Development of a Guidance, Navigation, and Control Simulation Framework for Small Satellites in Low Earth Orbit

a project presented to
The Faculty of the Department of Aerospace Engineering
San José State University

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

by

Edwin Christhuraj

June 2026

approved by

Dr. Periklis Papadopoulos
Faculty Advisor



© 2026
Edwin Christhuraj
ALL RIGHTS RESERVED

ABSTRACT

Development of a Guidance, Navigation, and Control Simulation Framework for Small Satellites in Low Earth Orbit

Edwin Christhuraj

Modern small satellite missions in low Earth orbit increasingly demand sophisticated Guidance, Navigation, and Control (GNC) subsystems to achieve complex operational objectives such as precise target pointing, station-keeping, and formation flight. While various commercial and open-source GNC simulation tools are available, their highly abstracted architectures often create a "black box" environment that obscures foundational engineering principles from university students and small satellite developers. This paper presents the development of GIMBALS, a modular, six degree-of-freedom GNC software simulation framework specifically designed to minimize software abstraction and maximize code interpretability. Implemented within the Matlab/Simulink environment, the framework models decoupled rigid body orbital and attitude dynamics under environmental perturbations. High-fidelity mathematical models are developed for space-segment hardware, including magnetometers, sun sensors, Earth sensors, star trackers, gyroscopes, accelerometers, magnetorquers, and reaction wheels. GIMBALS partitions the onboard flight software into explicit, isolated modules for mode management, guidance target generation, navigation and state estimation, and actuator control allocation. To validate the simulation framework, the magnetic detumbling of a 2U CubeSat is simulated and evaluated. An Extended Kalman Filter is implemented for state estimation and three-axis magnetorquers for attitude control. Simulation results confirm successful stabilization and match estimated body angular rates closely with real-world truth data, demonstrating the efficacy of GIMBALS as a highly transparent verification tool for future small satellite development.

Acknowledgments

I would like to begin by expressing my deepest gratitude to my advisor, Dr. Periklis Papadopoulos, for your guidance and patience in supporting my academic journey as a graduate student. You have been an invaluable resource in helping me cultivate and refine my technical understanding of modern aerospace concepts. Your eagerness to keep me involved and engaged, even amongst my many obligations, has been indispensable for maintaining my continued career and academic interests in aerospace engineering.

To Dr. Nikos Mourtos, I would like to extend my heartfelt gratitude for your encouragement in helping me choose the Master of Science in Aerospace Engineering at San Jose State University and for welcoming me into the Department of Aerospace Engineering as a graduate student.

To all my professors in the Department of Aerospace Engineering, I am honored to have had the opportunity to grow intellectually under your auspices and to contribute positively to your courses.

To Mark Danehy and all the Mission Operations Department staff at Lanteris Space Systems, I would like to extend my sincere thank you for your flexibility and accommodation while I balanced my professional obligations at Lanteris alongside my academic obligations at SJSU.

Finally, to my mother, sister, and grandmother for their unwavering support and faith in my ability to succeed as the first in my family to pursue graduate level studies, I am immeasurably grateful for your investment in my well-being and achievement.

Table of Contents

List of Symbols ..... **vii**
List of Tables ..... **viii**
List of Figures ..... **ix**
Chapter 1 – Introduction **1**

1.1 Motivation 1
1.2 Literature Review 1
1.3 Project Proposal 8
1.4 Methodology 8

Chapter 2 – Conceptual Overview **9**

2.1 GNC Subsystem Architecture 9
2.2 Spacecraft Dynamics 11
2.2.1 Orbital Dynamics 11
2.2.2 Attitude Dynamics 12
2.3 Sensors 14
2.3.1 Attitude Sensors 15
2.3.2 Inertial Sensors 22
2.3.3 Orbit Sensors 25
2.4 Actuators 25
2.4.1 Thrusters 26
2.4.2 Magnetorquers 27
2.4.3 Reaction Wheels 28
2.4.4 Control-Moment Gyroscopes 30
2.5 On-Board Computer 33
2.5.1 Mode Management 34
2.5.2 Guidance 36
2.5.3 Navigation 36
2.5.4 Control 38
2.6 Space Environment 39
2.6.1 Earth Gravity 39
2.6.2 Atmospheric Drag 40
2.6.3 Magnetic Field 41
2.6.4 Solar Radiation Pressure 42
2.6.5 Third Body Perturbations 42

Chapter 3 – Software Implementation **44**

3.1 Design Overview 44
3.2 Software Structure 44
3.2.1 The Simulation Object 45
3.2.2 The EarthModel Object 47
3.2.3 The Satellite Object 49
3.2.4 Sensor Objects 53
3.2.5 Actuator Objects 56
3.2.6 The Global Target Generator (GTG) 60
3.2.7 The Global State Estimator (GSE) 63

3.2.8 The Actuator Command Generator (ACG) 66
3.2.9 The Mode Manager 69
3.3 Software Requirements 73
Chapter 4 – Testing and Validation 75
Chapter 5 – Conclusion 78
References 79
Appendix A – GIMBALS Project Folder Structure 81
Appendix B – GIMBALS Input File for TestSat Detumbling Simulation 83
Appendix C – GIMBALS Global Target Generator for Detumbling Test Case 88
Appendix D – GIMBALS Global State Estimator for Detumbling Test Case 89
Appendix E – GIMBALS Actuator Command Generator for Detumbling Test Case ... 92
Appendix F – GIMBALS Satellite Dynamics Model Matlab Function 95

List of Symbols

Symbol	Definition	Units (SI)
$\vec{r}$	Position	m
$\vec{g}$	Gravitational Acceleration	m/s^2
μ	Gravitational Parameter	m^3/s^2
$\vec{F}$	Force	N
m	Mass	kg
$\vec{\omega}$	Angular Velocity	rad/s
$\vec{H}$	Angular Momentum	$kg \cdot m^2/s$
J	Mass Moment of Inertia	$kg \cdot m^2$
$\vec{M}$	Moment (Torque)	N·m
a	Keplerian Orbit Semi-Major Axis	m
e	Keplerian Orbit Eccentricity	—
t	Epoch Time	s
Ω	Keplerian Orbit Right Ascension	rad
i	Keplerian Orbit Inclination	rad
ω	Keplerian Orbit Argument of Perigee	rad
$\vec{q}$	Quaternion	—
$\vec{B}$	Magnetic Field	T
$\vec{s}$	Sun Vector	—
$\vec{n}$	Nadir Vector	—
$\vec{\epsilon}$	Sum of Errors and Biases	—
N	Number of Coils	—
I	Current	A
A	Cross Sectional Area	m^2
V_c	Ferromagnetic Core Volume	m^3
μ_r	Relative Magnetic Permeability	—
$\vec{m}$	Magnetic Dipole	$A \cdot m^2$
$(\ )^{\mathcal{N}}$	Inertial Frame Reference	—
$(\ )^{\mathcal{B}}$	Body Frame Reference	—
$(\ )^{\mathcal{W}}$	Wheel Frame Reference	—
$(\ )_{SC}$	Satellite Related Quantity	—
$(\ )_{MTM}$	Magnetometer Related Quantity	—
$(\ )_{SS}$	Sun Sensor Related Quantity	—
$(\ )_{GYRO}$	Gyroscope Related Quantity	—
$(\ )_{ACC}$	Accelerometer Related Quantity	—
$(\ )_{ST}$	Star Tracker Related Quantity	—
$(\ )_{GPS}$	GPS Related Quantity	—
$(\ )_{MTQ}$	Magnetorquer Related Quantity	—
$(\ )_{RW}$	Reaction Wheel Related Quantity	—
$(\ )_{CMG}$	CMG Related Quantity	—

List of Tables

Table 3.1 – GIMBALS Simulation object API reference. 45
Table 3.2 – GIMBALS EarthModel object API reference. 47
Table 3.3 – GIMBALS Satellite object API reference. 49
Table 3.4 – GIMBALS Accelerometer object API reference. 53
Table 3.5 – GIMBALS Magnetorquer object API reference. 56
Table 3.6 – GIMBALS ReactionWheel object API reference. 57
Table 3.7 – GIMBALS Guidance object API reference. 60
Table 3.8 – GIMBALS GTG object API reference. 61
Table 3.9 – GIMBALS Navigation object API reference. 63
Table 3.10 – GIMBALS GSE object API reference. 64
Table 3.11 – GIMBALS Control object API reference. 66
Table 3.12 – GIMBALS ACG object API reference. 67
Table 3.13 – GIMBALS ModeManager object API reference. 69
Table 3.14 – GIMBALS ADCSMode object API reference. 70
Table 3.15 – GIMBALS MCS object API reference. 72

List of Figures

Figure 1.1 – Explorer 1. left: planned spin axis. right: actual spin axis. 2
Figure 1.2 – Syncom 3. left: structural diagram. right: GNC subsystem diagram. 3
Figure 1.3 – ATS-6. left: GNC subsystem diagram. right: attitude control block diagram. 4
Figure 1.4 – GOES-8 structural diagram. 5
Figure 1.5 – Typical CubeSat design. 7
Figure 2.1 – Mission requirement inputs and subsystem inputs for GNC subsystem. 9
Figure 2.2 – Conventional space and ground segment GNC subsystem architecture. 10
Figure 2.3 – Maximally autonomous space segment only GNC subsystem architecture. 10
Figure 2.4 – Satellite orbit parametrization using Keplerian elements. 12
Figure 2.5 – Satellite undergoing rigid body rotational motion about its COM. 13
Figure 2.6 – Generic sensor model block diagram. 15
Figure 2.7 – Single axis coarse sun sensor diagram. 16
Figure 2.8 – Fine sun sensor diagram. 17
Figure 2.9 – Digital sun sensor diagram. 17
Figure 2.10 – Static Earth sensor diagram. 19
Figure 2.11 – Static Earth sensor thermopile operation. 19
Figure 2.12 – Star tracker hardware diagram. 21
Figure 2.13 – Generic inertial sensor model block diagram. 23
Figure 2.14 – Generic actuator model block diagram. 26
Figure 2.15 – Typical magnetorquer set layout. 28
Figure 2.16 – Generic reaction wheel layout. 30
Figure 2.17 – CMG working principle diagram. 31
Figure 2.18 – Four CMG pyramid layout. 33
Figure 2.19 – Typical CubeSat GNC flight software block diagram. 34
Figure 2.20 – Mode manager for Cat-2 CubeSat. 35
Figure 2.21 – Diagram of a sequential navigation filter. 37
Figure 2.22 – Earth atmospheric density model evolution. 41
Figure 3.1 – GIMBALS Simulation Simulink representation. 47
Figure 3.2 – GIMBALS EarthModel Simulink representation. 48
Figure 3.3 – GIMBALS Satellite overall Simulink representation. 51
Figure 3.4 – GIMBALS Satellite on-board computer Simulink representation. 52
Figure 3.5 – GIMBALS Satellite physical pystem Simulink representation. 53
Figure 3.6 – GIMBALS Accelerometer Simulink representation. 55
Figure 3.7 – GIMBALS Magnetorquer Simulink representation. 59
Figure 3.8 – GIMBALS ReactionWheel Simulink representation. 60
Figure 3.9 – GIMBALS Guidance Simulink representation. 62
Figure 3.10 – GIMBALS GTG Simulink representation. 63
Figure 3.11 – GIMBALS Navigation Simulink representation. 65
Figure 3.12 – GIMBALS GSE Simulink representation. 66
Figure 3.13 – GIMBALS Control Simulink representation. 68
Figure 3.14 – GIMBALS ACG Simulink representation. 69
Figure 3.15 – GIMBALS ModeManager Simulink representation. 72
Figure 3.16 – GIMBALS MCS Simulink representation. 73
Figure 4.1 – True satellite quaternion states for GIMBALS magnetic detumbling test case. 76
Figure 4.2 – True satellite body angular rates for GIMBALS magnetic detumbling test case. 76

Figure 4.3 – EKF estimated satellite body angular rates for GIMBALS magnetic detumbling test case. 77
Figure 4.4 – True satellite body angular rates overlayed with EKF estimated body angular rates for GIMBALS magnetic detumbling test case. 77
Figure A.1 – GIMBALS overall project folder structure. 81
Figure A.2 – GIMBALS userdata folder structure. 81
Figure A.3 – GIMBALS systems folder structure. 81
Figure A.4 – GIMBALS sensors folder structure. 81
Figure A.5 – GIMBALS environment folder structure. 82
Figure A.6 – GIMBALS algorithms folder structure. 82
Figure A.7 – GIMBALS actuators folder structure. 82

Chapter 1 – Introduction

1.1 Motivation

Historically, the process of designing, building, and launching satellites into space required costly investments into prolonged system design phases, manufacturing processes, regulatory compliance, and launch vehicle costs. With total end-to-end costs in the tens of millions, or even billions of dollars, satellites were prohibitively expensive for most organizations, aside from government entities like NASA and certain well-funded contractors. However, modern advances in spaceflight technologies and the rapid commercialization of space in recent years have enabled dramatic reductions in space mission development, launch, and operating costs, allowing greater accessibility to space as a domain for private enterprise, defense applications, and scientific experimentation. One of the most consequential outcomes of this has been the utilization of low Earth orbit (LEO) for small satellite missions by private companies and universities.

A small satellite is generally classified as an orbital spacecraft with a mass less than 180 kg, with different sub-classifications based on the exact mass range; for example, a nanosatellite is a small satellite that is in the 1 - 10 kg range. While many private companies build small satellites that are larger than nanosatellites, universities typically work with nanosatellites, often with a CubeSat form factor. To date, thousands of small satellites have successfully been launched into LEO, including a suite of TechEdSat CubeSats developed by SJSU in collaboration with NASA Ames. However, mission objectives for upcoming small satellite missions are becoming increasingly complex, often requiring precise attitude determination and control to point towards specific targets (i.e. solar, nadir, lunar, etc.) or even precise orbit determination and control (i.e. stationkeeping, formation flight). For these satellites to be able to properly reorient themselves and modify their orbits to meet mission objectives, they must be designed with a robust Guidance, Navigation, and Control (GNC) subsystem. While GNC subsystems have flight heritage for small satellite systems more broadly, they appear to have seen limited usage in past SJSU CubeSats [1], with the most sophisticated concept proposed only utilizing a single reaction wheel for 1 degree-of-freedom (DOF) attitude control and a single magnetorquer for wheel desaturation [2]. The author expects SJSU's future small satellite project initiatives and their collaborations with NASA to be greatly enhanced by demonstrating a practical understanding of modern spacecraft GNC subsystems.

1.2 Literature Review

The Guidance, Navigation, and Control (GNC) subsystem is responsible for estimating the kinematic states of a spacecraft, while also actively manipulating the system's dynamics to achieve states that are desirable to the mission objective. It is one of the most ubiquitous subsystems found on modern spacecraft as mission requirements usually necessitate its inclusion into the spacecraft's design. However, this has not always been the case historically for satellite systems. The earliest satellites had very rudimentary GNC architectures, if any at all, as they often did not require active orbit maintenance or pointing control to achieve mission objectives. For example, the first orbital satellite, Sputnik 1 (1957), was a technology demonstration by the Soviet Union and it acted as a simple radio beacon, tumbling freely in orbit with no active controls. Explorer 1 (1958), launched by the United States, was spin stabilized about its longitudinal axis by the Jupiter-C rocket it launched on; however, shortly after orbit insertion, Explorer 1 experienced disturbance forces (gravitational, magnetic, aerodynamic, solar, etc.) and kinetic energy dissipation through its flexible turnstile whip antennas that caused its planned minor axis spin to resolve into a more

dynamically stable flat spin about its major axis as shown in Figure 1.1 [3]. This and other lessons learned from early satellite missions motivated greater investment into the study of rigid body dynamics and control for space applications.

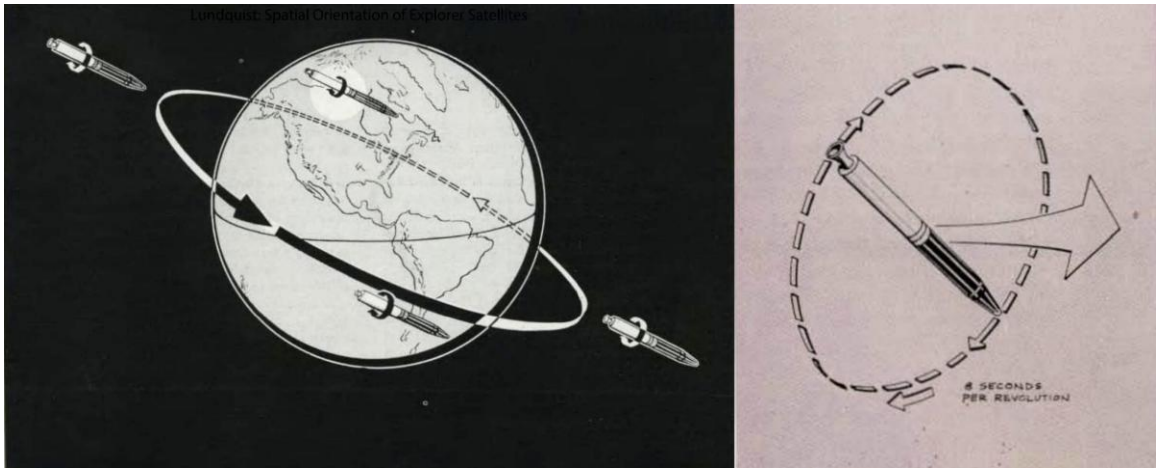


Figure 1.1 – Explorer 1. left: planned spin axis. right: actual spin axis.

In the 1960s, improvements in spin stabilization techniques and the introduction of onboard propulsion systems for attitude and orbit control (AOC) enabled the development of more reliable and versatile satellites for both government and commercial activities. A notable example is NASA's Syncom 3 (1964), the first ever geostationary communications satellite, which was used to live broadcast the 1964 Tokyo Olympic games and then later used by the Department of Defense for Vietnam War communications. Syncom 3 was spin stabilized and successfully utilized a solid rocket apogee motor to remove inclination and circularize its orbit at the geostationary altitude. It also used nitrogen and hydrogen peroxide AOC thrusters, which allowed for spin axis alignment and orbital velocity adjustments [4]. The GNC subsystem diagram for Syncom 3 is shown alongside a structural diagram in Figure 1.2. While very rudimentary by modern satellite GNC standards, it was sufficient for accomplishing mission objectives. The success of Syncom 3 was followed by the emergence of the commercial communications satellite industry, which remains the largest and most profitable private space-based industry to date [5]. The original market leaders were Ford Aerospace, RCA AstroElectronics, and Hughes Space and Communications, all of which have undergone numerous corporate mergers and acquisitions over time and are better known today as Lanteris Space Systems (Intuitive Machines), Lockheed Martin Space Systems, and Boeing Satellite Systems, respectively.

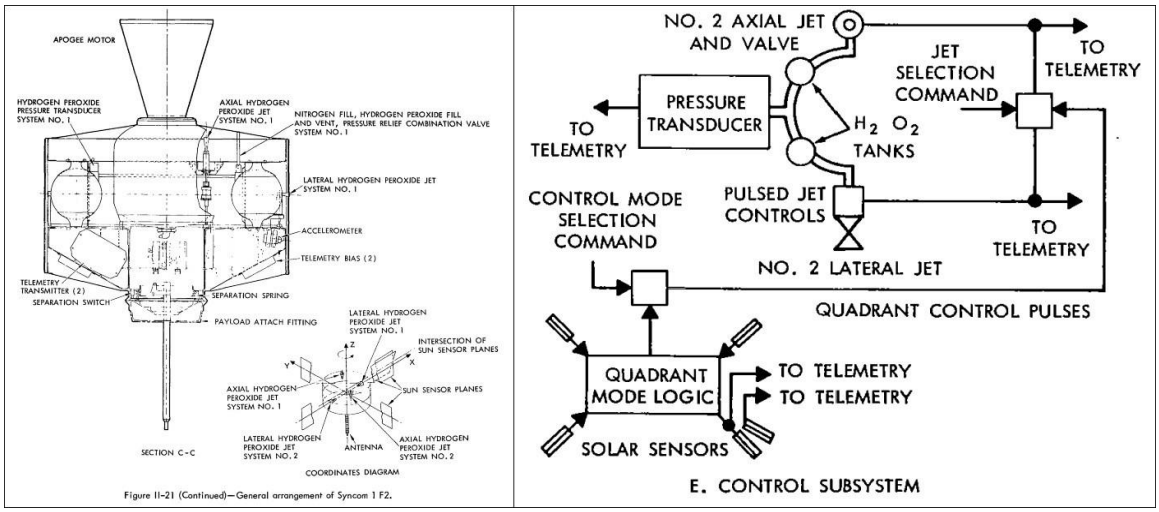


Figure 1.2 – Syncom 3. left: structural diagram. right: GNC subsystem diagram.

Spin stabilization is a largely passive control strategy that utilizes the conservation of angular momentum to provide gyroscopic rigidity about the spin axis. Therefore, it offers limited control flexibility in situations where a payload, such as a directional antenna or optical imager, requires a steady platform for pointing. NASA's experimental ATS-6 (1974) was the first satellite to utilize active three-axis stabilization as part of its GNC subsystem. Older approaches to three-axis control, like those used on the Apollo Command Module and the Space Shuttle, used thruster-based reaction control systems and human piloting for guidance and control [5]; however, these approaches are less practical for satellites, which are uncrewed and have much longer mission lifespans. The primary mission objective of ATS-6 was to demonstrate direct-to-home television broadcasting technology from geostationary orbit, requiring the satellite's directional payload antenna to provide continuously uninterrupted coverage to specific areas on the ground, thus necessitating the use of three-axis stabilization. To accomplish this, ATS-6 used a set of 3 reaction wheels and AOC thrusters, along with a suite of sensors including sun sensors, mechanical gyroscopes, an Earth sensor, and a Polaris star tracker. Closed-loop feedback control algorithms allowed these sensors and actuators to achieve pointing accuracies as low as 0.002 degrees. The ATS-6 GNC subsystem diagram is shown in Figure 1.3 [6].

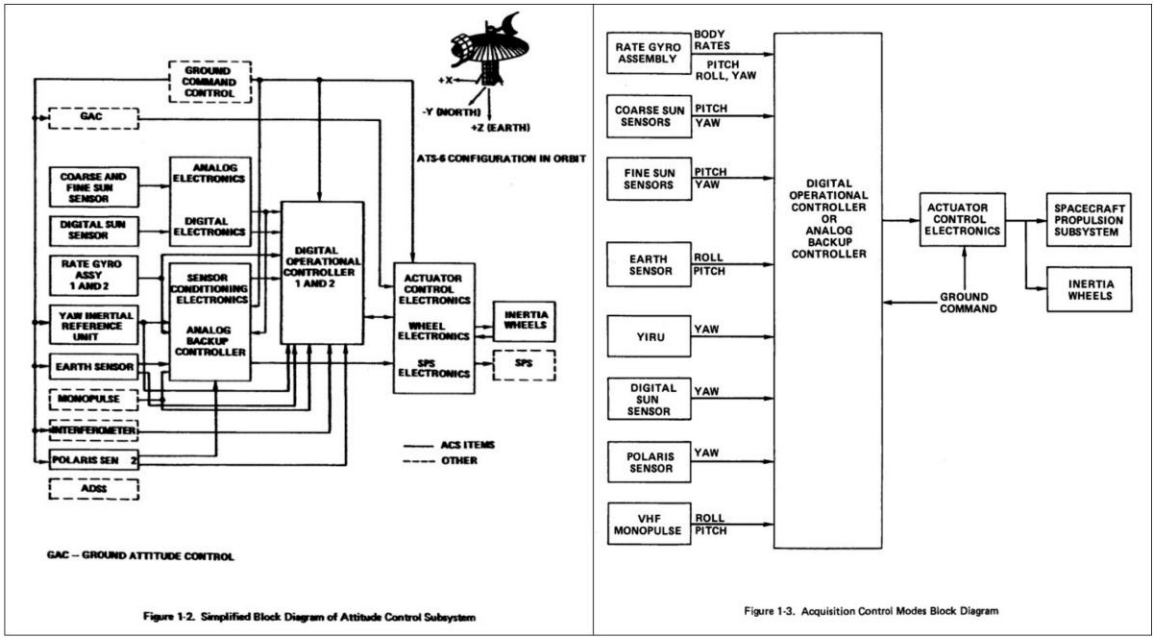


Figure 1.3 – ATS-6. left: GNC subsystem diagram. right: attitude control block diagram.

Three-axis stabilization quickly became the state-of-the-art attitude control strategy for NASA's satellites following its successful implementation on ATS-6. Nevertheless, most commercial satellite designs continued to use some version of spin stabilization in the three decades following the 1960s, using clever techniques to address the steady payload problem. For instance, in the 1980s, Hughes developed the HS 376, a dual-spin satellite bus design, in which the satellite rotates about its spin axis at some angular rate while another smaller section holding the payload, is rotated in the opposite direction at the same angular rate by a motor, effectively stabilizing the payload relative to an inertial reference frame. In competition with Hughes, RCA developed their Series 2000 through Series 5000 buses which used momentum-biasing, wherein the stabilization effect was achieved by a single spinning internal reaction wheel providing momentum bias while the external structure, on which the payload is mounted, remained stationary relative to an inertial reference frame [5]. This way, the satellite's overall momentum is being "stored" in the reaction wheel, and increasing or decreasing the wheel's angular rate applies torques along the spin axis. Thus, while the wheel can reorient the satellite along its spin axis, other actuators (like AOC thrusters) were still necessary for attitude control about axes with components perpendicular to the spin axis.

In the 1990s, demand for satellites that can accommodate higher-precision scientific missions and more powerful communications systems developed, leading to much broader industry adoption of full three-axis stabilization GNC architectures. Like NASA, commercially developed satellites began utilizing control actuators such as reaction wheels, control moment gyroscopes (CMGs), and magnetorquers to provide attitude control about all three principal axes of the satellite. Navigation was done using measurements from environment-referencing sensors like sun sensors, magnetometers, and star trackers, while inertial measurement units (IMUs) provided rate of change measurements that could be integrated and resolved into state estimates. Onboard digital computers allowed for the integration of closed-loop feedback control, which processed sensor data in real time to estimate states and adjust actuator commands to actively minimize errors between the desired states and the estimated states [7]. GOES-8 (1994), built by Space Systems

Loral (formerly Ford Aerospace and currently Lanteris Space Systems), was one of the earliest commercially developed satellites to use three-axis stabilization for on-orbit attitude control, necessitated by its mission objective to use continuous Earth monitoring to capture high accuracy meteorological data that allowed better long-range and short-range forecasting of severe weather events. The wheel-based three-axis stabilization technique was an evolution on single reaction wheel momentum-biasing: a system of three or more reaction wheels, whose spin axes have components in all three principal axis directions, store components of the overall momentum. The wheels can work together to apply torques in each principal axis direction independently, facilitating full three-axis control; however, the angular rates of the wheels cannot increase indefinitely or they will eventually saturate, so their "built-up" momentum needs to periodically be unloaded by actuators that can apply external torques, like AOC thrusters and magnetorquers. GOES-8 used a set of three reaction wheels and a combination of AOC thrusters and magnetorquers for momentum unloading in the roll and pitch directions, plus an additional solar sail and trim tab that rely on solar radiation pressure torques to supplement yaw direction unloading [8].

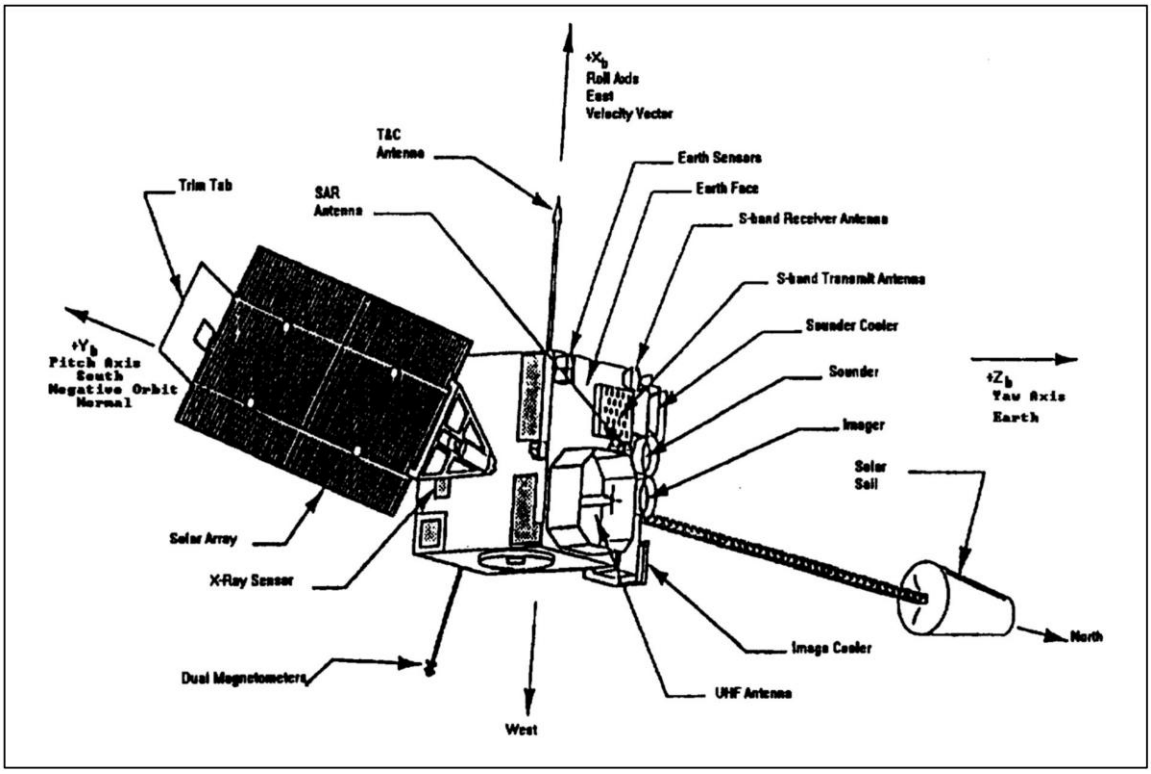


Figure 1.4 – GOES-8 structural diagram.

By the end of the century, most of the technologies that compose modern day satellite GNC subsystems were developed and successfully implemented on numerous commercial and scientific missions. The next frontier became miniaturization of these technologies to facilitate cost effectiveness and accessibility to space for a broader range of organizations, including universities. The CubeSat platform, developed in 1999 by Professors Bob Twiggs of Stanford University and Jordi Puig-Suari of Cal Poly, became the paradigm for low-cost, miniaturized satellites in the coming decades. The first CubeSats developed at these universities were launched in 2003, and

many more have been launched since then for commercial and scientific purposes. With falling launch costs and increasing interest in low Earth orbit (LEO) as a domain for satellite activity, small satellites like CubeSats are expected to be much more ubiquitous in the future [5].

The basic CubeSat form factor is a 10 cm x 10 cm x 10 cm cube called a 1U platform. As implied, this can be extended to the size of multiple units: a 2U CubeSat would be 10 cm x 10 cm x 20 cm and a 6U CubeSat might be 10 cm x 20 cm x 30 cm. Typically, most university CubeSats are up to 3U in size and weigh about 1.33 kg per U [9], which classifies them as nanosatellites. Due to form factor and weight limitations, CubeSat GNC subsystems generally need to be much smaller in size and complexity compared to traditional Earth orbiting satellites. However, the core concepts of GNC can be carried over for implementation on CubeSats. Like traditional satellites, attitude determination and control (ADCS) involves actuators, sensors, and an onboard computer with ADCS algorithms implemented in the flight software. The most common actuators include reaction wheels and magnetorquers while common sensors include IMUs, sun sensors, magnetometers, and even star trackers on some sophisticated designs. Unlike traditional satellites, however, CubeSats do not usually contain propulsion systems as they typically operate for short periods in low Earth orbit (LEO). They are also subject to constraints by the launch vehicle provider as they are usually included as “rideshare” satellites on rockets that contain a larger and more valuable primary payload; a rideshare CubeSat with a malfunctioning propulsion system could risk damaging the rocket’s performance and its primary payload. However, miniaturized chemical and electric propulsion systems have seen some limited usage on past experimental CubeSat missions. A notable example is NASA’s Starling (2023) mission, in which a swarm of four 6U CubeSats each used a self-contained cold gas propulsion system called Hamlet to perform small delta-v maneuvers and station-keeping [10].

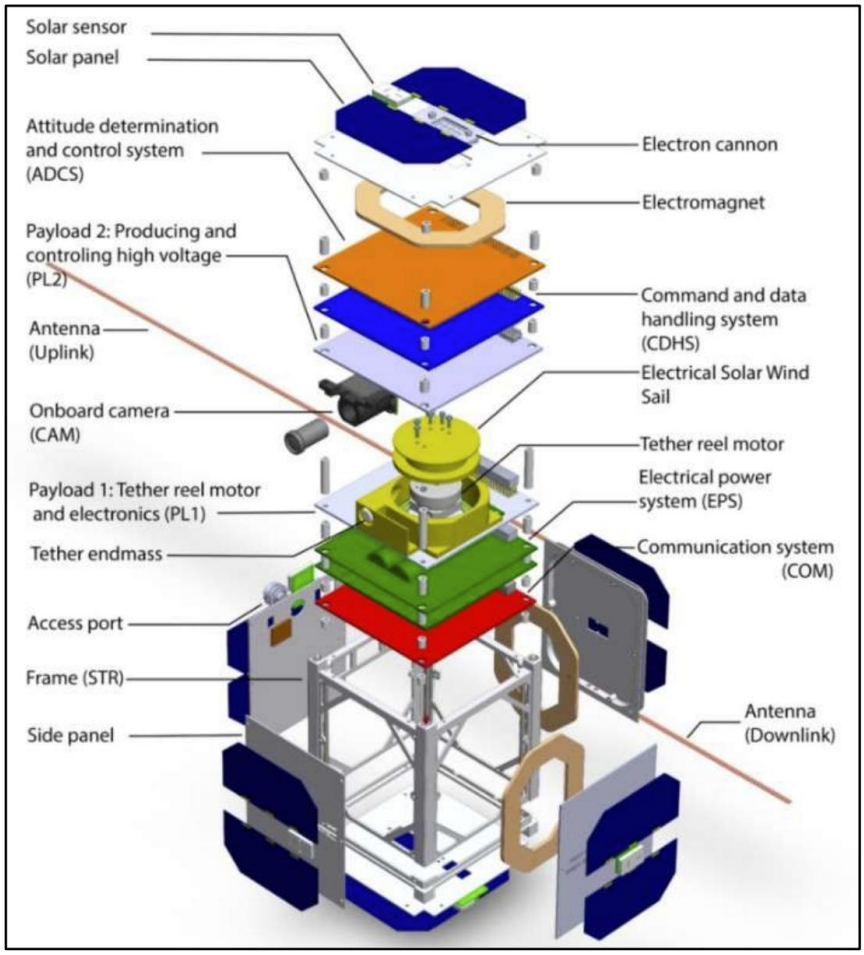


Figure 1.5 – Typical CubeSat design.

Mission objectives for many future small satellite missions are expected to have complex mission architectures, like swarms and formation flying, as well as requirements for greater levels of autonomy for individual satellites. This will necessarily require more sophisticated and reliable GNC architectures compared to past missions. Using software to model and simulate these architectures is a proven way to develop robust and workable GNC subsystems for these satellites.

Historically, simulations were impractical or unreliable for GNC development due to the lack of sufficiently high-performance computing hardware, and lack of software frameworks. Mainframe simulations of the Space Shuttle Orbiter, for instance, could only be run for 90 seconds at a time, and the Space Shuttle landing simulator was even done on an analog computer which did not always produce consistent results. The Space Shuttle's flight computer itself was based on an IBM 360, which was quite limited in storage and performance considering the complexity of the control system (multiple mission phases, phase-plane updates due to time-varying inertia matrices, etc.) [5]. The exponential growth in computing technologies since the Space Shuttle era enable issues like these to be easily overcome and allow for the design and development of far more sophisticated GNC architectures. A common software tool used today for developing and simulating GNC architectures is Matlab/Simulink. Developed by MathWorks, Matlab is a high-level programming language specialized in handling vector and matrix computations, while Simulink is a graphical language that allows control block diagram design and simulations.

Matlab/Simulink has become an industry standard for GNC development and is also used frequently in university settings. For example, University of Washington's Aeronautics and Astronautics CubeSat team developed their GNC simulation framework for HuskySat-1 using Matlab/Simulink [11]. If computational speed is of particular importance, like in the case of multi-satellite simulations, frameworks developed in low-level languages like C++ may be more useful. Basilisk is one such framework, developed by students and faculty at the University of Colorado, Boulder. Written in C/C++, it has become a popular open-source astrodynamics and GNC framework for research applications [12]. Another is Eric Stoneking's 42, a general-purpose open-source framework for satellite GNC design and verification, also written in C/C++ [13]. For validation of flight software derived from the GNC architecture, NASA's open-source Trick Simulation Toolkit, written in C/C++ and Java, provides a simulation environment that can run on Linux and MacOSX operating systems, provided that the flight software implementation specifically uses NASA's Core Flight Software (cFS) framework [14]. Additionally, several commercially available applications also exist for general-purpose astrodynamics and GNC simulations, such as Ansys STK and a.i. solutions FreeFlyer. Many of these software tools are robust and versatile, and have already been used reliably as GNC design and simulation frameworks on several prior space missions. However, their computationally optimized, end-user focused designs can often be a "black box," obscuring the core technical aspects of satellite GNC under several layers of software abstraction. A GNC engineer may desire to understand the inner workings of the "black box" to confirm its functionality and compatibility with mission requirements, while spending minimal effort interpreting and decrypting the software code. This may be especially true for organizations building CubeSats and small satellites for LEO, as they may choose to design custom GNC subsystems with specialized interfaces to reduce cost, weight, and complexity. To create a GNC simulation framework with a minimal "black box" for these types of applications would be both a challenging software development task, and a demonstration of proficiency in modern spacecraft GNC engineering.

1.3 Project Proposal

In this project, the objective will be to develop a modular, 6 DOF GNC software simulation framework for the purpose of providing SJSU with a tool to create modern satellite GNC architectures, which may be implemented onto future SJSU TechEdSat CubeSats and other LEO small satellite systems that SJSU might develop. Similar work done in the past at other universities and organizations will serve as valuable references. Emphasis will be placed on presentability and interpretability of code and minimizing the "black box" for prospective SJSU students and engineers.

