NEIGHBOR: A RELIABLE DE-ORBIT DEVICE FOR CUBESATS

BY

JUSTIN GREGORY LORENZ

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Aerospace Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Adviser:

Clinical Associate Professor Michael Lembeck

ABSTRACT

The Federal Communications Commission has recently adopted a Low Earth Orbit spacecraft disposal time requirement of five years following mission completion. CubeSats and nanosatellites allow increased access to space for smaller organizations and educational institutions, have lifetimes of up to 25 years in high orbits, but usually have no de-orbit capability. Preventative and removal technologies are essential for addressing this growing problem of inactive satellites remaining in orbit after their mission is completed. Preliminary design of the Next Era Individualized Geo-conscious High Body Orbit Remediator (NEIGHBOR) is assessed for its feasibility in curbing the risk that small satellites pose to the increasingly problematic space debris environment.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Michael Lembeck, for his insightful feedback and steadfast support over the years. His expertise in the field of Aerospace Engineering has guided me through my education in technical areas such as CubeSat development, Systems Engineering, human spaceflight, and beyond. He fosters a unique learning environment in the Laboratory for Advanced Space Systems at Illinois (LASSI), where students receive valuable training in the design, integration, and testing of CubeSats. His uncanny sense of humor and approach to managing satellite missions instills a sense of direction and camaraderie to the students he oversees.

My profound thanks also go to the students I worked alongside at LASSI, including but not limited to Eric Alpine, Michelle Zosky, Chris Young, Isabel Anderson, Courtney Trom, Hongrui Zhao, Qi Lim, Dave Gable, Michael Harrigan, and Rick Eason. Their technical and moral support throughout the process of writing this thesis and in the classes that we shared made an immeasurable difference to my growth as a person and completion of this work. Thanks go to Zachary Miller for his help with background information and leads on existing de-orbit devices, as well as Quentin Shaw for preliminary information regarding the discussion of NEIGHBOR's operational modes.

None of this would be possible without the strong support system I had outside of school. I would not be who I am today without my parents, William and Janet Lorenz, my brother, Jarod, and my girlfriend, Iulia. Your love and patience throughout this process has made all the difference in the world.

During the writing of this thesis, I have had the privilege of coming to know many bright and compassionate people that I am proud to call my friends. The memories we have shared will stay with me forever.

This work is dedicated to each and every one of my friends, family, instructors, and fellow students. I could not explore the stars without your support.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION 1

1.1 Defining and Addressing Space Debris 1

1.2 Space Debris History 3

1.3 CubeSats and Their Potential to Become Space Debris 8

1.4 The Costs of Space Debris 12

1.5 Introducing NEIGHBOR 17

CHAPTER 2: MISSION CONCEPTUALIZATION 19

2.1 Existing Debris Removal Technologies 19

2.2 Relevant Guidelines 20

2.3 Concept of Operations 21

2.4 Capabilities and Requirements 22

2.5 Orbit Dynamics 23

CHAPTER 3: NEIGHBOR DESIGN 26

3.1 Reference Design Mission 26

3.2 Applicable Orbits 28

3.3 Attitude Determination and Control 33

3.4 Command and Data Handling 79

3.5 Power Supply 81

3.6 Data Collection and Downlink 84

3.7 Interfaces 87

3.8 Mass and Volume 90

3.9 Risk Analysis and Mitigation 91

CHAPTER 4: CONCLUSION AND FUTURE WORK 94

REFERENCES 96
APPENDIX A: ATTITUDE CONTROL AND DE-ORBIT OBJECTS CODE 99
APPENDIX B: ATTITUDE CONTROL AND DE-ORBIT PLOT SETTINGS CODE 102
APPENDIX C: ATTITUDE CONTROL AND DE-ORBIT PROCEDURES CODE 122
APPENDIX D: DETUMBLE MANEUVER CODE 130
APPENDIX E: SLEW MANEUVER CODE 132
APPENDIX F: DE-ORBIT MANEUVER CODE 134
APPENDIX G: SLEW MANEUVER VALIDATION CASE THREE CODE 137
APPENDIX H: TANK SIZING CODE 141
APPENDIX I: POWER AND DATA ANALYSIS CODE 143

CHAPTER 1: INTRODUCTION

1.1 Defining and Addressing Space Debris

According to the Inter-Agency Space Debris Coordination Committee, “space debris is defined as all artificial objects including fragments and elements thereof, in Earth orbit or re-entering the atmosphere, that are non-functional [1].” Space debris comes in all shapes and sizes: NASA’s Orbital Debris Program Office estimates that there are tens of thousands of pieces of debris 10 cm in size and larger, hundreds of thousands of pieces of debris 1 cm in size and larger, and over a hundred million pieces of debris 1 mm in size and larger [2] . For size comparison of these objects, refer to Table 1 [2]. Space debris can also be as large as defunct satellites and rocket stages. It also includes fragments of hardware left behind from spacecraft collisions, breakups, or failures—intentional or unintentional.

Table 1. Sizes of debris are paired with representative and familiar household objects for visual understanding.

Representative Object	Debris Size
	10 cm and larger
	1 cm and larger
	1 mm and larger

Eventually, objects in orbit around Earth experience a gradual decrease in altitude. Orbit decay is caused principally by atmospheric drag and other perturbing forces. With respect to the space debris environment around Earth, orbit decay naturally removes some of the objects from the total space debris population [3]. Space debris re-entry times can vary due to a number of factors like mass, surface area, starting altitude, and seasonal variations in perturbing forces. Objects at higher altitudes in Low Earth Orbit (LEO) can take decades or even centuries to re-enter Earth's atmosphere.

Studies have shown that a compliance rate of 90% for all spacecraft de-orbiting within twenty-five years of end of mission and active removal of five space debris objects per year are required to stabilize the growing space debris environment [4]. However, responsibility, accountability, and international consensus are hard to assign to the growing problem of space debris. NASA's compliance rate is around 96% as of January 17, 2021, while the global compliance rate remains somewhere between 20% and 30% on average [2]. NASA maintains that prevention of future space debris outweighs the pursuit and usage of remediation technology that addresses existing space debris [2]. In the meantime, the Federal Communications Commission has shifted its orbital lifetime requirement following mission completion from the longstanding time of twenty-five years to five years as of September 29, 2022 [5]. Both space debris prevention and removal technologies are needed to confront the issue and provide pathways for reducing it.

The proliferation of a small class of spacecraft under 25 kg known as CubeSats is contributing to the space debris environment in LEO. The number of CubeSat missions is expected to increase significantly in the future. The increase in number of objects in LEO may force satellite providers to place more satellites in higher orbits. While CubeSat missions at lower altitudes can be naturally compliant with debris mitigation guidelines, active de-orbit devices are necessary for CubeSat missions that require higher operational altitudes and will greatly help in mitigating them as a source of space debris as their popularity increases. In this thesis, a potential technical solution is introduced for mitigating the potential for CubeSats to contribute to the space debris problem.

1.2 Space Debris History

The accumulation of space debris in the near-Earth space environment began with the earliest space operations conducted by the United States and the Soviet Union during the Space Race of the Cold War. Since the Soviet Union launched Sputnik-1, the first artificial satellite to orbit the Earth, on October 4, 1957, thousands of payloads—space objects designed to perform a specific function in space, according to the European Space Agency (ESA)—have been launched into LEO [6]. Figure 1 shows the evolution of the number of payloads launched into LEO between perigee altitudes of 200 and 1750 km since the start of the Space Race [6]. As of May 3, 2022, NASA reports over 10000 metric tons of objects in Earth orbit catalogued by the United States Space Surveillance Network [7].

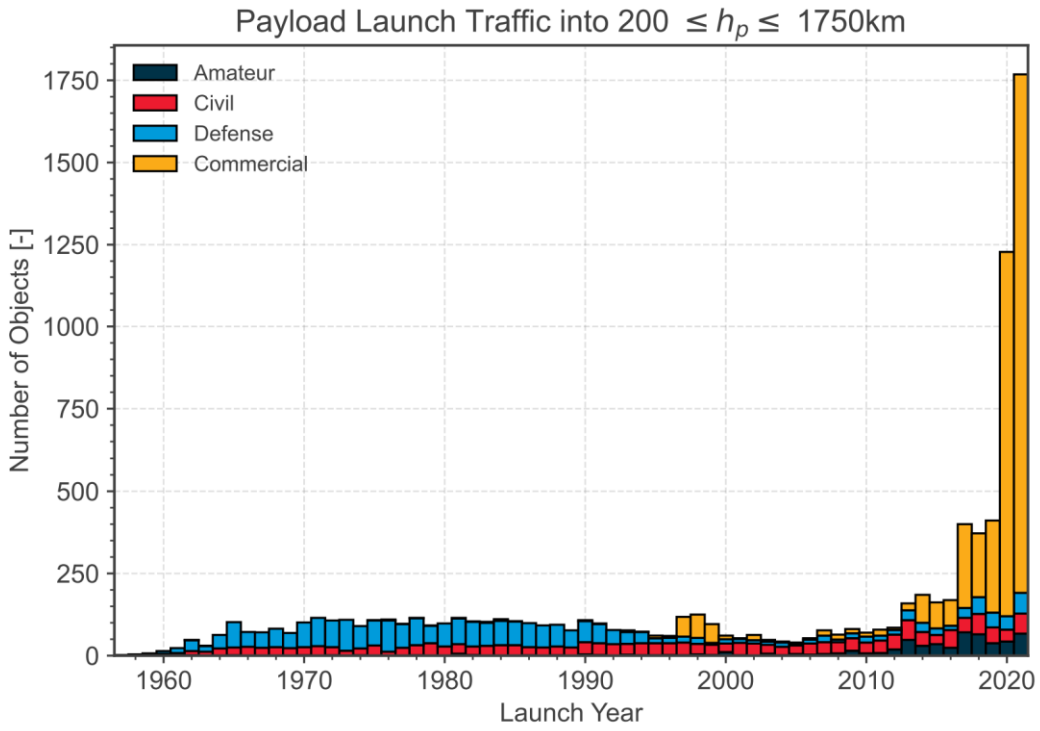


Figure 1. The evolution of the space environment of LEO between perigee altitudes of 200 and 1750 km is shown.

Some types and densities of space debris are manageable. The greatest problems, however, arise with collisions and fragmentation of objects in space, which have the potential to scatter many more independent objects on uncontrolled trajectories. It can be challenging to respond and account for these sudden events. As the number of independent objects in space increases, so too does the likelihood for collisions of objects and fragmentation events. In the history of human activities in space, 566 collision events have occurred. As of January 27, 2021, 260 of these collision events had occurred in the previous 20 years [2]. Considering a timeframe of several decades of human operations in space for this data point, this shows a dramatic uptick in the number of collision events in the past few decades.

The occurrence of satellite breakups increased in frequency through the 1970s, and an extrapolation of this trend into the new century results in an average rate of roughly four fragmentation events per year as of July 4, 2018 [8]. In the 1980s, Delta rocket second stages, weapons tests, and satellite self-destructs made up over 25% of satellite debris [8]. Mitigation efforts and awareness of space debris played a part in the 2010s where lower levels of fragmentation events were observed in 2013 and 2017 [8]. However, energetic devices left over from space operations lie in wait and are a risk to current space operations. A Russian Sistema Obespecheniya Zapuska (SOZ) ullage motor, an auxiliary motor for Russian SL-12 launch vehicles, fragmented on April 15, 2022, raising further concerns about other intact and fragmented SOZ motors in space [9]. As of June 2022, NASA estimates 27 intact and 35 fragmented SOZ motors on orbit. Figure 2 depicts the ullage motor attached to an SOZ unit [10]. The units have overall dimensions of 60 cm x 60 cm x 100 cm and a dry mass of 56 kg. These motors, and other energetic devices, remain in orbit as ticking time bombs.

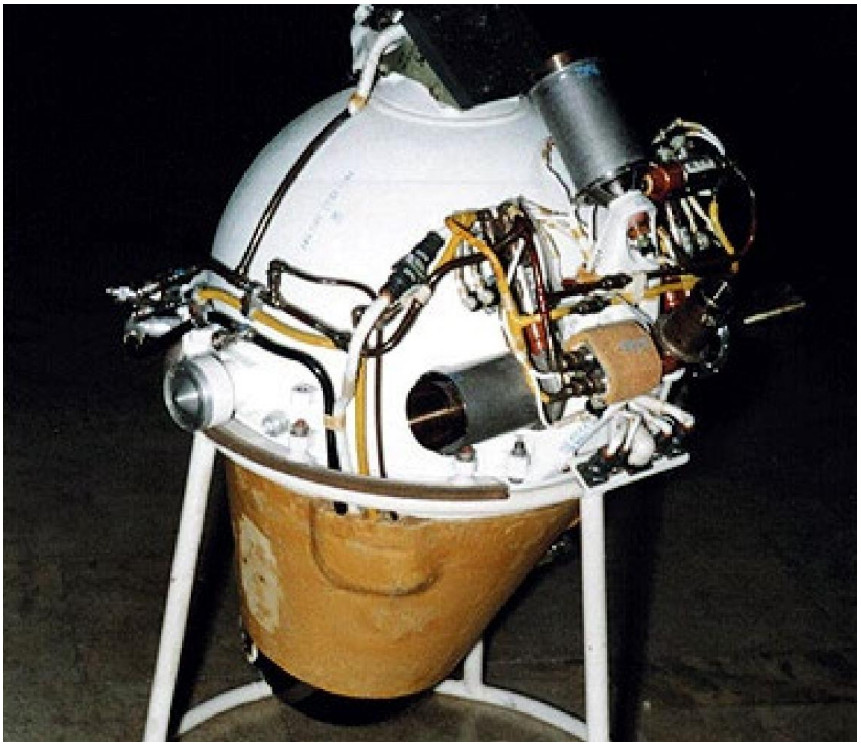


Figure 2. A Russian SOZ unit with a 11D79 ullage motor is depicted.

Some countries have conducted deliberate anti-satellite weapons tests that generated large debris clouds. On January 11, 2007, a Chinse anti-satellite missile was used against the communications satellite known as Fengyun-1C, generating over 3000 pieces of debris [11]. At the time, the collision doubled the amount of debris at an altitude of 800 km [12]. On November 15, 2021, a Russian anti-satellite missile test on the defunct signals intelligence satellite Kosmos-1408 generated over 1500 pieces of trackable debris as of March 1, 2022 [13]. The situation is evolving as more objects are identified and tracked.

Collisions can also occur by accident. In 2009, the collision of Iridium-33 and Kosmos-2251 generated over 1800 trackable pieces of debris according to the United States Space Surveillance Network [14]. Iridium-33 was a functional United States communications satellite and Kosmos-2251 was a non-functional Russian communications satellite. Iridium-33 ceased functioning after the collision. It is noteworthy that fragmentations of Iridium-33, Kosmos-2251, and Fengyun-1C generated over 30% of all catalogued space debris as of and over 13% of all catalogued objects after the launch of Sputnik-1 in October of 1957, as of July 4, 2018 [8]. The

risk for an increase in the levels of space debris surrounding Earth due to weapons tests and catastrophic satellite collisions is ever-present.

Figure 3 illustrates the historical rise of tracked space objects—including satellites—in LEO from the 1960s onwards [7]. In the figure, there are three numbered vertical jumps in the amount of catalogued fragmentation debris objects [13]. The first increase comes from the intentional destruction of Fengyun-1C by China. The second comes from the accidental collision of Iridium-33 and Kosmos-2251. The third comes from the more recent intentional destruction of Kosmos-1408 by the Russian Federation on November 15, 2021. Over 26000 orbital debris objects are catalogued by the United States Space Surveillance Network. The total number of catalogued objects has more than doubled in just the last two decades with respect to the entire history of human spaceflight. Fragmentation debris makes up the lion's share of the total tracked debris population.

