax.yaxis.get_offset_text().set_fontsize(16)  # Adjust the font size of the 1e-6
        plt.legend(fontsize=16)
        plt.show()

    def control_torque_versus_time(self, T_c):

        # Apply seaborn style
        plt.style.use('seaborn-v0_8')

        # Create a single plot with all components of control torque
        plt.figure(figsize=(15, 5))
        plt.plot(self.t, T_c[0, :], label="$T_{c,x}$", linestyle="dashed", color="red")
        plt.plot(self.t, T_c[1, :], label="$T_{c,y}$", linestyle="dotted", color="blue")
        plt.plot(self.t, T_c[2, :], label="$T_{c,z}$", linestyle="solid", color="green")
        plt.xlabel("Time␣[s]", fontsize=16)
        plt.ylabel("$T_c$␣[Nm]", fontsize=16)
        plt.xlim(left = 0)
        plt.tick_params(axis='both', which='major', labelsize=16)
        ax = plt.gca()  # Get the current axis
        ax.yaxis.get_offset_text().set_fontsize(16)  # Adjust the font size of the 1e-6
        plt.legend(fontsize=16)  # Add labels for x, y, z components
        plt.show()

    def RW_torque_per_wheel(self, torque_matrix):
        """
        Visualizes the torque values for each reaction wheel and their total torque.

        Args:
            torque_matrix (numpy array): 4xlen(self.t) array with torque values for the
                reaction wheels.

        """
        # Apply seaborn style
        plt.style.use('seaborn-v0_8')
        plt.rcParams.update({"font.size": 16})

        # Plot torque for individual reaction wheels and total torque
        plt.figure(figsize=(15, 5))
        for i in range(4):  # Loop through the 4 reaction wheels
            plt.plot(self.t / (24 * 3600), torque_matrix[i, :], label=f"RW␣{i␣+␣1}")
```

```python

        # Labels and settings
        plt.xlabel("Time␣[days]", fontsize=16)
        plt.ylabel("$T_{rw}$␣[Nm]", fontsize=16)
        # plt.xlim(left=0)
        ax = plt.gca()  # Get the current axis
        ax.yaxis.get_offset_text().set_fontsize(16)  # Adjust the font size of the 1e-6
        plt.tick_params(axis='both', which='major', labelsize=16)
        plt.legend(fontsize=16, loc='upper␣left')
        plt.show()


    def RW_PE_versus_time(self, PE, select):
        # Apply seaborn style
        plt.style.use('seaborn-v0_8')
        plt.rcParams.update({"font.size": 16})

        # Define labels based on selection
        if select == "P":
            y_label = "$P_{rw}$␣[W]"
        elif select == "E":
            y_label = "$E_{rw}$␣[J]"
        else:
            raise ValueError("Invalid␣'select'␣argument.␣Use␣'P'␣for␣power␣or␣'E'␣for␣energy.
                ")

        # Plot power/energy for individual reaction wheels and total
        plt.figure(figsize=(15, 5))
        for i in range(4):  # Loop through the 4 reaction wheels
            plt.plot(self.t / (24 * 3600), PE[i, :], label=f"RW␣{i␣+␣1}")

        # Plot total power/energy
        total_PE = np.sum(PE, axis=0)
        plt.plot(self.t / (24 * 3600), total_PE, label="Total", linestyle="dashed", color="
            black", linewidth=2)

        # Labels and settings
        plt.xlabel("Time␣[days]", fontsize=16)
        plt.ylabel(y_label, fontsize=16)
        # plt.xlim(left=0)
        plt.ylim(bottom=0)
        plt.tick_params(axis='both', which='major', labelsize=16)
        plt.legend(fontsize=16, loc='upper␣left')
        plt.show()

    def thruster_PE_vs_time(self, PE, select):
        num_thrusters = PE.shape[0]  # Determine the number of thrusters from the array shape

        # Apply seaborn style
        plt.style.use('seaborn-v0_8')

        for i in range(num_thrusters):
            plt.figure(figsize=(5, 5))  # Individual plot size
            plt.plot(self.t / (24*3600), PE[i, :])
            plt.xlabel("Time␣[days]", fontsize=14)
            plt.xlim(left = 0)
            if select == "P":
                plt.ylabel(f"$P_{{thrust,{i+1}}}$␣[W]", fontsize=14)
            elif select == "E":
                plt.ylabel(f"$E_{{thrust,{i+1}}}$␣[J]", fontsize=14)
            plt.tick_params(axis='both', which='major', labelsize=14)
            plt.show()

        # Combined plot for total power or energy of all thrusters
        plt.figure(figsize=(15, 5))  # Combined plot size
        total_PE = np.sum(PE, axis=0)  # Summing up all thrusters' power or energy over time
        for i in range(num_thrusters):
            plt.plot(self.t/ (24*3600), PE[i, :], label=f'Thruster␣{i+1}')
        plt.plot(self.t/ (24*3600), total_PE, label='Total', linestyle='--', linewidth=2.0,
            color = 'black')
        plt.xlabel("Time␣[days]", fontsize=16)
        plt.xlim(left = 0)
        if select == "P":
            plt.ylabel("$P_{thrust}$␣[W]", fontsize=16)
        elif select == "E":
            plt.ylabel("$E_{thrust}$␣[J]", fontsize=16)
        plt.tick_params(axis='both', which='major', labelsize=16)
        plt.legend(fontsize=16)
        plt.show()

    # TO DO
    def combined_PE_vs_time(self, RW_PE, Thruster_PE, select):
        plt.rcParams.update({"font.size": 16})

        # Calculate the total power or energy over time for reaction wheels and thrusters
        total_RW_PE = np.sum(RW_PE, axis=0)
        total_Thruster_PE = np.sum(Thruster_PE, axis=0)

        plt.figure(figsize=(15, 5))

        # Plotting the total from reaction wheels
        plt.plot(self.t / (24*3600), total_RW_PE, label='Total␣Reaction␣Wheels')

        # Plotting the total from thrusters
        plt.plot(self.t / (24*3600), total_Thruster_PE, label='Total␣Thrusters')
        plt.yscale('log')
        plt.xlabel("Time␣[days]")
        if select == "P":
            plt.ylabel("Total␣Power␣[W]")
            plt.title("Total␣power␣usage␣of␣reaction␣wheels␣and␣thrusters␣over␣time")
        elif select == "E":
            plt.ylabel("Total␣Energy␣[J]")
            plt.title("Total␣energy␣consumption␣of␣reaction␣wheels␣and␣thrusters␣over␣time")

        plt.legend()
        plt.show()

    def omega_versus_time(self, omega_array):

        # Apply seaborn style
        plt.style.use('seaborn-v0_8')

        # Create a single plot with all angular velocity components
        plt.figure(figsize=(15, 5))
        plt.plot(self.t, omega_array[0, :], label="$\\omega_x$", linestyle="dashed", color="
            red")
        plt.plot(self.t, omega_array[1, :], label="$\\omega_y$", linestyle="dotted", color="
            blue")
        plt.plot(self.t, omega_array[2, :], label="$\\omega_z$", linestyle="solid", color="
            green")
        plt.xlabel("Time␣[s]", fontsize=16)
        plt.ylabel("$\\omega$␣[rad/s]", fontsize=16)
        plt.xlim(left = 0)
        plt.tick_params(axis='both', which='major', labelsize=16)
        ax = plt.gca()  # Get the current axis
        ax.yaxis.get_offset_text().set_fontsize(16)  # Adjust the font size of the 1e-6
        plt.legend(fontsize=16)  # Add labels for x, y, z components
        plt.show()

    def h_versus_time(self, h_array):
        # Import Seaborn style
        plt.style.use('seaborn-v0_8')

        # Reaction wheel saturation limits
        h_sat_1_2_3 = np.full(len(self.t), 0.1)  # Saturation limit for wheels 1, 2, and 3 [
            Nms]
        h_sat_4 = np.full(len(self.t), 0.05)  # Saturation limit for wheel 4 [Nms]

        # Combine angular momentum for reaction wheels 1, 2, and 3
        plt.figure(figsize=(15, 5))
        plt.rcParams.update({"font.size": 16})
        plt.plot(self.t / (24 * 3600), h_array[0, :], label="RW␣1")
```