1.4 Methodology

Due to familiarity with the software and its standardized use in the aerospace industry for GNC simulations, Matlab/Simulink will be used to implement the simulation framework. First, a Matlab-based programmatic interface for the user to define satellite properties and the GNC configuration will be developed. Next, the backend processes to build a Simulink model integrating all the components of the subsystem architecture (dynamics, environment, sensors, actuators, and algorithms) will be created. Next, a sample scenario with realistic design parameters and GNC objective(s) will be conceived. Natural dynamics will be simulated first to baseline the simulation builder followed by a simulation of the complete sample scenario. Simulation outputs will be created and the results will be presented and analyzed in this report.

Chapter 2 – Conceptual Overview

2.1 GNC Subsystem Architecture

The GNC subsystem is one of several subsystems found on a satellite system. Its design is guided by mission requirements and inputs from other spacecraft subsystems as shown in Figure 2.1 [15].

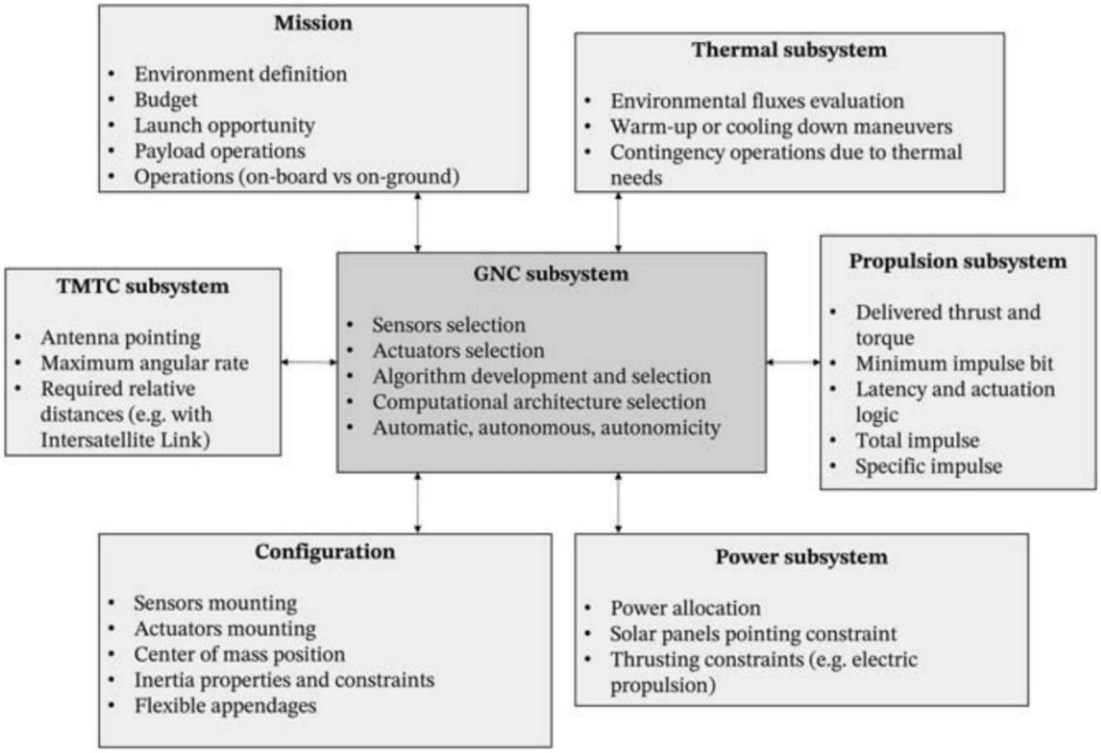


Figure 2.1 – Mission requirement inputs and subsystem inputs for GNC subsystem.

Modern satellite GNC subsystems are composed of both hardware and software components which work together in closed loop feedback. Hardware refers to the physical sensors and actuators assembled onto the spacecraft, while software refers to the guidance, navigation, and control algorithms implemented onto the spacecraft's on-board computer, or uplinked and downlinked via the ground segment. Figure 2.2 illustrates the conventional GNC subsystem architecture [15].

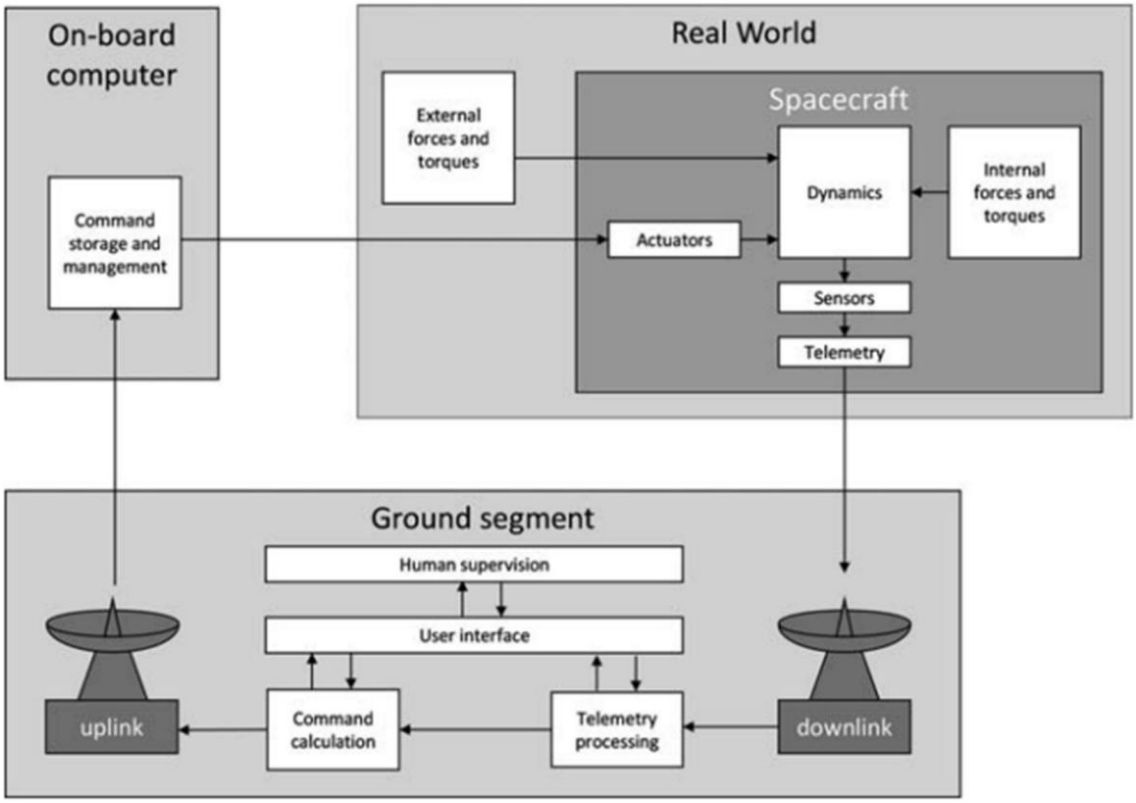


Figure 2.2 – Conventional space and ground segment GNC subsystem architecture.

Advanced satellite system designs which minimize ground segment interactions and maximize autonomy are becoming increasingly commonplace, especially in LEO. These systems incorporate nearly all the software GNC algorithms and the GNC mode management system as flight software on the on-board computer, as illustrated in Figure 2.3 [15].

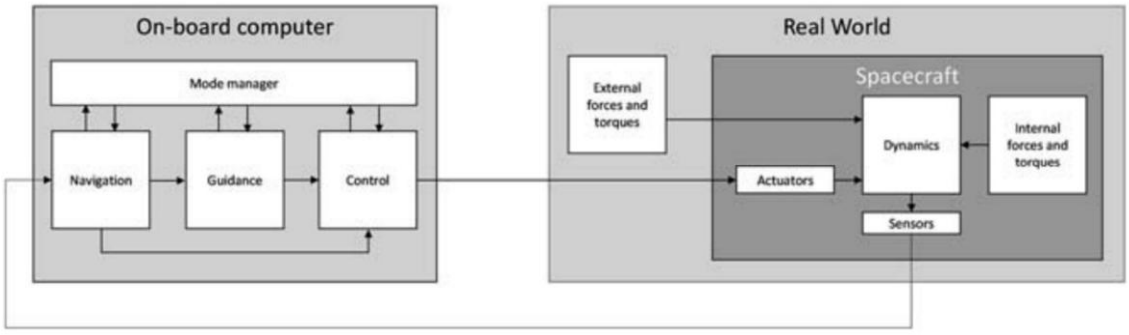


Figure 2.3 – Maximally autonomous space segment only GNC subsystem architecture.

This project will focus on modeling and simulation of the space segment of the GNC subsystem. The following sections in this chapter provide a brief overview of each major component that composes or influences the GNC subsystem architecture.

2.2 Spacecraft Dynamics

In its operational environment, a satellite exhibits dynamic motions which can be characterized as either translational or rotational motions about each of the three principal axes of a spatial coordinate system. Therefore, a satellite can generally be modeled as a 6 degree-of-freedom (DOF) rigid body system. However, real satellites are not true rigid bodies as they have flexible structural components which bend back and forth or may have propellants stored in liquid form which slosh around inside a fuel tank. For example, the turnstile whip antennas on Explorer 1 and structural appendages on GOES-8, such as the solar array, exhibit flexural dynamics. Many geostationary communications satellites also have large fuel tanks filled with liquid bipropellants that exhibit fuel sloshing dynamics. In some situations, these effects can be significant enough to warrant their own modeling and inclusion into the overall satellite dynamics model, as they may result in notable additions to, or coupling of, the translational and rotational dynamics. However, small satellites operating in LEO, especially CubeSats, generally avoid highly flexible components and liquid fuel storage in their designs to minimize complexity; therefore, for the purposes of this project, any dynamic additions from these effects can be neglected and the satellite can be modeled as a 6 DOF rigid body with decoupled translational and rotational motions.

2.2.1 Orbital Dynamics

A satellite's translational motion is described by its orbital dynamics. Spacecraft orbit models vary widely in complexity based on how many orbiting bodies and perturbing effects are deemed significant enough for consideration. For a satellite with a position $\vec{r}_{SC}^{\mathcal{N}}$ in an inertial reference frame, $\mathcal{N}$, the generalized approach for describing the translational dynamics can be presented as follows [7]:

$$\ddot{\vec{r}}_{SC}^{\mathcal{N}} = \vec{g}^{\mathcal{N}} + \frac{\vec{F}_{ext}^{\mathcal{N}}}{m_{SC}} \tag{2.1}$$

where $\vec{g}^{\mathcal{N}}$ is the net gravitational acceleration experienced by the satellite, m_{SC} is the satellite's total mass, and $\vec{F}_{ext}^{\mathcal{N}}$ is the sum of external forces acting on the satellite. $\vec{F}_{ext}^{\mathcal{N}}$ can include net thrust forces applied by the satellite's propulsion system, as well as non-gravitational disturbance forces from the space environment. Equation 2.1 effectively isolates conservative forces due to the gravitational field, and non-conservative external forces that add and remove energy to the satellite as it moves along its trajectory. While Equation 2.1 applies for general gravitational fields and trajectories, in the special case of Earth-orbiting satellites, the restricted two-body assumption ($m_{SC} \ll m_{Earth}$) can be applied and gravitational acceleration can be modeled as the gravity in a uniform spherical gravitational field with the addition of gravitational perturbation effects. For a satellite with a position $\vec{r}_{SC}^{\mathcal{N}}$ in the Earth-Centered Inertial (ECI) reference frame, $\mathcal{N}$, the gravitational acceleration is:

$$\vec{g}^{\mathcal{N}} = -\frac{\mu}{\left\|\vec{r}_{SC}^{\mathcal{N}}\right\|^3}\vec{r}_{SC}^{\mathcal{N}} + \frac{\vec{F}_{pertb}^{\mathcal{N}}}{m_{SC}} \tag{2.2}$$

where μ is the gravitational parameter of the central body (Earth) and $\vec{F}_{pertb}^{\mathcal{N}}$ is the sum of the gravitational perturbation forces [15]. Perturbations are typically modeled by approximating Earth's non-spherical mass distribution and gravity potential using Legendre Polynomials and

spherical harmonic functions. Taking the gradient of the gravity potential function will yield Equation 2.2, where $\vec{F}^{\mathcal{N}}_{pertb}$ will include terms that characterize the influence of zonal harmonics (latitude-dependent harmonic gravity variations), sectoral harmonics (longitude-dependent harmonic gravity variations), and tesseral harmonics (latitude/longitude-dependent harmonic gravity variations) [7].

Satellites orbiting Earth have elliptical orbits, so the geometry of the orbit itself at a given instant, called the osculating orbit, can be parametrized by the set of six classical (Keplerian) orbital elements:

- a = semi-major axis
- e = eccentricity ($0 \leq e \leq 1$)
- t = perigee epoch
- Ω = right ascension of the ascending node
- i = inclination
- ω = argument of perigee

Figure 2.4 illustrates the physical meaning of all these parameters. In the diagram, the $\mathcal{N}$ frame is represented by the $\{X, Y, Z\}$ coordinate system. The satellite's angular position relative to the perigee is represented by the true anomaly, θ.

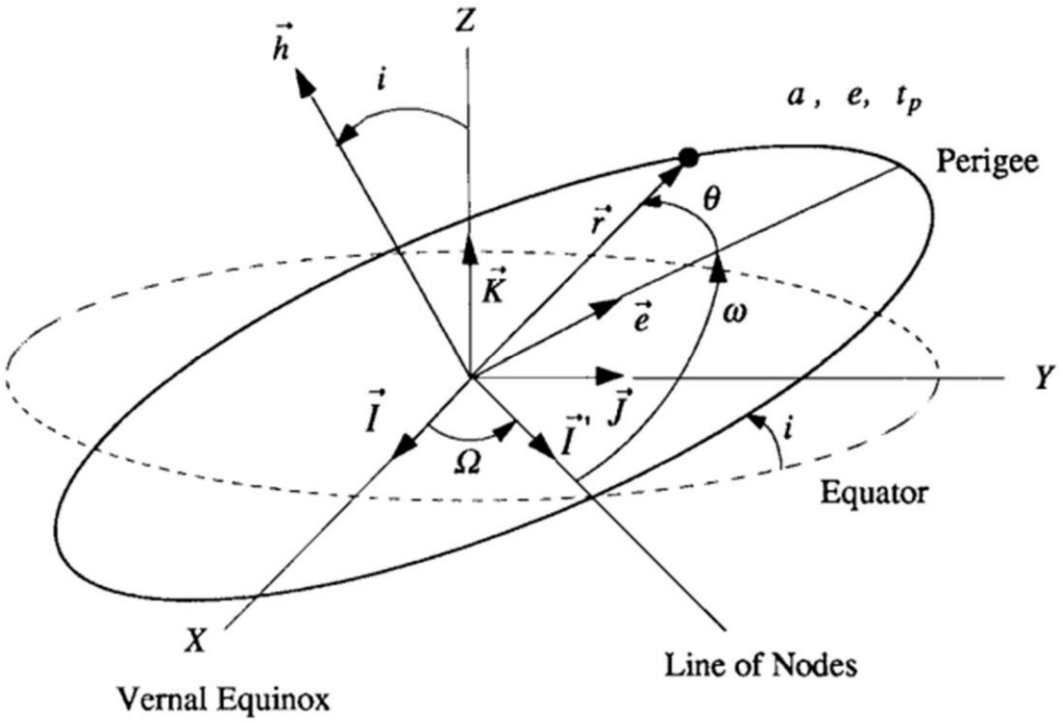


Figure 2.4 – Satellite orbit parametrization using Keplerian elements.

2.2.2 Attitude Dynamics

A satellite's rotational motion is described by its attitude dynamics. Modeling attitude dynamics involves considering the angular momentum of the satellite, external torques applied on

the satellite, and any components inside the satellite which "store" angular momentum. For example, the reaction wheels on ATS-6 and GOES-8, and the control-moment gyroscopes (CMGs) on the International Space Station are sources of internal "stored" momentum. The satellite can be modeled as a generalized rigid body with internal momentum sources undergoing rotational motion about its center of mass (COM) some distance away from the central body (Earth), as shown in Figure 2.5.

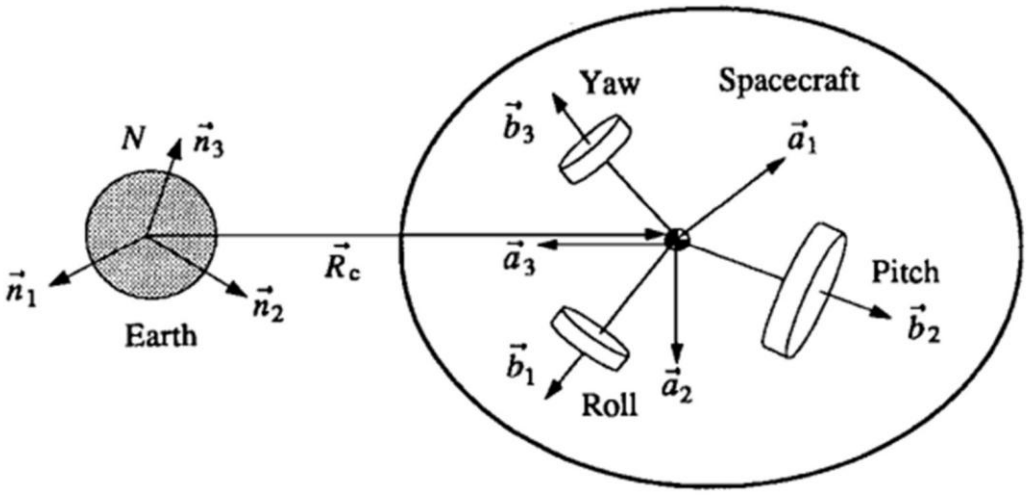


Figure 2.5 – Satellite undergoing rigid body rotational motion about its COM.

where $\mathcal{N}$ is the ECI frame with unit basis vectors $\{\vec{n}_1, \vec{n}_2, \vec{n}_3\}$, and $\mathcal{B}$ is the satellite body frame with unit basis vectors $\{\vec{b}_1, \vec{b}_2, \vec{b}_3\}$ and origin at the COM. The satellite's angular momentum in the $\mathcal{N}$ frame, $\vec{H}_{SC}^{\mathcal{N}} \equiv J_{SC}\vec{\omega}_{SC}^{\mathcal{N}}$, is subject to dynamics given by the following EOM [7]:

$$\dot{\vec{H}}_{SC}^{\mathcal{N}} = \dot{\vec{H}}_{SC}^{\mathcal{B}} + \dot{\vec{H}}_{int}^{\mathcal{B}} + \vec{\omega}_{SC}^{\mathcal{B}} \times \left(\vec{H}_{SC}^{\mathcal{B}} + \vec{H}_{int}^{\mathcal{B}}\right) = \vec{M}_{ext}^{\mathcal{B}} \tag{2.3}$$

where $\vec{H}_{SC}^{\mathcal{B}} \equiv J_{SC}\vec{\omega}_{SC}^{\mathcal{B}}$ is the satellite's angular momentum in the $\mathcal{B}$ frame, $\vec{H}_{int}^{\mathcal{B}} \equiv J_{int}\vec{\omega}_{int}^{\mathcal{B}}$ is the sum of all "stored" angular momentums from the internal momentum sources, and $\vec{M}_{ext}^{B}$ is the sum of the external moments acting on the satellite. External moments can include net torques applied by the satellite's propulsion system, as well as disturbance torques from the space environment. Equation 2.3 can become highly coupled and nonlinear depending on the level of generality used in modeling. Therefore, the following assumptions and simplifications were applied for this project:

- **Assumption 1:** Satellite mass properties (mass, inertia tensor, and COM) are constants.
- **Assumption 2:** Satellite $\mathcal{B}$ frame origin is located at the COM.

The actual attitude, or orientation, of the satellite can be parametrized in several ways, including Euler angles, direction cosine matrices (DCMs), quaternions, Rodriguez parameters, etc. However, some of these representations are prone to kinematic differential equation singularities or are computationally inefficient; therefore, quaternions, which avoid these pitfalls, are often the

parameterization of choice. The satellite's attitude quaternion, $\vec{q}_{SC} \equiv \left[q_{SC_1}, q_{SC_2}, q_{SC_3}, q_{SC_4}\right]^T \equiv \left[q_{SC_1}, \left[\vec{q}_{SC_v}\right]^T\right]^T$, has the following kinematic differential equation [7]:

$$\dot{\vec{q}}_{SC_v} = \frac{1}{2}\left(q_{SC_1}\vec{\omega}_{SC}^{\mathcal{B}} - \vec{\omega}_{SC}^{\mathcal{B}} \times \vec{q}_{SC_v}\right) \tag{2.4}$$

$$\dot{q}_{SC_1} = -\frac{1}{2}\left[\vec{\omega}_{SC}^{\mathcal{B}}\right]^T \vec{q}_{SC_v} \tag{2.5}$$

where $\vec{q}_{SC_v}$ is the vector component of $\vec{q}_{SC}$, q_{SC_1} is the scalar component of $\vec{q}_{SC}$, and $[\vec{\cdot}]^T$ means the transpose of a column vector $\vec{\cdot}$.

An important property of quaternions is the Hamilton product. For two quaternions $\vec{q}_A$ and $\vec{q}_B$, a Hamilton product is defined as:

$$\vec{q}_B\vec{q}_A = \left[q_{B_1}q_{A_1} - \vec{q}_{B_v} \cdot \vec{q}_{A_v}, \left[q_{B_1}\vec{q}_{A_v} + q_{A_1}\vec{q}_{B_v} + \vec{q}_{B_v} \times \vec{q}_{A_v}\right]^T\right]^T \tag{2.6}$$

Physically, the Hamilton product represents consecutive reorientations by quaternions; $\vec{q}_B\vec{q}_A$ represents a reorientation from the base reference frame to $\vec{q}_A$ (attitude is relative to the base frame), followed by a reorientation from this intermediary reference frame to $\vec{q}_B$ (attitude is relative to the intermediary frame) [7]. The Hamilton product can also be used to rotate a vector, $\vec{v}$, by a quaternion, $\vec{q}$, as follows:

$$[0, [\vec{q} \otimes \vec{v}]^T]^T = \vec{q}[0, [\vec{v}]^T]^T\vec{q}^{-1} \tag{2.7}$$

where $\vec{q} \otimes \vec{v}$ is the rotated vector, and $\vec{q}^{-1} \equiv [q_1, -[\vec{q}_v]^T]^T$ is the quaternion conjugate [7]. A cleaner way to write Equation 2.7 is as follows:

$$\vec{q} \otimes \vec{v} = \vec{v} + 2\left(q_1(\vec{q}_v \times \vec{v}) + \vec{q}_v \times (\vec{q}_v \times \vec{v})\right) \tag{2.8}$$

2.3 Sensors

GNC sensors are hardware components used to make measurements of the satellite's internal dynamic states, or the surrounding space environment. These measurements can be used by state estimation algorithms implemented on the on-board computer to estimate the satellite's dynamic states. Typically, satellites use sensors that can be classified into one of the following categories [15]:

- Attitude sensors
- Inertial sensors
- Orbit sensors

Other less commonly used sensor types include electro-optical sensors, such as cameras and LIDAR, as well as altimeters. To model these sensors within a GNC simulation framework, mathematical representations of their functionalities must be developed. In the most abstract sense, a sensor measurement can be described by the following equation:

$$\tilde{\vec{x}}(t) = \vec{x}(t) + \vec{\epsilon}(t) \tag{2.9}$$

where $\breve{\vec{x}}(t)$ is the vector of measurements at time t, $\vec{x}(t)$ is the real-world truth/physical signal, and $\vec{\epsilon}(t)$ is the sum of all process errors and biases. The block diagram in Figure 2.6 shows the various types of errors and their place in the measurement acquisition sequence for a generic sensor model [15].

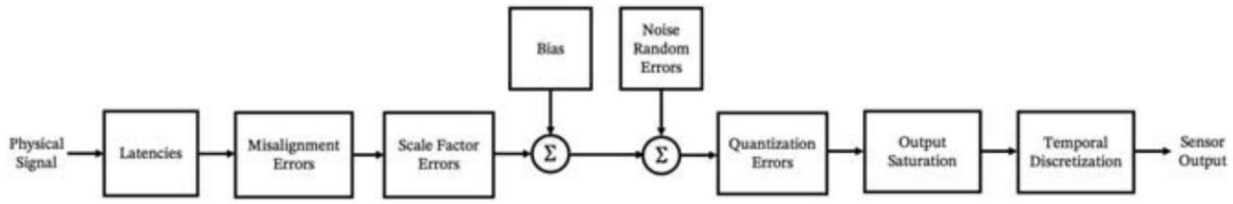


Figure 2.6 – Generic sensor model block diagram.

2.3.1 Attitude Sensors

Attitude sensors are used to make measurements of space environment references, which can be resolved into deterministic attitude estimates using attitude estimation algorithms. The four attitude sensors most common in small satellite designs are briefly described in the subsections below.

2.3.1.1 Magnetometers

Magnetometers measure the direction of the Earth's magnetic field vector with respect to the sensor frame of reference. Larger satellites can use nickel-iron alloy magnetoresistive sensors, with coarse accuracies of 1° or more in each spatial direction and resolutions of 10 to 50 nT, or they can use fluxgate magnetometers with finer accuracies of 0.1° in each direction and resolutions of 0.1 to 10 nT. While small satellites may also use these types of magnetometers, sizing and cost considerations usually encourage the use of microelectromechanical systems (MEMS) magnetometers, which operate on the principle of measuring Lorentz force effects. MEMS magnetometers are miniature, power efficient sensors with accuracies around 1° [15].

A magnetometer direction vector measurement cannot itself be resolved into a deterministic attitude; it must first be combined with satellite position estimates and an Earth magnetic field model (a function of the satellite position) to indirectly produce an attitude estimate. Due to this limitation, magnetometers are generally utilized only for coarse attitude determination [15]. For implementation in a GNC simulation framework, a magnetometer sensor model following from Equation 2.9 can be written as:

$$\breve{\vec{B}} = \vec{q}_{MTM} \otimes \left(\vec{q}_{SC} \otimes \vec{B}\left(\vec{r}_{SC}^{\mathcal{N}}\right)\right) + \vec{\epsilon}_{MTM} \tag{2.10}$$

where $\breve{\vec{B}}$ is the sensor frame magnetometer measurement output, $\vec{B}\left(\vec{r}_{SC}^{\mathcal{N}}\right)$ is the inertial frame Earth magnetic field vector modeled as a function of the satellite's orbital position $\vec{r}_{SC}^{\mathcal{N}}$, $\vec{q}_{SC}$ is the satellite's attitude quaternion, $\vec{q}_{MTM}$ represents the magnetometer's sensor frame orientation relative to the satellite's body frame, and $\vec{\epsilon}_{MTM}$ is any modeled sources of errors and biases in the sensor frame. The $\otimes$ symbol in Equation 2.10 represents rotation of a vector by a quaternion, as described in Equation 2.8.

2.3.1.2 Sun Sensors

Sun sensors measure the direction of the sun with respect to the sensor frame of reference. They can be categorized as analog or digital based on their principle of operation. Analog sun sensor outputs are a continuous function of the sun incidence angle. The most basic analog sun sensor is a course sun sensor (CSS), which operates based on Lambert's cosine law, where the intensity of the sunlight striking a surface is proportional to the cosine of the light ray incident angle with the surface normal vector:

$$J = J_s \cos \alpha \tag{2.11}$$

where J is the light intensity at the sensor surface, J_s is the light intensity at $\alpha = 0°$, and α is the light ray incident angle as depicted in Figure 2.7 [15].

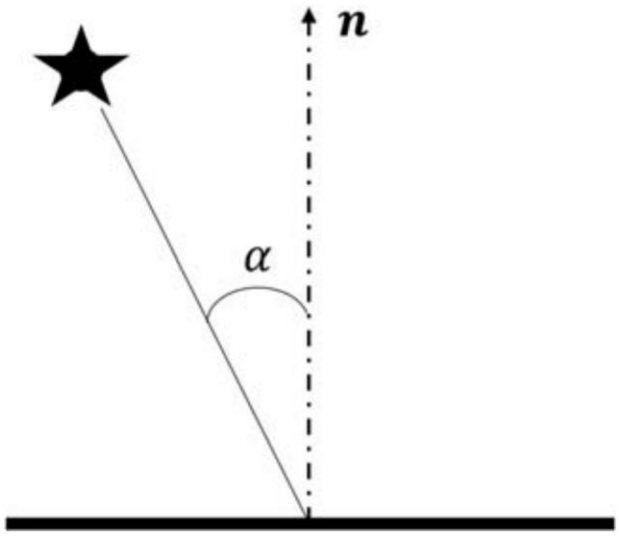


Figure 2.7 – Single axis coarse sun sensor diagram.

Measurements of two perpendicular incident angles can be resolved into a sun direction vector. CSS hardware can simply be composed of a single photosensitive detector surface; the satellite's solar arrays may even be used as a CSS. Accuracies vary for CSS based on the exact hardware used but are generally 1° or more. Fine sun sensors (FSS) are another type of analog sun sensor. Unlike CSS, they are composed of two or more photosensitive detector surfaces placed in an enclosure with a window and lens above the detector surfaces. Sunlight enters through the window and the rays are collimated by the lens to form an apparent image on the detector surfaces as shown in Figure 2.8 [15].

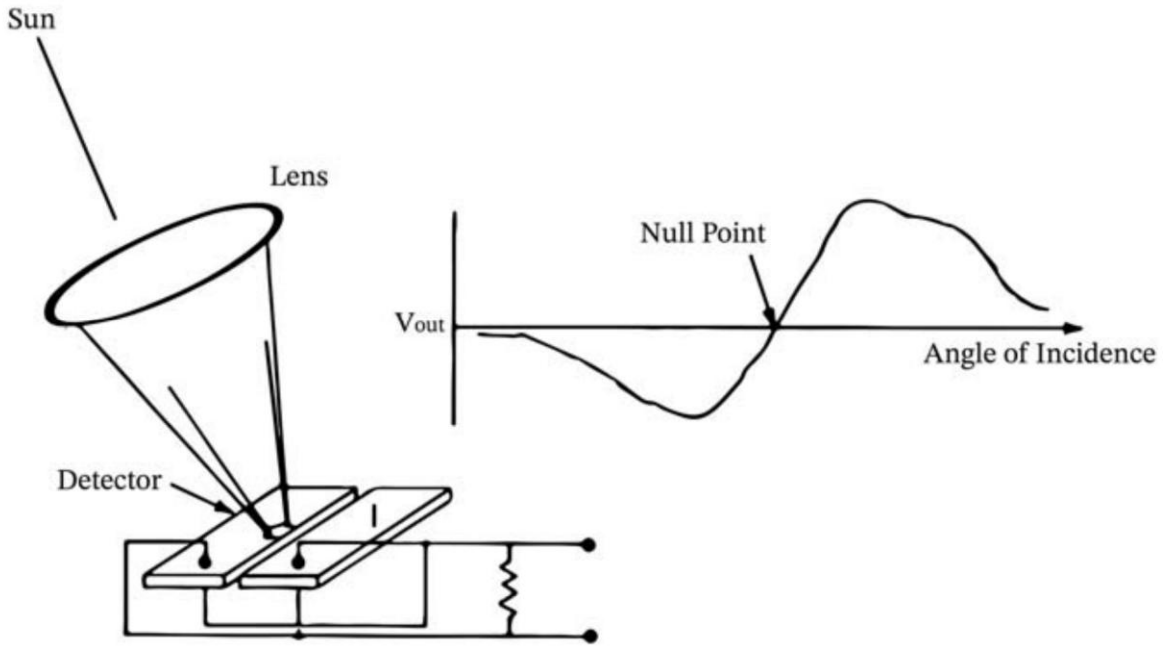


Figure 2.8 – Fine sun sensor diagram.

Voltage differences between the detector surfaces can be resolved into angle of incidence, and subsequently a direction vector. When the sun is directly above the sensor, the voltage differences will be null. FSS typically have accuracies of <0.01°.

Digital sun sensors (DSS) use a combination of binary masking and voltage thresholds to determine angle of incidence. Sunlight enters through a slit aperture and is filtered through a binary mask, as shown in Figure 2.9 [15].

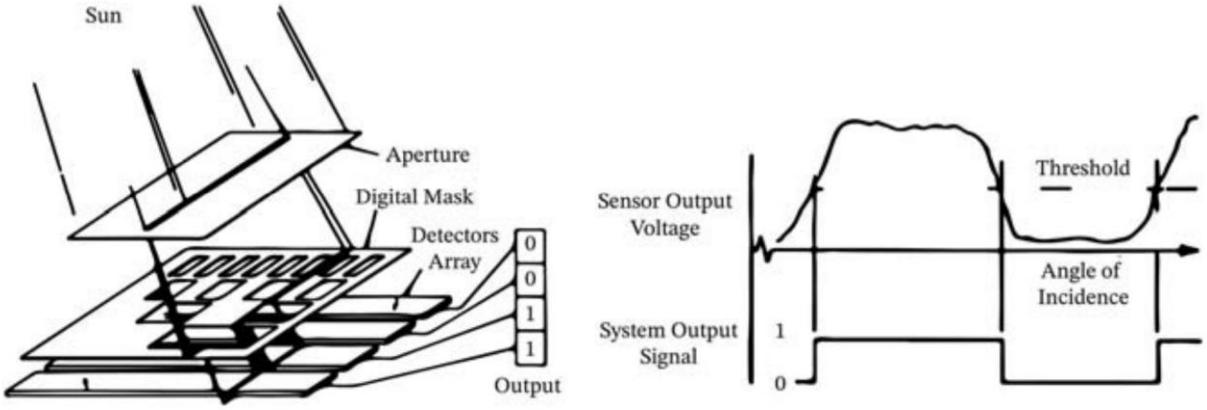


Figure 2.9 – Digital sun sensor diagram.

The unique binary pattern at the location of the line image modifies the pattern before it strikes the array of detectors. If a voltage threshold is exceeded on the illuminated detectors, the output signal will flip from 0 to 1 for those detectors. These output signals can be converted to angle of incidence, and subsequently a direction vector. DSS have accuracies on par with FSS at <0.01°.

Sun sensors are fairly accurate and inexpensive and are useful for moderately precise pointing objectives. However, they have field of view limitations and are unusable during orbital eclipse periods. Albedo effects from Earth reflecting sunlight may also be a source of interference that needs to be accounted for to obtain accurate measurements [15]. For implementation in a GNC simulation framework, a sun sensor model following from Equation 2.9 can be written as:

$$\check{\vec{s}} = \vec{q}_{SS} \otimes (\vec{q}_{SC} \otimes \vec{s}) + \vec{\epsilon}_{SS} \tag{2.12}$$

where $\check{\vec{s}}$ is the sensor frame sun vector measurement output, $\vec{s}$ is the inertial frame sun vector model, $\vec{q}_{SC}$ is the satellite's attitude quaternion, $\vec{q}_{SS}$ represents the sun sensor's frame orientation relative to the satellite's body frame, and $\vec{\epsilon}_{SS}$ is any modeled sources of errors and biases in the sensor frame. $\vec{\epsilon}_{SS}$ may depend on whether the sun sensor is a CSS, FSS, or a DSS. The $\otimes$ symbol in Equation 2.12 represents rotation of a vector by a quaternion, as described in Equation 2.8.

2.3.1.3 Earth Sensors

Earth sensors detect infrared (IR) radiation in the 14 to 16 μm range emitted by the Earth's atmosphere. They come in two forms: static and scanning. Static Earth sensors have no moving parts. They are composed of an IR optical system, which captures the Earth image and applies an IR filter, and an image plane, which receives the IR image and processes it using radiance detectors [15].

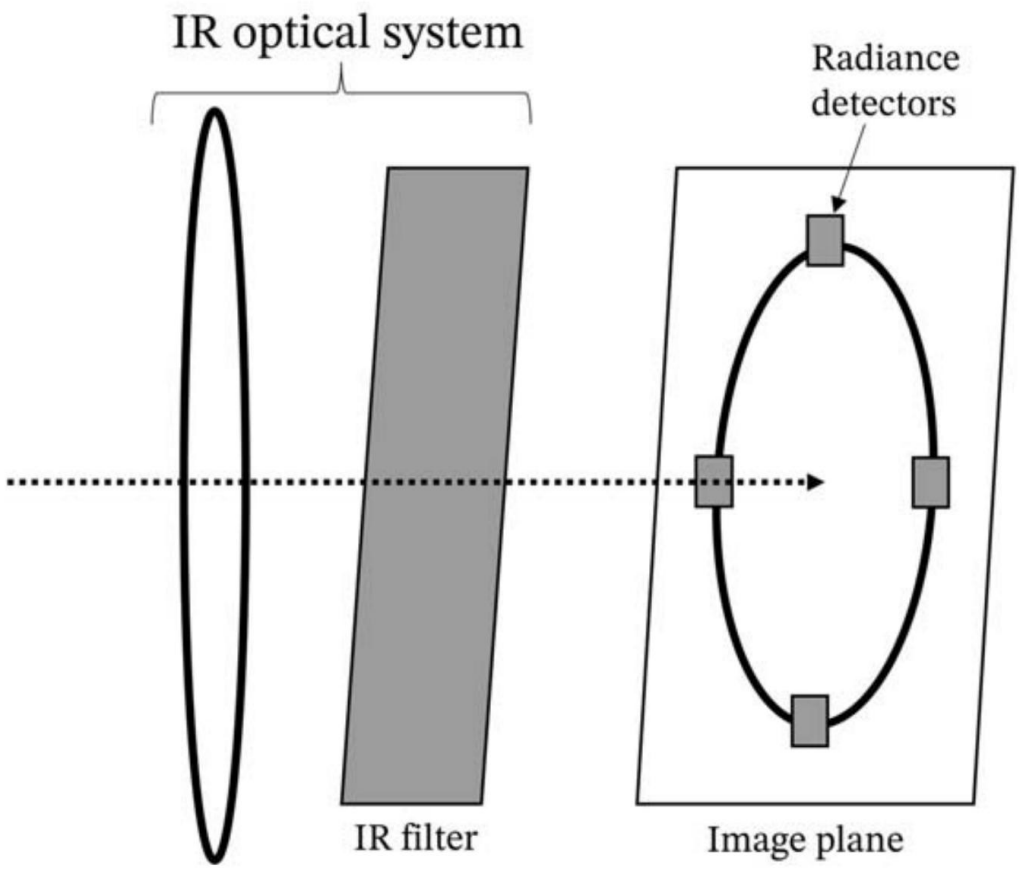


Figure 2.10 – Static Earth sensor diagram.