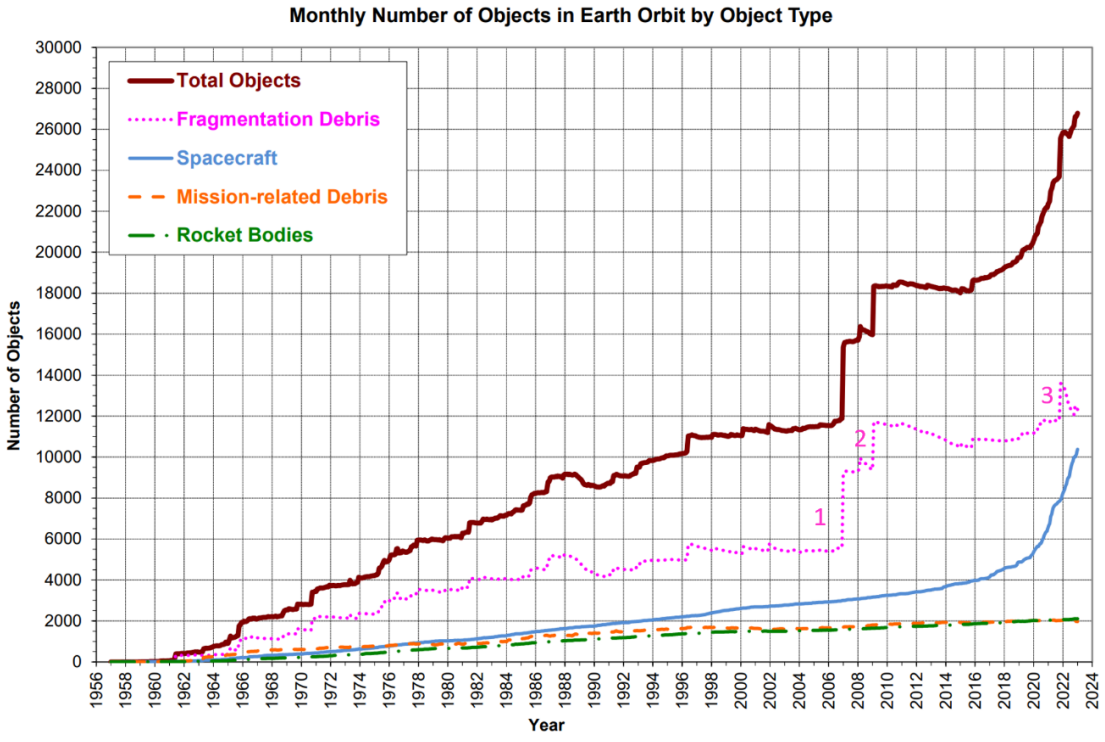


Figure 3. An evolution of tracked space debris objects as of February 3, 2023. Three vertical jumps in the catalogued fragmentation debris correspond to (1) the intentional destruction of Fengyun-1C by China, (2) the accidental collision of Iridium-33 and Kosmos-2251, (3) and the intentional destruction of Kosmos-1408 by the Russian Federation.

A few influential trailblazers made early contributions in response to the growing threat of space debris to human space operations. John Gabbard of the North American Aerospace Defense Command created and updated his own database in the 1960s and 1970s to improve the tracking of space objects and developed a technique to predict the orbital paths of post-explosion or collision debris, known as Gabbard Diagrams or Gabbard Plots.

In 1978, Donald J. Kessler and Burton G. Cour-Palais co-authored a piece arguing that space debris would be a higher risk to spacecraft than micrometeoroids by the year 2000 [15]. They also proposed the concept of cascading of space debris collisions. This is the theoretical scenario where LEO object density becomes so high that any collisions of objects would cause a chain reaction of subsequent collisions that could make space significantly more difficult to access or even inaccessible for decades or centuries. The term "Kessler Syndrome" refers to this theory of collisional cascading. At the time, the main belief was that drag was pulling debris down faster than it was being created, but Gabbard knew the database vastly underestimated the real amount of debris. Kessler received funding in 1979 from NASA to study space debris further.

When or if such a collisional cascading event could occur is subject to great uncertainty with current modelling methods. In 2011, the United States National Research Council published a report that predicted a collisional cascading event could occur within two decades [16]. Kessler suggested such an event could take place by the year 2000 [15]. Since then, he has revised his prediction to somewhere in the 21st Century [17]. Some claim that economic infeasibility of space operations would occur before a collisional cascading event [18]. Going forward, human spaceflight missions may be deemed too risky in the lower regions of LEO due in part to increasing density of CubeSats and internet broadband satellites. Despite these uncertainties, one thing remains clear: a continued net increase in the space debris population over time increases the likelihood of a collisional cascading event.

Many investigations into the nature of the space debris environment have shown that collision fragments of the larger variety cannot be removed by atmospheric drag alone more quickly than they can be generated by the current number of intact objects in space [4]. Kessler et al. concluded that 90% compliance of post mission disposal and active debris removal of five objects per year would help stabilize the current amount of catalogued space debris [4]. Both

preventative and active debris removal technologies are required to stabilize and decrease the amount of space debris in orbit around Earth.

A dramatic shift in space operations is already here. In recent years, commercial satellite operators—most notably SpaceX with its Starlink Project—have competed in a veritable new space race to seize the opportunity to provide internet broadband services to tens of millions of people in the United States and across the globe [19]. Their plans involve large satellite constellations to provide these services. In 2018, the United States Federal Communications Commission, which has a role in limiting the accumulation of orbital debris via its licensing process for commercial satellite operations, approved plans by SpaceX to increase the number of satellites in LEO by nearly 12000. In 2019, SpaceX made submissions to the International Telecommunications Union for 30000 additional satellites [12].

There are still more satellites to come after these massive, planned additions. Amazon filed with the Federal Communications Commission for over 3000 satellites as part of its Kuiper System in 2019. OneWeb plans to have over 650 satellites in their own constellation [12]. China Aerospace Science and Industry Corporation's Hongyun Project also plans to be a large constellation [19]. The beginnings of this unprecedented change in commercial satellite operation are already being felt. Some of the first satellites to make up OneWeb's constellation made their journey to space in 2019 and over 200 Starlink satellites were already in orbit at the start of 2020 [12]. These constellations will create an unprecedented increase in the total number of satellites orbiting Earth and provide a much higher risk for collision events and the production of space debris in the coming decades.

1.3 CubeSats and Their Potential to Become Space Debris

The CubeSat concept was first developed in 1999 by Robert Twiggs of Stanford University and Jordi Puig-Suari of California Polytechnic University at San Luis Obispo [20]. It makes use of a standard form factor called the "U" or "unit." The "U" has dimensions of 10 cm x 10 cm x 10 cm and can be joined to other single U segments to create various styles of CubeSat configuration. A CubeSat that occupies a volume of one unit is called a "1U," one that is six

units in volume is called a “6U,” and so on. Figure 4 illustrates the “U” and the scalable nature of the CubeSat [21].

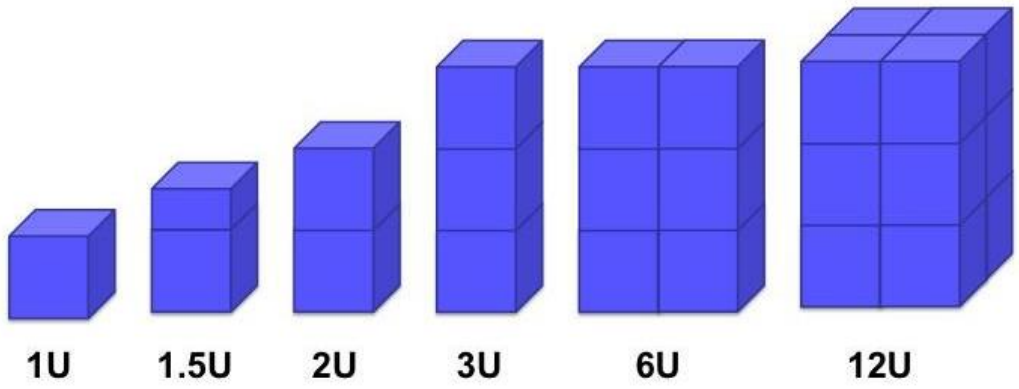


Figure 4. Different styles of CubeSats are depicted based on the corresponding multiple of volume units.

Educational institutions and smaller commercial entities are drawn to CubeSat missions due to the smaller scale, lower complexity, and decreased costs associated with these classes of missions. Prior to the introduction of CubeSats, spacecraft missions were much larger in terms of mass and volume. They also required significant budgets to fully develop and launch. CubeSats, on the other hand, can typically be developed for less than USD 1 million [22]. CubeSat missions in educational and research institutions may further benefit from the low-cost or volunteer labor of students and research assistants [22]. Additionally, these missions benefit from the cost reductions associated with the procurement of commercial-off-the-shelf parts and usage of open-source software [22]. Rideshare opportunities on launch vehicles also benefit CubeSat missions, where satellite operators can share the cost of launch as secondary payloads of a launch vehicle. CubeSats are used to perform a variety of functions like internet backhaul, remote sensing, space geology, and communications relay as part of a multi-satellite constellation [3] [22].

As of January 1, 2023, over 1800 CubeSats have successfully deployed in orbit as shown in Figure 5 [23]. Based on the data, most CubeSats do not have propulsion modules, cutting down on mass or complying with launch provider requirements for safety purposes [3]. When it comes to space debris mitigation, CubeSats without propulsion units or some other means of

motion control cannot perform collision avoidance maneuvers, leaving them at risk, albeit relatively low, of adding to the space debris population after a collisional breakup. Additionally, CubeSats with higher operational altitudes in LEO may take longer than five years to naturally decay and re-enter in Earth's atmosphere, limiting the scope of orbital regimes they can occupy while mitigating the proliferation of space debris.

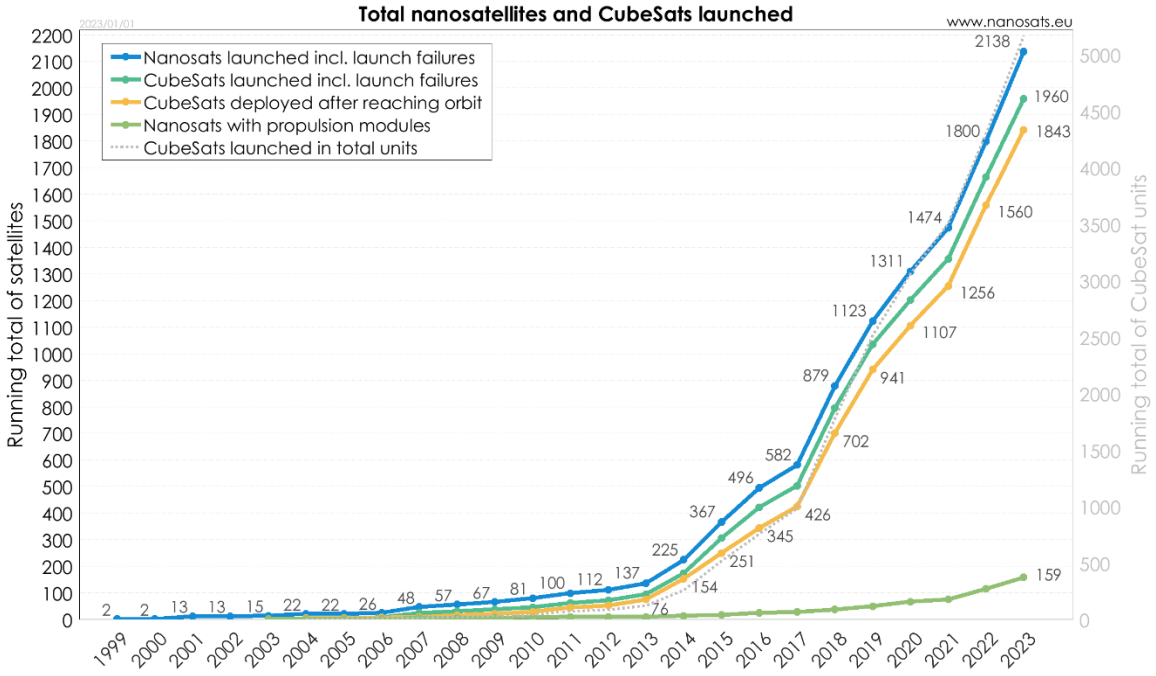


Figure 5. A summary of nanosatellites and CubeSats launched into space.

CubeSat launches can also be characterized by the types of organizations developing them. The historical trends of CubeSat launch by organization are visualized in Figure 6 [23]. A 2018 analysis of university-class CubeSat missions by Michael Swartwout found that approximately 40% fail to complete any of their main mission objectives [24]. Another study by Swartwout in 2017 found that a quarter of university-class CubeSats were dead-on-arrival, meaning they fail to begin mission operations once deployed in orbit [25]. Mission failures may stem from a variety of sources such as power system failures, mechanical failures, communications failures, system design issues, and lack of attention to requirements and the

systems engineering process [22]. Typically, students work on these missions to get hands-on exposure in a classroom or research setting. Many are getting their first exposure and may make mistakes in the design, analysis, integration, and testing of CubeSats. One study found that proper testing shows a greater enhancement of CubeSat reliability in comparison to redundancy [26]. Funding may not always be available and proper facilities may be unavailable or insufficient for adequate testing.

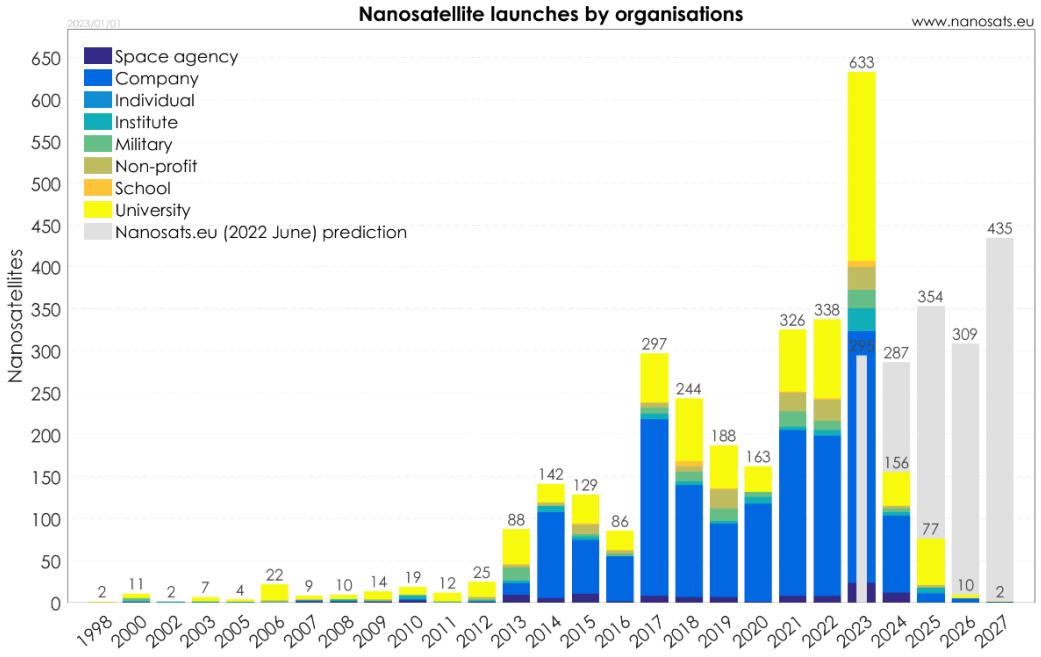


Figure 6. CubeSat launches are categorized by organization type as of January 1, 2023.

Another statistical analysis of small satellite reliability performed with respect to failure data of 222 satellites characterized by a launch mass of under 500 kg found that the smallest of satellites experienced higher rates of "infant mortality" and shorter mission lifetimes due to the failures [27]. Infant mortality of CubeSats is defined as failure in the early stages of a mission or during commissioning, the satellite mission phase that typically includes initial contact with the ground station, powering of devices, calibration of components, and initial orbit determination [28]. The subsystems that contributed the most to infant mortality of small satellites were

Telemetry, Tracking and Command, Thermal Control System, and Structures and Mechanisms [27]. In a 2013 statistical analysis of the first one hundred CubeSats, Swartwout found that a third of the failed missions never sent radio signals to ground [29]. The risk is present for CubeSats to become space debris as more and more universities enter a fledgling state of CubeSat development. Naturally, commercial entities with greater pools of resources and experience see lower levels of failure in their own CubeSat missions.

1.4 The Costs of Space Debris

Tracking and modeling space debris is a challenging and resource-intensive task. Information on the total current nature of the space debris environment is incomplete and difficult to ascertain, as certain existing orbital debris objects are untraceable by current methods and must be estimated [12]. Tracking space debris involves monitoring massive expanses of space surrounding Earth with objects that can be as tiny as grains of sand. Collaboration is essential to maximizing knowledge regarding the space debris environment so that the damage inflicted on space assets is minimized. This involves partnerships of not only military and government satellite operators and space object trackers, but also commercial and civilian entities.

The United States has agreements with numerous countries and organizations to share observation and tracking data of space debris objects [12]. In Europe, the European Union is providing funding to a Consortium of Member States including Germany, France, the United Kingdom, Italy, Spain and more recently Poland, Portugal, and Romania, to join their efforts in observational and data processing capabilities. The United States Department of Defense's Space Surveillance Network consists of terrestrial and space-based radars, lasers, and telescopes responsible for tracking and cataloguing space objects that are 10 cm and larger in LEO and 1 m and larger in Geo-synchronous Earth Orbit [12]. To fill in some gaps in knowledge, NASA radars, telescopes and in-situ measurements identify space objects that the Space Surveillance Network cannot that are large enough to pose risk to ongoing space missions [12].

However, certain existing assets for space surveillance are found to be lacking in their capabilities. In its January 2021 report, NASA's Office of Inspector General found the agency's data on the orbital debris environment to be lacking [2]. The Orbital Debris Program Office of NASA uses data from three radar systems to locate and estimate debris larger than 3 mm. Two of these are HUSIR and HAX, managed by the Massachusetts Institute of Technology. The third is Goldstone, managed by NASA Jet Propulsion Laboratory. Access to these radar systems is negatively affected by lack of funding, equipment maintenance issues, and an overlap of use for different missions. The Orbital Debris Program Office has also had issues with sourcing a means of identifying debris 3 mm and smaller in the 400 to 1000 km range of LEO. This is a serious problem since millimeter-sized orbital debris has the greatest potential to penetrate and disrupt most space operations in LEO. Furthermore, NASA lacks a means to track debris smaller than 10 cm in LEO around the International Space Station (ISS). It plans to use the Department of Defense's Space Fence, another radar system, for this purpose. As long as Space Fence is not fully operational, the ISS is subject to the risks posed by lack of knowledge of these small but damaging pieces of orbital debris.