```python
        plt.plot(self.t / (24 * 3600), h_array[1, :], label="RW␣2")
        plt.plot(self.t / (24 * 3600), h_array[2, :], label="RW␣3")
        plt.plot(self.t / (24 * 3600), h_sat_1_2_3, color="red", linestyle="dashed", label="Saturation␣limit␣(+/-␣0.1␣Nms)")
        plt.plot(self.t / (24 * 3600), -h_sat_1_2_3, color="red", linestyle="dashed")
        plt.yscale('symlog', linthresh=1e-8)
        plt.xlabel("Time␣[days]", fontsize=16)
        plt.ylabel("$h_{rw}$␣[Nms]", fontsize=16)
        plt.xlim(left=0)
        plt.tick_params(axis='both', which='major', labelsize=16)
        plt.legend(fontsize=16, loc = 'upper␣left')
        plt.show()

        # Angular momentum for reaction wheel 4
        plt.figure(figsize=(15, 5))
        plt.rcParams.update({"font.size": 16})
        plt.plot(self.t / (24 * 3600), h_array[3, :], label="RW␣4")
        plt.plot(self.t / (24 * 3600), h_sat_4, color="red", linestyle="dashed", label="Saturation␣limit␣(+/-␣0.05␣Nms)")
        plt.plot(self.t / (24 * 3600), -h_sat_4, color="red", linestyle="dashed")
        plt.yscale('symlog', linthresh=1e-8)
        plt.xlabel("Time␣[days]", fontsize=16)
        plt.ylabel("$h_{rw}$␣[Nms]", fontsize=16)
        plt.xlim(left=0)
        plt.tick_params(axis='both', which='major', labelsize=16)
        plt.legend(fontsize=16)
        plt.show()


    def thrust_values(self, thruster_force_values):
        # Apply seaborn style
        plt.style.use('seaborn-v0_8')

        # Number of thrusters per graph
        thrusters_per_graph = 6
        num_thrusters = thruster_force_values.shape[0]
        num_graphs = (num_thrusters + thrusters_per_graph - 1) // thrusters_per_graph  # Calculate number of graphs

        # Loop through groups of thrusters
        for group_idx in range(num_graphs):
            start_idx = group_idx * thrusters_per_graph
            end_idx = min((group_idx + 1) * thrusters_per_graph, num_thrusters)

            # Plot combined graph for this group of thrusters
            plt.figure(figsize=(15, 5))
            for i in range(start_idx, end_idx):
                plt.plot(
                    self.t/ (24*3600),
                    thruster_force_values[i, :],
                    label=f"Thruster␣{i␣+␣1}"
                )
            plt.xlabel("Time␣[days]", fontsize=16)
            plt.ylabel("$F_{thrust}$␣[N]", fontsize=16)
            plt.xlim(left=0)
            plt.ylim(bottom=0)
            plt.tick_params(axis='both', which='major', labelsize=16)
            ax = plt.gca()  # Get the current axis
            ax.yaxis.get_offset_text().set_fontsize(16)  # Adjust the font size of the 1e-6
            plt.legend(fontsize=16, loc='upper␣left', ncol=2)
            plt.show()

    def thrust_values_difference(self, thruster_force_values):
        # Apply seaborn style
        plt.style.use('seaborn-v0_8')

        # Number of thrusters per graph
        thrusters_per_graph = 6
        num_thrusters = thruster_force_values.shape[0]
        num_graphs = (num_thrusters + thrusters_per_graph - 1) // thrusters_per_graph  # Calculate number of graphs

        # Loop through groups of thrusters
        for group_idx in range(num_graphs):
            start_idx = group_idx * thrusters_per_graph
            end_idx = min((group_idx + 1) * thrusters_per_graph, num_thrusters)

            # Plot combined graph for this group of thrusters
            plt.figure(figsize=(15, 5))
            for i in range(start_idx, end_idx):
                plt.plot(
                    self.t/ (24*3600),
                    thruster_force_values[i, :],
                    label=f"Thruster␣{i␣+␣1}"
                )
            plt.xlabel("Time␣[days]", fontsize=16)
            plt.ylabel("$\Delta␣F_{thrust}$␣[N]", fontsize=16)
            plt.tick_params(axis='both', which='major', labelsize=16)
            ax = plt.gca()  # Get the current axis
            ax.yaxis.get_offset_text().set_fontsize(16)  # Adjust the font size of the 1e-6
            plt.legend(fontsize=16, loc='upper␣left', ncol=2)
            plt.show()


    def impulse_versus_time(self, impulse):
        # Apply seaborn style
        plt.style.use('seaborn-v0_8')

        # Define custom behavior for 8-thruster setup
        num_thrusters = impulse.shape[0]
        line_styles = ['solid'] * 6 + ['dashed'] * (num_thrusters - 6)  # First 6 solid, rest
             dashed

        # Plot
        plt.figure(figsize=(15, 5))
        for i in range(num_thrusters):
            plt.plot(
                self.t / (24 * 3600),
                impulse[i, :],
                label=f"Thruster␣{i␣+␣1}",
                linestyle=line_styles[i % len(line_styles)]  # Use solid for first 6, dashed
                    for 7 and 8
            )

        # Labels and settings
        plt.xlabel("Time␣[days]", fontsize=16)
        plt.ylabel("$J_{thrust}$␣[Ns]", fontsize=16)
        plt.xlim(left=0)
        plt.ylim(bottom=0)
        plt.tick_params(axis='both', which='major', labelsize=16)

        # Adjust y-axis offset text font size
        ax = plt.gca()
        ax.yaxis.get_offset_text().set_fontsize(16)

        # Legend with multiple columns
        plt.legend(fontsize=16, loc='upper␣left', ncol=2)
        plt.show()

    def impulse_versus_time_2(self, impulse):
        # Apply seaborn style
        plt.style.use('seaborn-v0_8')

        # Total number of thrusters
        num_thrusters = impulse.shape[0]

        # Split into two groups
        thrusters_per_plot = 6
        num_plots = int(np.ceil(num_thrusters / thrusters_per_plot))

        for plot_idx in range(num_plots):
            start_idx = plot_idx * thrusters_per_plot
            end_idx = min((plot_idx + 1) * thrusters_per_plot, num_thrusters)

            # Plot each group of thrusters
            plt.figure(figsize=(15, 5))
            for i in range(start_idx, end_idx):
                plt.plot(
                    self.t / (24 * 3600),
                    impulse[i, :],
                    label=f"Thruster {i + 1}"  # Thruster numbering starts from 1
                )

            # Labels and settings
            plt.xlabel("Time [days]", fontsize=16)
            plt.ylabel("$J_{thrust}$ [Ns]", fontsize=16)
            plt.xlim(left=0)
            plt.ylim(bottom=0)
            plt.tick_params(axis='both', which='major', labelsize=16)

            # Adjust y-axis offset text font size
            ax = plt.gca()
            ax.yaxis.get_offset_text().set_fontsize(16)

            # Legend with multiple columns if needed
            plt.legend(fontsize=16, loc='upper left', ncol=2)
            plt.show()

    def half_cone_versus_time(self, cone_3, cone_4, cone_base_case):

        # Apply seaborn style
        plt.style.use('seaborn-v0_8')
        requirement = np.ones(len(cone_3)) * 0.18

        # Plot combined graph for all configurations
        plt.figure(figsize=(15, 5))  # Set combined graph size
        plt.plot(self.t / (24 * 3600), cone_3, label="STF Conf. 3", linestyle="-")
        plt.plot(self.t / (24 * 3600), cone_4, label="STF Conf. 4", linestyle="--")
        plt.plot(self.t / (24 * 3600), cone_base_case, label="Base case", linestyle="-.")
        plt.plot(self.t / (24 * 3600), requirement, label="ADCS-01 requirement", color="red",
             linestyle="dashed")

        # Add labels and customize ticks
        plt.xlabel("Time [days]", fontsize=16)
        plt.ylabel("$\\beta$ [$\\degree$]", fontsize=16)
        plt.xlim(left=0)
        plt.ylim(bottom=0)
        plt.tick_params(axis='both', which='major', labelsize=16)
        ax = plt.gca()  # Get the current axis
        ax.yaxis.get_offset_text().set_fontsize(16)  # Adjust the font size of the offset
            text

        # Add legend
        plt.legend(fontsize=16)

        # Show plot
        plt.show()

        # Plot combined graph for all configurations
        plt.figure(figsize=(15, 5))  # Set combined graph size
        plt.plot(self.t / (24 * 3600), cone_3, label="STF Conf. 3", linestyle="-")
        plt.plot(self.t / (24 * 3600), cone_4, label="STF Conf. 4", linestyle="--")
        plt.plot(self.t / (24 * 3600), cone_base_case, label="Base case", linestyle="-.")

