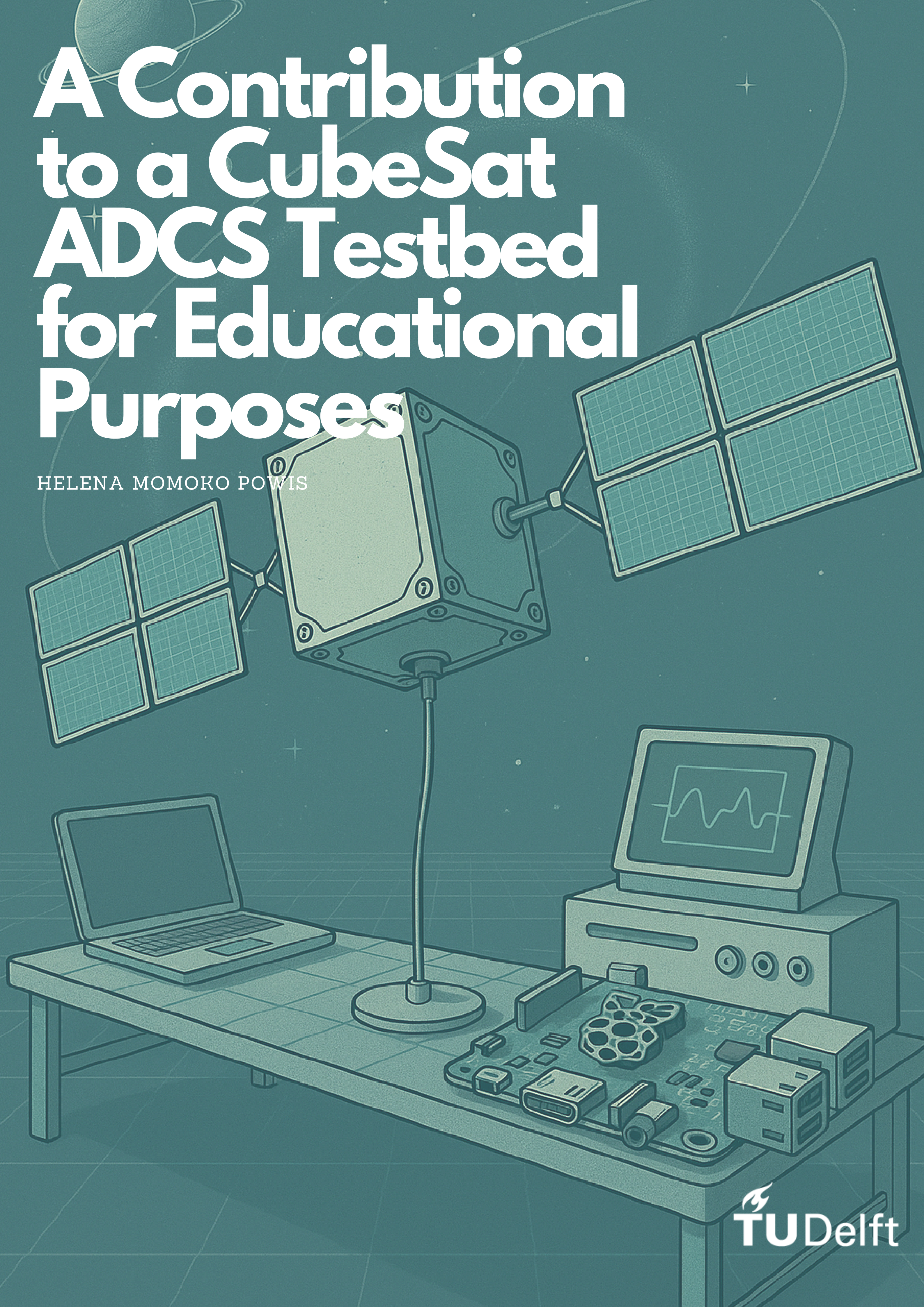
A Contribution to a CubeSat ADCS Testbed for Educational Purposes

HELENA MOMOKO POWIS



TU Delft

A Contribution to a CubeSat ADCS TestBed for Educational Purposes

by
Helena Momoko Powis

Master of Science Thesis

in partial fulfilment of the requirements for the degree of
Master in Science in Aerospace Engineering

Department of Space Engineering
Faculty of Aerospace Engineering Delft University of Technology
The Netherlands

Student number: 4534751
Supervisor: Dr.ir. Erwin Mooij

May 26, 2025

Preface

The completion of this thesis marks the end of a truly transformative nine-year journey at TU Delft and the conclusion of my formal education. Though my path may have been longer than most, I look back with immense pride and appreciation. Every twist and turn has helped shape who I am today, and I wouldn't change a thing. These years have been filled with growth, discovery, and unforgettable moments, and I feel incredibly fortunate for the opportunities I've experienced and the people I've met along the way. I've forged friendships that will last a lifetime and carry a profound sense of gratitude for the support and kindness I've received throughout this chapter of my life.

I would also like to thank Leon Bremer and Airbus for the hours of help with implementing Eurosim and for the Eurosim License for this thesis.

First and foremost, I wish to express my heartfelt gratitude to Dr. Erwin Mooij for his exceptional guidance as my supervisor for both my MSc and BSc theses. Your belief in my capabilities revealed potential within me that I hadn't fully recognised, potential that has now led me to achievements I once considered beyond my reach, including my job at ESA! Our Wednesday morning meetings transformed what could have been a daunting process into an engaging and enlightening experience. Even during periods of uncertainty when solutions seemed elusive, your direction kept me motivated and focused. Your patience and encouragement exemplify not only your excellence as an educator but also your compassion as a mentor. I'm delighted to have connected with another fellow cat enthusiast!

To my life partner, Tristan: your steadfast presence has been my foundation throughout this endeavour. Thank you for providing not only practical support but also the emotional strength that allowed me to persevere. Your unconditional love, endless patience, and unwavering belief in my abilities have empowered me to overcome every challenge. I cherish our partnership and look forward to supporting your dreams as you've supported mine. And to Spock, my beloved cat, your quiet company brought me comfort and joy throughout this journey.

To my cherished friends, Kiva, Serena, Aïcha, Sara, Siya, Maxim, and Todor, thank you for your unwavering support, encouragement, and the countless ways you each made this journey brighter. I am endlessly grateful. I especially want to acknowledge Kiva and Serena, my dream team. Serena, your consistent faith in my abilities has sustained me even when self-doubt crept in. I'm privileged to call such a brilliant and inspiring woman my friend. Kiva, my dedicated TA partner, thesis buddy and steadfast study companion, Thank you for always being my supporter in this journey. Your exceptional work ethic and generosity of spirit are qualities I deeply admire, and I aspire to reciprocate the support you've so freely given.

To my family: my father, my sister, and my stepmother, thank you for your loving support throughout this journey. To my おばあちゃんとおじいちゃん, いつも応援してくれてありがとう. Thank you all for believing in me. I hope I've made you proud and will continue to do so in all my future endeavours. Finally, my mother, I hope I have made you proud.

This is dedicated to my dad and her.

Abstract

Attitude Determination and Control Systems (ADCS) are a critical subsystem in CubeSat missions, responsible for controlling the orientation of the satellite in space. However, for student-led and academic missions, the validation of ADCS algorithms often proves challenging due to limited budgets, lack of access to flight hardware, and the absence of real-time testing facilities.

This thesis addresses the research question: *To what extent can low-cost hardware and simulation tools support real-time testing of CubeSat ADCS algorithms in an educational context?*

By combining MATLAB/Simulink models with the GRADS and GGNCSim libraries, alongside the EuroSim real-time framework, a modular ADCS simulation and test environment was developed for under €600 in hardware costs.

A functional ADCS simulator was implemented in MATLAB, incorporating environmental models (e.g., gravity, atmospheric drag, magnetic field), sensor models (magnetometer and sun sensors), actuator models (magnetorquers), and both B-dot and proportional-derivative (PD) control algorithms. Extended Kalman Filter (EKF) was developed to estimate satellite attitude, with the dual-sensor fusion case achieving the lowest quaternion error and numerical stability over 1,000 seconds of simulation. Real-time code generation and deployment were performed using Simulink Coder and Embedded Coder, targeting Raspberry Pi 5 and Orange Pi 5 Plus platforms. Despite EuroSim limitations, such as manual model integration, function duplication, and MUX/bus conflicts, a structured twelve-step workflow was developed to ensure consistent and repeatable builds.

Software-in-the-loop (SIL) testing validated the control logic and demonstrated that the test bench could operate at 100 Hz without deadline overruns. Closed-loop detumbling using B-dot control reduced angular velocity effectively, while PD control achieved sun-pointing within 50 seconds and maintained a steady-state error below 0.5°, well within the requirements for CubeSat mission SA01. Although full hardware-in-the-loop (HIL) integration was not realised within the scope of this thesis, stimulus-response tests suggest the platform is ready for the addition of physical sensors and actuators.

The primary limitations of the system are the complexity of integrating large models into EuroSim and the manual effort required to split and configure models for real-time deployment. These limitations inform several key recommendations, including automating the model export process, evaluating alternative real-time frameworks, and implementing low-risk HIL configurations as a next step. Nonetheless, the developed platform successfully demonstrates a low-cost, educationally accessible ADCS test bench that supports real-time experimentation. This contributes a practical and scalable approach for academic institutions aiming to enhance hands-on training in space systems engineering and reduce mission risk through improved pre-flight validation.

Contents

List of Symbols

Latin

Symbol	Description	Units
A	Exposed surface area of satellite	m^2
a	Semi-major axis of the orbit	m
$\mathbf{B}$	Earth magnetic-field vector	T
C_D	Aerodynamic drag coefficient	–
C_r	Solar-radiation reflection coefficient	–
c	Speed of light	ms^{-1}
e	Eccentricity	–
$\mathbf{e}$	Euler rotation axis	–
F	Force	N
g	Standard gravitational acceleration	ms^{-2}
h	Altitude	m
H_0	Scale height (atmospheric model)	m
$\mathbf{I}$	Inertia tensor (body-fixed)	kgm^2
i	Inclination angle	rad
$\mathbf{m}$	Magnetic dipole moment	Am^2
M	Mean anomaly	rad
m	Satellite mass	kg
$\mathbf{q}$	Attitude quaternion	–
$\mathbf{r}$	Position vector (inertial frame)	m
T_c	Command torque	Nm
T_d	Disturbance torque	Nm
Δt	Time increment	s
$\mathbf{u}$	Control input vector	–
V	Voltage	V
$\mathbf{x}$	State vector	–

Greek

Symbol	Description	Units
$\alpha(t)$	Scaling factor (sensor model)	–
β	Bias term	–
ϵ	Surface emissivity / reflectivity	–
μ_{mag}	Magnetic permeability of free space	Hm^{-1}
Ω	Right ascension of the ascending node	rad
ω	Angular rate	$rads^{-1}$
ω	Argument of perigee (orbital)	rad
ϕ	Solar flux	Wm^{-2}
ρ	Atmospheric Density	kgm^{-3}
θ	Pitch angle	rad

List of Abbreviations

Abbreviation	Description
3DOF	Three Degrees of Freedom
ACMS	Attitude Control and Measuring System
ADCS	Attitude Determination and Control System
AGI	Ansys Government Initiatives
AIT	Assembly, Integration and Testing
AMR	Anisotropic Magneto-Resistive (sensor)
AOCS	Attitude and Orbit Control System
B-dot	Magnetic-rate Detumbling Algorithm (controller)
CDH	Command and Data Handling
CDMU	Command and Data Management Unit
COTS	Commercial Off-The-Shelf
DCM	Direction Cosine Matrix
ECSS	European Cooperation for Space Standardisation
EKF	Extended Kalman Filter
EPS	Electrical Power System
ERC32	Embedded Real-Time Computer 32-bit
ESA	European Space Agency
FOV	Field of View
FSS	Fine Sun Sensor
GDOP	Geometric Dilution of Precision
GGNCSim	Generic Guidance, Navigation and Control Simulator
GNC	Guidance, Navigation and Control
GPS	Global Positioning System
HIL	Hardware-In-The-Loop
IMU	Inertial Measurement Unit
LVLH	Local Vertical Local Horizontal
NED	North East Down
OBC	On-Board Computer
PD	Proportional-Derivative (controller)
PID	Proportional-Integral-Derivative (controller)
RF	Reference Frame
RMS	Root-Mean-Square
RTS	Real-Time Simulator
SCOE	Software Checkout Equipment
SIL	Software-In-The-Loop
SSO	Sun-Synchronous Orbit
SPS	Stackable Platform Structure
STK	Systems Tool Kit (AGI software)

CHAPTER 1

Introduction

CubeSats first appeared in 1999 (Puig-Suari et al., 2001) as a cheaper and faster alternative for educational and research purposes. Figure 1.1 shows a continuous and exponential increase in the number of CubeSats, alongside a steadily growing percentage of satellites created by educational institutions[1]. Although the methods of production and manufacturing are significantly cheaper than those of large-scale satellites, testing for small-scale platforms such as CubeSats remains inconvenient and expensive for education-based missions, often leading to avoidable failures.

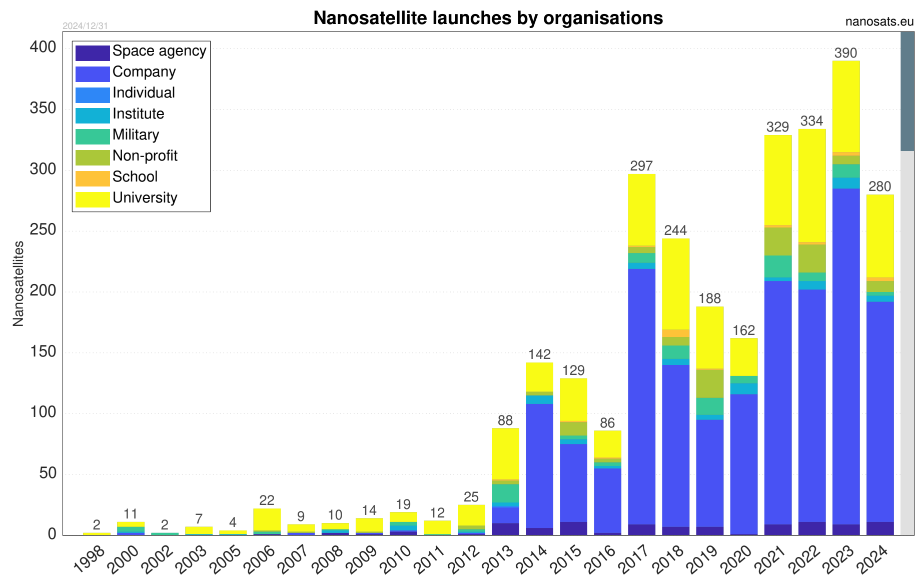


Figure 1.1: CubeSat launches per year separated by organisation [1]

In a study by Guon et al. (2014), which analysed 222 small satellite failures between 1990 and 2010, it was found that early-stage deployment failures were significantly higher for small satellites compared to larger ones. This trend can be attributed to different design philosophies, shorter testing periods, and the use of lower-cost components. Additionally, university-built satellites were found to be less reliable than those developed by companies or agencies. Another study examining 156 small satellites (Tafazoli, 2009) reported that 32% of in-orbit failures were due to issues within the Attitude and Orbit Control System (AOCS), particularly related to control processors and components. This represented the largest single failure category, followed by power distribution (27%) and computer data handling (15%).

Popular testing methods for satellites include Software-in-the-Loop (SIL) and Hardware-in-the-Loop (HIL) testing. SIL utilises software tools to simulate system inputs and outputs, replicating space environment perturbations such as magnetic fields, solar radiation, and atmospheric drag, along with control-induced movements. HIL testing extends this by replacing some simulated components with real hardware, for instance using actual sensors or actuators interfaced with a simulated environment. While major space agencies and private companies

[1] https://www.nanosats.eu/figures, last accessed 11/11/23

typically build dedicated in-house test facilities, educational institutes often lack the necessary resources, resulting in minimal or insufficient testing.

Real-time simulators are increasingly employed in satellite development to test subsystems under conditions that closely mimic actual operational timings. Unlike traditional offline simulations, real-time environments require the system to process inputs and outputs within strict time constraints, enabling more realistic integration and HIL testing. Examples of their use include communication simulations with the Real-Time Satellite Network Emulator at ESOC (European Space Agency, 2022), and ADCS testing for coordinated manoeuvres such as those required in satellite constellations (Kassem and Sastry, 2024). Although implementing real-time simulation is typically more expensive both computationally and economically, it offers substantial advantages by improving testing fidelity, revealing timing-related faults earlier in development, and supporting more robust system validation.

Commercial real-time test systems cost tens of thousands of euros and require specialist support. Until recently this put HIL-grade validation out of reach for most universities. The rapid advance of low-cost, single-board computers changes that picture. A Raspberry Pi 5, priced below €100, delivers a 2.4 GHz quad-core CPU and hardware floating-point unit, which is enough to run a 6-DOF (Degree Of Freedom) satellite model and closed-loop controller at 100 Hz in real time. When combined with free or academic-licence software such as MATLAB or Simulink and the EuroSim kernel, these boards provide a credible alternative to a set-up at a fraction of the cost and complexity. Harnessing such hardware for ADCS development therefore promises two linked benefits: it lowers financial barriers for student teams and shortens the design–test cycle by bringing laboratory-grade HIL capability onto the desktop.

To help reduce failure rates for educational CubeSats, this thesis focuses on developing a cost-effective, rapid testbed for Attitude Determination and Control Systems (ADCS), employing Software-in-the-Loop methods with the capability for Hardware-in-the-Loop extensions. This leads to the following central research question:

Main Research Question

To what extent can low-cost hardware and simulation tools support real-time testing of CubeSat ADCS algorithms in an educational context?

The remainder of this thesis is structured as follows. It begins with a discussion of mission heritage in Chapter 2, reviewing relevant past, present, and future satellite missions, along with descriptions of their ADCS systems and associated testing methods. The chapter concludes with a description of the reference mission, its requirements, and an analysis of the main research question and sub-questions.

Next, Chapter 4 provides an overview of the relevant flight dynamics and mechanics used to analyse satellite motion. Chapter 5 examines the environmental disturbances encountered by satellites and their mathematical representations. Chapter 6 discusses navigation aspects, focusing on how satellite sensors perceive the environment and how this information is processed through navigation filters.

Chapter 7 covers the control modes for the reference mission, the actuators involved, and their mathematical modelling. This is followed by Chapter 8 on functional simulation, which examines the non-real-time simulation environment, the simulator's construction, and the implementation of verification and validation (V&V) procedures in MATLAB.

Chapter 9 presents the implementation of the real-time simulator using EuroSim, including model development and its associated verification and validation processes. Finally, the thesis conclusions and recommendations are discussed in Chapter 10.

CHAPTER 2

Mission Heritage

Advances in miniaturisation, space-qualified electronics and real time simulation have greatly expanded capabilities of small satellites. Yet many university and school based CubeSat projects continue to experience high failure rates, frequently attributed to faults in the Attitude Determination and Control System; the lack of access to representative test facilities remains a key factor.

This chapter presents a survey of historical and contemporary ADCS development and validation practices. Section 2.1 examines agency scale missions, detailing control architectures, hardware selections and multi stage verification workflows. Section 2.2 addresses CubeSat programmes, identifying prevailing control modes, sensor and actuator suites and the software in the loop and hardware in the loop methods adopted. Section 2.3 distils these findings into two principal test approaches, software in the loop and hardware in the loop, which underpin the low cost ADCS test bench described in later chapters.

2.1 Large-Scale Satellite Missions

This section looks at past, present, and future missions for large-scale satellites. The objective is to analyse trends in ADCS building and testing to determine the limitations and benefits of real-time testing and their application on low-cost equipment. First, the mission objectives are given. The ADCS is described, including chosen control algorithms and hardware. This is followed by a description of CubeSat-specific missions, and then the reference mission design is provided. This will help determine the reference mission designed in this thesis, as well as the testing method.

2.1.1 Gaia

Gaia, a 2,029 kg spacecraft launched on December 19, 2013, was designed to reach the Sun-Earth L2 Lagrange Point. Led by ESA and developed by Astrium SAS (prime), with Astrium Ltd handling the electrical service module and ADCS, the mission aimed to map the Milky Way by measuring stellar positions and velocities. Its payload included an Astrometric Instrument (ASTRO), a Photometric Instrument, and a Radial Velocity Spectrometer (RVS).[1] Figure 2.1 shows the ADCS mode flow: from standby to Sun acquisition, inertial guidance, orbit control, and finally normal operation. The mission cost 740 M€,[2], indicating that budget and resources were not major constraints for this ESA-led project.

Testing

Dutch Space BV developed a real-time simulator (RTS)[3] and avionics Software CheckOut Equipment (SCOE) to test Gaia's systems, including the CDMU and the ERC32-based emulator, SIMERC32. Testing followed a three-step software verification: SIMERC32 emulates the

[1]https://science.nasa.gov/mission/gaia/ accessed on 11/01/24

[2]https://www.esa.int/Science_Exploration/Space_Science/Gaia/Frequently_Asked_Questions_about_Gaia accessed on 11/01/24

[3]chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://indico.esa.int/event/108/contributions/135/attachments/222/255/02_01_Beerthuizen_presentation.pdf accessed 15/12/23

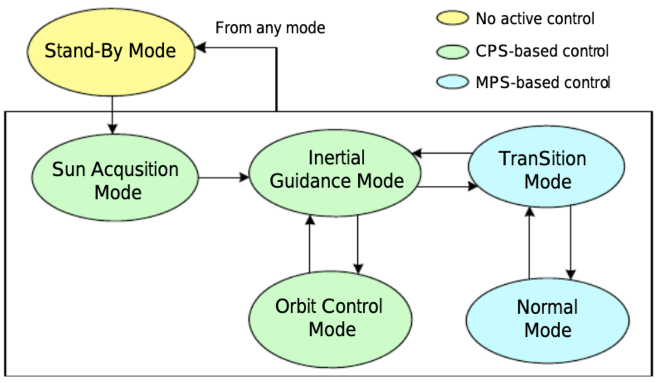


Figure 2.1: Gaia - ADCS mode flow diagram (Chapman et al., 2008)

ERC32 processor; SIMAIT adds sensor and actuator models for closed-loop simulation; and EuroSim performs real-time testing, enabling hardware-in-the-loop without recompilation.[4][5]

The ADCS was tested entirely in software, simulating control algorithms and data handling. Figure 2.2 shows the ADCS simulation setup, which considered five perturbations: external (solar radiation, micrometeoroids, thermal infrared emissions) and internal (thruster noise, thermal "clanks"). Radio emissions from the antenna were also included.

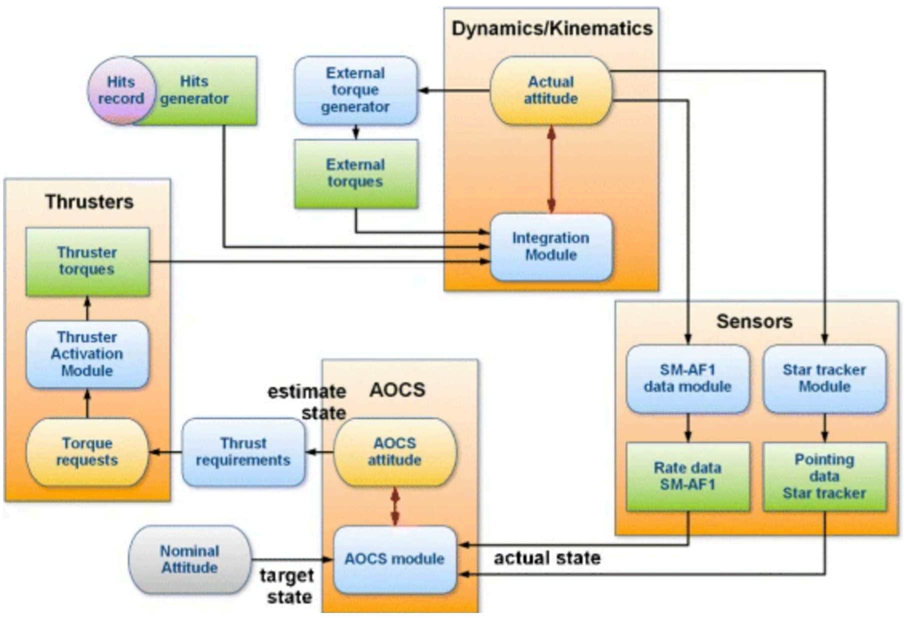


Figure 2.2: Gaia - ADCS simulation flow diagram (Risquez et al., 2012)

2.1.2 Stackable Platform Structure (SPS) -2

The stackable platform structure 2 (SPS-2) is a proposed CubeSat and small satellite deployment module. SPS-1 served as a technology demonstrator, however, SPS-2 now has a service module to allow for in-orbit demonstrations of new technologies. It will be in an elliptical orbit of 350x850 km with an inclination between 94-99 degrees.

[4]https://indico.esa.int/event/108/contributions/153/attachments/182/214/10_02_Cazenave_presentation.pdf, accessed 02 Jan 2023

[5]https://www.eurosim.nl/applications/gaia-rts.shtml, accessed 10 Jan 2024

Testing

The ADCS software was developed and tested using ADCS Design software and the Generic Guidance, Navigation and Control Simulator (GGNCSim), which is a product add-on of Eurosim. This can be used in MATLAB/Simulink. Oomen (2020) performed a trade-off for simulators. One module considered was ADS (AOCS Design Software), which is a commercially available software package created in 1998, also used by Airbus for other projects. The next was GGNCSim on Eurosim, which contains a selection of models of sensors, actuators and environment models used for simulating and modelling the satellite ADCS system. It was found that GGNCSim was the better component, as there were many limitations to ADS, including linearisation in equations and complicated user interfaces. The simulation flow diagram can be found Figure 2.3. The system was run at a frequency of 1 Hz.

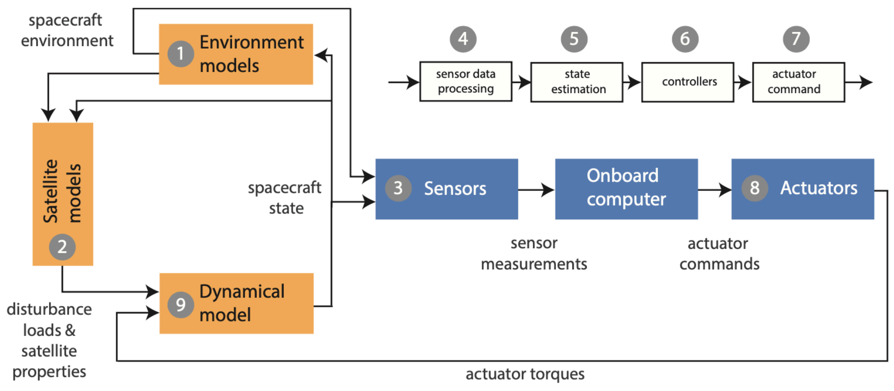


Figure 2.3: SPS-2 - ADCS simulation flow diagram (Oomen, 2020)

The sensor and actuator models from GGNCSim include biases, nonlinearity, scale factor errors, noise, misalignment errors, saturation and quantisation. Within the Eurosim package, the following models were used. The gravitational model used was `env-gravity-grim5c1-sfun`. The magnetic field model used was `env-magnetic-igrf-epoch-1995-sfun`. The solar radiation model used was `env-pressure-sun-radiation-sfun` and the atmospheric density model used was `env-atmosphere-msis86-min-sfun`, `env-atmosphere-msis86-nom-sfun` and `env-atmosphere-msis86-max-sfun`. The sun sensor was based on the Fine sun sensor model without the Earth's albedo error. The reaction wheel model, gyroscope, magnetorquer and magnetometer were developed in-house, and the control models used included a Bdot controller and a linear pi controller.

2.1.3 Herschel-Planck

The Herschel Space Observatory was an ESA-built space observatory operational between 2009 to 2013. It carried a 3.5 m mirror, which was the largest infrared telescope launched at the time [6] Planck was also a space observatory aimed to map the anisotropies of the cosmic microwave background. The Planck satellite rotated one revolution per minute.

Testing

Airbus Defence and Space B/V developed the Herschel-Planck Attitude Control and Measuring System (ACMS) Special Checkout Equipment (SCOE). It was developed in MATLAB and tested with the onboard software. The simulator was built using EuroSim and the control

[6] https://www.esa.int/Enabling_Support/Operations/Herschel accessed on 11/01/24

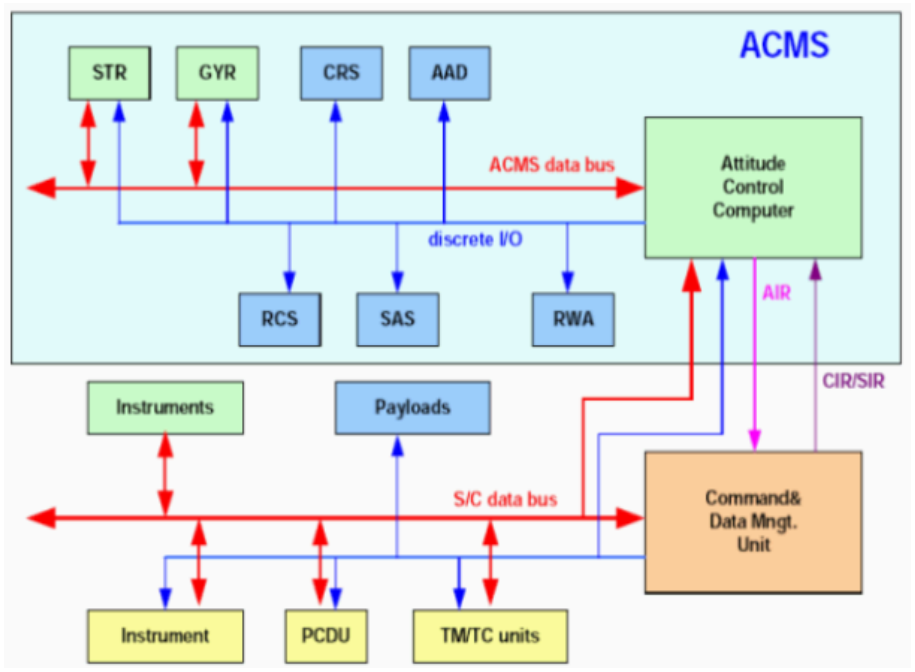


Figure 2.4: Herschel - ADCS simulation flow diagram (Sanchez-Portal et al., 2014)

algorithms were created with the help of GGNCSim. This setup included a variety of input-output (IO) interfaces to allow for analogue and digital interfaces.[7] This meant the simulation set up was HILT capable, and different pieces of hardware could be added and tested in software.

2.1.4 Discussion of Large Scale Missions

Large-scale programmes span LEO service satellites through to deep-space explorers. They pursue broad objectives planetary science, Earth system monitoring or interplanetary operations and thus employ large platforms, advanced subsystems, substantial budgets and multi-phase test campaigns. Their control architectures feature specialised modes for each mission phase, validated via multi-stage SIL and HIL and supported by extensive ground test facilities, yielding high confidence in mission success.

Key practices transferable to education-based CubeSat ADCS testing include:

- Employing EuroSim for high-fidelity, fixed-rate real-time runs under an educational licence.
- Using flat-sat layouts (external electrical harnesses) to verify hardware early in development.
- Adopting a unified SIL to HIL workflow to migrate smoothly from pure software tests to integrated hardware loops.
- Leveraging modular Simulink toolboxes (e.g.GGNCSim) for pre-validated sensor, actuator and environment models under academic licences.

These elements form a practical foundation for a low-cost, resource-aware ADCS test bench.

[7] https://www.eurosim.nl/applications/hpscoe.shtml accessed 10/01/24

2.2 Cubesat specific missions

This section focuses on CubeSat-specific missions following the same format as the previous section. First, the mission objectives are given. The ADCS is described, including software tools and hardware components, followed by the testing methods.

2.2.1 MOVE-II (Munich Orbital Verification Experiment II)

MOVE-II (Munich Orbital Verification Experiment II) is a 1U CubeSat launched in December 2018 by students and staff of the Technical University of Munich. An image of the MOVE-II CubeSat can be found in Figure 2.5. The purpose of the mission was educational and to test and verify the payload implementation. This payload included communication systems, onboard data handling processes, attitude control, and power and thermal control systems.

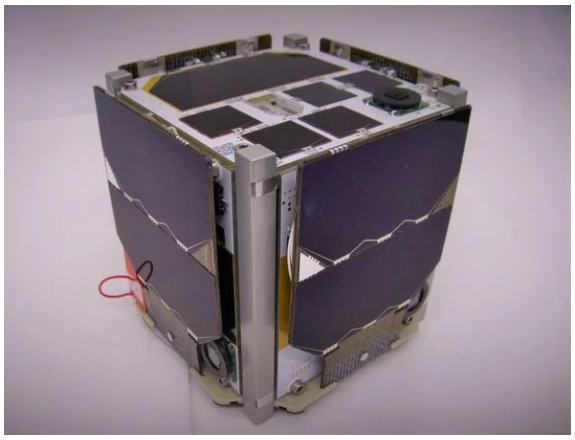


Figure 2.5: Image of MOVE-II[7]

Testing

The ADCS, as well as electrical power systems (EPS) were tested with HIL and SIL testing methods (Kiesbye et al., 2019). The main focus was to verify the software implementation and controllers in an integrated configuration. This was implemented using MATLAB/Simulink, and the block diagram can be seen in Figure 2.6

The testing environment was created using real and simulated hardware. This meant mimicking signals and simulating actuator commands from the response of the processing hardware. The HIL environment, therefore, contained the space environment, the physics models, the interface model with the ADCS, EPS and temperature sensors of the command & data handling (CDH) subsystem. It also computed the distance and orientation to the ground station. The CubeSat was suspended from a string inside of a Helmholtz cage, which simulated a magnetic field.

For testing, the reference Sun-synchronous circular orbit was 575 km (LEO) with an orbital period of 5770 seconds. The Helmholtz cage tests were successful in testing the detumbling controller mode, but were considered limited, as there were not three rotational degrees of freedom. Therefore, the long-term stability was not confirmed. It was assumed the maximum

[7]https://www.asg.ed.tum.de/en/lrt/research-at-the-chair-of-astronautics/satellite-technology/cubesats/move-ii accessed 21/12/23

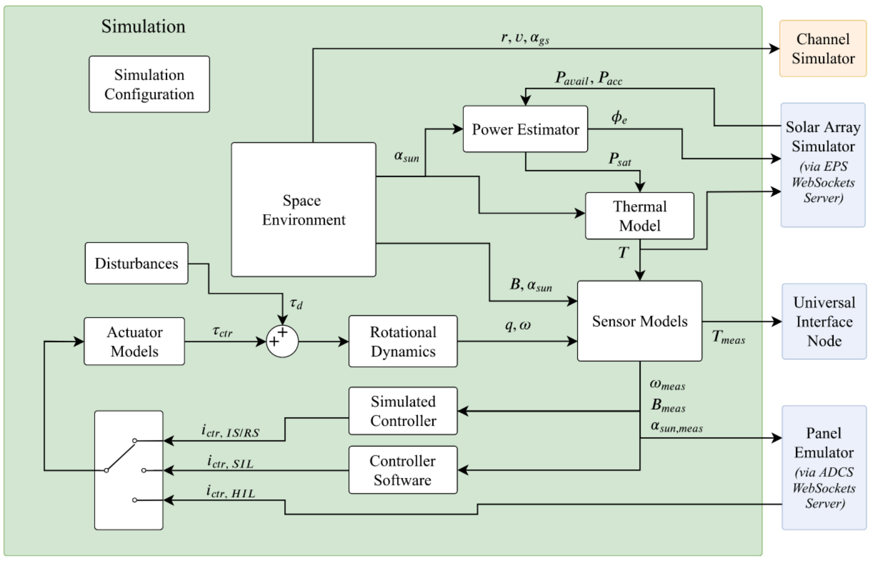


Figure 2.6: Simulink block model of MOVE-II (Kiesbye et al., 2019)

separation velocity from the chosen launcher (ISIS Quadpack Deployer) was around $10^\circ\,\text{s}^{-1}$. The system would consider the detumbling procedure successful when the angular velocity of $7.5^\circ\,\text{s}^{-1}$ was reached. The ideal case of $10^\circ\,\text{s}^{-1}$ to $7.5^\circ\,\text{s}^{-1}$ was complete in 12 minutes. The maximum tests up to an initial angular velocity of $50^\circ\,\text{s}^{-1}$ were also performed and were successful in detumbling down to $1^\circ\,\text{s}^{-1}$in 162 minutes.

When an angular velocity of $7.5^\circ\,\text{s}^{-1}$ was reached, the Sun pointing control mode was activated. Over many orbits, the CubeSat controller exhibited unstable behaviour, where it would align with the axis of the desired torque with the direction of the magnetic field. It was mentioned that an air-bearing table would have made this sort of behaviour hard to detect. This was overcome by adding a counter torque around the z-axis. A mean pointing error of 20.6° and a standard deviation of 7.7° was found. A Monte Carlo simulation was used to provide a random initial attitude and velocity vector to the controller. An Extended Kalman filter (EKF) was used to estimate the satellite's attitude as well as the gyroscope bias. Overall, the real simulation, the SIL and the hardware in the loop were considered powerful tools to test the attitude determination algorithms.

The electrical power system was also simulated and integrated into the simulations. This caused latency of up to 100ms, therefore, the author recommends using universal interface nodes which collect the information packets from the simulation for each sensor and group them as one packet into the control system. Google Protobuf is an open-source, cross-platform data format that was used to serialise the sensor data, so a fixed packet format did not have to be defined.

2.2.2 ITU-PSAT II (Istanbul Technical University PicoSatellite II)

ITU-PSAT II (Istanbul Technical University PicoSatellite II) (Kemal Ure et al., 2011) is the second generation student-built 3U (10x10x30 cm) satellite, weighing 4 kg of ITU Controls and Avionics Laboratory and was launched in September 2009. The purpose of the satellite

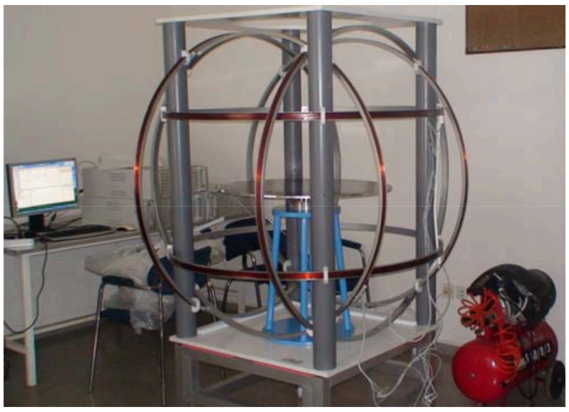


Figure 2.7: Block model of ITU-PSAT II ADC system (Kemal Ure et al., 2011)

was to demonstrate on-orbit ADCS advancements for nanosatellites. This CubeSat operated in a Sun-synchronous orbit (SSO) of 640-840 km.

Testing

The controller has two modes. Detumbling for initial separation from the launcher and high precision attitude control for image capturing. The spacecraft dynamics were parameterised with quaternions and rigid-body dynamics was assumed. Disturbances were modelled as Gaussian white noise. The detumbling mode uses a Bdot controller and the high-precision attitude control mode is done using a linear quadratic performance controller. A Bdot controller uses magnetic field measurements from a magnetometer and a magnetorquer to reduce the change in magnetic field measurements to reduce the angular velocity of the satellite. The control system can also detect sensor and actuator failures.

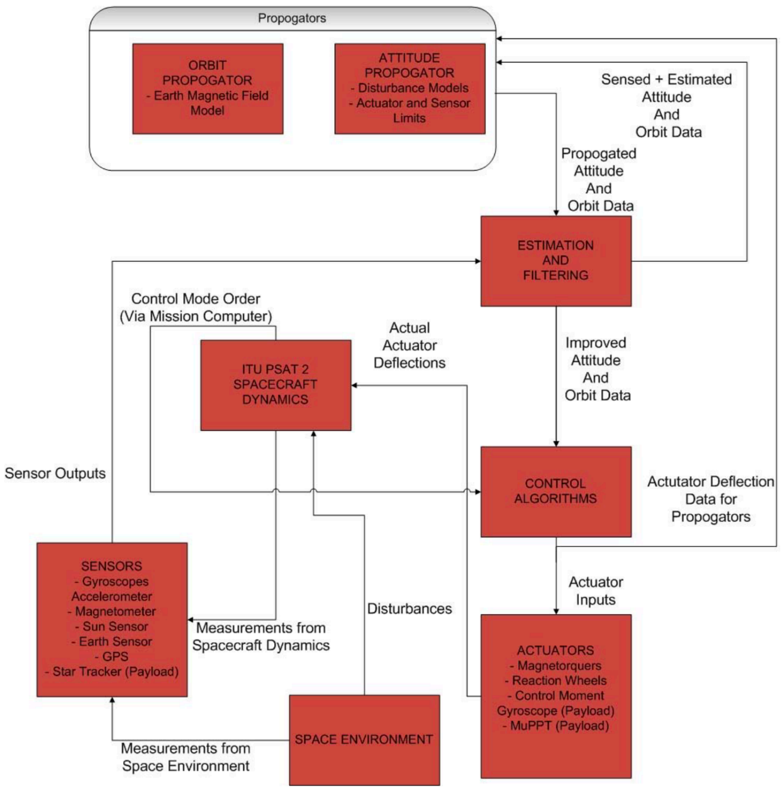

Propogators
ORBIT PROPOGATOR - Earth Magnetic Field Model
ATTITUDE PROPOGATOR - Disturbance Models - Actuator and Sensor Limits
Sensed + Estimated Attitude And Orbit Data
Propogated Attitude And Orbit Data
ESTIMATION AND FILTERING
Control Mode Order (Via Mission Computer)
ITU PSAT 2 SPACECRAFT DYNAMICS
Actual Actuator Deflections
Improved Attitude And Orbit Data
Sensor Outputs
CONTROL ALGORITHMS
Actutator Deflection Data for Propogators
Actuator Inputs
SENSORS - Gyroscopes Accelerometer - Magnetometer - Sun Sensor - Earth Sensor - GPS - Star Tracker (Payload)
Measurements from Spacecraft Dynamics
Disturbances
ACTUATORS - Magnetorquers - Reaction Wheels - Control Moment Gyroscope (Payload) - MuPPT (Payload)
SPACE ENVIRONMENT
Measurements from Space Environment

Figure 2.8: block model of ITU-PSAT II ADC system

The SIL simulation makes use of the simulation tool kit (STK) inside MATLAB. The HIL and SIL testing was done using a 3D air-bearing table and a Helmholtz Coil frame.

2.2.3 1 Kenyan University Nano-Satellite Precursor Flight (1KUNS-PF)

1KUNS (Mwangi-Mbuthia and Ouma, 2016) was the first satellite launched in May 2018 by the Kenyan Space Agency (KSA) in collaboration with the University of Nairobi, Machakos University, and Sapienza University of Rome. The mission's objective was for this 1U cube to send colour images of Earth, as well as test in-orbit in-house technology. This first mission was a predecessor to a 6U cube satellite, which was developed off the back of the successes of 1KUNS-PF. The tested components included the silicone solar panels, the telemetry electronic board, and the three degrees of freedom (3DOF) attitude control system.