The radiance detectors each have an array of photosensitive cells called a thermopile array which detect where the Earth horizon outline is relative to the centroid of the sensor, as shown in Figure 2.11 [15].

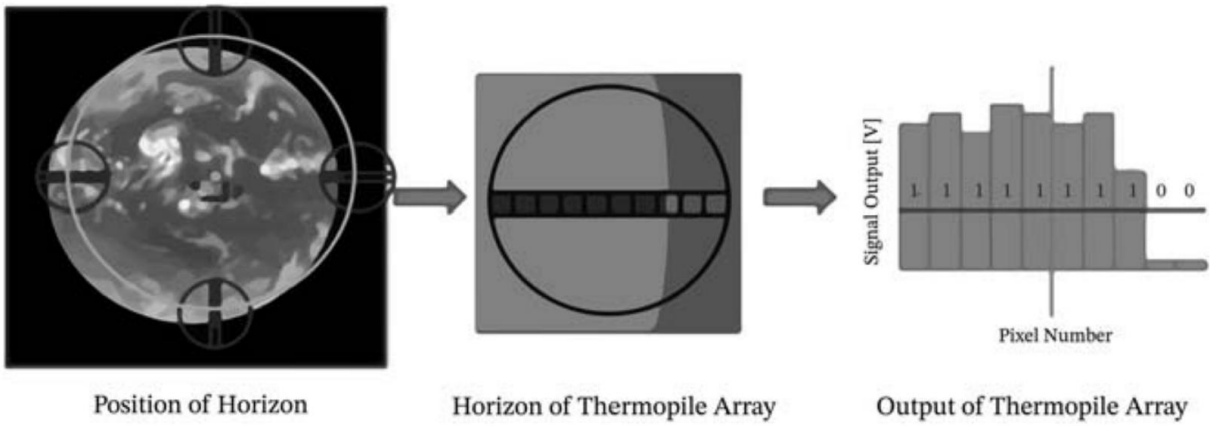


Figure 2.11 – Static Earth sensor thermopile operation.

If the sensor's centroid coincides with the IR Earth image capture, the satellite is nadir (Earth) pointed through the sensor centroid. Therefore, the Earth sensor can be used to detect the angular

offset (roll and pitch) of the satellite's pointing vector relative to the nadir vector. How it is rotated about the pointing vector (yaw) cannot be provided by an Earth sensor however, other sensors and algorithms need to be used to fully determine the attitude. Roll and pitch measurement accuracies are typically between 0.1° and 0.5° [15].

Unlike static Earth sensors, scanning Earth sensors perform rotating scans of the part of the Earth horizon in their field of view. These rotating scans can be done mechanically, with a physically rotating IR detector, or electronically on a fixed detector with a scan pattern. Roll and pitch can be determined using two perpendicular scanning Earth sensors. Due to their weight and complexity, scanning Earth sensors are typically not used on small satellites.

Earth sensors offer relatively high accuracy and are operational during eclipses. However, they require a constant view of the Earth and do not provide sufficient information on their own for complete attitude determination [15]. For implementation in a GNC simulation framework, an Earth sensor model following from Equation 2.9 can be written as:

$$\breve{\vec{n}} = \vec{q}_{ES} \otimes \left(\vec{q}_{SC} \otimes \vec{n}\left(\vec{r}_{SC}^{\mathcal{N}}\right)\right) + \vec{\epsilon}_{ES} \tag{2.13}$$

where $\breve{\vec{n}}$ is the sensor frame nadir vector measurement output, $\vec{n}\left(\vec{r}_{SC}^{\mathcal{N}}\right)$ is the inertial frame nadir vector modeled as a function of the satellite's orbital position $\vec{r}_{SC}^{\mathcal{N}}$, $\vec{q}_{SC}$ is the satellite's attitude quaternion, $\vec{q}_{ES}$ represents the Earth sensor's frame orientation relative to the satellite's body frame, and $\vec{\epsilon}_{ES}$ is any modeled sources of errors and biases in the sensor frame. The $\otimes$ symbol in Equation 2.13 represents rotation of a vector by a quaternion, as described in Equation 2.8.

2.3.1.4 Star Trackers

Star trackers are advanced optical sensors that capture and process images of stars in the celestial sphere. The sensor compares the captured image with a database, called a star catalog, that is integrated into the sensor hardware. The relative location and orientation of captured stars are compared to their match in the star catalog to fully determine attitude in the sensor frame of reference. Figure 2.12 shows the hardware layout of a typical star tracker [15].

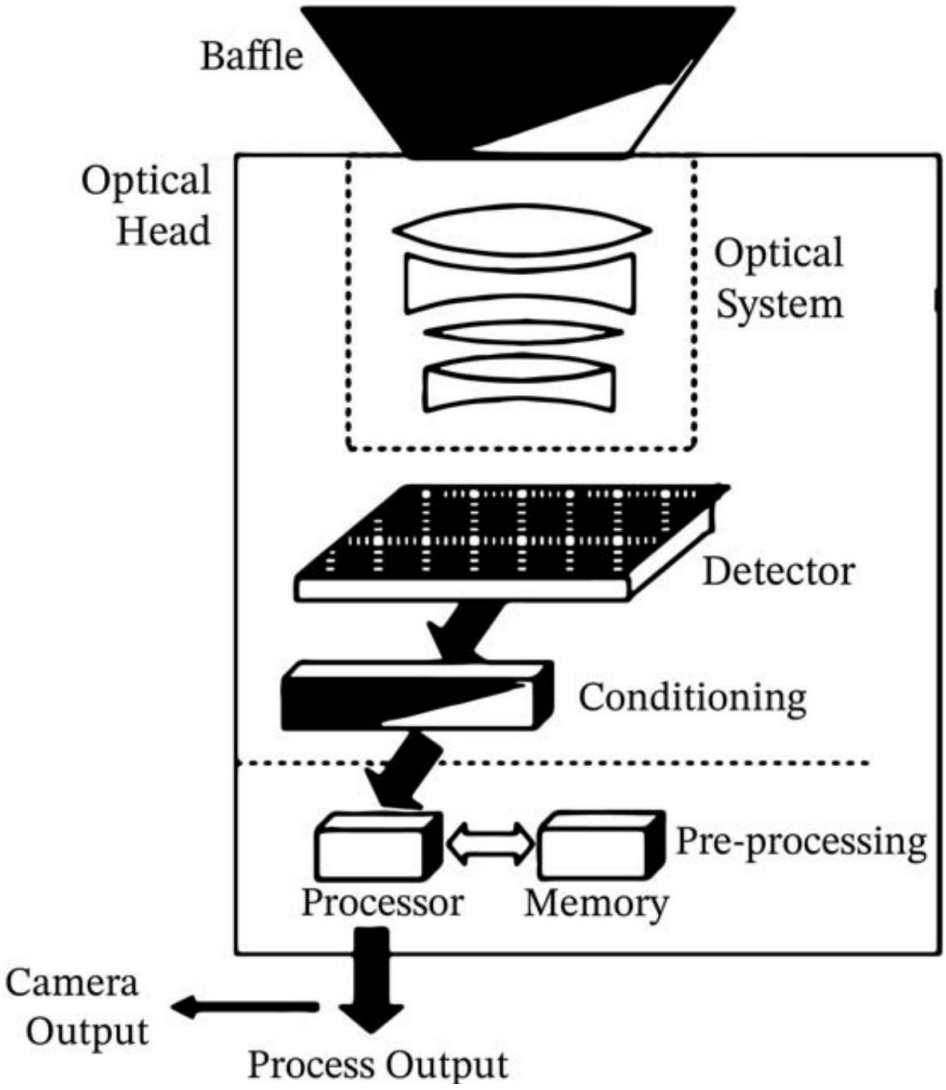


Figure 2.12 – Star tracker hardware diagram.

The baffle is used to mitigate interference from stray light sources such as the sun and Earth albedo. The optical system captures photons in the field of view and the detector system converts the capture into a pixel image to be processed by the integrated sensor software. At the integrated software level, star trackers have the following capabilities:

- Cartography: star positions are detected in the sensor frame of reference.
- Autonomous attitude determination: centroiding algorithms are applied to acquired clusters of stars, brightness and arc length values are computed, and pattern searching algorithms are applied to match with stars in the catalog.
- Star tracking: the initial attitude acquired using the autonomous attitude determination algorithm is propagated by tracking the apparent change in position of a select set of stars.

An expression for approximating the accuracy of a star tracker is given by:

$$accuracy = \frac{FOV}{n_{px}\sqrt{N}} \tag{2.14}$$

where FOV is the field of view, n_{px} is the number of pixels composing the detector, and N is the number of detected stars in the FOV [15]. Star trackers can achieve accuracies on the order of arcseconds (~0.000278°), which is generally much finer than what other types of attitude sensors can achieve. Additionally, they can compute a valid initial attitude solution independently (the "lost-in-space" problem) without the need for external processing of sensor data like attitude determination algorithms. However, they have higher mass, volume, and power requirements compared to other sensors and are very sensitive to light interference from luminous objects like the sun. Additionally, star trackers require considerable time to solve the lost-in-space problem; they may be unable to converge on an attitude solution if the satellite's angular rates are too high [15]. For implementation in a GNC simulation framework, a star tracker model following from Equation 2.9 can be written as:

$$\breve{\vec{q}} = \vec{q}_{\epsilon_{ST}}\vec{q}_{ST}\vec{q}_{SC} \tag{2.15}$$

where $\breve{\vec{q}}$ is the sensor frame attitude quaternion measurement output at time t, $\vec{q}_{SC}$ is the satellite's attitude quaternion, $\vec{q}_{ST}$ represents the orientation of the star tracker sensor frame relative to the satellite's body frame, and $\vec{q}_{\epsilon_{ST}}$ is the error quaternion resulting from any modeled sources of attitude errors and biases in the sensor frame. The quaternion products in Equation 2.15 are Hamilton products as described by Equation 2.6.

2.3.2 Inertial Sensors

Inertial sensors are used to measure accelerations and angular rates of the satellite. In hardware form, their functionality is encapsulated into a single device called an inertial measurement unit (IMU). The functional block diagram for an IMU is shown below [15].

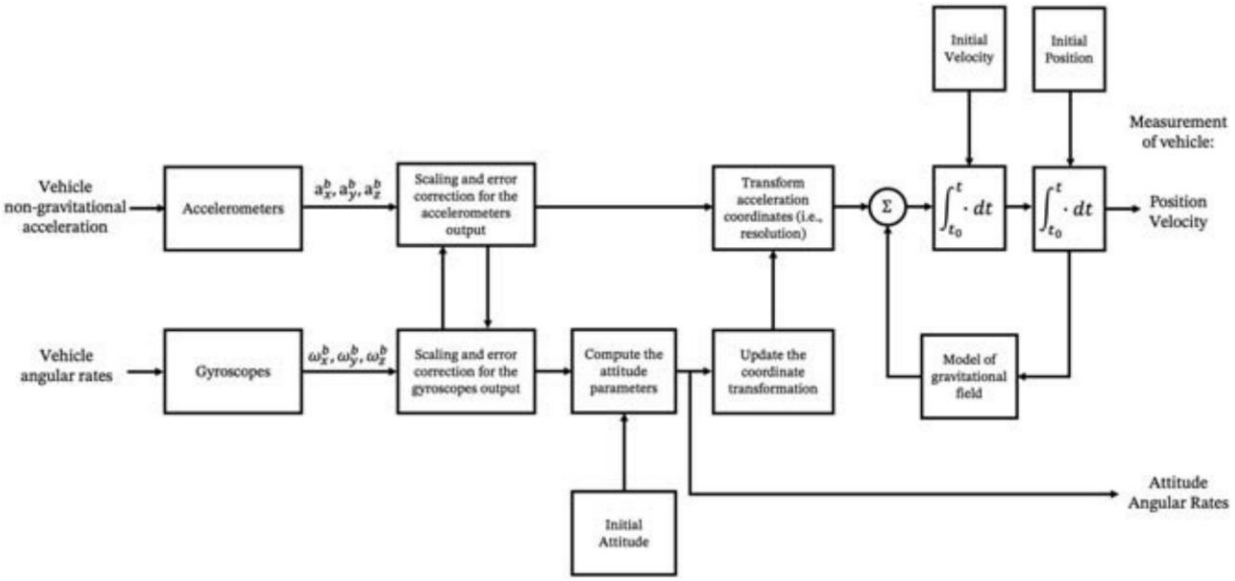


Figure 2.13 – Generic inertial sensor model block diagram.

Accelerometers and gyroscopes have a variety of different error sources that must be considered for sensor model development. The following common error sources are described in detail in [15]:

- Bias errors
  - Constant bias
  - Bias repeatability
  - Temperature-dependent bias
  - Bias instability
- Random walk errors
  - Velocity random walk (accelerometer)
  - Angle random walk (gyroscope)
  - Rate random walk
- Scale factor error
- Non-orthogonality (cross-axis sensitivity) error
- Misalignment error

Other characteristics that require modeling include input range limits, output quantization, and sampling rate [15].

2.3.2.1 Accelerometers

Accelerometers measure the total translational acceleration experienced by the satellite in each spatial direction. They are typically used for measuring actuator-applied forces on the satellite. Accelerometers for small satellites are usually MEMS sensors integrated into an IMU which measure pendulum displacements of micro-lithographic integrated circuit components. These displacements are effects of the apparent forces experienced by the accelerometer, which the accelerometer resolves into acceleration measurements in its frame of reference. For implementation in a GNC simulation framework, an accelerometer model following from Equation 2.9 can be written as:

$$\breve{\vec{a}} = \vec{q}_{ACC} \otimes \left(\vec{q}_{SC} \otimes \ddot{\vec{r}}^{\mathcal{N}}_{ACC}\right) + \vec{\epsilon}_{ACC} \tag{2.16}$$

where $\ddot{\vec{r}}^{\mathcal{N}}_{ACC}$ is the total apparent acceleration experienced by the accelerometer at its mounting location expressed in inertial frame coordinates. $\ddot{\vec{r}}^{\mathcal{N}}_{ACC}$ can be written in terms of the apparent acceleration of the satellite plus lever arm effects due to the accelerometer's mounting location being offset from the satellite's COM. This results in:

$$\breve{\vec{a}} = \vec{q}_{ACC} \otimes \left(\vec{q}_{SC} \otimes \left(\ddot{\vec{r}}^{\mathcal{B}}_{SC} - \vec{g}^{\mathcal{B}} + \dot{\vec{\omega}}^{\mathcal{B}}_{SC} \times \vec{r}^{\mathcal{B}}_{ACC/COM} + \vec{\omega}^{\mathcal{B}}_{SC} \times \left(\vec{\omega}^{\mathcal{B}}_{SC} \times \vec{r}^{\mathcal{B}}_{ACC/COM}\right)\right)\right) + \vec{\epsilon}_{ACC} \tag{2.17}$$

In Equation 2.17, $\breve{\vec{a}}$ is the net measured acceleration in the sensor frame, $\ddot{\vec{r}}^{\mathcal{B}}_{SC}$ is the satellite's total acceleration in body frame coordinates, $\vec{q}_{SC}$ is the satellite's attitude quaternion, $\vec{q}_{ACC}$ represents the accelerometer's frame orientation relative to the satellite's body frame, $\vec{\omega}^{\mathcal{B}}_{SC}$ is the angular rate of the satellite in the body frame, $\vec{r}^{\mathcal{B}}_{ACC/COM} \equiv \vec{r}^{\mathcal{B}}_{ACC} - \vec{r}^{\mathcal{B}}_{COM}$ is the accelerometer location relative to the satellite's center of mass (COM) location, $\vec{g}^{\mathcal{B}}$ is the gravitational acceleration experienced by the satellite in body frame coordinates, and $\vec{\epsilon}_{ACC}$ is any modeled sources of errors and biases in the sensor frame [15]. The $\otimes$ symbol in Equation 2.17 represents rotation of a vector by a quaternion, as described in Equation 2.8. The term $\ddot{\vec{r}}^{\mathcal{B}}_{SC} - \vec{g}^{\mathcal{B}}$ in Equation 2.17 is the apparent body frame acceleration of the satellite due to external non-gravitational forces acting on the satellite. The two cross product terms are the lever arm effects, where the term involving $\dot{\vec{\omega}}^{\mathcal{B}}_{SC}$ is the Euler acceleration and the triple cross product term is the centrifugal acceleration. If only the linear acceleration of the satellite is of interest for state estimation, the estimation algorithm must estimate and subtract the lever arm accelerations from the raw accelerometer measurement.

2.3.2.2 Gyroscopes

Gyroscopes measure the angular rates experienced by the satellite in each spatial direction. Their measurements are a critical input into attitude propagation algorithms which often use angular rate measurements for state estimation. Due to their importance for obtaining accurate attitude estimates, gyroscopes are sometimes used as standalone units instead of being integrated into an IMU. These standalone units can be mechanical gyroscopes, ring-laser gyroscopes (RLGs), or fiber-optic gyroscopes (FOGs). However, small satellites like nanosatellites may not always be able to accommodate these types of gyroscopes due to size and power limitations and will typically opt for MEMS gyroscopes. These gyroscopes, often integrated into an IMU, work by measuring Coriolis accelerations of a miniature quartz piezoelectric mass, which can be converted to an angular rate measurement [15]. For implementation in a GNC simulation framework, a gyroscope model following from Equation 2.9 can be written as:

$$\breve{\vec{\omega}} = \vec{q}_{GYR} \otimes \vec{\omega}^{\mathcal{B}}_{SC} + \vec{\epsilon}_{GYR} \tag{2.18}$$

where $\breve{\vec{\omega}}$ is the sensor frame sun vector measurement output, $\vec{\omega}^{\mathcal{B}}_{SC}$ is the satellite angular rate in the body frame, $\vec{q}_{GYR}$ represents the gyroscope's frame orientation relative to the satellite's body frame, and $\vec{\epsilon}_{GYR}$ is any modeled sources of errors and biases in the sensor frame. The $\otimes$ symbol in Equation 2.18 represents rotation of a vector by a quaternion, as described in Equation 2.8.

2.3.3 Orbit Sensors

Orbit sensors are used to make measurements of the position and velocity of the satellite. Satellite architectures that use ground segment communications, as in Figure 2.2, can utilize a process called ranging, wherein a groundstation sends an uplink signal to the satellite and the satellite subsequently sends a downlink signal to the groundstation. The uplink-downlink transit time can be used to compute the distance between the groundstation and the satellite. Several ranging data points can be collected, and orbit determination algorithms can be used to compute an orbit solution.

For satellite architectures with a more autonomous GNC architecture, as in Figure 2.3, a Global Navigation Satellite System (GNSS) such as the U.S. Global Positioning System (GPS) can be utilized, and orbit determination can be done using the satellite's on-board computer. This type of architecture is ideal for small satellites in LEO, which have high orbital velocities, are below the GNSS orbit, and have short groundstation pass durations. GNSS receivers can compute pseudorange measurements and/or carrier phase differences, which the receiver can resolve into satellite position and velocity in the Earth-centered Earth-fixed (ECEF) frame [15]. For implementation in a GNC simulation framework, a GNSS position model (Equation 2.19) and velocity model (Equation 2.20), both following from Equation 2.9, can be written as:

$$\check{\vec{r}} = \vec{q}_{ECEF} \otimes \vec{r}_{SC}^{\mathcal{N}} + \vec{r}_{\epsilon_{GNSS}} \tag{2.19}$$

$$\check{\dot{\vec{r}}} = \vec{q}_{ECEF} \otimes \dot{\vec{r}}_{SC}^{\mathcal{N}} + \dot{\vec{r}}_{\epsilon_{GNSS}} \tag{2.20}$$

where $\check{\vec{r}}$ (Equation 2.19) is the satellite position vector measurement in the ECEF frame, $\vec{r}_{SC}^{\mathcal{N}}$ is the satellite's orbital position in the inertial frame, $\vec{q}_{ECEF}$ represents the ECEF frame orientation relative to the inertial (ECI) frame, and $\vec{r}_{\epsilon_{GNSS}}$ is any modeled sources of positional errors and biases in the ECEF frame. Likewise, $\check{\dot{\vec{r}}}$ (Equation 2.20) is the satellite velocity vector measurement in the ECEF frame, $\dot{\vec{r}}_{SC}^{\mathcal{N}}$ is the satellite's orbital velocity in the inertial frame, and $\dot{\vec{r}}_{\epsilon_{GNSS}}$ is any modeled sources of velocity errors and biases in the ECEF frame. The $\otimes$ symbol in Equations 2.19 and 2.20 represents rotation of a vector by a quaternion, as described in Equation 2.8.

2.4 Actuators

GNC actuators are hardware components used to apply control forces and torques on the satellite to achieve a desired reference state. Thrusters are the enabling technology for orbit control as they operate based on Newton's Third Law of Motion, expelling propellant to apply thrust forces on the satellite. Attitude control torques can be achieved by a variety of different actuators with reaction wheels, control-moment gyroscopes (CMGs), and magnetorquers being the most common on small satellites. In general, actuator forces and torques can be modeled as follows [15]:

$$\check{\vec{f}}(t) = \vec{f}(t) + \vec{\epsilon}_f(t) \tag{2.21}$$

$$\check{\vec{\tau}}(t) = \vec{\tau}(t) + \vec{\epsilon}_\tau(t) \tag{2.22}$$

where $\check{\vec{f}}(t)$ and $\check{\vec{\tau}}(t)$ are the output actuator force and torque, $\vec{f}(t)$ and $\vec{\tau}(t)$ are the commanded force and torque, and $\vec{\epsilon}_f(t)$ and $\vec{\epsilon}_\tau(t)$ are the error torque and force which result from internal

frictions, resistances, oscillations, and other effects. The following diagram shows a block diagram representation of a generic actuator model [15].

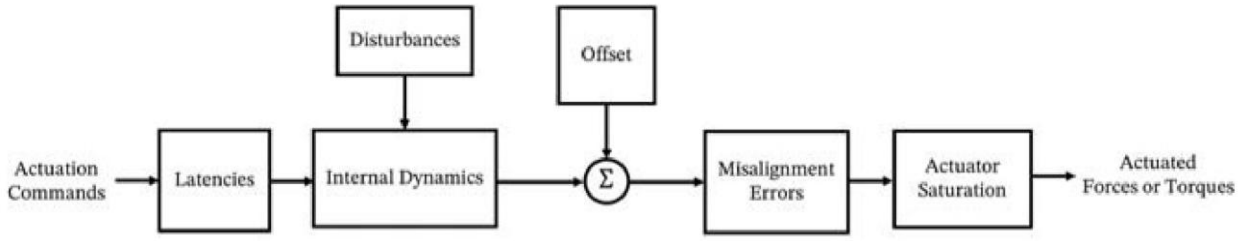


Figure 2.14 – Generic actuator model block diagram.

2.4.1 Thrusters

Thrusters, while historically uncommon on university CubeSats and small satellites, are necessary for performing orbit control maneuvers and can also be used for attitude control. Typical thruster types include cold gas, chemical, and electric thrusters. When its thrust direction is in line with the satellite's center of mass (COM), a thruster increases the velocity by an amount that can be ideally modeled by the Tsiolkovsky equation [15]:

$$\Delta V = I_{sp} g_0 \ln\left(\frac{m_0}{m_0 - m_p}\right) \tag{2.23}$$

where ΔV is the velocity increment, I_{sp} is the thruster's specific impulse, g_0 is the gravitational acceleration at Earth's surface, m_0 is the initial satellite mass, and m_p is the propellant mass used. ΔV is an important orbit control design parameter that must be considered for planning orbital maneuvers, station-keeping, or collision avoidance maneuvers. The magnitude of the thrust force is given by:

$$F_{THR} = \dot{m} I_{sp} g_0 \tag{2.24}$$

where $\dot{m}$ is the propellant mass flow rate at the exhaust.

Thrusters can also be used for attitude control if their thrust force vector acts perpendicular to a lever arm at a distance $\vec{r}^{\mathcal{B}}_{THR/COM} \equiv \vec{r}^{\mathcal{B}}_{THR} - \vec{r}^{\mathcal{B}}_{COM}$ from the COM, which results in a torque given by:

$$\vec{M}^{\mathcal{B}}_{THR} = \vec{r}^{\mathcal{B}}_{THR/COM} \times \vec{F}^{\mathcal{B}}_{THR} \tag{2.25}$$

Thruster induced torques are external moments as they do not result in momentum being transferred between components on the satellite. Therefore, when considering the overall rotational dynamics given by Equation 2.3, thruster induced torques are encapsulated in the $\vec{M}^{\mathcal{B}}_{ext}$ term.

Thrusters are necessary for orbit control and can deliver high torques for fast attitude control. Due to their induced torques being external moments on the satellite, they can be used to perform desaturation of internal momentum devices like reaction wheels. However, they require on-board

fuel storage which adds weight and potential launch risks that launch vehicle providers may need to assess.

As stated in rotational dynamics modeling Assumption 1 under Section 2.2.2, for the purposes of this project, the satellite will be assumed to maintain constant mass properties; therefore, thrusters will not be modeled in the final simulation architecture and are presented here for informational purposes only.

2.4.2 Magnetorquers

Magnetorquers are coils of conductive wire, often wrapped around a ferromagnetic core with high magnetic permeability, that generate a magnetic dipole when current is passed through the coil. As with thrusters, a magnetorquer induced torque is an external torque applied to the satellite and is therefore encapsulated in the $\vec{M}_{ext}^{\mathcal{B}}$ term of Equation 2.3. The control torque produced by a magnetorquer is given by:

$$\vec{M}_{MTQ}^{\mathcal{B}} = \vec{m}_{MTQ}^{\mathcal{B}} \times \vec{B}^{\mathcal{B}} \tag{2.26}$$

where $\vec{m}_{MTQ}^{\mathcal{B}}$ is the magnetic dipole normal to the coil cross-section and $\vec{B}^{\mathcal{B}}$ is the external (Earth) magnetic field vector. For magnetorquers with a core, the magnetic dipole magnitude is given by:

$$m = NIA + V_c M(\mu_r, I, G) \tag{2.27}$$

where N is the number of magnetorquer wire coils, I is the current passed through the coil, A is the cross-sectional area of the coil, V_c is the ferromagnetic core volume, and $M(\mu_r, I, G)$ is the magnetization factor, which is a function of the relative magnetic permeability μ_r, the current, and geometry of the core G. For the purposes of this project, the ferromagnetic core can be assumed to be isotropic and cylindrical with a constant cross-sectional area, allowing the following linear relationship to be used for the second term of Equation 2.27:

$$V_c M(\mu_r, I, G) = \left(\frac{\mu_r - 1}{\mu_r}\right) NIA \tag{2.28}$$

For magnetorquers without a core, also called air-core magnetorquers, only the first term of Equation 2.27 is applicable. A typical small satellite magnetorquer set layout is shown in the figure below.

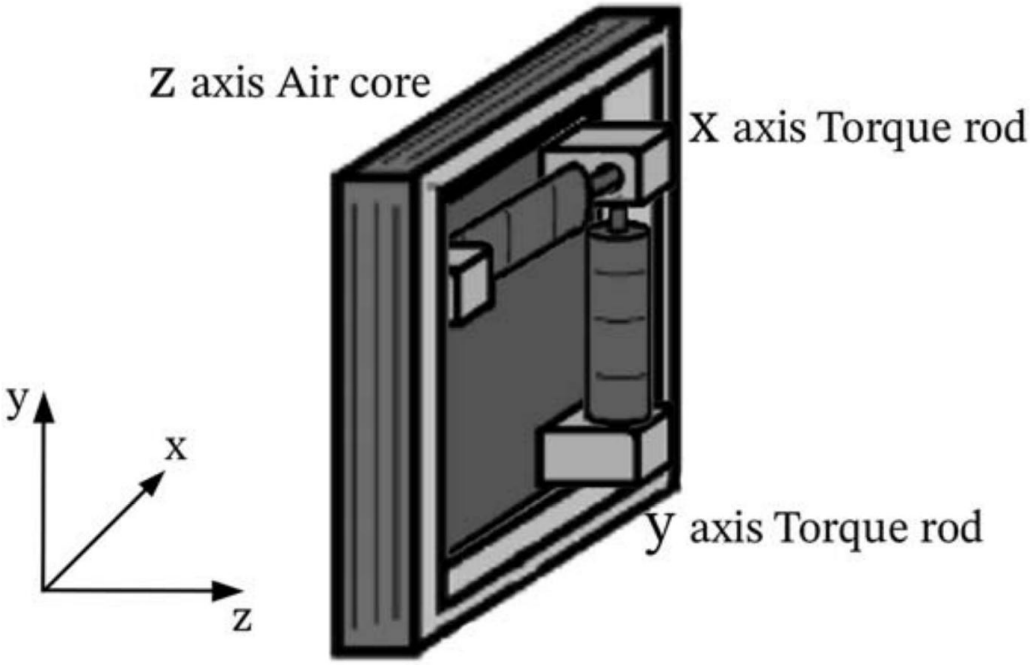


Figure 2.15 – Typical magnetorquer set layout.

Magnetorquers are relatively small, simple actuators that require no moving parts and have low power consumption. They are very useful for LEO applications, where Earth's magnetic field is strong. As with thrusters, due to their induced torques being external moments on the satellite, they can be used to perform desaturation of internal momentum devices like reaction wheels. However, they require interaction with an external magnetic field to operate and are generally limited to producing small torques on the order of 10^{-6} N-m for small satellites.

2.4.3 Reaction Wheels

Reaction wheels are rotating masses whose angular speed can be electronically controlled using a motor. They apply torques to the satellite by increasing or decreasing their speed; due to the conservation of momentum, changes in wheel momentums result in a reciprocal change in the satellite's momentum. If the reaction wheel's COM coincides with the satellite's COM, the total angular rate of the wheel in the satellite body frame, $\vec{\omega}^{\mathcal{B}}_{RW@COM}$, is equal to the sum of the satellite's body frame angular rate, $\vec{\omega}^{\mathcal{B}}_{SC}$, and the angular rate of the wheel in the wheel's frame of reference, $\vec{\omega}^{\mathcal{W}}_{RW}$, with a wheel-to-body frame transformation, ${}^{\mathcal{B}}R^{\mathcal{W}}$, applied:

$$\vec{\omega}^{\mathcal{B}}_{RW@COM} = \vec{\omega}^{\mathcal{B}}_{SC} + {}^{\mathcal{B}}R^{\mathcal{W}}\vec{\omega}^{\mathcal{W}}_{RW} \tag{2.29}$$

$$\vec{H}^{\mathcal{B}}_{RW@COM} = J_{RW}\vec{\omega}^{\mathcal{B}}_{RW@COM} \tag{2.30}$$

where J_{RW} is the reaction wheel's inertia about its spin axis. When the reaction wheel's COM is located at a distance away from the satellite's COM, an additional term is added to Equation 2.30, resulting in:

$$\vec{H}_{RW}^{\mathcal{B}} = \vec{H}_{RW@COM}^{\mathcal{B}} + m_{RW}\left(\vec{r}_{RW/COM}^{\mathcal{B}} \times \left(\vec{\omega}_{SC}^{\mathcal{B}} \times \vec{r}_{RW/COM}^{\mathcal{B}}\right)\right) \tag{2.31}$$

where m_{RW} is the mass of the reaction wheel and $\vec{r}_{RW/COM}^{\mathcal{B}} \equiv \vec{r}_{RW}^{\mathcal{B}} - \vec{r}_{COM}^{\mathcal{B}}$ is the reaction wheel COM location relative to the satellite's center of mass (COM) location. The second term in Equation 2.31 can be rearranged to isolate $\vec{\omega}_{SC}^{\mathcal{B}}$, resulting in the following:

$$\vec{H}_{RW}^{\mathcal{B}} = \vec{H}_{RW@COM}^{\mathcal{B}} + m_{RW}\left(\left(\left[\vec{r}_{RW/COM}^{\mathcal{B}}\right]^T \vec{r}_{RW/COM}^{\mathcal{B}}\right) \mathrm{I}_{3x3} - \vec{r}_{RW/COM}^{\mathcal{B}}\left[\vec{r}_{RW/COM}^{\mathcal{B}}\right]^T\right)\vec{\omega}_{SC}^{\mathcal{B}} \tag{2.32}$$

where I_{3x3} is the 3x3 identity matrix. The $\vec{\omega}_{SC}^{\mathcal{B}}$ term in $\vec{H}_{RW@COM}^{\mathcal{B}}$ from Equations 2.29 and 2.30 can then be combined with the second term in Equation 2.32, resulting in the following:

$$\begin{aligned}\vec{H}_{RW}^{\mathcal{B}} &= J_{RW}\,{}^{\mathcal{B}}R^{\mathcal{W}}\vec{\omega}_{RW}^{\mathcal{W}} \\ &+ \left[J_{RW} + m_{RW}\left(\left(\left[\vec{r}_{RW/COM}^{\mathcal{B}}\right]^T \vec{r}_{RW/COM}^{\mathcal{B}}\right) \mathrm{I}_{3x3} - \vec{r}_{RW/COM}^{\mathcal{B}}\left[\vec{r}_{RW/COM}^{\mathcal{B}}\right]^T\right)\right]\vec{\omega}_{SC}^{\mathcal{B}}\end{aligned} \tag{2.33}$$

where the bracketed term multiplying $\vec{\omega}_{SC}^{\mathcal{B}}$ in Equation 2.33 can be recognized as the tensor generalization of the parallel axis theorem.

Since reaction wheels are fixed to the satellite, their body-frame angular momentum is an internal (or "stored") momentum; $\vec{H}_{RW}^{\mathcal{B}}$ is therefore encapsulated within $\vec{H}_{int}^{\mathcal{B}}$ in Equation 2.3. Taking the derivative of Equation 2.33, noting that $\dot{\vec{r}}_{RW/COM}^{\mathcal{B}} = \vec{\omega}_{SC}^{\mathcal{B}} \times \vec{r}_{RW/COM}^{\mathcal{B}}$, results in the following expression for the reaction wheel angular momentum rate in the satellite's body frame:

$$\begin{aligned}\dot{\vec{H}}_{RW}^{\mathcal{B}} &= J_{RW}\,{}^{\mathcal{B}}R^{\mathcal{W}}\dot{\vec{\omega}}_{RW}^{\mathcal{W}} \\ &+ \left[J_{RW} + m_{RW}\left(\left(\left[\vec{r}_{RW/COM}^{\mathcal{B}}\right]^T \vec{r}_{RW/COM}^{\mathcal{B}}\right) \mathrm{I}_{3x3} - \vec{r}_{RW/COM}^{\mathcal{B}}\left[\vec{r}_{RW/COM}^{\mathcal{B}}\right]^T\right)\right]\dot{\vec{\omega}}_{SC}^{\mathcal{B}} \\ &- \vec{\omega}_{SC}^{\mathcal{B}} \times \left(m_{RW}\vec{r}_{RW/COM}^{\mathcal{B}}\left[\vec{r}_{RW/COM}^{\mathcal{B}}\right]^T \vec{\omega}_{SC}^{\mathcal{B}}\right)\end{aligned} \tag{2.34}$$

The net torque experienced by the satellite due to the reaction wheel is then given by:

$$\vec{M}_{RW}^{\mathcal{B}} = -\dot{\vec{H}}_{RW}^{\mathcal{B}} \tag{2.35}$$

where the negative sign indicates a reaction torque according to Newton's Third Law. Current inputs to the motor used to spin the wheel can modify the wheel angular rate, as the dynamics of the reaction wheel, $\dot{\vec{\omega}}_{RW}^{\mathcal{W}}$, are coupled with the electrical dynamics of the motor. Effective modeling of reaction wheel dynamics involves consideration of the various spurious effects that affect the dynamics, including friction, saturation limits, and other effects that result in added terms to Equation 2.35. High fidelity reaction wheel modeling may also consider static imbalances, where the reaction wheel's spin axis is not coincident with the wheel's COM, and dynamic imbalances, where the spin axis is not aligned with the principal moment of inertia direction. Modeling these effects would introduce further coupling terms with the satellite's angular rate and would also make the satellite's COM and inertia tensor time-varying properties, greatly complicating the equations of motion. As stated in rotational dynamics modeling Assumption 1 under Section 2.2.2, for the purposes of this project, the satellite will be assumed to maintain constant mass properties;

therefore, reaction wheel static and dynamic imbalances will not be included in the final simulation architecture.

Figure 2.16 illustrates a generic reaction wheel layout with three wheels for three-axis control [7]. Many satellite designs may also use a redundant fourth wheel to improve robustness to actuator failure.

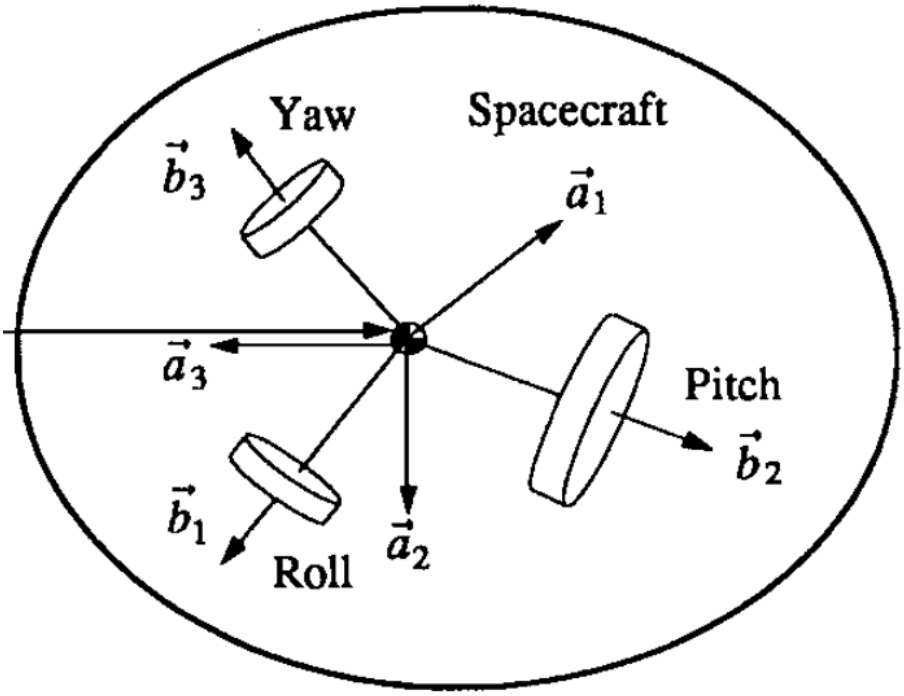


Figure 2.16 – Generic reaction wheel layout.