        # Add labels and customize ticks
        plt.xlabel("Time [days]", fontsize=16)
        plt.ylabel("$\\beta$ [$\\degree$]", fontsize=16)
        plt.xlim(left=0)
        plt.ylim(bottom=0)
        plt.tick_params(axis='both', which='major', labelsize=16)
        ax = plt.gca()  # Get the current axis
        ax.yaxis.get_offset_text().set_fontsize(16)  # Adjust the font size of the offset
            text

        # Add legend
        plt.legend(fontsize=16)

        # Show plot
        plt.show()


def simulation(
    data_file_CAPSTONE,
    data_file_Sun,
    omega_0,
    kp,
    kd,
    ks,
    T_matrix,
    matrixnumber
):
    # Load data files
    data_CAPSTONE = np.loadtxt(data_file_CAPSTONE)
    data_Sun = np.loadtxt(data_file_Sun)

    # Load data files
    data_CAPSTONE = np.loadtxt(data_file_CAPSTONE)
    data_Sun = np.loadtxt(data_file_Sun)

    # Extract metadata from data_file_CAPSTONE filename using regular expressions
    # Looks for date and time in the format YYYY-MM-DD HH:MM
    filename = os.path.basename(data_file_CAPSTONE)
    datetime_matches = re.findall(r'\d{4}-\d{2}-\d{2}␣\d{2}:\d{2}', filename)

    if len(datetime_matches) >= 2:
        startdate = datetime_matches[0].replace(":", "-").replace("␣", "_")
        enddate = datetime_matches[1].replace(":", "-").replace("␣", "_")
    else:
        raise ValueError("Could␣not␣extract␣start␣and␣end␣date-times␣from␣the␣filename.")

    # Define save directory
    save_dir = "/Users/Pieter/Library/Mobile␣Documents/com~apple~CloudDocs/Thesis/Research␣
        Phase/lunar_CubeSat/results/data"
    os.makedirs(save_dir, exist_ok=True)

    # Create time array based on the input data files and check if they cover the same epochs
    time_array = data_CAPSTONE[:, 0]
    time_array_check = data_Sun[:, 0]
    if time_array.all() != time_array_check.all():
        raise ValueError(
            "Epochs␣of␣the␣data␣files␣are␣not␣the␣same;␣check␣ephemeris␣retrieval."
        )

    # Obtain position data over time
    position_CAPSTONE = data_CAPSTONE[:, 1:4]
    position_Sun = data_Sun[:, 1:4]

    # Retrieve time step from data file
    dt = time_array[1] - time_array[0]

    vis = Visualization(time_array)
    const = Constants()
    rot = Rotation()
    pd = PID(const.inertia_matrix)
    dist = DisturbanceTorques(const.inertia_matrix)

    # Initialize individual vectors for iteration
    # q = quaternion_0_vector # For custom initial condition
    q = pd.reference_quaternion_paper(position_CAPSTONE[0,:], position_Sun[0,:]) # For
        standard initial position
    omega = omega_0

    # Dummy values for PID
    factor_p = kp
    factor_i = 0
    factor_d = kd
    factor_s = ks

    # Create arrays
    k_p = np.array([1, 1, 1]) * factor_p
    k_i = np.array([1, 1, 1]) * factor_i
    k_d = np.array([1, 1, 1]) * factor_d
    k_s = np.array([1, 1, 1]) * factor_s


    # Initialize storage arrays for visualization
    q_ref_vis = np.empty((4, len(time_array)))
    q_vis = np.empty((4, len(time_array)))
    T_GG_vis = np.empty((3, len(time_array)))
    T_SRP_vis = np.empty((3, len(time_array)))
    T_d_vis = np.empty((3, len(time_array)))
    T_c_vis = np.empty((3, len(time_array)))
    T_RW_vis = np.empty((3, len(time_array)))
    P_RW_vis = np.empty((3, len(time_array)))
    E_RW_vis = np.empty((3, len(time_array)))
    omega_vis = np.empty((3, len(time_array)))
    h_vis = np.empty((3, len(time_array)))
    thrust_vis = np.empty((len(T_matrix[0]), len(time_array)))
    impulse_vis = np.empty((len(T_matrix[0]), len(time_array)))
    P_thrust_vis = np.empty((len(T_matrix[0]), len(time_array)))
    E_thrust_vis = np.empty((len(T_matrix[0]), len(time_array)))
    half_cone_offset_vis = np.empty((len(time_array)))

    # Index count and previous q_ref value
    index = 0
    previous_q_ref = None

    # Initiate angular momentum, reaction wheel energy, thrust energy, impulse
    h = 0
    E_RW = 0
    E_thrust = 0
    impulse = np.zeros(len(T_matrix[0]))

    for t in time_array:

        # LUMIO paper-based reference quaternion update
        q_ref = pd.reference_quaternion_paper(position_CAPSTONE[index,:], position_Sun[index
            ,:])

        # Check quaternion continuity; do we need to flip signs?
        if previous_q_ref is not None:
            if np.dot(q_ref, previous_q_ref) < 0:
                q_ref = -q_ref

        # Store q_ref for next iteration
        previous_q_ref = q_ref

        # Half-cone requirement validation
        y_panel_body = np.array([0,-1,0])
        y_panel_inert = np.linalg.inv(rot.quaternion_321_rotation(q[0], q[1], q[2], q[3])) @
            y_panel_body
        y_panel_inert /= np.linalg.norm(y_panel_inert)
        Moon_pointing_vector = - position_CAPSTONE[index,:]
        Moon_pointing_vector /= np.linalg.norm(Moon_pointing_vector)
        half_cone_offset = np.rad2deg(np.arccos(np.clip(np.dot(y_panel_inert,
            Moon_pointing_vector), -1.0, 1.0)))

        # Calculate real-time disturbance torques
        T_GG = dist.GGMoon(q, position_CAPSTONE[index, :])
        T_SRP = dist.SRP_deployed(q, position_Sun[index, :], position_CAPSTONE[index, :])
        # Dummy input zeros
        T_d = T_GG + T_SRP

        T_c = pd.control_torque(q, q_ref, omega, k_p, k_i, k_d, k_s, dt)

        ######### RW ANALYSIS #######################
        # Calculate incremental increase in angular momentum of the momentum wheel
        h = const.inertia_matrix_undeployed @ omega
        # Assembly configuration matrix, each reaction wheel covers one axis
        A_RW = np.array([[1,0,0], [0,1,0], [0,0,1]])
        # Torque imposed on RW assembly, nominal mode
        T_ass = np.linalg.pinv(A_RW) @ (-T_c + np.cross(A_RW @ h, omega))
        # Torque generated by the RW set, total
        T_RW = - (A_RW @ T_ass + np.cross(omega, A_RW @ h))
        # Power consumption at this time step, linear relation assumed
        P_RW = const.P_peak_max / const.T_RW_max * np.abs(T_RW)
        # Total energy consumption, power integrated over time
        E_RW += P_RW * dt
        #############################################

        ######### THRUSTER ANALYSIS #################
        # Linear Programming Solution
        thrust = cp.Variable(len(T_matrix[0]))
        constraints = [T_matrix @ thrust == T_c, thrust >= 0, thrust <= const.F_SSP_max]
        objective = cp.Minimize(cp.sum(thrust))
        problem = cp.Problem(objective, constraints)
        problem.solve()