The primary mission was to verify the performance of the onboard subsystems receiving telemetry data. The main design drivers were to simplify the onboard systems for basic but well-proven functionality as well as use commercial off-the-shelf (COTS) components, ideally with no custom developments.

The mission had two nominal operation modes. This is the setup phase and an experimental phase. Phase one encompasses the detumbling procedure using passive magnetic attitude stabilisers as well as data gathering of onboard systems before transmitting a beacon. The second phase, triggered by a ground station command, starts the payload experiments. This includes taking images of Earth and sending them back, and testing the momentum wheel. The operation modes are further explained in Table 2.1, where a description of all three operational modes and the conditions of the safety mode are provided.

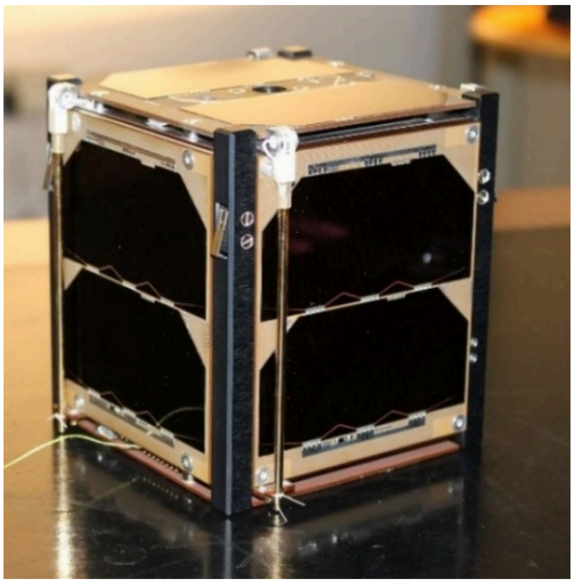


Figure 2.9: 1KUNS-PF (Frezza et al., 2022)

Table 2.1: Operational modes of 1KUNS-PF (Mwangi-Mbuthia and Ouma, 2016)

Mode	Description
Commissioning	After deployment, 1KUNS-PF sends a Beacon Signal once per minute until reliable communication with ground stations is established. The desired attitude is achieved using a passive magnetic stabilisation system.
Nominal	In the first phase, telemetry is stored on board and downloaded to the ground regularly, to assess the performance of the on-board systems and, in particular, the experimental solar panel developed at the University of Nairobi. In the second phase, the camera is switched on/off upon command, sending panchromatic pictures of the Earth to the ground. Experiments on the momentum wheel functionality and performance are conducted.
De-commissioning	The UHF transmitter and all payloads are permanently turned off, including the camera and momentum wheel. Batteries are fully discharged.
Safe	Non-essential subsystems, such as payloads, are turned off; only the receiver remains active, sending a beacon signal. The satellite waits for a command from the ground to re-establish the nominal mode of operation.

Testing

As this was a low-budget, fast production mission, the ADC systems were not specially tested other than simple software tests. This was due to financial and resource limitations, as this was a university project. The tests were focused on TTC and Power systems.

2.2.4 Taifa-1

Taifa-1 was a 3U CubeSat launched in April 2023 and was the first Earth observation satellite launch by the Kenya Space Agency. It was developed by SyariLabs and EnduroSat. It took over two years and cost over $350,000. The mission's purpose was to collect agricultural and environmental data. This satellite was launched into a SSO of 550 km and an inclination of 97°. The satellite was deployed at 508 km and had an orbit period of 95 minutes.[8]

[8] https://ksa.go.ke/news/taifa-1-satellite-launch accessed 12/01/24

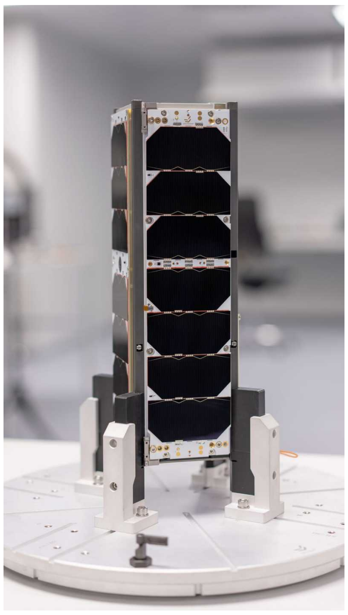


Figure 2.10: Taifa-1[9]

Testing

The system was simulated using the Systems Tool Kit (STK) from AGI (Ansys Government Initiatives) simulation toolkit software available on MATLAB. Ansys STK allows the satellite subsystems to be simulated using physics-based modelling environments[10]. The code itself was written in MATLAB, and STK was used to model sensors to determine pointing accuracy for the Nadir pointing, detumbling rates, also considering perturbations.

2.2.5 Discussion of CubeSat Missions

The CubeSat missions considered in this study ranged from 1U to 6U, with 3U platforms being most common. They typically operated in Low Earth Orbit, often in Sun-synchronous trajectories between 400 km and 600 km. This regime balances mission ambition, data quality and resource limits, making it ideal for educational, research and technology-demonstration objectives.

Most missions adopted two principal control modes: a *detumbling mode* to damp post-deployment spin and a *pointing mode* usually Sun or Earth pointing to support payload functions such as imaging or communications. The detumbling phase commonly employed B-dot controllers driven by magnetometer and magnetorquer readings, offering a low-cost, low-power solution well suited to CubeSat form factors.

Testing strategies for CubeSats have been less extensive than for larger spacecraft. Control algorithms were often validated first in SIL, then in simple HIL setups such as Helmholtz cages or air-bearing tables. MATLAB/Simulink dominated as the development environment, leveraged under academic licences and supplemented by pre-existing libraries.

CubeSat missions thus suggest the following best practices for cost-conscious ADCS test beds:

[9] https://currentaffairs.adda247.com/kenya-launched-its-first-operational-earth-observation-satellite-taifa-1 last accessed 09/01/24

[10] https://www.ansys.com/products/missions/ansys-stk accessed 30-03-2024

- Develop and validate controllers rapidly in MATLAB/Simulink using pre-validated library blocks under an academic licence.
- Implement B-dot detumbling with inexpensive magnetometers and magnetorquers, as demonstrated across multiple missions.
- Design simple proportional–derivative Sun- or Nadir-pointing controllers and verify their performance in SIL.
- Integrate compact HIL fixtures (Helmholtz cages, air-bearing tables) to inject realistic magnetic and rotational stimuli, while managing space and budget requirements.
- Prioritise COTS sensors and actuators to reduce custom hardware effort and improve reproducibility.
- Address gaps in documented HIL procedures by providing a unified, accessible test bench framework.

These insights inform the design of an inexpensive, real-time ADCS test bench tailored to CubeSat-scale constraints.
Having seen how both large programmes and student teams have approached ADCS validation, the following section distils those lessons into formal test-modality definitions. In particular, it contrasts variable-step ('non-real-time') runs with fixed-step, real-time execution, and shows how SIL and HIL can interoperate.

2.3 Validation Approaches

This section reviews the principal validation approaches available for ADCS development, comparing their capabilities, resource requirements and suitability for education-based CubeSat projects. Emphasis is placed on distinguishing non real-time versus real-time execution, and on exploring how SIL and HIL methods may be combined into a cost-effective yet flight-representative test bench.

2.3.1 CubeSat Trends and Reference Configuration

Recent surveys of CubeSat missions show that over 40 percent use 3 U form factors, largely because this size balances payload capacity, power budget and attitude-control performance under modest mass and volume constraints (Polat et al., 2016). [11] Most fly in 400–600 km SSO or LEO and employ two primary control modes, magnetic-rate detumbling and Sun- or Earth-pointing using low-cost magnetometers, magnetorquers and Sun sensors. These trends directly inform the reference design chosen for this study, ensuring that the test bench is representative of the majority of education-based CubeSat missions.

2.3.2 Non Real-Time Simulation

In non real-time environments, control algorithms execute with variable time steps, often within high-fidelity desktop simulators (MATLAB/Simulink, AGI STK). These platforms offer rich modelling of orbital dynamics, environmental perturbations and sensor imperfections, facilitating rapid prototyping and parameter sweeps without concern for strict timing constraints.

[11] https://www.nanosats.eu/ accessed on 19/12/23

However, such setups cannot reveal latency-related faults, scheduling jitter or worst-case execution times that commonly occur on embedded processors. Consequently, a purely non real-time approach may leave time-critical issues undetected until late in development.

2.3.3 Real-Time Execution

Real-time testing enforces fixed-step execution of the ADCS code on target hardware or deterministic operating systems. By matching the control loop frequency to flight-like rates, real-time execution exposes timing faults, callback overruns and interface latencies that would be invisible in variable-step runs.

2.3.4 Software-in-the-Loop (SIL)

Software-in-the-Loop testing embeds the control software under development within a real-time simulation environment, replacing physical hardware with high-fidelity models. Unlike HIL, SIL requires no physical actuators or sensors, which dramatically lowers cost and increases flexibility for early-stage validation. By executing control algorithms in a fixed-step real-time operating system and interfacing with external simulation packages, SIL enables rapid prototyping and performance assessment under realistic, yet fully virtual, conditions. A representative aerospace example is the real-time SIL framework described by **?**, where electric-power-steering and motor-drive controllers were validated at millisecond resolution on a PC cluster by interfacing Simulink code with high-fidelity plant models.

2.3.5 Hardware-in-the-Loop (HIL)

HIL testing integrates real hardware components with virtual simulation models in a closed-loop environment to replicate operational conditions. This approach enables safe validation of embedded control systems without risking damage to physical equipment. HIL is widely used in aerospace, automotive and power-electronics industries to verify system performance, reduce development costs and ensure reliability before full-scale implementation Mihalic et al. (2022).

Several aerospace HIL setups have been reported. Formation-flying and object-capture rigs often employ large moving platforms. For example, Carignan et al. (2022) describes a ground-based robotic testbed using a Rotopod R2000 platform, providing full 6-DOF motion, and Motoman SIA50D robots equipped with custom Schunk grippers (Figure 2.11). While these platforms demonstrate realistic dynamics, their size and cost are prohibitive for university budgets.

A more compact formation-flying HIL testbed uses wheeled robots to achieve 2-DOF motion for formation control experiments (Scharnagl and Schilling, 2016). Although financially accessible, it does not match our reference mission, which should be a simple CubeSat mission.

For sensor validation, Farissi et al. (2019) employs a Helmholtz cage, similar to ITU-PSAT II, to recreate the geomagnetic field and drive real-time dynamics in Simulink. This method is directly relevant but requires substantial laboratory space.

Finally, Haraguchi (2024) tests a star tracker using a Raspberry Pi-driven display that shows simulated star fields (Figure 2.12). This compact, low-cost arrangement closely resembles the HIL architecture intended for development and will be evaluated further.

Several compact HIL setups are directly relevant to CubeSat ADCS:

- A three-axis Helmholtz cage drives magnetometers and magnetorquers with controlled field vectors (Farissi et al., 2019).

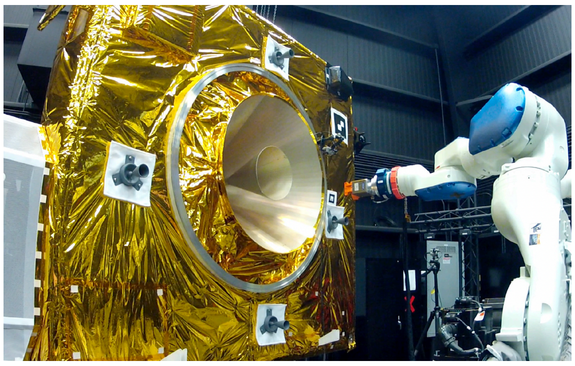


Figure 2.11: Ground-based robotic formation-flying testbed (Carignan et al., 2022)

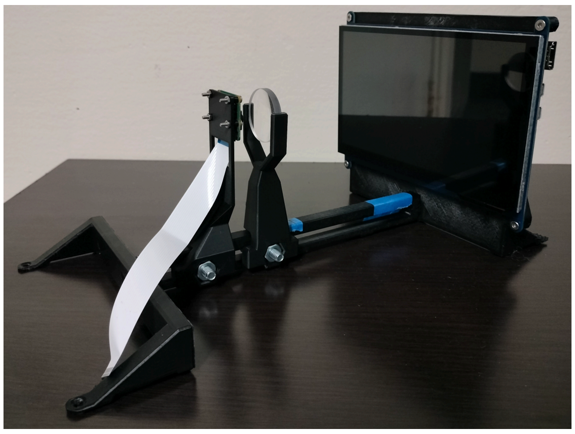


Figure 2.12: Compact star-tracker HIL setup (Haraguchi, 2024)

- A Raspberry Pi-driven display emulates Sun-sensor or star-tracker inputs in real time (Figure 2.12) (Haraguchi, 2024).

These fixtures attach to EuroSim without recompilation, allowing seamless progression from SIL to HIL and revealing quantisation effects, misalignment errors and interface mismatches that SIL cannot capture.

2.4 Chapter Summary and Transition

This chapter has reviewed ADCS development and validation across three scales. Agency-class missions demonstrate the benefits of real-time, fixed-step simulation, flat-sat layouts for early hardware integration and seamless progression from SIL to HIL. University-led CubeSat programmes highlight rapid prototyping in MATLAB/Simulink, magnetic-rate detumbling and simple pointing loops, together with compact HIL fixtures such as Helmholtz cages and air-bearing tables. Finally, the two principal validation approaches, SILT and HILT, have been compared in terms of fidelity, cost and educational accessibility.

In Chapter 3, these lessons will inform the design of a generic CubeSat ADCS test bench. The methodology for selecting a representative mission profile, defining performance requirements and implementing the SIL framework on EuroSim will be presented. Although initial emphasis will be placed on software-in-the-loop execution, the architecture will be structured to allow straightforward extension to hardware-in-the-loop in future work.

CHAPTER 3

Design context and Methodology

The preceding survey revealed that most university CubeSats employ 3 U platforms, B-dot detumbling and simple pointing modes, yet few teams have a repeatable, real-time test infrastructure. This chapter, therefore, defines a mission and toolchain that balances realism, cost and educational accessibility.

3.1 Reference Mission

Considering the test bed should be educational, low-cost, and durable, a SILT setup with HILT and real-time capabilities was chosen. This will be implemented using EuroSim and MATLAB/Simulink, as both are available under educational licenses, and support was available for their implementation.

Based on the preceding analysis, an Earth observation mission using a 3U CubeSat configuration was selected as the reference mission. From the previously analysed missions in Section 2.1, the 1KUNS-PF and Taifi-1 missions were combined. Taifa-1's 3 U form factor and well-characterised Sun-sensor suite define our hardware envelope, while 1KUNS-PF's extensive open-source requirements informed our performance benchmarks.

A circular low Earth orbit (LEO) at an altitude of 500km was selected as the reference orbit. Mission specifications are summarised in Table 3.1. The hardware configuration is primarily based on Taifi-1 and the SPS mission, with some variations to better suit the research objectives. SPS provided simulated hardware and sensor specifications, which were particularly helpful during the development phase. A summary of these specifications is given in Table 3.3. The placement of hardware components is shown in Figure 3.1, with detailed positions listed in Table 3.2.

To align with standard practices observed in CubeSat missions, similar control modes were applied. Starting with a detumbling mode, followed by a pointing or orientation mode and orbit manoeuvring modes. As this thesis is for demonstration purposes, focusing on orientation (angular velocity and quaternions), the following modes were chosen. This research focuses on the first two attitude control modes: the detumbling mode immediately after deployment from the launcher, and the Sun acquisition mode. Upon release, the satellite is expected to rotate

Table 3.1: Reference missions specifications

Parameter	Specification
Orbit	LEO of 500 km
Size	3U – 100 × 100 × 340.5 [mm]
Weight	5 [kg]
Mission Objective	Earth observation
Sensors	Six Sun sensors, a Gyroscope, and a Magnetometer
Actuators	A Magnetorquer and a Reaction Wheel
Control Modes	Detumbling (passive) and Sun acquisition (active)

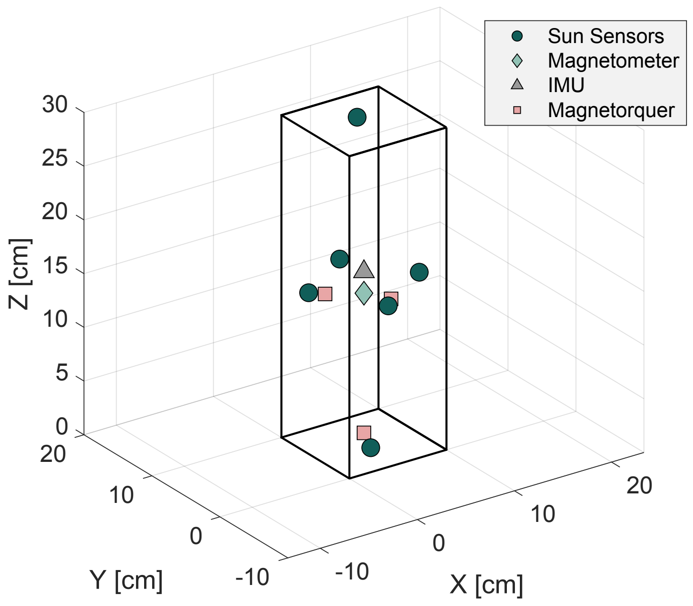


Figure 3.1: Hardware Positions

at a maximum rate of $10\,^\circ\,\mathrm{s}^{-1}$. The detumbling process will use magnetorquers in combination with magnetometer data to reduce this to below $3\,^\circ\,\mathrm{s}^{-1}$, at which point the Sun acquisition mode will be initiated.

In the Sun acquisition mode, six Sun sensors and a gyroscope will be used to determine the Sun's position and update the attitude estimate. The attitude control system includes a magnetometer, magnetorquers, reaction wheels, Sun sensors, and a gyroscope. These components will be modelled in software for simulation purposes, as the control system will initially be verified in a purely simulated environment.

The simulation focuses on quaternion-based attitude representation and angular velocity tracking. Linear position and velocity will not be modelled, as they are not critical for the initial control objectives. The required attitude control specifications are outlined in Table 3.1.

[1] https://satsearch.co/products/bradford-mini-fine-sun-sensor accessed 15-03-25
[2] https://satsearch.co/products/newspace-systems-ngps-01-422-gps-receiver accessed 15-01-24
[3] https://satsearch.co/products/newspace-systems-nmrm-bn25o485-magnetometer accessed 15-01-24
[4] https://satsearch.co/products/newspace-systems-nctr-m003-magnetorquer-rod accessed 15-01-24
[5] https://satsearch.co/products/newspace-systems-nrwa-t065-reaction-wheel accessed 15-01-24
[6] https://satsearch.co/products/newspace-systems-nsgy-001-stellar-gyro accessed 15-01-24
[7] https://satsearch.co/products/iactec-space-drago-2 accessed 15-01-24

Table 3.2: Sensor and Actuator Positions for 3U CubeSat where (0,0,0) is a corner where the z-axis points along the long axis.

Component	X (cm)	Y (cm)	Z (cm)
Sun Sensor 1 (-Y)	4	0	15
Sun Sensor 2 (+Y)	6	10	15
Sun Sensor 3 (-X)	0	6	15
Sun Sensor 4 (+X)	10	4	15
Sun Sensor 5 (-Z)	5	4	0
Sun Sensor 6 (+Z)	5	6	30
Magnetometer (Center)	5	5	14
IMU (Center)	5	5	16
GPS Antenna (+Z)	4	5	30
X-Axis Magnetorquer	5	1	15
Y-Axis Magnetorquer	1	5	15
Z-Axis Magnetorquer	5	5	1

Table 3.3: Hardware specifications.

Sensor	Mass [g]	Size (L×W×H) [mm]	Key Parameters
Mini-FSS Fine Sun Sensor[1]	<50	50×46×17	FOV ±30° (X, Y) Noise 0.0333° (1σ) Quantisation 0.0557° Bias 0.01°
NMRM-Bn25o485 Magnetometer[3]	<85	99×43×17	Range ±100 μT Noise 5×10^{-11} T/$\sqrt{\text{Hz}}$ Scale Factor 150 ppm Quantisation 0.0488 μT Drift 1 nT/°C
NCTR-M003 Magnetorquer[4]	<30	72×15×13	Max Moment 400 Am^2 Min Moment 0.4889 Am^2 Resolution 0.2 Am^2 Time Constant 0.2216 s Bias 2 Am^2
NRWA-T065 Reaction Wheel[5]	1550	102×102×105	Max Torque 0.075 Nm Time Delay 0.02 s Resolution 2×10^{-5} Nm
NSGY-001 Gyro[6]	<55	37×35.5×49	Range ±30°/s Noise 1.7×10^{-5} rad/$\sqrt{\text{s}}$ Scale Factor 500 ppm Quantisation 0.0146°/s Drift 1×10^{-7} rad/s

Requirements

The requirements for the overall simulation and GNC system are given below. Table 3.4 describes the requirements which are given for the hardware which is simulated, while Table 3.5 describes the requirements for the controller of the simulation. The values in the requirements were found using research into previous missions and the specifications of the hardware used. This list will be referred to at different points in the study to justify the completeness of different components of the mission.

Table 3.4: Table of hardware requirements for reference mission. Sun Sensor (SS), Magnetometer (MM), Reaction Wheels (RW), Magnetorquer (MT) and Gyroscope (GYR).

ID	Requirement
SS01	The Sun Sensor shall provide the Sun vector every 0.1 seconds.
SS02	The Sun Sensor shall provide the Sun position with an accuracy of 1° (3σ).
SS03	The Sun Sensor shall locate the Sun's position within 10 minutes after activation.
SS04	The Sun Sensor shall have a field of view (FOV) of $30^\circ \times 30^\circ$.
SS05	The Sun Sensor noise shall be less than 0.05° (1σ).
MM01	The Magnetometer shall provide the magnetic field vector every 0.5 seconds.
MM02	The Magnetometer shall achieve measurement accuracy better than 5 nT (1σ).
GYR01	The Gyroscope shall have sensitivity better than 0.01°/s (36 arcsec/s).
GYR02	The Gyroscope shall provide measurements every 0.01 seconds (100 Hz).
RW01	The Reaction Wheels shall provide at least 0.05 Nm at 6000 rpm.
RW02	The Reaction Wheels shall not have drift exceeding 0.01°/s.
MT01	The Magnetorquer shall provide at least 0.005 Nm continuous torque.

Table 3.5: Table of control mode requirements for the reference mission. Detumbling mode (DT), Sun acquisition mode (SA), and Guidance, Navigation, and Control (GNC).

ID	Requirement
DT01	The detumbling of the satellite shall be completed within 2 orbits.
DT02	The angular rate shall be reduced below 0.5°/s around the x-axis at the end of the manoeuvre.
DT03	The angular rate shall be reduced below 0.5°/s around the y-axis at the end of the manoeuvre.
DT04	The angular rate shall be reduced below 0.5°/s around the z-axis at the end of the manoeuvre.
SA01	The z-axis of the satellite shall align with the Sun vector within 1 orbit (90 minutes).
SA02	The z-axis of the satellite shall align with the Sun vector within 1.5 hours after detumbling.
SA03	The pointing accuracy during sun acquisition shall be within 5°.
SA04	The sun shall be found within 1 orbit (90 minutes) after activation.
GNC01	The control loop shall operate at 10 Hz.
GNC02	The 5% settling time shall be less than 10 minutes.
GNC03	The 2% settling time shall be less than 15 minutes.
GNC04	The rise time shall be no more than 5 minutes.
GNC05	The overshoot shall be less than 10%.

3.2 Test Bed Configuration

Previous sections have highlighted how real-time simulation and HIL and SIL methods have been employed to validate ADCS in both large-scale and CubeSat missions. While missions such as Gaia and Herschel-Planck employed highly customised and resource-intensive real-time simulation environments (e.g., using EuroSim and bespoke Software Checkout Equipment), CubeSat missions like MOVE-II and ITU-PSAT II demonstrated that lower-cost, adaptable test environments can still provide critical validation for ADCS performance, particularly in educational settings. However, many educational missions continue to suffer from high failure rates due to limited access to comprehensive, real-time test facilities.

To address these challenges within the context of this research, a real-time CubeSat ADCS test bench is developed based on the following methodology:

- **MATLAB/Simulink** is selected as the primary development environment for modelling, simulation, and controller development. This decision leverages the availability of university-wide licences and ensures that the tools used are widely accessible to educational institutions.
- The **Generic Guidance Navigation and Control Simulator (GGNCSim)** and the **Generic Rendezvous And Docking Simulator (GRADS) library** are utilised for the sake of revising existing software for model development. These libraries provide a set of modular, pre-validated Simulink blocks for sensors, actuators, propagation, environment modelling, and utility functions. Originally developed for GNC system testing in rendezvous and docking missions, their modularity and flexibility allow efficient adaptation for CubeSat ADCS simulations, significantly reducing development time and ensuring reliable model behaviour.
- **EuroSim** is integrated to provide a flexible real-time simulation environment. EuroSim's compatibility with MATLAB/Simulink models and its real-time scheduling capabilities make it an ideal platform for HIL implementation and external hardware interfacing. This software is also available under an educational license.
- A **Raspberry Pi** is selected as the embedded platform to run real-time simulations. The Raspberry Pi provides an affordable, compact, and accessible platform capable of supporting real-time execution of models and controller code, aligning with the project's objectives of cost-efficiency and educational accessibility.

This combination of tools and hardware enables the creation of a real-time test bench that is technically robust and educationally accessible. The requirements listed in Table 9.6 define the necessary functional, hardware, software, usability, and educational constraints for the development of the real-time CubeSat ADCS test bench.

3.3 Research Questions

Reflecting on the research of Chapter 2 and the discussion above, the following research questions are given as a guide to the thesis. This thesis focuses on improving the accessibility of CubeSat Attitude Determination and Control Systems (ADCS) testing while using low-cost hardware. A particular emphasis is placed on evaluating how real-time simulation methods can be leveraged to better represent system behaviour during development and prototyping, compared to traditional non-real-time simulation methods. Furthermore, the study considers

Table 3.6: Requirements for the real-time CubeSat ADCS test bench.

ID	Requirement
SIM01	The simulator shall simulate CubeSat attitude dynamics in real-time.
SIM02	The simulator shall support real-time interaction with CubeSat ADCS hardware under test.
SIM03	The simulator shall support real-time interaction with CubeSat ADCS software under test.
SIM04	The test bench shall use commercially available off-the-shelf (COTS) components where possible.
SIM05	The simulator shall be based on open-source or educational-licensed software tools.
SIM06	The simulator limitations and common errors will be highlighted to allow varying levels of students to use and understand the implementation process.
SIM07	The total test bench cost shall not exceed €2000.

how testing environments can be made more accessible for educational and research-focused CubeSat missions, where resources are often limited.

Research Questions

Main Research Question:
To what extent can low-cost hardware and simulation tools support real-time testing of CubeSat ADCS algorithms in an educational context?

Sub-Research Questions:

- What components are required to develop a functional ADCS simulation and test environment?
- What are the benefits and limitations of using simulated environments for testing ADCS algorithms without access to flight hardware?
- Can the proposed setup incorporate hardware-in-the-loop (HIL) testing?
- How can a real-time test bench be designed to meet the needs of student-led and institution-based CubeSat projects?

The **Main Research Question** centres on whether a combination of low-cost hardware and openly available simulation tools can deliver a *real-time* test bench suitable for validating CubeSat Attitude-Determination and Control Systems (ADCS) in an academic setting. University missions often suffer ADCS failures because time-critical software is tested only in slow, offline simulations or, even worse, when in orbit. By shifting validation into an affordable, real-time environment, this thesis aims to raise the capabilities of ground testing while keeping costs within reach of student teams.

The four **Sub-Research Questions** provide a step-by-step path toward that goal:

(SQ1) By first designing a complete ADCS for the reference CubeSat, the thesis dissects the subsystem into sensors, actuators, estimation algorithms, and control laws. From this bottom-up exercise, it extracts the *minimum* set of models, interfaces, and computing resources a test bench must offer so that future teams can iterate on their own ADCS designs with confidence.

(SQ2) Building on those components, the work evaluates pure simulation as an early-stage validation tool, fast, risk-free, and inexpensive, yet inevitably limited by modelling accuracy. Quantifying the trade-offs between speed and realism clarifies when software-only testing suffices and when additional resources are required.

(SQ3) The study then investigates extending the same architecture to HITL, analysing data-rate constraints, interface latencies, and synchronisation demands. Demonstrating even a partial HIL loop shows how the bench can bridge the gap from laboratory models to flight-like performance.

(SQ4) Finally, the thesis turns to practical deployment: cost, licence choices, documentation, and training material tailored to student teams with mixed experience levels. Emphasis is placed on usability and robustness so that the bench remains effective even when operated by non-expert users under academic time pressures.

Together, these four strands form a coherent roadmap: design and decompose an ADCS, test its limits in simulation, extend the bench toward HIL, and package the result for widespread educational use. Each subsequent chapter of the thesis is mapped explicitly to one or more of these sub-questions.

3.4 Thesis Roadmap

The remainder of this thesis proceeds in six focused stages, each building on the last to deliver a complete, low-cost, real-time ADCS test bench for a 3 U CubeSat:

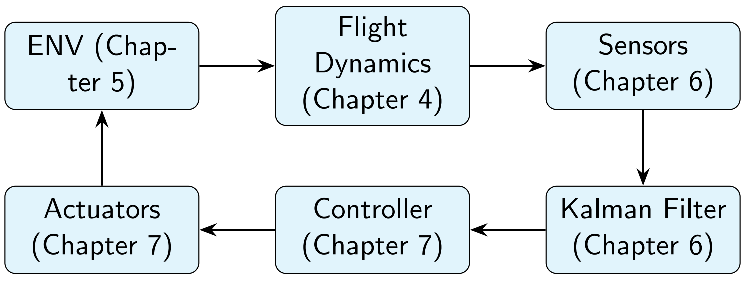


Figure 3.2: Flowchart illustrating the system with ENV - Environment models, Flight Dynamics, Sensors, Kalman Filter, Controller, and Actuators

Chapter 4 – Flight Dynamics. This chapter establishes the kinematic and dynamic foundations required to describe CubeSat motion. It introduces the key state-variable representations (Cartesian coordinates, orbital elements, Euler angles and quaternions), defines the body-fixed, inertial and local frames, and derives the rigid-body equations of motion that govern rotational dynamics.

Chapter 5 – Space Environment. Here the principal external disturbances are developed: the Earth's gravity field (including higher-order harmonics), geomagnetic field, aerodynamic drag and solar-radiation pressure. Each model is presented in theory, then its implementation in the GRADS library is validated via acceptance tests.

Chapter 6 – Navigation. This chapter describes how the simulated spacecraft "sees" its environment. Detailed models of sun sensors, magnetometers, gyroscopes and the IMU are introduced, including noise and bias characteristics. The extended Kalman filter architecture is then developed, showing how these measurements are fused to estimate attitude (quaternion) and angular velocity.

Chapter 7 – Control. With state estimates in hand, the ADCS control laws are presented. The B-dot algorithm for magnetorquer-only detumbling is derived, followed by a proportional–derivative controller for Sun-acquisition. Stability, performance and switching logic between modes are discussed.

Chapter 8 – Functional Simulator. The GRADS-based Simulink simulator is assembled here. Its software architecture, solver selection and component acceptance tests are described in detail. This functional (non-real-time) bench verifies that the combined environment, navigation and control models meet the mission requirements.

Chapter 9 – Real-Time Implementation. The final test-bench is realised in EuroSim on a Raspberry Pi. This chapter covers MATLAB code generation, EuroSim model integration ("dos and don'ts"), hardware interfaces and end-to-end acceptance tests that run the detumbling and Sun-acquisition sequences in true real-time.

Chapter 10 – Conclusion. The thesis closes by revisiting the research questions, summarising key contributions, reflecting on limitations and outlining future extensions toward full HILT validation.

CHAPTER 4

Flight Dynamics

For an Attitude Determination and Control System (ADCS) to operate reliably, the underlying dynamics model must reproduce a satellite's translational and rotational motion with sufficient accuracy. This chapter establishes the mathematical foundation used by the CubeSat simulator. It first clarifies how a satellite's state is numerically represented, ranging from Cartesian position and velocity to attitude descriptions such as Euler angles and quaternions, highlighting why quaternions are ultimately preferred for real-time attitude propagation. The analysis then explains the role of reference frames: an inertial frame for orbital motion, a body-fixed frame for onboard hardware, and intermediate frames such as LVLH for disturbance modelling. Transformations between these frames are essential, for example, when converting an inertial magnetic-field vector into body coordinates for a magnetometer model. Finally, Newton–Euler translation and rigid-body rotation are combined into a six-degree-of-freedom equation set that drives the simulation core. Together, these elements provide the kinematic and dynamic context required for the sensor models, Kalman filters, and control laws developed in later chapters.

As stated earlier, the simulator does not model position and velocity in full detail, focusing instead on attitude representation; however, these elements are still briefly discussed in this chapter for completeness.

4.1 State variables

This section will discuss the various variables which are used in the development and creation of the simulator.

4.1.1 Cartesian Coordinates

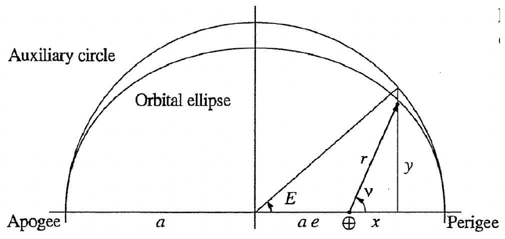
Cartesian coordinates represent an object's position and velocity using six parameters: three for position (x, y, z) and three for velocity $(\dot{x}, \dot{y}, \dot{z})$. The x, y, and z axes are mutually perpendicular and intersect at the origin, which is typically defined relative to an inertial reference frame.

This system allows any point in space to be uniquely identified by its intersection with planes perpendicular to each axis. This will not be used in the simulation set-up as it relates to position and velocity. However, it is used to describe the behaviour of the satellite in a later chapter. Therefore, a brief description was included.

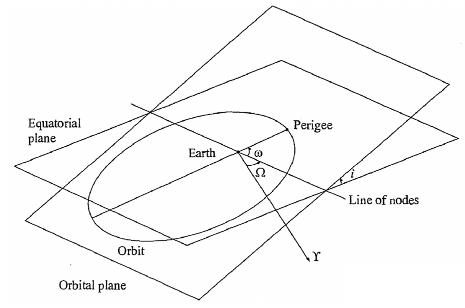
4.1.2 Orbital elements

The orbital elements can be used when describing the position and direction of the orbit. Again, this set of state variables will not be used in the simulator but is used in later chapters for discussion of results. Consider an elliptical orbit, Figure 4.1a and Figure 4.1b show the variables used to describe this orbit. The size is determined by the semi-major axis a, and the shape is determined by the eccentricity e.

The inclination angle i is between the orbital plane and the equator. An angle of more than 90 degrees means the satellite is in a retrograde orbit, and it rotates opposite that of Earth's rotation. The right ascension of the ascending node Ω is the angle between the vernal



(a) Orbital elements 1 Montenbruck and Gill (2001)



(b) Orbital elements 2 Montenbruck and Gill (2001)

Figure 4.1: Comparison of orbital elements

equinox and the location where the satellite crosses the equator from south to north. Taking the angle between the direction of the ascending node and perigee gives the argument of perigee ω(Montenbruck and Gill, 2001). To determine the location of a satellite at a given time, a sixth parameter is needed. This could be the time of pericenter passage τ, or most commonly, the mean anomaly at t_0 is used, M_0. This is defined as (Mooij, 2022b):

$$M_0 = n(t_0 - \tau) \tag{4.1}$$

where

$$n = \sqrt{\frac{\mu^3}{a}} \tag{4.2}$$

This means that the mean anomaly can be expressed with:

$$M = M_0 + n(t_0 - \tau) = n(t - \tau) \tag{4.3}$$

4.1.3 Euler Angles

Euler angles represent a spacecraft's orientation through three sequential rotations about its body-fixed axes: roll (ϕ), pitch (θ), and yaw (ψ), corresponding to rotations about the x, y, and z axes, respectively (Wertz, 1980). Although intuitive and easy to visualise, they suffer from singularities such as gimbal lock, where two axes align and one rotational degree of freedom is lost. Gimbal lock can be mitigated by altering the rotation sequence at the limits of Euler angles, but this introduces additional computational overhead, and residual numerical error may still compromise detumbling performance. Because the control modes used in this thesis, particularly B-dot detumbling, require continuous, singularity-free attitude propagation, Euler angles are retained solely for plotting results (using a 1-2-3 sequence).

4.1.4 Quaternions

Quaternions, also known as Euler symmetric parameters, provide a four-parameter attitude description consisting of a scalar part and a three-component vector:

$$\mathbf{q} = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{pmatrix}, \qquad q_1^2 + q_2^2 + q_3^2 + q_4^2 = 1. \tag{4.4}$$

They are free from kinematic singularities, require only basic arithmetic, and permit smooth interpolation of orientations, making them well suited to real-time simulation. Classical Rodrigues parameters and Modified Rodrigues Parameters were also evaluated; both offer three-parameter descriptions with reduced singular-axis issues, yet each retains a singularity at 180° and necessitates parameter switching or shadow-set logic. Given the modest computational load of quaternions on the target hardware and their lack of singularities over the full attitude range, quaternions were adopted as the primary internal representation.

4.1.5 Angular Velocity

In addition to position and orientation, which are not the main focus of this study, the rotational motion of the spacecraft is characterised by its angular velocity vector $\boldsymbol{\omega}$. This vector represents the rate of rotation of the body frame relative to the inertial frame, typically expressed in the body-fixed reference frame. Angular velocity is crucial for describing the spacecraft's rotational dynamics and is used directly in both the equations of motion and attitude control algorithms. It complements the attitude representation (Euler angles or quaternions) in defining the complete rotational state of the spacecraft.

4.2 Reference frames

Modelling a satellite's orientation and rotation in space requires a thorough understanding of the various reference frames involved. With the correct implementation and theoretical knowledge, the simulation can better represent the mission conditions and successfully be used to validate the control algorithms. Different systems and system performances require different origins of their reference frame. This section will describe three reference frames and their use cases. First is the body fixed frame, then the Earth-centred inertial frame, followed by the instrument frame.

4.2.1 Earth-Centred Inertial Frame (ECI) $(\mathcal{F}_\mathrm{I})$

The Earth-Centred Inertial (ECI) frame provides the inertial reference in which Newton's translational and rotational equations of motion are formulated. It is quasi-fixed with respect to the distant stars; that is, it does not co-rotate with the Earth and is therefore convenient for expressing forces, torques, and absolute attitude.

The frame's origin is at the Earth's centre of mass. Its z-axis is directed towards the mean north-celestial pole, the x-axis towards the mean vernal equinox, and the y-axis completes a right-handed triad. In this thesis, the ECI axes correspond to the conventional J2000 system, defined by the mean equator and mean equinox at noon on 1 January 2000 (Mooij, 2022a; Vallado and McClain, 2013). Although a spacecraft's attitude is ultimately expressed with respect to a body-fixed frame, the ECI frame supplies the non-rotating reference against which that attitude is measured or propagated.

4.2.2 Hardware Frame $(\mathcal{F}_\mathrm{H})$

The satellite carries several instruments—sensors and actuators—each with its own local reference frame. In the simulator, every sensor reading and actuator command is generated in its native frame and then transformed into a common frame before being used in the state-estimation process. The simulated sensors comprise six Sun sensors, an IMU, a GPS receiver, and a magnetometer; the simulated actuators comprise a set of reaction wheels and a three-axis magnetorquer. Scientific payloads are not modelled and are therefore excluded. The spatial locations of all simulated hardware components are listed in Table 3.2.

4.2.3 Body fixed ($\mathcal{F}_{\mathcal{B}}$)

This reference frame is used to describe the orientation of the sensor actuators and other instruments on board with respect to the satellite's body. The origin lies in the centre of mass of the body, and the three axes coincide with the body's geometry. This reference frame describes the angular rotations represented by Euler angles, roll ϕ, pitch θ and yaw ψ which show the rotations around the x, y and z axes respectively (Wertz, 1980).

4.2.4 Vertical Frame ($\mathcal{F}_{\mathrm{V}}$)

The vertical frame, often called the local-horizontal local-vertical (LHLV) frame, is commonly used to express gravitational acceleration acting on a spacecraft. It treats the Earth as a perfect sphere: the z-axis points radially towards the planet's geometric centre, while the x-axis lies in the local meridian plane and the y-axis completes the right-hand triad.

Because the Earth is an oblate spheroid, the true gravity vector is not perfectly radial; small north-south and east-west components arise in addition to the dominant radial term. These departures are usually negligible for CubeSat-scale analyses but should be noted when high-precision modelling is required.

4.3 Reference Frame Transformations

To move between reference frames Mooij (2022a), appropriate transformations must be applied. For instance, while the equations of motion are typically expressed in the inertial frame, sensor measurements are provided in the hardware (or body) frame. To propagate the motion using sensor data, it must first be transformed into the appropriate reference frame. This section presents several commonly used transformation techniques.

A basic transformation between two reference frames is given in Equation (4.5), where $\mathbf{T}$ is the translation vector between the origins of frames $\mathcal{F}_A$ and $\mathcal{F}_B$, and $\mathbf{C}_{B,A}$ is the direction cosine matrix (DCM) representing the rotation from $\mathcal{F}_A$ to $\mathcal{F}_B$:

$$\mathbf{v}_B = \mathbf{T} + \mathbf{C}_{B,A}\mathbf{v}_A \tag{4.5}$$

A rotation about a single axis is referred to as a unit-axis rotation. These are defined to be positive according to the right-hand rule. The matrices in Equation (4.6), Equation (4.7), and Equation (4.8) represent rotations by an arbitrary angle θ about the x, y, and z axes, respectively:

$$\mathbf{C}_{\mathbf{X}}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \tag{4.6}$$

$$\mathbf{C}_{\mathbf{Y}}(\theta) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \tag{4.7}$$

$$\mathbf{C}_{\mathbf{Z}}(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.8}$$

The order of rotations is critical and non-commutative: for example, an *x*-*y*-*z* rotation sequence is not equivalent to a *z*-*y*-*x* sequence. Each axis rotation is orthonormal, and a composite rotation in *x*-*y*-*z* order is given by:

$$\mathbf{C_{XYZ}}(\theta) = \mathbf{C_Z}(\theta)\mathbf{C_Y}(\theta)\mathbf{C_X}(\theta) \tag{4.9}$$

When Euler angles (ϕ, θ, ψ) are used (typically representing roll, pitch, and yaw), the rotation matrix becomes:

$$\mathbf{R} = \begin{bmatrix} \cos\psi\cos\theta & \cos\theta\sin\psi & -\sin\theta \\ \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \sin\phi\cos\theta \\ \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi & \cos\phi\cos\theta \end{bmatrix} \tag{4.10}$$

Alternatively, rotations can be efficiently represented using quaternions, which define a rotation about a unit vector. Quaternion-based transformations avoid singularities and reduce computational load by eliminating the need for multiple matrix multiplications. The DCM derived from a unit quaternion $\mathbf{q} = [q_1, q_2, q_3, q_4]$ is shown in Equation 4.11, where $\mathbf{v}_I$ is the vector in the inertial frame and $\mathbf{v}_B$ is the transformed vector in the body frame:

$$\mathbf{C}_{I,B} = \begin{bmatrix} q_1^2 + q_4^2 - q_2^2 - q_3^2 & 2(q_1q_2 + q_3q_4) & 2(q_1q_3 - q_2q_4) \\ 2(q_1q_2 - q_3q_4) & q_2^2 + q_4^2 - q_1^2 - q_3^2 & 2(q_2q_3 + q_1q_4) \\ 2(q_1q_3 + q_2q_4) & 2(q_2q_3 - q_1q_4) & q_3^2 + q_4^2 - q_1^2 - q_2^2 \end{bmatrix} \tag{4.11}$$

4.4 Equations of Motion

The simulator models the satellite as a rigid body with fixed mass properties; internal mass motion and orbital translation are not considered here. All vectors are expressed in the body frame $\mathcal{F}_B$ unless stated otherwise.