For small satellites, reaction wheels generally have torque capacities on the order of 10^{-3} N-m, which is much higher compared to magnetorquers, allowing for faster and more precise attitude maneuvering. They also allow for more precise pointing towards targets. However, their design involves moving parts which are more prone to failure and can potentially induce undesirable vibrations to the satellite during operation. Additionally, when used for three-axis stabilization, reaction wheels will counteract external torques to maintain the pointing objective, which will come at the cost of the wheel "absorbing" additional momentum. If the saturation momentum limit for a wheel is reached, it requires an actuator that induces external torques (i.e. thrusters or magnetorquers) to maintain attitude control while the wheel "desaturates" by returning to a nominal momentum [15].

2.4.4 Control-Moment Gyroscopes

Control-moment gyroscopes (CMGs) are rotating wheels mounted on gimbals that can be turned to change the direction of the wheel's angular momentum vector. This results in a gyroscopic effect that generates an actuation torque in the direction that is orthogonal to both the rotating wheel's spin axis and the gimbal axis rotation vector, as shown in Figure 2.17 [15].

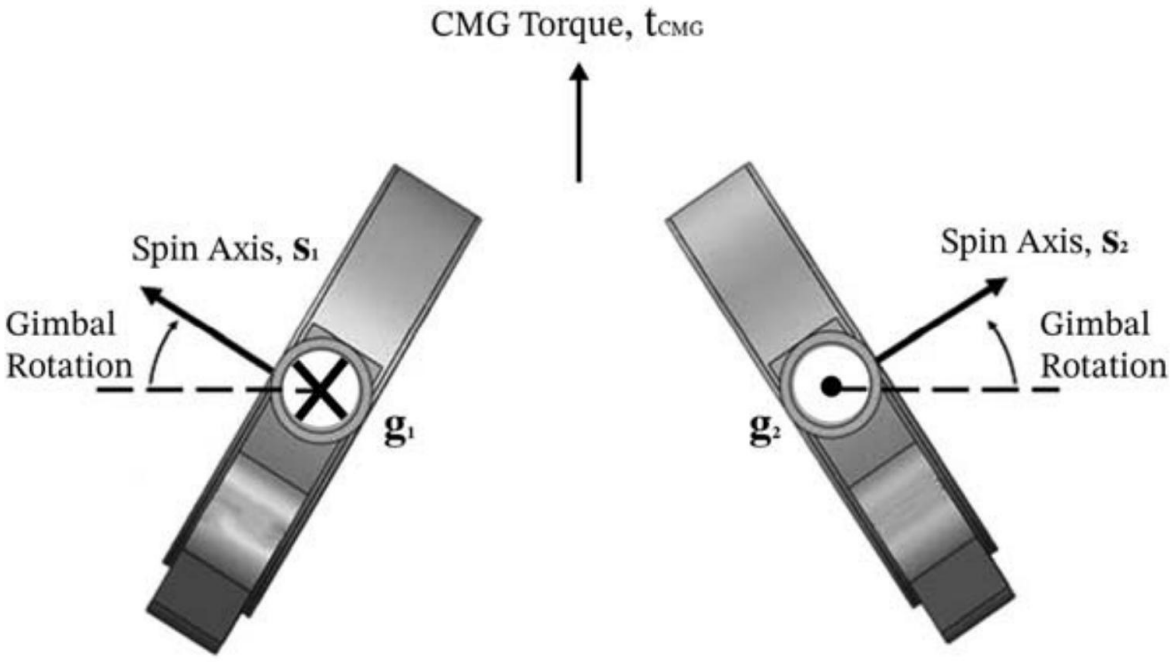


Figure 2.17 – CMG working principle diagram.

The most widely used CMGs have a single gimbal axis with a fixed speed rotating wheel; however, dual gimbal axis CMGs and variable speed CMGs also exist but descriptions of their complexity is beyond the scope of this project. If the CMG's COM coincides with the satellite's COM, the angular rate of a given CMG's wheel in the satellite body frame, $\vec{\omega}^{\mathcal{B}}_{CMG@COM}$, is equal to the sum of the satellite's body frame angular rate, $\vec{\omega}^{\mathcal{B}}_{SC}$, the rotating wheel's wheel-frame angular rate, $\vec{\omega}^{\mathcal{W}}_{WHL}$, and the gimbal's wheel-frame angular rate, $\vec{\omega}^{\mathcal{W}}_{GMBL}$, with a wheel-to-body frame transformation, ${}^{\mathcal{B}}R^{\mathcal{W}}$, applied to $\vec{\omega}^{\mathcal{W}}_{WHL}$ and $\vec{\omega}^{\mathcal{W}}_{GMBL}$:

$$\vec{\omega}^{\mathcal{B}}_{CMG@COM} = \vec{\omega}^{\mathcal{B}}_{SC} + {}^{\mathcal{B}}R^{\mathcal{W}}\left(\vec{\omega}^{\mathcal{W}}_{WHL} + \vec{\omega}^{\mathcal{W}}_{GMBL}\right) \tag{2.36}$$

$$\vec{H}^{\mathcal{B}}_{CMG@COM} = J_{CMG}\vec{\omega}^{\mathcal{B}}_{CMG@COM} \tag{2.37}$$

where J_{WHL} is the CMG wheel's inertia about its spin axis. When the CMG's COM is located at a distance away from the satellite's COM, an additional term is added to Equation 2.37, resulting in:

$$\vec{H}^{\mathcal{B}}_{CMG} = \vec{H}^{\mathcal{B}}_{CMG@COM} + m_{CMG}\left(\vec{r}^{\mathcal{B}}_{CMG/COM} \times \left(\vec{\omega}^{\mathcal{B}}_{SC} \times \vec{r}^{\mathcal{B}}_{CMG/COM}\right)\right) \tag{2.38}$$

where m_{CMG} is the mass of the CMG and $\vec{r}^{\mathcal{B}}_{CMG/COM} \equiv \vec{r}^{\mathcal{B}}_{CMG} - \vec{r}^{\mathcal{B}}_{COM}$ is the reaction wheel COM location relative to the satellite's center of mass (COM) location. The second term in Equation 2.38 can be rearranged to isolate $\vec{\omega}^{\mathcal{B}}_{SC}$, resulting in the following:

$$\vec{H}^{\mathcal{B}}_{CMG} = \vec{H}^{\mathcal{B}}_{CMG@COM} + m_{CMG}\left(\left(\left[\vec{r}^{\mathcal{B}}_{CMG/COM}\right]^{T}\vec{r}^{\mathcal{B}}_{CMG/COM}\right)\mathrm{I}_{3x3} - \vec{r}^{\mathcal{B}}_{CMG/COM}\left[\vec{r}^{\mathcal{B}}_{CMG/COM}\right]^{T}\right)\vec{\omega}^{\mathcal{B}}_{SC} \tag{2.39}$$

where I_{3x3} is the 3x3 identity matrix. The $\vec{\omega}_{SC}^{\mathcal{B}}$ term in $\vec{H}_{CMG@COM}^{\mathcal{B}}$ from Equations 2.36 and 2.37 can then be combined with the second term in Equation 2.39, resulting in the following:

$$\vec{H}_{CMG}^{\mathcal{B}} = J_{CMG}\,{}^{\mathcal{B}}R^{\mathcal{W}}\left(\vec{\omega}_{WHL}^{\mathcal{W}} + \vec{\omega}_{GMBL}^{\mathcal{W}}\right) + \left[J_{CMG} + m_{CMG}\left(\left(\left[\vec{r}_{CMG/COM}^{\mathcal{B}}\right]^{T}\vec{r}_{CMG/COM}^{\mathcal{B}}\right)\mathrm{I}_{3x3} - \vec{r}_{CMG/COM}^{\mathcal{B}}\left[\vec{r}_{CMG/COM}^{\mathcal{B}}\right]^{T}\right)\right]\vec{\omega}_{SC}^{\mathcal{B}} \quad (2.40)$$

where the bracketed term multiplying $\vec{\omega}_{SC}^{\mathcal{B}}$ in Equation 2.40 can be recognized as the tensor generalization of the parallel axis theorem.

Unlike for reaction wheels, ${}^{\mathcal{B}}R^{\mathcal{W}}$ is not a constant due to the gimbal's rotation. Therefore:

$$\frac{d}{dt}\left({}^{\mathcal{B}}R^{\mathcal{W}}\vec{\omega}_{WHL}^{\mathcal{W}}\right) = {}^{\mathcal{B}}R^{\mathcal{W}}\left(\dot{\vec{\omega}}_{WHL}^{\mathcal{W}} + \vec{\omega}_{GMBL}^{\mathcal{W}} \times \vec{\omega}_{WHL}^{\mathcal{W}}\right) \quad (2.41)$$

$$\frac{d}{dt}\left({}^{\mathcal{B}}R^{\mathcal{W}}\vec{\omega}_{GMBL}^{\mathcal{W}}\right) = {}^{\mathcal{B}}R^{\mathcal{W}}\left(\dot{\vec{\omega}}_{GMBL}^{\mathcal{W}} + \vec{\omega}_{GMBL}^{\mathcal{W}} \times \vec{\omega}_{GMBL}^{\mathcal{W}}\right) = {}^{\mathcal{B}}R^{\mathcal{W}}\dot{\vec{\omega}}_{GMBL}^{\mathcal{W}} \quad (2.42)$$

Like reaction wheels however, CMGs store momentum internally; $\vec{H}_{CMG}^{\mathcal{B}}$ is therefore encapsulated within $\vec{H}_{int}^{\mathcal{B}}$ in Equation 2.3. Taking the derivative of Equation 2.40, noting that $\dot{\vec{r}}_{CMG/COM}^{\mathcal{B}} = \vec{\omega}_{SC}^{\mathcal{B}} \times \vec{r}_{CMG/COM}^{\mathcal{B}}$, results in the following expression for the CMG angular momentum rate in the satellite's body frame:

$$\dot{\vec{H}}_{CMG}^{\mathcal{B}} = J_{CMG}\,{}^{\mathcal{B}}R^{\mathcal{W}}\left(\dot{\vec{\omega}}_{WHL}^{\mathcal{W}} + \dot{\vec{\omega}}_{GMBL}^{\mathcal{W}} + \vec{\omega}_{GMBL}^{\mathcal{W}} \times \vec{\omega}_{WHL}^{\mathcal{W}}\right) + \left[J_{CMG} + m_{CMG}\left(\left(\left[\vec{r}_{CMG/COM}^{\mathcal{B}}\right]^{T}\vec{r}_{CMG/COM}^{\mathcal{B}}\right)\mathrm{I}_{3x3} - \vec{r}_{CMG/COM}^{\mathcal{B}}\left[\vec{r}_{CMG/COM}^{\mathcal{B}}\right]^{T}\right)\right]\vec{\omega}_{SC}^{\mathcal{B}} - \vec{\omega}_{SC}^{\mathcal{B}} \times \left(m_{CMG}\vec{r}_{CMG/COM}^{\mathcal{B}}\left[\vec{r}_{CMG/COM}^{\mathcal{B}}\right]^{T}\vec{\omega}_{SC}^{\mathcal{B}}\right) \quad (2.43)$$

The net torque experienced by the satellite due to the CMG is then given by:

$$\vec{M}_{CMG}^{\mathcal{B}} = -\dot{\vec{H}}_{CMG}^{\mathcal{B}} \quad (2.44)$$

where the negative sign indicates a reaction torque according to Newton's Third Law. Current inputs to the motor used to turn the gimbal can modify the gimbal angular rate, as the dynamics of the gimbal, $\dot{\vec{\omega}}_{GMBL}^{\mathcal{W}}$, are coupled with the electrical dynamics of the motor. Effective modeling of CMG dynamics involves consideration of the various spurious effects that affect the dynamics, which like reaction wheels, may include friction, static and dynamic imbalances, saturation limits, and other effects that result in added terms to Equation 2.44. Figure 2.18 illustrates a CMG layout with a typical four-CMG pyramid configuration, where the $\vec{s}$ vectors are the rotating wheel spin axes (direction of $\vec{\omega}_{WHL}^{\mathcal{W}}$) and the $\vec{g}$ vectors are the gimbal axes (direction of $\vec{\omega}_{GMBL}^{\mathcal{W}}$) [15]:

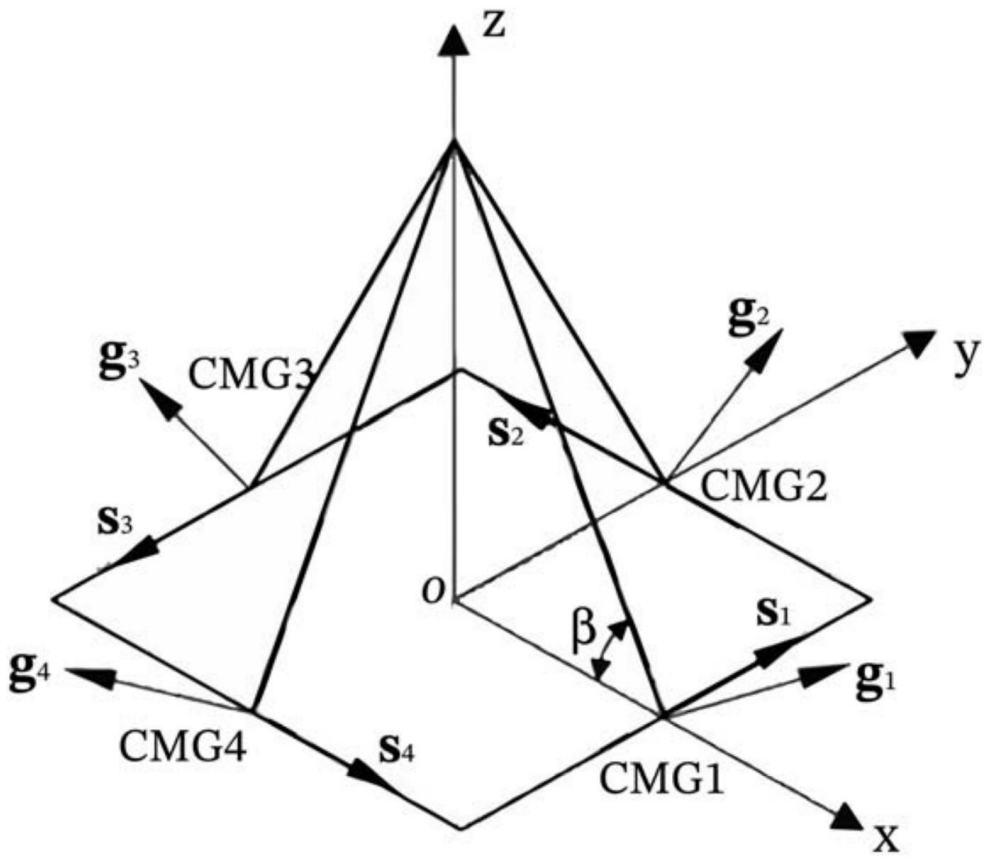


Figure 2.18 – Four CMG pyramid layout.

For small satellites, CMGs generally have torque capacities on the order of 10^{-3} to 10^{-2} N-m, which is slightly higher compared to reaction wheels. For this reason, they are useful for attitude maneuvers involving satellites with large moments of inertia. However, like reaction wheels, their design involves moving parts which are more prone to failure and can potentially induce undesirable vibrations to the satellite during operation. They are also susceptible to saturation when the spin axes of all the CMGs become aligned, resulting in the angular momentum of the CMG cluster to maximize. In this situation, actuators that induce external torques (i.e. thrusters or magnetorquers) must be utilized to maintain attitude control while the CMGs "desaturate" by returning to nominal gimbal orientations. Additionally, certain gimbal configuration may lead to singularities where no net torque is produced, which must be resolved by the CMG control algorithm [15].

As stated in rotational dynamics modeling Assumption 1 under Section 2.2.2, for the purposes of this project, the satellite will be assumed to maintain constant mass properties; because gimbal rotations of CMGs make the satellite's inertia tensor a time-varying quantity, they will not be modeled in the final simulation architecture and are presented here for informational purposes only.

2.5 On-Board Computer

The on-board computer (OBC) is the central computing element of the satellite, containing and executing all the flight software that enables it to function. As shown in Figure 2.3, the GNC

software on the OBC is divided into 4 functional categories: Mode Management, Guidance, Navigation, and Control. A detailed block diagram of how these categories interface with each other for a typical CubeSat GNC subsystem is shown in Figure 2.19 below [16]:

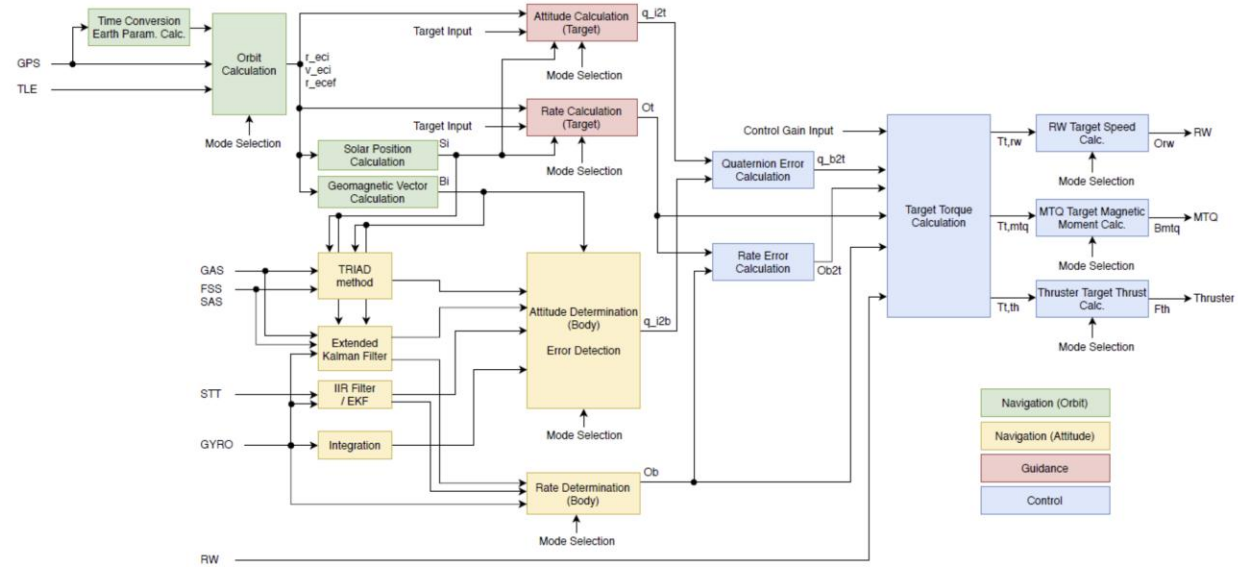


Figure 2.19 – Typical CubeSat GNC flight software block diagram.

2.5.1 Mode Management

The mode manager behaves as a finite state machine, with a discrete set of modes that control all the operational states that the satellite can have, as well as the criteria for transitioning between different modes. Typically, the operational states controlled by each mode in the mode manager extend beyond just the GNC subsystem and may include Power, Thermal, and other subsystems. However, GNC and ADCS-related functionalities are usually the primary distinguishing features between operational states, and therefore most significantly influence the contents of the mode manager. Figure 2.20 shows a diagram of the mode manager for Cat-2, a 6U CubeSat developed by the Universitat Politècnica de Catalunya [17]:

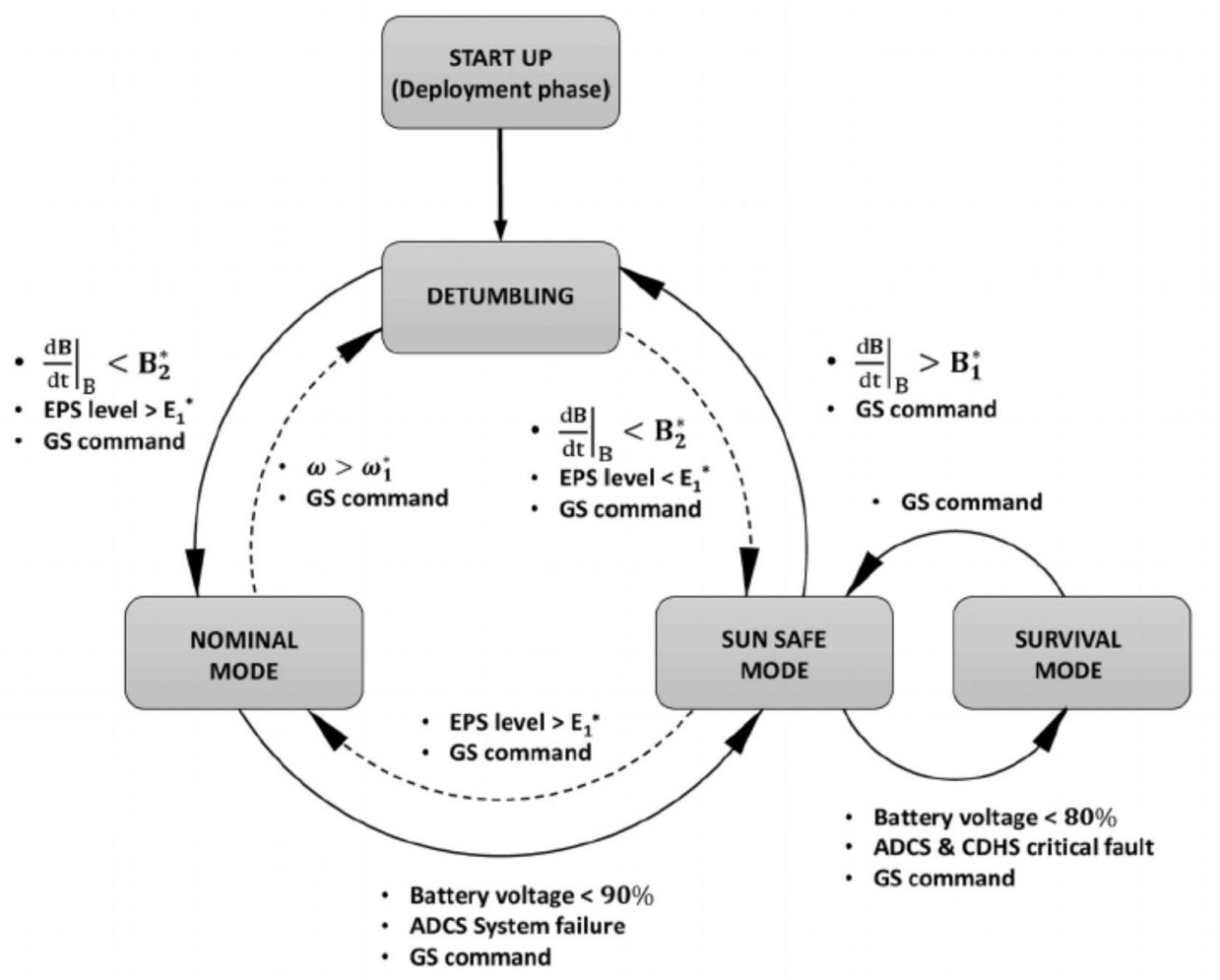


Figure 2.20 – Mode manager for Cat-2 CubeSat.

The gray boxes in the diagram represent each discrete mode that the satellite can be in at any given time. The arrows connecting the modes represent transitions between modes. The boolean statements associated with each transition define the criteria for whether mode transitions can occur. A GNC engineer may design different sets of guidance, navigation, and control algorithms to run for each commanded mode, while defining fault detection, isolation, and recover (FDIR) logic in the mode manager state machine to transition to a safe mode in contingency situations. In Figure 2.20 for instance, the "Nominal Mode" transitions to the "Sun Safe Mode" if the ADCS system experiences a fault or if the battery voltage goes below 90%.

As indicated by "GS command" in Figure 2.20, state machine modes can be actively commanded from sources such as groundstation command uplinks. Maximally autonomous GNC designs, like those described by Figure 2.3, are typically designed to limit groundstation uplinks and therefore may implement a mode command sequencer on the OBC to autonomously command modes to operate the satellite.

This project will not model groundstation interfaces to the satellite; therefore, the final simulation framework will include implementation of the mode manager state machine and a mode command sequencer.

2.5.2 Guidance

Guidance algorithms are responsible for generating the target dynamic states that the satellite must achieve, as well as the desired trajectories to take to achieve those states. Guidance targets can include translational states (i.e. desired inertial and/or relative orbital position and velocity) and rotational states (i.e. desired attitude and angular rates), and the targets generated may depend on which mode is active in the mode manager. For mission architectures that include a ground segment, as depicted in Figure 2.2, guidance targets may be uplinked to the satellite from ground operators and/or generated by the flight software on board the satellite. However, with the adoption of greater levels of autonomy in modern mission architectures, guidance algorithms are increasingly implemented in full on the OBC, as depicted in Figure 2.3.

Since the focus of this project will be on modeling the GNC subsystem for LEO satellites without orbit control systems, only ADCS guidance algorithms will be implemented in the final simulation framework.

2.5.3 Navigation

Navigation algorithms are used to estimate the current dynamic states of the satellite using a combination of raw measurements (i.e. observations from a suite of on-board sensors) and predictions from mathematical models of the state dynamics. States determined directly using raw sensor measurements can be noisy, while state predictions from models can have errors due to modeling simplifications. This makes neither of them ideal for accurate state estimation on their own. However, when they are combined through a "filtering" algorithm, the effects of sensor noise and modeling errors can be effectively smoothed out to provide accurate state estimates.

Navigation filtering can typically be done in two ways: batch filtering and sequential filtering. Batch filtering involves obtaining a set (or batch) of raw measurements over a time window and computing a model "best fit" solution of how the states evolved over the time window. This is often done offline and on the ground for tasks such as orbit determination, especially for mission architectures that include a ground segment: the ground operators collect ranging data over a time window, and orbit determination software on the ground uses a batch least squares filtering algorithm to compute the evolution of the orbital position and velocity in the time window. While the offline batch filtering approach can provide accurate estimates of translational states of the satellite, it may not be able to provide consistent, real-time state estimates that might be necessary for on-board orbit and attitude control systems. Instead, a sequential filtering approach can be used. This approach involves propagating the previous state estimate/uncertainties through a dynamic model to obtain predicted states/uncertainties, then passing these through a measurement model to obtain predicted measurements/uncertainties, which are finally filtered with the raw measurements/uncertainties to apply an update (or correction) to the predicted states/uncertainties to get the estimated states/uncertainties. A diagram of this process is shown in Figure 2.21 [15]. A notable feature of sequential filtering algorithms is that they are statistical in nature and require uncertainties to be propagated alongside the states themselves. These algorithms are usually implemented directly as flight software onto the OBC, allowing for navigation solutions to be computed fully online. This is necessary for mission architectures which do not utilize ground segments for GNC-related tasks. For example, LEO satellites can use GPS receivers to obtain raw measurements of the satellite's translational states: whenever a GPS measurement is obtained, an on-board orbit propagator can propagate the orbit from the previous translational state estimates to get the predicted states at the current timestep, and this can be filtered with the raw GPS measurements to correct the predictions and get the translational state estimates at the current

timestep. The following are sequential filtering algorithms commonly implemented on satellite GNC subsystems for navigation [15]:

- Extended Kalman Filter (EKF)
- Unscented Kalman Filter (UKF)
- Particle Filter (PF)

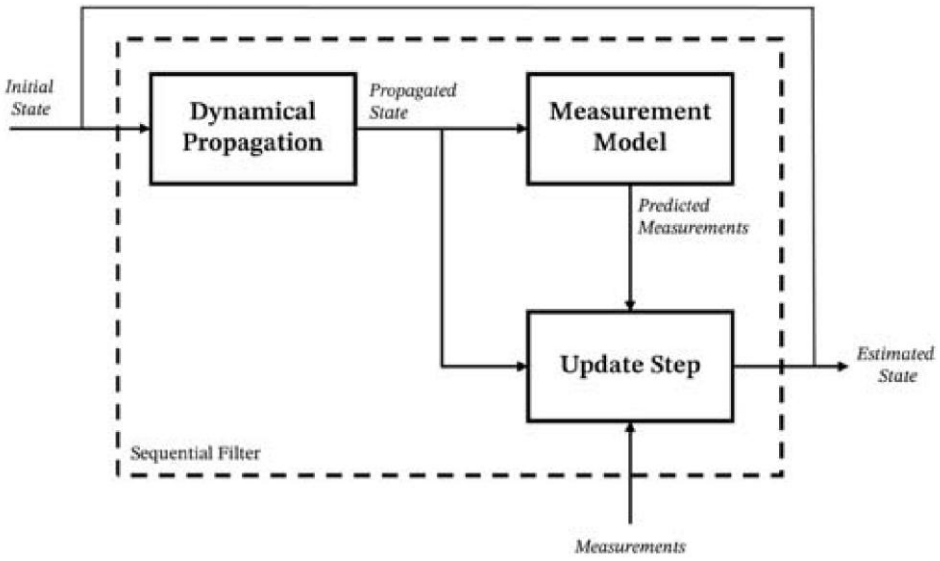


Figure 2.21 – Diagram of a sequential navigation filter.

In addition to filtering algorithms, another critical aspect of navigation is absolute state determination. Sequential filters require previous state estimates to generate new estimates at the current timestep; therefore, they must first be initialized before they can be used. An initial state estimate that is relatively accurate can improve how quickly the sequential filter (i.e. an EKF) converges. To determine the current state in an absolute way, measurements of external references from one or more sensors can be collected and passed through an algorithm, which can deterministically estimate the current state by identifying how the measurements are shifted relative to their expected values in a base reference frame (i.e. the inertial frame). Attitude determination algorithms are examples of absolute state determination algorithms that use attitude sensor measurements to construct estimates of the satellite's attitude. The following are common examples of attitude determination algorithms [15]:

- TRIAD
- Singular Value Decomposition (SVD)
- Davenport's q-method
- QUEST

Each of these are described in further detail in [15]. Once state targets are generated by the guidance algorithm, and state estimates are generated by the navigation algorithm, they can be fed as inputs to the control algorithm.

2.5.4 Control

Control algorithms are used to determine the necessary control forces and torques to control the satellite's translational and rotational dynamics in desired ways to achieve mission objectives. They process the "error state," which is computed as the difference between the guidance targets and the navigation state estimates, and subsequently use this error determine the necessary control forces and torques. Ultimately, the objective of a control algorithm is to drive trajectory or attitude tracking errors to zero while maintaining system stability. For attitude control in small satellites, proportional-integral-derivative (PID) control loops and quaternion-based feedback algorithms are frequently selected due to their computational efficiency and ease of implementation on standard OBCs. When dealing with highly coupled or non-linear rigid body dynamics, more advanced methods such as Linear Quadratic Regulators (LQR) or sliding mode controllers may be considered [7]. In addition to being functions of the error state, control algorithms are also functions of tuning parameters which affect the performance of the controller. For example, the control input generated by a PID controller for a multi-input multi-output (MIMO) system is described by the following equation:

$$\vec{u}(t) = K_p\vec{e}(t) + K_i\int_0^t \vec{e}(\tau)d\tau + K_d\frac{d\vec{e}(t)}{dt} \tag{2.45}$$

where $\vec{u}(t)$ is the control input vector, $\vec{e}(t)$ is the error state vector, K_p is the proportional gain tuning matrix, K_i is the integral gain tuning matrix, and K_d is the derivative gain tuning matrix [7]. Adjusting K_p influences how quickly the controller drives the system towards the target state due to its direct scaling of the error state. Adjusting K_i influences how quickly the controller removes any steady state error due to its scaling of the accumulated error. Adjusting K_d influences the rate at which the system is driven to its target state by the controller due to its scaling of the error rate. GNC engineers can use a variety of different techniques to tune these three gains depending on the objective, environment, and operating conditions for their system.

In general, satellite control algorithms are implemented using the following sequential logic:

1. Compute translational/rotational error states of the satellite.
2. Generate the desired control input vector using a controller algorithm.
3. Project the control input vector onto the actuator control space.
4. Use allocation logic to divide control input into components (commands) for each actuator.
5. Send commands to each actuator.

The GNC engineer designs a controller algorithm for each satellite mode to generate the desired control input vector ($\vec{u}(t)$). This can be a desired thrust force vector, applied torque vector, or other type of vector depending on the mode. This vector is then projected onto the actuator control space, which is the vector space spanned by all the independent directions that each actuator being utilized by the control algorithm can generate forces/torques in. This is usually a function of each actuator's mounting location and/or orientation relative to the satellite's body frame. Allocation logic considers how much of the control input each actuator should receive to generate the required total control torque. This is necessary when actuator control authority is overdetermined in the actuator control space. For example, it is common for a satellite to use 4 reaction wheels in a pyramid configuration for attitude control. In this situation, multiple solutions may exist for the wheels to generate any given control torque vector; therefore, the allocation logic must determine which solution to use and divide the control inputs accordingly. In the case of 4 identical reaction wheels,

the Moore-Penrose pseudoinverse of the 3x4 alignment matrix (composed of the spin axis directions of each wheel) multiplied by the control torque vector is typically used as the allocation logic to obtain an evenly distributed command torque distribution amongst the 4 wheels [7]. Finally, the commands must be sent to each actuator as a hardware level command. Modeling of the hardware level command and data handling system is beyond the scope of this project; simulation; all control algorithms and other user defined inputs will use Matlab syntax.

2.6 Space Environment

Satellites continuously interact with the space environment they operate within, with various environmental factors affecting the satellite's dynamics. These environmental factors generate disturbance forces and torques that alter both the orbital trajectory and the attitude of the satellite. To ensure accurate modeling in the final simulation framework, these perturbation sources must be mathematically modeled as external inputs affecting the satellite's orbital dynamics and rigid body rotational equations of motion.

2.6.1 Earth Gravity

Gravitational force due to the Earth's gravity potential is the most significant environmental factor that affects LEO satellites. As implied by the existence of a potential function, gravitational force is a conservative force: it does not add or remove mechanical energy from the satellite. Previously, it was shown that the gravity vector is modeled using Equation 2.2, which is derived from a function that approximates non-spherical variations of Earth's gravity potential using Legendre Polynomials and spherical harmonic functions. The determination of the coefficients multiplying the harmonic terms in the gravity function is important to generate an accurate model of Earth's gravity. For practical purposes, the U.S. National Geospatial Intelligence Agency developed several Earth Gravitational Models (EGMs) that are the U.S. scientific standards for Earth gravity modeling. The state-of-the-art model, EGM2008, was developed using experimental data from the Gravity Recovery and Climate Experiment (GRACE) satellites launched in 2002. This model contains spherical harmonic coefficients up to degree 2190, however, for nearly all LEO simulation applications, using truncations between degree 60 and 120 is sufficiently accurate. Most of the gravity variation is actually captured by just one coefficient at degree 2 (J2, the zonal harmonic term capturing the equatorial oblateness of the Earth) and using the truncation only up to this term may be sufficient for most applications [15].

Depending on the geometry of the satellite, gravity may also be a source of disturbance torques on the satellite. Generally, the magnitude of the gravitational force decreases in the radial direction moving further away from Earth. This results in stronger gravitational force applied to parts of the satellite closer to the Earth, and weaker force applied to parts of the satellite that are further away, manifesting in a force gradient along the geometry of the satellite. The result is a "gravity gradient" naturally restoring torque can be modeled in the satellite's body frame using the following expression:

$$\vec{M}_{\vec{g}}^{\mathcal{B}} = \frac{3\mu}{\left|\vec{r}_{SC}^{\mathcal{B}}\right|^3}\left(\frac{\vec{r}_{SC}^{\mathcal{B}}}{\left|\vec{r}_{SC}^{\mathcal{B}}\right|} \times J_{SC}\frac{\vec{r}_{SC}^{\mathcal{B}}}{\left|\vec{r}_{SC}^{\mathcal{B}}\right|}\right) \tag{2.46}$$

where μ is Earth's gravitational parameter, $\vec{r}_{SC}^{\mathcal{B}}$ is the body frame position vector of the satellite, and J_{SC} is the inertia tensor of the satellite [15]. Like with gravitational force, the gravity gradient torque in Equation 2.46 is a conservative torque: it does not add or remove mechanical energy

from the satellite. It is derived from the same gravity potential used to derive the gravitational force and is a function of the spherical harmonic coefficients. Gravity gradient torques are typically one of the smallest environmental contributors to torque disturbances for a LEO satellite, and the gravity gradient torque contributions from terms involving the spherical harmonic coefficients are even more insignificant; neglecting them results in Equation 2.46, which is sufficient for most applications.

2.6.2 Atmospheric Drag

In low Earth orbit, drag due to the satellite body's interaction with Earth's atmosphere is typically the next most significant environmental perturbation affecting the satellite's dynamics. Although LEO is characterized as a near-vacuum, a sparse atmospheric density remains present and interacts with the spacecraft traveling at high orbital velocities. This interaction produces an aerodynamic drag force acting opposite to the satellite's velocity vector, which strips energy from the orbit and causes gradual orbital decay, making it a non-conservative force, unlike gravity. The aerodynamic drag is described by the following fluid dynamics relation [15]:

$$\vec{F}_{\vec{D}}^{\mathcal{N}} = -\frac{m_{SC}}{B_{SC}}\frac{1}{2}\rho\left|\vec{v}_{SC/E}^{\mathcal{N}}\right|^2\frac{\vec{v}_{SC/E}^{\mathcal{N}}}{\left|\vec{v}_{SC/E}^{\mathcal{N}}\right|} \tag{2.47}$$

where m_{SC} is the mass of the satellite, B_{SC} is the ballistic coefficient (mass divided by drag area) of the satellite, $\vec{v}_{SC/E}^{\mathcal{N}} \equiv \dot{\vec{r}}_{SC}^{\mathcal{N}} - \vec{\omega}_E^{\mathcal{N}} \times \vec{r}_{SC}^{\mathcal{N}}$ is the relative wind velocity, and ρ is the atmospheric density. Accurate models of Earth's atmospheric density in LEO involve consideration of complex compositional, thermodynamic, solar, and geomagnetic effects. A variety of analytical and empirical atmospheric density models exist, as shown in Figure 2.22 below [15].