        # # Check the solution status
        # if problem.status == cp.OPTIMAL:
        #     print("Solution found!")
        #     thrust_value = thrust.value
        #     print("Thrust values:", thrust_value)
        # elif problem.status == cp.INFEASIBLE:
        #     print("No solution exists (problem is infeasible).")
        # elif problem.status == cp.UNBOUNDED:
        #     print("Problem is unbounded (no finite solution).")
        # else:
        #     print(f"Solver status: {problem.status}")

        if thrust.value is None:
            thrust_value = np.zeros(len(T_matrix[0]))
        else:
            thrust_value = np.clip(thrust.value, 0, None)  # Set all values < 0 to 0

        # Add uncertainty
        sigma = 0.05  # 5% uncertainty introduced, random value
        thrust_value_actual = thrust_value.copy()  # Start with the original values
        non_zero_indices = thrust_value > 0  # Find indices where thrust value is positive

        # Apply noise only to non-zero values
        thrust_value_actual[non_zero_indices] = np.clip(
            np.random.normal(
                loc=thrust_value[non_zero_indices],
                scale=sigma * thrust_value[non_zero_indices]
            ),
            (1 - sigma) * thrust_value[non_zero_indices],
            (1 + sigma) * thrust_value[non_zero_indices]
        )

        # Update impulse value
        impulse += thrust_value_actual * dt

        # Power analysis
        # From data sheet: Input Power linear relation with thrust output, from T = 0 to T =
            200 [muN] and P = 0 to 20 [W]
        # Let power input be as in original thrust case, since uncertainty comes from the
            output only
        P_thrust = const.P_SSP_max / const.F_SSP_max * thrust_value
        E_thrust += P_thrust * dt
        #############################################

        # Append to visualisation arrays
        q_ref_vis[:, index] = q_ref
        q_vis[:, index] = q
        T_d_vis[:, index] = T_d
        T_GG_vis[:, index] = T_GG
        T_SRP_vis[:, index] = T_SRP
        T_c_vis[:, index] = T_c
        T_RW_vis[:, index] = T_RW
        P_RW_vis[:, index] = P_RW
        E_RW_vis[:, index] = E_RW
        omega_vis[:, index] = omega
        h_vis[:, index] = h
        half_cone_offset_vis[index] = half_cone_offset
        thrust_vis[:,index] = thrust_value_actual
        impulse_vis[:,index] = impulse
        P_thrust_vis[:,index] = P_thrust
        E_thrust_vis[:,index] = E_thrust

        # Update to t = 1
        # Integrate omega and quaternion
        omega_new = pd.rk4_integrator(pd.derivative_omega, omega, dt, T_d, T_c)
        q_new = pd.rk4_integrator(pd.derivative_quaternion, q, dt, omega)
        q = q_new / np.linalg.norm(q_new)
        omega = omega_new

        # Update index
        index += 1
        print(index)

    # At the end of the simulation loop, after populating the visualization arrays
    total_PE = np.sum(P_thrust_vis, axis=0)  # Summing up all thrusters' power over time
    total_E = np.sum(E_thrust_vis[:, -1])  # Total energy is the last value of summation

    # Save each "vis" array
    vis_arrays = {
        "q_ref_vis": q_ref_vis,
        "q_vis": q_vis,
        "T_GG_vis": T_GG_vis,
        "T_SRP_vis": T_SRP_vis,
        "T_d_vis": T_d_vis,
        "T_c_vis": T_c_vis,
        "T_RW_vis": T_RW_vis,
        "P_RW_vis": P_RW_vis,
        "E_RW_vis": E_RW_vis,
        "omega_vis": omega_vis,
        "h_vis": h_vis,
        "thrust_vis": thrust_vis,
        "impulse": impulse_vis,
        "P_thrust_vis": P_thrust_vis,
        "E_thrust_vis": E_thrust_vis,
        "half_cone_offset_vis": half_cone_offset_vis,
        "time_array": time_array
    }

    for var_name, data in vis_arrays.items():
        file_name = f"{startdate}_{enddate}_{var_name}_{matrixnumber}.dat"
        file_path = os.path.join(save_dir, file_name)
        np.savetxt(file_path, data.T, delimiter="␣", fmt="%.20f")  # Transpose data for
            column format

    # Plot generation
    vis.quaternion_versus_time(q_ref_vis, q_vis)
    # vis.euler_versus_time(q_ref_vis, q_vis)
    # vis.control_torque_versus_time(T_c_vis)
    # vis.RW_PE_versus_time(P_RW_vis, "P")
    # vis.RW_PE_versus_time(E_RW_vis, "E")
    # vis.h_versus_time(h_vis)
    # vis.disturbance_torque_versus_time(T_d_vis, T_GG_vis, T_SRP_vis)
    # vis.omega_versus_time(omega_vis)
    # vis.plot_trajectories(data_file_CAPSTONE)
    vis.thruster_PE_vs_time(P_thrust_vis, "P")
    vis.thruster_PE_vs_time(E_thrust_vis, "E")
    # vis.combined_PE_vs_time(P_RW_vis, P_thrust_vis, "P")
    # vis.combined_PE_vs_time(E_RW_vis, E_thrust_vis, "E")
    vis.thrust_values(thrust_vis)
```

```python

    # Return required values
    return {
        "max_total_power": np.max(total_PE),
        "max_total_energy": total_E
    }

############## INPUT ##################################

ephemeris_time_step = 1/60 # hours
begin_date = "2023-01-01␣00:00"
end_date = "2023-01-01␣00:05"
control_time_step = 0.001 # seconds

#######################################################

eph_CAPSTONE = EphemerisData(
    Constants.id_CAPSTONE,
    Constants.location_Moon_centre,
    ephemeris_time_step,
    begin_date,
    end_date,
)
eph_CAPSTONE.convert_data_interpolated(eph_CAPSTONE.vectors()[0], control_time_step)

eph_Sun = EphemerisData(
    Constants.id_Sun,
    Constants.location_Moon_centre,
    ephemeris_time_step,
    begin_date,
    end_date,
)
eph_Sun.convert_data_interpolated(eph_Sun.vectors()[0], control_time_step)

# Set-up 1
tau_11 = np.cross(np.array([0.1, 0.15, 0]), np.array([-1,0,0]))
tau_12 = np.cross(np.array([-0.1, 0.15, 0]), np.array([1,0,0]))
tau_13 = np.cross(np.array([0, 0.15, 0.1]), np.array([0,0,-1]))
tau_14 = np.cross(np.array([0, -0.15, 0.1]), np.array([0,0,-1]))
tau_15 = np.cross(np.array([0.1, 0, 0.1]), np.array([-1,0,0]))
tau_16 = np.cross(np.array([-0.1, 0, 0.1]), np.array([1,0,0]))

# Set-up 2
tau_21 = np.cross(np.array([0.1, 0.15, 0.1]), np.array([-1,0,0]))
tau_22 = np.cross(np.array([-0.1, 0.15, 0.1]), np.array([1,0,0]))
tau_23 = np.cross(np.array([0, 0.15, 0.1]), np.array([0,0,-1]))
tau_24 = np.cross(np.array([0, -0.15, 0.1]), np.array([0,0,-1]))
tau_25 = np.cross(np.array([0.1, -0.15, 0.1]), np.array([-1,0,0]))
tau_26 = np.cross(np.array([-0.1, -0.15, 0.1]), np.array([1,0,0]))

# Set-up 3
tau_31 = np.cross(np.array([0.1, 0.15, 0]), np.array([-1,0,0]))
tau_32 = np.cross(np.array([-0.1, 0.15, 0]), np.array([1,0,0]))
tau_33 = np.cross(np.array([0, 0.15, 0.1]), np.array([0,0,-1]))
tau_34 = np.cross(np.array([0, -0.15, 0.1]), np.array([0,0,-1]))
tau_35 = np.cross(np.array([0.1, 0, 0.1]), np.array([-1,0,0]))
tau_36 = np.cross(np.array([-0.1, 0, 0.1]), np.array([1,0,0]))
tau_37 = np.cross(np.array([0.1, -0.15, -0.1]), np.array([0,1,0]))
tau_38 = np.cross(np.array([-0.1, -0.15, -0.1]), np.array([0,1,0]))