4.4.1 Rotational Dynamics

Euler's equation gives the external moment

$$\mathbf{M} = \mathbf{I}\,\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \mathbf{I}\,\boldsymbol{\omega}, \tag{4.12}$$

where $\mathbf{I}$ is the inertia matrix and $\boldsymbol{\omega} = [\omega_1\,\omega_2\,\omega_3]^\mathsf{T}$ is the body-frame angular rate with respect to the local-vertical frame $\mathcal{F}_A$. Gravity-gradient, control, and disturbance torques are appended directly to Equation (4.12) when required.

4.4.2 Attitude Kinematics

Attitude is propagated with the unit quaternion $\mathbf{q} = [q_1\,q_2\,q_3\,q_4]^\mathsf{T}$. Its time derivative is

$$\begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \end{pmatrix} = \frac{1}{2} \begin{bmatrix} q_4 & -q_3 & q_2 \\ q_3 & q_4 & -q_1 \\ -q_2 & q_1 & q_4 \\ -q_1 & -q_2 & -q_3 \end{bmatrix} \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix}. \tag{4.13}$$

Because the local-vertical frame rotates at the orbital rate n about its transverse axis $\mathbf{a}_2$, the angular rate used in Equation (6.22) is

$$\boldsymbol{\omega}_{B/A} = \boldsymbol{\omega}_{B/N} + n\,\mathbf{a}_2. \tag{4.14}$$

Equations (4.12), (4.14) and (6.22) form the rotational state model required by the Extended Kalman Filter and by the pointing and detumbling controllers in later chapters.

CHAPTER 5

Space Environment

The space environment introduces a range of perturbations that significantly influence the motion and attitude of a satellite in orbit. This chapter discusses the primary environmental disturbances acting on the CubeSat, including atmospheric drag, solar radiation pressure, Earth's gravitational anomalies, and magnetic field interactions.

Each perturbation is mathematically characterised, and an estimation of its expected magnitude is provided. The models used to represent these effects are sourced from a simulation library, but dedicated acceptance tests are performed to verify that each model behaves as intended. The model library is called GRADS, and more information can be found in Chapter 8. The verified environmental models form a critical part of the simulation environment used for the development and validation of the CubeSat ADCS system.

5.1 Gravitational Field

To simulate the gravitational force acting on a satellite in orbit around Earth, Newton's law of gravitation can be used as shown in Equation 5.1 where **g** is the gravitational acceleration, m is the mass of the satellite, R is the distance from the satellite to the centre of Earth and μ_E is the gravitational parameter of Earth, which is $\mu_E = 3.9860047 \cdot 10^4 \text{m}^3/\text{s}^2$. $\hat{\mathbf{r}}$ is the normalised position vector. This is also referred to as the central field model because it describes the gravitational field around a central, spherically symmetric mass distribution. Assuming that the Earth is spherically symmetric simplifies calculations and allows the gravitational field to be expressed using radial coordinates (OpenCourseWare, 2004). The inverse-square law states that the gravitational force decreases proportionally to the square of the distance from the central mass.

$$\mathbf{F} = m\mathbf{g} = m\frac{\mu_E}{R^2}\hat{\mathbf{r}} \tag{5.1}$$

The gravitational acceleration **g**, can be simulated as a constant or varying with radius. One method for near-spherical bodies is the spherical harmonics model. This model describes the gravitational potential, which is the potential energy per unit mass and quantifies the work needed to move a unit mass from a reference point (infinity) to another point without acceleration. This can then be used to find g by finding the gradient of the potential. The gravitational potential is given by Equation 5.2, and the gradient of potential is shown in Equation (5.3) given in the vertical frame ($\mathcal{F}_\mathcal{V}$) where, R, τ and δ are the spherical positions where R is the radial distance, τ is the longitude and δ latitude.

$$U(R,\tau,\delta) = \frac{\mu}{R}\left[1 + \sum_{n=2}^{n_{\max}}\left(\frac{R_e}{R}\right)^n \sum_{m=0}^{n} P_n^m(\sin\delta) \times (C_n^m \cos m\tau + S_n^m \sin m\tau)\right] \tag{5.2}$$

$$\mathbf{g_v} = \begin{pmatrix} g_\delta \\ g_\tau \\ g_R \end{pmatrix} = \begin{pmatrix} g_n \\ g_e \\ g_d \end{pmatrix} = \begin{pmatrix} -\frac{1}{R}\frac{\partial U}{\partial \delta} \\ -\frac{1}{R\cos\delta}\frac{\partial U}{\partial \tau} \\ -\frac{\partial U}{\partial R} \end{pmatrix} \tag{5.3}$$

$$\frac{\partial U}{\partial \delta} = \frac{\mu}{R}\sum_{n=2}^{n_{\max}}\left(\frac{R_e}{R}\right)^n \sum_{m=0}^{n} (C_n^m \cos m\tau + S_n^m \sin m\tau)\frac{\partial P_n^m(\sin\delta)}{\partial \delta} \tag{5.4}$$

$$\frac{\partial U}{\partial \tau} = \frac{\mu}{R} \sum_{n=2}^{n_{\max}} \left(\frac{R_e}{R}\right)^n \sum_{m=0}^{n} m \left(S_n^m \cos m\tau - C_n^m \sin m\tau\right) P_n^m(\sin \delta) \tag{5.5}$$

$$\frac{\partial U}{\partial R} = \frac{\mu}{R^2} \left[1 + \sum_{n=2}^{n_{\max}} \left(\frac{R_e}{R}\right)^n (n+1) \sum_{m=0}^{n} P_n^m(\sin \delta) \left(C_n^m \cos m\tau + S_n^m \sin m\tau\right)\right] \tag{5.6}$$

The normalised Legendre polynomial is denoted as $P_n^m(sin\delta)$ and captures the angular variation in the gravitational potential. The Legendre polynomials are found using a recursive formula.

C_n^m and S_n^m represent the distribution of mass on Earth and how they deviate from a perfect sphere. They are derived from measurements of the gravitational field. Specifically, C_n^m represents the cosine components, while S_n^m represents the sine components.

n is the degree and represents the scale of spherical harmonics. The higher the degree, the more localised the representation of the gravitational field.

m is the order and represents the zonal or longitudinal variations. When $m = 0$, the variations are called zonal harmonics, which represent variations only in latitude and are symmetric around Earth's rotational axis. When $0 < m < n$, these are tesseral harmonics, representing variations in both latitude and longitude . When $m = n$, this is called sectorial harmonics, which capture significant longitudinal variations, found in Vallado and McClain (2013).

The standard gravitational parameter, μ, is taken as 3.986×10^{14} m^{3}s^{-2}, while the Earth's equatorial radius, R_e, is set to 6.3781×10^6 m. These values are provided by the ITG-Grace2010s gravity field model, which is derived from the Gravity Recovery and Climate Experiment (GRACE) mission and describes Earth's gravity field with high precision.

The model includes spherical harmonic coefficients up to degree and order 180, as indicated by the values of $n_{\max} = 180$ and $m_{\max} = 180$. The fully normalised coefficients C_{nm} and S_{nm} are stored as 180×181 matrices. These coefficients represent the cosine and sine components of the gravitational potential, respectively, enabling the modelling of Earth's gravity field.

Given that g is found in the format of Equation (5.3), Equation (5.1) can be used to find the appropriate force. m is the satellite's mass and has been set to 5kg.

5.2 Magnetic Field

The next pertubation to be simulated is the magnetic moments and forces. This affects the satellite in many (small) ways. First, the charged particles cause ionisation in the upper layers of the atmosphere, and therefore, the density causes a drag effect. Charged particles can also interact with the satellite, potentially influencing its electrostatic or electromagnetic environment and tracking and communications. Another is the interaction with the onboard electromagnets, affecting the torque and attitude control (Vallado and McClain, 2013). The magnetic field strength varies from 25,000 nT to 65,000 nT[1].

The Earths magnetic field or geomagnetic field, can be simulated with a spherical harmonics model. It is mainly a magnetic dipole which is tilted relative to Earths rotational axis. It is expressed by a gradient of the scalar potential given in Equation (5.7) where V is a series of spherical harmonics which can be found in Equation (5.8)

$$B = -\Delta V \tag{5.7}$$

[1]https://www.ncei.noaa.gov/products/geomagnetism-frequently-asked-questions, Last accessed 26/12/24

$$V(R,\tau,\delta) = R_{\mathrm{e,m}} \sum_{n=1}^{k} \sum_{m=0}^{n} \left[g_n^m \cos(m\tau) + h_n^m \sin(m\tau)\right] P_n^m(\sin\delta) \tag{5.8}$$

where

$$\cos(m\tau) = \cos\left[(m-1)\tau\right]\cos\tau - \sin\tau \sin\left[(m-1)\tau\right] \tag{5.9}$$

$$\sin(m\tau) = \sin\left[(m-1)\tau\right]\cos\tau + \cos\tau \sin\left[(m-1)\tau\right] \tag{5.10}$$

$R_{\mathrm{e,m}}$ is the equatorial radius of Earth, g_n^m and h_n^m are Gaussian coefficients and R, τ, δ are the radius, longitude and latitude. The Gauss coefficients are determined empirically and are available in the International Geomagnetic Reference Field (IGRF) model, which is updated periodically.

It is assumed that the Legendre coefficients are Schmidt normalised before a set of coefficients are applied. This means they are scaled in a way that ensures orthonormality under integration over a sphere.

$$\int_0^{\pi} \left[P_n^m(\sin\delta)\right]^2 \sin\delta \, d\delta = \frac{2}{2n+1} \tag{5.11}$$

The Gauss functions P'^m_n are related to the Schmidt function with

$$P_n^m = S_{n,m} P_n^{m,n} \tag{5.12}$$

where

$$S_{n,m} = \sqrt{\frac{(2n-m)(n-m)!}{(n+m)(2n-1)!}} \tag{5.13}$$

Evaluating using recursive relations gives:

$$S_{0,0} = 1, \quad S_{n,0} = S_{n-1,0} \quad \forall n \geq 1, \quad S_{n,m} = S_{n,m-1}\sqrt{\frac{(n-m+1)(\delta_m^1+1)}{n+m}} \quad \forall m \geq 1 \tag{5.14}$$

and P'^m_n can be found to be

$$P'_{0,0} = 1, \quad P'_{n,n} = \sin\delta^* P'_{n-1,n-1}, \quad P'_{n,m} = \cos\delta^* P'_{n-1,m} - K_{n,m} P'_{n-2,m} \tag{5.15}$$

where

$$K_{n,m} = \frac{(n-1)^2 - m^2}{(2n-1)(2n-3)} \quad \forall n > 1, \quad K_{n,m} = 0 \quad \text{for } n = 1 \tag{5.16}$$

Now the derivatives can be found as Equation (5.17) to find the radial, latitude, and longitude directions magnetic field components.

$$-\frac{\partial V}{\partial R} = \sum_{n=1}^{n_{\max}} \left(\frac{R_e}{R}\right)^{n+2} (n+1) \sum_{m=0}^{n} \left[g_n^m \cos(m\tau) + h_n^m \sin(m\tau)\right] P'_{n,m}(\cos\delta) \tag{5.17}$$

$$-\frac{1}{R}\frac{\partial V}{\partial \delta*} = \sum_{n=1}^{n_{\max}} \left(\frac{R_e}{R}\right)^{n+2} \sum_{m=0}^{n} \left[g_n^m \cos(m\tau) + h_n^m \sin(m\tau)\right] \frac{\partial P'_{n,m}}{\partial \delta*} \tag{5.18}$$

$$-\frac{1}{R\sin\delta*}\frac{\partial V}{\partial \tau} = -\sum_{n=1}^{n_{\max}} \left(\frac{R_e}{R}\right)^{n+2} \sum_{m=0}^{n} m\left[g_n^m \sin(m\tau) - h_n^m \cos(m\tau)\right] P'_{n,m}(\cos\delta) \tag{5.19}$$

to find

$$\mathbf{B_V} = \begin{pmatrix} B_n & B_e & B_d \end{pmatrix}^T = \begin{pmatrix} -\frac{1}{R}\frac{\partial V}{\partial \delta*} \\ -\frac{1}{R\sin\delta*}\frac{\partial V}{\partial \tau} \\ -\frac{\partial V}{\partial R} \end{pmatrix}^T \tag{5.20}$$

The magnetic field in the veritical frame can then be transformed to the body frame using Equation (5.21) and a magnetic moment can be found as shown in Equation (5.24). $\mathbf{M_{RM}}$ is the residual magnetic moment in Am^2 which can be found using Equation (5.22).

$$\mathbf{B_B} = \mathbf{C_{B,V}}\mathbf{B_V} \tag{5.21}$$

$$M_{RM} = R_e^m \sqrt{\left(g_1^0\right)^2 + \left(g_1^1\right)^2 + \left(h_1^1\right)^2} \tag{5.22}$$

where

$$\mathbf{C_{B,V}} = \mathbf{C_{B,I}}\mathbf{C_{I,V}} \tag{5.23}$$

and

$$\mathbf{M_{M,B}} = \mathbf{M_{RM}} \times \mathbf{B_B} \tag{5.24}$$

5.3 Aerodynamics Drag

As the reference mission will be in LEO a considerable amount of disturbance force will come from atmospheric drag. The atmospheric drag force is influenced by a number of factors such as time, altitude, solar activity. The force for this simulation is found in the following method. First the density is determined at a specific altitude. The density can be found mathematically, however measurement tables can also be used and are easily accessible. For this simulation the NRLMSISE-00 (0N, 0E) reference value table is used which is altitude and not time dependent. As this is a point-wise table, an interpolation method also needs to be considered to accurately determine the densities between each measurement. Depending on the time steps, these jumps in values can be larger or smaller depending on the point in the atmosphere. Plotting the density for different altitudes show that the density is exponential and decays the higher the altitude. Therefore taking a linear interpolation method could cause large jumps in values which is not ideal.

The disturbance force can then be found using this density and Equation (5.25).

$$\mathbf{F_{A,R}} = -\frac{1}{2} * Cd * \rho * |V_R| V_R * S_{ref} \tag{5.25}$$

Cd is assumed to be 2 for our reference mission (de Vries, 2010). The force is typically in the aerodynamic frame and therefore needs to be transformed into the inertial frame for it to be considered in the simulator. This is done through Equation 5.26.

$$\mathbf{F_{A,I}} = \mathbf{C_{I,R}}\mathbf{F_{A,R}} \tag{5.26}$$

In addition to the force, a moment is also generated as found in Equation (5.27) where the moment arm is given as Equation (5.28) and $\mathbf{C_{B,R}}$ is found using the transformation in Equation (5.29).

$$\mathbf{M_{A,B}} = \Delta\mathbf{r_B} \times (\mathbf{C_{B,R}}\mathbf{F_{A,R}}) \tag{5.27}$$

$$\Delta\mathbf{r_B} = \mathbf{r_{cp}} - \mathbf{r_{cm}} \tag{5.28}$$

$$\mathbf{C_{B,R}} = \mathbf{C_{B,I}}\mathbf{C_{I,R}} \tag{5.29}$$

5.4 Solar Radiation Pressure (SRP)

Solar radiation pressure applies a force on the satellite caused by the photon absorbing and reflecting the exposed surface of the satellite. The magnitude is small and normally only affects the orientation minimally, but it will be considered as the satellite will experience sunlight and eclipse as it orbits around the Earth. Solar radiation pressure can be expressed as a function of solar flux (Φ) and the speed of light (c) as given in Equation (5.30). The speed of light is taken as 2.99772458 x 10^10 cm/s (Vallado and McClain, 2013). Solar flux is 1367 W/m^2 at 1 AU radius from the Sun. This value is used in the simulation of this thesis as the orbit is around Earth. If a more accurate value is required, the inverse square law can be used as shown in Equation (5.31) where R_H is $|\mathbf{r}_H|$ and $r_H = r - r_{Sun}$. $P_{S,0} = 4.56 \cdot 10^{-6}\,\mathrm{N/m^2}$ and $R_{1au} = 1.49597870691 \cdot 10^{11}$ m.

$$P_S = \frac{\Phi}{c} \tag{5.30}$$

$$P_S = P_{S,0}\frac{R_H^2}{R_{1au}^2} \tag{5.31}$$

The solar radiation pressure will depend on many factors. This includes exposed surface area, distance from the sun, orientation to the sun and whether the satellite is in an eclipse or not.

Eclipse

The status of the illumination depending on the eclipse can be represented by the illumination ν. When the satellite is in direct sun light, the satellite is not in eclipse and the illumination is complete and is equal to 1. For partial eclipse, the illumination is between 0 and 1 and is called the penumbra. The full eclipse has an illumination value of 0. The conical shadow model, as shown in Figure 5.1(Mooij, 2022b), can be used to determine the eclipse status.
The model neglects the atmosphere or oblateness of the bodies. R_e is the radius of the Sun, R_e is the radius of the Earth, $s_{sun} = r_s - r_e$ and $s = r - r_e$ where, r is the distance to the satellite. The fundamental plane is perpendicular to the shadow axis. This will intersect the shadow axis at

$$s_0 = -\frac{\mathbf{s}^T\mathbf{s_{Sun}}}{|\mathbf{s_{Sun}}|} \tag{5.32}$$

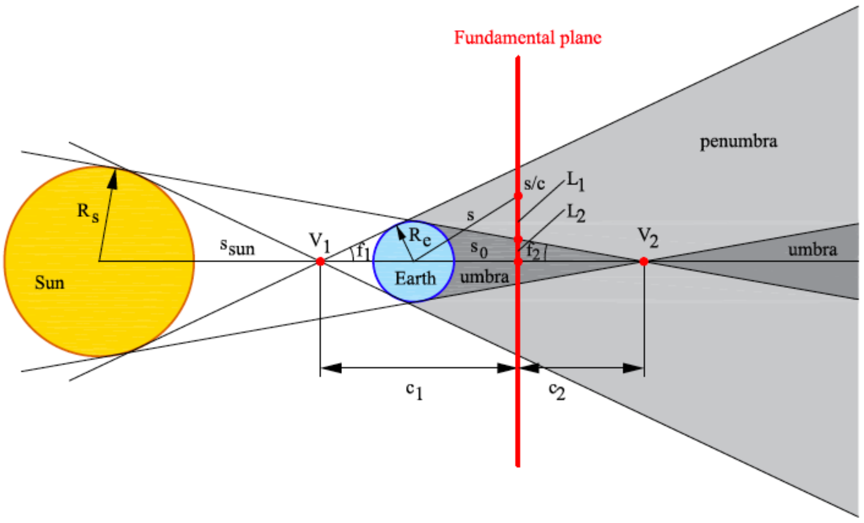


Figure 5.1: Conical shadow model

The distance of the satellite to the shadow axis is given as

$$L = \sqrt{s^2 - s_0^2} \tag{5.33}$$

The shadow cone angles are given as

$$\sin f_1 = \frac{R_s + R_e}{s_{\text{Sun}}} \quad \text{and} \quad \sin f_2 = \frac{R_s - R_e}{s_{\text{Sun}}} \tag{5.34}$$

For Earth, the half cone angle of the umbra (f_1) is 0.264 deg whereas the half angle for the penumbra (f2) is 0.269 deg. The distances for c_1 and c_2, which, are the distances from the fundamental plane to the vertices V1 and V2 of the shadow cones, are then given by the following

$$c_1 = s_0 + \frac{R_e}{\sin f_1} \quad \text{and} \quad c_2 = s_0 - \frac{R_e}{\sin f_2} \tag{5.35}$$

and the radii of the shadow cones in the fundamental plane is given by

$$R_1 = c_1 \tan f_1 \quad \text{and} \quad R_2 = c_2 \tan f_2 \tag{5.36}$$

The eclipse status is determined by the following conditions. No eclipse indicates when the Sun is fully visible. A complete eclipse, or Umbra, is where the satellite lies entirely in Earth's deep shadow and no sunlight reaches. The annular eclipse is where the Earth blocks the central disc of the sun, but a ring remains. Finally penumbra is where only a portion of the sun is blocked by the Earth.

1. **No Eclipse ($\nu = 1$):**

$$s_0 \leq -R_e \sin f_1 \quad \wedge \quad L > R_1 \tag{5.37}$$

2. **Complete Eclipse (Umbra, $\nu = 0$):**

$$R_2 < 0 \quad \vee \quad L < |R_2| \tag{5.38}$$

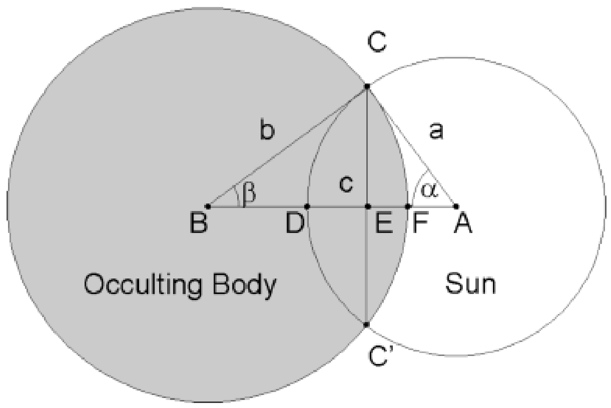


Figure 5.2: Occultation of the Sun by a spherical body

3. **Annular Eclipse:**

$$c_2 > 0 \quad \vee \quad R_2 > 0 \quad \vee \quad L < R_2 \tag{5.39}$$

4. **Partial Eclipse (Penumbra, $0 < \nu < 1$):** All other cases.

The degree of the suns' occultation by the Earth can be found by considering the overlap of the apparent circular disks as seen from the observer's point of view. The moon will be ignored for simplicity. Figure 5.2 shows the occultation of the sun by a spherical body.
where

$$a = \arcsin\left(\frac{R_s}{|\mathbf{r}_{\text{Sun}} - \mathbf{r}|}\right), \quad b = \arcsin\left(\frac{R_e}{s}\right), \quad \text{and} \quad c = \arccos\left(\frac{-\mathbf{s}^T(\mathbf{r}_{\text{Sun}} - \mathbf{r})}{s|\mathbf{r}_{\text{Sun}} - \mathbf{r}|}\right) \tag{5.40}$$

Given that $|a - b| < c < a + b$ the area of the occulted segment of the apparent solar disk is given by

$$A = ACFC' + ACDC' \tag{5.41}$$

The occulted area can be given by

$$A = a^2 \arccos\left(\frac{x}{a}\right) + b^2 \arccos\left(\frac{c - x}{b}\right) - cy \tag{5.42}$$

where the auxiliary parameters are given by

$$x = \frac{c^2 + a^2 - b^2}{2c} \quad \text{and} \quad y = \sqrt{a^2 - x^2} \tag{5.43}$$

Finally the remaining fraction of the sunlight can be given by

$$\nu = 1 - \frac{A}{\pi a^2} \tag{5.44}$$

The forces resulting from the solar radiation pressure, with consideration to the eclipse conditions, can then be found by the following method. Assuming the satellite is a sphere, the force can be found in Equation 5.45, where P_S is the energy flux of the solar radiation, S_{ref} is the effective cross-sectional area of the satellite and C_r is the reflectivity of the satellite. $\hat{\mathbf{r}}_H$ is the unit vector from the satellite to the Sun. This equation assumption can be applied as the satellite body being simulated is small. For larger bodies, a more detailed analysis can be done considering each surface of the body and its material coefficient.

$$\mathbf{F}_{S,I} = -C_r \frac{P_S S_{\text{ref}}}{c} \hat{\mathbf{r}}_H \tag{5.45}$$

The corresponding moment is obtained by taking the cross-product of the force with the vector from the centre of mass to the centre of pressure, the point at which the resultant radiation force is assumed to act. For a uniformly illuminated sphere, this point coincides with the geometric centre, so $\varDelta\mathbf{r}_B$ is typically set to zero unless slight offsets are introduced to model manufacturing tolerances:

$$\mathbf{M}_{S,B} = \varDelta\mathbf{r}_B \times (\mathbf{C}_{B,I}\mathbf{F}_{S,I}) \tag{5.46}$$

5.5 Acceptance tests

The environment models used in this thesis were taken from the GRADS library, where they have undergone prior verification and validation. Nevertheless, to ensure consistency and correctness within the specific simulation framework developed here, a series of acceptance tests were carried out. These tests focus on key model outputs and behaviours and are intended to confirm that the models respond realistically under expected operating conditions. A summary of the acceptance tests and their methods is presented in Table 5.1.

Five environment blocks: gravity, magnetic field, atmospheric drag, solar-radiation pressure, and the eclipse model, were verified. For each block, two checks were applied, an analytical spot calculation and a short numerical run to confirm trend behaviour.

- **Gravity.** The $1/r^2$ model reproduces $g = \mu/r^2$ at $r = 7000$ km; the spherical-harmonics model yields the expected $\approx 3\,\%$ increase in gravity magnitude from equator to pole, confirming the J_2 term.
- **Magnetic field.** Output strength remains within the IGRF 25–65 μT envelope, and a $0^\circ - 360^\circ$ longitude sweep produces the correct smooth sinusoid.
- **Atmospheric drag.** Model force at 300 km matches a hand calculation to within 3 %; doubling velocity increases drag by a factor of four, validating the $F \propto V^2$ relation.
- **Solar radiation pressure.** The force at 1 AU agrees with the flux-over-c estimate; the SRP term drops to zero when the eclipse flag is raised.
- **Eclipse logic.** A satellite placed on the Earth–Sun line behind Earth registers full eclipse, while a scripted penumbral transit yields a smooth illumination ramp.

These tests confirm that the GRADS environment blocks are correctly integrated and behave consistently with the theory presented earlier in this chapter.

Table 5.1: Environment Model Acceptance Tests

Model	**Acceptance Test**	**Method**	**Notes**
Gravity (Central Field)	Verify gravity acceleration at 7000 km matches theoretical $g = \mu/r^2$	Hand Calculation	Compare to $\approx 7.98\,\mathrm{m/s^2}$
Gravity (Spherical Harmonics)	Compare gravity magnitude at equator and pole (expect small difference)	Limited Simulation + Visual Check	Consistency check for oblateness effect
Magnetic Field	Confirm magnetic field strength falls within 25,000 nT – 65,000 nT range	Hand Calculation	Compare output to IGRF expected range
Magnetic Field	Verify smooth field rotation as satellite longitude varies 0° to 360°	Limited Simulation + Visual Plot	Smooth sinusoidal variation expected
Atmospheric Drag	Calculate drag force at 300 km altitude using table density	Hand Calculation	Use simple drag force formula for estimate
Atmospheric Drag	Confirm drag force scales with velocity squared	Limited Simulation	Double velocity, check force quadruples
Solar Radiation Pressure (SRP)	Calculate SRP force at 1 AU	Hand Calculation	Compare to $P_S = \Phi/c$ estimate
Solar Radiation Pressure (SRP)	Verify SRP force drops to zero during eclipse	Limited Simulation	Illumination factor ν goes to zero
Eclipse Model	Confirm full eclipse detection when satellite is behind Earth	Hand Calculation	Satellite along Earth-Sun axis should trigger eclipse
Eclipse Model	Check smooth illumination factor transition through penumbra	Limited Simulation + Visual Plot	Smooth decrease and increase in illumination

CHAPTER 6

Navigation

The navigation subsystem provides real-time attitude information by combining measurements from multiple sensors and an Extended Kalman Filter (EKF). The objective of this chapter is to develop, implement and validate the sensor models and the EKF used in the CubeSat simulator. First, the sensors are introduced, sun sensors, magnetometer, gyroscope and IMU, detailing their measurement principle, placement in the body frame and the error sources injected (bias, noise and misalignment). Next, the coordinate transformations are derived to express all measurements in the common body frame. Then, the design of the quaternion-based EKF is presented, including the state and measurement models, noise covariance tuning and consistency checks. Finally, the numerical results and acceptance tests, which demonstrate the filter's accuracy and robustness against the reference mission requirements, are shown.

6.1 Sensors

Now that the environment has been simulated and forces or moments have been calculated from each of the different environmental perturbations, the satellite must now "feel" these forces. For that, the sensors must also be simulated.

6.1.1 Sun sensor

There are three types of Sun sensors used in spacecraft attitude determination. These are analogue Sun sensors, Sun presence sensors and digital Sun sensors. The analogue Sun sensors, also referred to as cosine detectors, operate by outputting a current proportional to the solar flux incident on a group of photodiodes. The current varies depending on the angle of incidence. This is shown in Equation 6.1 where $I(\theta_0)$ is the maximum current when the sun is directly incident.

$$I(\theta) = I(\theta_0)\cos\theta \tag{6.1}$$

The analogue sensors are sensitive to small transmission losses due to Fresnel reflection as well as limited effective photocell area due to simplicity. Fresnel reflection refers to the partial reflection of light caused by an interface between two surfaces with different refractive indices. They typically have a conical field of view, which limits the field of view. Analogue sensors provide continuous data. A simple schematic can be seen in Figure 6.1 where **n** is the normal vector, P is the incident sunlight, and θ is the angle of incidence.

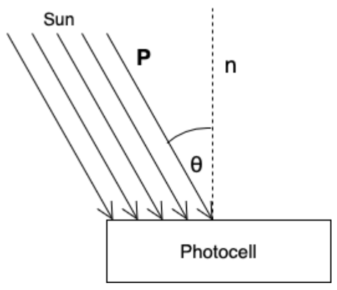


Figure 6.1: Analogue sun sensor

Sun presence sensors provide binary outputs to indicate whether the sun is within their field of view. They are mainly used to protect other instruments, navigation, and to activate hardware like solar panels when the Sun is detected. Sun presence sensors operate as a step function.

Digital Sun sensors are the most advanced type of Sun sensor as they output a discrete encoded function of the Sun's angle. They have a small field of view but provide high accuracy.

For this study, a Fine Sun sensor (FSS) was chosen. A fine Sun sensor is a type of analogue sun sensor and works as follows. The light passes through an aperture, triggering photosensitive cells, which are then used to determine the Sun vector. The Sun sensor model utilises the attitude quaternion **q**, the Sun's position $\mathbf{r}_{sun}$ in the inertial frame, and the position of the satellite **r** in the same frame. The output consists of the intensity currents from the four quadrants of the sensor, as well as a logic bit indicating whether the Sun is within the field of view (FOV).

The model begins by calculating the Sun vector in the instrument frame.

$$\mathbf{e_{sun-instru}} = \mathbf{C_{instr,B}} \mathbf{C_{BI}} \left(\frac{\mathbf{r_{sun}} - \mathbf{r}}{|\mathbf{r_{sun}}| \, |\mathbf{r}|} \right) \tag{6.2}$$

To determine which areas of the quadrants are illuminated, the angles between the Sun vector's x-component and the y- and z-axes are calculated as:

$$\tan \alpha_y = -\frac{\mathbf{e_{sun,inst}}(3)}{\mathbf{e_{sun,inst}}(1)} \tag{6.3}$$

$$\tan \alpha_z = -\frac{\mathbf{e_{sun,inst}}(2)}{\mathbf{e_{sun,inst}}(1)} \tag{6.4}$$

These angles are then corrected for the Sun sensor's accuracy and divided by the tangent of the FOV to account for the physical constraints of the sensor. The centre of the illuminated section is mapped to the instrument frame using:

$$\begin{pmatrix} Ny_{ss} \\ Nz_{ss} \end{pmatrix} = 0.5 \begin{pmatrix} \frac{\tan \alpha_y}{\tan(\text{FOV})} \\ \frac{\tan \alpha_z}{\tan(\text{FOV})} \end{pmatrix} \tag{6.5}$$

The current is then proportional to the illuminated area, and the sensor model outputs the ratio of the measured intensity to the maximum intensity. It is assumed that the thickness of the sensor material does not affect the illuminated area and is therefore ignored. The intensity ratios are given by

$$\begin{aligned} Q_1 &= \frac{I_1}{I_{max}} = (0.5 - Ny_{ss})(0.5 - Nz_{ss}) \\ Q_2 &= \frac{I_2}{I_{max}} = (0.5 + Ny_{ss})(0.5 - Nz_{ss}) \\ Q_3 &= \frac{I_3}{I_{max}} = (0.5 - Ny_{ss})(0.5 + Nz_{ss}) \\ Q_4 &= \frac{I_4}{I_{max}} = (0.5 + Ny_{ss})(0.5 + Nz_{ss}) \end{aligned} \tag{6.6}$$

Bias and white noise are added to these values to produce the final intensity measurements.

A slit sensor typically consists of two sensors: one aligned parallel to the satellite's spin axis and the other offset by an angle θ_o. The FOVs of these sensors intersect at the spin equator. The sensor sends an event pulse whenever its FOV covers the Sun. The Sun angle can then be determined as a function of the satellite's angular velocity $\boldsymbol{\omega}$ and the time step Δt, or by using Napier's rule for spherical triangles, as shown in Equation (6.7):

$$\tan\beta = \frac{\tan\theta_o}{\sin(\omega\Delta t)} \tag{6.7}$$

Napier's rule is used to solve right-angled spherical triangles.

The equation can be corrected for three possible physical misalignments of the instrument:

1. **Separation Misalignment:** This occurs when the two sensors' FOVs are not aligned. It can be corrected by applying $\theta_0 = \theta_0 + \Delta\theta$, resulting in the modified relation shown in Equation (6.8):

$$\tan\beta = \frac{\tan(\theta_0 + \Delta\theta)}{\sin(\omega\Delta t)} \tag{6.8}$$

2. **Elevation Misalignment:** This arises when the first sensor is not aligned with the satellite's spin axis, represented by an error ϵ. It is corrected by applying $\theta_o = \theta_o + \epsilon$, resulting in the following relation:

$$\tan\beta = \frac{\tan(\theta_0 + \epsilon)}{\sin(\phi + \omega\Delta t)} \tag{6.9}$$

3. **Azimuth Misalignment:** This refers to when the two FOVs intersect the Sun's equator, separated by an angle δ in the spin plane. Here, $\omega\Delta t = \omega\Delta t_0 - \delta$. The corrected equation is:

$$\tan\beta = \frac{\tan(\theta_0)}{\sin(\omega\Delta t - \delta)} \tag{6.10}$$

When all three misalignments are considered, the final expression for the corrected Sun angle is:

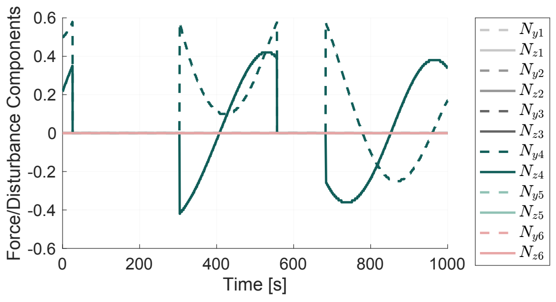
$$\tan\beta^2_{\Delta t\epsilon\delta} = \left(\frac{\tan(\theta_o + \Delta\theta + \epsilon) - \tan\epsilon\cos(\omega\Delta t - \delta)}{\sin(\sigma - \delta)}\right)^2 + \tan^2\epsilon \tag{6.11}$$

Mission Implementation

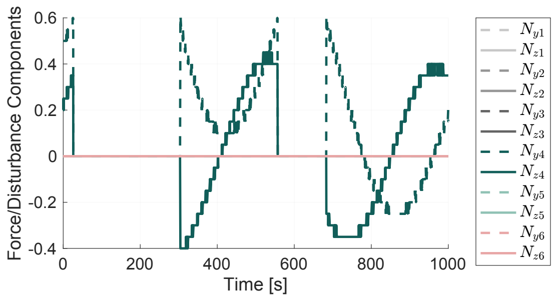
Six fine Sun sensors were chosen, one mounted on each face of the satellite, located at the centre of each surface. Each sensor is assumed to have a 60° FOV. The physical locations of the sensors are shown in Section 3.1, and their orientations are given in Table 6.1. The fine sun sensors are oriented relative to the CubeSat body frame using Euler angles (ϕ, θ, ψ) following the 3-2-1 rotation sequence.

Table 6.1: Fine Sun Sensor (FSS) orientations in body frame (Euler angles).

Sensor	**ϕ (deg)**	**θ (deg)**	**ψ (deg)**
SS1 (-Y)	90	0	0
SS2 (+Y)	-90	0	0
SS3 (-X)	0	-90	0
SS4 (+X)	0	90	0
SS5 (-Z)	180	0	0
SS6 (+Z)	0	0	0



(a) Sun Sensor Results without Noise



(b) Sun Sensor Results with Noise

Figure 6.2: Comparison of sun sensor outputs with and without noise.

The individual sun sensor model used took, the radius of the satellite, the sun vector and a transformation matrix (C_{bi}) for the sun vector to be transformed into the body frame. The outputs were the lit status, which indicated whether certain conditions were met for the sensor to send data. These conditions were if the vector was within the satellite's field of view and whether the satellite was in eclipse. The model has to be repeated six times for the six sensors, and a function was created to extract the outputs when a sensor would give a positive lit status to be then fed into the Kalman filter.

Some of the models in the GRADS library were not completely verified, as the library also had adjustments from other authors. This was the case for this model, as it was found that some vectors were being accepted as within the FOV when the Sun vector came from behind the sensors. Therefore, an extra clause was added to only allow the values when the third component of the normalised Sun vector (Sb_3) was positive. When implementing the model onto Eurosim, it was found that this model was incompatible. The initial tests were performed with the GRADS model and were later adjusted to allow for real-time implementation. The code for the fine sun section can be found in Listing A.1. These limitations will be discussed in chapter 9.

A test was performed where the outputs of the Sun sensors were extracted to determine whether the sensors correctly activated based on the sun vector's location. The simulation was run for 1000 seconds with an initial angular velocity of $[0.01, 0.01, 0.001]$ rad/s. This is a relatively high starting velocity, and since no controller was active in the model, large rotations persisted throughout the simulation. This approach allowed the sensor setup to be tested within a shorter simulation time, although such high angular velocities are not expected under real mission conditions. The simulation initially used idealised sensor models.

Figure 6.2a shows the extracted N_y and N_z components for the six Sun sensors. Only one sensor was active at a time, and transitions between active sensors were rare during the simulation. The sun vector path was also animated to verify that it crossed only one sensor field of view at a time. Although this sensor configuration is common and has heritage in previous missions, limitations are present: in periods where the sun is not visible to any sensor, additional sensing or data fusion techniques would be required.

In practice, Sun sensors are not ideal, and various errors must be introduced. The model parameters used to introduce these errors are summarised in Table 6.2. These values are based on the mission heritage reviewed in Chapter 2.

Figure 6.2 compares the ideal sun–vector components with the same signals after injecting bias, noise and quantisation error based on the Bradford Mini Fine Sun Sensor[1]. The added $0.01°$ bias, 1σ noise of $0.0333°$ and $0.0557°$ quantisation produce small perturbations in both

[1]https://satsearch.co/products/bradford-mini-fine-sun-sensor accessed 22/03/2025

Table 6.2: Fine Sun Sensor model parameters used for simulation.

Parameter	Value	Unit
Field of View (X, Y)	30	deg
Bias (X, Y)	0.01	deg
Noise (1σ) (X, Y)	0.0333	deg
Quantisation Step (X, Y)	0.0557	deg

amplitude and waveform. In particular, the quantisation manifests as a stepped profile, while the random noise slightly broadens the signal around its true value. Despite these effects, the overall temporal trend is preserved, and the error magnitudes match the sensor's datasheet specifications, confirming that our noise-injection model has been implemented correctly.

6.1.2 Magnetometer

Magnetometers are sensors that measure both the direction and magnitude of the magnetic field. They are lightweight, contain no moving parts, and provide a simple and reliable solution for attitude determination in Low Earth Orbit (LEO). These sensors generate three voltages proportional to the magnetic field intensity along three orthogonal axes corresponding to the body reference frame.

Magnetometers can be used to determine the attitude of a satellite by comparing the measured magnetic field with a model of the Earth's magnetic field. While they are not the most accurate sensors for attitude determination due to uncertainties in the magnetic field model and variations in Earth's magnetic field, they are often sufficient for many satellite applications where absolute pointing accuracy is not essential. The magnetic field strength decreases with $\frac{1}{r^3}$ of the Earth's radius, making magnetometers unsuitable for altitudes above 1000 km.

For attitude determination in space, particularly in LEO, the most commonly used magnetometers include: Fluxgate Magnetometers, known for their reliability and accuracy. Anisotropic Magneto-Resistive (AMR) Magnetometers, are valued for their compactness and low power consumption, making them ideal for small satellites. Giant Magneto-Resistive (GMR) Magnetometers, offer higher sensitivity than AMR sensors but are less common. Hall Effect Magnetometers are occasionally used for coarse measurements due to their simplicity and robustness. In this study, the model used is based on a triaxial **AMR magnetometer**, which provides voltage measurements along three orthogonal axes. The voltages are then corrected for biases, misalignments, and scale factors as part of the software model.

The magnetometer model provides a voltage, which can be corrected for biases, misalignments, and scale factors. The functional form is given as:

$$V_m = V(\mathbf{B}) + (\mathbf{b_B} + (\mathbf{I} + \mathbf{S_B} + \mathbf{M_B})\, V(\mathbf{B})) + \mathbf{v_B} \tag{6.12}$$

here $\mathbf{B}$ is the local magnetic intensity, $\mathbf{b_B}$ is the magnetometer bias, $\mathbf{S_B}$ is the magnetometer scale factor, $\mathbf{M_B}$ represents the magnetometer misalignment, and $\mathbf{v_B}$ is the white noise vector. The measurement voltage is then expressed as:

$$\frac{V_m}{V_{max}} = \frac{\mathbf{b_B}}{V_{max}} + (\mathbf{I} + \mathbf{S_B})\,\mathbf{e_B} + \frac{\mathbf{v_B}}{V_{max}} \tag{6.13}$$

Here, V_{max} represents the maximum voltage output, and $\mathbf{e_B}$ is the unit vector in the direction of the magnetic field lines.

Mission Implementation

The magnetometer was not available as a pre-built component in GRADS, so one was developed using the theory outlined earlier. Table Table 6.3 summarises the error parameters used for our simulated magnetometer, based on the NewSpace Systems NMRM-Bn25o485 flux-gate device[2]. In particular:

- **Drift** of $1 \times 10^{-9}\,\mathrm{T}/^{\circ}\mathrm{C}$ and **scale-factor error** of 150 ppm match the datasheet.
- **Noise density** of $1 \times 10^{-12}\,\mathrm{T}/\sqrt{\mathrm{Hz}}$ and a **quantisation step** of $0.0488\,\mu\mathrm{T}$ agree with the device's specifications.
- The **measurement range** is $\pm 100\,\mu\mathrm{T}$.
- The nominal **misalignment angle** is $\leq 0.0005^{\circ}$, but this proved too small to produce visible component errors in our plots. Therefore it was increased it to 0.005° to illustrate the impact of axis orthogonality error on the measured vector.