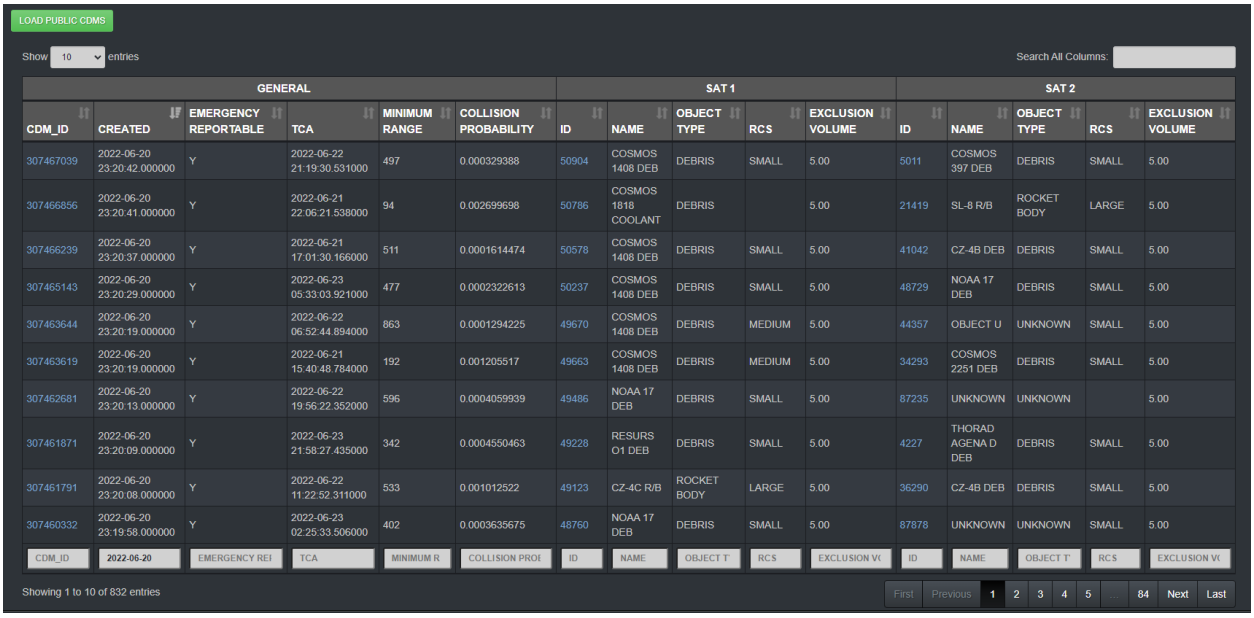
The gaps in knowledge of space objects that cannot be accounted for by direct observation are filled in by statistical models, but the models themselves are imperfect. Around 20,000 space debris objects are now tracked and catalogued by the United States Air Force. This is estimated to be fewer than 0.02% of the total theorized space debris object population [12]. Space Fence will help the situation in the sense of total tracked and catalogued objects but not in the sense of identification and tracking accuracy [12]. Despite this, collisions between spacecraft and debris smaller than 10 cm in size are often difficult to analyze. These kinds of collisions can leave satellite operators guessing whether a collision took place, whether the spacecraft has malfunctioned and is making false reports, or whether the spacecraft has sustained a structural failure of its own volition. Additionally, collisions of this scale can result in slight perturbations in the orbits of the spacecraft and even loss of partial function. The latter makes it less likely that a collision is reported. Preventative and active debris removal technologies would decrease the costs associated with the time and resources necessary to track and estimate space debris by eliminating objects in space from the total set to be tracked [2].

Many organizations and communities already rely on spacecraft in various Earth orbits for telecommunications, weather forecasting, global positioning, climate research, remote sensing, internet provision, and other critical services that create socioeconomic value. Commercial, civilian, and government entities have high stakes in the continued operations of the spacecraft that provide these services and the investment and development of new space-based services [2]. Notably, human spaceflight operations hosted at the ISS take place at 400 km altitude. Additionally, Earth observation tends to take place all throughout the altitude range of 600-700 km and some at 780-900 km. Examples include ESA's Sentinel missions that support the Copernicus Program in monitoring climate change. Furthermore, weather forecasting tends to take place at 800-830 km; specifically, the satellites of the World Meteorological Organization's Global Observation System occupy this orbital altitude. On top of this, some communications satellites such as those of Iridium's constellation inhabit the 700-800 km altitude regime [12]. The orbit regimes most susceptible to a collisional cascading event lie between 650 and 1000 km and close to 1400 km altitude in LEO [12]. This is where the highest density bands of space debris are located. The collision involving Iridium-33 and Kosmos-2251 occurred at 776 km altitude and contributed to the vulnerability of these operational altitudes [12]. Therefore, numerous valuable services are already at risk with the current state of the space debris environment.

While costs are incurred in the pursuit of technologies promoting spacecraft protection, collision avoidance, space surveillance, and debris mitigation, massive losses would be realized should the space debris environment become unsustainable via a collisional cascading event. Hosts of existing space assets could be damaged or destroyed in the fallout and certain orbits could be rendered unusable for the foreseeable future, confronting satellite operators and government agencies with immense costs of lost opportunity. Such a situation might also lead to heightened geopolitical tensions as nations negotiate and compete over the most desirable remaining orbits.

Conjunction events can be described as a "geometric close approach between two objects" according to the ESA [6]. Public information available on space-track.org, a tool for information on objects in space, shows over 800 conjunction events for June 20, 2022 as an

example [30]. This can be seen in Figure 7. Objects in space regularly come into close contact in a manner that does not quite merit evasive maneuvers.



LOAD PUBLIC CDMS

Show 10 entries Search All Columns:

GENERAL						SAT 1					SAT 2				
CDM_ID	CREATED	EMERGENCY REPORTABLE	TCA	MINIMUM RANGE	COLLISION PROBABILITY	ID	NAME	OBJECT TYPE	RCS	EXCLUSION VOLUME	ID	NAME	OBJECT TYPE	RCS	EXCLUSION VOLUME
307467039	2022-06-20 23:20:42.000000	Y	2022-06-22 21:19:30.531000	497	0.000329388	50904	COSMOS 1408 DEB	DEBRIS	SMALL	5.00	5011	COSMOS 397 DEB	DEBRIS	SMALL	5.00
307466856	2022-06-20 23:20:41.000000	Y	2022-06-21 22:06:21.538000	94	0.002699698	50786	COSMOS 1818 COOLANT	DEBRIS		5.00	21419	SL-8 R/B	ROCKET BODY	LARGE	5.00
307466239	2022-06-20 23:20:37.000000	Y	2022-06-21 17:01:30.166000	511	0.0001614474	50578	COSMOS 1408 DEB	DEBRIS	SMALL	5.00	41042	CZ-4B DEB	DEBRIS	SMALL	5.00
307465143	2022-06-20 23:20:29.000000	Y	2022-06-23 05:33:03.921000	477	0.0002322613	50237	COSMOS 1408 DEB	DEBRIS	SMALL	5.00	48729	NOAA 17 DEB	DEBRIS	SMALL	5.00
307463644	2022-06-20 23:20:19.000000	Y	2022-06-22 06:52:44.894000	863	0.0001294225	49670	COSMOS 1408 DEB	DEBRIS	MEDIUM	5.00	44357	OBJECT U	UNKNOWN	SMALL	5.00
307463619	2022-06-20 23:20:19.000000	Y	2022-06-21 15:40:48.784000	192	0.001205517	49663	COSMOS 1408 DEB	DEBRIS	MEDIUM	5.00	34293	COSMOS 2251 DEB	DEBRIS	SMALL	5.00
307462681	2022-06-20 23:20:13.000000	Y	2022-06-22 19:56:22.352000	596	0.0004059939	49486	NOAA 17 DEB	DEBRIS	SMALL	5.00	87235	UNKNOWN	UNKNOWN		5.00
307461871	2022-06-20 23:20:09.000000	Y	2022-06-23 21:58:27.435000	342	0.0004550463	49228	RESURS O1 DEB	DEBRIS	SMALL	5.00	4227	THORAD AGENA D DEB	DEBRIS	SMALL	5.00
307461791	2022-06-20 23:20:08.000000	Y	2022-06-22 11:22:52.311000	533	0.001012522	49123	CZ-4C R/B	ROCKET BODY	LARGE	5.00	36290	CZ-4B DEB	DEBRIS	SMALL	5.00
307460332	2022-06-20 23:19:58.000000	Y	2022-06-23 02:25:33.506000	402	0.0003635675	48760	NOAA 17 DEB	DEBRIS	SMALL	5.00	87878	UNKNOWN	UNKNOWN	SMALL	5.00

Showing 1 to 10 of 832 entries First Previous 1 2 3 4 5 ... 84 Next Last

Figure 7. Over 800 total publicly known conjunction events occurred on a single day, June 20, 2022.

Across the board, spacecraft operators are seeing more frequent use of collision avoidance maneuvers to dodge space debris. Additionally, spacecraft operators must commonly account for many different kinds of data and data sources in the course of their space operations. They must also deal with tens or hundreds of alerts related to space debris in the proximity of their spacecraft each year. A certain percentage of the alerts could be faulty, which imposes a higher cost of time and resources on sifting through the data. For a notable example, operators of the European Sentinel-2A spacecraft encountered over 8000 space debris collision alerts from 2015 to 2017 [12].

When a possible collision alert is labeled critical to the mission, a spacecraft performs a collision avoidance maneuver, using propellant and potentially missing out on data collection. This can last hours or a few days [12]. In 2017, United States Strategic Command sent hundreds of possible collision alerts to cooperative spacecraft operators and reported over ninety collision

avoidance maneuvers. The ISS, too, has experienced an uptick in collision avoidance maneuvers due to space debris. Seventeen collision avoidance maneuvers occurred from 2009 to 2017 whereas eight maneuvers were performed between 1999 and 2008 [12].

Risk of collision is expected to increase in a nontrivial manner over the coming decade. In 2013, before the uptick in CubeSat launches and plans for mega-constellations, models of the Inter-Agency Space Debris Coordination Committee estimated an average increase of 30% in the LEO space debris population over the next two centuries [31]. To add to this, catastrophic collisions were expected to occur every five to nine years even while assuming a compliance rate of 95% with mitigation guidelines and standards. In 2017, a University of Southampton study reported that increasing the spacecraft population by a single mega-constellation of thousands of satellites in LEO would cause an uptick in catastrophic collisions of 50% for the next two centuries [12].

Space traffic management is also a maturing process. Current processes for collision avoidance can be inefficient and manual in nature [12]. One key example involves the collision avoidance maneuver the ESA wind measurement satellite Aelous used to avoid the SpaceX Starlink-44 satellite. In September of 2019, Starlink-44 lowered itself to around 320 km altitude for de-orbit tests and came close to Aeolus. The calculated risk of collision surpassed the 1 in 10000 level of probability, so the United States Air Force sent a warning to SpaceX and the ESA. ESA operations personnel commanded Aeolus to perform a collision avoidance maneuver [12]. SpaceX and the ESA were in contact before and after the maneuver, but SpaceX later stated a communications issue stopped the transmission of the collision probability warning.

For the majority of objects in space, collision avoidance maneuvers are not possible. As the space debris object population expands, so too will the frequency of conjunction events that require collision avoidance maneuvers. Estimates have suggested that mega-constellations like Starlink could encounter as much as millions of warnings regarding conjunction events and would need to perform hundreds of thousands of collision avoidance maneuvers [12]. As a final side note, heightened issues and uncertainties with respect to space debris may deter investments into the space sector and cause redirection to terrestrial solutions for the relevant applications [12]. Preventative and active debris removal technologies would aid in the reduction of risk and cost to all space missions, especially in LEO.

While the presence of space debris creates costs and risks for larger satellite missions that can be addressed to a certain extent, CubeSats developers often have neither the resources to actively account for space debris during mission operations nor for making them sufficiently resilient to impact. CubeSats also typically do not have propulsion units or other motion-altering devices to quickly respond to a conjunction event even if the proper information regarding the incoming debris is available. Unlike larger satellite missions, CubeSats do not typically have the available mass to feature secondary shielding for protection from orbital debris.

1.5 Introducing NEIGHBOR

While some de-orbiting technologies require relatively low mass and little to no power, they can still take a long time to de-orbit a spacecraft. If a spacecraft encounters a mission-critical failure and cannot continue its operations, it is space debris by definition and is best de-orbited as soon as possible to minimize harm to other spacecraft. Failures also increase the likelihood of spacecraft break-up.

This thesis presents a design feasibility study for a novel, active de-orbit system for CubeSats called the Next Era Individualized Geo-conscious High Body Orbit Remediator (NEIGHBOR). When installed on a CubeSat mission, NEIGHBOR activates when its host spacecraft is no longer able to provide status or has completed its intended mission. In the sections that follow, a concept of operations is provided leading to the definition of a set of primary mission requirements from which NEIGHBOR's design evolved. Analysis is also provided to demonstrate NEIGHBOR's ability to satisfy these principal directives.

A system like NEIGHBOR can take many forms. On one end of the technical solution spectrum, it meets its objectives while embedded into the design of a satellite from its inception and uses the existing power, attitude control, communications, command and data handling systems, a de-orbit thruster, propellant supply, and fault detection algorithms. In the middle of the possible solutions, some of the host satellites systems are duplicated to enhance de-orbit reliability. At the far end of the scale, a self-contained, add-on box that operates independently from all systems on the satellite is possible. Such a system runs on its own battery and is capable

of downlinking short bursts of telemetry after it is awakened by the loss of contact with the host spacecraft and de-orbits it. This thesis examines the last configuration, evaluating impact to a satellite from accommodating the addition of a de-orbit system into its design.

CHAPTER 2: MISSION CONCEPTUALIZATION

2.1 Existing Debris Removal Technologies

A variety of technologies are being developed to mitigate the accumulation of space debris. Active debris removal methods are commonly defined as technical solutions that involve approaching space debris in Earth's orbit, capturing it, and then using an active or passive technique to de-orbit it. Some active debris removal methods involve spacecraft designed to harpoon space debris, capture it in a net, dock with it electromagnetically, attaching an enhanced drag device, or using propulsion to de-orbit a piece of debris.

Prime targets for removal are large rocket stages left in orbit from launches, failed or end-of-life satellites, and other pieces left over from collisions, explosions, or deployments. Commercial entities and government agencies are planning active debris removal missions. JAXA is in a partnership with the private sector to remove a large space debris object by 2025 [12]. Airbus and Thales Alenia Space are designing and manufacturing servicing vehicles that also carry debris removal capabilities. Some have been used in the RemoveDEBRIS mission [12]. Overall, as of January 2021, NASA's Office of Inspector General has reported that NASA has "made little to no progress" on developing active debris removal technologies [2]. In general, active debris removal technologies are still in the early stages of development and testing.

Passive de-orbit devices can help satellite operators save on mass with their spacecraft design [32]. Frequently, these types of solutions involve deployment of a drag-inducing device to accelerate the removal of a spacecraft from orbit. For example, electromagnetic tethers utilize electromagnetic forces to de-orbit a spacecraft. This involves long, electrically conductive strips of material that deploy from a spacecraft. Magnetic forces act on objects with currents running through them, and so Earth's magnetic field acts on the tether to de-orbit the spacecraft. Another method involves the use of solar sails. These types of sails interact with solar radiation pressure, which is the phenomenon of mechanical pressure being applied by radiation from the sun impinging on a surface. Depending on how direct the incoming radiation is, the solar sail

receives varying amounts of pressure that causes motion. Similarly, atmospheric drag sails can also be used to passively de-orbit spacecraft.

Some active systems use propulsion to bring objects out of orbit. Many chemical propulsion devices harness the energy of the chemical bonds of the selected propellant through combustion and use the high temperature, high velocity exhaust gas to impart a momentum change to the spacecraft. Electric propulsion devices use high currents or radio frequency energy to heat and emit high temperature, high velocity exhaust gas. The reaction forces of the motion of the emitted particles, applied in the opposite direction from a spacecraft's velocity vector, remove energy from the spacecraft's momentum and eventually lowers its altitude until atmospheric drag slows it enough for re-entry. Active de-orbit methods tend to increase the mass, volume, and power requirements for de-orbiting maneuvers. However, they also allow larger satellites to operate at higher altitudes because of their effectiveness in countering higher orbital velocities.

This thesis explores a novel use of independent attitude control and propulsion hardware to remove a non-functioning spacecraft from orbit. This type of debris mitigation is sometimes referred to as an end-of-life disposal. The independent system, previously introduced as NEIGHBOR, is designed as an active de-orbit system for CubeSat missions in LEO.