# Set-up 4
tau_41 = np.cross(np.array([0.1, -0.15, 0.1]), np.array([0,1,0]))
tau_42 = np.cross(np.array([0.1, -0.15, 0.1]), np.array([-1,0,0]))
tau_43 = np.cross(np.array([0.1, -0.15, 0.1]), np.array([0,0,-1]))
tau_44 = np.cross(np.array([-0.1, -0.15, 0.1]), np.array([1,0,0]))
tau_45 = np.cross(np.array([-0.1, -0.15, 0.1]), np.array([0,1,0]))
tau_46 = np.cross(np.array([-0.1, -0.15, 0.1]), np.array([0,0,-1]))
tau_47 = np.cross(np.array([0.1, -0.15, -0.1]), np.array([-1,0,0]))
tau_48 = np.cross(np.array([0.1, -0.15, -0.1]), np.array([0,1,0]))
tau_49 = np.cross(np.array([0.1, -0.15, -0.1]), np.array([0,0,1]))
tau_410 = np.cross(np.array([-0.1, -0.15, -0.1]), np.array([0,0,1]))
```

```python
tau_411 = np.cross(np.array([-0.1, -0.15, -0.1]), np.array([0,1,0]))
tau_412 = np.cross(np.array([-0.1, -0.15, -0.1]), np.array([1,0,0]))

# Stack the tau vectors into a 3x6 matrix
T_matrix_1 = np.column_stack((tau_11, tau_12, tau_13, tau_14, tau_15, tau_16))
T_matrix_2 = np.column_stack([tau_21, tau_22, tau_23, tau_24, tau_25, tau_26])
T_matrix_3 = np.column_stack([tau_31, tau_32, tau_33, tau_34,tau_35, tau_36, tau_37, tau_38])
T_matrix_4 = np.column_stack([tau_41, tau_42, tau_43, tau_44,tau_45, tau_46, tau_47, tau_48,
    tau_49, tau_410, tau_411, tau_412])
```

C

C Source Code

This appendix consists of tho main parts: first, the `main.c` file will be shown, in which the entire Nucleo board set-up is presented and the main functions are called. Then, `func.c` will be shown, that includes all the underlying functions used throughout the attitude control computations.

C.1. main.c

```c

/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2024 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "func.h"
#include "global.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
TIM_HandleTypeDef htim1;
TIM_HandleTypeDef htim2;

UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */
UART_HandleTypeDef huart2;
uint8_t rxByte[1] = {0};
uint8_t rxBuff[500] = {0};
int rxCounter = 0;

// Define a static previous_q_ref array to store the last reference quaternion
static double previous_q_ref[4] = {1.0, 0.0, 0.0, 0.0};  // Initialize with a default
    quaternion (identity)
double pos_CAP[3] = {0};
double pos_Sun[3] = {0};
double q[4], q_ref[4], omega[3], k_p[3], k_i[3], k_d[3], k_s[3], dt;
double T_c[3];  // Initialize these in main
double thrust_array[6];
/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_TIM1_Init(void);
static void MX_TIM2_Init(void);
/* USER CODE BEGIN PFP */
/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* HAL_UART_RxCpltCallback:
 *
 * Standard usage, leave the buffer parsing as is.
 * In parseBuffer parsing definition in func.c, adjust for the input parameters that are
     being taken.
 * PD: q, q_ref, omega, k_p, k_i, k_d, k_s, dt
 * PD & q_ref: pos_CAP, pos_Sun, q, omega, k_p, k_i, k_d, k_s
 *
 * Leave PD_control for all.
 * Optional sending of different parameters possible, 3 or 4 array (sendQuaternion definition
     )
 */

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
    if (rxByte[0] != '\n') {
        rxBuff[rxCounter] = rxByte[0];
        rxCounter++;
    } else {
        rxBuff[rxCounter] = '\0';  // Ensure null-terminated string
        parseBuffer(rxBuff, rxCounter);  // Parse the buffer

        // Reset buffer
        memset(rxBuff, 0, sizeof(rxBuff));  // Clear the buffer after parsing
        rxCounter = 0;  // Reset counter

        // Update the reference quaternion based on latest positions
        reference_quaternion_paper(); // Call to calculate q_ref

        send4Vector(&huart2, q_ref, 4);

        // Check quaternion continuity: flip q_ref if dot product with previous_q_ref is
            negative
        double dot_product = quaternion_dot_product(q_ref, previous_q_ref);
        if (dot_product < 0) {
            flip_quaternion(q_ref);
        }

        // Store q_ref for the next iteration
        memcpy(previous_q_ref, q_ref, 4 * sizeof(double));

        // Calculate control torques with the updated q_ref and other values taken from
            Python.
        PD_control(q, q_ref, omega, k_p, k_i, k_d, k_s, dt, T_c); // Calculate the control
            torques

        // Send control torque values back to Python
        send3Vector(&huart2, T_c, 3); // Send the control torques back to Python

        // Define number of thrusters based on thruster matrix
        idxint n_thrusters = 6;

        // Solve problem
        solve_thruster_problem(T_matrix_1, n_thrusters, T_c, F_SSP_max);

        // Test PWM adjustment
//        double test_thrust = thrust_array[0];
        updatePWMFrequency(&htim1, 1,  thrust_array[0]);
        updatePWMFrequency(&htim1, 2,  thrust_array[1]);
        updatePWMFrequency(&htim1, 3,  thrust_array[2]);
        updatePWMFrequency(&htim1, 4,  thrust_array[3]);
        updatePWMFrequency(&htim2, 1,  thrust_array[4]);
        updatePWMFrequency(&htim2, 2,  thrust_array[5]);

        // Send thruster values back to Python
        send6Vector(&huart2, thrust_array, n_thrusters);

    }

    // Reset rxByte[0] to 0 to ensure it's ready for the next byte
    rxByte[0] = 0;


    HAL_UART_Receive_IT(&huart2, rxByte, 1);  // Continue to receive the next byte
}

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{

  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();


  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();
  MX_TIM1_Init();
  MX_TIM2_Init();
  /* USER CODE BEGIN 2 */

  HAL_UART_Receive_IT(&huart2, rxByte, 1);
  HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
  HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_2);
  HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_3);
  HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_4);
  HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
  HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_2);
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {

//        HAL_GPIO_TogglePin(GREEN_LED_GPIO_Port, GREEN_LED_Pin);
//        HAL_Delay(1000);
        }


    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
  RCC_PeriphCLKInitTypeDef PeriphClkInit = {0};

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_0) != HAL_OK)
  {
    Error_Handler();
  }
  PeriphClkInit.PeriphClockSelection = RCC_PERIPHCLK_USART2|RCC_PERIPHCLK_TIM1
                              |RCC_PERIPHCLK_TIM2;
  PeriphClkInit.Usart2ClockSelection = RCC_USART2CLKSOURCE_PCLK1;
  PeriphClkInit.Tim1ClockSelection = RCC_TIM1CLK_HCLK;
  PeriphClkInit.Tim2ClockSelection = RCC_TIM2CLK_HCLK;
  if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInit) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief TIM1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM1_Init(void)
{

  /* USER CODE BEGIN TIM1_Init 0 */

  /* USER CODE END TIM1_Init 0 */

  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_OC_InitTypeDef sConfigOC = {0};
  TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig = {0};

  /* USER CODE BEGIN TIM1_Init 1 */

  /* USER CODE END TIM1_Init 1 */
  htim1.Instance = TIM1;
  htim1.Init.Prescaler = 8-1;
  htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim1.Init.Period = 1000-1;
  htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim1.Init.RepetitionCounter = 0;
  htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_PWM_Init(&htim1) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterOutputTrigger2 = TIM_TRGO2_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 250;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCNPolarity = TIM_OCNPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
  sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
  if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }
  sConfigOC.Pulse = 0;
  if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_2) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_3) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_4) != HAL_OK)
  {
    Error_Handler();
  }
  sBreakDeadTimeConfig.OffStateRunMode = TIM_OSSR_DISABLE;
  sBreakDeadTimeConfig.OffStateIDLEMode = TIM_OSSI_DISABLE;
  sBreakDeadTimeConfig.LockLevel = TIM_LOCKLEVEL_OFF;
  sBreakDeadTimeConfig.DeadTime = 0;
  sBreakDeadTimeConfig.BreakState = TIM_BREAK_DISABLE;
  sBreakDeadTimeConfig.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
  sBreakDeadTimeConfig.BreakFilter = 0;
  sBreakDeadTimeConfig.Break2State = TIM_BREAK2_DISABLE;
  sBreakDeadTimeConfig.Break2Polarity = TIM_BREAK2POLARITY_HIGH;
  sBreakDeadTimeConfig.Break2Filter = 0;
  sBreakDeadTimeConfig.AutomaticOutput = TIM_AUTOMATICOUTPUT_DISABLE;
  if (HAL_TIMEx_ConfigBreakDeadTime(&htim1, &sBreakDeadTimeConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM1_Init 2 */