Figure 6.4 contrasts the magnetometer output with noise disabled (but with scale-factor error and misalignment applied) against the fully noisy case. Without noise, the measured field closely follows the true model apart from the small offset introduced by scale and alignment errors. When noise is enabled, high-frequency jitter appears around the baseline, broadening the distribution of residuals and increasing the filter's steady-state covariance. This highlights the importance of realistic noise modelling for tuning the EKF's measurement covariance and verifying its robustness against spurious fluctuations.

Unlike the Sun sensors, of which there are multiple, only a single magnetometer is used in the system. Figure 6.4 compares the magnetometer's output with and without simulated noise, based on the configuration settings listed in Table 6.3. The magnetometer receives the magnetic field vector in the CubeSat's body frame and outputs the same vector, but with added simulated error components such as noise and potential bias.

As seen in Figure 6.4, the magnitude of the Earth's magnetic field at low Earth orbit is relatively small, typically between 20 and 60 μT. This means that any noise introduced by the sensor or environment can have a noticeable impact on the signal, making accurate modelling of these disturbances essential for realistic simulation results. To minimise interference from onboard electronics and structural components, the magnetometer is positioned at the centre of the CubeSat.

Table 6.3: Magnetometer model parameters used for simulation.

Parameter	Value	Unit
Drift	1×10^{-9}	T/°C
Scale Factor Error	150 ppm	-
Misalignment Angle	0.005	deg
Noise (at 100 Hz)	1×10^{-12}	$\mathrm{T}/\sqrt{\mathrm{Hz}}$
Quantisation Step	0.0488	μT
Measurement Range	± 100	μT

[2] https://satsearch.co/products/newspace-systems-nmrm-bn25o485-magnetometer accessed 22/03/2025

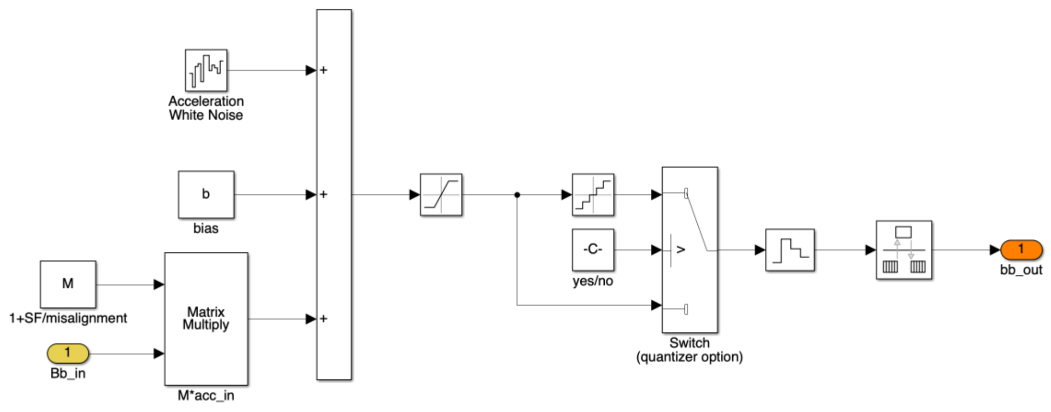
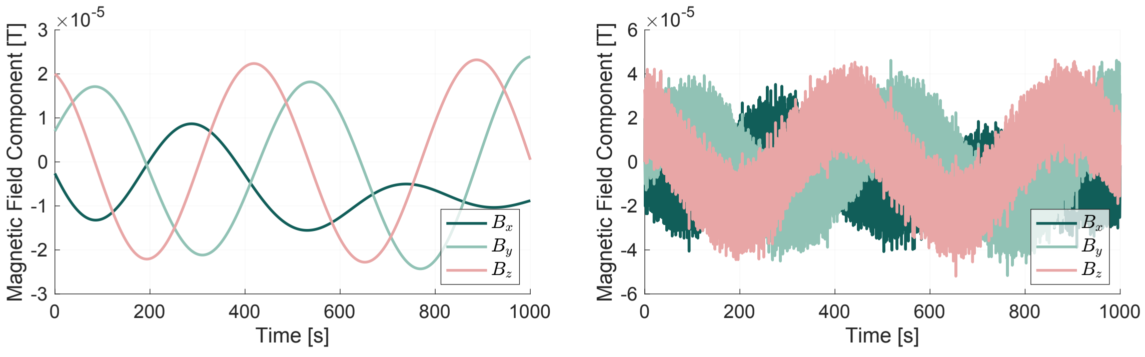


Figure 6.3: Magnetometer



(a) Magnetometer Results without Noise

(b) Magnetometer Results with Noise

Figure 6.4: Comparison of Magnetometer outputs without and with noise.

6.1.3 Inertial Measurement Unit (IMU)

The Inertial Measurement Unit (IMU) typically consists of accelerometers and gyroscopes, which measure linear acceleration and angular velocity, respectively, in three orthogonal axes.

Accelerometer

The accelerometer measures the linear acceleration of the satellite relative to an inertial frame. This measurement combines the real acceleration of the satellite with the acceleration due to gravity, both expressed in the body frame. The measured acceleration $\mathbf{a_m}$ is modelled as:

$$\mathbf{a_m} = \mathbf{a} + \mathbf{b_a} + (\mathbf{I} + \mathbf{S_a})\mathbf{a} \tag{6.14}$$

where $\mathbf{a}$ is the true acceleration in the body frame, $\mathbf{b_a}$ is the accelerometer bias, and $\mathbf{S_a}$ represents the scale factor and misalignment matrix. The bias $\mathbf{b_a}$ introduces a time-dependent offset in the measurements and is typically modelled as a random walk process, such that:

$$\frac{d\mathbf{b_a}}{dt} = \mathbf{w_a} \tag{6.15}$$

where $\mathbf{w_a}$ is normally distributed white noise. The scale factor and misalignment matrix $\mathbf{S_a}$ accounts for imperfections in sensor manufacturing and installation, and can be expressed, with the addition of a random walk, in matrix form as:

$$\mathbf{S_a} = \begin{bmatrix} s_x & m_{xy} & m_{xz} \\ m_{yx} & s_y & m_{yz} \\ m_{zx} & m_{zy} & s_z \end{bmatrix} \text{x random walks} \tag{6.16}$$

where s_x, s_y, s_z are the scale factors along each axis, and m_{ij} terms represent the cross-axis misalignments. These parameters cause deviations in the measured acceleration, which must be calibrated to improve accuracy.

6.1.4 Gyroscope

The gyroscope measures the angular velocity $\boldsymbol{\omega}$ of the satellite relative to an inertial frame. It outputs a voltage proportional to the angular velocity, which is used to calculate changes in orientation. The measured angular velocity $\boldsymbol{\omega}_m$ can be expressed as:

$$\boldsymbol{\omega}_m = \boldsymbol{\omega} + \mathbf{b}_\omega + (\mathbf{I} + \mathbf{S}_\omega)\boldsymbol{\omega}, \tag{6.17}$$

where $\boldsymbol{\omega}$ is the true angular velocity, $\mathbf{b}_\omega$ is the gyroscope bias (commonly referred to as gyroscopic drift), and $\mathbf{S}_\omega$ is the scale factor and misalignment matrix, similar in form to the accelerometer's scale factor matrix. The bias $\mathbf{b}_\omega$ is also modeled as a random walk:

$$\frac{d\mathbf{b}_\omega}{dt} = \mathbf{w}_\omega \tag{6.18}$$

where $\mathbf{w}_\omega$ represents white noise. The gyroscope can operate in two modes: rate gyro mode, where it measures angular velocity, and rate-integrating mode, where angular displacement is measured by integrating the angular velocity over time. For rate gyros, the angular rate ω_i^M can be modelled as:

$$\omega_i^M = s_r\theta_R \tag{6.19}$$

where s_r is the scale factor and θ_R is the angular velocity. For rate-integrating gyros, the angular displacement is obtained as:

$$\omega_i^M = \frac{s_i\theta_i}{\Delta t} \tag{6.20}$$

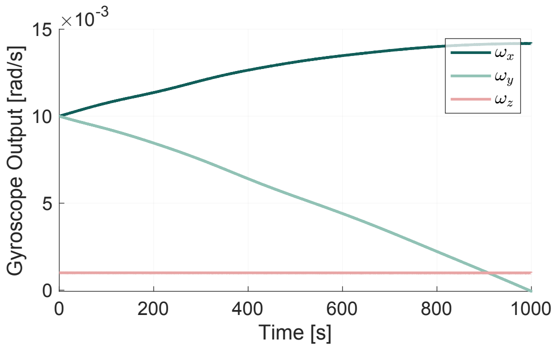
The misalignment and scale factor matrix for the gyroscope, $\mathbf{S}_\omega$, is similarly expressed as:

$$\mathbf{S}_\omega = \begin{bmatrix} s_p & m_{pq} & m_{pr} \\ m_{qp} & s_q & m_{qr} \\ m_{rp} & m_{rq} & s_r \end{bmatrix} \text{x random walks} \tag{6.21}$$

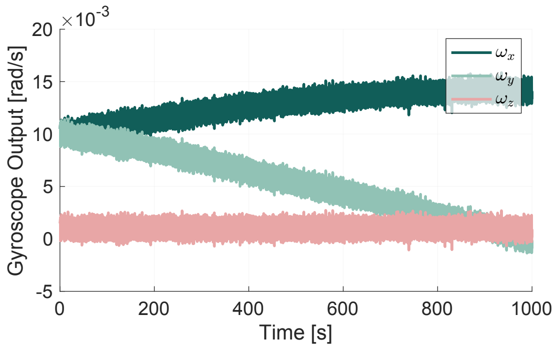
This matrix accounts for imperfections in sensor alignment and scaling, which can significantly impact the measured angular velocity.

6.1.5 Reference mission

The satellite utilises a single three-axis gyroscope to provide angular velocity measurements. The key sources of measurement errors include bias drift, scale factor errors, random noise, and quantisation effects. are detailed in Table 6.4. To validate the error modelling, Figure 6.5 compares the gyroscope outputs with ideal conditions (no errors) against outputs with the full error model applied. The comparison demonstrates the expected degradation in signal quality, confirming that the error models were implemented correctly. Table 6.4 details the



(a) Gyro Results without Noise



(b) Gyro Results with Noise

Figure 6.5: Comparison of Gyro outputs without and with noise.

error parameters inspired by the NewSpace Systems NSGY-001 stellar gyro[3]. The bias drift is 1×10^{-7}, rad/s, the scale factor error is 500, ppm, the misalignment angle is 0.5°, the angular random walk is 1.7×10^{-5}, rad/$\sqrt{\text{s}}$, and the quantisation step is 0.0146°/s over a measurement range of ±30°/s.

To validate the error modelling, Figure 6.5 compares the ideal output (scale factor and misalignment only) against the fully noisy output (including random walk and quantisation). Without noise, the gyroscope follows the true angular rate apart from a small static offset due to scale factor and alignment errors. When noise is enabled, high-frequency jitter appears, matching the angular random walk specification, and the output exhibits a stair step profile from quantisation. Over longer intervals, the bias drift produces a slow deviation from the true rate. This confirms that each error source has been correctly implemented and supports the tuning of the filter's measurement covariance.

Table 6.4: Three-Axis Gyroscope model parameters used for simulation.

Parameter	Value	Unit
Drift	1×10^{-7}	rad/s
Scale Factor Error	500 ppm	-
Misalignment Angle	0.5	deg
Noise (ARW)	1.7×10^{-5}	rad/$\sqrt{s}$
Quantisation Step	0.0146	deg/s
Measurement Range	± 30	deg/s

6.2 Modelling errors

This section highlights important errors present in an IMU and the output signal. The physical representation of the signal error can be seen in Figure 6.6. (Balaban et al., 2010) (Mooij, 2022b)

- **Bias** is the constant offset by a certain value. This can occur due to insufficient calibration, or a shift in the physical placement of a sensor or internal component. This can be symbolised by $Y_f = X + \beta + noise$, where β is the bias factor.

[3]https://satsearch.co/products/newspace-systems-nsgy-001-stellar-gyro accessed 22/03/2025

- **Drifting** is a time-varying offset. Much like the bias factor, this is applied to the output signal, however, the difference is that it varies with time. The error can be symbolised by $Y_f = X + \delta(t) + noise$, where $\delta(t)$ is the time-varying factor.
- **Scaling** is also known as gain failure, where a constant factor is applied to the magnitude of the signal received. This can be represented by $Y_f = X + \alpha x + noise$ where $\alpha(t)$ is the scaling factor and can be time-varying. It is usually caused by manufacturing tolerances or ageing.
- **Saturation** is where the maximum or minimum value that the sensor can measure is exceeded but displays itself as a constant max min value.
- **Deadzone** is where no measurements can be taken due to a form of static friction being felt or the sensor locks, which results in a lock-in.
- **Sign assymmetry** occurs with the scale factor error, which is caused by a misalignment of the push and pull amplifiers in the electronic sensor configuration.
- **Hystersis** is caused by past sensor readings interrupting the current readings, which is caused by the system being in more than one internal state.
- **Quantisation** occurs when an analogue signal is trying to be interpreted by a finite number of digits causing an error.
- **Misalignments** are caused by physical aspects, such as the mounting or placement of the sensor not being aligned to sensitive axes.
- **Noise** is a random factor, which can be applied to the signal. It can also vary randomly with time. This factor can be modelled under certain assumptions, however, the generation of these assumptions is often the problem.
- **Hard Faults** are when the signal is limited to a certain level and can be symbolised as C which can then be applied as $Y_f = C + noise$. This error can also be subdivided into two subcategories. First is the Loss of signal where the sensor no longer sends out a signal or is blocked, so the output is read as 0, whereas "Stuck sensor" refers to a specific value C which is constantly outputted.
- **Intermittents or Non-linearity** is when a factor is applied to the nominal readings in irregular intervals throughout its lifetime. This error is difficult to simulate and identify due to its random nature.

6.3 Navigation Filters

Now that the satellite senses the environment, the data from the sensors must be converted into the satellite's attitude, position, velocity or whatever is needed. Each sensor outputs a variable, which needs to be converted to useful information about the motion and orientation of the satellite. In this simulation, the attitude is represented by the quaternions with the angular velocity, providing the rate of change of attitude as well as the position and velocity in the ECEF frame.

The sun sensor outputs a sun vector, which, when combined with an onboard sun position model, helps determine the satellite's orientation relative to the Sun. The gyroscope provides the rate of change of orientation, allowing for updates to the attitude over time. The gyroscope

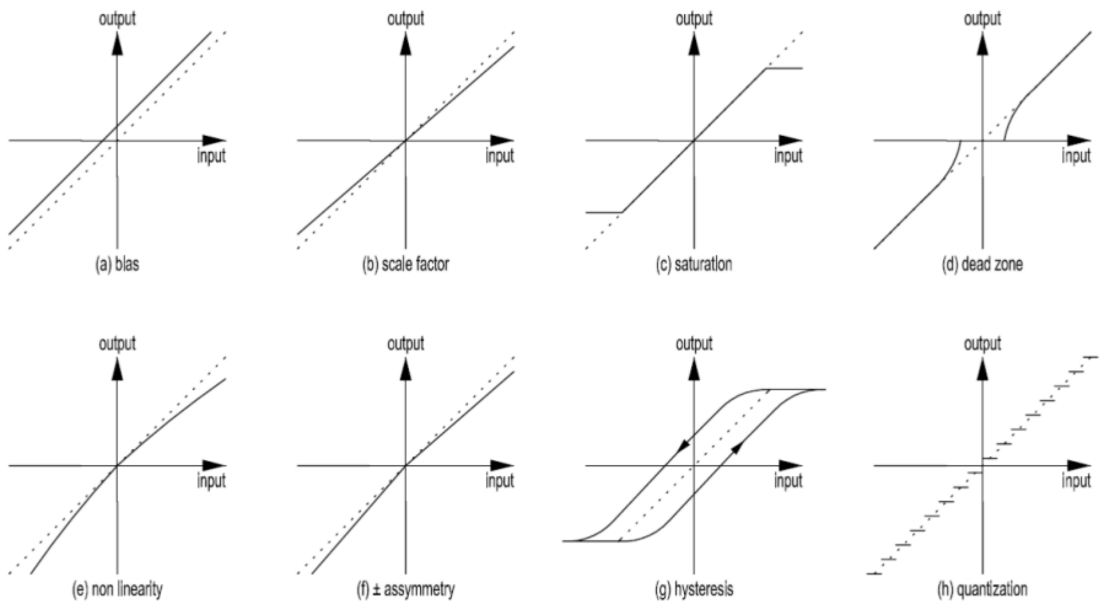


Figure 6.6: Sensor errors Mooij (2022b)

can be used to estimate changes in angular velocity, and the magnetometer determines the satellite's orientation relative to the Earth's magnetic field when used in conjunction with an onboard magnetic field model.

6.3.1 Filter Selection

The attitude-estimation problem combines nonlinear quaternion dynamics, gyroscope bias evolution and vector measurements (magnetometer plus sun sensor). Several Bayesian filters were considered:

- **Linear Kalman Filter (LKF)** is optimal only for linear models. Application to quaternion kinematics would require local linearisation at each step, incurring modelling error and poor convergence in highly nonlinear regimes.

- **Unscented Kalman Filter (UKF)** uses deterministic sigma points to capture second-order statistics without explicit Jacobians. Although more accurate than the EKF under strong nonlinearity, it requires roughly two to three times more function evaluations per step, straining the real-time budget on single-board computers.

- **Extended Kalman Filter (EKF)** applies a first-order Taylor expansion to exploit available Jacobians of the process and measurement models. It delivers acceptable accuracy for quaternion attitude estimation with minimal computational overhead and fits comfortably within the real-time constraints.

On balance, the EKF offers the best compromise among estimation accuracy, computational complexity and implementation simplicity for a 100Hz real-time attitude estimator on Raspberry Pi 5 / Orange Pi 5 Plus.

6.3.2 Extended Kalman Filter (EKF)

The EKF is used to fuse nonlinear sensor measurements into a best estimate of the attitude quaternion and gyroscope biases. At each discrete time step k, it executes:

First, the *prediction* uses the non-linear process model

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k) + \mathbf{w}_k, \quad \mathbf{w}_k \sim \mathcal{N}(0, \mathbf{Q}).$$

Here $\mathbf{x}$ contains the quaternion and bias, $\mathbf{u}$ is the control input (gyro readings), and $\mathbf{Q}$ is the process-noise covariance. The *a-priori* state estimate is

$$\hat{\mathbf{x}}_{\bar{k}+1} = f(\hat{\mathbf{x}}_k, \mathbf{u}_k).$$

To propagate uncertainty, the Jacobian of f with respect to $\mathbf{x}$ is computed,

$$\mathbf{F}_k = \frac{\partial f}{\partial \mathbf{x}}\bigg|_{\hat{\mathbf{x}}_k, \mathbf{u}_k},$$

and the covariance is updated as

$$\mathbf{P}_{\bar{k}+1} = \mathbf{F}_k \mathbf{P}_k \mathbf{F}_k^T + \mathbf{Q}.$$

Next, the *update* incorporates the non-linear measurement model

$$\mathbf{z}_{k+1} = h(\mathbf{x}_{k+1}) + \mathbf{v}_{k+1}, \quad \mathbf{v}_{k+1} \sim \mathcal{N}(0, \mathbf{R}),$$

where $\mathbf{z}$ contains magnetometer and sun–sensor outputs and $\mathbf{R}$ is the measurement-noise covariance. The measurement Jacobian is

$$\mathbf{H}_{k+1} = \frac{\partial h}{\partial \mathbf{x}}\bigg|_{\hat{\mathbf{x}}_{\bar{k}+1}}.$$

The Kalman gain follows as

$$\mathbf{K}_{k+1} = \mathbf{P}_{\bar{k}+1} \mathbf{H}_{k+1}^T \left(\mathbf{H}_{k+1} \mathbf{P}_{\bar{k}+1} \mathbf{H}_{k+1}^T + \mathbf{R}\right)^{-1}.$$

The state correction is applied to the innovation $\mathbf{z}_{k+1} - h(\hat{\mathbf{x}}_{\bar{k}+1})$,

$$\hat{\mathbf{x}}_{k+1} = \hat{\mathbf{x}}_{\bar{k}+1} + \mathbf{K}_{k+1}\left(\mathbf{z}_{k+1} - h(\hat{\mathbf{x}}_{\bar{k}+1})\right),$$

and the covariance is finalised by

$$\mathbf{P}_{k+1} = \left(\mathbf{I} - \mathbf{K}_{k+1} \mathbf{H}_{k+1}\right) \mathbf{P}_{\bar{k}+1}.$$

In words, the prediction step propagates both the quaternion and its uncertainty through the known dynamics, while the update step corrects that prediction using the newest sensor readings. The Jacobians $\mathbf{F}_k$ and $\mathbf{H}_k$ capture local linearisations, and the tuning of $\mathbf{Q}$ and $\mathbf{R}$ balances trust between model and measurements.

6.4 EKF Implementation

To implement the Extended Kalman Filter (EKF), the process was divided into several stages to address the various variables involved. The first step involved testing the quaternion propagation from angular velocity (ω) to confirm the correctness of the implementation. It is also important to test the propagation methods for the magnetic field lines and the sun vectors.

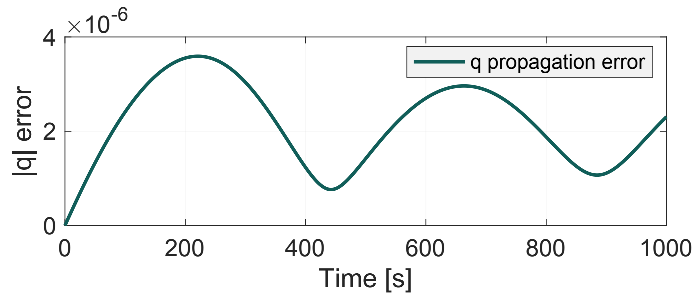


Figure 6.7: Quaternion Propagation error

This was followed by validating whether, under ideal conditions, the outputs from the sun sensors (Ny and Nz) could be used to back-calculate the Sun vector. Subsequently, a simplified EKF utilising data from a single sun sensor was developed and tested. An additional step incorporated a magnetometer (Bb) into the filter. After this, a multi-sun-sensor EKF configuration was evaluated. Finally, a fully integrated EKF system was developed, combining data from the magnetometer, gyroscope, and six sun sensors positioned at different locations on the CubeSat.

The initial conditions used for the simulation were:

- Propagators.State.w0 = [0.01; 0.01; 0.001];
- Propagators.State.q0 = [0.0; 0.0; 0.0; 1];

6.4.1 Propogation

This section starts with quarterion propagation based on the theory discussed previously. It then continues to propagate the magnetic field lines and sun vector, which will later be used in the EKF implementation. This section does not apply the filter yet but only tests the conversion methods.

Quarterion propagation

This process takes the angular velocity from the simulator and determines the quaternion using:

$$\begin{pmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \end{pmatrix} = \frac{1}{2} \begin{bmatrix} q_4 & -q_3 & q_2 \\ q_3 & q_4 & -q_1 \\ -q_2 & q_1 & q_4 \\ -q_1 & -q_2 & -q_3 \end{bmatrix} \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix}. \tag{6.22}$$

Figure 6.7 shows the magnitude of quaternion error over 1000 seconds with errors ranging from $4 \cdot 10^{-6}$ to $1 \cdot 10^{-6}$, making the propagator a good fit for the Kalman filter.

Quaternion-Error Computation

To quantify the attitude difference between a reference quaternion $\mathbf{q}_{\text{ref}}$ and an estimated quaternion $\mathbf{q}_{\text{est}}$, define the error quaternion by

$$\mathbf{q}_e = \mathbf{q}_{\text{est}} \otimes \mathbf{q}_{\text{ref}}^{-1}, \tag{6.23}$$

where $\otimes$ denotes quaternion multiplication and

$$\mathbf{q}^{-1} = \begin{bmatrix} -\mathbf{q}_v \\ q_4 \end{bmatrix} \quad \text{for} \quad \mathbf{q} = \begin{bmatrix} \mathbf{q}_v \\ q_4 \end{bmatrix}. \tag{6.24}$$

This formulation computes the relative rotation between the estimated and reference orientations. By multiplying the estimated quaternion with the inverse of the reference, the resulting error quaternion $\mathbf{q}_e$ represents the rotation required to align the estimated orientation with the true one. Since both quaternions are unit norm, the resulting error quaternion is also unit norm, i.e., $\|\mathbf{q}_e\| = 1$, and it encodes only the difference in orientation.

The corresponding rotation-angle error $\delta\theta$ is

$$\delta\theta = 2\arccos\big(q_{e,4}\big), \tag{6.25}$$

and the axis of rotation is

$$\hat{\mathbf{e}} = \frac{\mathbf{q}_{e,v}}{\sin\big(\delta\theta/2\big)}. \tag{6.26}$$

The angle error $\delta\theta$ provides a scalar measure of the misalignment between the two orientations, capturing the minimum rotation angle needed to correct the estimated attitude back to the reference frame.

Magnetic field vector propagation

The magnetometer measures the magnetic field, where the output of the sensor is a three-variable vector in the body frame. The reference data is taken in the inertial frame, therefore, the only conversion needed was that from body to inertial frame. This was done with Equation (4.11) and repeated again below. Sample data was taken from the magnetometer and converted using predetermined quaternion values, which showed that the conversion method converted the body frame back to an inertial frame correctly.

$$\mathbf{C}_{I,B} = \begin{bmatrix} q_1^2 + q_4^2 - q_2^2 - q_3^2 & 2(q_1q_2 + q_3q_4) & 2(q_1q_3 - q_2q_4) \\ 2(q_1q_2 - q_3q_4) & q_2^2 + q_4^2 - q_1^2 - q_3^2 & 2(q_2q_3 + q_1q_4) \\ 2(q_1q_3 + q_2q_4) & 2(q_2q_3 - q_1q_4) & q_3^2 + q_4^2 - q_1^2 - q_2^2 \end{bmatrix} \tag{6.27}$$

]

Sun Vector Propagation

Sun sensor measurements are taken in each sensor's local frame. Each of the six sensors is mounted in a different orientation. The Euler angles that define each sensor's orientation with respect to the body frame are listed in Table 6.1.

To compare the measured and reference sun vectors, three transformation strategies were evaluated:

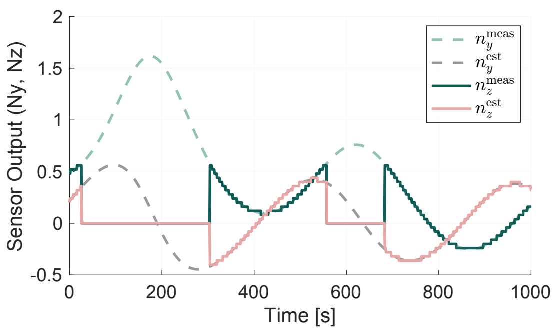
1. Convert the reference Sun vector into N_y, N_z components in the sun sensor frame, and compare directly with sensor outputs.

2. Convert the N_y, N_z components from the sensor back into the inertial frame.

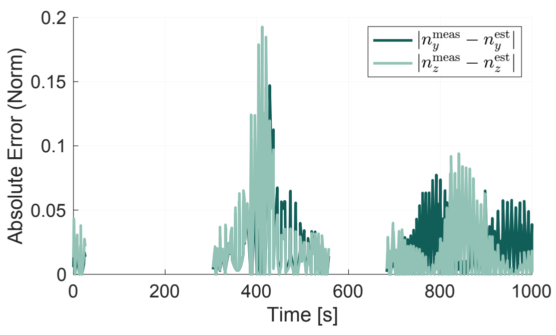
3. Convert both the N_y, N_z components and the reference sun vector into the body frame.

To transform N_y, N_z values into a 3D sun vector in the sun sensor frame, the following formula is used:

$$\mathbf{RSun} = \begin{pmatrix} -N_z \\ N_y \\ 1 \end{pmatrix} \cdot \frac{1}{\sqrt{N_y^2 + N_z^2 + 1}} \tag{6.28}$$



(a) Propagating sun vector in N_y, N_z components



(b) Error plot of sun vector components N_y, N_z

Figure 6.8: Sun vector components in the sensor frame

This vector is then rotated into the body frame using the rotation matrix defined by Equation (4.10) and the Euler angles in Table 6.1. Subsequently, the body-frame sun vector can be rotated into the inertial frame using Equation (4.11).

An analysis was performed to determine which transformation method introduces the least error. This decision is critical for sensor fusion, especially since Sun sensors have a limited field of view. Periods without data (visible in the figures) occur when the sun is outside the sensor's view. This was verified by checking the lit status was zero as well as checking the eclipse conditions when the values dropped. These blind spots highlight the need for data fusion from multiple sensors.

Method 1

Figure 6.8 illustrates the results for Method 1, where the reference sun vector is converted into N_y, N_z components in the sensor frame. Section 6.4.1 shows the propagated values including small discretisation effects and missing data during sensor blind periods. The body vector was converted to the sun sensor frame using the following equation where ϕ, θ, ψ are the transformation angles from the body frame of the satellite to the sun sensor frame :

$$\mathbf{R}^\top(\phi,\theta,\psi) = \begin{bmatrix} \cos\psi\cos\theta & \cos\theta\sin\psi & -\sin\theta \\ \sin\phi\cos\psi\sin\theta - \cos\phi\sin\psi & \sin\phi\sin\psi\sin\theta + \cos\phi\cos\psi & \sin\phi\cos\theta \\ \cos\phi\cos\psi\sin\theta + \sin\phi\sin\psi & \cos\phi\sin\psi\sin\theta - \sin\phi\cos\psi & \cos\phi\cos\theta \end{bmatrix} \tag{6.29}$$

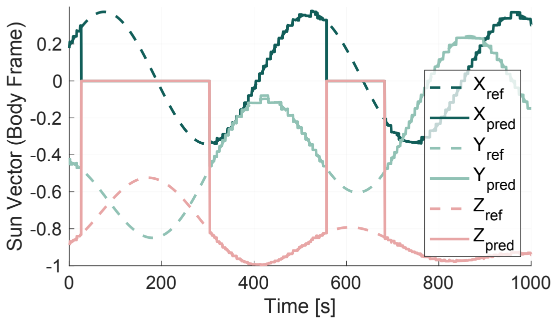
The corresponding error plot in Section 6.4.1 shows that this method produces the largest errors.

Method 3

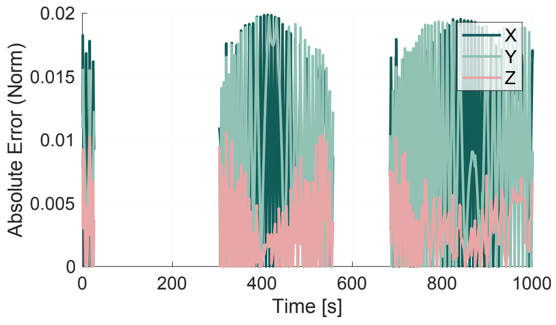
Figure 6.9 presents the results for Method 3, where both the measurements and the reference sun vector are converted into the body frame. This method results in significantly lower errors—approximately one order of magnitude lower than Method 1.

Method 2

Figure 6.10 shows the performance of Method 2, where sensor data are converted into the inertial frame. This method performs better than Method 1 but slightly worse than Method 3. Method 3 was found to be the most efficient likely due to reduced accumulation of transformation errors. Although all methods involve transforming between frames, Method 3 brings both the measured and reference sun vectors into the body frame, avoiding repeated



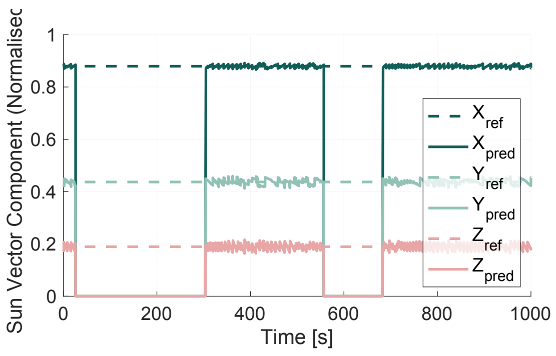
(a) Propagating sun vector in the body frame



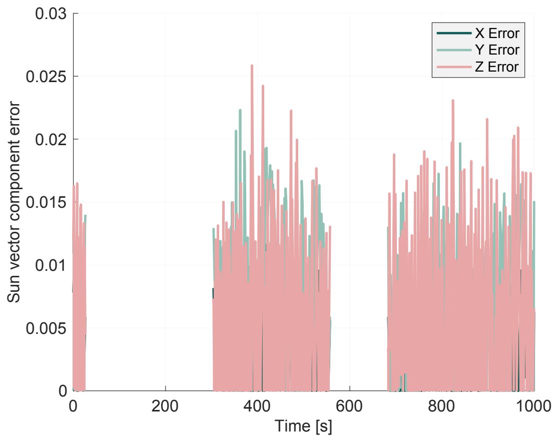
(b) Error plot of sun vector in the body frame

Figure 6.9: Sun vector in the body frame

back-and-forth transformations between sensor and inertial frames. This results in more stable comparisons and lower overall error. As such, converting both the measurements and reference vector to the body frame was chosen as the optimal method.



(a) Propagating sun vector in the inertial frame



(b) Error plot of sun vector in the inertial frame

Figure 6.10: Sun vector in the inertial frame

With the body-frame method selected, development of the Extended Kalman Filter (EKF) can proceed using this approach for Sun vector propagation.

6.4.2 EKF Structure

The theoretical foundations of the Extended Kalman Filter (EKF) have been introduced previously. This section describes the practical construction of the EKF used in this work.

Tuning Parameters

The EKF tuning relies on three main covariance matrices: the process noise matrix $\mathbf{Q}$, the measurement noise matrix $\mathbf{R}$, and the initial error covariance matrix $\mathbf{P}_0$.

The process noise matrix $\mathbf{Q}$ determines how closely the filter adheres to the internal model dynamics. It is typically tuned by adjusting the diagonal values. Smaller values indicate stronger confidence in the model, reducing reliance on potentially noisy sensor measurements.

The measurement noise matrix **R** characterises the uncertainty associated with the sensor inputs. Higher values indicate reduced trust in the sensor measurements, leading the filter to favour the model prediction during updates.

The initial error covariance matrix $\mathbf{P}_0$ represents the filter's initial confidence in the state estimate. Smaller values suggest high initial confidence and imply that the error between the initial predicted state and the true state is expected to be small.

The estimation methods for all were based on trial and error as each variable changed a different behaviour in the predictions and the filters nature being quite simple allowed for this.

State Transition Matrix F

The state transition matrix **F** is computed based on the relationship between the input angular velocity and the predicted quaternion state. Since the filter estimates only the vector part of the quaternion, the dynamics can be linearised assuming small angular velocity and discrete time steps.

Let the angular velocity be $\boldsymbol{\omega} = [\omega_x, \omega_y, \omega_z]^\top$. The quaternion kinematics are approximated as:

$$\mathbf{F} = \mathbf{I}_3 + \frac{1}{2}\Omega_\times \Delta t$$

where Δt is the sampling interval, and $\Omega_\times$ is the skew-symmetric matrix of $\boldsymbol{\omega}$, defined as:

$$\Omega_\times = \begin{bmatrix} 0 & \omega_z & -\omega_y \\ -\omega_z & 0 & \omega_x \\ \omega_y & -\omega_x & 0 \end{bmatrix}$$

This formulation yields a linear approximation of the continuous-time quaternion dynamics suitable for use in the prediction step of the EKF.

Measurement Matrix H

The measurement matrix **H** defines the relationship between the state and the expected sensor measurements. In this implementation, the measurement model assumes that a known inertial vector $\mathbf{v}_{\text{inertial}}$, such as the Sun vector or magnetic field direction, is observed in the body frame. This observation is modelled through a rotation using the quaternion:

$$\mathbf{z}_{\text{pred}} = \mathbf{R}(\mathbf{q}) \cdot \mathbf{v}_{\text{inertial}}$$

Here, $\mathbf{q} = [q_1, q_2, q_3]^\top$ represents the vector part of the quaternion, while the scalar part q_4 is reconstructed to maintain unit norm as:

$$q_4 = \sqrt{1 - q_1^2 - q_2^2 - q_3^2}$$

To obtain the Jacobian **H**, the rotation matrix is differentiated with respect to the quaternion vector components. The full quaternion is written as:

$$\mathbf{q} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix}$$

The Jacobian matrix $\mathbf{H} \in \mathbb{R}^{3\times3}$ is then defined as:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial \mathbf{R}(\mathbf{q}) \cdot \mathbf{v}_{\text{inertial}}}{\partial q_1} & \frac{\partial \mathbf{R}(\mathbf{q}) \cdot \mathbf{v}_{\text{inertial}}}{\partial q_2} & \frac{\partial \mathbf{R}(\mathbf{q}) \cdot \mathbf{v}_{\text{inertial}}}{\partial q_3} \end{bmatrix}$$

Each derivative includes both explicit dependence on q_1, q_2, and q_3, and implicit dependence through q_4, due to the unit quaternion constraint:

$$\frac{\partial q_4}{\partial q_i} = \frac{-q_i}{q_4}, \quad i \in \{1, 2, 3\}$$

Thus, the partial derivatives of the rotated vector become:

$$\frac{\partial \mathbf{R}(\mathbf{q}) \cdot \mathbf{v}_{\text{inertial}}}{\partial q_i} = 2 \left(\mathbf{M}_i \cdot \mathbf{v}_{\text{inertial}} + \frac{q_i}{q_4} \mathbf{M}_4 \cdot \mathbf{v}_{\text{inertial}} \right)$$

where $\mathbf{M}_i$ are matrices resulting from differentiating the rotation operation with respect to the quaternion components.

In this implementation, the structure of $\mathbf{H}$ is also made conditional on the availability of certain sensor inputs. For example, the presence or absence of sun sensor data (e.g., indicated by a "lit" status flag) dynamically modifies the rows of $\mathbf{H}$ during runtime.

6.4.3 EKF with One magnetometer

The first sensor integrated into the Extended Kalman Filter (EKF) was the magnetometer. The EKF receives measured magnetic field vectors from the sensor and compares them with the expected magnetic field calculated from an internal inertial model. Using this comparison and the appropriate coordinate transformation (e.g., the direction cosine matrix), the quaternion representing the spacecraft's attitude is updated.

The resulting covariance estimates from the filter are shown in Figure 6.12, while the corresponding component-wise attitude errors are presented in Figure 6.11. The filter settings used for this simulation are summarised in Section 6.4.3.

EKF Initial Parameters

Simulation Setup:

- **Final time:** $T_{\text{final}} = 1000$
- **Initial state:** $x_0 = [0 \quad 0 \quad 0 \quad 1]^T$

Covariance Matrices:

- **Process noise (Q):** $\text{diag}(5 \times 10^{-8},\ 2 \times 10^{-8},\ 5 \times 10^{-6})$
- **Measurement noise (R):** $\text{diag}(0.01,\ 0.01,\ 0.1)$
- **Initial error covariance ($\mathbf{P}_0$):** $\text{diag}(7 \times 10^{-3},\ 7 \times 10^{-3},\ 7 \times 10^{-3})$

6.4.4 EKF with One Sun Sensor

This experiment implements the Extended Kalman Filter (EKF) using data from a single sun sensor. Compared to the magnetometer case, the integration was more challenging due to using only one sensor. Specifically, the sun sensor has a limited field of view, meaning that during certain periods of the simulation, the sun was outside its observable range and no measurements were available.

To account for these conditions, the process noise was reduced while the measurement noise was increased, as detailed in Section 6.4.4. Additionally, the sun sensor provides only

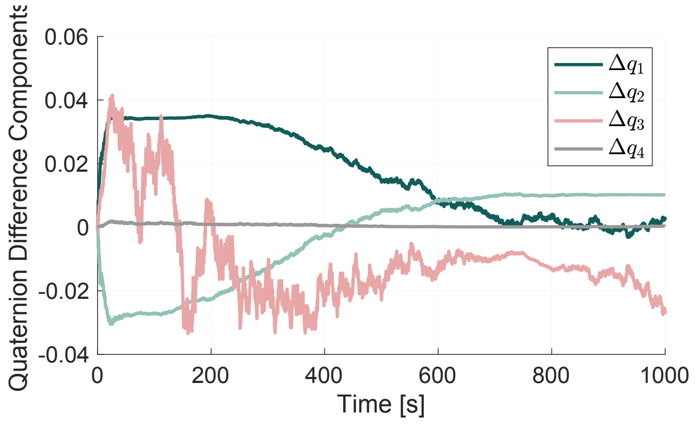


Figure 6.11: EKF with one magnetometer - q error

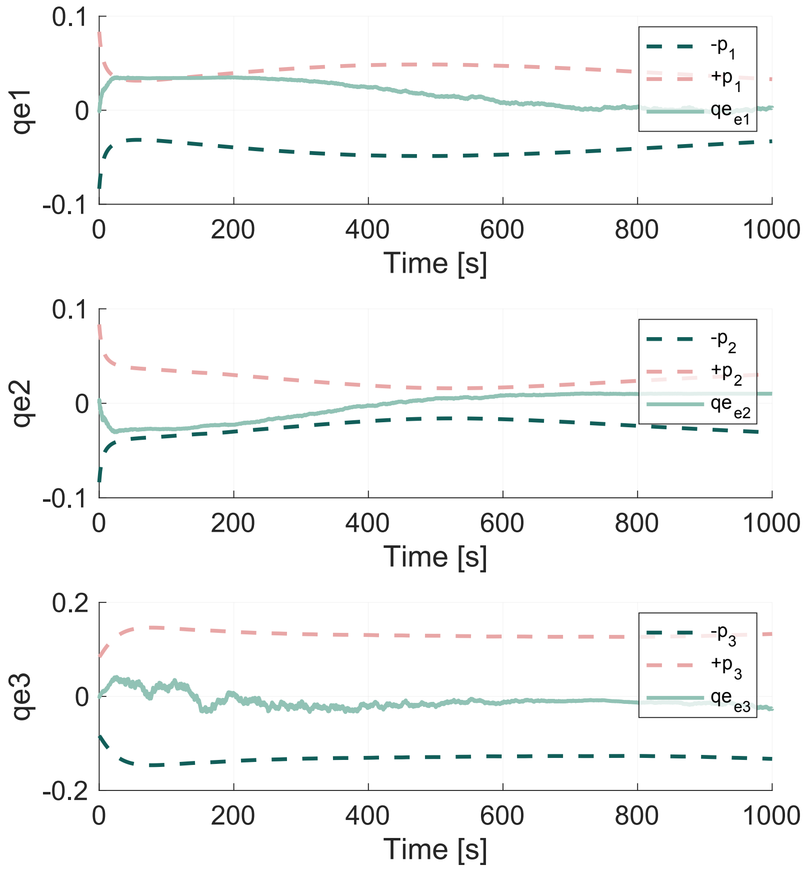


Figure 6.12: EKF with one magnetometer - q error with sigma

partial directional information in two dimensions (N_y, N_z), which introduces uncertainty when reconstructing the full three-dimensional sun vector. This was also present in the sun vector propagation previously discussed.

From an observability perspective, a single vector measurement constrains only two degrees of freedom in 3D space. The rotation about the sun vector itself remains unobservable. This under-constrained configuration leads to ambiguity: multiple attitudes can produce the same projected sun vector in the sensor frame. As a result, the EKF receives a low-rank measurement update, and the Kalman gain remains small. This weakens the correction step, causing the error covariance to contract slowly and reducing the filter's responsiveness to actual state deviations.