2.2 Relevant Guidelines

There is widespread interest in the reduction of the risk that space debris poses to current and future space assets. Civilian, commercial, government, and international entities have responded to the growing problem of space debris in a number of ways. In particular, requirements and guidelines have been developed with international collaboration. Notably, the Inter-Agency Space Debris Coordination Committee updated its landmark document titled "IADC Space Debris Mitigation Guidelines" in June of 2021 to a third revision [1]. In the introduction, three objectives are highlighted for mitigating space debris:

1. Preventing explosive and collisional on-orbit breakups.
2. Removing spacecraft and orbital stages that have reached the end of their mission operations from the useful densely populated orbit regions.
3. Limiting the objects released during normal operations.

To address these guidelines, NEIGHBOR's key capability, removing a failed CubeSat from orbit, significantly reduces the possibility that the satellite will explode or collide with other on-orbit objects. Additionally, NEIGHBOR's design eliminates the need for objects that are released from the host spacecraft to achieve de-orbiting.

2.3 Concept of Operations

NEIGHBOR's definition as a system starts with a Concept of Operations (Figure 8). NEIGHBOR remains in an idle mode onboard the host spacecraft until it detects the absence of a heartbeat signal (1). The absence of the heartbeat signal can result either from a mission-critical failure or a host spacecraft operator command following mission completion. Following a detection of failure, NEIGHBOR institutes a grace period to allow for ground intervention to potentially rectify the failure (2). After the grace period expires, NEIGHBOR detects spacecraft attitude and determines an optimal orientation in which to apply a retrograde de-orbit maneuver (3). The de-orbit maneuver sends the host spacecraft on a trajectory to an altitude where drag takes over and causes the spacecraft to quickly re-enter the atmosphere (4). This ensures that the spacecraft spends less time in orbit as it would when naturally decaying and is at far less risk of colliding with other objects in space that would cause the generation of more space debris. Finally, NEIGHBOR downlinks host spacecraft status information (5) before re-entering Earth's atmosphere (6).

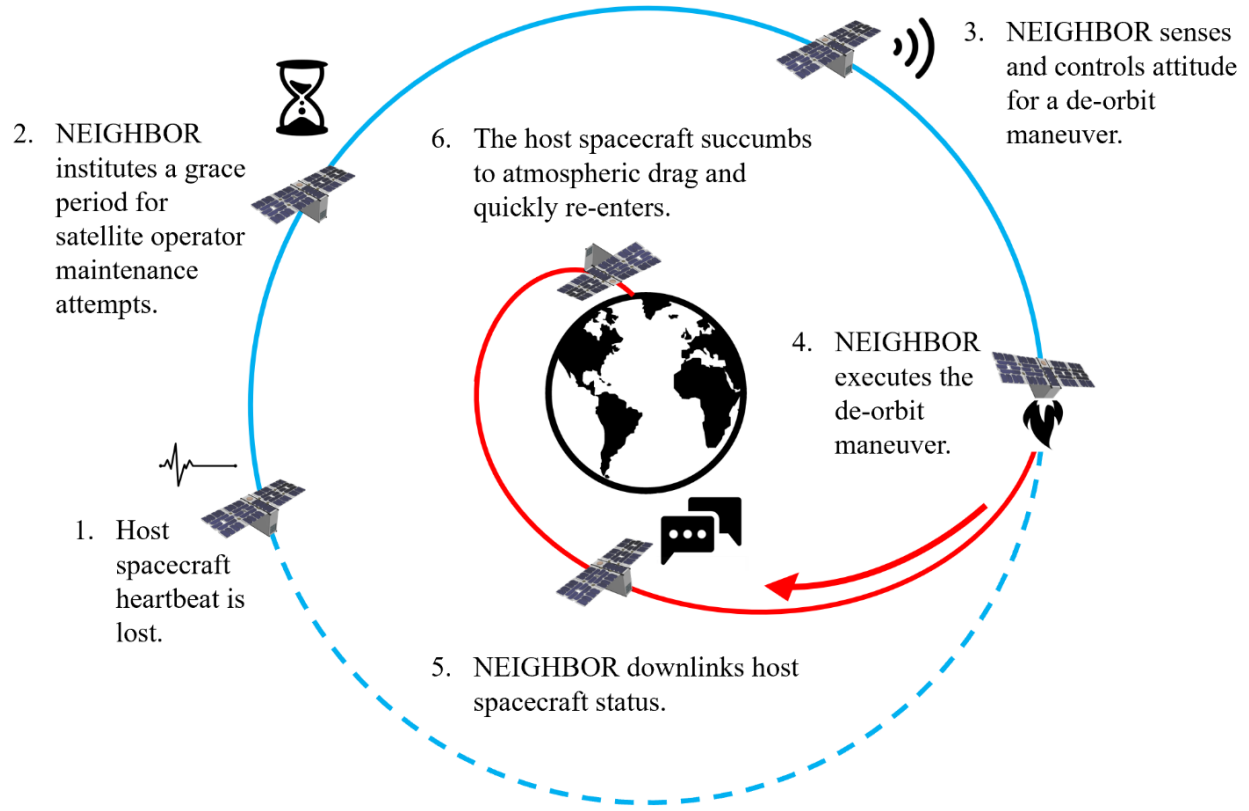


Figure 8. Concept of Operations for NEIGHBOR. Key events of the mission are highlighted and illustrated in a numbered sequence.

2.4 Capabilities and Requirements

Three key capability requirements are necessary for NEIGHBOR to complete its mission of preventing small satellites from becoming orbital debris. They are:

1. NEIGHBOR shall monitor host spacecraft status.
2. NEIGHBOR shall downlink confirmation of host spacecraft status upon loss of heartbeat signal.
3. NEIGHBOR shall, after a pre-determined grace period where the status does not change, de-orbit the host spacecraft.

The first requirement pertains to monitoring host spacecraft status. One of two conditions can initiate NEIGHBOR's de-orbit operations. The first is a failure that renders the spacecraft unable to continue its mission or leaves it with too much risk of becoming the newest addition to the orbital debris population. The second is completion of the host spacecraft's mission where it is no longer useful remaining in orbit. In either case, the host spacecraft becomes space debris and NEIGHBOR must know when to remove it from orbit.

The second requirement involves downlinking host spacecraft status data. This gives the client satellite operator the ability to understand the core health of the spacecraft if main communications are cut due to failure of a radio or other equipment. NEIGHBOR's downlinking of core health information creates a focus on key information that may aid the satellite operator in instituting a software fix or hardware reset.

The third requirement provides for the key capability of de-orbiting the host spacecraft once a mission-critical failure or mission completion is confirmed. As stated before, according to Newton's Laws, NEIGHBOR needs to leverage forces to cause the alteration of the state of the host spacecraft from its initial orbit along some trajectory to a re-entry point to complete its mission.

2.5 Orbit Dynamics

CubeSats in LEO are subject to two key forces that control and perturb their orbits: gravity and drag. Drag acts secularly to remove energy from a spacecraft's orbit while gravity perturbations are cyclical in nature and minimally impact the de-orbiting process. Depending on the initial orbit and physical characteristics of the spacecraft, the de-orbiting process can take several years or more in orbits below 600 km to upwards of decades and centuries above 1000 km [12].

FreeFlyer is an orbital analysis software tool made available by a.i. Solutions and is used to simulate the orbital dynamics of a host spacecraft that has been integrated with the NEIGHBOR de-orbit system. Free-Flyer has a Visual Basic like scripting language that was used

to customize analysis to characterize NEIGHBOR's performance (e.g., Appendix A). The starting date for each simulated scenario is January 1, 2023. The reference host spacecraft is described in further detail in Section 3.1. The impact of perturbing forces modelled in FreeFlyer are assessed to determine the operational effectiveness of NEIGHBOR.

A spacecraft's orbit is influenced by a planetary body's gravitational field. Earth does not have a uniform density distribution. With a spherical harmonics representation, Earth's gravitational potential is expressed in terms of zonal and tesseral terms. The zonal terms are a function of latitude relative to the central body while the tesseral terms are a function of both latitude and longitude. Both zonal and tesseral potential terms are activated for the default gravity potential model in FreeFlyer, EGM96, to simulate the effect of gravity on NEIGHBOR and its host spacecraft [33].

Atmospheric drag is the phenomenon of gas molecules at the top of Earth's atmosphere resisting the motion of the spacecraft. This force dominates the orbital decay of objects in LEO typically at altitudes near and below 350 km. The larger the surface area that faces the forward direction of motion along the spacecraft's orbit trajectory, the greater the force of drag. FreeFlyer's drag model is represented by Equation 1, where $\boldsymbol{F}_D$ is the atmospheric drag vector, ρ is the spatially-dependent atmospheric density, A_D is the drag surface area of the spacecraft facing the oncoming flow of molecules and particles, C_D is the drag coefficient of the spacecraft, and $\boldsymbol{v}$ is the velocity vector of the spacecraft [34]. Here, $\boldsymbol{v}$ is the velocity vector and $\breve{\boldsymbol{v}}$ is the unit velocity vector, i.e., it has a magnitude of one. The double vertical lines around the velocity vector, shown as $\|\boldsymbol{v}\|$, indicates the norm of the vector. The Jacchia-Roberts density model is used to simulate atmospheric drag on the host spacecraft [35].

$$\boldsymbol{F}_D = -\frac{1}{2}\rho A_D C_D \|\boldsymbol{v}\|^2 \breve{\boldsymbol{v}} \tag{1}$$

The FreeFlyer simulation uses a default Runge-Kutta propagator, which features a ninth order Runge-Kutta integration scheme that has eighth order numerical accuracy [36]. Sixteen intermediate integration stages occur between each simulation step to arrive at the next state with

this level of accuracy. It takes the activated forces within the force model and integrates them with respect to the selected time step to propagate the state of the host spacecraft forward in time. Earth's gravity with zonal and tesseral potential terms and drag with the Jacchia Roberts density model are activated in the force model to propagate the state.

CHAPTER 3: NEIGHBOR DESIGN

3.1 Reference Design Mission

Physical characteristics and orbit parameters are chosen for a reference host spacecraft to assess feasibility of NEIGHBOR's design. This configuration of NEIGHBOR is intended for 6U CubeSat missions that do not already carry their own propulsion systems but wish to integrate a de-orbit system package. The design reference mission is baselined around a host spacecraft with its system and payload occupying a volume of 4U while NEIGHBOR takes up the remaining 2U. Together, this is a spacecraft with a total volume of 6U with dimensions as depicted in Figure 9. The spatial breakdown between the host spacecraft and NEIGHBOR is shown in Figure 10, with the solar array removed form NEIGHBOR's side for illustrative purposes. The total mass of the integrated system is 12 kg.

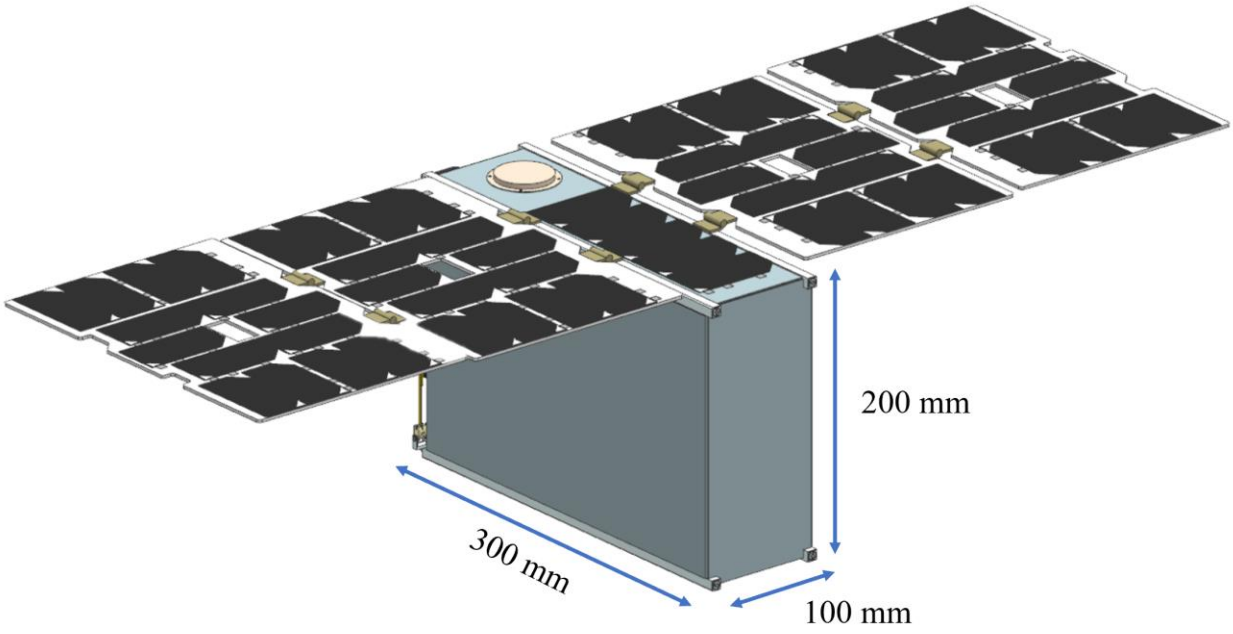


Figure 9. The design reference mission is baselined around a 6U CubeSat with a double deployable solar array.

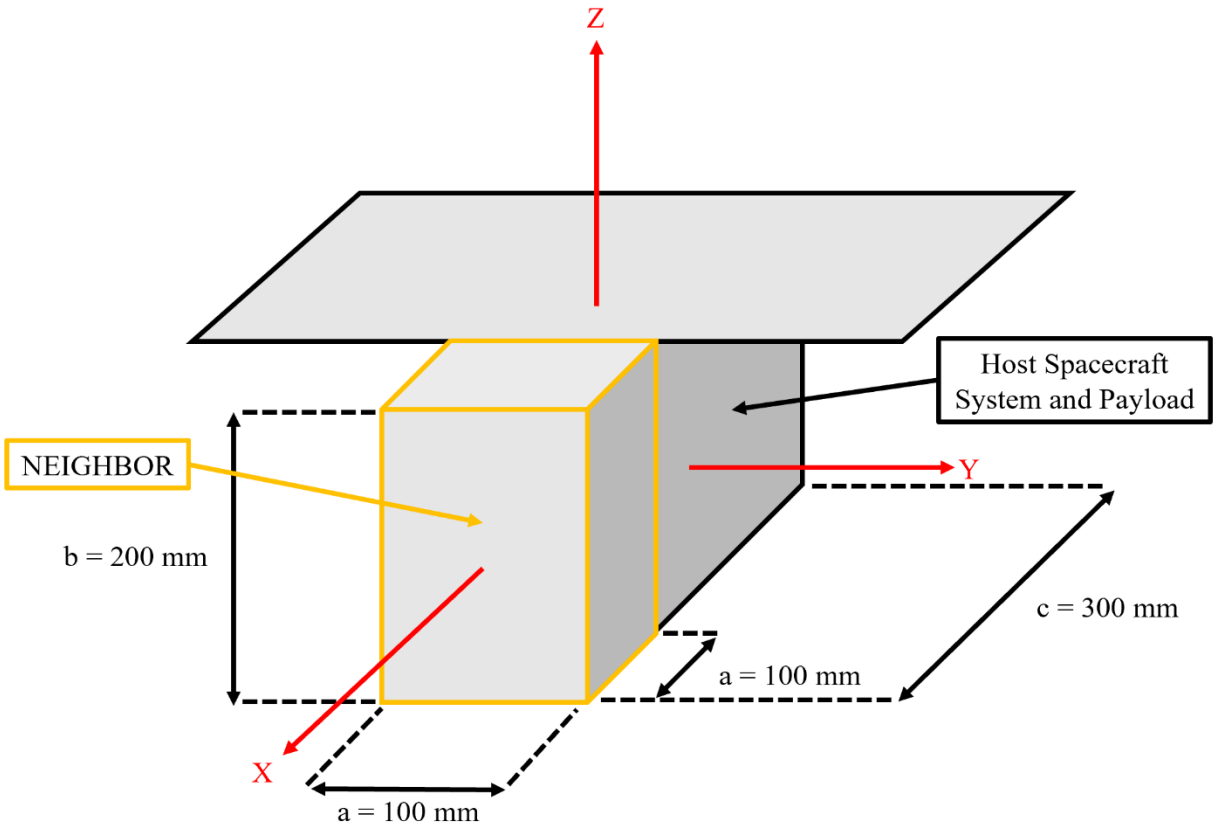


Figure 10. The host spacecraft's system and payload occupy a volume of 4U and NEIGHBOR occupies a volume of 2U (highlighted in gold) to form a complete CubeSat with a volume of 6U. The solar panel above NEIGHBOR is removed for illustrative purposes.

For the de-orbit simulations, a worst-case (i.e., minimal) drag condition is assumed. The minimum surface area expected to experience drag is used. That is represented by the 100 mm x 200 mm (surface area of 0.02 m^2) forward looking faceplate. The physical characteristics of the host spacecraft that were used in FreeFlyer mission simulations are summarized in Table 2. The host spacecraft's drag coefficient is set to 2.2.