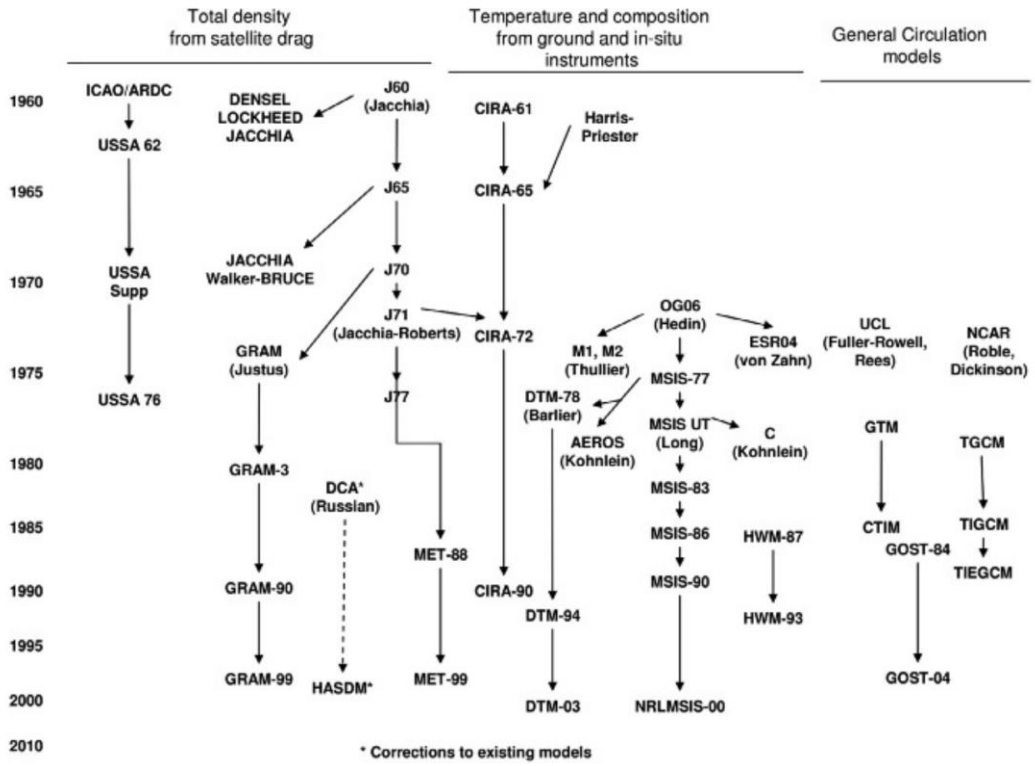


Figure 2.22 – Earth atmospheric density model evolution.

The most widely used atmospheric density model for modern applications is NRLMSISE-00, an empirical model developed in 2000 by the U.S. Naval Research Laboratory using empirical data from mass spectrometer and incoherent scatter radar of upper atmosphere chemical composition and temperature.

If the satellite's geometric center of pressure (COP) is offset from its center of mass (COM), the resultant aerodynamic force creates an external disturbance torque that perturbs the vehicle's attitude. The aerodynamic torque is given by:

$$\vec{M}_{\vec{D}}^{\mathcal{B}} = \vec{r}_{COP/COM}^{\mathcal{B}} \times \vec{F}_{\vec{D}}^{\mathcal{B}} \tag{2.48}$$

where $\vec{r}_{COP/COM}^{\mathcal{B}} \equiv \vec{r}_{COP}^{\mathcal{B}} - \vec{r}_{COM}^{\mathcal{B}}$ is the satellite's COP location relative to the COM location and $\vec{F}_{\vec{D}}^{\mathcal{B}}$ is the aerodynamic drag force in the satellite's body frame [15]. Equation 2.48 becomes significant as a non-conservative disturbance torque addition for satellites in LEO with large deployable appendages, such as solar arrays, and highly asymmetric geometries.

2.6.3 Magnetic Field

Earth's geomagnetic field behaves as a significant rotational perturbation source for small satellites operating in low altitude orbits. Residual magnetic dipoles within the spacecraft—generated by active electronic circuit loops, unshielded components, or magnetic hysteresis loops in structural materials—interact with the local geomagnetic field vector. This cross-product interaction yields an external torque that continually acts to align the spacecraft's residual magnetic

moment with the local Earth magnetic field line. The magnetic disturbance torque acting on a satellite is a non-conservative torque and is described by the following equation:

$$\vec{M}_{\vec{B}}^{\mathcal{B}} = \vec{m}_{SC}^{\mathcal{B}} \times \vec{B}^{\mathcal{B}} \tag{2.49}$$

where $\vec{m}_{SC}^{\mathcal{B}}$ is the net residual magnetic dipole (not including dipoles of magnetorquers) of the satellite and $\vec{B}^{\mathcal{B}}$ is the external (Earth) magnetic field vector [15].

2.6.4 Solar Radiation Pressure

Solar radiation pressure (SRP) stems from the continuous momentum exchange between solar photons and the exposed surfaces of the satellite. The magnitude of the resulting force is a function of the satellite's distance from the Sun, the surface area exposed to the incident solar rays, and the optical reflectivity properties of the surface materials. The simplest model for SRP force is the "cannonball" model, which approximates the satellite's geometry as a sphere:

$$\vec{F}_{SRP}^{\mathcal{N}} = -p_{SRP} C_R A \frac{\vec{s}^{\mathcal{N}}}{|\vec{s}^{\mathcal{N}}|} \tag{2.50}$$

where $p_{SRP} = 4.57 \times 10^{-6}\ N/m^2$ is the SRP constant (solar flux divided by the speed of light) at Earth's location relative to the sun, C_R is the reflectivity coefficient of the satellite (captures transparency/reflectivity/absorptivity features of the satellite), A is the sunlit area of the satellite (typically the largest cross-sectional area is used instead), and $\vec{s}^{\mathcal{N}}$ is the inertial frame Earth-to-Sun vector [15]. For most small satellites, Equation 2.50 is typically a sufficient model for capturing forces due to SRP. However, satellites with highly non-uniform geometries and large structural appendages (i.e. solar arrays) typically require higher fidelity models that consider specular and diffusive effects as well as the satellite's specific geometric features. These are discussed in greater detail in [15].

When an asymmetry exists between the center of solar pressure and the center of mass, the radiation force induces a continuous disturbance torque that must be neutralized by the active control system. This SRP torque is described by the following equation:

$$\vec{M}_{SRP}^{\mathcal{B}} = \vec{r}_{COSRP/COM}^{\mathcal{B}} \times \vec{F}_{SRP}^{\mathcal{B}} \tag{2.51}$$

where $\vec{r}_{COSRP/COM}^{\mathcal{B}} \equiv \vec{r}_{COSRP}^{\mathcal{B}} - \vec{r}_{COM}^{\mathcal{B}}$ is the satellite's center of solar radiation pressure (COSRP) location relative to the COM location and $\vec{F}_{SRP}^{\mathcal{B}}$ is the solar radiation pressure force in the satellite's body frame [15]. As with aerodynamic torque, Equation 2.51 becomes significant as a non-conservative disturbance torque addition for satellites in LEO with large deployable appendages, such as solar arrays, and highly asymmetric geometries.

At some locations along the satellite's orbital trajectory, visibility to the sun is hindered by Earth and moon eclipses. Eclipses are typically modeled using dual cone algorithms and are described in further detail in [15].

2.6.5 Third Body Perturbations

While Earth's gravity is by far the most significant component of the total gravitational force acting on a LEO satellite, small (conservative) forces due to the gravitational influence of other celestial bodies (collectively called "third bodies") also contribute perturbative additions to the

translational and rotational dynamics of the satellite. The most significant third body contributions come from the sun, the moon, Venus, Mars, Jupiter, and Saturn. These bodies are located at distances that are much larger than the satellite's orbital distance from Earth's center of mass. As a result, each of these bodies can be approximated as point masses with uniform gravitational potentials about their centers of mass. This allows the following N-body formulation to be used as the third body perturbative force model:

$$\vec{F}_{3BP}^{\mathcal{N}} = m_{SC} \sum_{i=1}^{N} \mu_i \left(\frac{\vec{r}_{SC}^{\mathcal{N}} - \vec{r}_{i/E}^{\mathcal{N}}}{\left| \vec{r}_{SC}^{\mathcal{N}} - \vec{r}_{i/E}^{\mathcal{N}} \right|^3} - \frac{\vec{r}_{i/E}^{\mathcal{N}}}{\left| \vec{r}_{i/E}^{\mathcal{N}} \right|^3} \right) \tag{2.52}$$

where m_{SC} is the mass of the satellite, μ_i is the gravitational parameter of the i-th body, $\vec{r}_{SC}^{\mathcal{N}}$ is the inertial frame position of the satellite, and $\vec{r}_{i/E}^{\mathcal{N}} \equiv \vec{r}_i^{\mathcal{N}} - \vec{r}_E^{\mathcal{N}}$ is the position of the i-th body relative to the central body (Earth). Due to $\vec{F}_{3BP}^{\mathcal{N}}$ being a conservative force, it is encapsulated within $\vec{F}_{pertb}^{\mathcal{N}}$ in Equation 2.2. Determining $\vec{r}_{i/E}^{\mathcal{N}}$ requires knowledge of celestial body positions at any given time, which can be provided by ephemerides that model the time evolution of their positions. Development Ephemerides 430 (DE-430) is a high accuracy ephemerides model created by the Jet Propulsion Laboratory in 2013 that serves as the international standard for celestial position references. The model is a polynomial curve fit solution of the integrated multi-body planetary dynamics equations of motion. Namely, Chebyshev polynomials of the first kind are typically the polynomial function of choice for the curve fit. The nature of these polynomials is described in further detail in [15].

Chapter 3 – Software Implementation

3.1 Design Overview

This chapter presents the software development and implementation details of the culminating work of this project: the Guidance, navigation, and control Integrated Matlab-Based Architecture for Low earth orbiting satellite Simulations (GIMBALS) program. GIMBALS is a software application written in Matlab that allows LEO satellite GNC engineers to translate their subsystem-level GNC architecture definition into a fully-fledged parametric Simulink model for simulation, testing, and analysis. The user provides GIMBALS with an input file (a standard Matlab .m script file) presenting the relevant architecture definition, including satellite/component mass properties, sensor/actuator hardware model characteristics, state machine modes/transitions and their associated GNC algorithms, and the mode command sequence. GIMBALS can then be used to automatically build a Simulink model according to the user specifications. The user can subsequently run the generated model to simulate their scenario, assess the feasibility of their physical system design, and evaluate the performance of their GNC algorithms for each mode. Depending on their desired development workflow, the user can choose to update their architecture definition input to GIMBALS to reflect design innovations and regenerate the Simulink model, or use GIMBALS to generate multiple Simulink models to do side-by-side comparisons and system level trade analyses, or simply use the GIMBALS generated Simulink model as a baseline for further development within Simulink itself.

3.2 Software Structure

For the user to be able to communicate their GNC architecture definition to GIMBALS, an application programming interface (API) must be developed. Object-oriented programming (OOP) was utilized to develop the GIMBALS API to reflect the modular, compartmentalized nature of satellite GNC subsystems. GIMBALS objects were defined as Matlab Classes and organized into the folder structure shown in Appendix A. In addition to providing an external facing API for the user, the Matlab Classes contain backend methods for building the Simulink model. This includes a Build method and a Connect method in each Matlab Class. The Build method programmatically creates the necessary Simulink blocks for modeling the associated satellite component in Simulink and all internal data links between these blocks. The Connect method programmatically creates links to other satellite components that the given component interfaces with.

The sections below will provide a description of each GIMBALS object, including its API reference, relationship to the satellite component hierarchy, and its generated graphical representation in Simulink. The graphical representation varies based on the specifics defined in the user's GNC architecture definition. For presentation in this report, the Simulink model that was generated by GIMBALS using the sample input in Appendix B is used to showcase the general graphical appearance of GIMBALS generated outputs. The sample input models a simple 2U sun-synchronous (SSO) LEO CubeSat called "TestSat" with 4 reaction wheels in a pyramid configuration along with 3 magnetorquers along each satellite body frame axis. It contains typical IMU sensors (magnetometer, gyroscope, and accelerometer) with their sensor frames aligned with the satellite body frame, as well as 6 sun sensors mounted on each face of the CubeSat, each with a 120° FOV. Due to project development time limitations, only one state machine mode (detumbling) was defined for TestSat that utilizes 3-axis magnetic detumbling.

For each GIMBALS object, the API references are provided as a table that includes the object constructor which the user must use in their input file, as well as the Matlab Class properties and

methods that the user has access to. For each Matlab Class property listed in the tables, the required units of the input are shown if relevant, as well as a "reference type" identifier classifying the property's access limitations within the scope of user-defined GNC methods.

The following explanation may help clarify the purpose of the reference type identifiers: user-defined GNC methods refer to the OBC algorithms described under section 2.5. In GIMBALS, the user defines these algorithms in separate Matlab (.m) function files which will be called by Simulink during simulation runs. The user can pass GIMBALS objects defined in their input file as parameter inputs into their GNC methods. For example, the TestSat example in Appendix B implements an EKF function as the state estimator (Appendix D), which passes in a Simulation object, Gyroscope object, and an Accelerometer object as parameter inputs. Within these methods, the user may or may not be able to query and/or set properties of the objects they pass in as inputs depending on the property's reference type. The following are reference type identifiers and their definitions:

- noref: This signifies that the given property is not available for active reference inside the user-defined GNC method while the Simulink model runs.
- staticref: This signifies that the given property value is not a time-varying quantity and can be queried for reference inside the user-defined GNC method, but it cannot be updated inside the method.
- dynamicref: This signifies that the given property value is a time-varying quantity and can be queried for reference inside the user-defined GNC method, but it cannot be updated inside the method.
- dynamicvar: This signifies that the given property value is a time-varying quantity and can be queried for reference AND updated inside the user-defined GNC method.

The purpose of creating these reference type classifications, in the context of the software design process, was to resolve compatibility issues with Simulink. Simulink cannot directly work with custom Matlab Class instances during simulation runtime. However, the structure of the Matlab Classes can be mirrored by compatible data containers like Matlab structs and Simulink Bus objects, provided that certain conditions are met (i.e. property must be numeric type or structure of numeric types, property data dimensions must be fixed, etc.). The reference type identifier indicates whether these conditions are met. It is important to note that while the reference types indicate limitations within the scope of user-defined GNC methods, the user may still need to assign values to them in their GIMBALS input file as part of their broader GNC architecture definition.

3.2.1 The Simulation Object

The Simulation Object is defined in the Simulation.m Matlab Class. It is located in the "systems" folder within the project folder structure as shown in Figure A.3. This object is the top level container for the user's GNC architecture definition and also allows the user to specify global simulation settings to be applied to the generated Simulink model. It is also the only object that provides a public facing Build method for the user to call at the very end of their GIMBALS input file.

Table 3.1 – GIMBALS Simulation object API reference.

Constructor*	`sim = Simulation(<name>=<value>)`

Property**	Units	Ref. Type	Description
ProjectFolder	——	noref	User project folder path (<User Config File Location>/<Simulation Name>/); read-only property
EarthModel	——	noref	Simulation Earth Model; read-only property
Name	——	noref	Simulation name
ID	——	staticref	Simulation integer ID (optional user variable)
StartTime	Matlab Datetime	noref	Simulation start time (UTC)
EndTime	Matlab Datetime	noref	Simulation end time (UTC)
Duration	Matlab Duration	noref	Simulation duration
ElapsedTime	s	dynamicref	Simulation elapsed (clock) time
JulianDate	days	dynamicref	Simulation Julian Date
ModifiedJulianDate	days	dynamicref	Simulation Modified Julian Date
Methods	**`sim.Build()`**		

* **`<name>=<value>`** in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.
** Properties can be assigned using standard dot notation (**`sim.<name> = <value>`**).

The following screenshot illustrates the appearance of the generated Simulink model's root. The top level blocks were programmatically generated by the Build method inside the Simulation object's Matlab Class definition.

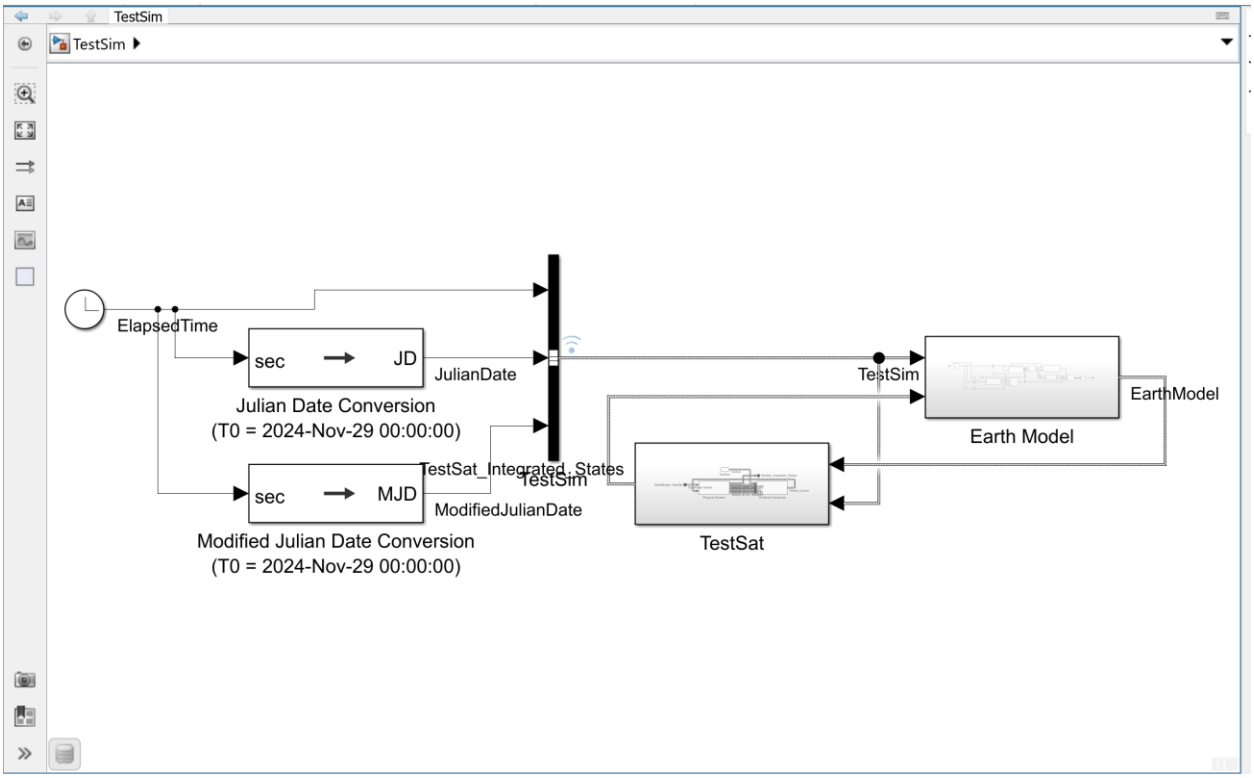


Figure 3.1 – GIMBALS Simulation Simulink representation.

The simulation time is sourced from the Simulink Clock Block, converted to different time representations, collected into a Simulink Bus through a Bus Creator block, and routed to the rest of the simulation.

3.2.2 The EarthModel Object

The EarthModel Object is defined in the EarthModel.m Matlab Class. It is located in the "environment" folder within the project folder structure as shown in Figure A.5. This object facilitates the creation of environmental model reference blocks in the generated Simulink model. The environmental disturbance forces/torques described in section 2.6 can be computed using various Earth model references (i.e. Earth-to-sun vector for SRP, magnetic field vector for magnetic torque disturbance, etc.). Due to time limitations near the end of this project's development period, only Earth's gravity, magnetic field, Earth-to-sun vector, and eclipse fraction models were included in the final Simulink model.

Table 3.2 – GIMBALS EarthModel object API reference.

Constructor*	`earthModel = EarthModel(Parent=sim, <name>=<value>)`		
Property**	**Units**	**Ref. Type**	**Description**
Parent	——	noref	Handle to parent Simulation object
Name	——	noref	Earth Model name

ID	———	staticref	Earth Model integer ID (optional user variable)
UpdateRate	Hz	noref	Earth Model update rate (rate at which environment dynamics are updated)

* **<name>=<value>** in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.
** Properties can be assigned using standard dot notation (**earthModel.<name> = <value>**). Note: while the constructor can be used to create an EarthModel object instance, the Simulation object uses its own internal EarthModel instance that it is already bound to. The user can edit the properties of this internal instance using nested dot notation (**sim.EarthModel.<name> = <value>**).

The following screenshot illustrates the appearance of the Simulink model's EarthModel block contents, which was programmatically generated by the Build method inside the EarthModel object's Matlab Class definition.

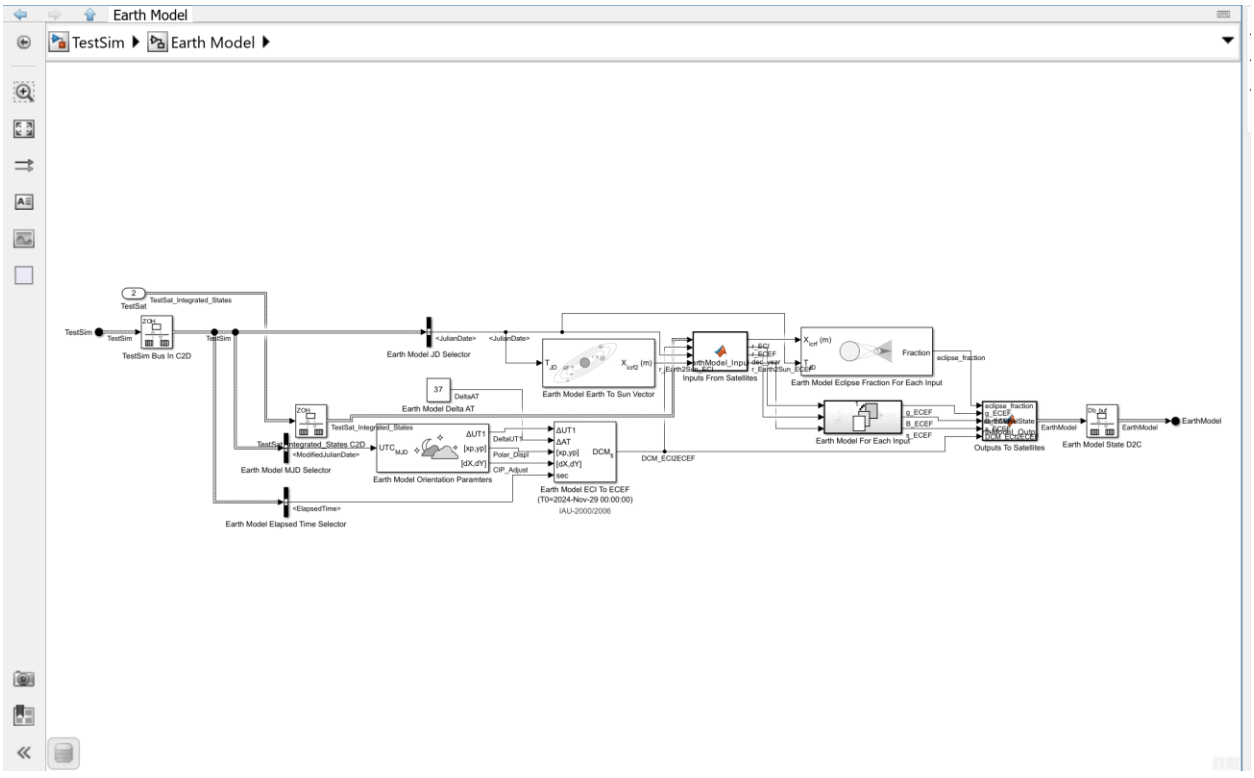


Figure 3.2 – GIMBALS EarthModel Simulink representation.

Bus data from each satellite that was defined in the GIMBALS input file is passed into the EarthModel block. Earth model references are computed in a vectorized fashion within a For-Each Input Block and finally sent to each satellite. Many blocks in Figure 3.2 are borrowed from the

Simulink Aerospace Blockset library because the library is a well validated official Matlab/Simulink block library, and because manual implementation would substantially increase the required development time.

3.2.3 The Satellite Object

The Satellite Object is defined in the Satellite.m Matlab Class. It is located in the "system" folder within the project folder structure as shown in Figure A.3. This object represents a user-defined satellite at the system level and is the container for all user-defined components of the satellite, including sensors, actuators, and OBC algorithms. The API reference for this object also allows the user to define overall mass properties and initial conditions for their satellite(s).

Table 3.3 – GIMBALS Satellite object API reference.

Constructor*	`sat = Satellite(Parent=sim, <name>=<value>)`		
Property**	**Units**	**Ref. Type**	**Description**
Parent	——	noref	Handle to parent Simulation object
Name	——	noref	Satellite name
ID	——	staticref	Satellite integer ID (optional user variable)
Guidance	——	noref	Satellite Guidance object
Navigation	——	noref	Satellite Navigation object
Control	——	noref	Satellite Control object
ModeManager	——	noref	Satellite ModeManager object
InitialMass	kg	staticref	Satellite initial net mass
InitialInertiaTensor	kg-m^2	staticref	Satellite initial inertia tensor
InitialCOMLocation	m	staticref	Satellite initial center of mass location vector (from body frame origin)
InitialPerigeeRadius	m	staticref	Satellite initial LEO perigee radius
InitialApogeeRadius	m	staticref	Satellite initial apogee radius
InitialSemimajorAxis	m	staticref	Satellite initial orbit semimajor axis (Keplerian)

InitialEccentricity	——	staticref	Satellite initial orbit eccentricity (Keplerian)
InitialInclination	deg	staticref	Satellite initial orbit inclination (Keplerian)
InitialRightAscension	deg	staticref	Satellite initial orbit right ascension of the ascending node (Keplerian)
InitialArgumentOfPerigee	deg	staticref	Satellite initial orbit argument of perigee (Keplerian)
InitialTrueAnomaly	deg	staticref	Satellite initial orbital true anomaly (Keplerian)
InitialPosition	m	staticref	Satellite initial orbital position (Earth Centered Inertial)
InitialVelocity	m/s	staticref	Satellite initial orbital velocity (Earth Centered Inertial)
InitialAttitudeEuler	deg	staticref	Satellite initial attitude Euler angles (ZYX)
InitialAttitudeDCM	——	staticref	Satellite initial attitude Direction Cosine Matrix
InitialAttitudeQuaternion	——	staticref	Satellite initial attitude quaternion
InitialAttitudeAxisAngle	deg (for angle)	staticref	Satellite initial attitude Axis-Angle Representation
InitialAngularVelocity	rad/s	staticref	Satellite initial body frame angular rates

* **<name>=<value>** in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.
** Properties can be assigned using standard dot notation (**sat.<name> = <value>**).

The following screenshot illustrates the appearance of the Simulink model's Satellite block contents, which was programmatically generated by the Build method inside the Satellite object's Matlab Class definition.

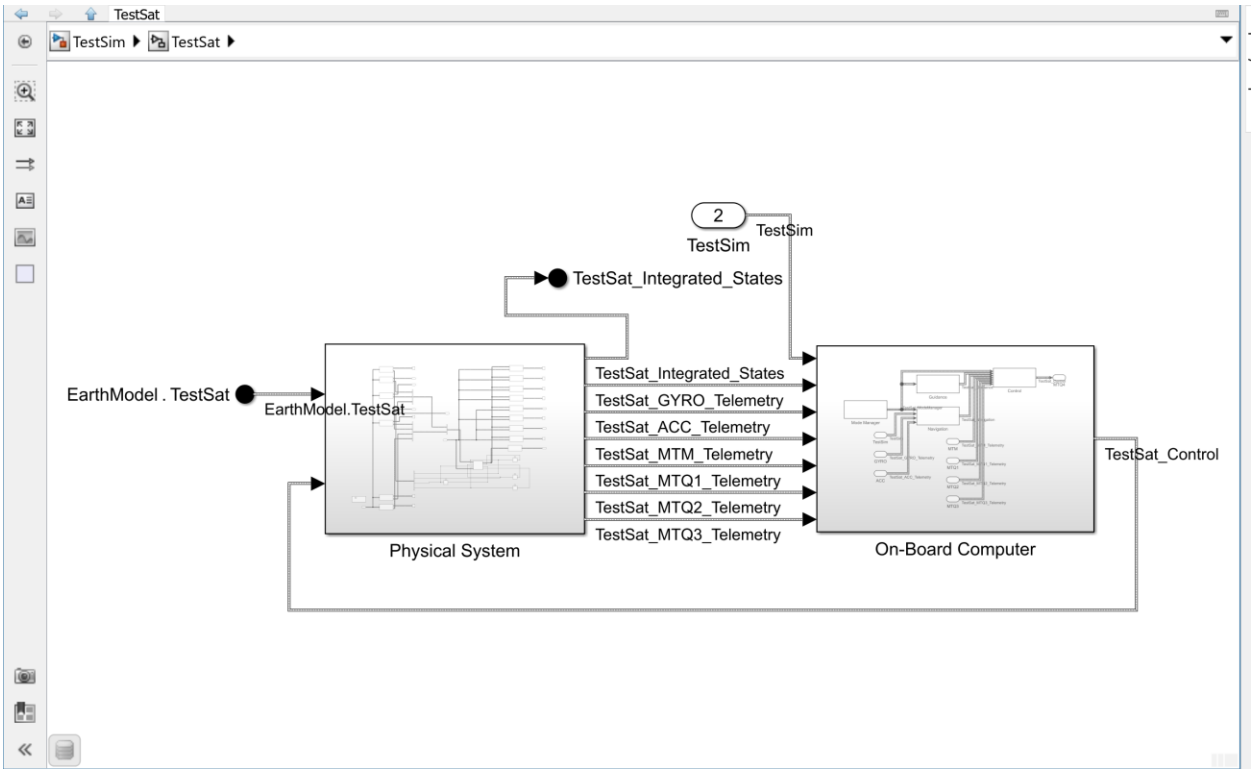


Figure 3.3 – GIMBALS Satellite overall Simulink representation.

As shown in Figure 3.3, the contents of Satellite Block are divided into the "Physical System" (also known as the plant model) and the "On-Board Computer." This precisely mirrors the distinction made in Figure 2.3 between the digital world in which the OBC algorithms exist, and the real world that the satellite body, sensors, and actuators interact with. At a given simulation timestep, the OBC receives hardware telemetry from the sensors and actuators and processes the data in the GNC algorithm loop to generate actuator commands. This process is illustrated by the contents of the "On-Board Computer" block:

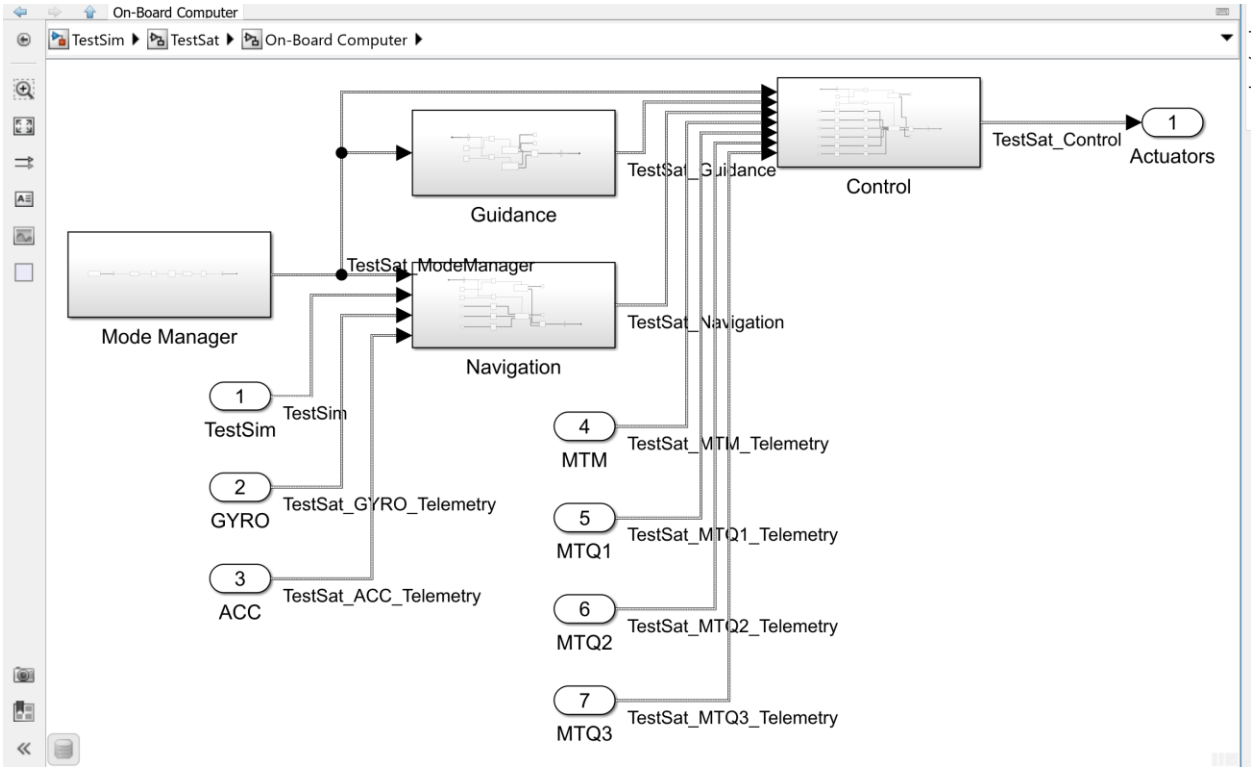


Figure 3.4 – GIMBALS Satellite on-board computer Simulink representation.

The commands are then sent to the actuators, which apply control forces/torques to the satellite. The satellite dynamic states will respond to the control forces/torques and to environmental forces/torques (from the EarthModel) accordingly. The sensors make measurements after the satellite state is updated, allowing the feedback process to commence. Figure 3.5 below showcases the contents of the “Physical System” block:

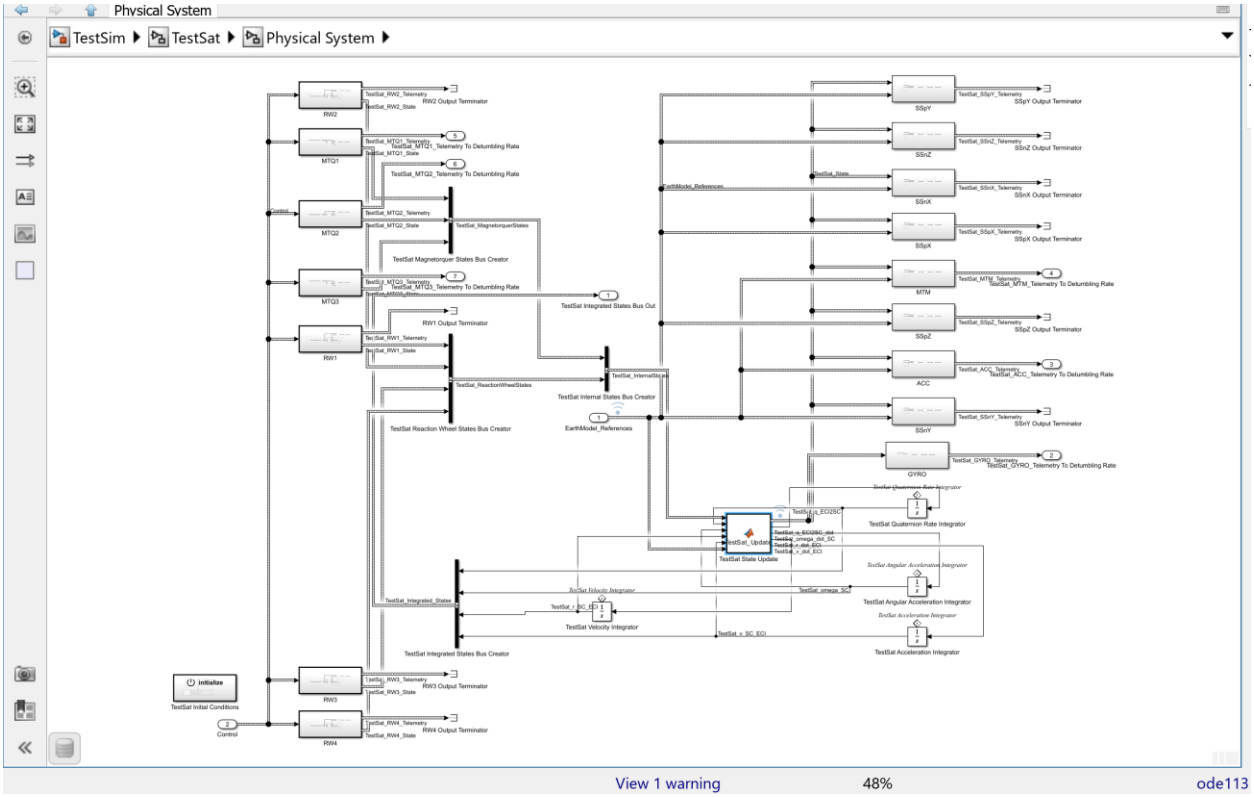


Figure 3.5 – GIMBALS Satellite physical pystem Simulink representation.

3.2.4 Sensor Objects

As described in section 2.3, satellites contain a variety of sensors to measure internal satellite states and external environment references. An object is defined in GIMBALS for each sensor described in section 2.3. Their .m Matlab Class definitions are located in the "sensors" folder within the project folder structure as shown in Figure A.4. The user may create high-fidelity models of each sensor by setting sensor object properties for noise, bias, resolution, alignment, and more. For conciseness, only the API reference for the Accelerometer object is shown below, as most of the other sensors have similar properties.

Table 3.4 – GIMBALS Accelerometer object API reference.