Furthermore, because sun sensor outputs are typically normalised direction vectors, the loss of scale information can amplify small angular errors. Without redundant measurements to compensate for noise, the filter becomes highly sensitive to even minor deviations in the sensor data.

Although the overall error was lower than in the magnetometer-only case, the estimation became less reliable over time. The error increases significantly toward the end of the simulation. This is in contrast to the magnetometer case, where the state error gradually returns to zero. Although the error is low, it should be noted that so is the Q; therefore, the filter is mainly using the internal model. This navigation method is not recommended and highlights the importance of multiple reliable sensors.

EKF Parameters for Sun Sensor Integration

Simulation Setup:

- **Initial state:** $\mathbf{x}_0 = [0 \quad 0 \quad 0 \quad 1]^T$

Covariance Matrices:

- **Process noise (Q):** $\text{diag}(5 \times 10^{-9},\ 2 \times 10^{-14},\ 2 \times 10^{-9})$
- **Measurement noise (R):** $\text{diag}(0.01,\ 0.01,\ 0.2)$
- **Initial error covariance ($\mathbf{P}_0$):** $\text{diag}(7 \times 10^{-6},\ 7 \times 10^{-6},\ 7 \times 10^{-6})$

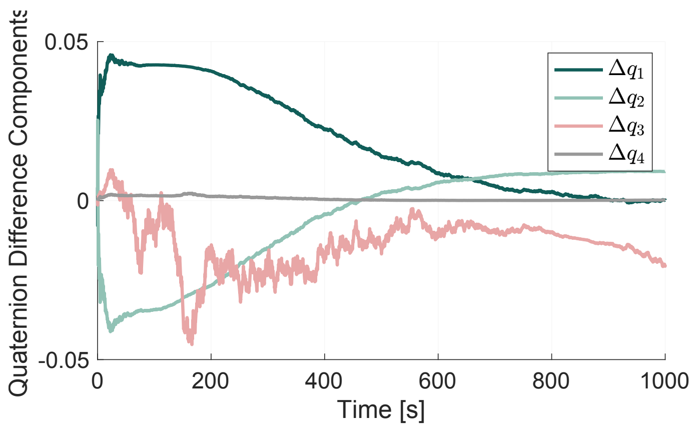


Figure 6.13: EKF with one sun sensor - q error

6.4.5 EKF with Magnetometer and Sun sensor

With both the sun sensor and magnetometer integrated into the Extended Kalman Filter (EKF), the system benefits from complementary and more stable sensing. The sun sensor's periodic data gaps are mitigated by the continuous, though noisy, magnetometer readings. Likewise, the sun sensor reinforces orientation estimation when visible, improving accuracy during periods of magnetometer drift or noise.

As shown in Figure 6.15, the combined EKF yields lower overall errors than the magnetometer-only case and is less sensitive to the instability observed in the sun sensor-only configuration. While the initial error covariance $\mathbf{P}_0$ is relatively large and the measurement noise for both sensors is high, the fused system demonstrates improved stability. Despite the magnetometer's consistent measurements, its signal magnitude is small and susceptible to noise. In contrast, the sun sensor provides stronger directional cues when in view. The upper and lower bounds of P are closer to the quarterion error indicating the filter is working as expected and is producing relevant values.

Overall, the combined EKF produces a more stable and robust estimate that remains comfortably within the covariance bounds, outperforming both individual sensor configurations. The Kalman filter can be found in Listing A.3.

Some improvements include more detailed methods of tuning. This process was done on some literature inspiration and trial and error. Therefore a Montecarlo process is recommended for future work and was not explored in this thesis due to time limitations. For a proper EKF implementation, more in-depth tests should be performed with a variety of different input and sensor data.

EKF Parameters for Magnetometer and Sun Sensor Integration

Simulation Setup:

- **Initial state:** $\mathbf{x}_0 = [0 \quad 0 \quad 0 \quad 1]^T$

Covariance Matrices:

- **Process noise (Q):** $\text{diag}(2 \times 10^{-6},\ 2 \times 10^{-6},\ 2 \times 10^{-7})$
- **Sun sensor noise ($\mathbf{R}_{\text{sun}}$):** $\text{diag}(7 \times 10^{-1},\ 7 \times 10^{-1},\ 8 \times 10^{-1})$
- **Magnetometer noise ($\mathbf{R}_{\text{mag}}$):** $\text{diag}(0.5,\ 0.5,\ 0.5)$
- **Initial error covariance ($\mathbf{P}_0$):** $\text{diag}(1 \times 10^{-3},\ 3 \times 10^{-3},\ 7 \times 10^{-3})$

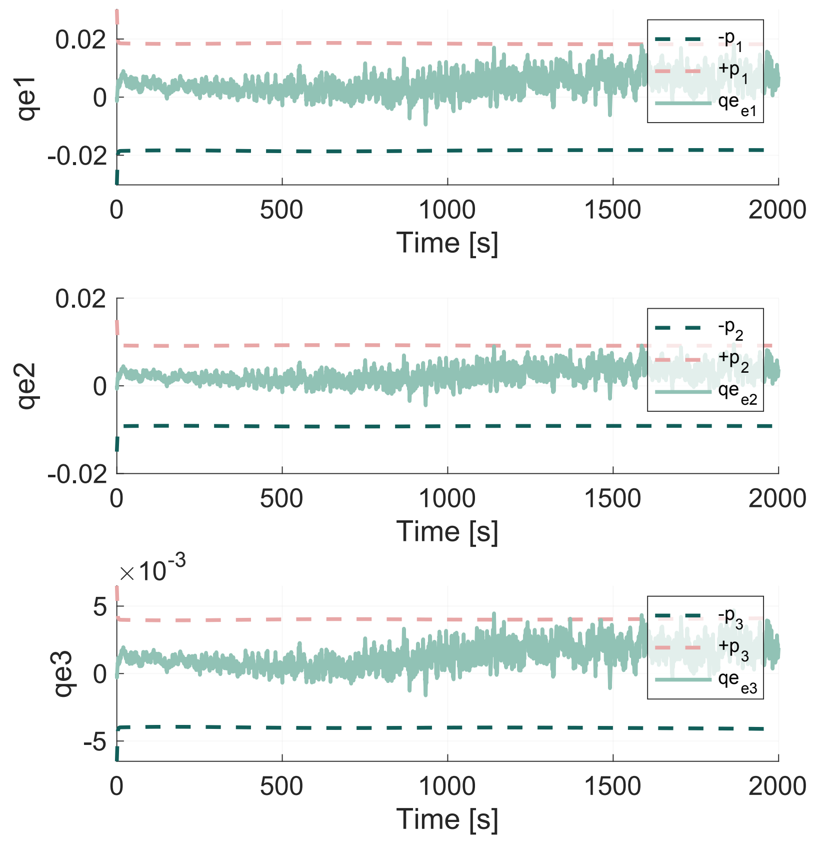


Figure 6.14: EKF with one sun sensor and one magnetometer - q error with sigma

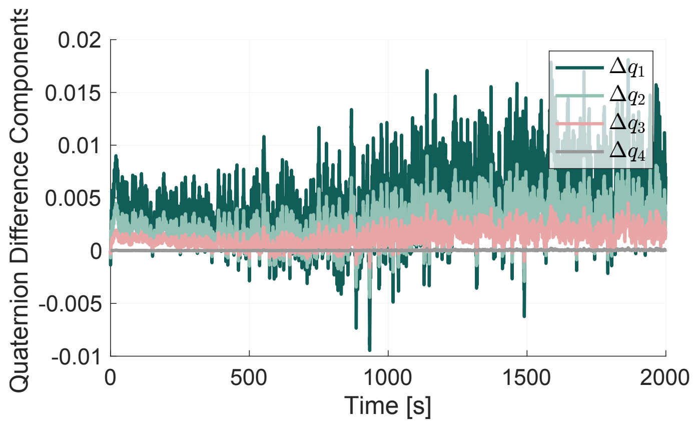


Figure 6.15: EKF with one sun sensor and one magnetometer- q error

CHAPTER 7

Control

Accurate attitude control is essential to meet the detumbling, sun-pointing and stability requirements of the reference CubeSat mission introduced in Chapter 2. Having established the dynamics (Chapter 4) and developed a real-time state estimator (Chapter 6), this chapter presents the control layer that uses those estimates to generate torques and drive the satellite toward its desired orientation.

It begins by recalling the mission control objectives, then introduces the primary control modes:

1. *Detumbling* via magnetic-rate (B–dot) control using the three-axis magnetorquer.
2. *Sun acquisition* using a proportional–derivative (PD) law on reaction wheels to align the $+z$ body-axis with the sun vector.

Each mode's mathematical law is derived in the body frame, and the actuator models, magnetorquer torque curves and reaction-wheel dynamics are described in Section 7.2.

The chapter then details the tuning process for each controller, showing how gains were chosen to satisfy settling-time, overshoot and steady-state error targets under real-time, noisy-measurement conditions. Finally, simulation results from the combined estimator–controller loop demonstrate compliance with the pointing and stability requirements on the real-time test bench introduced in Chapter 9.

7.1 Control modes

Now that the state variables required for controlling the satellite can be estimated, the control modes determine what the satellite is to do next. In this study, two modes will be used to determine what control torque is to be applied by the actuators. The two modes are detumbling and sun acquisition. Detumbling will occur when the satellite is first released or when a motion gets out of control. Detumbling will determine the rate of rotation and apply an opposite force to counteract the motion. This will be done with the magnetorquers using a b-dot algorithm. The Sun acquisition mode will be done using the sun sensors and reaction wheels to point the satellite to the desired orientation in relation to the Sun vector. This will be done with a simple PID controller.

7.1.1 Detumbling Mode

The detumbling mode aims to reduce the satellite's rotation rate, induced by its release, to a specific threshold by generating an opposing magnetic field using magnetorquers. This mode employs an open-loop control system, meaning the output is independent of the feedback from the system's response.
The $\dot{\mathbf{B}}$ control law determines the dipole moment $\mathbf{M}$, as shown in Equation (7.1), where $\mathbf{K_b}$ is the controller gain, which can be tuned for better performance.

$$\mathbf{M} = -\mathbf{K_b} \times \dot{\mathbf{B}} \tag{7.1}$$

Determining the rate of change of the magnetic field, $\dot{\mathbf{B}}$, can be challenging if the angular velocity is unknown. In such cases, the change in the magnetic field over a time step can be used as an approximation. However, if the angular velocity $\boldsymbol{\omega}$ is known, $\dot{\mathbf{B}}$ can be estimated more accurately using:

$$\dot{\mathbf{B}} \approx \mathbf{B}_0 \times \boldsymbol{\omega} \tag{7.2}$$

7.1.2 B-dot Algorithm Implementation

The B-dot algorithm was implemented is shown the simulation as a passive attitude stabilisation method. The model created can be found in Figure 7.1. Unlike active control strategies, B-dot is not fast-acting; complete detumbling can take several hours, depending on the initial angular velocity and environmental conditions. The algorithm operates using the control law defined in Equation (7.1), which applies a magnetic dipole moment proportional to the negative time derivative of the local magnetic field.

Since the Earth's magnetic field is relatively weak in low-Earth orbit—typically 20–60 μT—the torque produced by magnetorquers is small. For that reason, a comparatively large gain, $K_{\dot{B}}$, was chosen so the controller can meaningfully influence the satellite dynamics. The value is set on the high side to make the drop in angular velocity visible within the short simulation window.

Despite these adjustments, B-dot control remains fundamentally limited: magnetorquers cannot generate torque about the Earth's magnetic field vector itself. Consequently, angular velocity components aligned with the field (typically the ω_3 component in body coordinates) are not directly damped. This explains the persistent drift even as other axes show effective damping. The tuning process was done with a trial and error process starting at one over the expected magnetic field magnitude. The tuning process was not very sensitive and values could be tuned freely. It was expected that the angular velocity would decrease over time.

The simulation was run over a 1000-second window. Although this is a relatively short duration for evaluating B-dot performance, a gradual decline in angular velocity was observed. The initial angular velocity was intentionally low to highlight the damping trend, in contrast to the more extreme spin rates encountered immediately after deployment. Since this thesis focuses on test methods rather than full-on-orbit detumbling behaviour, the simplified scenario was deemed sufficient for demonstration.

For comparison, a control-off case was also simulated. In the absence of B-dot control, the spacecraft's angular velocity increased steadily, approximately 0.01 rad/s every 150 seconds, due to accumulated disturbances and numerical integration drift. This highlights the importance of incorporating detumbling logic in early mission phases. The B-dot controller effectively reversed this trend and initiated angular momentum damping, verifying its function within the scope of the test.

Bdot controller

B-dot Control Gain:

- $K_{\dot{B}} = [2 \quad 2 \quad 2] \times 10^5$ Nm $\cdot$ s/T

Used in magnetic rate damping: $T = -K_{\dot{B}} \cdot \dot{\mathbf{B}}$

7.1.3 Sun Acquisition Mode

The Sun acquisition mode utilises a PD (Proportional-Derivative) controller to align the satellite with the Sun based on data from the Sun sensor. The goal is to minimise the error between

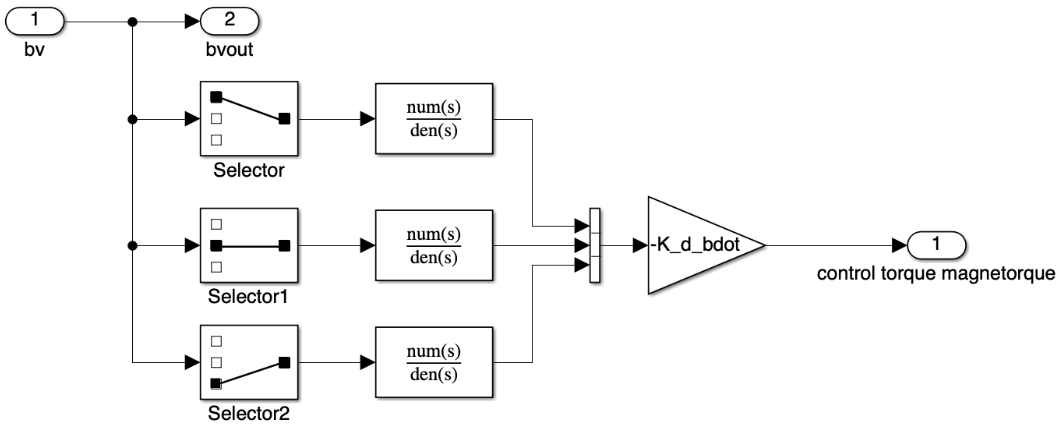
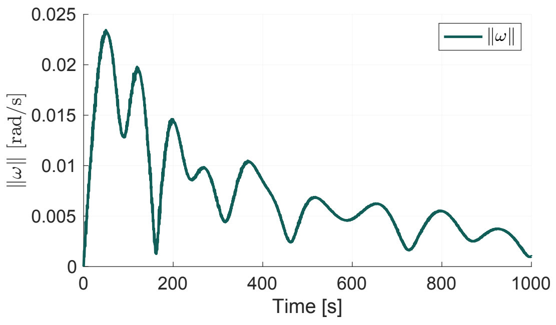
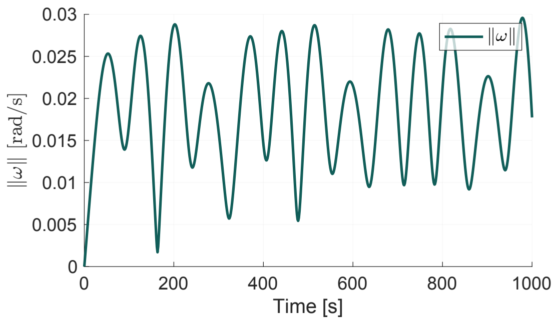


Figure 7.1: BDot Controller



(a) With B-dot controller applied



(b) Without B-dot controller

Figure 7.2: Comparison of angular velocity magnitude with and without B-dot controller.

the satellite's current state and its desired orientation by controlling the input variables accordingly.
A PD controller operates by addressing two components:

- The proportional term (P) depends on the present error.
- The derivative term (D) predicts and counteracts future errors based on their rate of change.

Each term has a gain parameter (K_p and K_d), which can be tuned to optimise the system's performance. These gains can be constant or time-dependent, depending on the specific requirements of the system.
The proportional controller establishes a direct relationship between the control input $u(t)$ and the error signal $e(t)$.

$$u(t) = K_p e(t) \tag{7.3}$$

Here, K_p is the proportional gain, which acts as an amplifier for the error signal. High gain values can lead to instability due to excessive system responses and noise amplification, whereas low gains result in slower corrections and possibly divergence of errors.
The derivative controller addresses the rate of change of the error, thereby enhancing system stability by reducing overshoots:

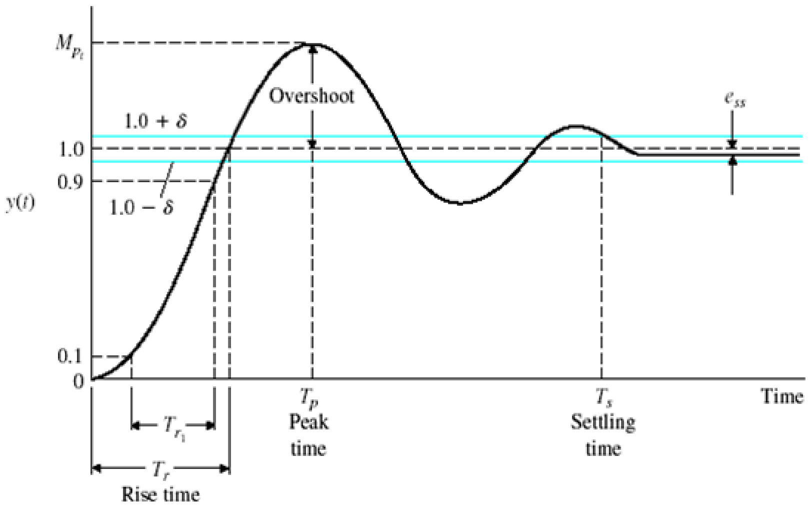


Figure 7.3: Transient and steady-state response analysis (Mooij, 2022b)

Table 7.1: Effects of gain manipulation (Mooij, 2022b)

Parameter	**Rise time**	**Overshoot**	**Settling time**	**Steady-state error**	**Stability**
$\mathbf{K_p}$	Decrease	Increase	Small change	Decrease	Degrade
$\mathbf{K_d}$	Minor decrease	Minor decrease	Minor decrease	No effect in theory	Improve (if $\mathbf{K_d}$ is small)

$$u(t) = K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{7.4}$$

By combining all components, the PD controller can leverage their strengths while minimising individual weaknesses. The complete PD equation is given in Equation (7.5):

$$u(t) = K_p e(t) + K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{7.5}$$

The impact of tuning each gain parameter on system performance is summarised in Table 7.1, while Figure 7.3 illustrates the transient and steady-state response characteristics.

7.1.4 PD Controller Implemetation

A proportional-derivative (PD) controller will use reaction wheels to actuate control torques. A more detailed discussion of reaction wheels will follow. The controller aimed to align the satellite's body-fixed $+X$ axis with the Sun vector, thereby achieving Sun-pointing attitude stabilisation. The gain values used are summarised in the box below. The complete simulator was used to tune the PD controller. The environment perturbation and EKF were active during the tuning of this controller. The gains for Kd and Kp were tuned with trial and error. The impact of Kp was larger than that of Kd. Therefore, the tuning of Kd was more aggressive than that of Kp. Requirement SA01 indicates that the controller should have an effect within 90 minutes. This is true as the position is steady after 100 seconds.

In contrast to the B-dot algorithm, the PD controller demonstrated significantly faster and more responsive behaviour. As shown in Figure 7.4, the system reached steady-state

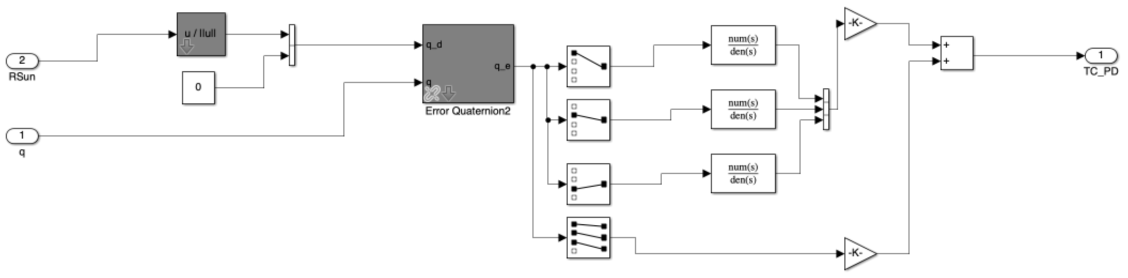


Figure 7.4: PD Controller

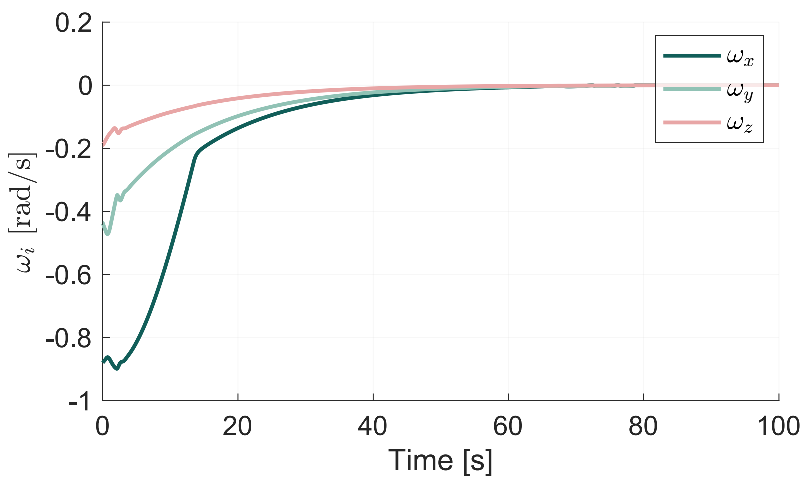


Figure 7.5: PD controller results

alignment in just under 50 seconds. The response was smooth, with no observable overshoot or oscillation, and remained stable for the remainder of the 100-second simulation.

The controller operated using noisy sensor data, adding realism to the test scenario. The gains were manually tuned to strike a balance between responsiveness and stability, without inducing instability or excessive oscillations. The proportional term provided corrective torque based on orientation error, while the derivative term added damping to suppress rapid movements.

The reaction wheels were able to produce sufficient torque to make precise adjustments, allowing the controller to compensate for sensor noise effectively. The absence of overshoot suggests that the system was not overly aggressive, and the chosen gains were appropriate for the level of noise and the dynamics involved.

It is important to note that this scenario assumes idealised actuator behaviour, with no reaction wheel saturation or momentum build-up. In practice, factors such as actuator limits, external disturbances, and eclipse periods would require additional handling. Nonetheless, within the scope of this simulation, the PD controller proved highly effective in achieving rapid and robust Sun-pointing control and serves as a useful benchmark for comparison with the slower, passive B-dot algorithm.

PD Controller Gains:

- **Proportional gain (K_p):** [0.017 0.017 0].017 Nm/rad
- **Derivative gain (K_d):** [0.15 0.15 0.15] Nm · s/rad

7.2 Actuators

With the use of the PD controller or the Bdot controller, a control torque is calculated. This is now channeled to the appropriate actuators. In this case, magnetorquer and reaction wheels will be used.

7.2.1 Magnetorquers

Magnetorquers, also known as magnetic coils or electromagnets, are devices used to generate magnetic dipole moments for controlling the angular momentum and attitude of a satellite. These systems are critical for correcting biases and attitude drifts that naturally occur during a satellite's mission. The fundamental working principle relies on applying an electric current to a series of looped coils, which generates a magnetic moment **m** perpendicular to the plane of the coils. This relationship is given in Equation (7.6), where **n** is the unit vector normal to the loop plane, A is the enclosed area, N is the number of coil loops, and I is the applied current:

$$\mathbf{m} = NIA\mathbf{n} \tag{7.6}$$

The resulting magnetic dipole **d** can be expressed as a function of the material permeability μ:

$$\mathbf{d} = \mu\mathbf{m} = \mu NIA\mathbf{n} \tag{7.7}$$

The magnetic moment generated by the magnetorquer interacts with the Earth's magnetic field to create a torque, enabling the satellite to control its orientation. The torque is generated as a result of the cross product between the magnetic dipole moment **m** and the Earth's magnetic field vector **B**.

$$\mathbf{M_{MT}} = \mathbf{m} \times \mathbf{B_B} \tag{7.8}$$

The magnetic torque $\mathbf{M_{MT}}$ is subject to various error sources that affect the system's output. These include biases ($\mathbf{b}_{MT}$), scaling factors ($\mathbf{S}_{MT}$), misalignments (cross-axis coupling $\mathbf{M}_{MT}$), and noise ($\mathbf{w}_{MT}$). The mathematical representation of the magnetic torque considering these factors is

$$\begin{aligned}\begin{pmatrix} M_{m_{x,\mathrm{MT}}} \\ M_{m_{y,\mathrm{MT}}} \\ M_{m_{z,\mathrm{MT}}} \end{pmatrix} &= \begin{pmatrix} M_{x,\mathrm{MT}} \\ M_{y,\mathrm{MT}} \\ M_{z,\mathrm{MT}} \end{pmatrix} + \begin{pmatrix} b_{x,\mathrm{MT}} \\ b_{y,\mathrm{MT}} \\ b_{z,\mathrm{MT}} \end{pmatrix} \\ &+ \begin{bmatrix} s_{x,\mathrm{MT}} & m_{xy,\mathrm{MT}} & m_{xz,\mathrm{MT}} \\ m_{yx,\mathrm{MT}} & s_{y,\mathrm{MT}} & m_{yz,\mathrm{MT}} \\ m_{zx,\mathrm{MT}} & m_{zy,\mathrm{MT}} & s_{z,\mathrm{MT}} \end{bmatrix} \begin{pmatrix} M_{x,\mathrm{MT}} \\ M_{y,\mathrm{MT}} \\ M_{z,\mathrm{MT}} \end{pmatrix} + \begin{pmatrix} w_{x,\mathrm{MT}} \\ w_{y,\mathrm{MT}} \\ w_{z,\mathrm{MT}} \end{pmatrix} .\end{aligned} \tag{7.9}$$

To compute the torque in the satellite's body frame, the Earth's magnetic field vector **B** must first be transformed from the inertial frame using the attitude matrix $\mathbf{C}_{BI}$:

$$\mathbf{B_B} = \mathbf{C}_{BI}\mathbf{B_I} \tag{7.10}$$

The magnetorquer output depends on the commanded dipole moment **m**, which is controlled by the magnetic control law and the geomagnetic field. The performance of the magnetorquer is further influenced by hardware limitations such as output delays, saturation, and quantisation errors, which can be included in the model if necessary to ensure realistic simulations.

Reference mission

The magnetorquers in the reference mission will have the following errors and characteristics.

Table 7.2: Magnetorquer model parameters used for simulation.

Parameter	Value	Unit
Time Constant	0.2216	s
Resolution	0.2000	Am^2
Max Moment	400	Am^2
Min Moment	0.4889	Am^2
Bias Moment	2	Am^2
Nonlinearity Gain	1	-

7.2.2 Reaction Wheels

Reaction wheels are essential components of satellite attitude control systems, providing precise and smooth adjustments to angular momentum without the use of propellant. They are widely used for stabilising the satellite and absorbing cyclic loads during manoeuvres. Reaction wheels operate by varying the angular velocity of spinning wheels, which generates angular momentum due to the conservation of angular momentum principle. This angular momentum is transferred to the satellite body, enabling precise orientation changes.
The angular momentum of a reaction wheel, h_{rw}, is expressed as:

$$h_{\mathrm{rw}} = \mathbf{I}\,\boldsymbol{\omega} \tag{7.11}$$

where I is the moment of inertia of the spinning wheel, and ω is its angular velocity. The total external torque acting on the satellite can be expressed as:

$$\mathbf{M}_{\mathrm{ext}} + \mathbf{M}_c = \dot{\mathbf{h}}_{\mathrm{cm}} + \boldsymbol{\omega} \times \mathbf{h}_{\mathrm{cm}} \tag{7.12}$$

Here, $\mathbf{M_{ext}}$ is the external torque, $\mathbf{M_c}$ is the commanded control torque, h_{cm} is the angular momentum of the satellite body, and ω is the angular velocity of the satellite relative to the inertial frame. Reaction wheels apply control torques by adjusting their rotational speed, which generates a counteracting torque on the satellite due to conservation of angular momentum.

A typical reaction wheel assembly consists of three orthogonally mounted wheels, with an optional fourth wheel for redundancy. In a pyramid configuration, the wheels are mounted on the XY plane and tilted toward the Z-axis by an angle β. This configuration ensures efficient torque application around all three axes. The relationship between the commanded control torque $\mathbf{M}_c$ and the wheel torques $\mathbf{M}_w$ is represented as:

$$\mathbf{M}_c = \mathbf{V}\,\mathbf{M}_w \tag{7.13}$$

where V is the configuration matrix. For the pyramid configuration, V is given by:

$$\mathbf{V} = \begin{bmatrix} \cos\beta & 0 & -\cos\beta & 0 \\ 0 & \cos\beta & 0 & -\cos\beta \\ \sin\beta & \sin\beta & \sin\beta & \sin\beta \end{bmatrix} \tag{7.14}$$

The wheel torques can then be calculated using the pseudo-inverse of V:

$$\mathbf{M}_w = \mathbf{V}^{-1}\,\mathbf{M}_c \tag{7.15}$$

Reaction wheels are subject to several physical and operational characteristics that influence their performance. Friction is one of the primary factors and consists of Coulomb friction, which is constant and depends on wheel rotation direction, and viscous friction, which depends on wheel speed. The total friction torque can be expressed as:

$$T_{\text{fric}} = N_C\,\text{sgn}(\omega) + f_s \tag{7.16}$$

where N_C is the Coulomb friction coefficient, and f_s is the viscous friction coefficient. In addition, misalignments in the mounting of reaction wheels introduce errors that are accounted for using a matrix $\mathbf{M}_w$, representing global or individual wheel misalignments. Other considerations include saturation, where the maximum angular momentum and torque achievable by the wheels are limited by their design, and output delays, which affect performance in fast-changing dynamic conditions.
The reaction wheel model incorporates a motor constant k_m, which relates the commanded current I_c to the motor torque $\mathbf{M}_m$:

$$\mathbf{M}_m = k_m\,\mathbf{I}_c \tag{7.17}$$

In practical applications, the delivered moment accounts for various error sources and is expressed as:

$$\mathbf{M}_w = \mathbf{V}^{-1}\,\mathbf{M}_c \;-\; k_v\,\boldsymbol{\omega}_w \;+\; k_c\,\text{sgn}\big(\boldsymbol{\omega}_w\big) \tag{7.18}$$

where k_v is the viscous friction coefficient vector, and k_c is the Coulomb friction coefficient vector. These factors allow for realistic modelling of reaction wheels and enable accurate control of satellite attitude in simulations.

Reference mission

For this reference mission, the following characteristics are used.

Table 7.3: Reaction Wheel model parameters used for simulation.

Parameter	Value	Unit
Bearing Noise Std Dev.	0	Nm
Max Torque Limit	0.075	Nm
Time Delay	0.02	s
Resolution	2×10^{-5}	Nm
Initial Torque	0	Nm
Initial Momentum	0	Nms

CHAPTER 8

Functional Simulator

This chapter outlines the methodology for developing the simulator. The simulator was created using MATLAB/Simulink, incorporating components from prebuilt libraries and custom modules. The software architecture is first described in Section 8.1, outlining key characteristics of the simulation such as solver choices and Model characteristics. Integration and acceptance tests were conducted to validate the simulator's performance prior to porting it to Eurosim.

8.1 Simulator Overview and Architecture

The purpose of the simulator is to create an environment that models the conditions in which the CubeSat will operate and to ensure that the satellite system can respond effectively to any changes. Initially, the simulator will be built using MATLAB Simulink. Once completed, it will be ported to Eurosim, the real-time simulation environment. Part of the simulation will run on a Raspberry Pi, enabling the user to send simulated commands to the virtual satellite for a SIL demonstration. Later, a hardware component can be added, allowing hardware sensors to interact with the simulator and send commands, resulting in a partial HIL demonstration. The main objective of this study is to determine if testing of a CubeSat ADCS using affordable resources is possible.
Figure 3.2 illustrates the top-level setup of the simulator. The environment models incorporate various aspects that the satellite will encounter, which will be discussed in more detail later in this section. These environment models serve as inputs to the sensor models, where expected errors (such as bias, saturation, and drift) are applied based on the specific sensors used. The output from the sensors is then processed by a Kalman filter, which estimates the states using the noisy measurements. These estimated states are utilised by the controller to generate actuator commands, which are subsequently executed in the actuator model. The actuator model adjusts the environment to close the loop. Each component will be described, and verification tests will be conducted to ensure that they exhibit the expected behaviours.

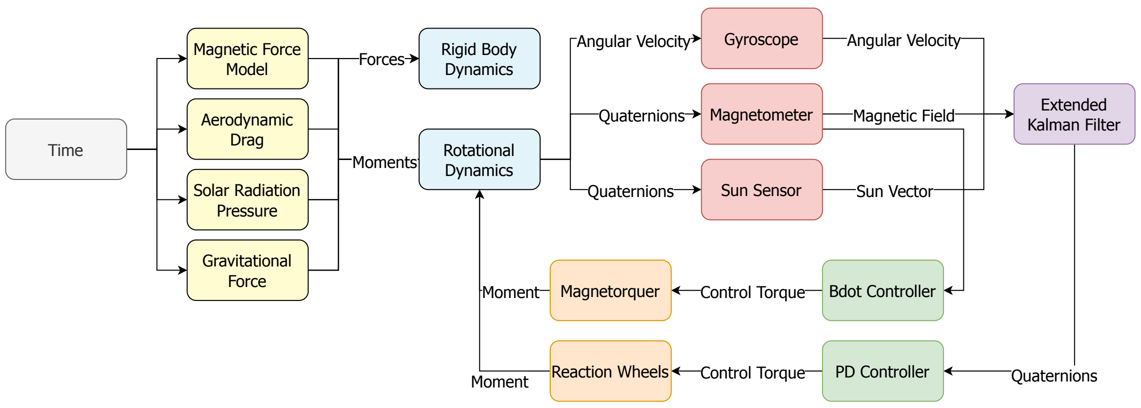


Figure 8.1: Flowdiagram of Simulator with I/O

8.2 Generic Rendezvous And Docking Simulator (GRADS)

To build up aspects of this simulator, prebuilt models were used. The GRADS library is a set of MATLAB Simulink models, originally built as a method of testing GNC systems for docking and rendezvous missions to clean up space debris (Mooij, 2022a). The simulator library contains sensors and actuators component models, as well as propagators, environment models, and utilities, which include reference frame transformations, time elements and mathematical operators.
The benefit of using the GRADS library compared to creating a new set of models is the modularity of the library. The models can be placed and tuned depending on the mission case. To be properly integrated into the simulator, several acceptance tests were performed, although the models themselves have been separately validated.

8.3 Solver Choice and Analysis

The simulator was implemented in Simulink and later ported to Eurosim for further analysis. Simulink provides a variety of numerical solvers, broadly categorised as discrete (fixed-step) and variable-step. By default, Simulink applies the ode45 variable-step solver, which is general-purpose and dynamically adjusts the integration step size to balance accuracy and performance. However, this default solver is not suitable for real-time systems. In embedded environments such as the onboard computer (OBC) of a CubeSat, the controller must run at fixed time intervals, with no dynamic adjustment of execution frequency. Additionally, Eurosim requires all models to use a discrete (fixed-step) solver, meaning the model must be explicitly configured for such execution.

- **Variable-step solvers** (e.g., *ode45*) adjust time steps based on estimated local error. They are effective in simulation for achieving accuracy with fewer steps but are *unpredictable in execution time* and unsuitable for deployment.
- **Fixed-step solvers** (e.g., *ode4*) use a constant time step, ensuring *predictable timing and compatibility with real-time systems*. This makes them ideal for embedded applications where timing determinism is critical.

To evaluate the effects of solver choice, a comparative analysis was performed using three solver configurations for a 1700-second simulation of the environment model:

1. *ode45* – variable-step, with a maximum tolerance of $10^{-}14$ and maximum step size of 40 seconds. (Simulink default)
2. *ode4* – fixed-step with a 10-second interval
3. *ode4* – fixed-step with a 0.01-second interval (100 Hz, used as reference)

The 0.01 s (100 Hz) configuration was selected as the reference because it closely matches the control update rates commonly used in OBC flight software. While individual components in the system (e.g., sensors, actuators) may operate at different frequencies, this rate provides a high-fidelity baseline for evaluating solver accuracy.
Results as shown in Figure 8.2 showed that:

- The *ode45* (variable-step) solver exhibited variable error across time, as expected from its adaptive behaviour.

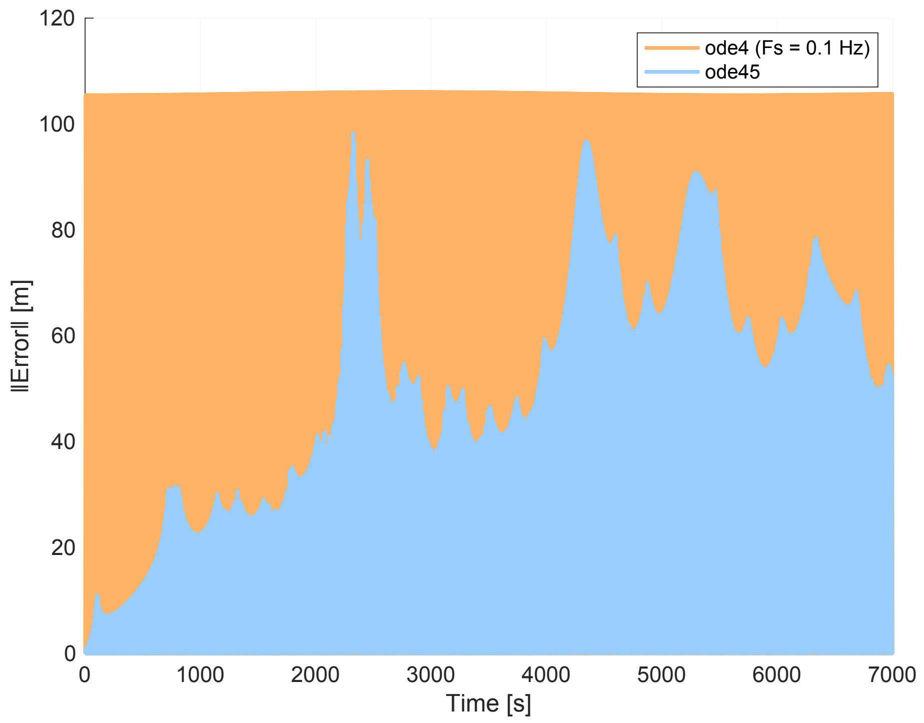


Figure 8.2: Frequency Analysis (normal error)

- The *ode4* with 10-second steps resulted in smooth but consistently offset outputs, due to the coarser temporal resolution.

These differences are not just academic, they highlight the importance of solver configuration in systems where real-time performance and accuracy are both crucial.
In addition to accuracy, solver selection directly impacts computational efficiency—a key constraint in CubeSat missions. Variable-step solvers incur higher CPU usage due to frequent internal calculations and error estimation. This can lead to increased power consumption and may exceed the capabilities of power- and compute-constrained OBCs.
By contrast, fixed-step solvers provide predictable execution at the cost of some precision. For this reason, choosing a fixed-step solver with a step size aligned to the system's control frequency (e.g., 10–100 Hz) offers a good trade-off between real-time feasibility, accuracy, and power efficiency.
This analysis informed the decision to use ode4 with a step size of 0.01 seconds as the reference configuration, and to compare all other results against it for evaluating real-time compatibility and model fidelity.

8.3.1 Environment Models

The environment model block is separated into the physics models and the forces and moment models. The physics models represent the fundamental physical phenomena that affect spacecraft dynamics and interaction in the space environment. The forces and moments models then take these models and convert them into what the satellite will feel in terms of forces and moments. These physics models include:

1. Celestial body positions: Sun and Moon position models in ECI (J2000) coordinates that are needed for gravity and illumination calculations.

2. Gravitational models of varying complexity:
    - Simple Gravitational Acceleration model for basic gravity effects
    - Spherical Harmonics Gravitational Acceleration for more complex gravity field modelling
    - Central Gravity Field Acceleration for point-mass approximations
    - Third-body acceleration to account for gravitational effects from bodies other than Earth
3. Magnetic field models:
    - Magnetic Dipole model for basic magnetic field representation
    - Central Magnetic Field model for Earth's main field
    - Spherical Harmonics Magnetic Field for more detailed field mapping
    - Simplified Magnetic Dipole for efficient approximations
4. Time reference models, particularly the Greenwich Mean Sidereal Time (GMST) which provides accurate time reference in the J2000 frame
5. Solar-related models:
    - Solar Flux models (both variable and constant)
    - Illumination factor between the Sun and Earth
    - Eclipse Sun-Earth status detection
6. Directional models:
    - Normalised Sun Direction in body frame
    - Normalised Earth Direction in both ECI and body frame
7. Atmospheric models, including the Fixed-Time Simplified NRLMSISE-00 density model for atmospheric drag calculations

Given that the case study mission is a LEO orbit, the following models from the environment model library were used.

- **Gravitational model:** Spherical Harmonics Gravitational Acceleration (vertical frame - NED (North East Down))
- **Magnetic field model:** Spherical Harmonics Magnetic Field (vertical frame - NED)
- **Time reference model:** Greenwich Mean Sidereal Time (J2000)
- **Solar model:** Varying Solar Flux model and Eclipse Sun-Earth model.
- **Atmospheric model:** Fixed-Time Simplified NRLMSISE-00 density

From here, the forces and moments blocks had to be chosen. GRADS has a large number of flexible modules which were chosen to fit this reference mission. The first being the 'Aerodynamic Drag Force and Moment: Box', where 'Box' refers to the dimensions of the box could be given to better determine the force generated by the density. The spherical harmonics, which output the gravitational acceleration, were also converted to a force given the dimensions and

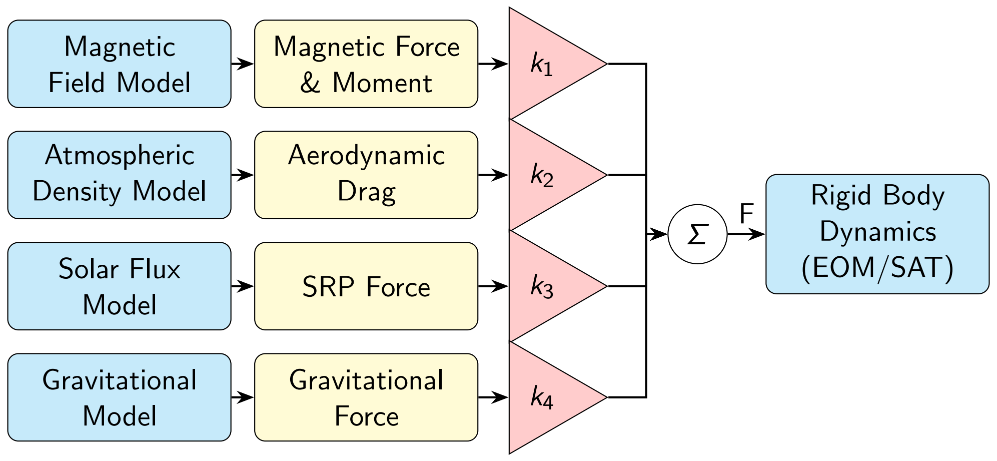


Figure 8.3: Integrated environment model showing selective activation of force models using coefficient parameters (k_1 to k_4).