Table 2. Physical characteristics of the host spacecraft integrated with NEIGHBOR are summarized for usage in FreeFlyer simulations.

Physical Characteristic	Value
Maximum Total Mass	12 kg
Volume	6 U
Drag coefficient	2.2
Drag area	0.02 m^2

3.2 Applicable Orbits

Some reference deployment orbits are considered to construct de-orbit scenarios for analysis. It is common for CubeSats to be deployed from the ISS. This circular orbit has an initial altitude of 420 km and 51.6 degrees of inclination. The initial semi-major axis is the sum of the initial altitude and the equatorial radius of the Earth, set as 6378.1363 km. The right ascension of the ascending node, argument of perigee, and true anomaly are each arbitrarily set to zero degrees. The Keplerian orbital elements defining the initial orbit of the first de-orbit scenario are listed in Table 3. The only difference between the de-orbit scenarios constructed in the following analysis is the initial semi-major axis, defined by an initial orbit altitude.

Table 3. The initial orbit of the first de-orbit scenario is defined by Keplerian orbital elements.

Orbital Element	Value
Semi-major Axis, A	6798.1363 km
Eccentricity, e	0
Inclination, i	51.6°
Argument of Perigee, ω	0°
Right Ascension of the Ascending Node, Ω	0°
True Anomaly, v	0°

For each reference mission, the host spacecraft is assumed to be deployed into a circular orbit. After the host satellite fails, NEIGHBOR takes control of the satellite's attitude and performs a single de-orbit maneuver after the grace period terminates. All maneuvers are modelled as discrete continuous operations of the propulsion system. During the maneuvers, the simulation uses a time step of 0.5 seconds. Outside of the maneuvers, it uses a time step of 20 minutes.

The de-orbit maneuver lowers the perigee of the host spacecraft to set it on an orbit with a remaining lifetime of less than five years, after which the host spacecraft re-enters Earth's atmosphere. For the first de-orbit scenario, the target perigee altitude is set at 350 km. This leaves the host spacecraft in a region of Earth's atmosphere where its orbit rapidly succumbs to atmospheric drag and further descends to an altitude of 250 km as the designated point of re-entry. De-orbit times are calculated from the start of the de-orbit maneuver to the re-entry point. De-orbit times are also determined for a reference host spacecraft re-entering without NEIGHBOR for comparison; for this, the mass of the host spacecraft is a fixed 12 kg.

The ΔV necessary to execute the de-orbit maneuver can be calculated the equation shown in Equation 2, where μ is Earth's gravitational parameter, 398600.4415 km^3/s^2, r_1 is the orbit radius of the initial circular orbit, and r_2 is the perigee distance of the post-maneuver orbit [3]. Since the initial orbit is circular, the initial semi-major axis value is also the initial orbit radius value used to calculate the required ΔV. As a result, the ΔV required in the first de-orbit scenario to send the host spacecraft from an initial altitude of 420 km to a target altitude of 350 km is 19.84 m/s.

$$\Delta V = \sqrt{\frac{\mu}{r_1}}\left(\sqrt{\frac{2r_2}{r_1 + r_2}} - 1\right) \tag{2}$$

The rocket equation is used to determine the ΔV that a thruster delivers with a particular specific impulse, I_{sp}, spacecraft starting mass, m_0, and spacecraft final mass, m_f. The propellant mass used for a particular maneuver is computed as the difference between the initial mass and

the final mass. Equation 3 symbolizes the relationship, where g_0 is the acceleration due to gravity at Earth's surface, 9.81 m/s^2.

$$\Delta V = I_{sp} g_0 \ln\left(\frac{m_0}{m_f}\right) \tag{3}$$

To simulate the de-orbit maneuver, representative thruster performance characteristics are needed. A typical CubeSat propulsion system that would be applicable to this mission provides 0.1 N of thrust from four individual axial thrusters, each delivering 25 mN of thrust [37]. The specific impulse for this reference thruster is 42 seconds [38]. The exact amount of propellant NEIGHBOR carries is tailored to the mission of a specific host spacecraft.

Assuming an initial host spacecraft total mass of 12 kg, the thruster utilizes a propellant mass of 0.564 kg to deliver a ΔV of 19.84 m/s. The maneuver lasts 38.74 minutes. The de-orbit time without NEIGHBOR is 4.38 years while the de-orbit time with NEIGHBOR is reduced to 2.20 years. The altitude plots for each case are shown in Figure 11. Altitude oscillations arise from the influence of Earth's gravitational field on the host spacecraft orbit.

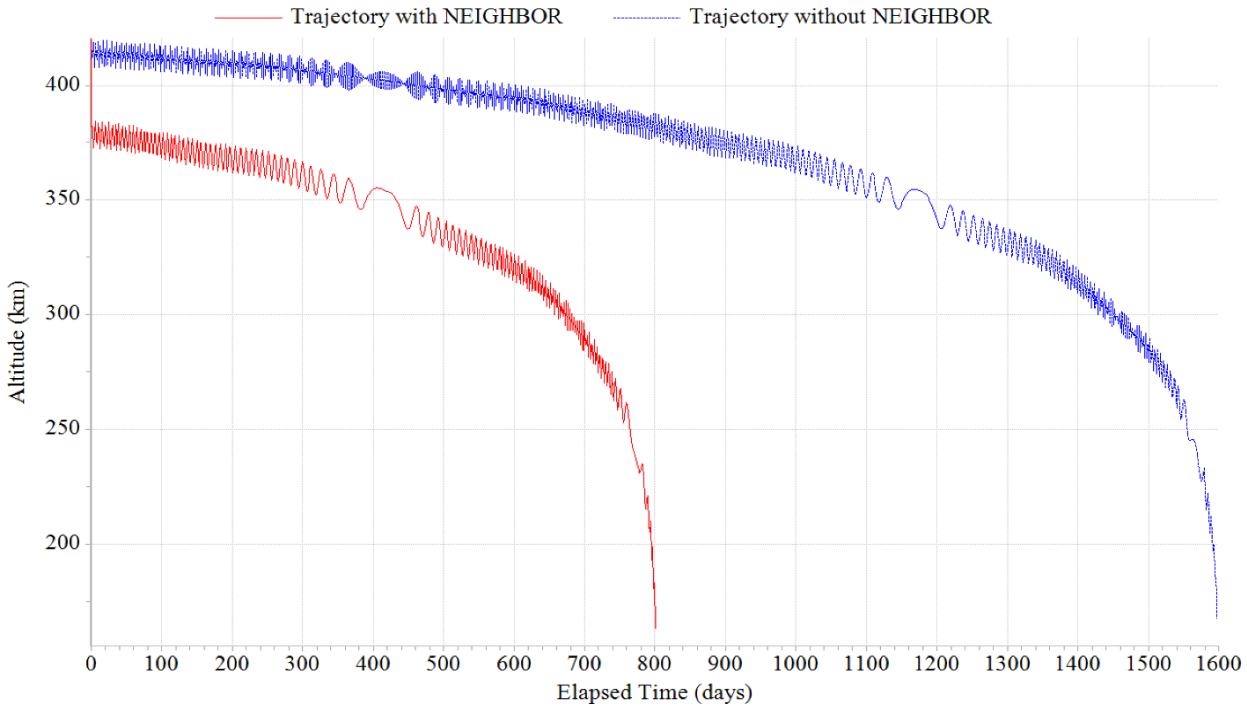


Figure 11. Orbital lifetime results are visualized for an initial altitude of 420 km.

Another common CubeSat deployment orbit is that provided by Rocket Labs' Electron launch vehicle. The rocket and its kick stage can deliver small satellites to a 475 km deployment altitude [39], setting the initial semi-major axis of the second de-orbit scenario. The orbital lifetime curves with and without NEIGHBOR's selected propulsion system are presented in Figure 12. The ΔV required to lower the host spacecraft from an initial altitude of 475 km to a target altitude of 350 km is 35.18 m/s. This corresponds to a propellant mass of 0.982 kg. The burn time for the maneuver is 67.42 minutes. The de-orbit time without NEIGHBOR is 25.04 years while the de-orbit time with NEIGHBOR is 3.44 years. NEIGHBOR significantly reduces the orbit lifetime of the host spacecraft and makes it compliant with current Federal Communications Commission regulations.

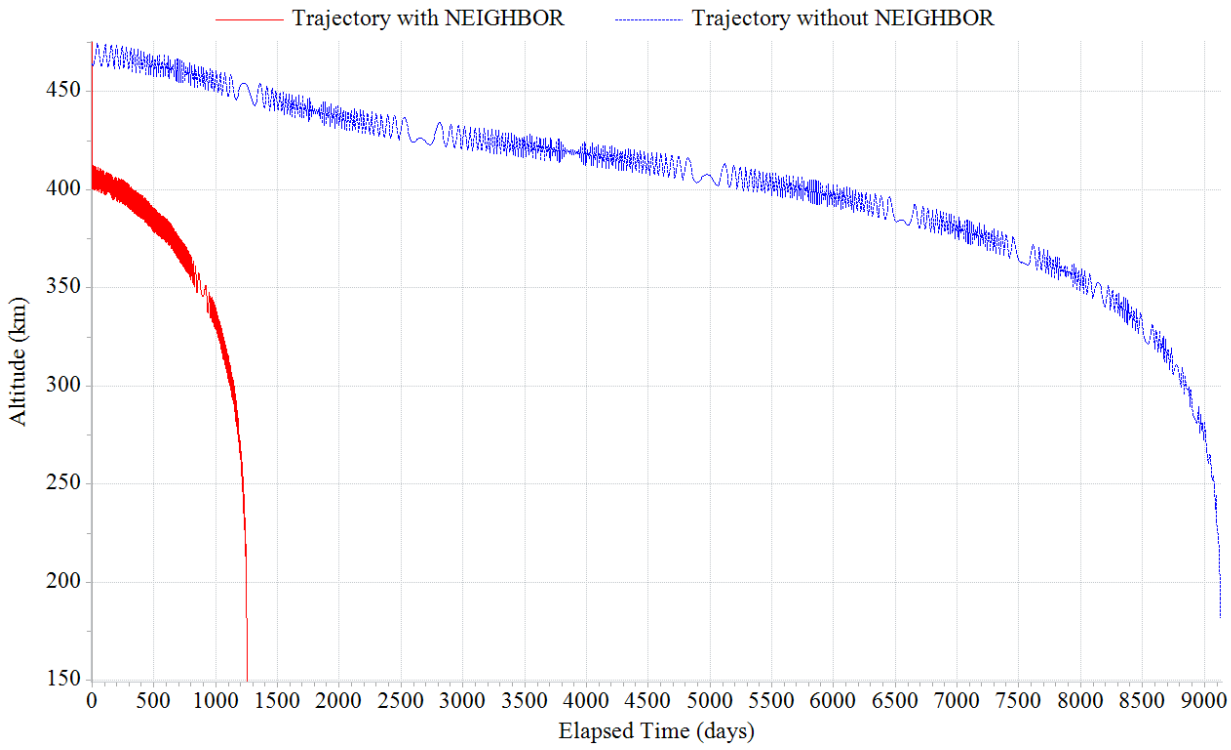


Figure 12. Orbital lifetime results are presented for an initial altitude of 475 km.

For the third de-orbit scenario, the Electron Rocket can also deliver CubeSats to 525 km. The results are displayed in Figure 13 for this deployment altitude. The ΔV required for the de-orbit maneuver at this altitude is 63.21 m/s to send the host spacecraft on a trajectory with a different, lower target altitude of 300 km. This corresponds to a required propellant mass of 1.707 kg. The maneuver takes 117.20 minutes. The de-orbit time with NEIGHBOR is 3.19 years while the de-orbit time without NEIGHBOR is 97.41 years. Once more, NEIGHBOR significantly reduces orbital lifetime and the debris generation threat posed by the spacecraft. NEIGHBOR causes the host spacecraft to be compliant with the Federal Communications Commission's de-orbit time requirement of five years.

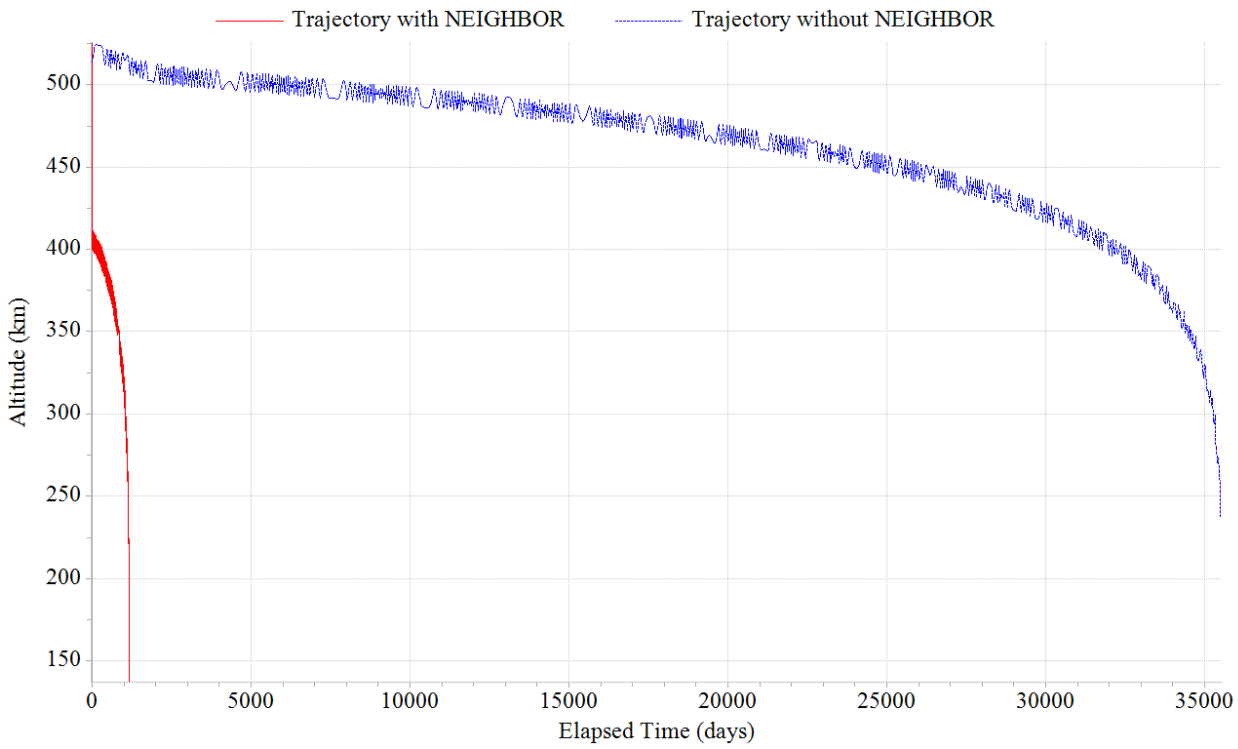


Figure 13. Orbital lifetime results are displayed for an initial altitude of 525 km.

3.3 Attitude Determination and Control

All host spacecraft reference failure scenarios envision the attitude sensors or control actuators failing, and termination of the heartbeat signal. NEIGHBOR then wakes up, waits out the grace period, and then determines the attitude of the host satellite before taking control of it to maneuver the satellite to the de-orbit attitude. This section describes the process, hardware, and algorithms NEIGHBOR will use to carry out these functions.

3.3.1 Attitude Determination

NEIGHBOR uses a horizon sensor in the host spacecraft attitude determination process [40]. Horizon sensors use the infrared band of the light spectrum to detect transition points along

Earth's horizon to background space [41]. As NEIGHBOR travels along its orbit, it scans the horizon in a circular fashion to distinguish the transition points, as depicted in Figure 14. The device takes multiple views of Earth's limb and the differentiated transition points to output the spacecraft nadir vector, pointing from the spacecraft to the center of the Earth [42]. This vector is used when constructing a basis of vectors for the host spacecraft's body frame axes.