  /* USER CODE END TIM1_Init 2 */
  HAL_TIM_MspPostInit(&htim1);

}

/**
  * @brief TIM2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM2_Init(void)
{

  /* USER CODE BEGIN TIM2_Init 0 */

  /* USER CODE END TIM2_Init 0 */

  TIM_MasterConfigTypeDef sMasterConfig = {0};
  TIM_OC_InitTypeDef sConfigOC = {0};

  /* USER CODE BEGIN TIM2_Init 1 */

  /* USER CODE END TIM2_Init 1 */
  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 8-1;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 1000-1;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sConfigOC.OCMode = TIM_OCMODE_PWM1;
  sConfigOC.Pulse = 250;
  sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
  sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
  if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM2_Init 2 */

  /* USER CODE END TIM2_Init 2 */
  HAL_TIM_MspPostInit(&htim2);

}

/**
  * @brief USART2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART2_UART_Init(void)
{

  /* USER CODE BEGIN USART2_Init 0 */

  /* USER CODE END USART2_Init 0 */

  /* USER CODE BEGIN USART2_Init 1 */

  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_EVEN;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */

  /* USER CODE END USART2_Init 2 */

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
/* USER CODE BEGIN MX_GPIO_Init_1 */
/* USER CODE END MX_GPIO_Init_1 */

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GREEN_LED_GPIO_Port, GREEN_LED_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin : GREEN_LED_Pin */
  GPIO_InitStruct.Pin = GREEN_LED_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GREEN_LED_GPIO_Port, &GPIO_InitStruct);