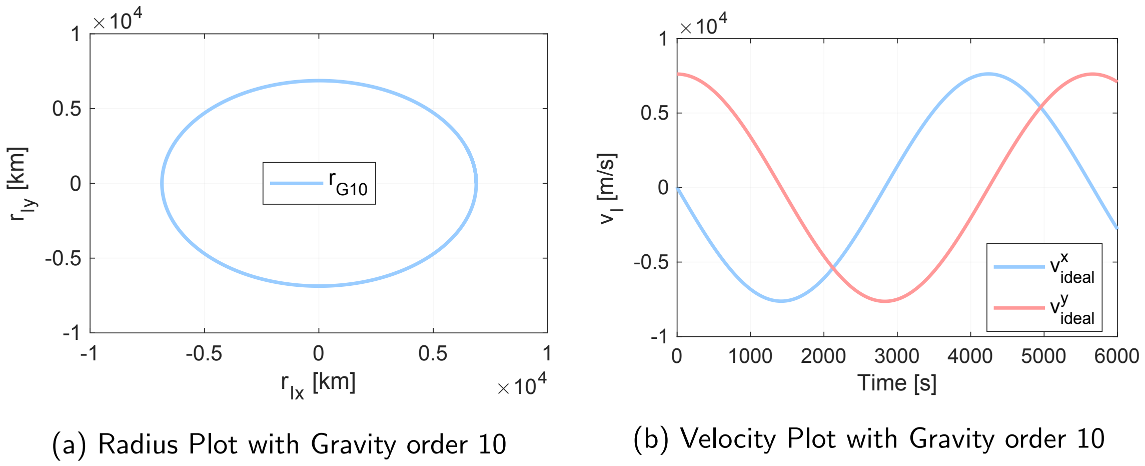
mass of the satellite. The magnetic field is considered the magnetic field generated by the satellite to determine what it could feel, and finally, the solar radiation pressure determines the force given the satellite's illumination factor and solar radiation pressure. From here, the forces are fed into the Rigid body propagator to determine the velocity and position in the inertial frame. The moments are fed into the Rotational Dynamics block, which determines the angular velocity as well as the quaternions of the satellite.

8.3.2 Acceptance testing for GRADS components

Even though the GRADS library has already been verified and tested, an acceptance test for this application is needed. All used environment components were connected to the Equations of Motion (EOM) for the satellite to show the combined satellite dynamics. Each environment block can be activated, and the effects can be measured by setting a variable 'K' seen in Figure 8.3 to '0' (deactivate) to '1' (activate).
The acceptance tests were done by introducing the different perturbations individually while monitoring the changes in the radius and velocity. The magnitudes of each perturbation have been discussed in Chapter 5, showing that gravity and drag are of a larger magnitude as compared to solar radiation pressure and the magnetic field. Using this logic, the environment models were integrated. Further description of the tests performed can be seen in Table 5.1.

8.3.3 Propagators and Utilities

There are several propagators available in the GRADS library, each suited for different types of orbital and attitude dynamics simulations. The simplest orbit propagators do not require an environmental block and only account for the effects of a central gravity field. More advanced options include those for rigid-body translational motion, such as Cowell's method (3 Degrees of Freedom (DOF)) and the Clohessy-Wiltshire equations. Additionally, there are multiple formulations for rigid-body rotational dynamics, allowing the use of Euler equations, quaternions, or Modified Rodrigues Parameters (MRP) depending on the application. The rigid-body rotational dynamics models also offer coupling with reaction wheels, making them useful for attitude control system simulations. Furthermore, 6-DOF rigid-body models are available, along with variable-mass variations, which are crucial for scenarios where fuel consumption significantly affects the system's behaviour.



(a) Radius Plot with Gravity order 10 (b) Velocity Plot with Gravity order 10

Figure 8.4: Gravity Model validation showing radius and velocity plots with gravity order 10

For this simulation, Cowell's method is chosen to propagate the translational motion. Since this simulation involves a single spacecraft in a simple Low Earth Orbit (LEO) without any relative motion considerations, Cowell's method is sufficient. It provides direct numerical integration of Newton's equations of motion without the need for additional assumptions, making it a straightforward and general approach for this scenario. For the spacecraft's attitude dynamics, quaternions are used instead of Euler angles to avoid singularities (gimbal lock), which can occur in the latter.
Other components, such as reference frame transformations and time converters (simulation time to Modified Julian Date (MJD)), were also used.

8.4 Requirements

This section reflects on the requirements given in Chapter 2 and discusses how each requirement was met. The requirements are split between sensor and actuator performance and controller performance.

Table 8.1: Compliance with sensor and actuator hardware requirements.

Requirement	Compliance Summary
SS01	The simulation used a 0.1 s time step, meeting the sun sensor sampling rate requirement.
SS02	Sensor noise was modelled with a 1° standard deviation (3σ), meeting the accuracy requirement.
SS03	The PD controller aligned with the Sun within 50 s, well below the 10-minute threshold.
SS04	The field of view was set to 30°, fulfilling the coverage requirement.
SS05	Sun sensor noise was modelled at 0.033°, which is within the 0.05° limit.
MM01	The magnetometer sampled at 0.02 s intervals, exceeding the 0.5 s requirement.
MM02	Simulated accuracy was better than 5 nT (1σ), meeting the requirement.
GYR01	Model noise allowed sensitivity near 0.01°/s, aligning with the requirement.
GYR02	The gyroscope sampling rate was 0.01 s (100 Hz), as required.
RW01	Reaction wheels provided 0.075 Nm, exceeding the 0.05 Nm minimum at 6000 rpm.
RW02	Drift was modelled as negligible, well below the 0.01°/s limit.
MT01	Magnetorquer output ranged from 0.4 to 400 μNm, exceeding the 0.005 Nm requirement.

Table 8.2: Compliance with control mode and performance requirements.

Requirement	Compliance Summary
DT01	Simulation duration of 1000 s is well within the 2-orbit (6000 s) limit.
DT02–DT04	Angular rates in all axes were reduced below 0.5°/s, fulfilling all detumbling axis requirements.
SA01	Sun alignment occurred within 50 s using the PD controller, satisfying the 90-minute limit.
SA02	Not explicitly tested; detumbling and acquisition were performed separately. Integration is recommended for future work.
SA03	Pointing accuracy during acquisition was consistently below 0.5°, well within the 5° requirement.
SA04	The Sun was found within 1 orbit (90 minutes), as required.
GNC01	The control loop operated at 10 Hz, meeting the required update rate.
GNC02–GNC05	For the PD controller, rise time, settling time, and overshoot were all within acceptable limits. These were not evaluated for the B-dot controller.

CHAPTER 9

Real Time Simulation

This chapter describes the translation of our CubeSat ADCS models from offline MATLAB/Simulink into a fully real-time test bench. Building on the functional simulation and control design in Chapter 8 and Chapter 7, it demonstrates how EuroSim, together with OrangePi and Raspberry Pi 5, can execute the same environment, sensor and controller code under hard real-time constraints.
First, Section 9.1 reviews the software choices. MATLAB for model development and EuroSim for the real-time aspects, and justifies them in terms of cost and license availability. Section 8.2 then describes the hardware platform, detailing the Raspberry Pi 5 and Orange Pi 5 Plus configurations used to run EuroSim and interface with ADCS hardware. Section 9.3 explains the MATLAB code-generation workflow, including model structuring, code export via rtwbuild and the workarounds required by its limitations. Section 9.4 provides a step-by-step guide to importing those generated modules into EuroSim's ModelEditor, ScheduleEditor and SimulationCtrl interfaces, highlighting the dos and don'ts discovered.
To ensure fidelity, Section 9.5 presents acceptance tests that verify each imported Simulink component behaves as expected in the real-time environment. Section 8.6 then details the final integrated model, showing how the EKF and PD/B-dot controllers run at 100 Hz and produce identical results to the offline benchmark. Finally, Section 8.7 revisits the original requirements, demonstrating how the real-time bench meets timing, accuracy and cost targets set out in Chapter 2. Together, these sections complete the thesis's aim of delivering an affordable, reproducible path from simulation to hardware-in-the-loop ADCS testing.

9.1 Software

This section gives a short description of the software used for the simulation development, not including real-time simulation set-up.

9.1.1 MATLAB

In Chapter 2, all three large-scale satellites were found to have used MATLAB for their controller design and testing. This, of course, is not a realistic view of the control designs of all satellites, but does indicate the wide-scale use of the software. Looking at the MATLAB website, it was found that projects such as the NASA Synchronised Position Hold Engage Reorient Experiment Satellites (SPHERES), which are control algorithm testing robots that move around the International Space Station (ISS), are first tested and simulated in MATLAB and Simulink according to the Mathworks customer stories page.[1]. Another similar story names Lockheed Martin Space, Indian Space Research Organisation, as well as Kenya Space Agency, to name a few larger organisations, which use MATLAB to test and simulate their systems and subsystems. This indicates the wide-scale use of the software in the space sector. Therefore, using something which is used in industry will make the current testbed more realistic.

MATLAB contains several features and add-ons which can be used for a variety of system testing purposes. For this thesis project, the add-ons used will be Simulink, Aerospace Blockset,

[1]https://nl.mathworks.com/company/user_stories/researchers-test-control-algorithms-for-nasa-spheres-\satellites-with-a-matlab-based-simulator.html Last accessed 23/01/24

Table 9.1: Matlab package pricing

Package	**Standard price (EUR per year)**	**Home price (EUR per year)**	**Student price (EUR per year)**	**Academic price (EUR per year)**
MATLAB	900	119	69	262
Simulink	1360	35	20	262
Simulink 3D Animation	520	35	20	105
Aerospace Blockset	780	35	20	105
Aerospace Toolbox	560	35	20	105

Aerospace Toolbox, and Simulink 3D Animation. Simulink is an add-on which allows you to use a block diagram environment to design and simulate systems. In this case, this is the control system. This does not need specifically written code, only for additional functions, but can be done fully by plugging together sub-system blocks. Simulink 3D Animation can then be used to animate the systems created using Simulink. Therefore, given the popularity and the features present, MATLAB will be used as the base language.

Costs

The pricing of MATLAB alone is relatively high.[2] Mathworks charges extra for the add ons. The standard, home, academic and student use licenses and the prices are indicated in Table 9.1. The standard price is used for governments and large-scale companies. The home price is for personal use. Finally, academic packages are for educational institutes. As shown in the table, the personal use packages (student and home) are considerably cheaper than the standard and academic prices; however, the standard price is nowhere near the academic package. As the purpose of this thesis is to benefit academic-run projects, the student and academic prices are considered in the total budget of the project.
For this thesis, an academic license is being used as supplied by the Delft, University of Technology.

9.1.2 Eurosim

Eurosim[3] is a simulation tool used for real-time simulations and hardware in the loop testing. It was first developed in 1996 for the European Robotic Arm program but has continued to be developed and used for more recent projects such as Gaia and Herschel-Planck. It is an engineering simulator framework which supports designs in the verification and development stages and can be used with MATLAB. Eurosim can be integrated with other simulators and has an inbuilt SGI clock for synchronising. The software was freely available for this thesis, but it is an expensive software, which was created and improved by the EuroSim consortium, of which Airbus is one of the partners. The price is not specified on the website, but after a conversation with members at Airbus, it was concluded that this is a commercially available product, which can be acquired through consultation with the Eurosim team. Educational purposes are granted a license, however, commercial companies would need to pay a fee, which is determined from a consultation. Given that the project aims to create a low-cost, reliable simulator, this software will be used to prove the essence of this project and other open-source alternatives, such as the sat-rs software developed by the University of Stuttgart

[1]Last accessed 27/02/24 https://nl.mathworks.com/help/simulink/gs/create-a-simple-model.html
[2]https://nl.mathworks.com/pricing-licensing accessed on 24/01/24
[3]https://www.eurosim.nl/products/addons/ggncsim/index.shtml, Last accessed 10/01/24

eurosim
......for simulation solutions

Figure 9.1: Eurosim logo[2]

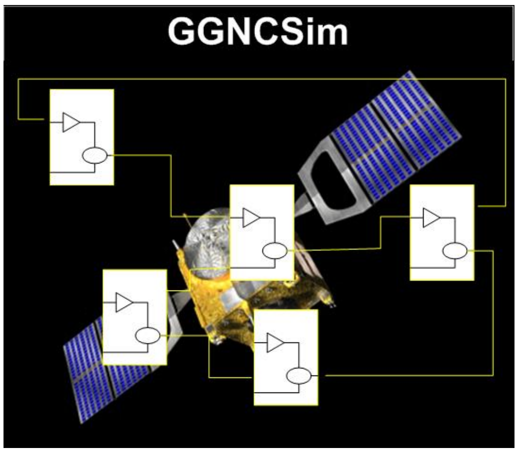


Figure 9.2: GGNCSim [4]

and Airbus, will also be considered and compared. The software allows for real-time simulations and would be used for the onboard computer simulation segment. The base language used is C code.

9.1.3 GGNCSim/GRADS

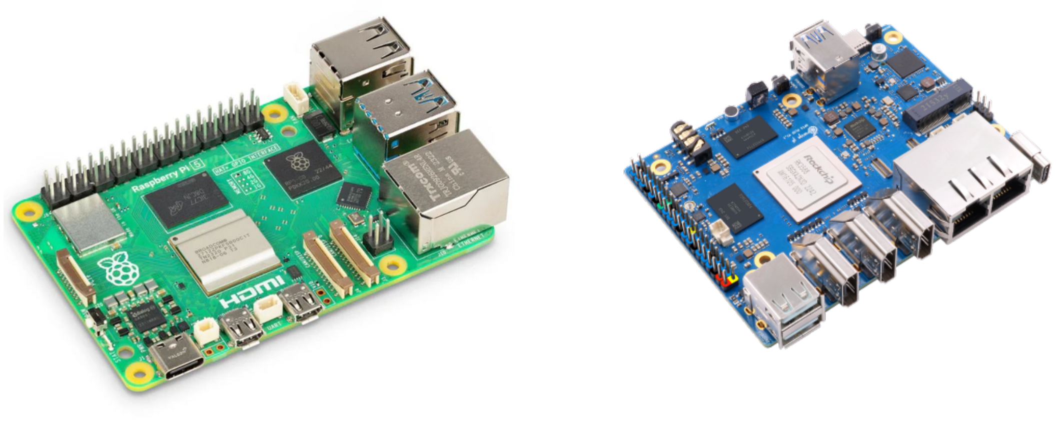
GGNCSim (Mooij and Ellenbroek, 2011) or Generic Guidance, Navigation and Control Simulation is a toolbox used to develop and test GNC simulation models. It was used to test the ADCS of Gaia, Herschel, Planck, and SPS-2 (Oomen, 2020) to name but a few. Others include non-space related projects, such as robotic arms and vehicle simulations. GGNCSim models have already been validated, which is beneficial for time in this research.
The models contain biases, nonlinearity, scale factor errors, noise, misalignment errors, saturation and quantisation.

9.2 Hardware

The hardware used and purchased for this test setup is summarised in Table 9.3 and Table 9.4. There were two ways of achieving the set-up discussed in this thesis. The tables describe the associated costs with Raspberry PI 5 in Table 9.4 and with an Orange Pi in Table 9.3. It is noted that the cost of the Raspberry Pi 5 configuration is cheaper than the Orange Pi configuration; however, the Raspberry Pi 5 has more components needed.

Initially, the Raspberry Pi 4 was considered for running the simulation software. However, after assessing the computational demands of EuroSim, it became clear that although the Pi 4 was capable, the Raspberry Pi 5 and the Orange Pi 5 Plus were more suitable options. Their specifications are compared in Table 9.5. While the Pi 4 and Pi 5 share many similarities, the Pi 5 introduces key advantages such as PCIe support and an onboard real-time clock (RTC). One limitation of lower-cost computing platforms like the Pi 4 is the absence of a hardware-based RTC, which can be critical for time-synchronised, real-time tasks.
To investigate the impact of the system clock on time-stamping accuracy, a BNO055 9-axis

[2]Last accessed 27/02/24 https://nl.mathworks.com/help/simulink/gs/create-a-simple-model.html
[4]https://www.eurosim.nl/products/addons/ggncsim/ggnclogobig.jpgLast accessed 27/02/24



(a) Raspberry Pi 5 (b) Orange Pi Plus

Figure 9.3: Comparison of Raspberry Pi 5 and Orange Pi Plus

absolute orientation sensor[4] was connected to a Raspberry Pi 4 using the I2C interface. This sensor includes a 3-axis accelerometer, 3-axis magnetometer, and a 3-axis gyroscope, and it also measures temperature. The board used is an LSM9DS1, operating at 3V. The IMU pin configuration is shown in Figure 9.4.

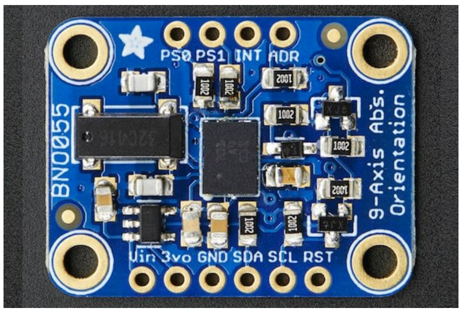


Figure 9.4: BNO055 IMU pins

The pinout of the Raspberry Pi 4 is shown in Figure 9.5, where pins 3 and 5 are used as the I^{2}C data line (SDA) and clock line (SCL).

[4]Last accessed 14/08/24 https://learn.adafruit.com/adafruit-lsm9ds1-accelerometer-plus-gyro-plus-magnetometer-9-dof-breakout/overview

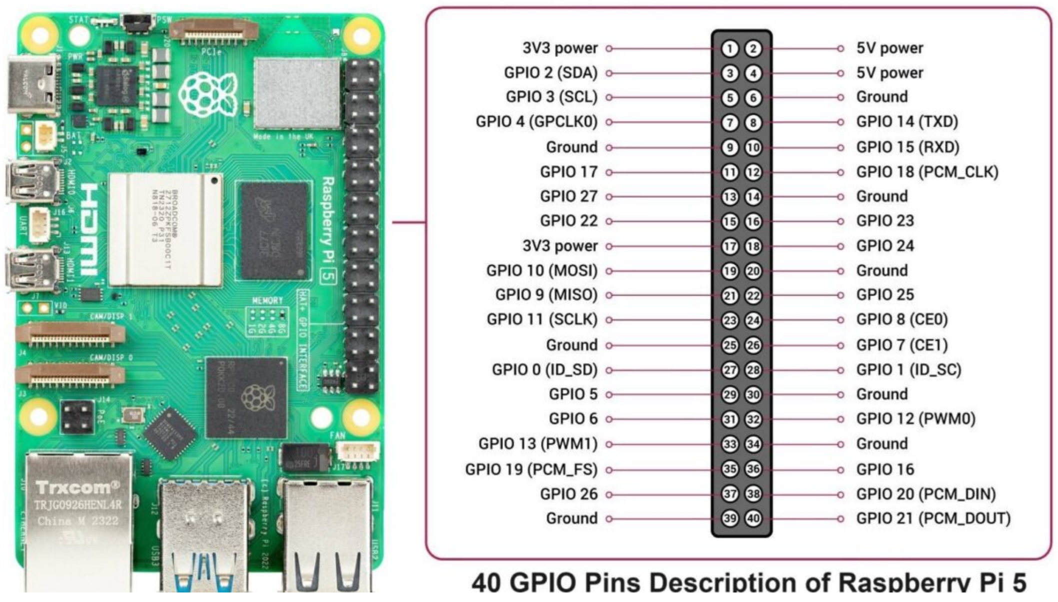


Figure 9.5: Raspberry Pi 5 pins

5

The IMU was used to collect data for 10 seconds with a script in Python. Below is a sample of the output. The first value represents the timestamp, followed by x, y, and z acceleration, gyroscope readings, magnetic field, and temperature. Although a 0.1-second interval was intended, the actual differences ranged from 0.1 to 0.12 seconds.

Listing 9.1: IMU measurements

```
1722256275.94 12.32 1.70 1.26 -0.45 0.67 -0.34 -33.00 -13.25 5.75 31.00 \\
1722256276.06 11.36 0.60 1.01 -3.81 4.22 -0.82 -32.88 -13.69 -1.25 31.00 \\
1722256276.17 1.64 -4.70 2.71 -2.79 4.44 -0.19 -27.69 -9.50 -18.25 31.00 \\
1722256276.29 -2.93 -4.88 4.96 -2.27 5.33 -0.50 -16.25 -4.50 -29.75 31.00
```

This discrepancy, although small in short durations, would compound over extended tests. To reduce this drift, an external RTC module (Seeed Studio DS1307)[6] was added to the setup. The RTC, powered independently and connected via I^{2}C (address 0x68), was used alongside the IMU (address 0x28). The schematic of the RTC is shown in Figure 9.6.

[5]Last accessed 14/08/24 https://www.hackatronic.com/raspberry-pi-5-pinout-specifications-pricing-a-complete-guide/

[6]Last accessed 14/08/24 https://wiki.seeedstudio.com/Pi_RTC-DS1307/

Table 9.2: Difference in time for IMU measurements. With and without an external RTC.

	Without RTC [s]	Difference [s]	With RTC [s]	Difference [s]
Measurement 1	1722256275.94	-	09:30:01.382	-
Measurement 2	1722256276.06	0.12	09:30:01.491	0.106
Measurement 3	1722256276.17	0.11	09:30:01.600	0.109
Measurement 4	1722256276.29	0.12	09:30:01.707	0.107

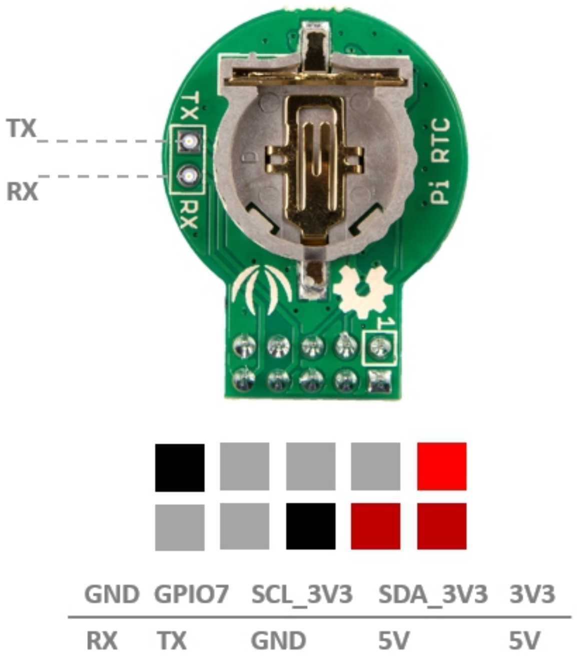


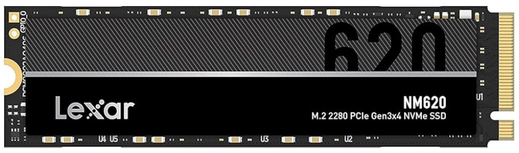
Figure 9.6: RTC DS1307 pins

The updated measurements below include time stamping using the RTC, formatted with full date and time. While still not perfect, the intervals improved slightly to around 0.106–0.11 seconds.

Listing 9.2: IMU Measurment after RTC

```
2024-08-14 09:30:01.382 -24.35 -8.2 6.34 2.07257 0.5115996
    5.3941582 -43.1875 37.5 11.5 -94\\
2024-08-14 09:30:01.491 29.05 4.69 -8.42 0.321795 -7.02601 -4.23351
     -32.0625 50.5625 0.0625 -94\\
2024-08-14 09:30:01.600 -12.26 -11.59 -1.98 -3.311762 9.48259
    5.3210 -38.75 33.0625 24.0625 -94\\
2024-08-14 09:30:01.707 2.61 -4.0600 1.32 -0.85630 -8.847728
    -3.0292 -36.75 47.5 10.375 -94
```

The time difference comparison for both configurations is summarised in Table 9.2.
Although an external RTC can be added to any Pi model, having built-in support as found on the Pi 5 and Orange Pi simplifies integration and improves reliability. These tests demonstrate the importance of hardware-based timekeeping for real-time applications and highlight how integrated RTCs contribute to more consistent data acquisition for simulation and control systems.



(a) SSD for Orange Pi plus

(b) SSD for Raspberry Pi 5

Figure 9.7: SSD for Raspberry Pi and Orange Pi Plus

PCIe support is another important differentiator. EuroSim requires high-speed Non-Volatile Memory Express (NVMe) solid-state drive (SSD) access for efficient operation, which the Pi 4 cannot accommodate directly. NVMe is a high-performance, scalable host controller interface designed specifically for accessing PCI Express-based solid-state drives. In contrast, the Pi 5 can support PCIe storage via an external HAT, while the Orange Pi offers native PCIe connectivity, simplifying the setup. This was a decisive factor in selecting hardware for EuroSim integration.

Other considerations included video output compatibility. The Pi 4 and Pi 5 use Micro HDMI, which is less common and slightly less convenient in educational or multi-user environments. In comparison, the Orange Pi 5 Plus includes full-sized HDMI ports, which are typically easier to source and use, especially in classroom or demonstration settings. The available I/O ports can be seen in Figure 9.8.

While the Raspberry Pi 5 requires an additional HAT to interface with NVMe SSDs, the Orange Pi can connect directly, offering a more integrated and hardware-efficient solution for high-speed data access.

Although the Orange Pi 5 Plus is approximately twice the price of the Raspberry Pi 5, it was selected for final testing due to its superior performance and native PCIe support. A notable limitation of the Orange Pi is its lack of integrated wireless connectivity, whereas the Raspberry Pi 5 includes both Wi-Fi and Bluetooth. Since EuroSim requires an internet connection for initial configuration and potential external communication, this limitation was resolved using a wired Ethernet connection or a compatible USB Wi-Fi module.

9.3 MATLAB Code Generation

Preparing models for EuroSim requires careful configuration using MATLAB's specialised code generation tools. The primary method involves the use of the `rtwbuild('ModelName')` function, which converts Simulink models into C code compatible with the EuroSim simulation environment.

9.3.1 Code Configuration

There are two main strategies for generating code: building the entire system model at once or compiling it block by block. The block-by-block approach provides greater flexibility and control, allowing for fine-tuned adjustments to individual model components. However, as models grow in size and complexity, this method becomes more error-prone due to increased signal routing and interdependencies.

Table 9.3: Set-up Cost for Orange Pi Plus

Hardware	**Cost [€]**
Orange Pi Plus (32GB)	150.00
Lexar NM620 1TB SSD	59.90
HDMI cable	3.98
Ethernet cable	8.99
Keyboard	27.99
Mouse	8.95
Mini screwdriver set	35.95
RTC clock module	5.19
Adapter (USB to flash drive)	23.99
Memory card	9.99
Total	**334,93**

Table 9.4: Set-up Cost with Raspberry Pi 5

Hardware	**Cost [€]**
Raspberry Pi 5 (8GB)	65.00
Pi 5 metal case	15.72
NVMe PCIe Gen3x4 Kingspec SSD	77.26
Pineberry Pi interface	14.95
Micro HDMI cable	10.99
Keyboard	27.99
Mouse	8.95
Mini screwdriver set	35.95
RTC clock module	5.19
Adapter (USB to flash drive)	23.99
(nice to have) Ethernet cable	8.99
Memory card	9.99
Total	**304,97**

Several essential rules must be followed during model development. For instance, each model must include at least one input or output port to ensure proper signal communication. Failure to do so may result in simulation or build errors.
To support this process, a dedicated build script was developed. This script checks and modifies model settings based on twelve validation criteria to ensure full compatibility with EuroSim. By systematically verifying each model and applying required adjustments, the script reduces the need for manual configuration and significantly lowers the risk of user error. This automation improves both the efficiency and reliability of the model preparation workflow. The script can be found in Listing A.4.
The build script implements several critical configuration settings across four main categories:

Table 9.5: Comparison of Raspberry Pi 4, Raspberry Pi 5, and Orange Pi 5 Plus

Feature	Raspberry Pi 4	Raspberry Pi 5	Orange Pi 5 Plus
CPU	Broadcom BCM2711, Quad-core Cortex-A72 @ 1.5 GHz	Broadcom BCM2712, Quad-core Cortex-A76 @ 2.4 GHz	Rockchip RK3588, Octa-core (4x A76 @ 2.4 GHz + 4x A55 @ 1.8 GHz)
GPU	VideoCore VI	VideoCore VII	ARM Mali-G610 MP4
RAM Options	2–8 GB LPDDR4	4–16 GB LPDDR4X-4267	8–32 GB LPDDR4/4X
USB Ports	2x USB 3.0, 2x USB 2.0	2x USB 3.0, 2x USB 2.0	2x USB 3.0 Type-A, 2x USB 2.0, 1x USB 3.0 Type-C
Ethernet	Gigabit Ethernet	Gigabit Ethernet with PoE+ support	2x 2.5 GbE RJ45 ports
Video Output	2x Micro HDMI (up to 4K@60)	2x Micro HDMI (4K@60 HDR)	2x HDMI 2.1 (up to 8K@60), 1x HDMI input, 1x USB-C DisplayPort (8K@30)
Storage	MicroSD	MicroSD (SDR104 mode)	MicroSD, eMMC module, M.2 NVMe SSD
Wireless	Wi-Fi 802.11ac, Bluetooth 5.0	Wi-Fi 802.11ac, Bluetooth 5.0	Optional Wi-Fi 6 / Bluetooth via M.2 E-Key
Power Input	USB-C (5 V / 3 A)	USB-C (5 V / 5 A) with PD support	USB-C (5 V / 4 A)
Real-Time Clock	No	Yes (external battery required)	Yes (external battery required)
Price (approx.)	€60 (4 GB model)	€65 (8 GB model)	€150 (32 GB model)

Solver Configuration

Fixed step discrete solver, 0.01 s step size (100 Hz) A fixed step solver ensures each simulation time step consumes the same wall-clock interval, which is essential for real-time scheduling in EuroSim. A 100 Hz rate matches typical ADCS loop frequencies, allowing sensors, filters and actuators to be serviced within a known deadline.

ode4 (4th-order Runge–Kutta) solver for continuous blocks Any continuous-time dynamics (e.g.simple disturbance models) are integrated with the classical fourth order Runge-Kutta method (ode4). This provides a favourable trade–off between numerical accuracy and execution time, without introducing variable-step behaviour that would violate real time constraints.

Generic Real–Time (GRT) target The GRT system target file is selected so that Simulink generates plain, POSIX-free ANSI C. This maximises portability and allows EuroSim's

Figure 9.8: RaspberryPi 5 and Orange Pi Plus

build system to compile the code into its own simulation libraries.

Code Generation Settings

Code only (no compile or simulate) Export C source and headers without invoking a local compiler or simulator. All subsequent builds and link steps are handled within EuroSim's environment, avoiding conflicts with Simulink's toolchain.

Structured folder layout Generated files are organised into `src/`, `inc/` and `lib/` subdirectories. This separation simplifies EuroSim integration and prevents overcrowded directories.

Reusable functions rather than inlining Blocks are configured to emit separate C functions for reusable logic rather than monolithic inlined code. This reduces code size, improves instruction-cache behaviour and makes the output easier to inspect and debug.

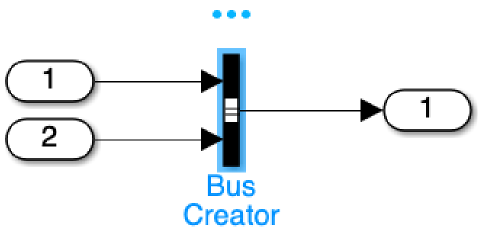
Parameter Management

Runtime-tunable parameters All controller gains, sensor biases and filter covariances are declared as tunable parameters. EuroSim can load these from external files or a user interface at startup, eliminating the need to regenerate code for each parameter tweak.

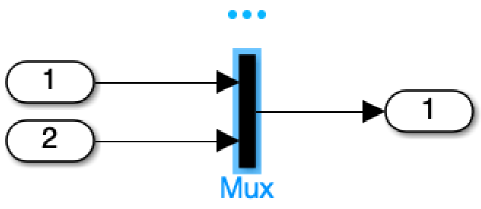
Separate `Outputs` and `Update` functions Estimation and control algorithms are split into distinct output–generation and state–update routines. This separation aligns with EuroSim's execution cycle and aids in isolating timing or numerical issues during debugging.

Performance Optimisation

Disable automatic logging MATLAB-format (`.mat`) logging is turned off to prevent dynamic memory allocations and file-I/O delays that would risk missed deadlines in a real-time loop.



(a) Bus Creator in Simulink



(b) MUX in Simulink

Disable SIMD optimisations Processor-specific vector instructions (e.g.SSE2, AVX) are disabled to ensure the generated C code uses only standard constructs. This maximises compatibility across diverse host machines and embedded boards supported by EuroSim.

Once compiled, each block should get a folder with a name in the format "ModelName_grt_rtw". These folders can then be transferred to the Raspberry Pi / Orange Pi. As each block of our simulator has been exported separately, MATLAB may repeat functions depending on the functionality of the block. When compiling in Eurosim, it may error due to repeated defined function, but these can be deleted easily. The functions are usually inside the ".c" files, which can be opened and the functions commented out or deleted. As long as one definition is present, it will work. On top of this, there are several files which are needed from MATLAB to compile the models. These can be exported from MATLAB, or, within Eurosim, there are several example simulations which have been exported from MATLAB and can be copied into your folder. As long as the version matches, it will work.

9.3.2 MATLAB Limitations

Several limitations of MATLAB Simulink were encountered during the development of the simulation model, particularly in the context of real-time code generation and model portability. One key technical constraint is the use of `bus creators` instead of `MUX` blocks. While MUX blocks are commonly used for signal grouping, they can introduce ordering issues during C code generation. These inconsistencies may cause the model to fail during compilation or execution in external environments. Bus signals, in contrast, maintain named signal structures that are more robust for embedded deployment.

Another major limitation is the inability to use multiple direct copies of the same model block. For instance, duplicating the Sun sensor model six times caused variable conflicts due to shared workspace definitions, leading to failures in variable resolution during simulation. To resolve this, model "instances", a Eurosim feature, must be used instead, ensuring that each block maintains a unique context.

MATLAB does support the use of initialisation scripts to configure mask parameters programmatically. However, this can become difficult to manage, especially when exporting models to environments such as EuroSim. In those cases, constants are often accessed and modified directly within the generated `data.c` files for each model instance, which may not be ideal for maintainability.

Randomisation within the model also introduces challenges. Functions such as `rand` or custom Gaussian noise generators are seeded globally, meaning all calls to the same random function use the same seed unless explicitly overridden. As a result, multiple "random" sources may produce identical outputs unless properly handled, which undermines the realism of simulated noise and variation.

Finally, the complexity of signal routing can become a major source of implementation error. In the initial version of the model, over 500 internal signal connections were present, making debugging and management impractical. This was eventually reduced to 17 blocks and around 100 meaningful connections. Even at this reduced level, managing connections in Simulink remains cumbersome, and mistakes can easily propagate if signals are misrouted or not clearly labelled. Tools such as the "execution order" viewer and debugging interface were instrumental in identifying and resolving these issues.

9.4 Eurosim implementation, How to, dos and donts

Once the model files have been uploaded to the Raspberry/ Orange PI, the generated `.c` files for each model must be further converted to enable compatibility with the EuroSim environment. This additional conversion step is required because Simulink generates standalone C code that references its own `main` function, which must be replaced by the main structure defined by EuroSim. During this process, variable names are standardised, and new EuroSim-specific variables are introduced.
In cases where multiple instances of the same model are used (e.g., multiple Sun sensor blocks), direct reuse of Simulink-generated code can lead to conflicts. To automate the conversion and ensure compatibility, a custom tool developed by Leon Bremer (Airbus) was employed[7]. This command-line utility, called `simulink2c`, converts the Simulink-generated code into EuroSim format. The tool is executed using the following syntax from the terminal:

```
simulink2c -i 1 NameOfFile
```

The number can be replaced with any number and refers to the number of instances required. In this thesis case, it will be six instances of the Sun sensor.
Once the conversion is complete, the code is ready for use in building a simulator. In addition to the converted source files, a Simulink library folder, which contains all the necessary shared scripts and an error-handling script, which prevents the models from erroring due to the individual main files not being called, is added to the model directory for use during simulation execution.
An example of this header is found in Listing A.5.
EuroSim itself is structured around three primary interfaces used to construct and operate the simulator:

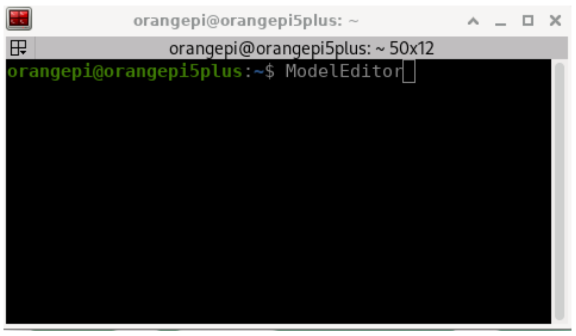
- **ModelEditor** for configuring model structure and input/output mappings
- **ScheduleEditor** for defining the model's execution order and timing
- **SimulationCtrl** for running and monitoring the simulation

Each of these tools is discussed in the following sections.

9.4.1 ModelEditor

The ModelEditor is used to define the build options for EuroSim. Within this interface, the various components of your model are introduced, and the connections between blocks are established.
It is recommended to open a terminal within the model directory that contains all the generated `.rtw` folders. From this terminal, launching the ModelEditor can be done simply by typing `ModelEditor`, which opens the interface shown in Figure 9.14.

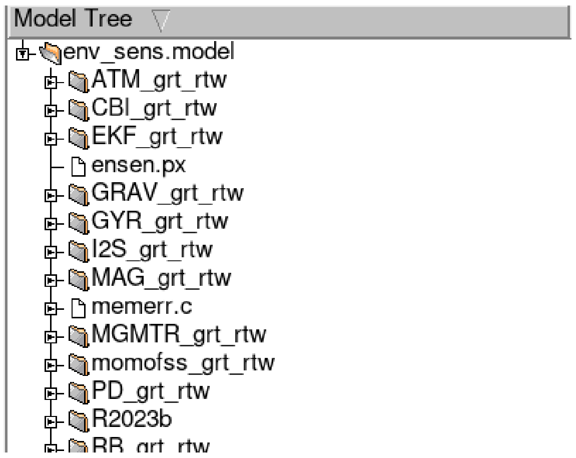
[7]Private communication



```
orangepi@orangepi5plus: ~
orangepi@orangepi5plus: ~ 50x12
orangepi@orangepi5plus:~$ ModelEditor
```

Figure 9.10: Opening ModelEditor in Terminal

The first step is to add all the model blocks. To do this, right-click on `Untitled.model` in the interface. A drop-down menu will appear; hover over `Add`, and then select `Add Directory`. This opens a file browser where each `.rtw` folder should be added individually. Unfortunately, bulk selection is not supported, so for large models this process can be time-consuming.
In addition to the model blocks, it is important to add the MATLAB library folder (named 2023b (Matlab version) in this thesis) and the error-handling script. The library folder should be added using `Add Directory`, while the error script must be added via `Add File`, since directory mode does not allow individual file selection.



```
Model Tree
env_sens.model
  ATM_grt_rtw
  CBI_grt_rtw
  EKF_grt_rtw
  ensen.px
  GRAV_grt_rtw
  GYR_grt_rtw
  I2S_grt_rtw
  MAG_grt_rtw
  memerr.c
  MGMTR_grt_rtw
  momofss_grt_rtw
  PD_grt_rtw
  R2023b
  RB_grt_rtw
```

Figure 9.11: Example folders in ModelEditor

Once all model components have been included, the build paths must be configured. This is done by selecting `Tools` from the top menu bar and choosing `Build Options` from the dropdown list. A window will appear containing multiple input fields. Only the first, larger field needs to be filled in. Here, specify the paths to all required folders necessary for compilation. For example, if the 2023b directory contains subfolders `extra`, `external`, and `Simulink`, then the paths 2023b/`extra`, 2023b/`external`, and 2023b/`Simulink` must be entered. Additionally, the paths to all `.rtw` module directories should be included.
Once the build paths are set, close the options window and press `Build All` from the secondary toolbar. EuroSim will then initialise the required files and prepare the simulator for setup.
Common Errors:

- *Missing or inaccessible files*: This typically indicates that the build paths were not configured correctly. EuroSim relies on the `Build Options` to locate files; if these are incomplete or incorrect, compilation will fail.

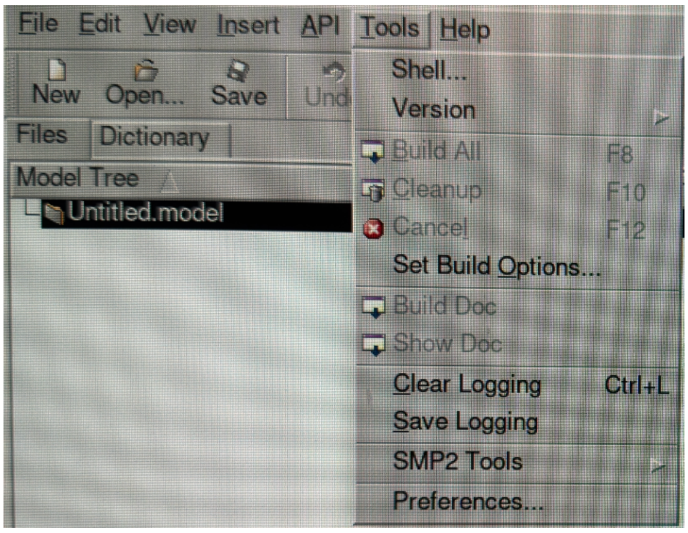


Figure 9.12: Finding BuildOptions

- *Multiple definition errors*: These often occur when functions are defined more than once across different model files. For instance, utility functions such as those in `rtGetNaN.c` may be duplicated in multiple `.rtw` folders. To avoid this, such files can be centralised into a shared folder and included only once.
- *Redundant function definitions*: Occasionally, the same function may appear in more than one generated `.c` file. A typical solution is to comment out one of the duplicates. An example of such a function is provided in Listing 9.3, where commenting out the second definition resolved the compilation issue.

.