Constructor*	`acc = Accelerometer(Parent=sat, <name>=<value>)`		
Property**	**Units**	**Ref. Type**	**Description**
Parent	——	noref	Handle to parent Satellite object
Name	——	noref	Accelerometer name
ID	——	staticref	Accelerometer integer ID (optional user variable)
MountingEuler	deg	staticref	Satellite-to-accelerometer-frame

			mounting Euler angles (ZYX)
MountingDCM	——	staticref	Satellite-to-accelerometer-frame mounting Direction Cosine Matrix
MountingQuaternion	——	staticref	Satellite-to-accelerometer-frame mounting quaternion
MountingAxisAngle	deg (for angle)	staticref	Satellite-to-accelerometer-frame Axis-Angle Representation
MountingCOMLocation	m	staticref	Accelerometer mounting center of mass location (from Satellite body frame origin) in Satellite body frame coordinates
MisalignmentEuler	deg	staticref	Accelerometer misalignment error Euler angles (ZYX)
MisalignmentDCM	——	staticref	Accelerometer misalignment error Direction Cosine Matrix
MisalignmentQuaternion	——	staticref	Accelerometer misalignment error quaternion
MisalignmentAxisAngle	deg (for angle)	staticref	Accelerometer misalignment error Axis-Angle Representation
NonOrthogonality	——	staticref	Accelerometer non-orthogonality (cross-axis sensitivity) error matrix
ScaleFactors	——	staticref	Accelerometer axis scale factors
WhiteNoiseDensity	m/s^2/$\sqrt{Hz}$	staticref	Accelerometer white noise spectral density
SamplingRate	Hz	staticref	Accelerometer sampling rate
ConstantBias	m/s^2	staticref	Accelerometer constant bias offset

Resolution	m/s^2	staticref	Accelerometer measurement resolution (quantization)
Acceleration	m/s^2	dynamicref	Accelerometer acceleration telemetry

* **<name>=<value>** in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.
** Properties can be assigned using standard dot notation (**acc.<name> = <value>**).

The following screenshot illustrates the appearance of the Simulink model's Accelerometer block contents, which was programmatically generated by the Build method inside the Accelerometer object's Matlab Class definition.

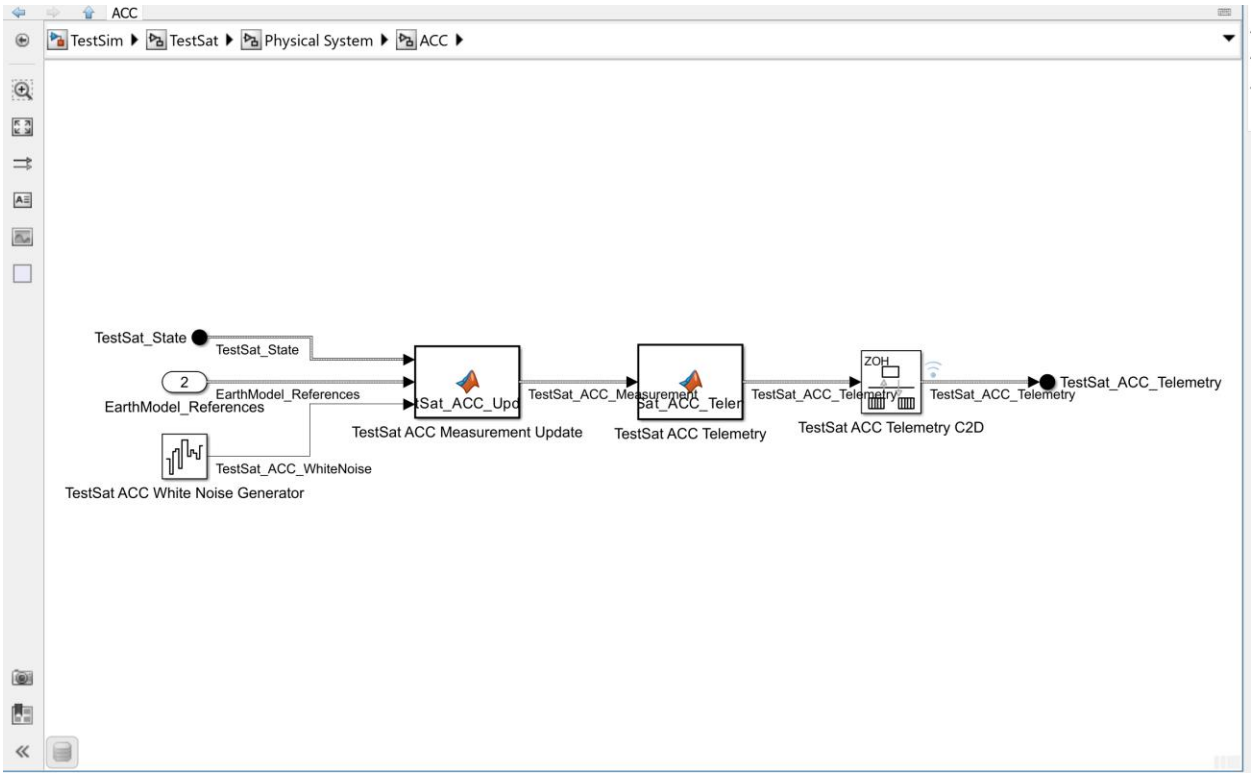


Figure 3.6 – GIMBALS Accelerometer Simulink representation.

All sensors modeled in GIMBALS have a Simulink representation like Figure 3.6. The sensor receives the current satellite dynamic state and/or EarthModel references if the sensor makes external environment measurements. In the case of the Accelerometer, the accelerometer model requires the gravitational force be subtracted per Equation 2.17, which necessitates the EarthModel connection to the measurement update Matlab Function Block. The Band-Limited White Noise Block generates zero-mean Gaussian noise for noise modeling additions to the sensor measurement. The Rate Transition Block outputs the sensor telemetry at the user-specified discrete sampling rate.

3.2.5 Actuator Objects

An object is defined in GIMBALS for two of the actuators described under section 2.4: magnetorquers and reaction wheels. Their .m Matlab Class definitions are located in the "actuators" folder within the project folder structure as shown in Figure A.7. The API reference for the Magnetorquer object and the ReactionWheel object are shown in the tables below:

Table 3.5 – GIMBALS Magnetorquer object API reference.

Constructor*	`mtq = Magnetorquer(Parent=sat, <name>=<value>)`		
Property**	**Units**	**Ref. Type**	**Description**
Parent	——	noref	Handle to parent Satellite object
Name	——	noref	Magnetorquer name
ID	——	staticref	Magnetorquer integer ID (optional user variable)
MountingEuler	deg	staticref	Satellite-to-magnetorquer-frame mounting Euler angles (ZYX)
MountingDCM	——	staticref	Satellite-to-magnetorquer-frame mounting Direction Cosine Matrix
MountingQuaternion	——	staticref	Satellite-to-magnetorquer-frame mounting quaternion
MountingAxisAngle	deg (for angle)	staticref	Satellite-to-magnetorquer-frame Axis-Angle Representation
NumberOfCoils	——	staticref	Magnetorquer number of coils
CrossSectionalArea	m^2	staticref	Magnetorquer cross-sectional area
CoreMagneticPermeability	——	staticref	Magnetorquer ferromagnetic core relative magnetic permeability (set equal to 1 for air core)
MaximumDipoleStrength	A-m^2	staticref	Magnetorquer maximum dipole magnitude (saturation limit)

TelemetryRate	Hz	staticref	Magnetorquer telemetry output rate
ControlSetpoint	A-m^2	dynamicvar	Magnetorquer magnetic dipole command (Z-axis is the positive dipole axis)
MagneticDipole	A-m^2	dynamicref	Magnetorquer magnetic dipole telemetry (Z-axis is the positive dipole axis)

* **<name>=<value>** in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.
** Properties can be assigned using standard dot notation (**mtq.<name> = <value>**).

Table 3.6 – GIMBALS ReactionWheel object API reference.

Constructor*	**rw = ReactionWheel(Parent=sat, <name>=<value>)**		
Property**	**Units**	**Ref. Type**	**Description**
Parent	——	noref	Handle to parent Satellite object
Name	——	noref	ReactionWheel name
ID	——	staticref	ReactionWheel integer ID (optional user variable)
MountingEuler	deg	staticref	Satellite-to-reaction-wheel-frame mounting Euler angles (ZYX)
MountingDCM	——	staticref	Satellite-to-reaction-wheel-frame mounting Direction Cosine Matrix
MountingQuaternion	——	staticref	Satellite-to-reaction-wheel-frame mounting quaternion
MountingAxisAngle	deg (for angle)	staticref	Satellite-to-reaction-wheel-frame Axis-Angle Representation
WheelMass	kg	staticref	Reaction Wheel mass
WheelInertiaTensor	kg-m^2	staticref	Reaction Wheel inertia tensor (Z-axis is the positive spin axis)

WheelCOMLocation	m	staticref	Reaction Wheel center of mass location (from Satellite body frame origin) in Satellite body frame coordinates
MotorTorqueConstant	(N-m)/A or (V-s)/rad	staticref	Reaction wheel DC motor torque/back-EMF constant
MotorRLCircuitImpedance	Ω	staticref	Reaction Wheel DC motor RL circuit electrical impedance (resistance)
MotorRLCircuitInductance	H	staticref	Reaction Wheel DC motor RL circuit electrical inductance
StaticFriction	N-m	staticref	Reaction Wheel static friction
CoulombFriction	N-m	staticref	Reaction Wheel Coulomb friction
ViscousFriction	N-m-s	staticref	Reaction Wheel viscous friction
StribeckSpeed	rad/s	staticref	Reaction Wheel Stribeck speed
SpeedUpperSaturationLimit	rad/s	staticref	Reaction Wheel speed upper saturation limit
SpeedLowerSaturationLimit	rad/s	staticref	Reaction Wheel speed lower saturation limit
InitialWheelSpeed	rad/s	staticref	Reaction Wheel initial speed (positive is CCW and negative is CW about RW frame Z-axis)
TelemetryRate	Hz	staticref	Reaction Wheel telemetry output rate
ControlMode	——	dynamicvar	Reaction Wheel control mode command (0 = speed, 1 = torque)
ControlSetpoint	N-m (torque mode) or rad/s (speed mode)	dynamicvar	Reaction Wheel speed/torque setpoint command (depending on control mode)

WheelSpeed	rad/s	dynamicref	Reaction Wheel speed telemetry
MotorCurrent	A	dynamicref	Reaction Wheel motor current telemetry

* **<name>=<value>** in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.
** Properties can be assigned using standard dot notation (**rw.<name> = <value>**).

The following screenshot illustrates the appearance of the Simulink model's Magnetorquer block contents, which was programmatically generated by the Build method inside the Magnetorquer object's Matlab Class definition.

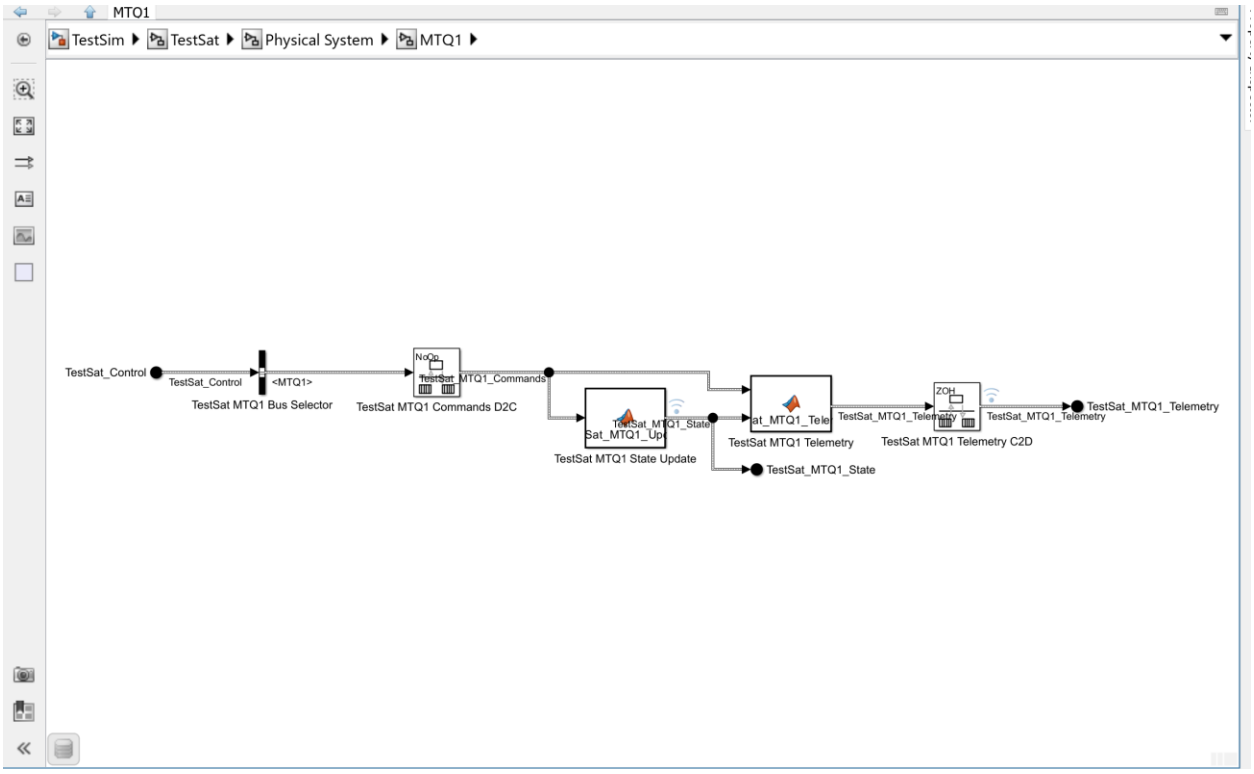


Figure 3.7 – GIMBALS Magnetorquer Simulink representation.

The magnetorquer receives a control signal from the user's control algorithm, updates its torque output in the State Update Matlab Function Block, and outputs its current magnetic dipole as telemetry. The Rate Transition block before the final output ensures that the data is being sent out at the user-specified discrete telemetry rate.

The following screenshot illustrates the appearance of the Simulink model's ReactionWheel block contents, which was programmatically generated by the Build method inside the ReactionWheel object's Matlab Class definition.

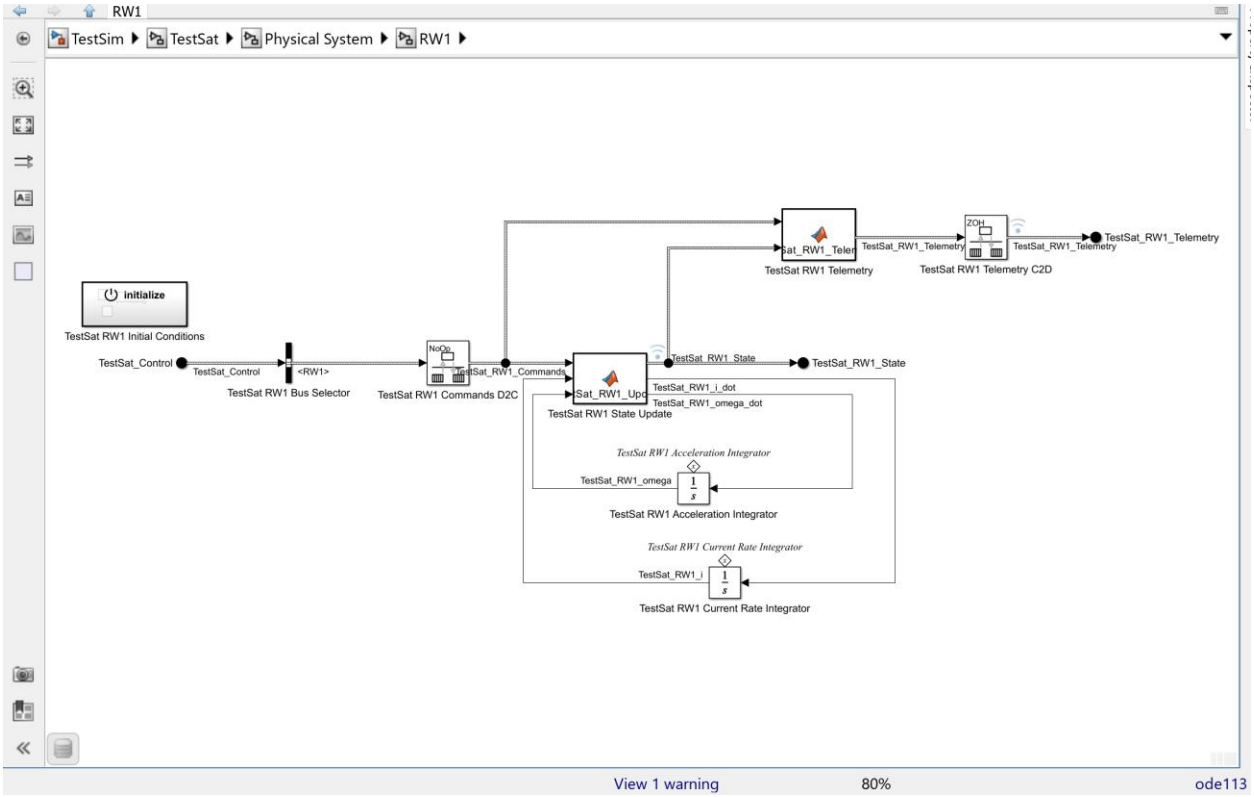


Figure 3.8 – GIMBALS ReactionWheel Simulink representation.

Compared to the Magnetorquer, Figure 3.8 highlights the added complexity of the ReactionWheel having dynamic states that need to be integrated. Both the Magnetorqer block and the ReactionWheel block have two outputs: the first is the telemetry, and the second is the state output that outputs any static information (i.e. mounting quaternions) and/or dynamic information (i.e. reaction wheel angular velocity) that is necessary for the Satellite's overall dynamics computation performed in its State Update Matlab Function Block (see Figure 3.5).

3.2.6 The Global Target Generator (GTG)

The On-Board Computer (OBC) contains the Guidance block as shown in Figure 3.4, which corresponds to a Guidance object. The associated .m Matlab Class definitions are located in the "algorithms" folder within the project folder structure as shown in Figure A.6. The API reference for the associated Matlab Class is provided in the table below.

Table 3.7 – GIMBALS Guidance object API reference.

Constructor*	`guidance = Guidance(Parent=sat, <name>=<value>)`		
Property**	**Units**	**Ref. Type**	**Description**
Parent	——	noref	Handle to parent Satellite object
Name	——	noref	Guidance object name

ID	——	staticref	Guidance object integer ID (optional user variable)
AttitudeTarget	——	dynamicref	Guidance object mode-selected GTG output

* **<name>=<value>** in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.
** Properties can be assigned using standard dot notation (**guidance.<name> = <value>**). Note: while the constructor can be used to create a Guidance object instance, the Satellite object uses its own internal Guidance object instance that it is already bound to. The user can edit the properties of this internal instance using nested dot notation (**sat.Guidance.<name> = <value>**).

In the context of the GIMBALS input file, the AttitudeTarget property of the Guidance object is a Global Target Generator (GTG) object. The GTG object has its own API reference as shown below:

Table 3.8 – GIMBALS GTG object API reference.

Constructor*	**gtg = GTG(Parent=sat.Guidance, <name>=<value>)**		
Property**	**Units**	**Ref. Type**	**Description**
Parent	——	noref	Handle to parent Guidance object
Name	——	noref	Global Target Generator name
ID	——	staticref	Global Target Generator integer ID (optional user variable)
Function	——	noref	Cell array of GTG function handle and input parameters: C = {@myGTGFcn, fixedArg1, fixedArg2, ... , fixedArgN}
LoopRate	Hz	staticref	Global Target Generator algorithm loop rate
Quaternion	——	dynamicvar	Global Target Generator reference attitude quaternion
AngularVelocity	rad/s	dynamicvar	Global Target Generator reference angular rate vector

* **<name>=<value>** in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.

** Properties can be assigned using standard dot notation (**`gtg.<name> = <value>`**).

The user defines a GTG object for each ADCSMode object they define in their GIMBALS input file and subsequently must use the GTG property of that ADCSMode to associate them with each other. The GTG object's Function property allows the user to specify a Matlab function (.m) file in the Matlab path as their global target generator method, as well as any parameter inputs to the function. These can be other GIMBALS objects or numeric constants. The aforementioned reference type restrictions apply for GIMBALS objects passed in as GTG Function parameters. The user must set the values of the Quaternion and AngularVelocity properties inside their GTG Function body.

The following screenshot illustrates the appearance of the Simulink model's Guidance block contents, which was programmatically generated by the Build method inside the Guidance object's Matlab Class definition.

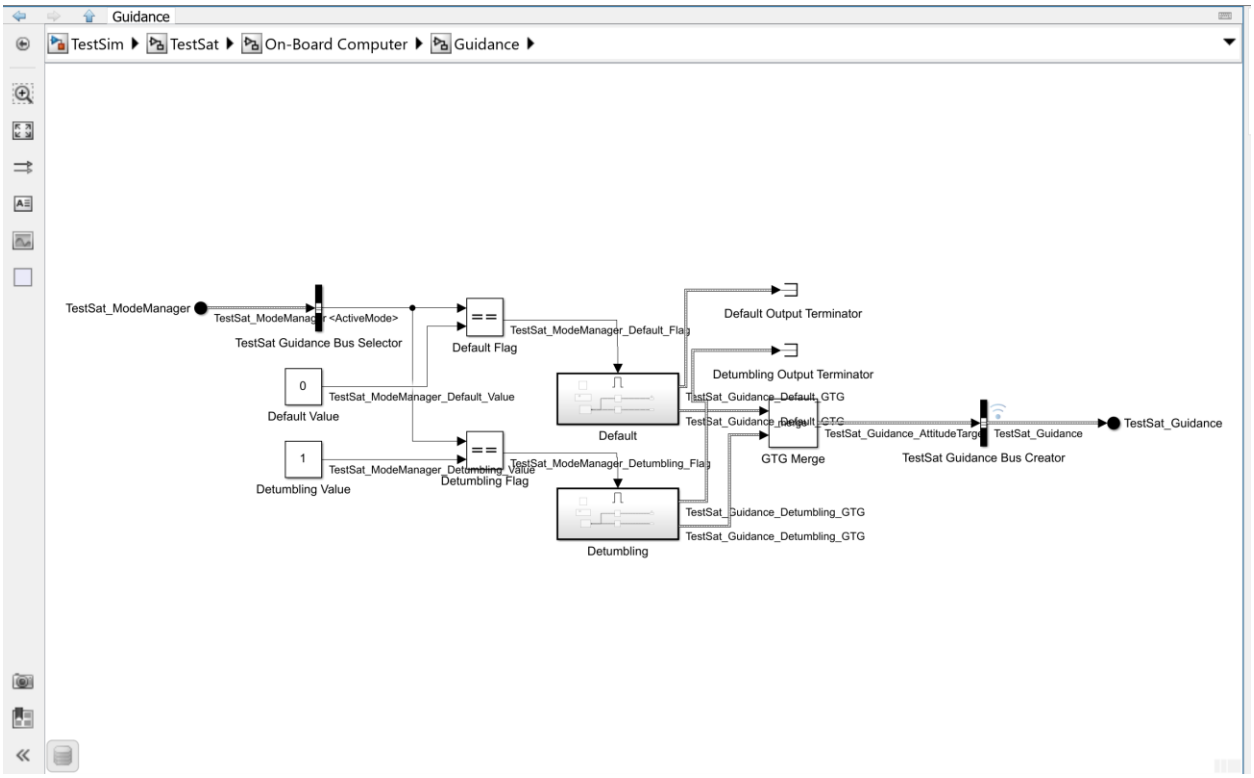


Figure 3.9 – GIMBALS Guidance Simulink representation.

The Guidance block compares the active mode ID with the user's defined modes to determine which GTG Function to run during the simulation; if the active mode is zero, the Default pathway in Figure 3.9 is taken. The following screenshot illustrates the appearance of the contents of the GTG called "Detumbling" in Figure 3.9, which was programmatically generated by the Build method inside the GTG object's Matlab Class definition.

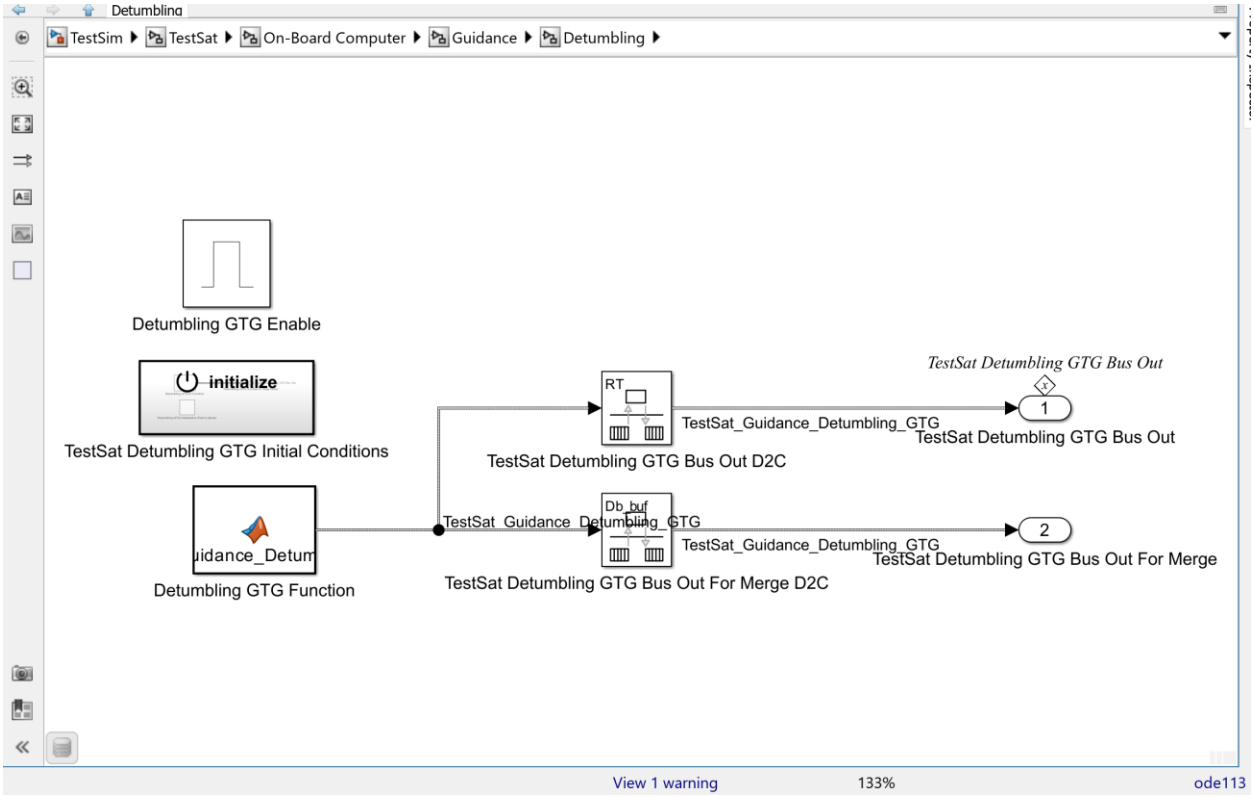


Figure 3.10 – GIMBALS GTG Simulink representation.

The GTG Function runs at the discrete loop rate specified by the user.

3.2.7 The Global State Estimator (GSE)

The On-Board Computer (OBC) contains the Navigation block as shown in Figure 3.4, which corresponds to a Navigation object. The associated .m Matlab Class definitions are located in the "algorithms" folder within the project folder structure as shown in Figure A.6. The API reference for the associated Matlab Class is provided in the table below.

Table 3.9 – GIMBALS Navigation object API reference.

Constructor*	`navigation = Navigation(Parent=sat, <name>=<value>)`		
Property**	**Units**	**Ref. Type**	**Description**
Parent	——	noref	Handle to parent Satellite object
Name	——	noref	Navigation object name
ID	——	staticref	Navigation object integer ID (optional user variable)
StateEstimate	——	dynamicref	Navigation object mode-selected GSE output

* **<name>=<value>** in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.
** Properties can be assigned using standard dot notation (**navigation.<name> = <value>**). Note: while the constructor can be used to create a Navigation object instance, the Satellite object uses its own internal Navigation object instance that it is already bound to. The user can edit the properties of this internal instance using nested dot notation (**sat.Navigation.<name> = <value>**).

In the context of the GIMBALS input file, the StateEstimate property of the Navigation object is a Global State Estimator (GSE) object. The GSE object has its own API reference as shown below:

Table 3.10 – GIMBALS GSE object API reference.

Constructor*	**gse = GSE(Parent=sat.Navigation, <name>=<value>)**		
Property**	**Units**	**Ref. Type**	**Description**
Parent	——	noref	Handle to parent Navigation object
Name	——	noref	Global State Estimator name
ID	——	staticref	Global State Estimator integer ID (optional user variable)
Function	——	noref	Cell array of GSE function handle and input parameters: C = {@myGSEFcn, fixedArg1, fixedArg2, ... , fixedArgN}
LoopRate	Hz	staticref	Global State Estimator algorithm loop rate
Quaternion	——	dynamicvar	Global State Estimator attitude quaternion estimate
AngularVelocity	rad/s	dynamicvar	Global State Estimator angular rate vector estimate
Position	m	dynamicvar	Global State Estimator orbital position estimate
Velocity	m/s	dynamicvar	Global State Estimator orbital velocity estimate

* **<name>=<value>** in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.
** Properties can be assigned using standard dot notation (**gtg.<name> = <value>**).

The user defines a GSE object for each ADCSMode object they define in their GIMBALS input file and subsequently must use the GSE property of that ADCSMode to associate them with each other. The GSE object's Function property allows the user to specify a Matlab function (.m) file in the Matlab path as their global state estimator method, as well as any parameter inputs to the function. These can be other GIMBALS objects or numeric constants. The aforementioned reference type restrictions apply for GIMBALS objects passed in as GSE Function parameters. The user must set the values of the Quaternion, AngularVelocity, Position, and Velocity properties inside their GSE Function body.

The following screenshot illustrates the appearance of the Simulink model's Navigation block contents, which was programmatically generated by the Build method inside the Navigation object's Matlab Class definition.

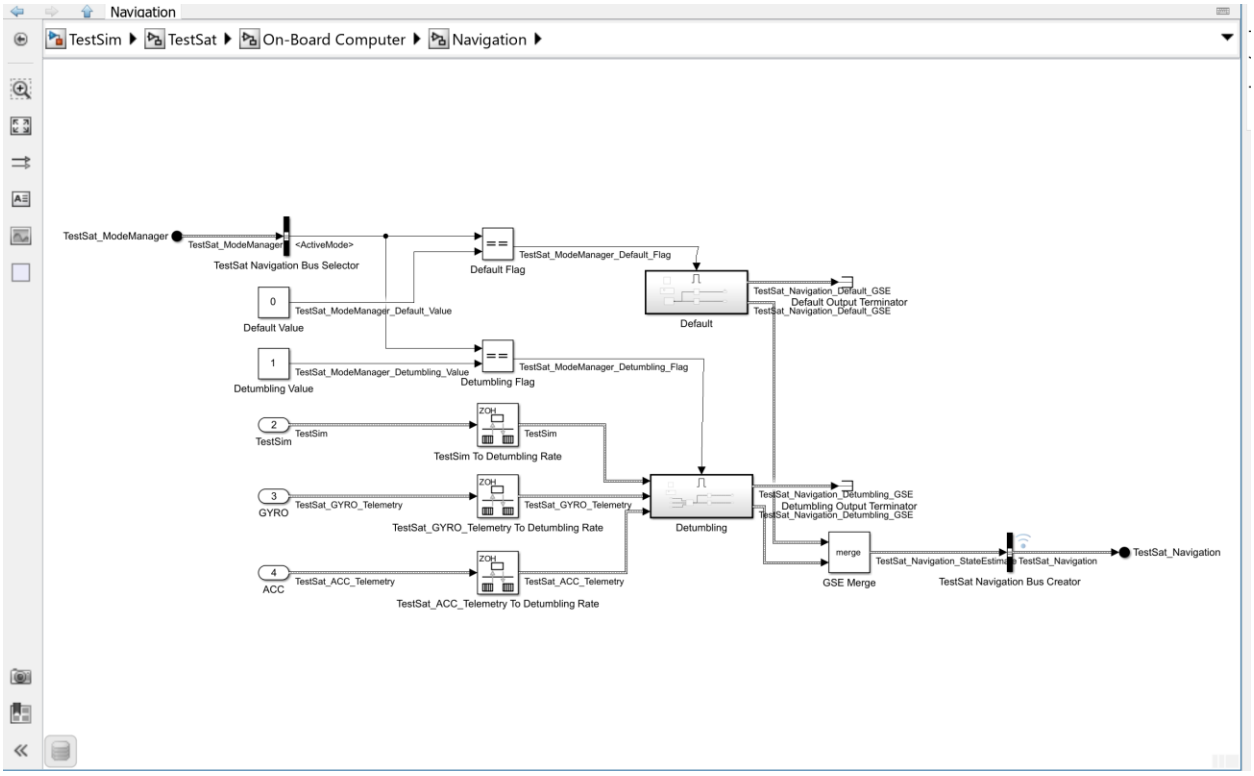


Figure 3.11 – GIMBALS Navigation Simulink representation.

The Navigation block compares the active mode ID with the user's defined modes to determine which GSE Function to run during the simulation; if the active mode is zero, the Default pathway in Figure 3.11 is taken. The following screenshot illustrates the appearance of the contents of the GSE called "Detumbling" in Figure 3.11, which was programmatically generated by the Build method inside the GSE object's Matlab Class definition.

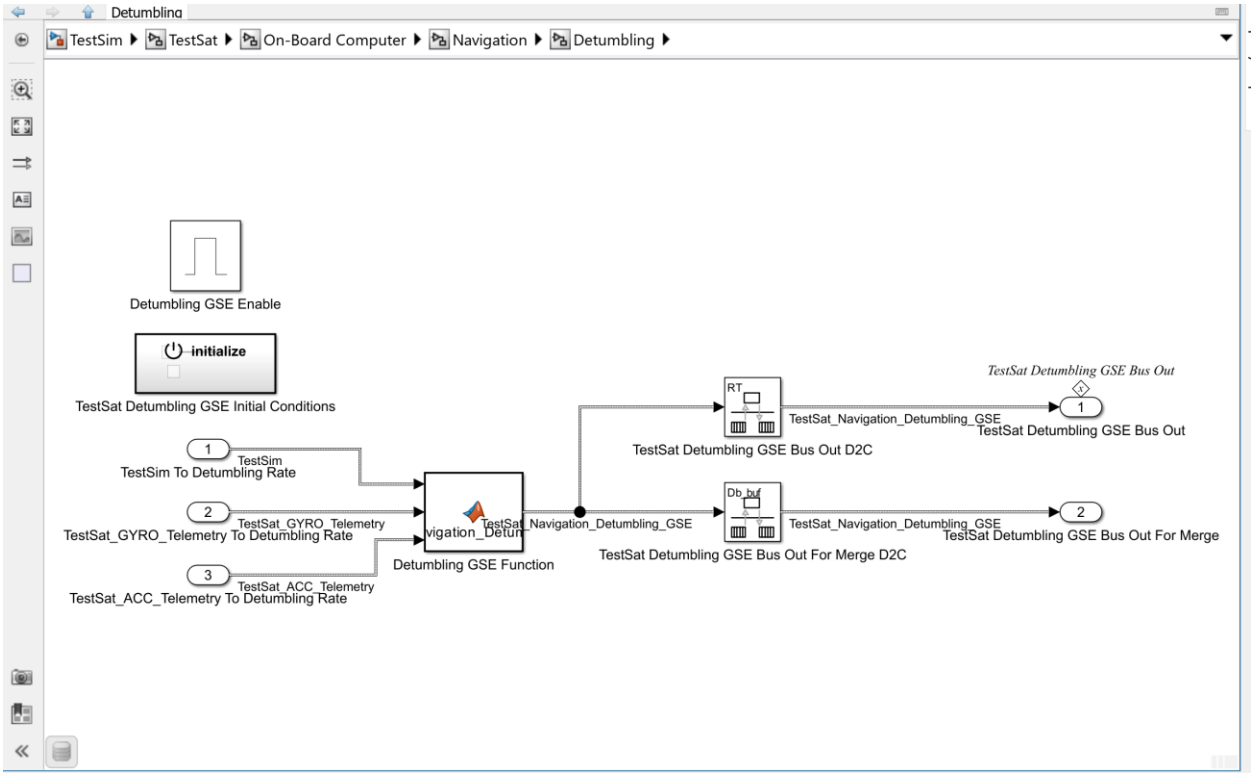


Figure 3.12 – GIMBALS GSE Simulink representation.

The GSE Function runs at the discrete loop rate specified by the user.

3.2.8 The Actuator Command Generator (ACG)

The On-Board Computer (OBC) contains the Control block as shown in Figure 3.4, which corresponds to a Control object. The associated .m Matlab Class definitions are located in the "algorithms" folder within the project folder structure as shown in Figure A.6. The API reference for the associated Matlab Class is provided in the table below.

Table 3.11 – GIMBALS Control object API reference.

Constructor*	`control = Control(Parent=sat, <name>=<value>)`		
Property**	**Units**	**Ref. Type**	**Description**
Parent	——	noref	Handle to parent Satellite object
Name	——	noref	Control object name
ID	——	staticref	Control object integer ID (optional user variable)
ActuatorCommands	——	dynamicref	Control object mode-selected ACG output

* `<name>=<value>` in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.

** Properties can be assigned using standard dot notation (`control.<name> = <value>`). Note: while the constructor can be used to create a Control object instance, the Satellite object uses its own internal Control object instance that it is already bound to. The user can edit the properties of this internal instance using nested dot notation (`sat.Control.<name> = <value>`).

In the context of the GIMBALS input file, the ActuatorCommands property of the Navigation object is an Actuator Command Generator (ACG) object. The ACG object has its own API reference as shown below:

Table 3.12 – GIMBALS ACG object API reference.

Constructor*	`acg = ACG(Parent=sat.Control, <name>=<value>)`		
Property**	**Units**	**Ref. Type**	**Description**
Parent	——	noref	Handle to parent Control object
Name	——	noref	Actuator Command Generator name
ID	——	staticref	Actuator Command Generator integer ID (optional user variable)
Function	——	noref	Cell array of ACG function handle and input parameters: C = {@myACGFcn, fixedArg1, fixedArg2, ... , fixedArgN}
LoopRate	Hz	staticref	Actuator Command Generator algorithm loop rate
<Actuator(1)_Name>	——	dynamicvar	Actuator Command Generator name-referenced actuator #1 in Parent Satellite
<Actuator(2)_Name>	——	dynamicvar	Actuator Command Generator name-referenced actuator 2 in Parent Satellite
. . .	. . .	. . .	. . .
<Actuator(k)_Name>	——	dynamicvar	Actuator Command Generator name-referenced actuator k in Parent Satellite

* **<name>=<value>** in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.
** Properties can be assigned using standard dot notation (**acg.<name> = <value>**).