/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

C.2. func.c

```
    #include "func.h"
#include <math.h>
#include <string.h>
#include "main.h"
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <global.h>
#include "ecos.h"


// Constants
const double F_SSP_max = 0.0000008; //N, test 200microNewtons of thrust from SSP thrusters
const uint32_t timer_period = 100000; //microseconds, frequency = 1 / (timer_period /
    1,000,000)
const double minimum_impulse_bit = 0.01; // microNewtonseconds

// note: matrices do not adjust correctly for difference in panel distance. Re-do all
    matrices in case of different set-up.
const double T_matrix_1[3][6] = {
                {0.0, 0.0, -0.15, 0.15, 0.0, 0.0},
                {0.0, 0.0, 0.0, 0.0, -0.1, 0.1},
                {0.15, -0.15, 0.0, 0.0, 0.0, 0.0}
};

const double T_matrix_2[3][6] = {
                {0.0, 0.0, -0.15, 0.15, 0.0, 0.0},
                {-0.1, 0.1, 0.0, 0.0, -0.1, 0.1},
                {0.15, -0.15, 0.0, 0.0, -0.15, 0.15}
};

const double T_matrix_3[3][8] = {
                {0.0, 0.0, -0.15, 0.15, 0.0, 0.0, 0.1, 0.1},
                {0.0, 0.0, 0.0, 0.0, -0.1, 0.1, 0.0, 0.0},
                {0.15, -0.15, 0.0, 0.0, 0.0, 0.0, 0.1, -0.1}
};

const double T_matrix_4[3][12] = {
                {-0.1, 0.0, 0.15, 0.0, -0.1, 0.15, 0.0, 0.1, -0.15, -0.15, 0.1, 0.0},
                {0.0, -0.1, 0.1, 0.1, 0.0, -0.1, 0.1, 0.0, -0.1, 0.1, 0.0, -0.1},
                {0.1, -0.15, 0.0, 0.15, -0.1, 0.0, -0.15, 0.1, 0.0, 0.0, -0.1, 0.15}
```

```c
};

const double inertia_matrix[3][3] = {
    {1.009, 0.0, 0.0},
    {0.0, 0.251, 0.0},
    {0.0, 0.0, 0.916}
};

// Retrieve buffer from incoming signal
void parseBuffer(uint8_t *buffer, uint16_t size) {
    char *token;

    // Convert buffer to null-terminated string
    buffer[size] = '\0'; // Ensure there is a null at the end of the buffer

    token = strtok((char *)buffer, ",");
    if (!token) {
        HAL_UART_Transmit(&huart2, (uint8_t*)"Parse␣Error\n", 12, HAL_MAX_DELAY);
        return;
    }

    // Parsing arrays from the buffer
    for (int i = 0; i < 3; i++, token = strtok(NULL, ",")) pos_CAP[i] = token ? atof(token): 0;
    for (int i = 0; i < 3; i++, token = strtok(NULL, ",")) pos_Sun[i] = token ? atof(token): 0;
    for (int i = 0; i < 4; i++, token = strtok(NULL, ",")) q[i] = token ? atof(token) : 0;
    for (int i = 0; i < 3; i++, token = strtok(NULL, ",")) omega[i] = token ? atof(token) : 0;
    for (int i = 0; i < 3; i++, token = strtok(NULL, ",")) k_p[i] = token ? atof(token) : 0;
    for (int i = 0; i < 3; i++, token = strtok(NULL, ",")) k_i[i] = token ? atof(token) : 0;
    for (int i = 0; i < 3; i++, token = strtok(NULL, ",")) k_d[i] = token ? atof(token) : 0;
    for (int i = 0; i < 3; i++, token = strtok(NULL, ",")) k_s[i] = token ? atof(token) : 0;
    token = strtok(NULL, ",");
    dt = token ? atof(token) : 0;


}


// Matrix multiplication standard
void matrix_multiply_vector(const Matrix4x4* mat, const Quaternion* vec, Quaternion* result) {
    // Adjusting order of assignments to match [qew, qe1, qe2, qe3]
    result->w = mat->m[3][0] * vec->w + mat->m[3][1] * vec->x + mat->m[3][2] * vec->y + mat->m[3][3] * vec->z;
    result->x = mat->m[0][0] * vec->w + mat->m[0][1] * vec->x + mat->m[0][2] * vec->y + mat->m[0][3] * vec->z;
    result->y = mat->m[1][0] * vec->w + mat->m[1][1] * vec->x + mat->m[1][2] * vec->y + mat->m[1][3] * vec->z;
    result->z = mat->m[2][0] * vec->w + mat->m[2][1] * vec->x + mat->m[2][2] * vec->y + mat->m[2][3] * vec->z;
}

// Quaternion error calculation function
void quaternion_error(const Quaternion* q, const Quaternion* q_ref, Quaternion* result) {
    // Extract quaternion components
    double qrw = q_ref->w, qr1 = q_ref->x, qr2 = q_ref->y, qr3 = q_ref->z;
    double qw = q->w, q1 = q->x, q2 = q->y, q3 = q->z;

    // Adjusted quaternion vector
    Quaternion adj_vec = {-q1, -q2, -q3, qw};

    // Construct the matrix
    Matrix4x4 adj_mat = {
        {
            {qrw, qr3, -qr2, qr1},
            {-qr3, qrw, qr1, qr2},
            {qr2, -qr1, qrw, qr3},
            {-qr1, -qr2, qr3, qrw}
        }
    };

    // Perform the matrix-vector multiplication
    matrix_multiply_vector(&adj_mat, &adj_vec, result);
}


// Function to convert float array to string and send via UART
void send3Vector(UART_HandleTypeDef *huart, const double *vector, uint8_t num_elements) {
    uint8_t buffer[200];  // Adjust size based on the expected length of the message

    sprintf((char*) buffer, "%.20lf,%.20lf,%.20lf\n", vector[0], vector[1], vector[2]);

    // Send the formatted string via UART
    HAL_UART_Transmit(huart, buffer, strlen((char*)buffer), HAL_MAX_DELAY);
}

// Function to convert float array to string and send via UART, for reference quaternion
void send4Vector(UART_HandleTypeDef *huart, const double *vector, uint8_t num_elements) {
    uint8_t buffer[200];  // Adjust size based on the expected length of the message

    sprintf((char*) buffer, "%.20lf,%.20lf,%.20lf,%.20lf\n", vector[0], vector[1], vector[2],
         vector[3]);

    // Send the formatted string via UART
    HAL_UART_Transmit(huart, buffer, strlen((char*)buffer), HAL_MAX_DELAY);
}

// Function to convert float array to string and send via UART, for reference quaternion
void send6Vector(UART_HandleTypeDef *huart, const double *vector, uint8_t num_elements) {
    uint8_t buffer[300];  // Adjust size based on the expected length of the message

    sprintf((char*) buffer, "%.20lf,%.20lf,%.20lf,%.20lf,%.20lf,%.20lf\n", vector[0], vector
        [1], vector[2], vector[3], vector[4], vector[5]);

    // Send the formatted string via UART
    HAL_UART_Transmit(huart, buffer, strlen((char*)buffer), HAL_MAX_DELAY);
}



// PD control loop, adapted for C
void PD_control(double *q, double *q_ref, double *omega, double *k_p, double *k_i, double *
    k_d, double *k_s, double dt, double *T_c) {
    Quaternion current_q = {q[0], q[1], q[2], q[3]};
    Quaternion reference_q = {q_ref[0], q_ref[1], q_ref[2], q_ref[3]};
    Quaternion error_q;

    // Calculate quaternion error
    quaternion_error(&current_q, &reference_q, &error_q);

    // Destructure error quaternion components
    double qew = error_q.w;
    double qex = error_q.x;
    double qey = error_q.y;
    double qez = error_q.z;

    // Angular velocity components
    double omega_x = omega[0], omega_y = omega[1], omega_z = omega[2];

    // PID gains
    double k_p_1 = k_p[0], k_p_2 = k_p[1], k_p_3 = k_p[2];
    double k_i_1 = k_i[0], k_i_2 = k_i[1], k_i_3 = k_i[2];
    double k_d_1 = k_d[0], k_d_2 = k_d[1], k_d_3 = k_d[2];
    double k_s_1 = k_s[0], k_s_2 = k_s[1], k_s_3 = k_s[2];

    // Assuming integral terms should be persistent or managed outside this function
    static double int_x = 0, int_y = 0, int_z = 0;  // Integral terms

    // Control torque calculations
    T_c[0] = k_s_1 * (k_p_1 * qex + k_i_1 * int_x - k_d_1 * omega_x);
    T_c[1] = k_s_2 * (k_p_2 * qey + k_i_2 * int_y - k_d_2 * omega_y);
    T_c[2] = k_s_3 * (k_p_3 * qez + k_i_3 * int_z - k_d_3 * omega_z);

    // Update integral terms for next iteration
    int_x += qex * dt;
    int_y += qey * dt;
    int_z += qez * dt;
}

// Helper function to compute the cross product of two 3D vectors
void cross_product(const double *v1, const double *v2, double *result) {
    result[0] = v1[1] * v2[2] - v1[2] * v2[1];
    result[1] = v1[2] * v2[0] - v1[0] * v2[2];
    result[2] = v1[0] * v2[1] - v1[1] * v2[0];
}

// Helper function to normalize a 3D vector
void normalize_vector(double *v) {
    double norm = sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
    for (int i = 0; i < 3; i++) {
        v[i] /= norm;
    }
}

void reference_quaternion_paper(void) {
        double Sun_pointing_vector[3];
        double Moon_pointing_vector[3];
        double cross_1[3], cross_2[3];
        double x1[3], x2[3], x3[3];
        double A_d[3][3];


        // Calculate Sun pointing vector
        for (int i = 0; i < 3; i++) {
                Sun_pointing_vector[i] = pos_Sun[i] - pos_CAP[i];
        }

        normalize_vector(Sun_pointing_vector);

    // Calculate Moon pointing vector
    for (int i = 0; i < 3; i++) {
        Moon_pointing_vector[i] = -pos_CAP[i];
    }

    normalize_vector(Moon_pointing_vector);

        // x1 = Moon pointing vector
        memcpy(x1, Moon_pointing_vector, 3 * sizeof(double));

        // x2 = cross product of Sun_pointing vector and x1, normalized
        cross_product(Sun_pointing_vector, x1, cross_1);
        normalize_vector(cross_1);
        memcpy(x2, cross_1, 3 * sizeof(double));

        // x3 = cross product of x1 and x2, normalized
        cross_product(x1, x2, cross_2);
        normalize_vector(cross_2);
        memcpy(x3, cross_2, 3 * sizeof(double));

        // Construct DCM matrix A_d to convert to quaternion representation
        for (int i = 0; i < 3; i++) {
                A_d[0][i] = x1[i];
                A_d[1][i] = x2[i];
                A_d[2][i] = x3[i];

        }

        // Final DCM to quaternion conversion
        DCM_to_quaternion(A_d);

}
```

```c

void DCM_to_quaternion(double DCM[3][3]) {
    double a11 = DCM[0][0];
    double a22 = DCM[1][1];
    double a33 = DCM[2][2];
    double trace = a11 + a22 + a33;

    double qw, qx, qy, qz;

    if (trace > 0) {
        qw = 0.5 * sqrt(1 + trace);
        qx = (DCM[2][1] - DCM[1][2]) / (4 * qw);
        qy = (DCM[0][2] - DCM[2][0]) / (4 * qw);
        qz = (DCM[1][0] - DCM[0][1]) / (4 * qw);
    } else if (a11 > a22 && a11 > a33) {
        qx = 0.5 * sqrt(1 + a11 - a22 - a33);
        qw = (DCM[2][1] - DCM[1][2]) / (4 * qx);