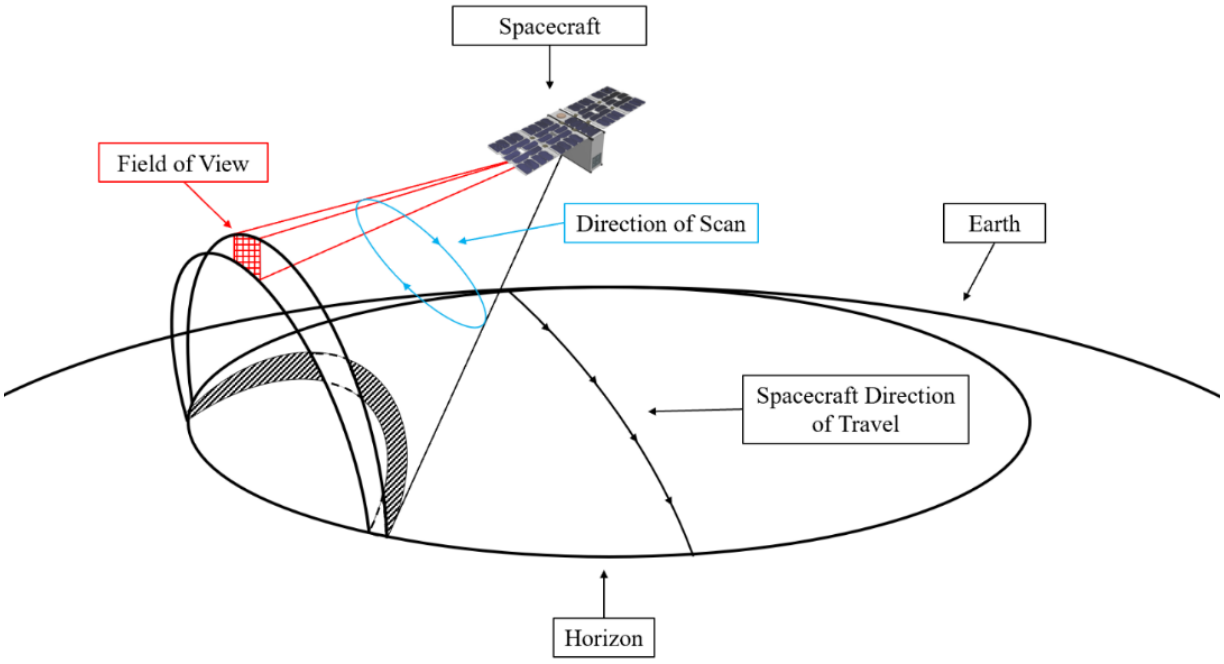


Figure 14. A horizon sensor provides Earth and nadir sensing capabilities.

Additional attitude information comes from a sun sensor [43]. Sun sensors are photo-sensitive devices that measure the incident angle of light from the sun to determine spacecraft orientation with respect to the spacecraft body frame axes, as seen in Figure 15. NEIGHBOR's digital sun sensor interprets the angle of incidence and converts it to a digital signal that is usable in determining a sun-to-spacecraft vector.

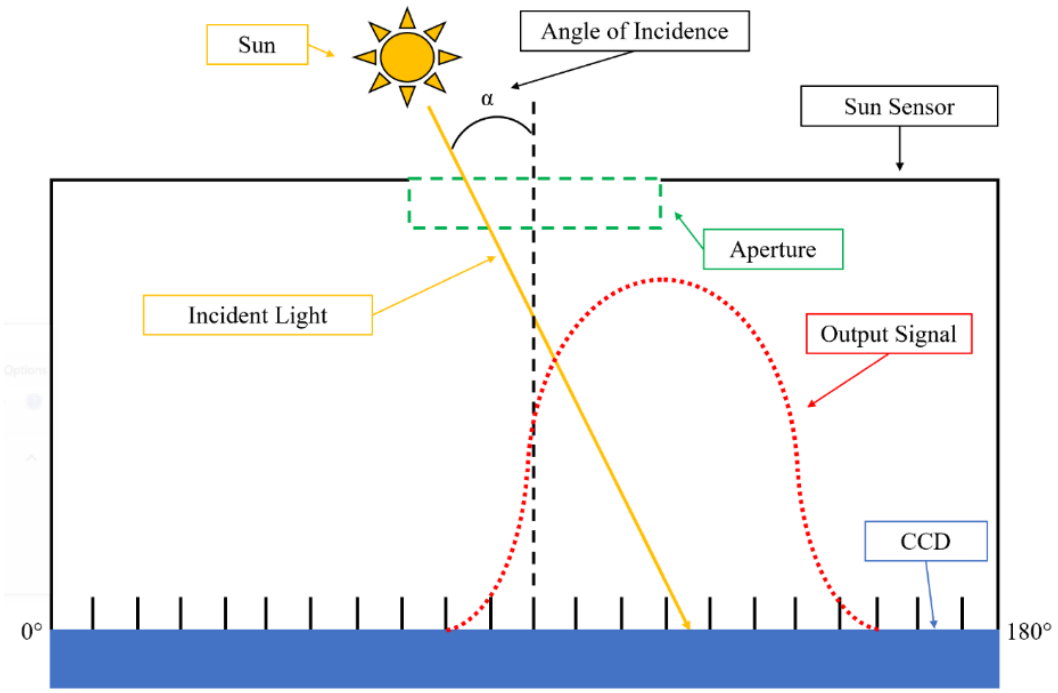


Figure 15. A sun sensor measures the angle of incidence of the Sun's radiation to determine attitude.

An attitude estimation algorithm obtains attitude knowledge from the horizon sensor and sun sensor [41]. The method assumes knowledge of two unit vectors in the host spacecraft's body frame as well as a reference inertial frame and also assumes that one measurement is more accurate than the other.

With an initial estimate for the spacecraft attitude, filtered gyroscope measurements are integrated to propagate the attitude in time. A 3-axis gyro integrated with NEIGHBOR's flight computer uses an extended Kalman filter to increase the accuracy of spacecraft attitude knowledge [44].

A Global Positioning System (GPS) receiver is also incorporated in NEIGHBOR's design to provide orbital position and time information for the spacecraft [45] [46]. With knowledge of the spacecraft attitude, position, and time, NEIGHBOR maneuvers the spacecraft to the burn attitude for the de-orbit maneuver. NEIGHBOR is equipped with reaction control system (RCS) thrusters to change the host spacecraft attitude. NEIGHBOR takes control of the host spacecraft by performing a detumble maneuver to stop any residual rotation, then performs

a slew maneuver to achieve the de-orbit maneuver attitude, and finally, holds that attitude throughout the de-orbit burn.

3.3.2 Attitude Control

The FreeFlyer scripting language was used to incorporate attitude dynamics simulations for NEIGHBOR's detumble maneuver, slew maneuver, and de-orbit attitude control maneuver. These simulations use control laws based on state errors and tolerances. The goal is to generate estimates for required propellant mass to determine feasibility and impact to the design of the host spacecraft. All simulations used a time step of 15 ms.

For these maneuvers, NEIGHBOR uses pairs of single RCS thrusters positioned on the center points of its external edges to apply torques and control the rotational dynamics of the host spacecraft. The thrusters are represented by blue cones in Figure 16. Those thrusters placed on the corners of the face intersected by the x-axis are used in the retrograde de-orbit maneuver. The performance of the RCS thrusters is the same as those that were used for the de-orbit maneuvers described in the previous section. The applied force of a single thruster is 25 mN.

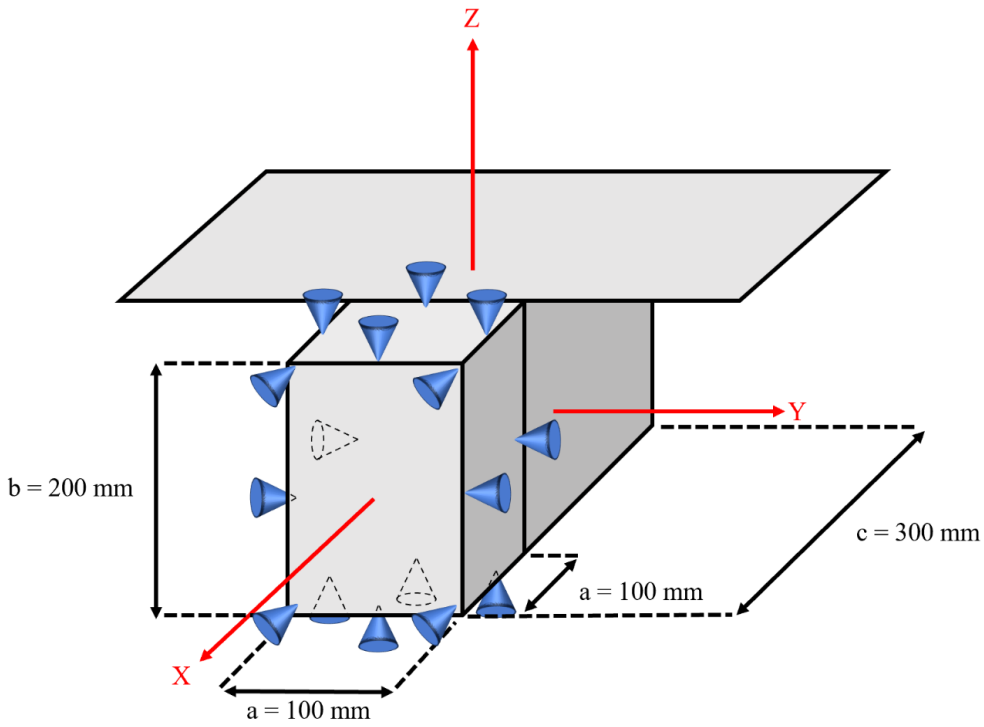


Figure 16. NEIGHBOR uses RCS thrusters (marked by blue cones and dashed lines when out of view) positioned on the center points of its edges to control the attitude of the host spacecraft.

For these simulations, the 6U reference host spacecraft's rotational dynamics are modelled. The spacecraft is assumed to be a rigid body in the shape of a rectangular prism with a uniform mass distribution. The moment of inertia I_i about an axis i of a rectangular prism is computed with Equation 4, where m is the mass of the rectangular prism, and a_i and b_i are the side lengths of the object cross section that the axis intersects. The computed moment of inertia about the x-axis, I_x, is 0.05 kg-m^2, about the y-axis, I_y, it is 0.13 kg-m^2, and about the z-axis, I_z, it is 0.10 kg-m^2. The moment of inertia tensor in the body frame, I_B, contains these values along its diagonal and is shown in Equation 5. The integrated spacecraft's side lengths and axis orientations used for these calculations are shown in Figure 16.

$$I_i = \frac{1}{12} m(a_i^2 + b_i^2) \tag{4}$$

$$I_B = \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix} = \begin{bmatrix} 0.05\ kg\text{-}m^2 & 0 & 0 \\ 0 & 0.13\ kg\text{-}m^2 & 0 \\ 0 & 0 & 0.10\ kg\text{-}m^2 \end{bmatrix} \quad (5)$$

3.3.3 Detumble Maneuver: Validation Case 1

The first validation case for the detumble maneuver has an initial orientation that corresponds to the identity quaternion. Quaternions are used to model and propagate the attitude of the spacecraft in the simulations. The attitude of the satellite is unconstrained after detumbling. The quaternion corresponding to the identity quaternion initializes the simulation and is shown in Equation 6.

$$\boldsymbol{q}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad (6)$$

For the detumble maneuver validation cases, the initial angular rate about each body axis is assumed to be ten degrees per second. This corresponds to the expected maximum tip-off rate described by the user manual for the NanoRacks External CubeSat Deployer aboard the ISS and therefore reflects a reasonable maximum rotation rate for which a satellite of this type would be designed [47]. The initial angular velocity, $\boldsymbol{\omega}_0$, is set in Equation 7.

$$\boldsymbol{\omega}_0 = \begin{bmatrix} 10°/s \\ 10°/s \\ 10°/s \end{bmatrix} \quad (7)$$

The RCS thrusters are commanded to apply torques to reduce the angular rates about each axis to zero degrees per second. The desired angular velocity, $\boldsymbol{\omega}_d$, is described by Equation 8.

$$\boldsymbol{\omega}_d = \begin{bmatrix} 0°/s \\ 0°/s \\ 0°/s \end{bmatrix} \tag{8}$$

The detumbling control algorithm takes the form of a Proportional Derivative (PD) controller to drive the initial angular velocity towards zero. The PD controller relies on a position error term, e_p, and a rate error term, e_d, for an arbitrary axis of rotation. The error terms are weighted by the proportional gain, k_p, and the derivative gain, k_d, to control the relative emphasis of control between the sources of state error. The controller output, u, is defined in general form for a PD controller in Equation 9.

$$u = k_p e_p + k_d e_d \tag{9}$$

As the simulation proceeds, the control law used for the detumble maneuver determines the torque to be applied to each body axis. For an arbitrary axis of rotation, the proportional error term, $e_{p,i}$, defined in Equation 10 is the difference between the desired (zero) rate, $\omega_{d,i}$, and the actual rate, ω_i. The negative of the time derivative of the angular velocity for a particular axis, $\dot{\omega}_i$, constitutes the derivative error term, $e_{d,i}$, to damp the change in the angular velocity. The derivative error term is defined in Equation 11.

$$e_{p,i} = \omega_{d,i} - \omega_i \tag{10}$$

$$e_{d,i} = -\dot{\omega}_i \tag{11}$$

Substituting the definitions of Equation 10 and Equation 11 into Equation 9 yields the output torque for an arbitrary axis of rotation, L_i, in Equation 12.

$$L_i = k_{p,i}\left(\omega_{d,i} - \omega_i\right) - k_{d,i}\dot{\omega}_i \tag{12}$$

Since discrete thruster pulses are being used to apply the control torque, the output torque is put through a threshold test to determine if the required torque value exceeds that provided by a single thruster pulse. If the required torque is greater in magnitude than the threshold, the thruster is fired until the threshold test fails. If an output torque has a smaller magnitude than the threshold, the angular velocity error is deemed insufficient to warrant a thruster pulse, and no torque is applied. The appropriate thruster is fired based on the sign of the error terms.

The torques applied by the thrusters, $\boldsymbol{L}_{app}$, are calculated using Equation 13, where $\boldsymbol{l}$ is the moment arm vector with respect to the host spacecraft center of mass, and $\boldsymbol{F}$ is the thruster force vector. The magnitudes of the applied torques are determined with Equation 14, where θ is the angle between the moment arm vector and the force vector.

$$\boldsymbol{L}_{app} = \boldsymbol{l} \times \boldsymbol{F} \tag{13}$$

$$\left\|\boldsymbol{L}_{app}\right\| = l * F * \sin(\theta) \tag{14}$$

The applied torques can be determined from the arrangement of the thrusters, the force they apply, and the dimensions of the host spacecraft as visualized in Figure 17, Figure 18, and Figure 19. The force applied by a single RCS thruster is denoted by F and the side lengths a, b, and c of the host spacecraft are used as shown in Figure 16.

In Figure 17, the thrusters that apply torques about the x-axis are shown in the 3D model to the left. The thrusters that operate as a pair to apply a positive torque to induce counterclockwise rotation are highlighted in yellow. The other two thrusters shown in blue generate a negative torque and induce clockwise rotation about the x-axis when operated as a pair. The thrusters of each pair apply equal and opposite forces, so no translational motion is induced. In the observed operation case, the operating thrusters highlighted in yellow both apply positive torques. The diagram to the right of the figure shows the lateral dimension of the cross section, “a,” and the moment arm, “r,” to describe the distance between the applied forces on the edges of the cross section and the axis of rotation, the x-axis. The applied force of a single thruster, “F,” takes the value of a single thruster, 25 mN.

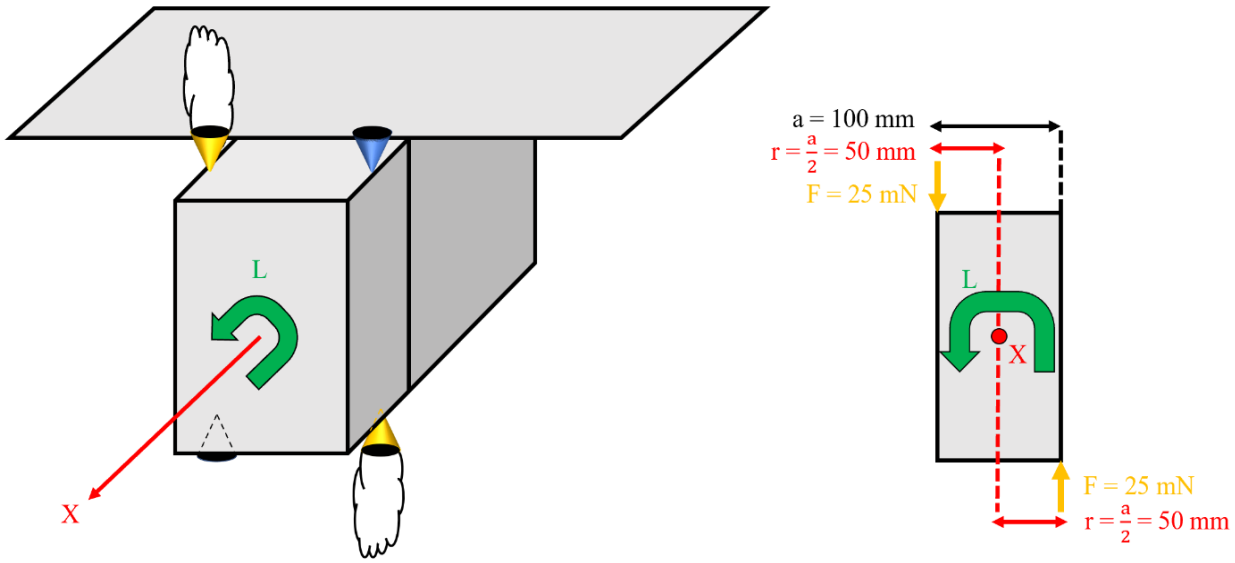


Figure 17. The magnitude of the torque applied due to the RCS thrusters about the x-axis is determined.