The user defines an ACG object for each ADCSMode object they define in their GIMBALS input file and subsequently must use the ACG property of that ADCSMode to associate them with each other. The ACG object's Function property allows the user to specify a Matlab function (.m) file in the Matlab path as their actuator command generator method, as well as any parameter inputs to the function. These can be other GIMBALS objects or numeric constants. The aforementioned reference type restrictions apply for GIMBALS objects passed in as ACG Function parameters. The <Actuator(k)_Name> property listed in Table 3.10 is simply the Name property of actuators 1 through k that the user defined in their GIMBALS input file. Within the scope of the ACG Function body, these name-referenced actuator properties become stand-ins for the actual actuator objects themselves, allowing commands to properly be routed to the correct actuators. The user must set the properties with reference type "dynamicvar" for each actuator via this actuator command routing approach.

The following screenshot illustrates the appearance of the Simulink model's Control block contents, which was programmatically generated by the Build method inside the Control object's Matlab Class definition.

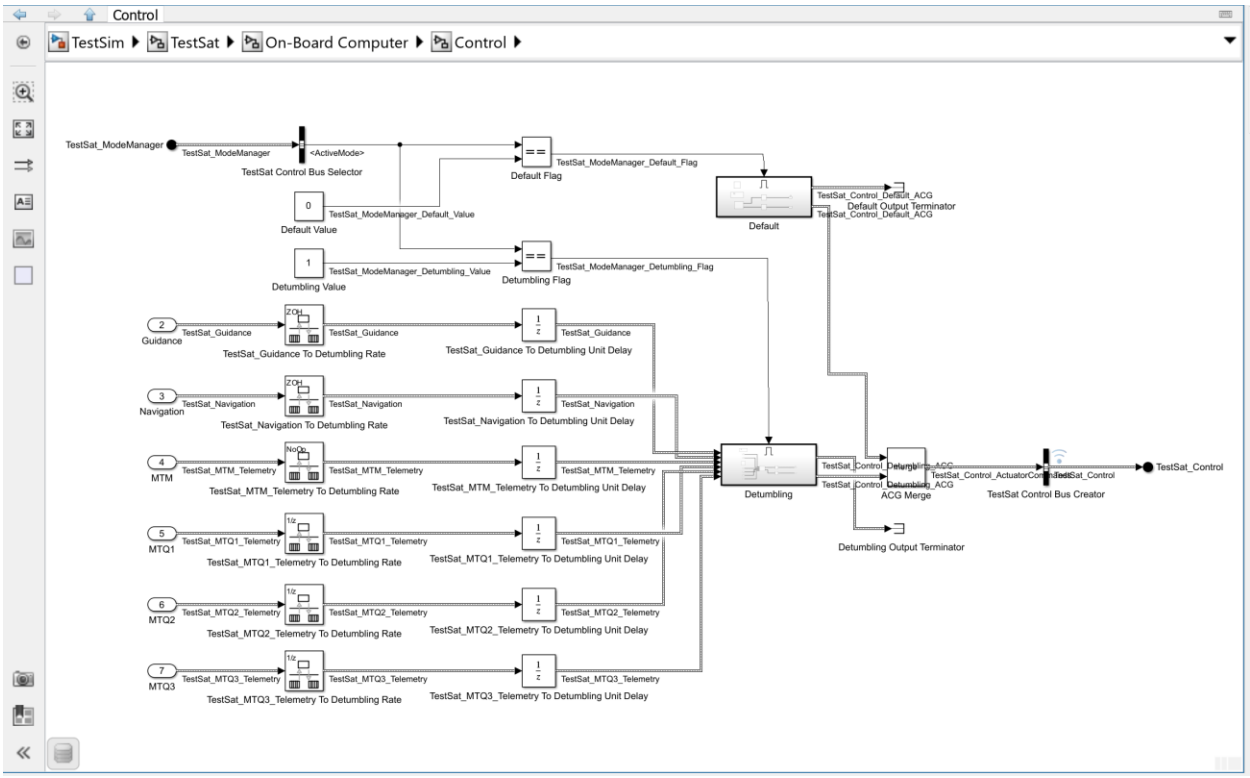


Figure 3.13 – GIMBALS Control Simulink representation.

The Control block compares the active mode ID with the user's defined modes to determine which ACG Function to run during the simulation; if the active mode is zero, the Default pathway in Figure 3.13 is taken. The following screenshot illustrates the appearance of the contents of the

ACG called “Detumbling” in Figure 3.13, which was programmatically generated by the Build method inside the ACG object’s Matlab Class definition.

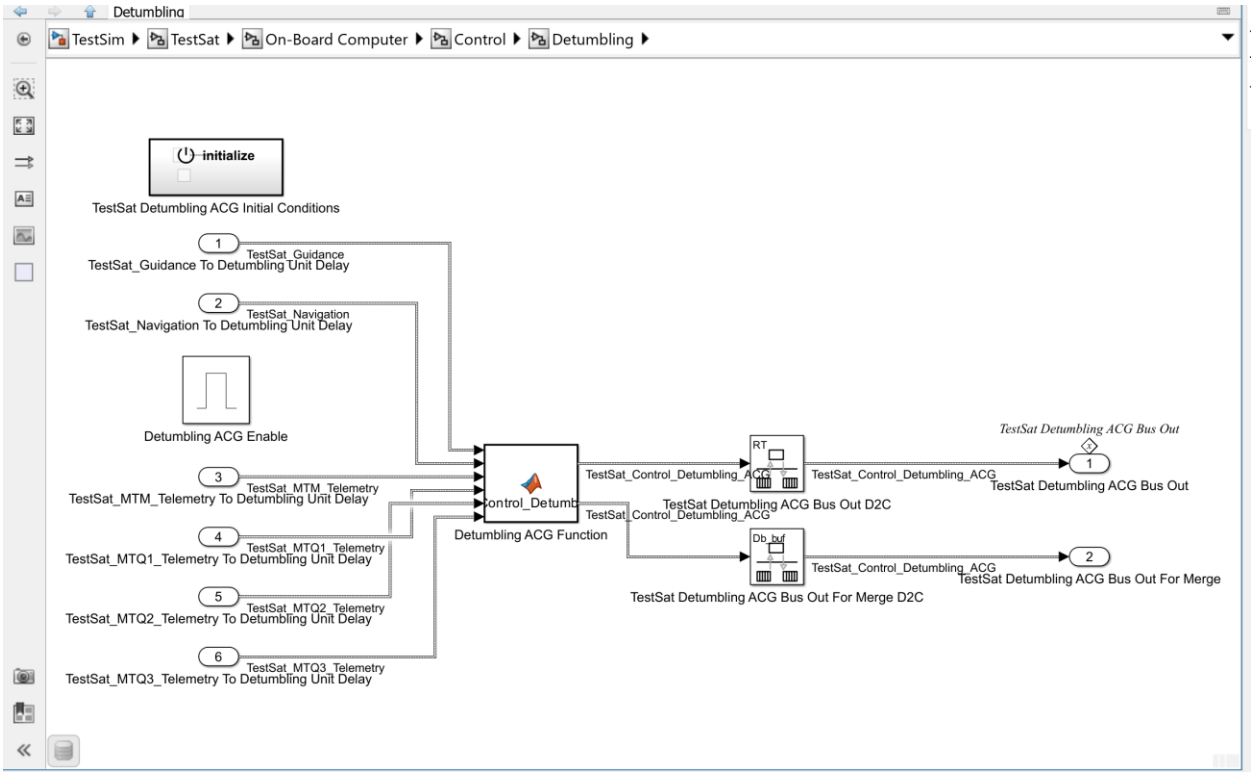


Figure 3.14 – GIMBALS ACG Simulink representation.

The ACG Function runs at the discrete loop rate specified by the user.

3.2.9 The Mode Manager

The ModeManager Object is defined in the ModeManager.m Matlab Class. It is located in the “algorithms” folder within the project folder structure as shown in Figure A.6. The API reference for the ModeManager object is provided in the table below:

Table 3.13 – GIMBALS ModeManager object API reference.

Constructor*	`modeManager = ModeManager(Parent=sat.Control, <name>=<value>)`		
Property**	**Units**	**Ref. Type**	**Description**
Parent	——	noref	Handle to parent Satellite object
Name	——	noref	Mode Manager name
ID	——	staticref	Mode Manager integer ID (optional user variable)

StateflowLoopRate	Hz	staticref	Mode Manager Stateflow chart loop rate
StateflowInitialMode	——	noref	Mode Manager Stateflow chart initial ADCS mode
StateflowVariables	——	noref	Mode Manager Stateflow chart input and local variables/names
StateflowDisplay	——	noref	Mode Manager Stateflow logic display option
MCS	——	noref	Mode Manager Mode Command Sequencer
ActiveMode	——	dynamicref	Mode Manager ID of active ADCS mode

* **<name>=<value>** in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.
** Properties can be assigned using standard dot notation (**modeManager.<name> = <value>**). Note: while the constructor can be used to create a ModeManager object instance, the Satellite object uses its own internal ModeManager object instance that it is already bound to. The user can edit the properties of this internal instance using nested dot notation (**sat.ModeManager.<name> = <value>**).

The modes themselves are defined using the ADCSMode object defined in ADCSMode.m Matlab Class. It is located in the "algorithms" folder within the project folder structure as shown in Figure A.6. This object allows the user to specify satellite modes and transitions according to the state machine philosophy described in section 2.5.1. The API reference for the ADCSMode object is provided in the table below:

Table 3.14 – GIMBALS ADCSMode object API reference.

Constructor*	**adcsMode = ADCSMode(Parent=sat.ModeManager, <name>=<value>)**		
Property**	**Units**	**Ref. Type**	**Description**
Parent	——	noref	Handle to parent ModeManager object
Name	——	noref	ADCS Mode name
ID	——	staticref	ADCS Mode integer ID (optional user variable)
EntryAction	——	noref	ADCS mode entry action code
DuringAction	——	noref	ADCS mode during action code

ExitAction	——	noref	ADCS mode exit action code
Transitions	——	noref	ADCS mode transitions to other modes: C = [{TargetMode1, '[criteria1]{action1}'}; {TargetMode2, '[criteria2]{action2}'}; ... ; {TargetModeN, '[criteriaN]{actionN}'}]
GTG	——	noref	ADCS mode Global Target Generator
GSE	——	noref	ADCS mode Global State Estimator
ACG	——	noref	ADCS mode Actuator Command Generator

* **<name>=<value>** in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.
** Properties can be assigned using standard dot notation (**adcsMode.<name> = <value>**).

The following screenshot illustrates the appearance of the Simulink model's ModeManager block contents, which was programmatically generated by the Build method inside the ModeManager object's Matlab Class definition.

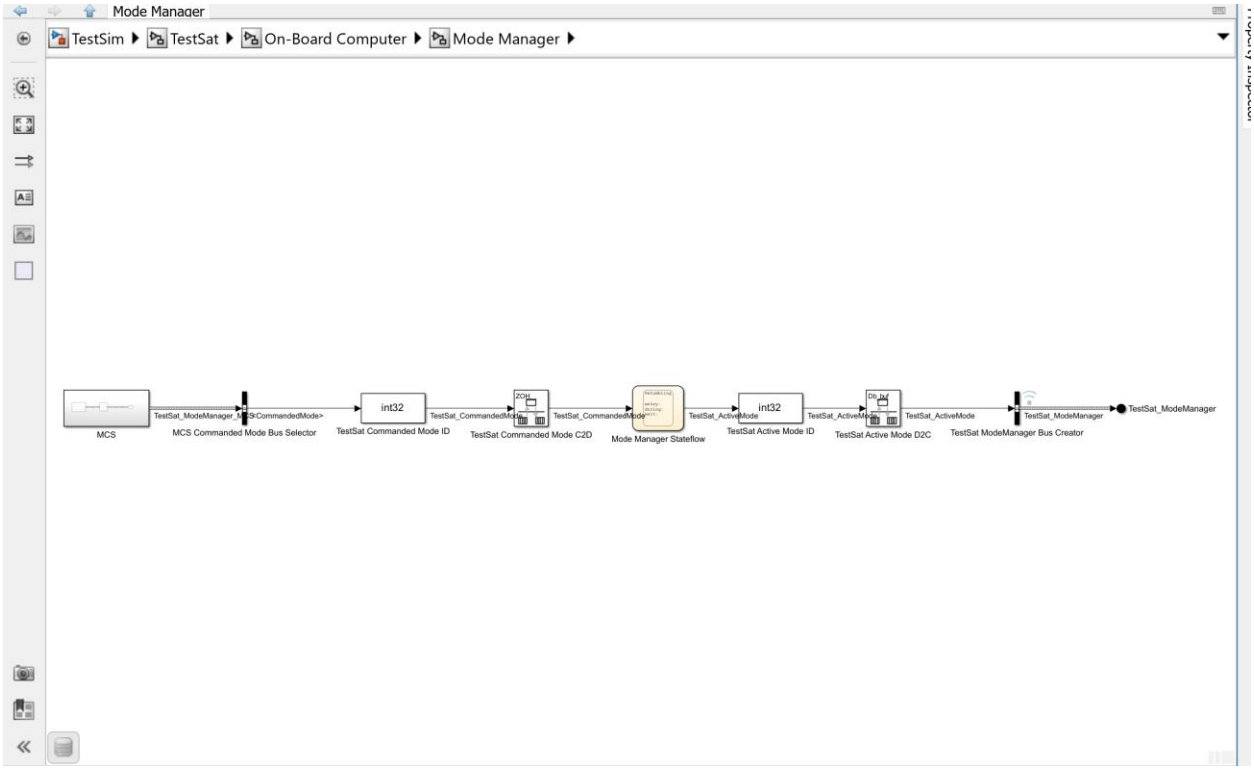


Figure 3.15 – GIMBALS ModeManager Simulink representation.

GIMBALS uses Stateflow to model the state machine logic defined by the user via the ModeManager object.

Within the ModeManager block shown in Figure 3.15, the mode command sequencer (MCS) allows the user to command modes to the Stateflow chart. The MCS object is defined with a similar philosophy to the GTG, GSE, and ACG objects. The user defines their mode command sequencer algorithm in a standalone Matlab function (.m) file on the Matlab path and sets the Function property of the MCS object to the handle of this function, plus parameter inputs. The API reference for the MCS object is provided in the table below:

Table 3.15 – GIMBALS MCS object API reference.

Constructor*	`mcs = MCS(Parent=sat.ModeManager, <name>=<value>)`		
Property**	**Units**	**Ref. Type**	**Description**
Parent	——	noref	Handle to parent ModeManager object
Name	——	noref	Mode Command Sequencer name
ID	——	staticref	Mode Command Sequencer integer ID (optional user variable)
Function	——	noref	Cell array of MCS function handle and

			input parameters: C = {@myMCSFcn, fixedArg1, fixedArg2, ... , fixedArgN}
LoopRate	Hz	staticref	Mode Command Sequencer algorithm loop rate
CommandedMode	——	dynamicvar	Mode Command Sequencer ID of commanded mode

* **<name>=<value>** in the constructor indicates that property names can optionally be set inside the constructor call with arguments using Matlab's key value pair syntax.
** Properties can be assigned using standard dot notation (**mcs.<name> = <value>**).

The following screenshot illustrates the appearance of the Simulink model's MCS block contents, which was programmatically generated by the Build method inside the MCS object's Matlab Class definition.

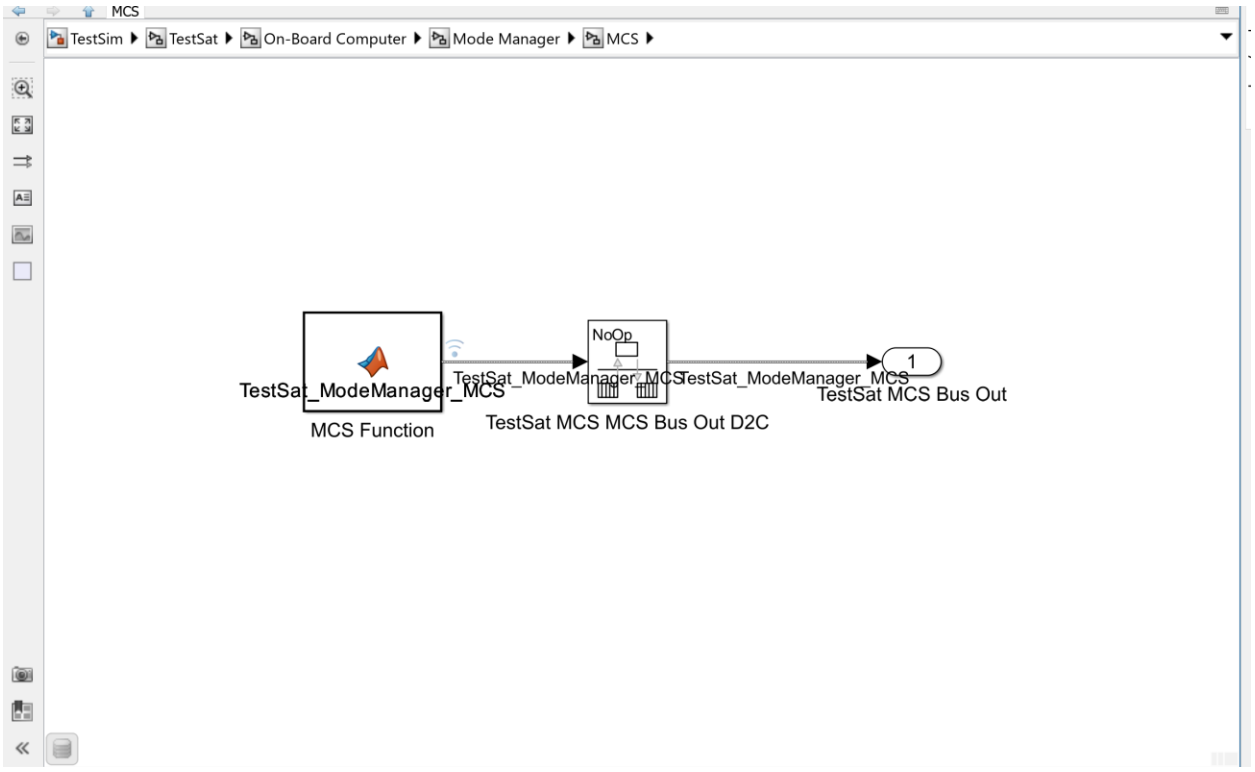


Figure 3.16 – GIMBALS MCS Simulink representation.

The MCS Function runs at the discrete loop rate specified by the user.

3.3 Software Requirements

To successfully run GIMBALS, the following Matlab software and dependencies are required:

- Matlab + Simulink R2025b (later versions may work)
- Aerospace Toolbox
- Aerospace Blockset
- Ephemeris Data for Aerospace Toolbox

When the software is first opened in a new Matlab session, the folders shown in Figure A.1 may need to be added to the Matlab path before they can be used. The user can run GIMBALS.m to add the relevant folders to the path.

Chapter 4 – Testing and Validation

Testing and Validation of the Guidance, navigation, and control Matlab-Based Architecture for LEO satellite Simulations (GIMBALS) was done using the sample input in Appendix B. Due to significant time constraints near the close of this project, a relatively simple scenario was chosen to test the software: a 2U CubeSat in sun-synchronous (SSO) LEO undergoing magnetic detumbling. The actuator hardware on the CubeSat includes 4 parked reaction wheels in a pyramid configuration along with 3 magnetorquers fixed along the satellite body frame axes. The sensor hardware includes the standard IMU sensor suite (a magnetometer, a gyroscope, and an accelerometer) as well as 6 sun sensors on each face of the satellite, each with 120° FOV. The reaction wheels and sun sensors are unused during the detumbling mode and were only included for completeness of the architecture definition. The objective of detumbling is to drive the CubeSat's angular velocity to zero along all axes. Therefore, the Global Target Generator (GTG) function was implemented as shown in Appendix C. Because the satellite does not begin attitude determination and control tasks until after detumbling, quaternion targets were not considered. An Extended Kalman Filter (EKF) was developed for the Global State Estimator (GSE) function; its implementation is shown in Appendix D. Finally, a B-dot controller was utilized inside the Actuator Command Generator (ACG) function; it is shown in Appendix E.

Running the GIMBALS input file generated no errors and the entire Simulink model build process performed by GIMBALS completed in under 30 seconds. In addition to the Simulink model (.slx), GIMBALS also generates a Simulink Data Dictionary (.sldd) to store user input information in a form that is compatible with usage in Simulink (i.e. Matlab structs, Simulink Buses, etc.). The GIMBALS input file was run from inside the "TestSat" project subfolder inside the "userdata" folder as shown in Figure A.2. As expected, the Simulink model and data dictionary were generated inside the "TestSim" folder inside "TestSat." The generated Simulink model is independent of the GIMBALS object Matlab Class files as well as the user's input file. The only dependencies are the GNC algorithm files (GTG, GSE, etc.), which must be on the Matlab path with the Simulink model and data dictionary.

TestSim.slx was opened and was compiled successfully after clicking the "Simulate model." The model successfully completed simulating the scenario over the 3 hour simulation timespan using ode113 (variable-step, variable-order Adams-Bashforth-Moulton) as the solver; from start to completion, the model took approximately 10 minutes to complete. The simulation output was created in the Matlab base workspace with the variable name "out." The logged data was extracted from the simulation output and plotted.

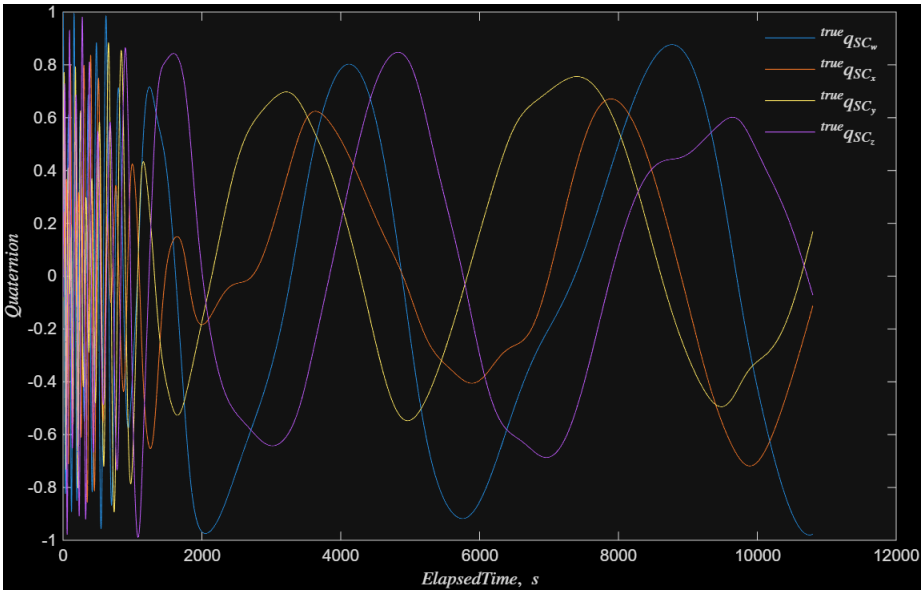


Figure 4.1 – True satellite quaternion states for GIMBALS magnetic detumbling test case.

The true quaternion states are shown in Figure 4.1. The plot indicates high quaternion rates of change early in the simulation with the rates noticeably decreasing to slow periodics around 2000 s, suggesting that successful detumbling occurred. To confirm successful detumbling, the CubeSat's true body frame angular rates are also plotted:

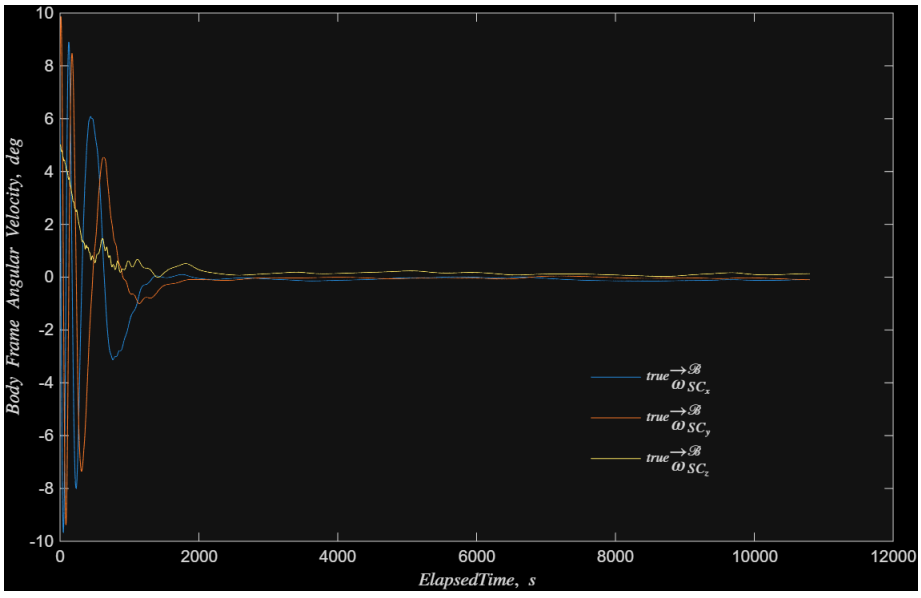


Figure 4.2 – True satellite body angular rates for GIMBALS magnetic detumbling test case.

Figure 4.2 confirms that the 3-axis magnetorquers were able to detumble the satellite from initial tipoff rates of $\vec{\omega}^{\mathcal{B}}_{SC\ init} = [6, 8, 5]^T$ deg/s after about 2000 s, with the rates falling below 0.5 deg/s about each axis after about 1800 s. The EKF's estimates of the angular rates can also be plotted for comparison:

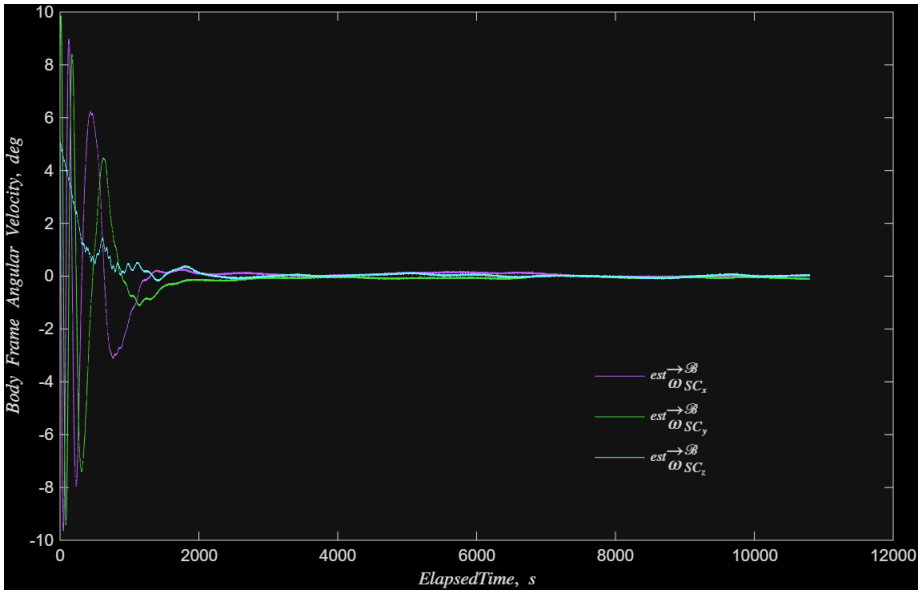


Figure 4.3 – EKF estimated satellite body angular rates for GIMBALS magnetic detumbling test case.

Figure 4.3 appears to show close correlation with Figure 4.2. Overlaying the two plots results in:

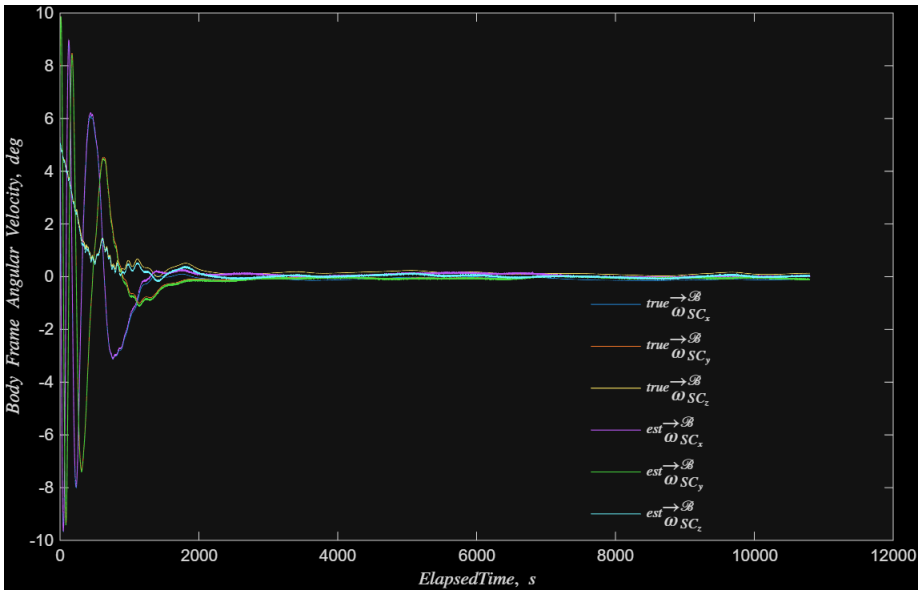


Figure 4.4 – True satellite body angular rates overlayed with EKF estimated body angular rates for GIMBALS magnetic detumbling test case.

Figure 4.4 confirms satisfactory performance from the EKF for estimating the satellite's body frame angular rates, and satisfactory performance from the B-dot controller for detumbling the CubeSat.

Chapter 5 – Conclusion

This project successfully developed the Guidance, navigation, and control Matlab-Based Architecture for LEO satellite Simulations (GIMBALS), a modular six degree-of-freedom GNC simulation framework. It provides an intuitive, high-fidelity verification tool for modeling low Earth orbiting small satellites. By utilizing Matlab/Simulink, the architecture eliminates much of the "black box" limitations of commercial software and current research-grade software. It ensures that future GNC engineers can easily interpret, modify, and validate custom flight code. It also provides current GNC engineers with a useful tool to quickly formulate and simulate multiple designs for trade analyses, or to quickly iterate on a baseline design. Development of this framework involved conceptual and practical integration of concepts involving translational and rotational rigid body dynamics, space environment modeling, mathematical modeling of sensors and actuators, and flight software algorithms. Testing successfully verified the framework's core algorithms through a magnetic detumbling test case. While GIMBALS is theoretically capable of handling much more complicated scenarios, including multiple commanded ADCS modes, FDIR implementations in the Mode Manager Stateflow chart, and target pointing, time limitations prevented creation and assessment of these scenarios. The final Matlab codebase for GIMBALS contains slightly more than 15,000 individual lines of code and will be provided to SJSU as a deliverable accompanying this report.

To meet the deadlines required for the submission of this project, several compromises had to be made which may serve as good starting points for further development of GIMBALS in the future. For example, thruster and CMG model implementations and the subsequent necessity to include mass and inertia dynamics modeling into the satellite rigid body dynamics model may be pursued. Improvements to the modeling of sensor/actuator hardware interfaces may also be considered. For example, the current sun sensor models in GIMBALS outputs the normalized sun vector and an eclipse fraction, whereas a real sun sensor might output current signals that need to be resolved into a sun vector by the flight software. Further ability to customize Simulink solver settings could be added. Alternative programming language interfaces could be implemented (i.e. the user would implement the GTG, GSE, ACG, and MCS algorithms in C/C++, Python, or other language). Artificial Intelligence (AI) compatibility could be incorporated. Comparison with existing tools like 42 and Basilisk may aid the ideation process for these future efforts.

References

[1] Mares, J.R., "SJSU Virtual Reality, View from a Cube Satellite," April 2019, URL: http://mstl.atl.calpoly.edu/~workshop/archive/2019/Spring/Day%203/Session%204/JesusRosilaMares.pdf.

[2] Westfall, A.J., "Design of an Attitude Control System for Spin-Axis Control of a 3U CubeSat," *SJSU ScholarWorks*, San Jose State University, December 2015, URL: https://doi.org/10.31979/etd.u6fr-6t4m.

[3] Lundquist, C., "Spatial Orientation of Explorer Satellites," *Space Journal*, Vol. 1, No. 3, Article 6, 1958, URL: https://louis.uah.edu/space-journal/vol1/iss3/6.

[4] "Syncom Engineering Report – Volume II," NASA TR-R-252, April 1967, URL: https://ntrs.nasa.gov/api/citations/19670015715/downloads/19670015715.pdf.

[5] Paluszek, M., *Spacecraft Attitude Determination and Control*, Elsevier, Cambridge, MA, 2023, URL: https://www.sciencedirect.com/book/9780323999151/adcs-spacecraftattitude-determination-and-control.

[6] Wales, R.O., "ATS-6 Final Engineering Report Performance Report Volume II – Orbits and Attitude Controls," NASA RP-1080-VOL-2, November 1981, URL: https://ntrs.nasa.gov/api/citations/19820008275/downloads/19820008275.pdf.

[7] Wie, B., *Space Vehicle Dynamics and Control Second Edition*, AIAA, Reston, VA, 2008.

[8] Noonan, C.H., McIntosh, R.J., Rowe, J.N., DeFazio, R.L., Galal, K.F., "Geostationary Operational Environmental Satellite (GOES)-8 Mission Flight Experience," NASA N9527798, 1995, URL: https://ntrs.nasa.gov/api/citations/19950021377/downloads/19950021377.pdf.

[9] El Wafi, I., Haloua, M., Guennoun, Z., Moudden, Z., "A framework for developing an attitude determination and control simulator for CubeSats: Processor-in-loop testing approach," *Results in Engineering*, Vol. 22, June 2024, URL: https://www.sciencedirect.com/science/article/pii/S2590123024004559.

[10] Stevenson, T., "Flight Results and Lessons Learned form the Starling Propulsion System," *Proceedings of the 38th Annual Small Satellite Conference*, SSC24-VII-01, Logan, UT, August 2024, URL: https://ntrs.nasa.gov/api/citations/20240007110/downloads/Stevenson_Smallsat_2024_v5.pdf.

[11] Reynolds, T.P., Kaycee, K., Barzgaran, B., Hudoba de Badyn, M., Rice, S., Hansen, E., Adler, A., Acikmese, B., and Mesbahi, M., "Development of a Generic Guidance Navigation & Control System for Small Satellites: Application to HuskySat-1," AIAA Space Forum, Orlando, FL, 2018, URL: https://s3vi.ndc.nasa.gov/ssri-kb/static/resources/main.pdf.

[12] Kenneally, P.W., Piggott, S., and Schaub, H., "Basilisk: A Flexible, Scalable and Modular Astrodynamics Simulation Framework," *Journal of Aerospace Information Systems*, Vol. 17, No. 9, May 2020, URL: https://doi.org/10.2514/1.I010762.

[13] Stoneking, E., “An Open-Source Simulation Tool for Study and Design of Spacecraft Attitude Control Systems,” NASA Goddard Space Flight Center, Greenbelt, MD, February 2018, URL: https://ntrs.nasa.gov/api/citations/20180000954/downloads/20180000954.pdf.

[14] Penn, J.M. and Lin, A.S., “The Trick Simulation Toolkit: A NASA/Open source Framework for Running Time Based Physics Models,” AIAA Modeling and Simulation Technologies Conference, San Diego, CA, January 2016, URL: https://doi.org/10.2514/6.2016-1187.

[15] Pesce, V., Colagrossi, A., and Silvestrini, S., *Modern Spacecraft Guidance, Navigation, and Control*, Elsevier, Cambridge, MA, 2023.

[16] Kuwahara, T., “Introduction to CubeSat Attitude Control System,” Department of Aerospace Engineering, Tohoku University, 2021, URL: https://www.unoosa.org/documents/pdf/psa/access2space4all/KiboCUBE/AcademySeason2/On-demand_Pre-recorded_Lectures/KiboCUBE_Academy_2021_OPL14.pdf.

[17] Cortiella, A., Vidal, D., Jané, J., Juan, E., Olivé, R., Amézaga, A., Munoz, J.F., Carreno-Luengo, P.V.H., and Camps, A., “CAT-2: Attitude Determination and Control System for a GNSS-R Earth Observation 6U CubeSat Mission,” *European Journal of Remote Sensing*, Vol. 49, No. 1, pp. 759–776, Sept. 2016, URL: https://doi.org/10.5721/EuJRS20164940.

Appendix A – GIMBALS Project Folder Structure

The following screenshots show the files contained in the GIMBALS project folder structure.

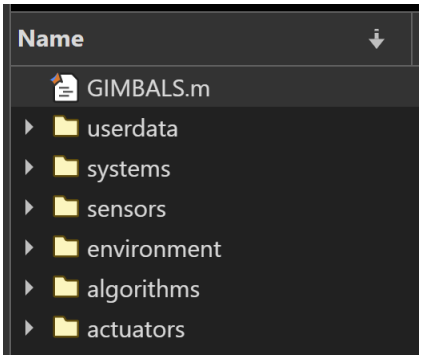


Figure A.1 – GIMBALS overall project folder structure.

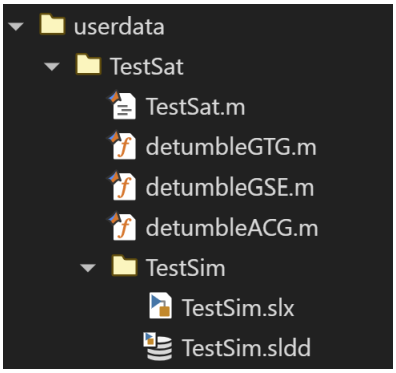


Figure A.2 – GIMBALS userdata folder structure.

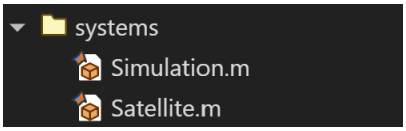


Figure A.3 – GIMBALS systems folder structure.

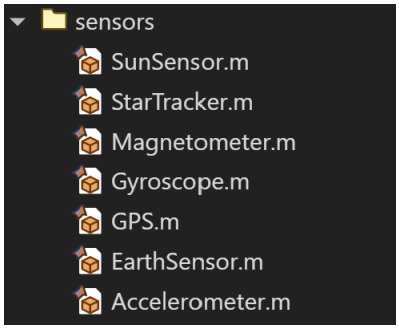


Figure A.4 – GIMBALS sensors folder structure.