        qy = (DCM[0][1] + DCM[1][0]) / (4 * qx);
        qz = (DCM[0][2] + DCM[2][0]) / (4 * qx);
    } else if (a22 > a33) {
        qy = 0.5 * sqrt(1 + a22 - a11 - a33);
        qw = (DCM[0][2] - DCM[2][0]) / (4 * qy);
        qx = (DCM[0][1] + DCM[1][0]) / (4 * qy);
        qz = (DCM[1][2] + DCM[2][1]) / (4 * qy);
    } else {
        qz = 0.5 * sqrt(1 + a33 - a11 - a22);
        qw = (DCM[1][0] - DCM[0][1]) / (4 * qz);
        qx = (DCM[0][2] + DCM[2][0]) / (4 * qz);
        qy = (DCM[1][2] + DCM[2][1]) / (4 * qz);
    }

    q_ref[0] = qw;
    q_ref[1] = qx;
    q_ref[2] = qy;
    q_ref[3] = qz;

}


// Function to calculate the dot product of two quaternions
double quaternion_dot_product(const double q1[4], const double q2[4]) {
    return q1[0] * q2[0] + q1[1] * q2[1] + q1[2] * q2[2] + q1[3] * q2[3];
}

// Function to flip the sign of a quaternion
void flip_quaternion(double q[4]) {
    q[0] = -q[0];
    q[1] = -q[1];
    q[2] = -q[2];
    q[3] = -q[3];
}

// Function to calculate quaternion-based rotation matrix (3-2-1 rotation)
void quaternion_321_rotation(double qw, double qx, double qy, double qz, double R[3][3]) {
    R[0][0] = 1 - 2 * (qy * qy + qz * qz);
    R[0][1] = 2 * (qx * qy - qz * qw);
    R[0][2] = 2 * (qx * qz + qy * qw);

    R[1][0] = 2 * (qx * qy + qz * qw);
    R[1][1] = 1 - 2 * (qx * qx + qz * qz);
    R[1][2] = 2 * (qy * qz - qx * qw);

    R[2][0] = 2 * (qx * qz - qy * qw);
    R[2][1] = 2 * (qy * qz + qx * qw);
    R[2][2] = 1 - 2 * (qx * qx + qy * qy);
}

// Function to calculate the norm of a 3-element vector
double vector_norm(double v[3]) {
    return sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
}
```

```c

// Function to multiply a 3x3 matrix with a 3-element vector (dot product)
void matrix_vector_dot_product(double matrix[3][3], double vector[3], double result[3]) {
    for (int i = 0; i < 3; i++) {
        result[i] = matrix[i][0] * vector[0] + matrix[i][1] * vector[1] + matrix[i][2] *
            vector[2];
    }
}

// Function to calculate the inverse of a 3x3 matrix
void matrix_inverse(double input[3][3], double output[3][3]) {
    double det = input[0][0] * (input[1][1] * input[2][2] - input[1][2] * input[2][1])
               - input[0][1] * (input[1][0] * input[2][2] - input[1][2] * input[2][0])
               + input[0][2] * (input[1][0] * input[2][1] - input[1][1] * input[2][0]);

    if (det == 0) {
        // Matrix is singular, cannot be inverted
        printf("Error:␣Singular␣matrix,␣cannot␣compute␣inverse.\n");
        return;
    }

    double inv_det = 1.0 / det;

    output[0][0] = (input[1][1] * input[2][2] - input[1][2] * input[2][1]) * inv_det;
    output[0][1] = (input[0][2] * input[2][1] - input[0][1] * input[2][2]) * inv_det;
    output[0][2] = (input[0][1] * input[1][2] - input[0][2] * input[1][1]) * inv_det;

    output[1][0] = (input[1][2] * input[2][0] - input[1][0] * input[2][2]) * inv_det;
    output[1][1] = (input[0][0] * input[2][2] - input[0][2] * input[2][0]) * inv_det;
    output[1][2] = (input[0][2] * input[1][0] - input[0][0] * input[1][2]) * inv_det;

    output[2][0] = (input[1][0] * input[2][1] - input[1][1] * input[2][0]) * inv_det;
    output[2][1] = (input[0][1] * input[2][0] - input[0][0] * input[2][1]) * inv_det;
    output[2][2] = (input[0][0] * input[1][1] - input[0][1] * input[1][0]) * inv_det;
}

// Function to multiply a 3x3 matrix with a 3-element vector
void matrix_vector_multiply(double matrix[3][3], double vector[3], double result[3]) {
    for (int i = 0; i < 3; i++) {
        result[i] = 0;
        for (int j = 0; j < 3; j++) {
            result[i] += matrix[i][j] * vector[j];
        }
    }
}

void solve_thruster_problem(const double* T_matrix, idxint n_thrusters, double T_c[3], double
     F_SSP_max) {

    // Dynamically calculate the number of thrusters (columns of T_matrix)
    idxint n = n_thrusters;  // Number of variables (thrusters)
    idxint m = 2 * n;        // Number of inequalities (0 <= thrust <= F_SSP_max)
    idxint p = 3;            // Number of equality constraints (T_matrix * thrust == T_c)
    idxint l = 2 * n;            // Number of simple bounds (0 <= thrust)
    idxint ncones = 0;       // Number of second-order cones (none in this case)
    idxint nex = 0;          // Number of exponential cones (none in this case)

        // Objective vector (sum of thrust is minimized)
        double* c = (double*)malloc(n*sizeof(double));
        for (idxint i = 0; i < n; i++) {
                c[i] = 1.0;
        }

        // Define G matrix as sparse in compressed column format
        idxint* G_i = (idxint*)malloc(2 * n * sizeof(idxint)); // Row indices for G's non-
            zero entries
        idxint* G_j = (idxint*)malloc((n + 1) * sizeof(idxint)); // Column indices (
            compressed format)
        double* G_x = (double*)malloc(2 * n * sizeof(double)); // Non-zero values

        for (idxint i = 0; i < n; i++) {
            G_i[2 * i] = i;        // Lower bound row index
            G_x[2 * i] = -1.0;     // Represents thrust >= 0 (negative because inequalities are G * x   h)

            G_i[2 * i + 1] = n + i; // Upper bound row index
            G_x[2 * i + 1] = 1.0;   // Represents thrust   F_SSP_max

            G_j[i] = 2 * i; // Start of each column in `G_x` (compressed column index)
        }
        // End marker for the last column
        G_j[n] = 2 * n;

        // Vector h (right-hand side of the inequality constraints)
        double* h = (double*)malloc(2*n*sizeof(double));
        for (idxint i = 0; i < n; i++) {
                h[i] = 0.0; // Lower bound 0 <= thrust
                h[n+i] = F_SSP_max; // Upper bound thrust <= F_SSP_max
        }

    // Define sparse matrix arrays
    double A_x[] = {0.15, -0.15, -0.15, 0.15, -0.1, 0.1};
    idxint A_i[] = {2, 2, 0, 0, 1, 1};
    idxint A_j[] = {0, 1, 2, 3, 4, 5, 6};

    // RHS vector (T_c)
    double* b = (double*)malloc(p * sizeof(double));
    for (idxint i = 0; i < p; i++) {
        b[i] = T_c[i];
    }

    // Sparse G and A matrix arrays in compressed column storage (ccs)
    // Gpr: Non-zero values of G
    // Gjc: Column index array of G
    // Gir: Row index array of G
    // Similar structure for A
    pwork *work = ECOS_setup(
        n,        // Number of variables (thrusters)
        m,        // Number of inequalities
        p,        // Number of equality constraints
        l,        // Dimension of the positive orthant
        ncones,   // Number of second-order cones
        NULL,     // No second-order cones (hence NULL)
        nex,      // Number of exponential cones
        G_x,      // Sparse G matrix data
        G_j,      // G matrix column index array (compressed)
        G_i,      // G matrix row index array (compressed)
        A_x,      // Sparse A matrix data
        A_j,      // A matrix column index array (compressed)
        A_i,      // A matrix row index array (compressed)
        c,        // Cost function
        h,        // RHS of inequalities
        b         // RHS of equalities
    );

    // Solve the problem
    idxint exitflag = ECOS_solve(work);

    // Buffer to hold the UART messages
    char uart_msg[100];

    // Check for optimal solution and store the result in thrust_array
    if (exitflag == ECOS_OPTIMAL) {
        snprintf(uart_msg, sizeof(uart_msg), "Optimal␣solution␣found:\r\n");
//      HAL_UART_Transmit(&huart2, (uint8_t*)uart_msg, strlen(uart_msg), HAL_MAX_DELAY);
        for (idxint i = 0; i < n; i++) {
            thrust_array[i] = work->x[i];  // Store the thrust values in the global array
        }
    } else {
        snprintf(uart_msg, sizeof(uart_msg), "ECOS␣failed␣with␣exitflag:␣%ld\r\n", exitflag);
//      HAL_UART_Transmit(&huart2, (uint8_t*)uart_msg, strlen(uart_msg), HAL_MAX_DELAY);
    }
```

```c

    // Clean up workspace
    ECOS_cleanup(work, 0);

    // Free dynamically allocated memory
    free(c);
    free(G_i);
    free(G_j);
    free(G_x);
    free(h);
    free(b);
}


void updatePWMFrequency(TIM_HandleTypeDef *htim, uint32_t channel, double value) {
    // Only the duty cycle actually has to be updated
    // Based on hydrazine varying thrust paper
    // This is a first assumption, later refinement will follow

    uint32_t duty_cycle;

    if (value < (double)(minimum_impulse_bit / timer_period)) {
        duty_cycle = 0;
    } else {
        // Calculate the duty cycle normally
        duty_cycle = (uint32_t)((value / F_SSP_max) * timer_period);
    }

    uint8_t buffer[50];  // Adjust size based on the expected length of the message

    // Format string to include timer period and duty cycle
    sprintf((char*) buffer, "%u,%u\n", timer_period, duty_cycle);

    // Send the formatted string via UART
    HAL_UART_Transmit(&huart2, buffer, strlen((char*)buffer), HAL_MAX_DELAY);

    // Dynamically set the duty cycle for the specified channel
    switch (channel) {
        case 1:
            __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_1, duty_cycle);
            break;
        case 2:
            __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_2, duty_cycle);
            break;
        case 3:
            __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_3, duty_cycle);
            break;
        case 4:
            __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_4, duty_cycle);
            break;
        default:
            // Invalid channel
            sprintf((char*) buffer, "Error:␣Invalid␣Channel\n");
            HAL_UART_Transmit(&huart2, buffer, strlen((char*)buffer), HAL_MAX_DELAY);
            break;
    }
}
```