In Figure 18, the thrusters that generate torques about the y-axis are shown. The thruster pair that generates a positive torque is highlighted in yellow. The diagram to the right of the figure shows the width of the spacecraft cross section that the y-axis intersects, "c." The dimensions for the moment arms r_1 and r_2 are based on this width and the side length of NEIGHBOR, "a." NEIGHBOR is sized to fit within a 2U volume with a 100 mm gap denoted by "a" between the thruster pairs. This still leaves a 50 mm gap between the rightmost thruster and the y-axis, which goes through the center of the cross section. In this operating case, the highlighted operating thruster furthest from the y-axis creates a positive torque that is larger in magnitude than the negative torque generated by the operating thruster closest to the y-axis, creating a net positive torque to induce counterclockwise rotation about the y-axis. Since the thrusters apply equal and opposite forces, translational motion is negated.

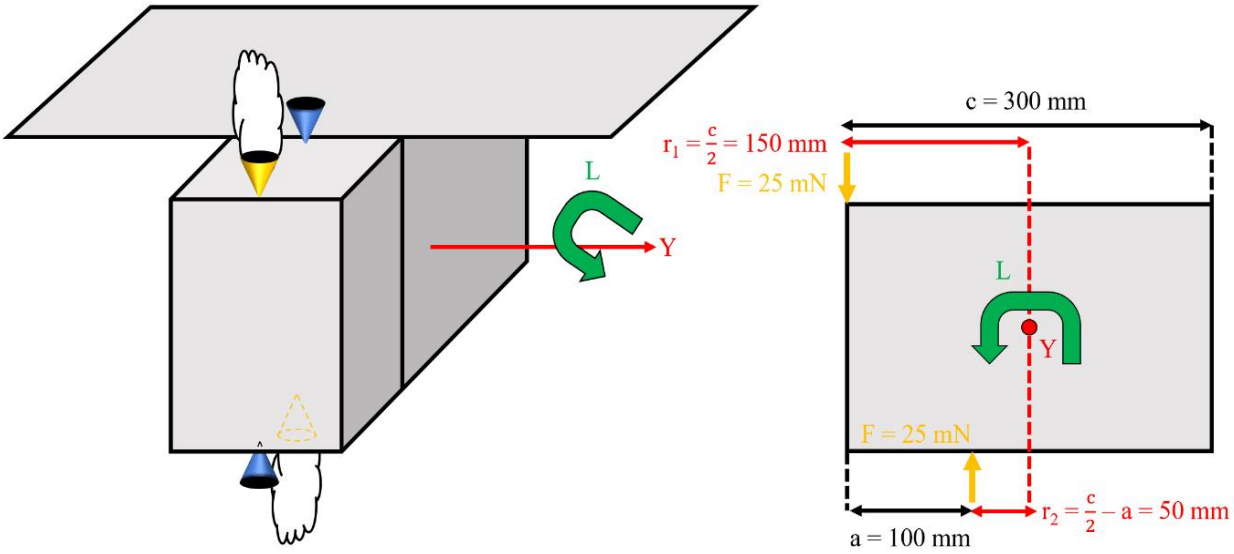


Figure 18. The magnitude of the torque applied due to the RCS thrusters about the y-axis is determined.

In Figure 19, the thrusters that generate torques about the z-axis are highlighted in yellow on the left. The thrusters that generate a positive torque as a pair have thruster plumes and the ones that generate a negative torque do not. The diagram to the right of the figure shows the width of the spacecraft cross section, "c," that the z-axis intersects. The dimensions for the moment arms of this setup, r_1 and r_2, are based on this width. In this case, the thruster furthest from the z-axis creates a positive torque that is larger in magnitude than the negative torque generated by the thruster closest to the z-axis, creating a net positive torque to induce counterclockwise rotation about the z-axis. Since the thrusters apply equal and opposite forces, translational motion is negated.

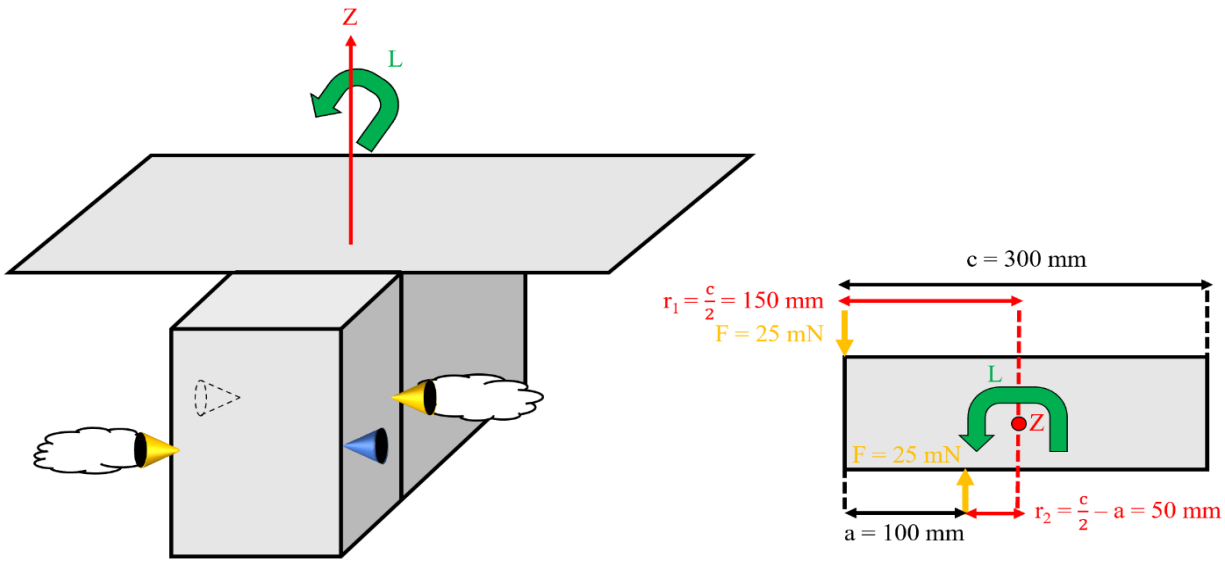


Figure 19. The magnitude of the torque applied due to the RCS thrusters about the z-axis is determined.

All thruster and moment arm pairs are at right angles with each other, so the magnitude of the torque as calculated in Equation 14 reduces to the product of the moment arm length and the applied force. Net torques about each axis can be calculated in Equation 15 by summing the products of the moment arm lengths and the applied forces. Each torque applied by a single thruster is assigned a direction with a positive or a negative sign depending on the direction of rotational motion the torque would induce, where counterclockwise is positive and clockwise is negative.

$$L_{net,i} = \sum(F_i * l_i) \tag{15}$$

The magnitudes of the torques applied about each axis are collected in Equation 16, where $\boldsymbol{L}_{thruster}$ is the array containing torques applied by the thrusters and $L_{net,x}$, $L_{net,y}$, and $L_{net,z}$ are the net torques applied about the x-axis, the y-axis, and the z-axis, respectively.

$$\boldsymbol{L}_{thruster} = \begin{bmatrix} L_{net,x} \\ L_{net,y} \\ L_{net,z} \end{bmatrix} = \begin{bmatrix} F\left(\frac{a}{2}\right) + F\left(\frac{a}{2}\right) \\ F\left(\frac{c}{2}\right) - F\left(\frac{c}{2} - \frac{c}{3}\right) \\ F\left(\frac{c}{2}\right) - F\left(\frac{c}{2} - \frac{c}{3}\right) \end{bmatrix} = \begin{bmatrix} 0.0025\ N\text{-}m \\ 0.0025\ N\text{-}m \\ 0.0025\ N\text{-}m \end{bmatrix} \tag{16}$$

The control torques are used as input for the equations of motion that govern the rotational dynamics of the host spacecraft. The equation that governs the time rate of change of the angular velocity vector, $\dot{\boldsymbol{\omega}}$, is derived from Euler's equation of rigid body dynamics and is represented in Equation 17 [41], where $\boldsymbol{L}_{ext}$ is the current external torque experienced by the rigid body. The angular velocity is updated by using the time rate of change of the angular velocity in Equation 18, where $\boldsymbol{\omega}_{new}$ is the new angular velocity, $\boldsymbol{\omega}$ is the current angular velocity, and Δt is the simulation time step in seconds.

$$\dot{\boldsymbol{\omega}} = I_B^{-1}[\boldsymbol{L}_{ext} - \boldsymbol{\omega} \times (I_B \boldsymbol{\omega})] \tag{17}$$

$$\boldsymbol{\omega}_{new} = \boldsymbol{\omega} + \dot{\boldsymbol{\omega}} * \Delta t \tag{18}$$

Attitude is updated by calculating the time rate of change of the quaternion in Equation 19, where $\dot{\boldsymbol{q}}$ is the time rate of change of the quaternion and $\Xi(\boldsymbol{q})$ is the Xi operator function applied to the current quaternion. The Xi operator function is defined in Equation 20. The quaternion is integrated and updated in Equation 21, where $\boldsymbol{q}_{new}$ is the new quaternion and $\boldsymbol{q}$ is the current quaternion.

$$\dot{\boldsymbol{q}} = \frac{1}{2}\Xi(\boldsymbol{q})\boldsymbol{\omega} \tag{19}$$

$$\Xi(\boldsymbol{q}) = \begin{bmatrix} q_4 & -q_3 & q_2 \\ q_3 & q_4 & -q_1 \\ -q_2 & q_1 & q_4 \\ -q_1 & -q_2 & -q_3 \end{bmatrix} \tag{20}$$

$$\boldsymbol{q}_{new} = \boldsymbol{q} + \dot{\boldsymbol{q}} * \Delta t \tag{21}$$

Quaternions are difficult to interpret for visualization of the orientation of the host spacecraft. The initial quaternion is converted to a set of Euler angles to provide attitude state information that is easier to interpret. First, the quaternion is converted to its associated attitude matrix, D, in Equation 22 [41]. The subscript "1:3" symbolizes the array containing the first three elements of a quaternion. The matrix I_3 symbolizes the three-by-three identity matrix, represented in Equation 23. The matrix symbolized by $[\boldsymbol{q}_{1:3}\times]$ is the cross-product matrix derived from the array containing the first three elements of the quaternion, defined in Equation 24.

$$D = (q_4^2 - \|\boldsymbol{q}_{1:3}\|^2)I_3 - 2q_4[\boldsymbol{q}_{1:3}\times] + 2\boldsymbol{q}_{1:3}\boldsymbol{q}_{1:3}^T \tag{22}$$

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{23}$$

$$[\boldsymbol{q}_{1:3}\times] = \begin{bmatrix} 0 & -q_3 & q_2 \\ q_3 & 0 & -q_1 \\ -q_2 & q_1 & 0 \end{bmatrix} \tag{24}$$

The attitude matrix can be computed for a particular Euler angle sequence [41]. A 1-2-3 Euler angle sequence is used, i.e., three sequential rotations can be used to arrive at the attitude described by the Euler angles: first by an angle ϕ about the x-axis (represented by the unit vector $\hat{\boldsymbol{e}}_1$), then by an angle θ about the y-axis (represented by the unit vector $\hat{\boldsymbol{e}}_2$), and finally by an angle ψ about the z-axis (represented by the unit vector $\hat{\boldsymbol{e}}_3$). The Euler angle sequence is expressed as a chain of products of the attitude matrices representing the individual rotations in Equation 25. The attitude matrix representing the final orientation of the Euler angle sequence is computed and expressed in terms of the Euler angles in Equation 26.

$$D_{123}(\phi, \theta, \psi) = D(\hat{\boldsymbol{e}}_3, \psi)D(\hat{\boldsymbol{e}}_2, \theta)D(\hat{\boldsymbol{e}}_1, \phi) \tag{25}$$

$$D_{123}(\phi,\theta,\psi) = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta\cos\psi & \sin\phi\sin\theta\cos\psi + \cos\phi\sin\psi & \sin\phi\sin\psi - \cos\phi\sin\theta\cos\psi \\ -\cos\theta\sin\psi & \cos\phi\cos\psi - \sin\phi\sin\theta\sin\psi & \cos\phi\sin\theta\sin\psi + \sin\phi\cos\psi \\ \sin\theta & -\sin\phi\cos\theta & \cos\phi\cos\theta \end{bmatrix} \quad (26)$$

The Euler angles are computed from the trigonometric expressions in Equation 26 using the elements of the known attitude matrix, where the arbitrary attitude matrix element D_{ij} denotes the attitude matrix residing in the row numbered by "i" and the column numbered by "j." Equation 27 is used to calculate ϕ (the angular position with respect to rotation about the x-axis), Equation 28 is used to calculate θ (the angular position with respect to rotation about the y-axis), and Equation 29 is used to calculate ψ (the angular position with respect to rotation about the z-axis). Since there are numerous solutions, the Euler angles are restricted to the solution space [-π,π].

$$\phi = \tan^{-1}\left(-\frac{D_{32}}{A_{33}}\right) \quad (27)$$

$$\theta = \sin^{-1}(D_{31}) \quad (28)$$

$$\psi = \tan^{-1}\left(-\frac{D_{21}}{D_{11}}\right) \quad (29)$$

By using the initial quaternion, the initial Euler angles are calculated. They are reported in Equation 30, noting that the angles all being zero degrees corresponds to the identity quaternion as the initial quaternion.

$$\begin{bmatrix} \phi_0 \\ \theta_0 \\ \psi_0 \end{bmatrix} = \begin{bmatrix} 0° \\ 0° \\ 0° \end{bmatrix} \quad (30)$$

The angles are then integrated throughout the simulation by using the angular velocity in Equation 31, where ω_x is the angular rate about the x-axis, ω_y is the angular rate about the y-axis, and ω_z is the angular rate about the z-axis.

$$\begin{bmatrix}\phi\\ \theta\\ \psi\end{bmatrix}_{new} = \begin{bmatrix}\phi\\ \theta\\ \psi\end{bmatrix} + \boldsymbol{\omega} * \Delta t = \begin{bmatrix}\phi\\ \theta\\ \psi\end{bmatrix} + \begin{bmatrix}\omega_x\\ \omega_y\\ \omega_z\end{bmatrix} * \Delta t \tag{31}$$

The simulation terminates when the angular rates simultaneously fall within a threshold of 0.01 degrees per second with respect to the desired angular rate of zero degrees per second about each axis. The proportional gains, K_p, and the derivative gains, K_d, are set in Equation 32 and Equation 33, respectively. The gains are optimized across several trial simulations so that the angular velocity is critically damped and driven towards the desired final conditions. The gains are organized in array form and each element corresponds to a different axis, denoted by subscript.

$$K_p = \begin{bmatrix}k_{p,x}\\ k_{p,y}\\ k_{p,z}\end{bmatrix} = \begin{bmatrix}100\\ 100\\ 100\end{bmatrix} \tag{32}$$

$$K_d = \begin{bmatrix}k_{d,x}\\ k_{d,y}\\ k_{d,z}\end{bmatrix} = \begin{bmatrix}1\\ 1\\ 1\end{bmatrix} \tag{33}$$

The angular rates are plotted as a function of time in Figure 20. The angular rate about the x-axis is labeled “w1,” the angular rate about the y-axis is labeled “w2,” and the angular rate about the z-axis is labeled “w3” in the legend. It takes about 9.97 seconds for the thruster pairs to reduce the angular rate about each axis to within tolerance. Figure 21 provides an enhanced view of the angular rates as they approach zero. The angular rate increase seen after completion of thruster firing for the x-axis is a result of angular momentum transfer from the rotation of the other axes, which are accounted for in Equation 17. Euler angles representing the orientation of

the spacecraft are plotted with respect to time in Figure 22 to see the change in orientation of the host spacecraft as it slows to a stop. The variable names “phi,” “theta,” and “psi” correspond to the angles of rotation about the x-axis, the y-axis, and the z-axis, respectively. The final quaternion, Euler angles, and angular velocity are reported in Equation 34.

$$\boldsymbol{q}_f = \begin{bmatrix} 0.1460 \\ 0.4400 \\ 0.2256 \\ 0.8573 \end{bmatrix}, \qquad \begin{bmatrix} \phi_f \\ \theta_f \\ \psi_f \end{bmatrix} = \begin{bmatrix} 22.32° \\ 53.52° \\ 21.71° \end{bmatrix}, \qquad \boldsymbol{\omega}_f = \begin{bmatrix} 0.0019°/s \\ 0.0013°/s \\ -0.0003°/s \end{bmatrix} \tag{34}$$

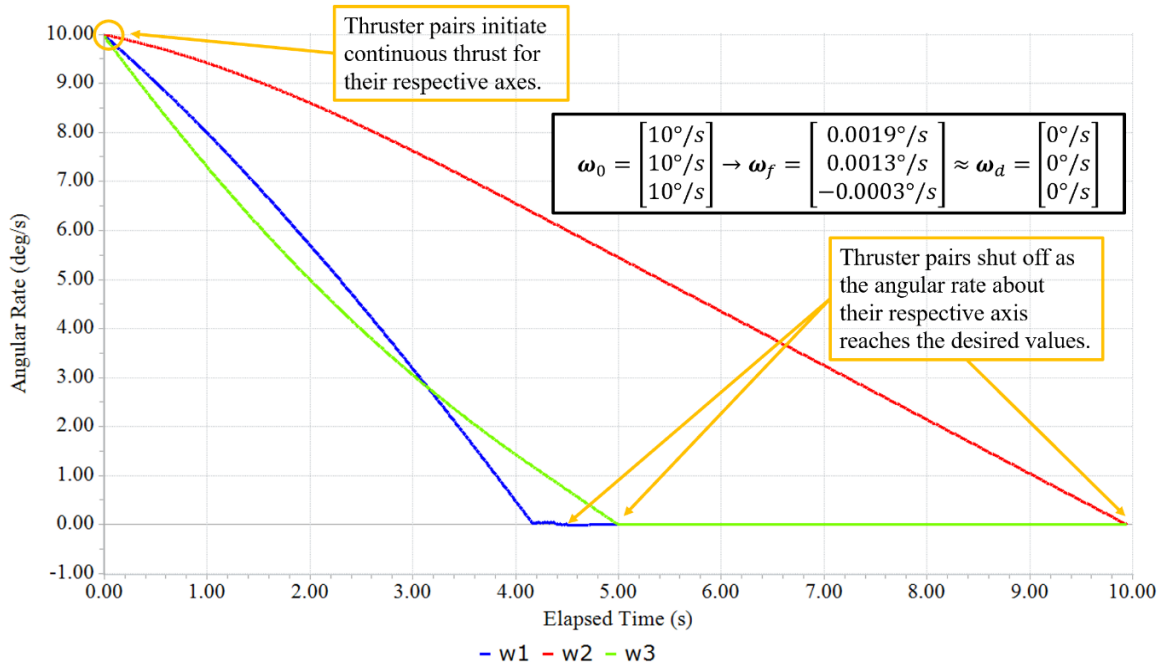


Figure 20. Angular velocity components are plotted as a function of time for the first validation case of the detumble maneuver.