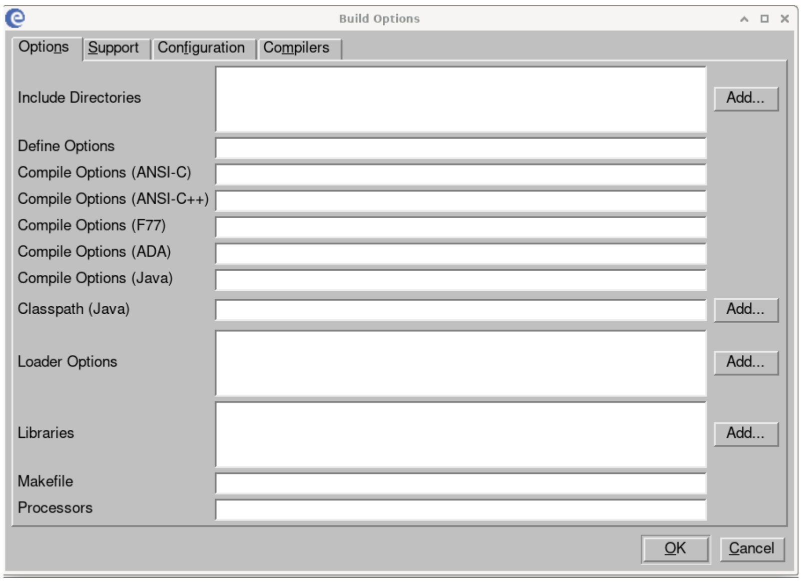
Listing 9.3: rt_roundd_snf(real_T u)

```
real_T rt_roundd_snf(real_T u)
{
  real_T y;
  if (fabs(u) < 4.503599627370496E+15) {
    if (u >= 0.5) {
      y = floor(u + 0.5);
    } else if (u > -0.5) {
      y = u * 0.0;
    } else {
      y = ceil(u - 0.5);
    }
  } else {
    y = u;
  }

  return y;
}
```

ParameterExchange in ModelEditor

Once the compilation is complete, the ModelEditor interface will display `ALL DONE`. It is good practice at this stage to use `Clean All` followed by `Build All` to ensure that all blocks are



Include Directories
Define Options
Compile Options (ANSI-C)
Compile Options (ANSI-C++)
Compile Options (F77)
Compile Options (ADA)
Compile Options (Java)
Classpath (Java)
Loader Options
Libraries
Makefile
Processors

Figure 9.13: BuildOptions

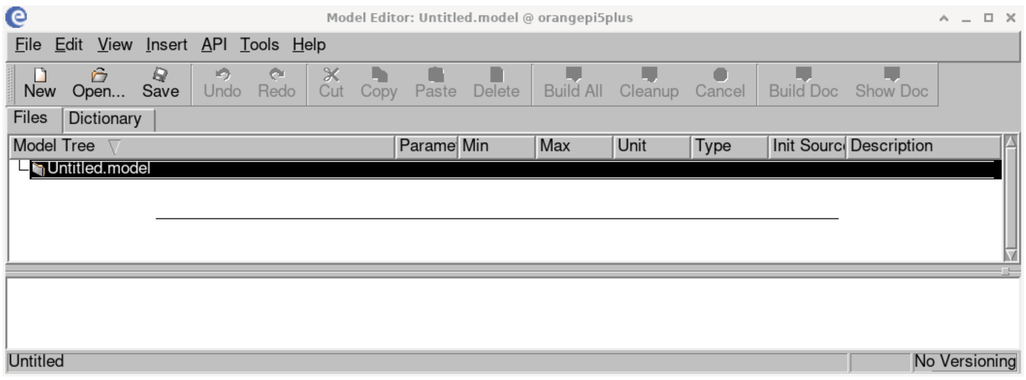

Model Editor: Untitled.model @ orangepi5plus
File Edit View Insert API Tools Help
New Open... Save Undo Redo Cut Copy Paste Delete Build All Cleanup Cancel Build Doc Show Doc
Files Dictionary
Model Tree Parame Min Max Unit Type Init Sourc Description
Untitled.model
Untitled
No Versioning

Figure 9.14: ModelEditor Interface on Eurosim

correctly initialised.
The next step involves establishing connections between the model blocks using a ParameterExchange file. To create this file, right-click on the `.model` name in the ModelEditor interface, select `New`, and then choose `New ParameterExchange`. This will open a new window with three panes, as shown in Figure 9.15.
All signal connections from the Simulink model must now be re-established manually within this interface. A helpful rule of thumb is to group related signals into *exchange groups*. To do this, right-click on the lower wide pane and select `Add Exchange Group` from the title bar menu.
One effective strategy is to group the connections by inputs to blocks. For each model block, focus only on its inputs, this ensures that all essential data pathways are accounted for. It is not necessary to connect every single input and output: some outputs can be used purely for monitoring, while some inputs may be set later via `SimulationCtrl`.
Once all necessary parameter connections have been added, the ParameterExchange file can be saved and closed. The same can then be done for the overall ModelEditor session.

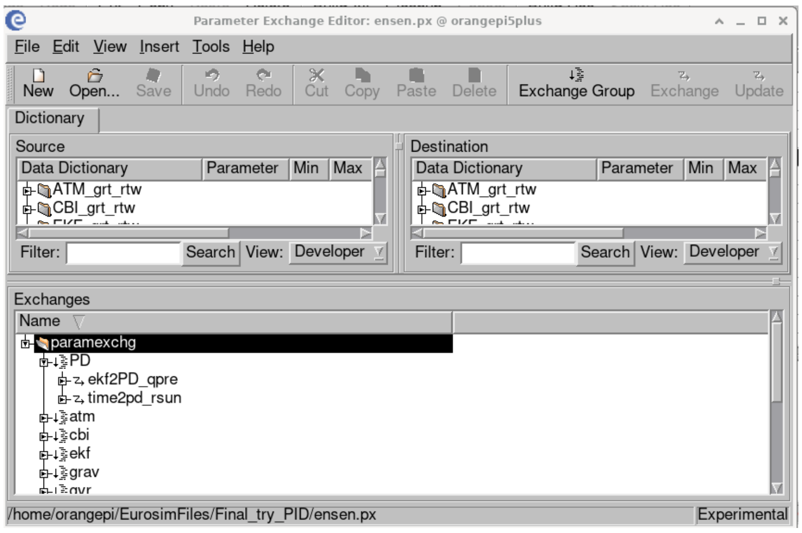

Parameter Exchange Editor: ensen.px @ orangepi5plus
File Edit View Insert Tools Help
New Open... Save Undo Redo Cut Copy Paste Delete Exchange Group Exchange Update
Dictionary
Source
Data Dictionary Parameter Min Max
ATM_grt_rtw
CBI_grt_rtw
Filter: Search View: Developer
Destination
Data Dictionary Parameter Min Max
ATM_grt_rtw
CBI_grt_rtw
Filter: Search View: Developer
Exchanges
Name
paramexchg
PD
ekf2PD_qpre
time2pd_rsun
atm
cbi
ekf
grav
/home/orangepi/EurosimFiles/Final_try_PID/ensen.px
Experimental

Figure 9.15: Parameter Exchange Editor in ModelEditor Interface on Eurosim

9.4.2 ScheduleEditor

After the ParameterExchange configuration is complete, the connection schedule must be defined using the ScheduleEditor. This tool can also be launched from the terminal by entering the command `ScheduleEditor`. The interface is illustrated in Figure 9.16.
Upon opening the ScheduleEditor, you will be prompted to connect a `.model` file. If this dialogue does not appear automatically, it can be accessed by selecting `File > Connect Model File` from the menu. Connecting the model file allows the ScheduleEditor to access all functions and parameter definitions.
The interface consists of several key components. The top grey bar provides buttons for `Flow`, `Task`, and `Timer`, which are used to configure simulation logic in the workspace below. Below this, there are four main state windows: `Initialising`, `Standby`, `Executing`, and `Exiting`.

- **Initialising:** This state defines how the model starts up. Typically, a `Task` block is placed between the `State_Entry` and `Pause` arrows. Double-clicking the Task block opens a new window where you can add all `Initialise` functions from each model block into the right-hand panel.
- **Standby:** This tab is often unused in basic setups. In this thesis, no functions were added here.
- **Executing:** This defines the main simulation loop. You must add a `Timer` block and a `Task` block, connected by an arrow. The default timer rate is 100 Hz, though this can be adjusted. Multiple Task-Timer pairs may be used for components requiring different update rates. Within each Task block, the `Output`, `Update`, and `ParameterExchange` functions must be added. The order in which these functions are listed determines execution priority and signal timing. If the update or execution order is unclear, MATLAB's debugging tools (under the `Execution Order` tab) can provide a visual breakdown of both high-level and low-level call sequences.
- **Exiting:** In this final state, a `Task` block should be added and connected to the `Terminate` arrow. Inside the Task, all `Terminate` functions from each model block must be listed to ensure proper shutdown.

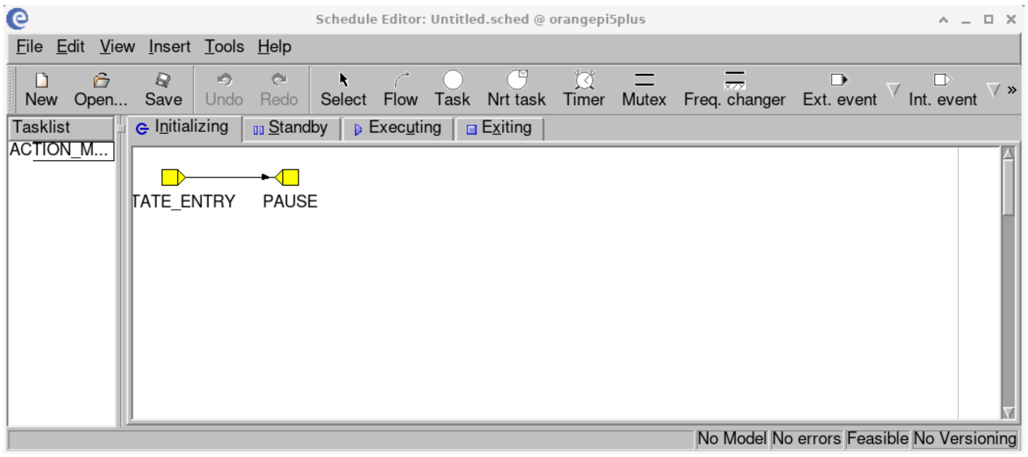
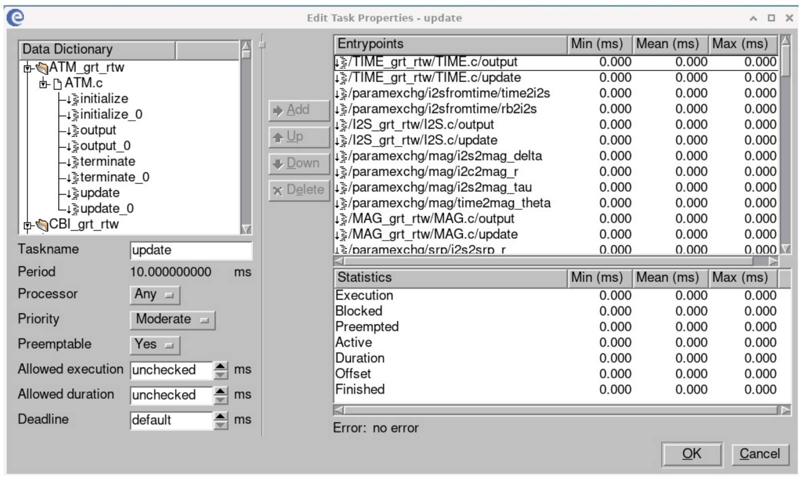


Figure 9.16: ScheduleEditor Interface on Eurosim



Taskname	update	
Period	10.000000000	ms
Processor	Any	
Priority	Moderate	
Preemptable	Yes	
Allowed execution	unchecked	ms
Allowed duration	unchecked	ms
Deadline	default	ms

Entrypoints	Min (ms)	Mean (ms)	Max (ms)
/TIME_grt_rtw/TIME.c/output	0.000	0.000	0.000
/TIME_grt_rtw/TIME.c/update	0.000	0.000	0.000
/paramexchg/i2sfromtime/time2i2s	0.000	0.000	0.000
/paramexchg/i2sfromtime/rb2i2s	0.000	0.000	0.000
/I2S_grt_rtw/I2S.c/output	0.000	0.000	0.000
/I2S_grt_rtw/I2S.c/update	0.000	0.000	0.000
/paramexchg/mag/i2s2mag_delta	0.000	0.000	0.000
/paramexchg/mag/i2c2mag_r	0.000	0.000	0.000
/paramexchg/mag/i2s2mag_tau	0.000	0.000	0.000
/paramexchg/mag/time2mag_theta	0.000	0.000	0.000
/MAG_grt_rtw/MAG.c/output	0.000	0.000	0.000
/MAG_grt_rtw/MAG.c/update	0.000	0.000	0.000
/paramexchg/srp/i2s2srp_r	0.000	0.000	0.000

Statistics	Min (ms)	Mean (ms)	Max (ms)
Execution	0.000	0.000	0.000
Blocked	0.000	0.000	0.000
Preempted	0.000	0.000	0.000
Active	0.000	0.000	0.000
Duration	0.000	0.000	0.000
Offset	0.000	0.000	0.000
Finished	0.000	0.000	0.000

Error: no error

Figure 9.17: Finding BuildOptions

If confusion arises about which functions to call (e.g., `Output` vs. `Update`), you can open the associated `.c` files in the `.rtw` folders to inspect the function definitions. Once all tasks are configured across all four states, the schedule setup is complete.
An example of a Task is shown as follows:

9.4.3 SimulationCtrl

The `SimulationCtrl` interface is used to run the model by handling simulation initialisation, execution, and termination. A typical view of the interface is shown in Figure 9.21.
To begin, both the schedule file and the model file must be loaded. This can be done by clicking `New`, which will prompt the user to select these files. Once loaded, the interface layout updates as shown in Figure 9.22.
The left-hand pane displays a tree menu containing several key features:

- **Scenarios:** This section allows the user to define specific simulation conditions or configurations, useful when testing particular system behaviours or fault cases.
- **Real-Time Monitoring:** During simulation, variables can be observed in real time either as live value displays or plotted graphs. This enables immediate feedback and

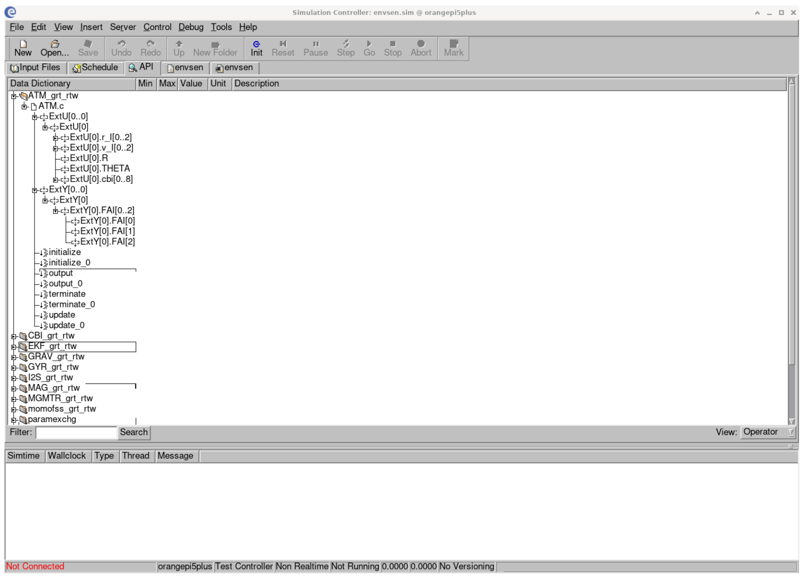


Figure 9.18: API Interface showing variable values

performance assessment.

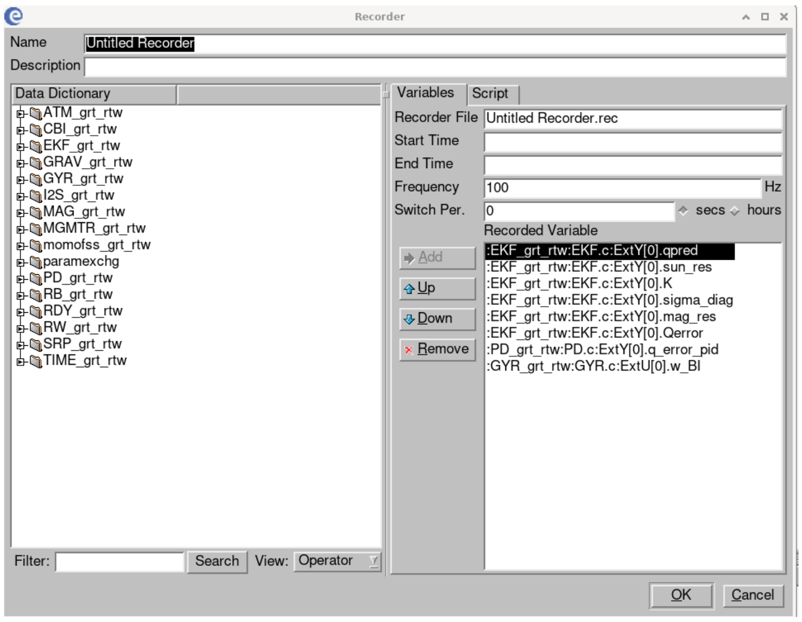
- **MMIs (Man-Machine Interfaces):** This tab allows the user to set up data recorders, which log specified variables over the course of the simulation. Additionally, stimulus files can be added, which contain predefined or real-time data that can be fed back into the model. Stimuli may represent environmental conditions, sensor inputs, or even data from live hardware. An example of this code can be seen in Listing 9.4. This highlights EuroSim's support for hardware-in-the-loop (HIL) testing.

- **Initial Conditions:** As the name suggests, this menu allows users to set initial values for simulation variables during the model's initialisation phase.

Once setup is complete, the model can be initialised by pressing the `Init` button. This activates the `Go` button, which starts the simulation when pressed.
Some common errors here involve incorrect parameter exchanges, incorrect scheduling of those parameter exchanges, and not building your ModelEditor .model file before running your simulation on SimulationCtrl. Even if a model has been run before successfully and you turn off your Pi, you should always build the model on ModelEditor first.

Listing 9.4: Example of Stimulus File

```
# EuroSim recording file
Version: Mk7-rev3-pl10
Date recorded: Mon-May-05-08:55:33-CST-2025
Mission: ../../ekf.mdl
# enough data for stimulating 100812 timesteps with recordsize 344
Dict: ../../please.Linux/please.dict
SimTime: /simulation_time
TimeFormat: relative
Number of variables: 9
/simulation_time: struct timespec
/EKF_grt_rtw/EKF.c/ExtY[0].qpred: double[4], C array
/EKF_grt_rtw/EKF.c/ExtY[0].sun_res: double[3], C array
/EKF_grt_rtw/EKF.c/ExtY[0].mag_res: double[3], C array
```



Name: Untitled Recorder
Description:

Data Dictionary:
- ATM_grt_rtw
- CBI_grt_rtw
- EKF_grt_rtw
- GRAV_grt_rtw
- GYR_grt_rtw
- I2S_grt_rtw
- MAG_grt_rtw
- MGMTR_grt_rtw
- momofss_grt_rtw
- paramexchg
- PD_grt_rtw
- RB_grt_rtw
- RDY_grt_rtw
- RW_grt_rtw
- SRP_grt_rtw
- TIME_grt_rtw

Filter: View: Operator

Recorder File: Untitled Recorder.rec
Start Time:
End Time:
Frequency: 100 Hz
Switch Per.: 0 secs hours

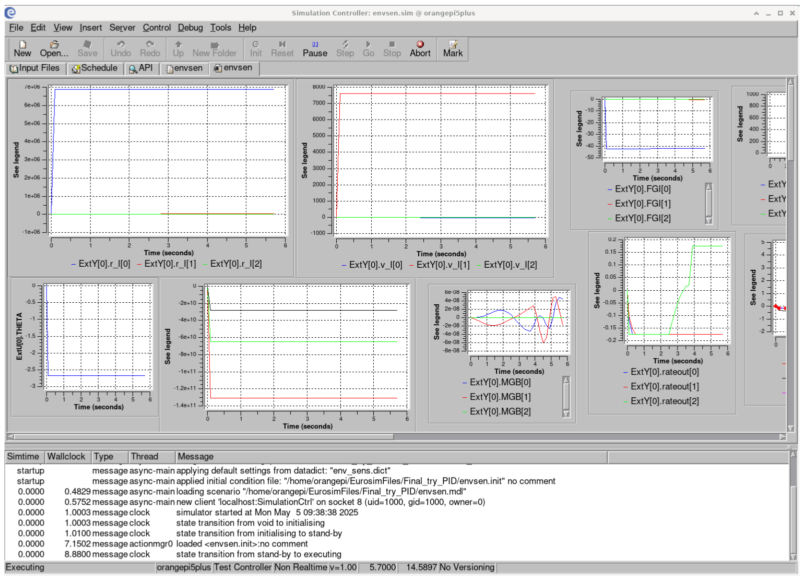
Recorded Variable
```
:EKF_grt_rtw:EKF.c:ExtY[0].qpred
:EKF_grt_rtw:EKF.c:ExtY[0].sun_res
:EKF_grt_rtw:EKF.c:ExtY[0].K
:EKF_grt_rtw:EKF.c:ExtY[0].sigma_diag
:EKF_grt_rtw:EKF.c:ExtY[0].mag_res
:EKF_grt_rtw:EKF.c:ExtY[0].Qerror
:PD_grt_rtw:PD.c:ExtY[0].q_error_pid
:GYR_grt_rtw:GYR.c:ExtU[0].w_BI
```

Figure 9.19: Recorder tab which records the selected outputs

```
/EKF_grt_rtw/EKF.c/ExtY[0].K: double[18], C array
/EKF_grt_rtw/EKF.c/ExtY[0].sigma_diag: double[3], C array
/EKF_grt_rtw/EKF.c/ExtY[0].Qerror: double[4], C array
/RDY_grt_rtw/RDY.c/ExtY[0].w_BI: double[3], C array
/RDY_grt_rtw/RDY.c/ExtY[0].q: double[4], C array
```

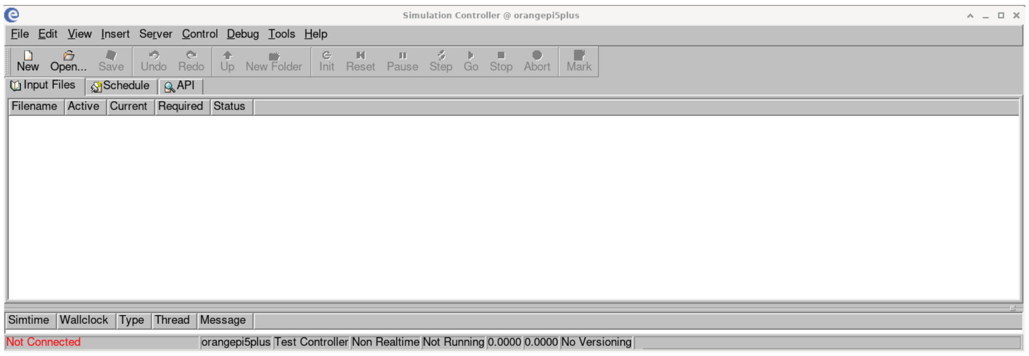
9.5 Acceptance and Implementation Testing of Simulink Models onto Eurosim

To make sure the method of implementation is correct, multiple tests were performed. These test was done by comparing the output of a Simulink model to the model in EuroSim. The first test was done using the model shown in Figure 9.23. The example model only contains two blocks, LQR and RB. The LQR block is a simple LQR controller, and the RB block is a rigid body dynamics model. There is one input for the RB block and two outputs, which feed into the LQR block, making the parameter exchanges simple. The results of this test can be seen in Figure 9.24. The thick lines show a common error of frequency mismatch. The model was exported with a 40-second time step while the blocks were run at 100Hz (0.01s) in Eurosim, leading to an error in results. The second plot shows the actual errors when the model was implemented correctly. This shows large spikes in errors, which occur when the lines cross.
The second test was done using the model shown in Figure 9.25. This model is more complex as it includes parallel blocks. It also has a loop which doesn't include all blocks which makes parameter exchange scheduling interesting. This model was exported correctly, and the outputs from both models match nearly perfectly.


Simulation Controller: envsen.sim @ orangepi5plus
File Edit View Insert Server Control Debug Tools Help
New Open... Save Undo Redo Up New Folder Init Reset Pause Step Go Stop Abort Mark
Input Files Schedule API envsen envsen
ExtY[0].r_I[0] ExtY[0].r_I[1] ExtY[0].r_I[2]
ExtY[0].v_I[0] ExtY[0].v_I[1] ExtY[0].v_I[2]
ExtY[0].FGI[0] ExtY[0].FGI[1] ExtY[0].FGI[2]
ExtY[0].MGB[0] ExtY[0].MGB[1] ExtY[0].MGB[2]
ExtY[0].rateout[0] ExtY[0].rateout[1] ExtY[0].rateout[2]
ExtU[0].THETA
Time (seconds)
See legend

```
Simtime  Wallclock Type    Thread       Message
 startup           message async-main   applying default settings from datadict: "env_sens.dict"
 startup           message async-main   applied initial condition file: "/home/orangepi/EurosimFiles/Final_try_PID/envsen.init" no comment
  0.0000    0.4829 message async-main   loading scenario "/home/orangepi/EurosimFiles/Final_try_PID/envsen.mdl"
  0.0000    0.5752 message async-main   new client 'localhost:SimulationCtrl' on socket 8 (uid=1000, gid=1000, owner=0)
  0.0000    1.0003 message clock        simulator started at Mon May  5 09:38:38 2025
  0.0000    1.0003 message clock        state transition from void to initialising
  0.0000    1.0100 message clock        state transition from initialising to stand-by
  0.0000    7.1502 message actionmgr0   loaded <envsen.init>:no comment
  0.0000    8.8800 message clock        state transition from stand-by to executing
Executing    orangepi5plus Test Controller Non Realtime v=1.00  5.7000  14.5897 No Versioning
```

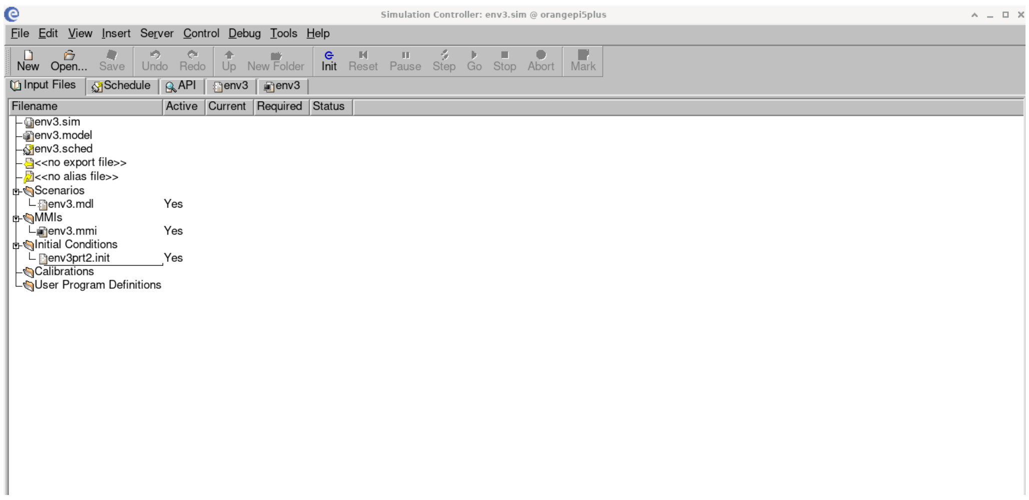
Figure 9.20: Graphs in SimulationCtrl


Simulation Controller @ orangepi5plus
File Edit View Insert Server Control Debug Tools Help
New Open... Save Undo Redo Up New Folder Init Reset Pause Step Go Stop Abort Mark
Input Files Schedule API

```
Filename  Active  Current  Required  Status

Simtime  Wallclock  Type  Thread  Message
Not Connected    orangepi5plus Test Controller Non Realtime Not Running 0.0000 0.0000 No Versioning
```

Figure 9.21: SimulationCtrl Interface on Eurosim


Simulation Controller: env3.sim @ orangepi5plus
File Edit View Insert Server Control Debug Tools Help
New Open... Save Undo Redo Up New Folder Init Reset Pause Step Go Stop Abort Mark
Input Files Schedule API env3 env3

```
Filename                     Active  Current  Required  Status
env3.sim
env3.model
env3.sched
<<no export file>>
<<no alias file>>
Scenarios
  env3.mdl                   Yes
MMIs
  env3.mmi                   Yes
Initial Conditions
  env3prt2.init              Yes
Calibrations
User Program Definitions
```

Figure 9.22: SimulationCtrl Interface on Eurosim

9.6 Final Model Implementation

The final model with all environment, sensors, actuators, propagators and more was condensed to 17 blocks and over 100 parameter exchanges. All of these must be done manually. The

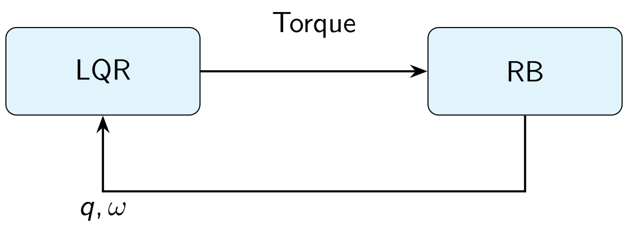
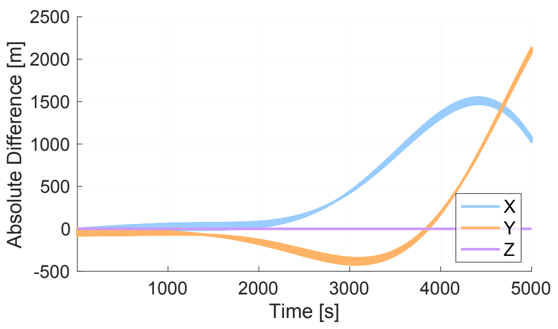
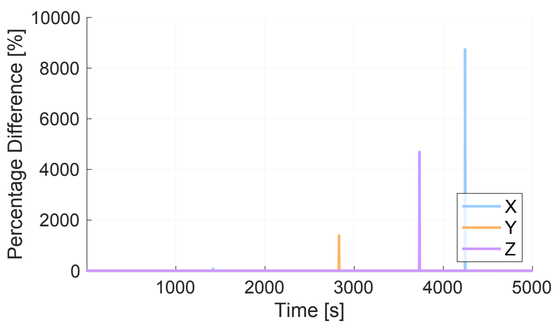


Figure 9.23: Flowchart illustrating the interaction for first example between LQR and RB. LQR provides torque to RB, and RB outputs q and ω back to LQR.



(a) Absolute difference in position between Simulink and Eurosim



(b) Percentage difference in position between Simulink and Eurosim

Figure 9.24: Validation of Eurosim implementation against Simulink reference model

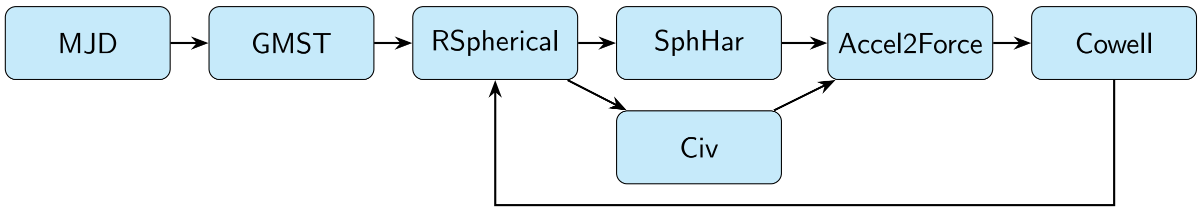
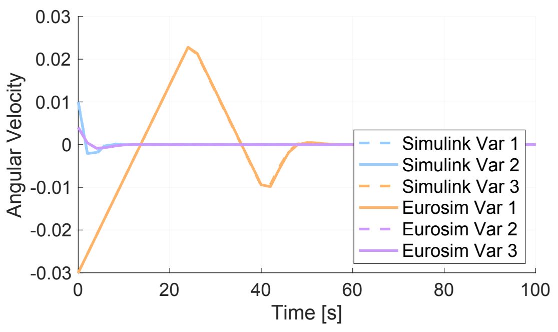
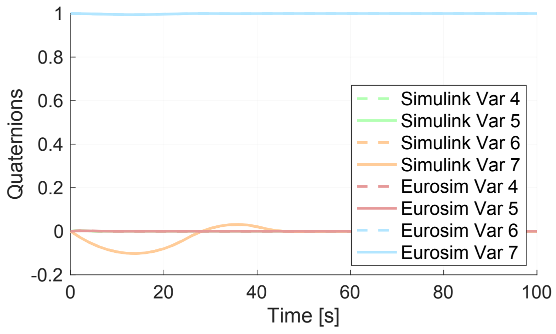


Figure 9.25: Detailed system architecture showing all main components and data flow.



(a) Angular velocity comparison between Simulink and Eurosim



(b) Quaternion orientation comparison between Simulink and Eurosim

Figure 9.26: Comparison of attitude dynamics between Simulink and Eurosim implementations

flow diagram of the simulation can be seen below. Of course, this was not done at once but over different stages. First, the environment models are added and tested. Compared with the

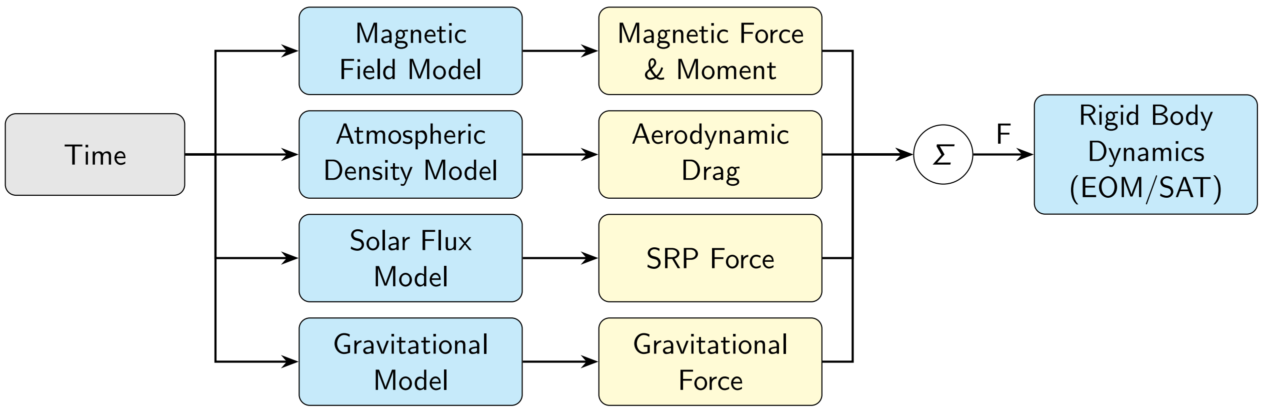
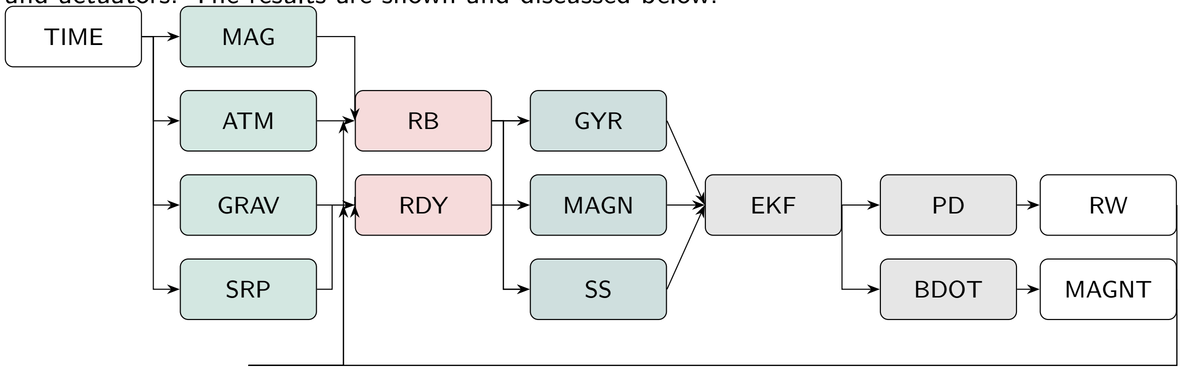


Figure 9.27: Compact layout of the environment-to-dynamics model with left-to-right flow.

MATLAB Simulink data. Then the sensors and actuators, the EKF and finally the controller and actuators. The results are shown and discussed below.

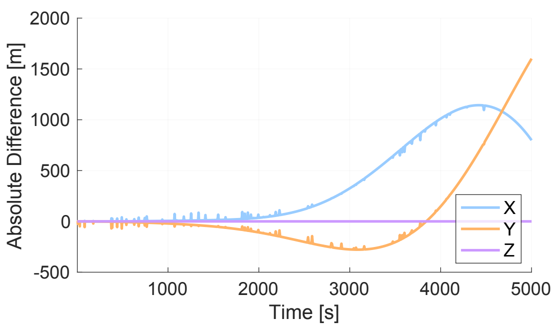


9.6.1 Environment

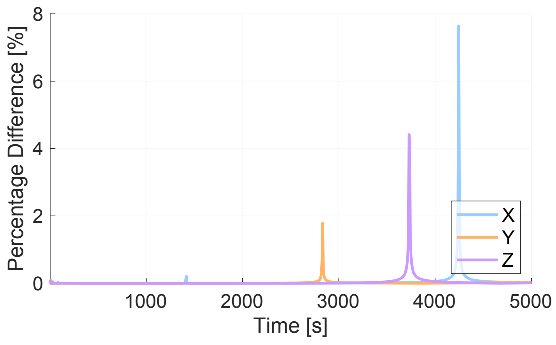
The environment model can be found in Figure 9.27. Each block is generated separately based on the methodology discussed previously. The results of the model can be found in Figure 9.28. The model is simple with no controller and with ideal conditions. The absolute difference shows a growing error with time. There is a presence of small jitter which is coming from the variability of Eurosim. Figure b shows large jumps similar to the acceptance tests when the values cross each other or the zero bar. This can be explained as if the value is supposed to be zero, and a value near zero is found; this can be seen as a large offset. Therefore was concluded not to be a problem. One difference is the exponential curve, which can be seen but could not be seen on the acceptance test. This could be explained by step size differences or Eurosim model jitter.

9.6.2 Sensor Addition

Figure 9.29 shows the environment model with the addition of rotational dynamics as well as sensor models. These sensor models include one gyroscope, six sun sensors and a magnetometer. The results shown in Figure 9.30, show the expected jumps when the zero line is crossed. Other results, however, overlap perfectly, showing good model configuration.



(a) Absolute difference for radius Plot with environment models



(b) Percentage difference for velocity Plot with environment models

Figure 9.28: Results of environment models

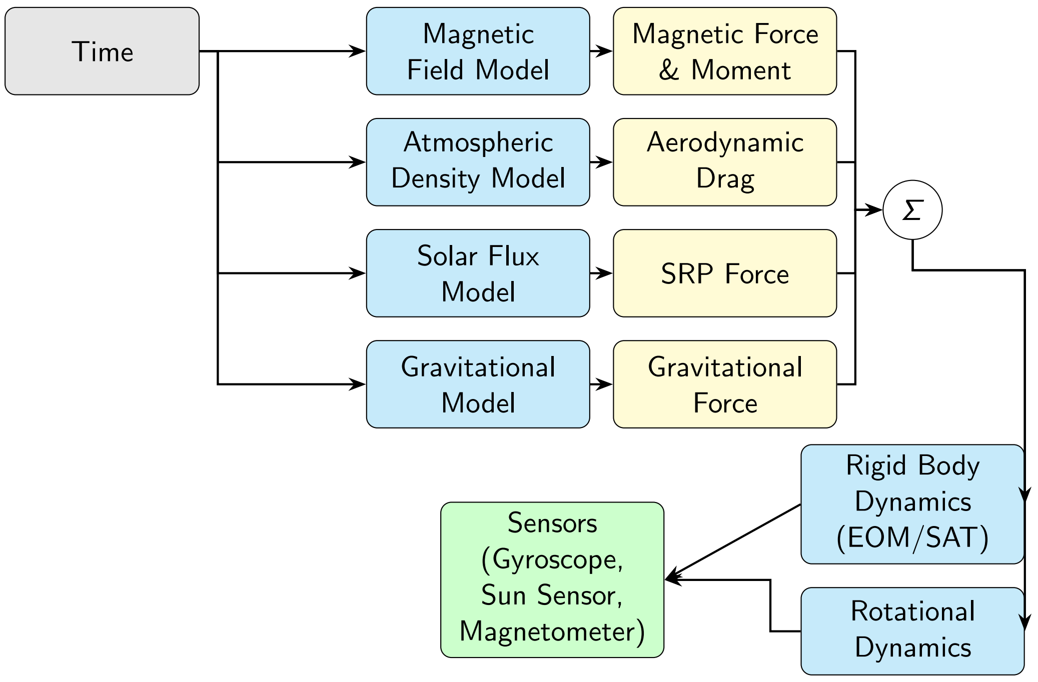
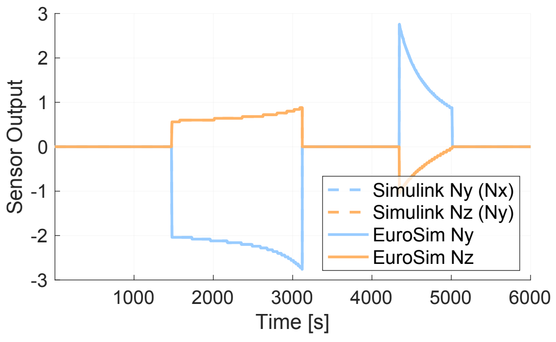


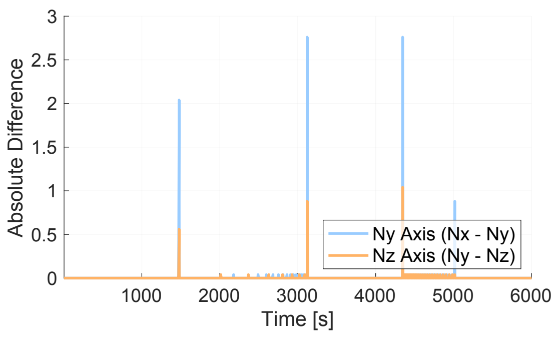
Figure 9.29: Environment-to-dynamics model with clockwise flow: Time input, forces, summation, dynamics, and sensors.