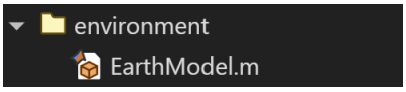


Figure A.5 – GIMBALS environment folder structure.

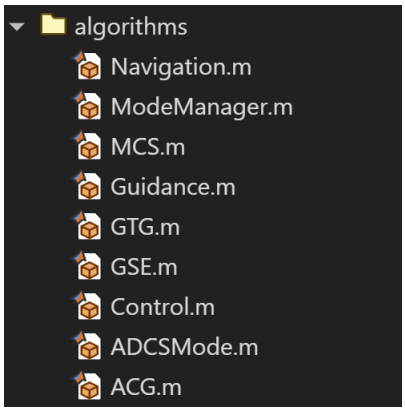


Figure A.6 – GIMBALS algorithms folder structure.

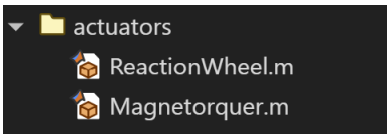


Figure A.7 – GIMBALS actuators folder structure.

Appendix B – GIMBALS Input File for TestSat Detumbling Simulation

```
%% TestSat.m
% GIMBALS input file for magnetic detumbling of a 2U CubeSat.

%% Define Simulation
Sim = Simulation(Name="TestSim");
Sim.StartTime = datetime("29-Nov-2024 00:00:00");
Sim.EndTime = datetime("29-Nov-2024 03:00:00");

%% Define Satellite
Sat1 = Satellite(Parent=Sim, Name="TestSat");
Sat1.InitialMass = 2.6;
Sat1.InitialInertiaTensor = [0.0125, 0, 0;    0, 0.0125, 0;    0, 0, 0.0045];
Sat1.InitialCOMLocation = [0; 0; 0];
Sat1.InitialSemimajorAxis = 6928140;
Sat1.InitialEccentricity = 0.0001;
Sat1.InitialInclination = 97.6;
Sat1.InitialRightAscension = 45;
Sat1.InitialArgumentOfPerigee = 0;
Sat1.InitialTrueAnomaly = 0;
Sat1.InitialAttitudeEuler = [10; 10; 10];
Sat1.InitialAngularVelocity = deg2rad([6; 8; 5]);

%% Define Sensors
MTM_Sat1 = Magnetometer(Parent=Sat1, Name="MTM");
MTM_Sat1.MountingDCM = [1.0, 0.0, 0.0;    0.0, 1.0, 0.0;    0.0, 0.0, 1.0];
MTM_Sat1.MisalignmentDCM = [0.999990, 0.004363, -0.000010;    -0.004363, 0.999990,
0.004363;    0.000010, -0.004363, 0.999990];
MTM_Sat1.NonOrthogonality = [0.0000, 0.0012, -0.0008;    0.0000, 0.0000, 0.0015;
0.0000, 0.0000, 0.0000];
MTM_Sat1.ScaleFactors = [1.0025; 0.9982; 1.0011];
MTM_Sat1.WhiteNoiseDensity = [1.2e-9; 1.2e-9; 1.2e-9];
MTM_Sat1.SamplingRate = 20;
MTM_Sat1.ConstantBias = [2.0e-7; -1.5e-7; 2.5e-7];
MTM_Sat1.Resolution = [1.83e-9; 1.83e-9; 1.83e-9];

GYRO_Sat1 = Gyroscope(Parent=Sat1, Name="GYRO");
GYRO_Sat1.MountingDCM = [1.0, 0.0, 0.0;    0.0, 1.0, 0.0;    0.0, 0.0, 1.0];
GYRO_Sat1.MisalignmentDCM = [0.999999, 0.000873, -0.000436;    -0.000873, 0.999999,
0.000873;    0.000436, -0.000873, 0.999999];
GYRO_Sat1.NonOrthogonality = [0.0000, 0.0004, -0.0002;    0.0000, 0.0000, 0.0003;
0.0000, 0.0000, 0.0000];
GYRO_Sat1.ScaleFactors = [1.0005; 0.9992; 1.0012];
GYRO_Sat1.WhiteNoiseDensity = [7.27e-5; 7.27e-5; 7.27e-5];
GYRO_Sat1.SamplingRate = 100;
GYRO_Sat1.ConstantBias = [4.8e-5; -3.5e-5; 5.1e-5];
GYRO_Sat1.Resolution = [8.3e-7; 8.3e-7; 8.3e-7];

ACC_Sat1 = Accelerometer(Parent=Sat1, Name="ACC");
ACC_Sat1.MountingDCM = [1.0, 0.0, 0.0;    0.0, 1.0, 0.0;    0.0, 0.0, 1.0];
ACC_Sat1.MountingCOMLocation = [0.015; -0.010; 0.020];
```

```
ACC_Sat1.MisalignmentDCM = [0.999999, -0.000523, 0.000712;    0.000523, 0.999999, -
0.000345;    -0.000712, 0.000345, 0.999999];
ACC_Sat1.NonOrthogonality = [0.0000, -0.0003, 0.0005;    0.0000, 0.0000, -0.0001;
0.0000, 0.0000, 0.0000];
ACC_Sat1.ScaleFactors = [1.0015; 0.9978; 1.0022];
ACC_Sat1.WhiteNoiseDensity = [4.9e-4; 4.9e-4; 4.9e-4];
ACC_Sat1.SamplingRate = 100;
ACC_Sat1.ConstantBias = [0.049; -0.038; 0.052];
ACC_Sat1.Resolution = [1.17e-5; 1.17e-5; 1.17e-5];

SSpX_Sat1 = SunSensor(Parent=Sat1, Name="SSpX");
SSpX_Sat1.MountingDCM = [0, 0, 1;    0, 1, 0;    -1, 0, 0];
SSpX_Sat1.FieldOfView = 120.0;
SSpX_Sat1.MisalignmentDCM = [0.999997, 0.002618, 0.000000;    -0.002618, 0.999997,
0.002618;    0.000000, -0.002618, 0.999997];
SSpX_Sat1.NonOrthogonality = [0.0000, 0.0005, 0.0000;    0.0000, 0.0000, 0.0003;
0.0000, 0.0000, 0.0000];
SSpX_Sat1.WhiteNoiseDensity = [8.7e-5; 8.7e-5; 8.7e-5];
SSpX_Sat1.SamplingRate = 5;
SSpX_Sat1.ConstantBias = [8.7e-4; -8.7e-4; 0.0];
SSpX_Sat1.Resolution = [1.7e-4; 1.7e-4; 1.7e-4];

SSnX_Sat1 = SunSensor(Parent=Sat1, Name="SSnX");
SSnX_Sat1.MountingDCM = [0, 0, -1;    0, 1, 0;    1, 0, 0];
SSnX_Sat1.FieldOfView = 120.0;
SSnX_Sat1.MisalignmentDCM = [0.999997, 0.002618, 0.000000;    -0.002618, 0.999997,
0.002618;    0.000000, -0.002618, 0.999997];
SSnX_Sat1.NonOrthogonality = [0.0000, 0.0005, 0.0000;    0.0000, 0.0000, 0.0003;
0.0000, 0.0000, 0.0000];
SSnX_Sat1.WhiteNoiseDensity = [8.7e-5; 8.7e-5; 8.7e-5];
SSnX_Sat1.SamplingRate = 5;
SSnX_Sat1.ConstantBias = [8.7e-4; -8.7e-4; 0.0];
SSnX_Sat1.Resolution = [1.7e-4; 1.7e-4; 1.7e-4];

SSpY_Sat1 = SunSensor(Parent=Sat1, Name="SSpY");
SSpY_Sat1.MountingDCM = [1, 0, 0;    0, 0, 1;    0, -1, 0];
SSpY_Sat1.FieldOfView = 120.0;
SSpY_Sat1.MisalignmentDCM = [0.999997, 0.002618, 0.000000;    -0.002618, 0.999997,
0.002618;    0.000000, -0.002618, 0.999997];
SSpY_Sat1.NonOrthogonality = [0.0000, 0.0005, 0.0000;    0.0000, 0.0000, 0.0003;
0.0000, 0.0000, 0.0000];
SSpY_Sat1.WhiteNoiseDensity = [8.7e-5; 8.7e-5; 8.7e-5];
SSpY_Sat1.SamplingRate = 5;
SSpY_Sat1.ConstantBias = [8.7e-4; -8.7e-4; 0.0];
SSpY_Sat1.Resolution = [1.7e-4; 1.7e-4; 1.7e-4];

SSnY_Sat1 = SunSensor(Parent=Sat1, Name="SSnY");
SSnY_Sat1.MountingDCM = [1, 0, 0;    0, 0, -1;    0, 1, 0];
SSnY_Sat1.FieldOfView = 120.0;
SSnY_Sat1.MisalignmentDCM = [0.999997, 0.002618, 0.000000;    -0.002618, 0.999997,
0.002618;    0.000000, -0.002618, 0.999997];
SSnY_Sat1.NonOrthogonality = [0.0000, 0.0005, 0.0000;    0.0000, 0.0000, 0.0003;
0.0000, 0.0000, 0.0000];
SSnY_Sat1.WhiteNoiseDensity = [8.7e-5; 8.7e-5; 8.7e-5];
SSnY_Sat1.SamplingRate = 5;
```

```
SSnY_Sat1.ConstantBias = [8.7e-4; -8.7e-4; 0.0];
SSnY_Sat1.Resolution = [1.7e-4; 1.7e-4; 1.7e-4];

SSpZ_Sat1 = SunSensor(Parent=Sat1, Name="SSpZ");
SSpZ_Sat1.MountingDCM = [1, 0, 0;    0, 1, 0;    0, 0, 1];
SSpZ_Sat1.FieldOfView = 120.0;
SSpZ_Sat1.MisalignmentDCM = [0.999997, 0.002618, 0.000000;    -0.002618, 0.999997,
0.002618;    0.000000, -0.002618, 0.999997];
SSpZ_Sat1.NonOrthogonality = [0.0000, 0.0005, 0.0000;    0.0000, 0.0000, 0.0003;
0.0000, 0.0000, 0.0000];
SSpZ_Sat1.WhiteNoiseDensity = [8.7e-5; 8.7e-5; 8.7e-5];
SSpZ_Sat1.SamplingRate = 5;
SSpZ_Sat1.ConstantBias = [8.7e-4; -8.7e-4; 0.0];
SSpZ_Sat1.Resolution = [1.7e-4; 1.7e-4; 1.7e-4];

SSnZ_Sat1 = SunSensor(Parent=Sat1, Name="SSnZ");
SSnZ_Sat1.MountingDCM = [-1, 0, 0;    0, 1, 0;    0, 0, -1];
SSnZ_Sat1.FieldOfView = 120.0;
SSnZ_Sat1.MisalignmentDCM = [0.999997, 0.002618, 0.000000;    -0.002618, 0.999997,
0.002618;    0.000000, -0.002618, 0.999997];
SSnZ_Sat1.NonOrthogonality = [0.0000, 0.0005, 0.0000;    0.0000, 0.0000, 0.0003;
0.0000, 0.0000, 0.0000];
SSnZ_Sat1.WhiteNoiseDensity = [8.7e-5; 8.7e-5; 8.7e-5];
SSnZ_Sat1.SamplingRate = 5;
SSnZ_Sat1.ConstantBias = [8.7e-4; -8.7e-4; 0.0];
SSnZ_Sat1.Resolution = [1.7e-4; 1.7e-4; 1.7e-4];

%% Define Actuators
MTQ1_Sat1 = Magnetorquer(Parent=Sat1, Name="MTQ1");
MTQ1_Sat1.MountingDCM = [0, 0, 1;    0, 1, 0;    -1, 0, 0];
MTQ1_Sat1.NumberOfCoils = 1200;
MTQ1_Sat1.CrossSectionalArea = 3.85e-5;
MTQ1_Sat1.CoreMagneticPermeability = 80000;
MTQ1_Sat1.MaximumDipoleStrength = 0.2;
MTQ1_Sat1.TelemetryRate = 10;

MTQ2_Sat1 = Magnetorquer(Parent=Sat1, Name="MTQ2");
MTQ2_Sat1.MountingDCM = [1, 0, 0;    0, 0, 1;    0, -1, 0];
MTQ2_Sat1.NumberOfCoils = 1200;
MTQ2_Sat1.CrossSectionalArea = 3.85e-5;
MTQ2_Sat1.CoreMagneticPermeability = 80000;
MTQ2_Sat1.MaximumDipoleStrength = 0.2;
MTQ2_Sat1.TelemetryRate = 10;

MTQ3_Sat1 = Magnetorquer(Parent=Sat1, Name="MTQ3");
MTQ3_Sat1.MountingDCM = [1, 0, 0;    0, 1, 0;    0, 0, 1];
MTQ3_Sat1.NumberOfCoils = 1200;
MTQ3_Sat1.CrossSectionalArea = 3.85e-5;
MTQ3_Sat1.CoreMagneticPermeability = 80000;
MTQ3_Sat1.MaximumDipoleStrength = 0.2;
MTQ3_Sat1.TelemetryRate = 10;

RW1_Sat1 = ReactionWheel(Parent=Sat1, Name="RW1");
RW1_Sat1.MountingDCM = [0.4472, 0.0000, 0.8944;    0.0000, 1.0000, 0.0000;    -
0.8944, 0.0000, 0.4472];
```

```
RW1_Sat1.WheelMass = 0.15;
RW1_Sat1.WheelInertiaTensor = [2.5e-5, 0, 0;    0, 2.5e-5, 0;    0, 0, 4.5e-5];
RW1_Sat1.WheelCOMLocation = [0.03; 0.00; -0.05];
RW1_Sat1.MotorTorqueConstant = 0.0055;
RW1_Sat1.MotorRLCircuitImpedance = 12.5;
RW1_Sat1.MotorRLCircuitInductance = 0.00018;
RW1_Sat1.StaticFriction = 8.0e-4;
RW1_Sat1.CoulombFriction = 3.5e-4;
RW1_Sat1.ViscousFriction = 1.2e-6;
RW1_Sat1.StribeckSpeed = 1.5708;
RW1_Sat1.SpeedUpperSaturationLimit = 628.3;
RW1_Sat1.SpeedLowerSaturationLimit = -628.3;
RW1_Sat1.InitialWheelSpeed = 0;
RW1_Sat1.TelemetryRate = 10;

RW2_Sat1 = ReactionWheel(Parent=Sat1, Name="RW2");
RW2_Sat1.MountingDCM = [1.0000, 0.0000, 0.0000;    0.0000, 0.4472, 0.8944;    0.0000,
-0.8944, 0.4472];
RW2_Sat1.WheelMass = 0.15;
RW2_Sat1.WheelInertiaTensor = [2.5e-5, 0, 0;    0, 2.5e-5, 0;    0, 0, 4.5e-5];
RW2_Sat1.WheelCOMLocation = [0.00; 0.03; -0.05];
RW2_Sat1.MotorTorqueConstant = 0.0055;
RW2_Sat1.MotorRLCircuitImpedance = 12.5;
RW2_Sat1.MotorRLCircuitInductance = 0.00018;
RW2_Sat1.StaticFriction = 8.0e-4;
RW2_Sat1.CoulombFriction = 3.5e-4;
RW2_Sat1.ViscousFriction = 1.2e-6;
RW2_Sat1.StribeckSpeed = 1.5708;
RW2_Sat1.SpeedUpperSaturationLimit = 628.3;
RW2_Sat1.SpeedLowerSaturationLimit = -628.3;
RW2_Sat1.InitialWheelSpeed = 0;
RW2_Sat1.TelemetryRate = 10;

RW3_Sat1 = ReactionWheel(Parent=Sat1, Name="RW3");
RW3_Sat1.MountingDCM = [-0.4472, 0.0000, -0.8944;    0.0000, 1.0000, 0.0000;
0.8944, 0.0000, 0.4472];
RW3_Sat1.WheelMass = 0.15;
RW3_Sat1.WheelInertiaTensor = [2.5e-5, 0, 0;    0, 2.5e-5, 0;    0, 0, 4.5e-5];
RW3_Sat1.WheelCOMLocation = [-0.03; 0.00; -0.05];
RW3_Sat1.MotorTorqueConstant = 0.0055;
RW3_Sat1.MotorRLCircuitImpedance = 12.5;
RW3_Sat1.MotorRLCircuitInductance = 0.00018;
RW3_Sat1.StaticFriction = 8.0e-4;
RW3_Sat1.CoulombFriction = 3.5e-4;
RW3_Sat1.ViscousFriction = 1.2e-6;
RW3_Sat1.StribeckSpeed = 1.5708;
RW3_Sat1.SpeedUpperSaturationLimit = 628.3;
RW3_Sat1.SpeedLowerSaturationLimit = -628.3;
RW3_Sat1.InitialWheelSpeed = 0;
RW3_Sat1.TelemetryRate = 10;

RW4_Sat1 = ReactionWheel(Parent=Sat1, Name="RW4");
RW4_Sat1.MountingDCM = [1.0000, 0.0000, 0.0000;    0.0000, -0.4472, -0.8944;
0.0000, 0.8944, 0.4472];
RW4_Sat1.WheelMass = 0.15;
```

```
RW4_Sat1.WheelInertiaTensor = [2.5e-5, 0, 0;    0, 2.5e-5, 0;    0, 0, 4.5e-5];
RW4_Sat1.WheelCOMLocation = [0.00; -0.03; -0.05];
RW4_Sat1.MotorTorqueConstant = 0.0055;
RW4_Sat1.MotorRLCircuitImpedance = 12.5;
RW4_Sat1.MotorRLCircuitInductance = 0.00018;
RW4_Sat1.StaticFriction = 8.0e-4;
RW4_Sat1.CoulombFriction = 3.5e-4;
RW4_Sat1.ViscousFriction = 1.2e-6;
RW4_Sat1.StribeckSpeed = 1.5708;
RW4_Sat1.SpeedUpperSaturationLimit = 628.3;
RW4_Sat1.SpeedLowerSaturationLimit = -628.3;
RW4_Sat1.InitialWheelSpeed = 0;
RW4_Sat1.TelemetryRate = 10;

%% Define Algorithms

% Mode Manager
Sat1.ModeManager.StateflowLoopRate = 5;
Sat1.ModeManager.StateflowVariables = {};
Sat1.ModeManager.StateflowDisplay = "Chart";

% Mode 1: Detumbling
% Global Target Generator (Guidance)
GTG1_Sat1 = GTG(Parent=Sat1.Guidance, Name="Detumbling", LoopRate=20,
Function={@detumbleGTG});
% Global State Estimator (Navigation)
GSE1_Sat1 = GSE(Parent=Sat1.Navigation, Name="Detumbling", LoopRate=20,
Function={@detumbleGSE, Sim, GYRO_Sat1, ACC_Sat1});
% Actuator Command Generator (Control)
ACG1_Sat1 = ACG(Parent=Sat1.Control, Name="Detumbling", LoopRate=20,
Function={@detumbleACG, Sat1.Guidance, Sat1.Navigation, MTM_Sat1, MTQ1_Sat1,
MTQ2_Sat1, MTQ3_Sat1});
Mode1_Sat1 = ADCSMode(Parent=Sat1.ModeManager, Name="Detumbling", GTG=GTG1_Sat1,
GSE=GSE1_Sat1, ACG=ACG1_Sat1);

% Mode Command Sequencer
Sat1.ModeManager.MCS.LoopRate = 20;

%% Configure Environment Models
Sim.EarthModel.UpdateRate = 0.2;

%% Build Simulation
Sim.Build();
```

Appendix C– GIMBALS Global Target Generator for Detumbling Test Case

```
function gtg = detumbleGTG()
% Generates a target satellite angular velocity of 0 rad/s along each axis
% to achieve detumbling.

    %% Set targets
    gtg.Quaternion = nan(4, 1);
    gtg.AngularVelocity = [0; 0; 0];
end
```

Appendix D – GIMBALS Global State Estimator for Detumbling Test Case

```
function gse = detumbleGSE(sim, gyro, acc)
% Uses an Extended Kalman Filter (EKF) to estimate the angular rate of the
% satellite using gyroscope and accelerometer measurements.

    %% 0. Inputs
    % Current simulation timestamp [s]
    t_sim = sim.ElapsedTime;
    % Gyroscope measurement vector in satellite body frame (3x1) [rad/s]
    gyro_meas = qrotate(qinv(gyro.MountingQuaternion), gyro.AngularVelocity);
    % Accelerometer measurement vector in satellite body frame (3x1) [m/s^2]
    accel_meas = qrotate(qinv(acc.MountingQuaternion), acc.Acceleration);

    %% 1. Declare Persistent Internal Filter Memory
    persistent x_state; % x_state(1:3) is the angular velocity estimate; x_state(4:6)
is the gyroscope drift bias estimate
    persistent P_cov;
    persistent t_last;
    persistent is_initialized;

    % Initialization phase on the very first operational step
    if (isempty(is_initialized))
        x_state = [gyro_meas; zeros(3, 1)];
        P_cov = blkdiag(eye(3) * 0.1, eye(3) * 1.0e-4);
        t_last = t_sim;
        is_initialized = true;
        gse.Quaternion = nan(4, 1);
        gse.AngularVelocity = x_state(1:3);
        gse.Position = nan(3, 1);
        gse.Velocity = nan(3, 1);
        return;
    end

    %% 2. Calculate Internal Time Increment (dt)
    dt = t_sim - t_last;
    if dt <= 0
        gse.Quaternion = nan(4, 1);
        gse.AngularVelocity = x_state(1:3);
        gse.Position = nan(3, 1);
        gse.Velocity = nan(3, 1);
        return;
    end

    %% 3. Unpack Parameters and State
    w = x_state(1:3);
    b = x_state(4:6);

    %% 4. Kinematic-Only Prediction Step
    % The unexpected acceleration (from torques) is handled by higher process noise
Q.
    w_pred = w;
```

```
    b_pred = b;
    x_pred = [w_pred; b_pred];

    % Kinematic Jacobian F_c = df/dx is completely zeroed out
    Phi = eye(6);

    % Process Noise Covariance Setup (Q)
    % Angular rate is assumed to be constant over dt (w_dot = 0).
    % Tuning note: The unexpected acceleration (from torques) will be
    % handled with higher process noise Q (Q_w is increased slightly here
    % to tell the EKF that the true angular rate can drift due to
    % unmodeled magnetorquer inputs.
    Q_w = eye(3) * (5.0e-4)^2;
    Q_b = eye(3) * (1.0e-7)^2;
    Q = [Q_w, zeros(3,3); zeros(3,3), Q_b] * dt;

    % Propagate Covariance
    P_pred = Phi * P_cov * Phi' + Q;

    %% 5. Measurement Update Step (Multi-Sensor Fusion)

    % --- SENSOR 1: Gyroscope Model ---
    h_gyro = w_pred + b_pred;
    H_gyro = [eye(3), eye(3)];
    R_gyro = diag(qrotate(qinv(gyro.MountingQuaternion), (gyro.WhiteNoiseDensity .*
sqrt(gyro.SamplingRate)).^2));

    % --- SENSOR 2: Accelerometer Model (Lever-Arm Centrifugal Dynamics) ---
    r = acc.MountingCOMLocation;
    w_cross_r = cross(w_pred, r);
    h_accel = cross(w_pred, w_cross_r);
    % w_dot is assumed to be zero to allow kinematics only estimation

    skew_w_cross_r = [0, -w_cross_r(3), w_cross_r(2);
                      w_cross_r(3), 0, -w_cross_r(1);
                     -w_cross_r(2), w_cross_r(1), 0];
    skew_r = [0, -r(3), r(2); r(3), 0, -r(1); -r(2), r(1), 0];
    skew_w = [0, -w(3), w(2); w(3), 0, -w(1); -w(2), w(1), 0];
    H_accel_w = -skew_w_cross_r - skew_w * skew_r;
    H_accel = [H_accel_w, zeros(3,3)];
    R_accel = diag(qrotate(qinv(acc.MountingQuaternion), (acc.WhiteNoiseDensity .*
sqrt(acc.SamplingRate).^2)));

    % --- Concatenate Matrices for Multi-Sensor Processing ---
    z = [gyro_meas; accel_meas];
    h = [h_gyro; h_accel];
    H = [H_gyro; H_accel];
    R = blkdiag(R_gyro, R_accel);

    %% 6. Compute Kalman Gain & Update Memory Fields
    Innovation = z - h;
    S_mat = H * P_pred * H' + R;
    K = P_pred * H' / S_mat;

    % Correct state and covariance estimates
```

```
    x_state = x_pred + K * Innovation;
    P_cov = (eye(6) - K * H) * P_pred;
    P_cov = (P_cov + P_cov') / 2;

    % Update persistent history vectors
    t_last = t_sim;

    %% 7. Outputs
    gse.Quaternion = nan(4, 1);
    gse.AngularVelocity = x_state(1:3);
    gse.Position = nan(3, 1);
    gse.Velocity = nan(3, 1);
end

function q_B2A = qinv(q_A2B)
    q_B2A = [q_A2B(1); -q_A2B(2); -q_A2B(3); -q_A2B(4)] ./ sqrt(sum(q_A2B.^2));
end

function v_B = qrotate(q_A2B, v_A)
    v_B = v_A + 2 * cross(q_A2B(2:4), cross(q_A2B(2:4), v_A) + q_A2B(1) .* v_A);
end
```

Appendix E – GIMBALS Actuator Command Generator for Detumbling Test Case

```
function acg = detumbleACG(guidance, navigation, mtm, mtq1, mtq2, mtq3)
% Generates 3-axis magnetic dipole commands for detumbling using the
% satellite angular velocity estimate and a low-pass filter on the raw
% magnetometer input.

    %% 0. Inputs
    % Satellite angular velocity target in the body frame (3x1) [rad/s]
    omega_target = guidance.AttitudeTarget.AngularVelocity;
    % Satellite angular velocity estimate in the body frame (3x1) [rad/s]
    omega_est = navigation.StateEstimate.AngularVelocity;
    % Magnetometer measurement vector in satellite body frame (3x1) [T]
    mag_meas = qrotate(qinv(mtm.MountingQuaternion), mtm.MagneticField);

    %% 1. Initialize Persistent Variables for Low-Pass Filter State
    persistent initialized;
    persistent mag_filt_prev;

    if isempty(initialized)
        mag_filt_prev = mag_meas; % Seed with first measurement to prevent turn-on
lag
        initialized = true;

        acg.MTQ1.ControlSetpoint = 0;
        acg.MTQ2.ControlSetpoint = 0;
        acg.MTQ3.ControlSetpoint = 0;

        acg.RW1.ControlMode = 0;
        acg.RW1.ControlSetpoint = 0;
        acg.RW2.ControlMode = 0;
        acg.RW2.ControlSetpoint = 0;
        acg.RW3.ControlMode = 0;
        acg.RW3.ControlSetpoint = 0;
        acg.RW4.ControlMode = 0;
        acg.RW4.ControlSetpoint = 0;
        return;
    end

    %% 2. Low Pass Filter Configuration & Execution (1 Hz Cutoff at 10 Hz Sample
Rate)
    alpha = 0.386;
    mag_filtered = alpha * mag_meas + (1 - alpha) * mag_filt_prev;
    mag_filt_prev = mag_filtered; % Store for the next iteration

    %% 3. Controller Tuning & Hardware Constraints
    K_gain = 40000.0; % Control scale factor

    %% 4. Compute Angular Velocity Error & Relative Field Derivative
    if norm(mag_filtered) < 1e-12
        acg.MTQ1.ControlSetpoint = 0;
        acg.MTQ2.ControlSetpoint = 0;
```

```
        acg.MTQ3.ControlSetpoint = 0;

        acg.RW1.ControlMode = 0;
        acg.RW1.ControlSetpoint = 0;
        acg.RW2.ControlMode = 0;
        acg.RW2.ControlSetpoint = 0;
        acg.RW3.ControlMode = 0;
        acg.RW3.ControlSetpoint = 0;
        acg.RW4.ControlMode = 0;
        acg.RW4.ControlSetpoint = 0;
        return;
    end

    % Compute the tracking velocity error vector
    omega_error = omega_est - omega_target;

    % Target B-Dot derivation mapping using the tracking error velocity
    b_dot_error = cross(mag_filtered, omega_error);

    % Ideal 3-axis commanded dipole vector in the Body Frame using proportional
tracking B-dot control law
    m_body_ideal = -K_gain * b_dot_error;

    %% 5. Allocate Body-Frame Dipole to Arbitrary Rod Axes
    % Extract the positive dipole axes (3rd columns)
    m_axis_X = mtq1.MountingDCM(:, 3); % X-torquer dipole vector in body frame
    m_axis_Y = mtq2.MountingDCM(:, 3); % Y-torquer dipole vector in body frame
    m_axis_Z = mtq3.MountingDCM(:, 3); % Z-torquer dipole vector in body frame

    % Construct the 3x3 Alignment Matrix
    % Column 1 = Rod 1, Column 2 = Rod 2, Column 3 = Rod 3
    A_mtq = [m_axis_X, m_axis_Y, m_axis_Z];

    % Project the ideal body frame vector onto the actual physical rods.
    m_rods_cmd = pinv(A_mtq) * m_body_ideal;

    % Apply command outputs
    acg.MTQ1.ControlSetpoint = m_rods_cmd(1);
    acg.MTQ2.ControlSetpoint = m_rods_cmd(2);
    acg.MTQ3.ControlSetpoint = m_rods_cmd(3);

    acg.RW1.ControlMode = 0;
    acg.RW1.ControlSetpoint = 0;
    acg.RW2.ControlMode = 0;
    acg.RW2.ControlSetpoint = 0;
    acg.RW3.ControlMode = 0;
    acg.RW3.ControlSetpoint = 0;
    acg.RW4.ControlMode = 0;
    acg.RW4.ControlSetpoint = 0;
end

function q_B2A = qinv(q_A2B)
    q_B2A = [q_A2B(1); -q_A2B(2); -q_A2B(3); -q_A2B(4)] ./ sqrt(sum(q_A2B.^2));
end
```

```
function v_B = qrotate(q_A2B, v_A)
    v_B = v_A + 2 * cross(q_A2B(2:4), cross(q_A2B(2:4), v_A) + q_A2B(1) .* v_A);
end
```

Appendix F – GIMBALS Satellite Dynamics Model Matlab Function

The satellite equations of translational and rotational motion are implemented programmatically by GIMBALS inside a Matlab Function Block. The exact appearance of this block varies based on the specifics of the user input file. The following is the generated function for the sample input defined in Appendix B:

```
function [satState, q_ECI2SC_dot, omega_dot_SC, r_dot_ECI, v_dot_ECI] =
TestSat_Update(satInternalStates, q_ECI2SC, omega_SC, r_ECI, v_ECI, earthModelRef,
TestSat_StructOut)
    %% Satellite Current State
    q_ECI2SC = q_ECI2SC / sqrt(sum(q_ECI2SC.^2));
    r_SC = qrotate(q_ECI2SC, r_ECI);
    r_SC_magnitude = sqrt(sum(r_SC.^2));
    m_SC = TestSat_StructOut.InitialMass;
    J_SC = TestSat_StructOut.InitialInertiaTensor;

    %% Environment References
    g_ECI = earthModelRef.Gravity;
    B_ECI = earthModelRef.MagneticField;
    mu_E = 398600435507000;

    %% Actuator Contributions
    J_COM = J_SC;
    J_DIS = zeros(3);
    J_ROT = zeros(3);
    H_int = [0; 0; 0];
    M_int = [0; 0; 0];
    M_ext = [0; 0; 0];
    F_ext_ECI = [0; 0; 0];
    q_SC2MTQ = satInternalStates.Magnetorquers.MTQ1.Quaternion;
    m_MTQ = satInternalStates.Magnetorquers.MTQ1.MagneticDipole;
    M_ext = M_ext + cross(qrotate(qinv(q_SC2MTQ), m_MTQ), qrotate(q_ECI2SC, B_ECI));
    q_SC2MTQ = satInternalStates.Magnetorquers.MTQ2.Quaternion;
    m_MTQ = satInternalStates.Magnetorquers.MTQ2.MagneticDipole;
    M_ext = M_ext + cross(qrotate(qinv(q_SC2MTQ), m_MTQ), qrotate(q_ECI2SC, B_ECI));
    q_SC2MTQ = satInternalStates.Magnetorquers.MTQ3.Quaternion;
    m_MTQ = satInternalStates.Magnetorquers.MTQ3.MagneticDipole;
    M_ext = M_ext + cross(qrotate(qinv(q_SC2MTQ), m_MTQ), qrotate(q_ECI2SC, B_ECI));
    q_SC2RW = satInternalStates.ReactionWheels.RW1.Quaternion;
    m_RW = satInternalStates.ReactionWheels.RW1.WheelMass;
    J_RW = satInternalStates.ReactionWheels.RW1.WheelInertiaTensor;
    r_RW_COM = satInternalStates.ReactionWheels.RW1.WheelCOMLocation;
    omega_RW = satInternalStates.ReactionWheels.RW1.WheelAngularVelocity;
    omega_dot_RW = satInternalStates.ReactionWheels.RW1.WheelAngularAcceleration;
    J_COM = J_COM + J_RW;
    J_DIS = J_DIS + m_RW * (r_RW_COM' * r_RW_COM) * eye(3);
    J_ROT = J_ROT - m_RW * (r_RW_COM * r_RW_COM');
    H_int = H_int + J_RW * qrotate(qinv(q_SC2RW), omega_RW);
    M_int = M_int - J_RW * qrotate(qinv(q_SC2RW), omega_dot_RW);
    q_SC2RW = satInternalStates.ReactionWheels.RW2.Quaternion;
    m_RW = satInternalStates.ReactionWheels.RW2.WheelMass;
```

```
    J_RW = satInternalStates.ReactionWheels.RW2.WheelInertiaTensor;
    r_RW_COM = satInternalStates.ReactionWheels.RW2.WheelCOMLocation;
    omega_RW = satInternalStates.ReactionWheels.RW2.WheelAngularVelocity;
    omega_dot_RW = satInternalStates.ReactionWheels.RW2.WheelAngularAcceleration;
    J_COM = J_COM + J_RW;
    J_DIS = J_DIS + m_RW * (r_RW_COM' * r_RW_COM) * eye(3);
    J_ROT = J_ROT - m_RW * (r_RW_COM * r_RW_COM');
    H_int = H_int + J_RW * qrotate(qinv(q_SC2RW), omega_RW);
    M_int = M_int - J_RW * qrotate(qinv(q_SC2RW), omega_dot_RW);
    q_SC2RW = satInternalStates.ReactionWheels.RW3.Quaternion;
    m_RW = satInternalStates.ReactionWheels.RW3.WheelMass;
    J_RW = satInternalStates.ReactionWheels.RW3.WheelInertiaTensor;
    r_RW_COM = satInternalStates.ReactionWheels.RW3.WheelCOMLocation;
    omega_RW = satInternalStates.ReactionWheels.RW3.WheelAngularVelocity;
    omega_dot_RW = satInternalStates.ReactionWheels.RW3.WheelAngularAcceleration;
    J_COM = J_COM + J_RW;
    J_DIS = J_DIS + m_RW * (r_RW_COM' * r_RW_COM) * eye(3);
    J_ROT = J_ROT - m_RW * (r_RW_COM * r_RW_COM');
    H_int = H_int + J_RW * qrotate(qinv(q_SC2RW), omega_RW);
    M_int = M_int - J_RW * qrotate(qinv(q_SC2RW), omega_dot_RW);
    q_SC2RW = satInternalStates.ReactionWheels.RW4.Quaternion;
    m_RW = satInternalStates.ReactionWheels.RW4.WheelMass;
    J_RW = satInternalStates.ReactionWheels.RW4.WheelInertiaTensor;
    r_RW_COM = satInternalStates.ReactionWheels.RW4.WheelCOMLocation;
    omega_RW = satInternalStates.ReactionWheels.RW4.WheelAngularVelocity;
    omega_dot_RW = satInternalStates.ReactionWheels.RW4.WheelAngularAcceleration;
    J_COM = J_COM + J_RW;
    J_DIS = J_DIS + m_RW * (r_RW_COM' * r_RW_COM) * eye(3);
    J_ROT = J_ROT - m_RW * (r_RW_COM * r_RW_COM');
    H_int = H_int + J_RW * qrotate(qinv(q_SC2RW), omega_RW);
    M_int = M_int - J_RW * qrotate(qinv(q_SC2RW), omega_dot_RW);

    %% External Disturbances
    M_gg = ((3 * mu_E) / r_SC_magnitude^3) * cross(r_SC / r_SC_magnitude, J_SC *
(r_SC / r_SC_magnitude));
    M_ext = M_ext + M_gg;

    %% Equations of Motion
    q_ECI2SC_dot = nan(4, 1);
    q_ECI2SC_dot(1) = -0.5 * omega_SC' * q_ECI2SC(2:4);
    q_ECI2SC_dot(2:4) = 0.5 * (q_ECI2SC(1) * omega_SC - cross(omega_SC,
q_ECI2SC(2:4)));
    omega_dot_SC = (J_COM + J_DIS + J_ROT)\(M_ext + M_int + cross(omega_SC, H_int +
(J_COM + J_DIS) * omega_SC));
    r_dot_ECI = v_ECI;
    v_dot_ECI = g_ECI + F_ext_ECI / m_SC;

    satState.Mass = m_SC;
    satState.InertiaTensor = J_SC;
    satState.Quaternion = q_ECI2SC;
    satState.QuaternionRate = q_ECI2SC_dot;
    satState.AngularVelocity = omega_SC;
    satState.AngularAcceleration = omega_dot_SC;
    satState.Position = qrotate(q_ECI2SC, r_ECI);
    satState.Velocity = qrotate(q_ECI2SC, r_dot_ECI);
```

```
    satState.Acceleration = qrotate(q_ECI2SC, v_dot_ECI);
end

function q_B2A = qinv(q_A2B)
    q_B2A = [q_A2B(1); -q_A2B(2); -q_A2B(3); -q_A2B(4)] ./ sqrt(sum(q_A2B.^2));
end

function v_B = qrotate(q_A2B, v_A)
    v_B = v_A + 2 * cross(q_A2B(2:4), cross(q_A2B(2:4), v_A) + q_A2B(1) .* v_A);
end
```