9.6.3 EKF in Eurosim

Given the previously mentioned method, the Extended Kalman filter was then ported to Eurosim. The results can be seen in Figure 9.31. Here, it was found that the sun sensor was not compatible when in a non-ideal condition due to the noise module. Therefore, a function was used in its place. These new results, shown as a covariance plot in Figure 9.31, show a small deviation in the results. Although by the end of the simulation, the results are closer together, and the error decreases. This could be due to the addition of a different sun sensor, although the MATLAB results were also collected with the new sun sensor function. Another source of the error could be that the errors are very small and in the previous tests, the errors were clouded by the spikes, which could have misled the strong alignment. Lastly, the order of execution could be a cause of this error. The dips and jumps in the errors align, which is



(a) Absolute difference of sun sensor measurements



(b) Percentage difference for sun sensor measurements

Figure 9.30: Results of sensor models

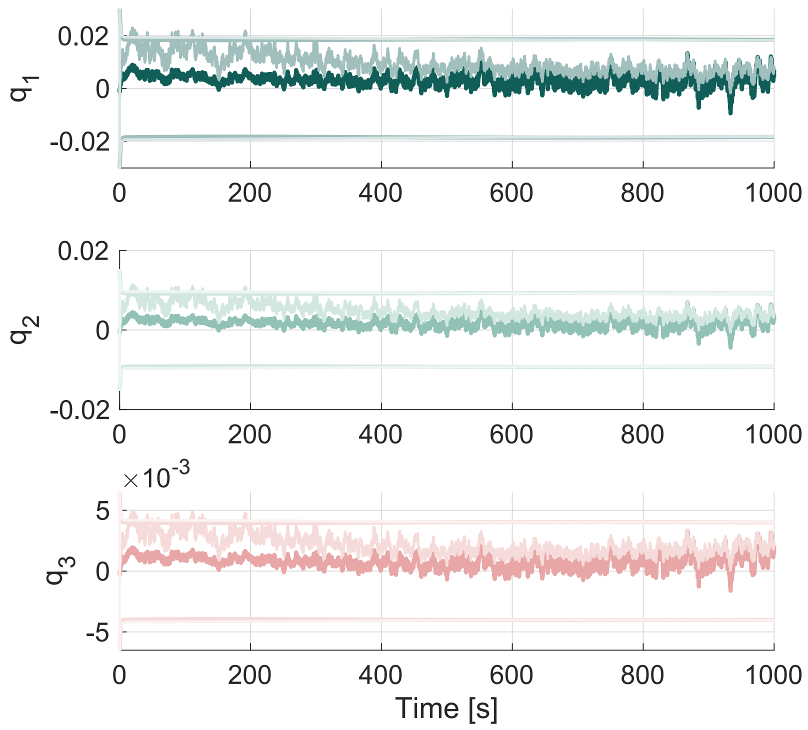


Figure 9.31: Error covariance plot. The darker shades are Simulink, and the lighter shade is Eurosim.

encouraging.

9.6.4 PD in Eurosim

Here, the PD controller was added. As shown in Figure 9.32, an even larger difference is found; however, the settling time is close to identical. This could be an accumulated error from the EKF, which then applies to the PD controller. This deviation indicates there may be a block which does not respond as intended.
Overall, the performance of Eurosim is encouraging but also challenging. Eurosim is a very promising tool as it can offer real-time environments and the possibility to include hardware; however, more complex models can be hard to implement without a full understanding of the

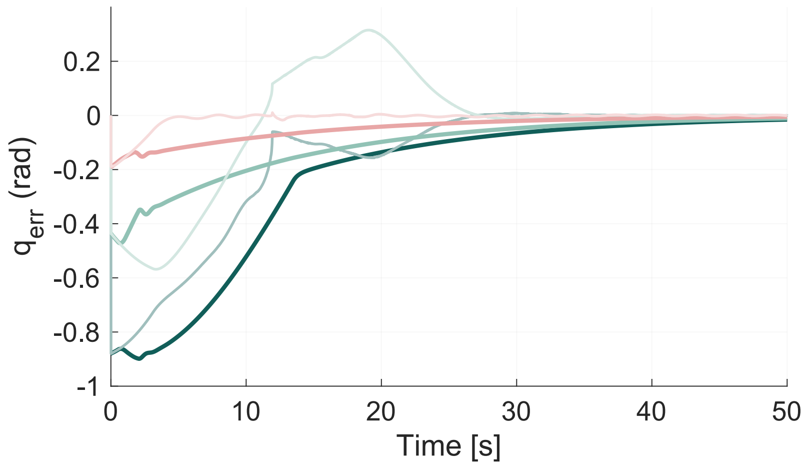


Figure 9.32: PD Error covariance plot for first 50 seconds

backend operations of Eurosim. It is also good that it can run on low-cost hardware, which is ideal for educational projects with limited budgets. The interface is also simple to understand, which can help with usability. For simple unit tests of ADCS, this can be a great tool for student engineers.

9.7 Requirements

This section shows the compliance with the requirements given in Chapter 3.

Table 9.6: Compliance summary for simulator requirements.

ID	Compliance Summary
SIM01	Achieved. Real-time simulation of CubeSat attitude dynamics was successfully demonstrated.
SIM02	Partially achieved. Hardware was not tested directly, but its capabilities were investigated and documented for future development.
SIM03	Achieved. Real-time interaction with ADCS software was successfully demonstrated.
SIM04	Achieved. The setup used widely available COTS components, including Raspberry Pi and Orange Pi boards.
SIM05	Achieved. All tools used were educationally licensed, although not fully open-source.
SIM06	Partially achieved. The interface was accessible, but the documentation was complex and assumed prior software knowledge. Further improvements are suggested.
SIM07	Achieved. The complete test bench was assembled for under €500, well below the €2000 limit.

CHAPTER 10

Conclusion and Recommendations

This study set out to answer the following research question:

> *To what extent can low-cost hardware and simulation tools support real-time testing of CubeSat ADCS algorithms in an educational context?*

By combining MATLAB/Simulink models with the GRADS and GGNCSIM libraries and the EuroSim real-time framework, the project demonstrates that a credible and functional ADCS test bench can be assembled for under €500. The platform currently supports SILT) experiments and shows significant promise for future HIL testing.
A complete functional simulator was built in MATLAB, incorporating environmental perturbations, realistic sensor and actuator models, and both B-dot and PD control laws. EKFs were evaluated under three configurations: magnetometer-only, single sun sensor, and dual-sensor fusion. The dual-sensor fusion case produced the lowest quaternion error and remained numerically stable for at least 1 000 s. Real-time code was auto-generated and ported to EuroSim. Despite some build system limitations, particularly function replication and `bus` versus `mux` conflicts, a twelve-step build workflow was developed that ensures consistent compilation on both Raspberry Pi 5 and Orange Pi 5 Plus hardware.
Closed-loop tests verified successful detumbling and sun acquisition at a rate of 100Hz with no deadline overruns. B-dot control reliably reduced angular velocity, while the open-loop configuration diverged, validating the controller. The PD controller aligned the $+X$ axis to the Sun in under 50 seconds, well within the SA01 requirement of 90 minutes, and achieved a steady-state pointing error below 0.5°.

Addressing the Sub-Research Questions

- **What components are required to develop a functional ADCS simulation and test environment?**
  The system requires models for environmental effects, sensors (magnetometer, sun sensors), actuators (magnetorquers), filters (EKF), and controllers (B-dot, PD). Simulink, along with add-ons like the Aerospace Toolbox and Simulink 3D Animation, provided the modeling framework. Real-time execution relied on EuroSim and supported libraries such as GRADS and GGNCSim.

- **What are the benefits and limitations of using simulated environments for testing ADCS algorithms without access to flight hardware?**
  Simulated environments allow early-stage validation and debugging in a controlled setting, reducing risk before deployment. They also provide repeatable scenarios for educational use. However, they can't capture all real-world dynamics and are limited by the fidelity of the models and numerical integration challenges. EuroSim in particular becomes more cumbersome with increasing model complexity.

- **Can the proposed setup incorporate hardware-in-the-loop (HIL) testing?**
  Yes. Stimulus-file-based experiments confirmed that EuroSim can support HIL operation. While full HIL integration was beyond this thesis's scope, the system architecture is capable of supporting it. The next logical step is to interface with physical sensors and actuators.

- **How can a real-time test bench be designed to meet the needs of student-led and institution-based CubeSat projects?**
  By leveraging widely-used tools (e.g., MATLAB), low-cost embedded hardware, and real-time frameworks like EuroSim, a test bench can be constructed affordably and incrementally. Documenting each stage of integration helps manage complexity and lowers the barrier for other student teams.

Limitations

The major limitation encountered was the increasing complexity of porting advanced Simulink models to EuroSim. As models grow, issues such as redundant function generation and manual block separation become more frequent, slowing the development process. Automating this process, or developing a higher-level interface to manage modular integration, would significantly streamline workflow.
Hardware itself was not a bottleneck: modern single-board computers like the Raspberry Pi 5 and Orange Pi 5 Plus handled real-time execution without deadline misses. Their performance is expected to improve over time, further increasing the system's robustness.
Lastly, although the current system supports SITL, the absence of full HIL integration leaves a gap in practical testing capabilities. Connecting actual sensors and actuators remains an essential next step.

10.1 Recommendations

To extend this work and build a more complete test ecosystem, the following recommendations are proposed:

1. **Implement full hardware-in-the-loop (HIL) integration.** Start with low-risk components (e.g., sun sensors or magnetometers) that do not involve mechanical motion. Gradually progress to actuators like magnetorquers, and finally consider motion platforms for 3D testing.

2. **Adopt more flight-representative ADCS algorithms.** Explore control approaches such as quaternion-based PID, nonlinear observers, or adaptive filtering to evaluate the scalability and precision of the current test bench for real-world missions.

3. **Automate model modularisation and code generation.** Investigate scripting tools or APIs to partition Simulink models into EuroSim-compatible modules automatically. This would significantly reduce manual effort and allow for rapid iteration and experimentation.

4. **Streamline EuroSim integration.** Explore command-line control or API-level access to automate build, test, and deployment procedures. This would enhance reproducibility and open the door to continuous integration pipelines for model testing.

5. **Expand educational outreach.** Package this test bench as a teaching tool, including tutorials, troubleshooting guides, and example missions. This could improve CubeSat training programs and reduce ADCS-related mission failures in academic settings.

6. **Benchmark alternative real-time frameworks.** Compare EuroSim with open-source platforms like `sat-rs` or Xcos for Scilab to assess cost-benefit tradeoffs and compatibility with educational or low-cost space initiatives.

This thesis contributes a low-cost, real-time CubeSat ADCS test bench that bridges the gap between purely numerical studies and expensive professional facilities. Despite some limitations, the architecture is scalable, the process is reproducible, and the outcomes are educationally meaningful. With modest extensions, the system can support full HIL testing and serve as a hands-on platform for training future spacecraft engineers.

Appendix A

A.1 Simplified Fine Sun Sensor Model (MATLAB)

Listing A.1: Simplified Fine-Sun-Sensor model

```
function [lit, Ny, Nz] = fss_model( ...
        Sb_B, eclipse,                ... % inputs
        phi_deg, theta_deg, psi_deg,  ... % mount (deg)
        FOV_X_deg, FOV_Y_deg,         ...
        BIAS_X_rad, BIAS_Y_rad,       ...
        NOISE_X_sigma, NOISE_Y_sigma, ...
        Q_X_rad, Q_Y_rad)                 % quantisation (rad)
%#codegen

deg2rad = pi/180;

%% --- convert once
phi   = deg2rad * phi_deg;
theta = deg2rad * theta_deg;
psi   = deg2rad * psi_deg;

hf1 = deg2rad * FOV_X_deg;
hf2 = deg2rad * FOV_Y_deg;

%% --- body -> sensor frame rotation
cph = cos(phi);   sph = sin(phi);
cth = cos(theta);sth = sin(theta);
cps = cos(psi);   sps = sin(psi);

C = [ cth*cps,                cth*sps,                -sth ; ...
      sph*sth*cps-cph*sps,    sph*sth*sps+cph*cps,    sph*cth ; ...
      cph*sth*cps+sph*sps,    cph*sth*sps-sph*cps,    cph*cth ];

Sf  = C * Sb_B(:);                                % [ux; uy; uz]
ux = Sf(1);  uy = Sf(2);  uz = Sf(3);

%% FOV and front side test

front   = (uz <= 0);
insideY =  abs(uy/uz) <= tan(hf1);
insideZ =  abs(-ux/uz) <= tan(hf2);
noEcl   = (eclipse == 0);

lit = double(front && insideY && insideZ && noEcl);

if ~lit        % not lit: return zeros fast
    Ny = 0; Nz = 0;
    return
end

%% -deal tangent angles
```

```
Ny_ideal =  uy / uz;      % tan(alpha_y)
Nz_ideal = -ux / uz;      % tan(alpha_z)

%% --- bias + Gaussian noise
   ----------------------------------------
Ny_meas = Ny_ideal + BIAS_Y_rad       + NOISE_Y_sigma * randn();
Nz_meas = Nz_ideal + BIAS_X_rad       + NOISE_X_sigma * randn();

%% --- quantise
   ------------------------------------------------------
Ny = round(Ny_meas / Q_Y_rad) * Q_Y_rad;
Nz = round(Nz_meas / Q_X_rad) * Q_X_rad;
end
```

A.2 Initialise Model Script (MATLAB)

Listing A.2: Simulation set-up

```
%% ---------------- Simulation timing
   -----------------------------------
tfinal        = 1000;       % [s]
T_sample      = 0.01;       % main Simulink sample time

% sensor sample times [s]
Sensors.FineSunSensor.SAMPLE            = 0.05;
Sensors.Magnetometer.SAMPLERATE         = 0.02;
Sensors.ThreeAxisAccelerometer.SAMPLERATE = 0.04;
Sensors.ThreeAxisGyroscope.SAMPLERATE   = 0.01;

%% ---------------- Satellite characteristics
   ---------------------------
Sat.mass                    = 5;            % [kg]
Sat.ReflectionCoefficient= 0.8;
Sat.SurfaceArea             = 0.02;         % [m^2]
Sat.DragCoefficient         = 2;

%% ---------------- Environment
   -----------------------------------------
% Gravity model (ITG GRACE 2010 s, n=m=10)
load itg_grace2010s
Environment.Gravity.SH.RE = RE;
Environment.Gravity.SH.MU = MU;
Environment.Gravity.SH.C  = C;
Environment.Gravity.SH.S  = S;
Environment.Gravity.Degree = 10;
Environment.Gravity.Order  = 10;

% Magnetic dipole (IGRF like simplification)
Environment.MagneticDipole.g10 = -2.9557E-05;
Environment.MagneticDipole.g11 = -1.6718E-06;
Environment.MagneticDipole.h11 =  5.0800E-06;
Environment.MagneticDipole.B0  = 3.12e-5;          % [T]
Environment.EarthAngularRate    = 7.2921151467e-5;% [rad/s]
Environment.Mag.M_RM           = [ 1.5; -1.0; 0.5] * 1e-3; % [A m2]
```

```
%% ---------------- Initial orbital state
    -------------------------------
h       = 500e3;                                      % 500 km
    circular
x_sat   = 0.10;   y_sat = 0.10;   z_sat = 0.30;        % bus dims [m]

Propagators.System.mass = Sat.mass;
Propagators.System.mu   = MU;

r0 = [RE + h; 0; 0];
v0 = [0; sqrt(MU/r0(1)); 0];
Propagators.State.Translational.r0 = r0;
Propagators.State.Translational.v0 = v0;

Ix = (1/12)*Sat.mass*(y_sat^2 + z_sat^2);
Iy = (1/12)*Sat.mass*(x_sat^2 + z_sat^2);
Iz = (1/12)*Sat.mass*(x_sat^2 + y_sat^2);
Propagators.System.I    = diag([Ix, Iy, Iz]);
Propagators.System.Iinv = diag(1./[Ix, Iy, Iz]);

Propagators.State.w0 = [0; 0; 0];
Propagators.State.q0 = [0; 0; 0; 1];

%% ---------------- Epoch (UTC)
    ----------------------------------------
Utilities.TimeUtilities = struct('Year',2021,'Month',10,'Day'
    ,21,...
                                 'Hour',23,'Minute',45,'Sec',10.8);

%% ---------------- Orbital elements (for OrbCar)
    ----------------------
Utilities.CoordinateTransformations = struct( ...
    'e', 0, ...
    'a', RE + h, ...
    'i', 0, ...
    'w', 0, ...
    'O', 0, ...
    'tp', 45*60, ...
    'mu', MU );

[Vi0, Xi0] = OrbCar( Utilities.CoordinateTransformations.mu, ...
                     Utilities.CoordinateTransformations.e,  ...
                     Utilities.CoordinateTransformations.a,  ...
                     Utilities.CoordinateTransformations.i,  ...
                     Utilities.CoordinateTransformations.w,  ...
                     Utilities.CoordinateTransformations.O,  ...
                     Utilities.CoordinateTransformations.tp );

%% ------------------ Magnetometer
    -------------------------------------
Sensors.Magnetometer = struct( ...
    'MISALIGNMENT_1',  0.5e-3*pi/180*[1 1], ...
    'MISALIGNMENT_2',  0.5e-3*pi/180*[1 1], ...
    'MISALIGNMENT_3',  0.5e-3*pi/180*[1 1], ...
    'DRIFT',           1e-9*[1 1 1], ...
    'SCALE_FACTOR',    8e-9*[1 1 1], ...
    'NOISE',           1e-12*[1 1 1], ...
    'SEED',            0, ...
    'QUANTIZATION',    0.0488e-6*[1 1 1], ...
    'RATE_MIN',       -1e-4, ...
    'RATE_MAX',        1e-4 );

%% ------------------ Fine sun sensor (6 faces)
   -------------------------
FSS = Sensors.FineSunSensor;
FSS = struct( ...
  'SAMPLE',        0.05, ...
  'FOV_X',         30, 'FOV_Y', 30, ...
  'BIAS_X',        deg2rad(0.01)*0.01,...
  'BIAS_Y',        deg2rad(0.01)*0.01,...
  'NOISE_X',       deg2rad(0.0333)*100,...
  'NOISE_Y',       deg2rad(0.0333)*100,...
  'QUANTIZATION_X',deg2rad(0.0557),...
  'QUANTIZATION_Y',deg2rad(0.0557),...
  'PHI',  90,'THETA',   0,'PSI',0, ...   % SS1  -Y
  'PHI1',-90,'THETA1',  0,'PSI1',0, ...   % SS2  +Y
  'PHI2',  0,'THETA2',-90,'PSI2',0, ...   % SS3  -X
  'PHI3',  0,'THETA3', 90,'PSI3',0, ...   % SS4  +X
  'PHI4',180,'THETA4',  0,'PSI4',0, ...   % SS5  -Z
  'PHI5',  0,'THETA5',  0,'PSI5',0  );    % SS6  +Z
Sensors.FineSunSensor = FSS;

%% ------------------ Gyroscope
   ----------------------------------------
Sensors.ThreeAxisGyroscope = struct( ...
  'DRIFT',           1e-7*[1 1 1], ...
  'SCALE_FACTOR',    5e-5*[1 1 1], ...
  'MISALIGNMENT_1', 0.5e-5*pi/180*[1 1], ...
  'MISALIGNMENT_2', 0.5e-5*pi/180*[1 1], ...
  'MISALIGNMENT_3', 0.5e-5*pi/180*[1 1], ...
  'NOISE',           1.7e-9*[1 1 1], ...
  'QUANTIZATION',    0.00146*pi/180*[1 1 1], ...
  'SEED',            0, ...
  'RATE_MIN',       -0.175, ...
  'RATE_MAX',        0.175 );

%% ------------------ EKF tunings
   ---------------------------------------
rng(42)
x0  = [0;0;0;1];
Q   = diag([2e-10 2e-10 2e-10]);
Rsun  = diag([0.7 0.7 0.8])*100;
Rmag  = diag([0.5 0.5 0.5])*100;
P0 = diag([1e-2 3e-2 7e-2])*0.1;

%% Attitudecontrol gains
K_p      = 0.017*[1 1 1];
K_d      = 0.15 *[1 1 1];
K_d_bdot = 2e5 *[1 1 1];
```

```
%% ------------------ Actuators
   ----------------------------------------
Actuators.Magnetorquers = struct( ...
  'Tc',0.2216,'mResolution',0.002*[1 1 1], ...
  'mMax',40*[1 1 1],'mMin',0.001*[1 1 1], ...
  'Bias',0.5*[1 1 1],'C',[1 1 1], ...
  'M',[ 0.9993  0.0267 -0.0256;
       -0.0260  0.9993  0.0267;
        0.0263 -0.0260  0.9993]);

Actuators.ReactionWheelsSimple = struct( ...
  'bearingnoise',zeros(3,1), ...
  'limit',0.1*[1 1 1], ...
  'delaytime',0.01*[1 1 1], ...
  'RES',5e-6*[1 1 1], ...
  'initial_t',zeros(3,1), ...
  'initial_m',zeros(3,1));
```

A.3 Extended Kalman Filter

Listing A.3: Extended Kalman Filter

```
% EKF Update Using Top 3 Sun Sensors + Magnetometer
function [x_out, P_out, sun_residual, mag_residual, K_out_combined,
     sigma_diag] = ekf_update_sun_mag_combined( ...
     Ny, Nz, lit_status, B_meas_body, omega_meas, x_in, P_in, psi,
        theta, phi, MJD, B_inertial, dt, R_sun, R_mag, Q)

    % --- Unpack state
    q_vec = x_in(1:3);
    q4 = x_in(4);
    omega_corr = omega_meas;

    % --- Prediction step
    q_pred = propagate_quaternion(x_in, omega_corr, dt);
    x_pred = q_pred(1:3);
    F = compute_F(q_pred, omega_corr, dt);
    P_pred = F * P_in * F' + Q;

    % --- Initialize outputs
    sun_residual = zeros(3, 1);
    mag_residual = zeros(3, 1);
    K_out_combined = zeros(3, 6);  % Fixed size

    x_out = x_in;
    P_out = P_pred;
    diagP = real(diag(P_out));
    diagP(~isfinite(diagP)) = 0;     % Handle NaN/Inf
    diagP(diagP < 0) = 0;            % Clip negative to zero
    sigma_diag = sqrt(diagP);        % Now safe to take sqrt

    %lit_status = 0;
    if lit_status == 1
        % SUN SENSOR UPDATE
        R_ib = quat_to_dcm(q_pred);
        S_inertial = SunPos(MJD);
        S_pred_body = R_ib * S_inertial;
        S_pred_body = S_pred_body / (norm(S_pred_body) + 1e-8);

        S_meas_ss = sun_vec_meas(Ny, Nz);
        Rbss = rotation_matrix_transpose(phi, theta, psi);
        S_meas_body = Rbss \ S_meas_ss;
        S_meas_body = S_meas_body / (norm(S_meas_body) + 1e-8);

        sun_residual = S_meas_body - S_pred_body;
        H_sun = compute_H_vector_sensor(x_pred, S_inertial);
        v = S_pred_body;
        J_norm = eye(3) - v * v';
        H_full_sun = J_norm * H_sun;

        % MAGNETOMETER UPDATE
        B_pred_body = R_ib * B_inertial;
        B_meas_body = B_meas_body / (norm(B_meas_body) + 1e-8);
        B_pred_body = B_pred_body / (norm(B_pred_body) + 1e-8);
        B_inertial = B_inertial / (norm(B_inertial) + 1e-8);

        mag_residual = B_meas_body - B_pred_body;
        H_mag = compute_H_vector_sensor(x_pred, B_inertial);
        v = B_pred_body;
        J_norm = eye(3) - v * v';
        H_full_mag = J_norm * H_mag;

        % COMBINED UPDATE
        residual_combined = [sun_residual; mag_residual];
        H_combined = [H_full_sun; H_full_mag];
        R_combined = blkdiag(R_sun, R_mag);

        S_combined = H_combined * P_pred * H_combined' + R_combined
            ;
        K_combined = P_pred * H_combined' / S_combined;

        delta_x_combined = K_combined * residual_combined;
        q_vec_updated = x_pred + delta_x_combined(1:3);
        q4_updated = sqrt(max(1e-12, 1 - dot(q_vec_updated,
            q_vec_updated)));
        x_out = [q_vec_updated; q4_updated];

        P_out = (eye(3) - K_combined * H_combined) * P_pred;
        P_out = 0.5 * (P_out + P_out');

         % --- Diagonal diagnostics
        diagP = real(diag(P_out));
        diagP(~isfinite(diagP)) = 0;
        diagP(diagP < 0) = 0;
        sigma_diag = sqrt(diagP);

        K_out_combined = K_combined;

    elseif lit_status == 0
        % MAGNETOMETER-ONLY UPDATE
        R_ib = quat_to_dcm(q_pred);
        B_pred_body = R_ib * B_inertial;

        B_meas_body = B_meas_body / (norm(B_meas_body) + 1e-8);
        B_pred_body = B_pred_body / (norm(B_pred_body) + 1e-8);
        B_inertial = B_inertial / (norm(B_inertial) + 1e-8);

        mag_residual = B_meas_body - B_pred_body;
        H_mag = compute_H_vector_sensor(x_pred, B_inertial);
        v = B_pred_body;
        J_norm = eye(3) - v * v';
        H_full_mag = J_norm * H_mag;

        residual_combined = mag_residual;
        H_combined = H_full_mag;
        R_combined = R_mag;

        S_combined = H_combined * P_pred * H_combined' + R_combined
            ;
        K_combined = P_pred * H_combined' / S_combined;

        delta_x_combined = K_combined * residual_combined;
        q_vec_updated = x_pred + delta_x_combined(1:3);
        q4_updated = sqrt(max(1e-12, 1 - dot(q_vec_updated,
            q_vec_updated)));
        x_out = [q_vec_updated; q4_updated];

        P_out = (eye(3) - K_combined * H_combined) * P_pred;
        P_out = 0.5 * (P_out + P_out');
         % --- Diagonal diagnostics
        diagP = real(diag(P_out));
        diagP(~isfinite(diagP)) = 0;
        diagP(diagP < 0) = 0;
        sigma_diag = sqrt(diagP);

        % Pad to 3x6 for consistent size
        K_out_combined = [K_combined, zeros(3, 3)];
    end

end


% Sun Vector Measurement Calculation
function u = sun_vec_meas(Ny, Nz)
    u3 = 1 / sqrt(Ny^2 + Nz^2 + 1);
    u1 = -Nz * u3;
    u2 = Ny * u3;
    u = [u1; u2; u3];
end

% Quaternion to Direction Cosine Matrix (DCM)
function C_q = quat_to_dcm(q)
    q1 = q(1); q2 = q(2); q3 = q(3); q4 = q(4);
    C_q = [
        q1^2 + q4^2 - q2^2 - q3^2,    2*(q1*q2 + q4*q3),     2*(q1*
            q3 - q4*q2);
        2*(q1*q2 - q4*q3),           q2^2 + q4^2 - q1^2 - q3^2,
           2*(q2*q3 + q4*q1);
        2*(q1*q3 + q4*q2),           2*(q2*q3 - q4*q1),      q3^2 +
           q4^2 - q1^2 - q2^2
    ];
end

% Rotation Matrix Transpose
function R_transpose = rotation_matrix_transpose(phi, theta, psi)
    cphi = cosd(phi);   sphi = sind(phi);
    ctheta = cosd(theta); stheta = sind(theta);
    cpsi = cosd(psi);   spsi = sind(psi);

    Cfssb = [cpsi*ctheta,      ctheta*spsi,      -stheta;
             sphi*cpsi*stheta - spsi*cphi,  stheta*spsi*sphi + cpsi
                *cphi,  sphi*ctheta;
             stheta*cphi*cpsi + spsi*sphi,  stheta*cphi*spsi - cpsi
                *sphi,  cphi*ctheta];

    R_transpose = Cfssb;
end

% Compute Jacobian of the Sensor with Respect to Quaternion State
function H = compute_H_vector_sensor(q_in, V_inertial)
    if length(q_in) == 3
        % Only vector part provided
        q_vec = q_in;
        q4 = sqrt(max(1e-12, 1 - dot(q_vec, q_vec)));
        q_full = [q_vec; q4];  % Create full quaternion
    else
        q_full = q_in;
    end

    % Unpack full quaternion
    q1 = q_full(1); q2 = q_full(2); q3 = q_full(3); q4 = q_full(4);

    % Compute partial derivatives
    dRv_dq1 = 2 * ( ...
        [q1, q2, q3; q2, -q1, q4; q3, -q4, -q1] * V_inertial ...
        + (q1/q4) * [q4, q3, -q2; -q3, q4, q1; q2, -q1, q4] *
           V_inertial );

    dRv_dq2 = 2 * ( ...
        [-q2, q1, -q4; q1, q2, q3; q4, q3, -q2] * V_inertial ...
        + (q2/q4) * [q4, q3, -q2; -q3, q4, q1; q2, -q1, q4] *
           V_inertial );

    dRv_dq3 = 2 * ( ...
        [-q3, q4, q1; -q4, -q3, q2; q1, q2, q3] * V_inertial ...
        + (q3/q4) * [q4, q3, -q2; -q3, q4, q1; q2, -q1, q4] *
           V_inertial );

    H = [dRv_dq1, dRv_dq2, dRv_dq3];  % Jacobian matrix
end

% Propagate Quaternion State
```

```
function q_next = propagate_quaternion(q, omega, dt)
    q = q / norm(q);  % Ensure unit quaternion
    q1 = q(1); q2 = q(2); q3 = q(3); q4 = q(4);

    Omega = [ ...
         q4, -q3,  q2;
         q3,  q4, -q1;
        -q2,  q1,  q4;
        -q1, -q2, -q3 ];

    q_dot = 0.5 * Omega * omega;
    q_next = q + q_dot * dt;
    q_next = q_next / norm(q_next);  % Re-normalize to avoid drift
end

function y = Frac(x)
    % Output the fraction of a real number
    y = x - floor(x);
end

% Compute the State Transition Matrix
function F = compute_F(q, omega, dt)
    Omega_cross = 0.5 * [ ...
          0,     omega(3), -omega(2);
        -omega(3),     0,     omega(1);
         omega(2), -omega(1),     0 ];

    F = eye(3) + Omega_cross * dt;
end

function RSun = SunPos(MJD)
    % Computes the Sun's geocentric position based on Julian date
    eps = deg2rad(23.43929111); % Earth's axial tilt
    MJD_J2000 = 51544.5; % Modified Julian Date of J2000.0
    T = (MJD-MJD_J2000)/36525.0; % Julian century since J2000
    twopi = 2*pi;
    M = twopi * Frac(0.9931267 + 99.9973583*T);
    L = twopi * Frac(0.7859444 + M/twopi + (6892.0*sin(M)+72.0*sin
       (2.0*M)) / 1296.0e3);
    r = 149.619e9 - 2.499e9*cos(M) - 0.021e9*cos(2*M);
    seps = sin(eps);
    ceps = cos(eps);
    Cx = [1 0 0; 0 ceps -seps; 0 seps ceps];
    RSun = Cx * [r*cos(L); r*sin(L); 0.0];
end
```

A.4 rtwbuild Script (MATLAB)

Listing A.4: rtwbuild Script

```
% Define the list of model names
modelNames = {'TIME','SS', 'SRP', 'RDY', 'RB', 'MGMTR', 'MAG', 'I2S
    ', 'GYR', 'GRAV','EKF','CBI','ATM','BDOT'}; % Add all model
    names here
```

```

% Loop through each model and apply settings
for i = 1:length(modelNames)
    modelName = modelNames{i};

    try
        % Open the model
        open_system(modelName);
        fprintf('Applying␣settings␣to:␣%s\n', modelName);

        % Set solver settings
        set_param(modelName, 'Solver', 'FixedStepDiscrete'); % Correct fixed-step solver
        set_param(modelName, 'FixedStep', '0.01'); % Fixed step size
        set_param(modelName, 'SolverName', 'ode4'); % 4th order Runge-Kutta

        % Set Code Generation settings
        set_param(modelName, 'SystemTargetFile', 'grt.tlc'); % Ensure GRT target for code generation
        set_param(modelName, 'GenCodeOnly', 'on'); % Generate code only
        set_param(modelName, 'PackageGeneratedCodeAndArtifacts', 'on'); % Package code and artifacts

        % Optimize interface settings for reusable functions
        set_param(modelName, 'CodeInterfacePackaging', 'Reusable␣function'); % Make functions reusable

        % Optimize parameter handling
        set_param(modelName, 'DefaultParameterBehavior', 'Tunable'); % Ensure parameters can be changed
        set_param(modelName, 'InlineParams', 'off'); % Avoid hardcoded parameters

        % Disable unnecessary logging
        set_param(modelName, 'CombineOutputUpdateFcns', 'off'); % Disable Single output/update function
        set_param(modelName, 'MatFileLogging', 'off'); % Disable MAT-File Logging

        % **Disable SSE2 (emmintrin.h) Optimization (Correct way)**
        set_param(modelName, 'InstructionSetExtensions', 'None'); % Turns off SSE2, AVX, and all SIMD optimizations

        % Save model after changes
        save_system(modelName);

        % Generate code
        fprintf('Generating␣code␣for:␣%s\n', modelName);
        rtwbuild(modelName);

        % Close the model (optional)
        close_system(modelName, 0);

        fprintf('Successfully␣updated␣and␣generated␣code␣for:␣%s\n\n', modelName);

    catch ME
        % Handle errors without stopping the loop
        fprintf(Error processing %s: %s\n', modelName, ME.message);
    end
end

fprintf('Code␣generation␣completed␣for␣all␣models.\n');
```

A.5 Example of .c files after simulink2c applied

Listing A.5: simulink2c Converter outcome

```
/*
  'Global_State_Variables
␣␣␣␣ExtU_ATM_T␣ExtU[1],
␣␣␣␣ExtY_ATM_T␣ExtY[1]
␣␣
␣␣'Entry_Point initialize
  'Entry_Point␣initialize_0
␣␣
␣␣'Entry_Point output
  'Entry_Point␣output_0
␣␣
␣␣'Entry_Point update
  'Entry_Point␣update_0
␣␣
␣␣
␣␣'Entry_Point terminate
  'Entry_Point␣terminate_0
*/


/*␣-----------------------------------------------------------␣*/
/*␣This␣model␣is␣converted␣by␣EuroSim's simulink2c convertor   */
/* Date: 2025-05-04T21:58:48+08:00                    */
/* Model: 1.7                     */
/* Coder: 23.22023012023                    */
/* ----------------------------------------------------------- */
#ifdef __eurosim__
#define malloc(size) esimMalloc(size)
#define calloc(nmemb,membsize) esimCalloc(nmemb,membsize)
#define realloc(ptr,size) esimRealloc(ptr,size)
#define free(address) esimFree(address)
#define rt_FREE(address) esimFree(address)
#include "ATM.h"
#include "rtwtypes.h"
#include <math.h>
#include <string.h>
#include "ATM_private.h"
#include <esim.h>
static RT_MODEL_ATM_T Model[1];
static ExtU_ATM_T ExtU[1];
static ExtY_ATM_T ExtY[1];
static int model_index;
#endif

/*
 * ATM.c
 *
 * Academic License - for use in teaching, academic research, and
    meeting
 * course requirements at degree granting institutions only.  Not
    for
 * government, commercial, or other organizational use.
 *
 * Code generation for model "ATM".
 *
 * Model version               : 1.7
 * Simulink Coder version : 23.2 (R2023b) 01-Aug-2023
 * C source code generated on : Sun May  4 15:42:38 2025
 *
 * Target selection: grt.tlc
 * Note: GRT includes extra infrastructure and instrumentation for
    prototyping
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objectives: Unspecified
 * Validation result: Not run
 */
 \end{list}

\section{Cost}

\begin{table}[h!]
\centering
\caption{Hardware cost breakdown for the simulator setup.}
\renewcommand{\arraystretch}{1.2}
\begin{tabular}{|p{8cm}|S[table-format=3.2]|}
\hline
\rowcolor{chapterband}
\textcolor{chaptercolor}{\textbf{Hardware}} & \textcolor{
   chaptercolor}{\textbf{Cost [Euros]}} \\
\hline
Raspberry Pi 4 (4GB) & 60.00 \\
Raspberry Pi 5 (8GB) & 65.00 \\
Orange Pi (32GB) & 150.00 \\
Pi 5 metal case & 15.72 \\
NVMe PCIe Gen3x4 Kingspec SSD & 77.26 \\
Pineberry Pi interface & 14.95 \\
Lexar NM620 1TB SSD & 59.90 \\
HDMI cable & 3.98 \\
Micro HDMI cable & 10.99 \\
Ethernet cable & 8.99 \\
Keyboard & 27.99 \\
Mouse & 8.95 \\
Mini screwdriver set & 35.95 \\
RTC clock module & 5.19 \\
Adapter (USB to flash drive) & 23.99 \\
\hline
\rowcolor{gray!15}
\textbf{Total} & \textbf{568.86} \\
\hline
\end{tabular}
\label{tab:hardware_cost}
\end{table}






%\begin{lstlisting}[language=Matlab,caption={rtwbuild Script},label
    ={lst:rtwbuild}]
%
```

Bibliography

Balaban, E., Saxena, A., Bansal, P., Goebel, K., and Curran, S. Modeling, detection, and disambiguation of sensor faults for aerospace applications. *Sensors Journal, IEEE*, 9:1907 – 1917, 01 2010. doi: 10.1109/JSEN.2009.2030284.

Bashier, F. Design process-system and methodology of design research. In *IOP Conference Series: Materials Science and Engineering*, volume 245, page 082030. IOP Publishing, 2017. doi: 10.1088/1757-899X/245/8/082030. URL https://doi.org/10.1088/1757-899X/245/8/082030.

Carignan, C., Scott, N., and Roderick, S. Hardware-in-the-loop simulation of satellite capture on a ground-based robotic testbed. In *ISAIRAS Conference Proceedings*, USA, 2022. NASA Goddard Space Flight Center.

Chapman, P., Colegrove, T., Di Filippantonio, D., Walker-Deemin, A., Davies, A., Myatt, J., Ecale, E., and Girouart, B. The gaia attitude & orbit control system. 06 2008.

Dam, R. F. and Siang, T. Y. 5 stages in the design thinking process. Interaction Design Foundation, 2024. Accessed: 2024-11-06.

de Vries, W. Cubesat drag calculations, Sep 2010. Accessed: 2025-05-05.

European Space Agency. Real-time satellite network emulator. https://connectivity.esa.int/projects/realtime-satellite-network-emulator, 2022. Accessed: 2025-04-26.

Farissi, M. S., Carletta, S., Nascetti, A., and Teofilatto, P. Implementation and hardware-in-the-loop simulation of a magnetic detumbling and pointing control based on three-axis magnetometer data. *Aerospace*, 6(12):133, 2019.

Frezza, L., Marzioli, P., Moretti, A., Kumar, S., Boscia, M., Bedetti, E., Picci, N., Gianfermo, A., Amadio, D., Curiano, F., Piergentili, F., Gugliermetti, L., and Santoni, F. Shared cubesat bus approach for the design and development of the sapienza s5lab nano-satellites. pages 480–485, 06 2022. doi: 10.1109/MetroAeroSpace54187.2022.9856103.

García Ortega, J., Tarrida, C. L., Quero, J. M., Delgado, F. J., Ortega, P., Castañer, L., Reina, M., Angulo, M., Morilla, Y., and García López, J. Mems solar sensor testing for satellite applications. In *Conference on Design of MEMS for Aerospace Applications*. IEEE, 2009. doi: 10.1109/SCED.2009.4800503.

Guon, J., Monas, L., and Gill, E. Statistical analysis and modelling of small satellite reliability. *Faculty of Aerospace Engineering, Delft University of Technology, The Netherlands*, January 2014.

Haraguchi, A. A hardware-in-the-loop star tracker test bed. Master's thesis, California Polytechnic State University, San Luis Obispo, 2024.

Kassem, M. M. and Sastry, N. xeoverse: A real-time simulation platform for large leo satellite mega-constellations. In *Proceedings of IFIP/IEEE Networking 2024*, 2024. doi: 10.48550/arXiv.2406.11366. URL https://arxiv.org/abs/2406.11366.

Kemal Ure, N., Kaya, Y., and Inalhan, G. The development of a software and hardware-in-the-loop test system for itu-psat ii nano satellite adcs. pages 1 – 15, 04 2011. doi: 10.1109/AERO.2011.5747481.

Kiesbye, J., Messmann, D., Preisinger, M., Reina, G., Nagy, D., Schummer, F., Mostad, M., Kale, T., and Langer, M. Hardware-in-the-loop and software-in-the-loop testing of the move-ii cubesat. *Aerospace*, 6(12), 2019. ISSN 2226-4310. doi: 10.3390/aerospace6120130.

Mihalic, F., Truntic, M., and Hren, A. Hardware-in-the-loop simulations: A historical overview of engineering challenges. *Electronics*, 11(24):2462, 2022. doi: AddDOIifavailable.

Montenbruck, O. and Gill, E. *Satellite Orbits: Models, Methods and Applications*. Springer, first edition, January 2001.

Mooij, E. On-board constrained optimisation of final rendezvous: Technical notes 2 - functional simulator design (unpublished). Delft University of Technology, 2022a. TU Delft.

Mooij, E. *Lecture notes in Re-entry Systems*. Delft University of Technology, October 2022b.

Mooij, E. and Ellenbroek, M. *Multi-Functional Guidance, Navigation and Control Simulation Environment - Rapid Prototyping of Space Simulations*. IntechOpen, United Kingdom, 2011. ISBN 978-953-307-970-7. doi: 10.5772/830.

Mwangi-Mbuthia, J. and Ouma, H. *1KUNS-PF: 1st Kenyan University NanoSatellite-Precursor Flight*. University of Nairobi, March 2016.

Oomen, M. M. The design of an attitude control system for the sps-2 satellite. Master's thesis, TU Delft, July 2020.

OpenCourseWare, M. Essentials of geophysics, chapter 2, 2004. URL https://ocw.mit.edu/courses/12-201-essentials-of-geophysics-fall-2004/7fa24d336366b74c52adb48ae6c8cf6f_ch2.pdf. Accessed: 2024-12-03.

Polat, H., Virgili-Llop, J., and Romano, M. Survey, statistical analysis and classification of launched cubesat missions with emphasis on the attitude control method. *Journal of Small Satellites*, 5:513–530, 01 2016.

Post, M. A., Li, J., and Lee, R. A low-cost photodiode sun sensor for cubesat and planetary microrover. *International Journal of Aerospace Engineering*, 2013:Article ID 549080, 2013. doi: 10.1155/2013/549080.

Puig-Suari, J., Turner, C., and Ahlgren, W. Development of the standard cubesat deployer and a cubesat class picosatellite. In *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*, volume 1, pages 1/347–1/353 vol.1, 2001. doi: 10.1109/AERO.2001.931726.

Risquez, D., van Leeuwen, F., and Brown, A. G. A. Dynamical attitude model for gaia. *Experimental Astronomy*, 34:669–703, July 2012.

Rodrigues, P. M. and Ramos, P. M. Design and characterization of a sun sensor for the sseti-eseo project. In *XVIII IMEKO World Congress: Metrology for a Sustainable Development*, Rio de Janeiro, Brazil, 2006. IMEKO.

Sanchez-Portal, M., Marston, A., Altieri, B., Aussel, H., Feuchtgruber, H., Klaas, U., Linz, H., Lutz, D., Merın, B., Muller, T., Nielbock, M., Oort, M., Pilbratt, G., Schmidt, M., Stephenson, C., Tuttlebee, M., and Group, T. H. P. W. The pointing system of the herschel space observatory. *Experimental Astronomy*, 37(2), May 2014. ISSN 1572-9508. doi: 10.1007/s10686-014-9396-z.

Scharnagl, J. and Schilling, K. New hardware-in-the-loop testing concept for small satellite formation control based on mobile robot platforms. In *IFAC-PapersOnLine*, volume 49, pages 65–70. Zentrum für Telematik e.V., Würzburg, Germany, Elsevier Ltd, 2016. doi: 10.1016/j.ifacol.2016.11.127.

Tafazoli, S. A study of on-orbit spacecraft failures. *Acta Astronautica - ACTA ASTRONAUT*, 64:195–205, 02 2009. doi: 10.1016/j.actaastro.2008.07.019.

Vallado, D. A. and McClain, W. D. *Fundamentals of Astrodynamics and Applications*. Space Technology Library. Microcosm Press, Hawthorne, CA, fourth edition edition, 2013. ISBN 978-1881883180. First Printing, printed on acid-free paper.

Wertz, J. *Spacecraft Attitude Determination and Control*. Kluwer Academic Publishers, first edition, January 1980.