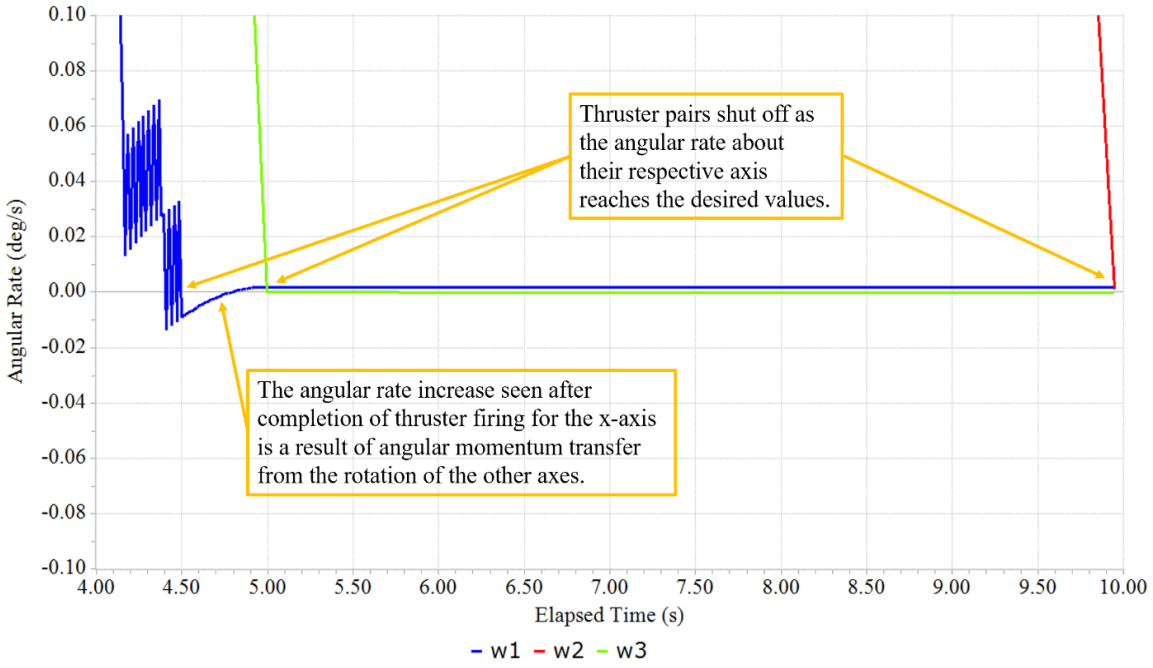


Figure 21. An enhanced view of the thruster control applied close to the desired angular velocity values is provided for the first validation case of the detumble maneuver.

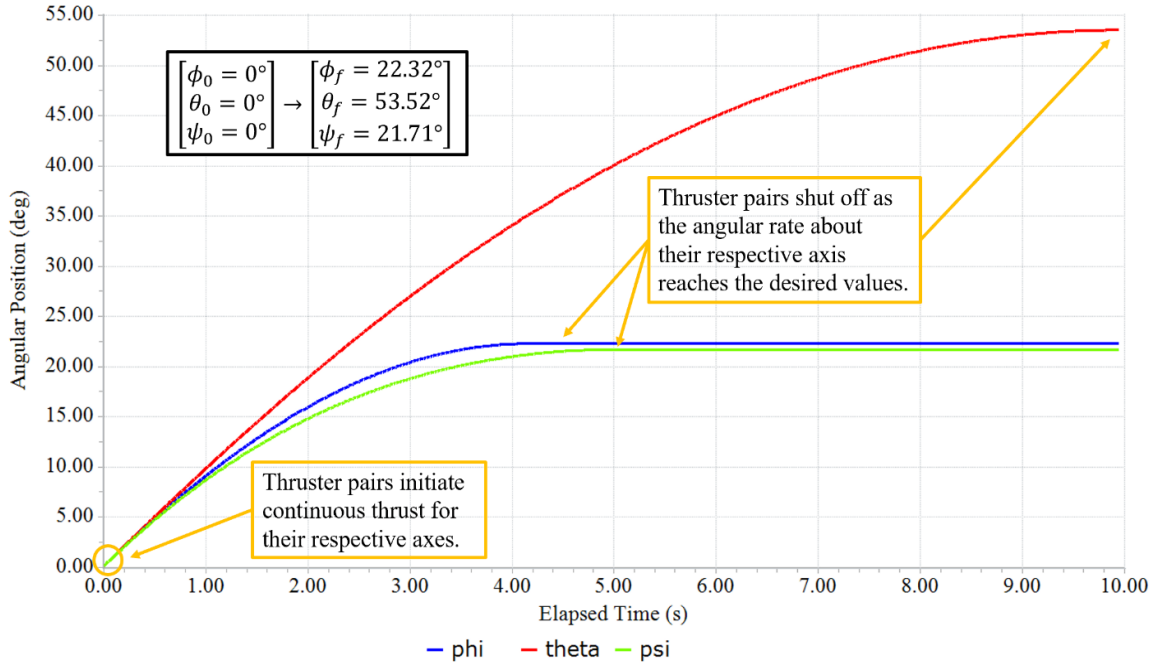


Figure 22. Euler angles are plotted with respect to time to visualize the change of the host spacecraft's orientation during the first validation case of the detumble maneuver.

Thruster signals are also plotted as a function of time for each axis of rotation in Figure 23 to aid in validation of the controller and dynamics model. The signals were used in the software development of the maneuver to troubleshoot and verify that outputs are meeting conceptual expectations associated with the rotational dynamics. They are presented for this maneuver case to provide visual understanding of the simulation. Each axis of rotation has a thruster signal value at each time step. A signal of "1" indicates that a thruster pair is applying force such that a positive torque about an axis occurs. A signal of "-1" indicates that a thruster pair is applying force such that a negative torque about an axis occurs. A signal of "0" indicates that no thrust is applied.

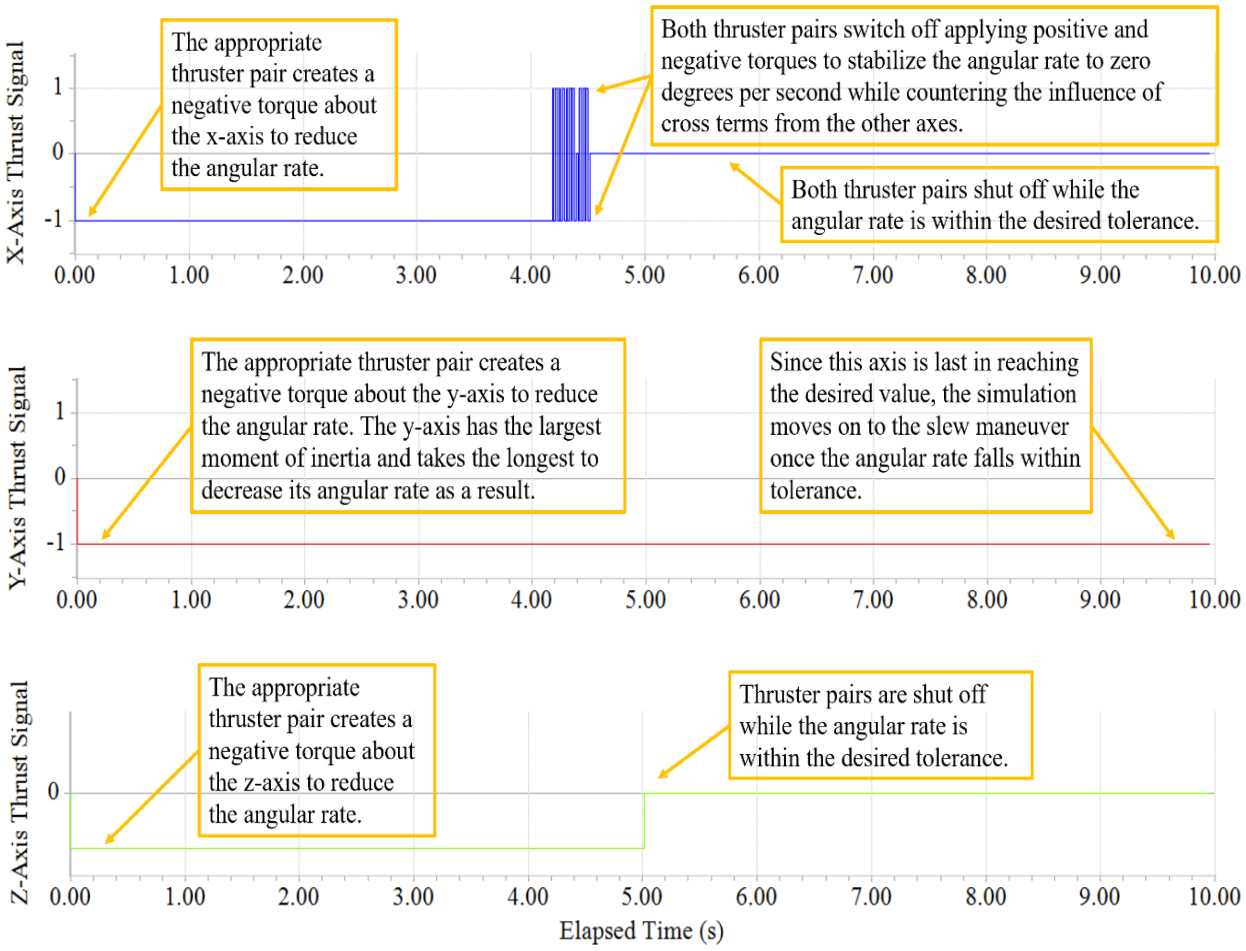


Figure 23. The thrust signals for each of the axes of rotation are depicted as a function of time for the duration of the first validation case of the detumble maneuver.

3.3.4 Detumble Maneuver: Validation Case 2

The second validation case of the detumble maneuver has an initial quaternion orientation with elements set to arbitrary non-zero values. A uniform distribution is applied in the FreeFlyer script to generate initial quaternion values while complying with Equation 35, a property of quaternions. Quaternion values for the simulation are generated with the uniform distribution function U, where the first argument is the mean of the uniform distribution and the second argument is the width of the distribution equivalent to three standard deviations from the mean. This is detailed in Equation 36, yielding the actual values on the right-hand side. The initial Euler angles are calculated using the same process as in the first validation case and are shown in Equation 37.

$$q_1^2 + q_2^2 + q_3^2 + q_4^2 = 1 \tag{35}$$

$$\boldsymbol{q}_0 = \begin{bmatrix} q_1 \in U(0.55, 0.05) \\ q_2 \in U(0.35, 0.05) \\ q_3 \in U(0.15, 0.05) \\ q_4 = \sqrt{1 - q_1^2 + q_2^2 + q_3^2} \end{bmatrix} = \begin{bmatrix} 0.5597 \\ 0.3484 \\ 0.1653 \\ 0.7335 \end{bmatrix} \tag{36}$$

$$\begin{bmatrix} \phi_0 \\ \theta_0 \\ \psi_0 \end{bmatrix} = \begin{bmatrix} 79.51° \\ 44.12° \\ -11.86° \end{bmatrix} \tag{37}$$

The initial angular velocity is the same as that used for the first validation case of the detumble maneuver, where each angular rate is ten degrees per second for each axis of rotation, as seen in Equation 7. The desired angular velocity is still the same as in Equation 8, set at zero degrees per second about each axis of rotation. A tolerance of 0.01 degrees per second is still applied to each angular rate of the desired angular velocity as a trigger for termination of the simulation. A controller featuring the same control torque determination algorithm, proportional gains, and derivative gains as the first validation case of the detumble maneuver is used for the second validation case.

The angular rates are plotted for the second validation case of the detumble maneuver as a function of time in Figure 24. It takes about 9.97 seconds for the thruster pairs to reduce the angular rate about each axis to within tolerance. Figure 25 provides an enhanced view of the angular rates as they approach zero. Once again, the angular rate increase seen after completion of thruster firing for the x-axis is a result of angular momentum transfer from the rotation of the other axes, which are accounted for in Equation 17. Euler angles representing the orientation of the spacecraft are plotted with respect to time in Figure 26. The final state of the second validation case of the detumble maneuver is expressed by the final quaternion, Euler angles, and angular velocity in Equation 34. The second validation case of the detumble maneuver has the same initial angular velocity and desired angular velocity as the first validation case, so the duration of the simulation and integration of the angular rates manifests identically in the results regardless of the difference in initial orientation between the cases.

$$\boldsymbol{q}_f = \begin{bmatrix} 0.5928 \\ 0.5193 \\ 0.5026 \\ 0.3566 \end{bmatrix}, \qquad \begin{bmatrix} \phi_f \\ \theta_f \\ \psi_f \end{bmatrix} = \begin{bmatrix} 101.83° \\ 97.64° \\ 9.85° \end{bmatrix}, \qquad \boldsymbol{\omega}_f = \begin{bmatrix} 0.0019°/s \\ 0.0013°/s \\ -0.0003°/s \end{bmatrix} \tag{38}$$

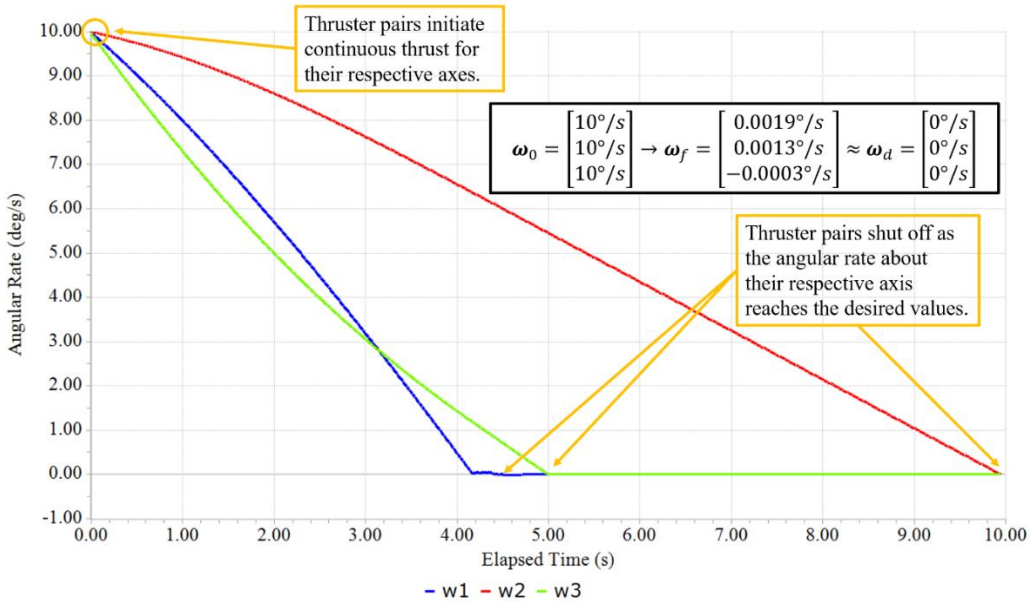


Figure 24. Angular velocity components are plotted as a function of time for the second validation case of the detumble maneuver.

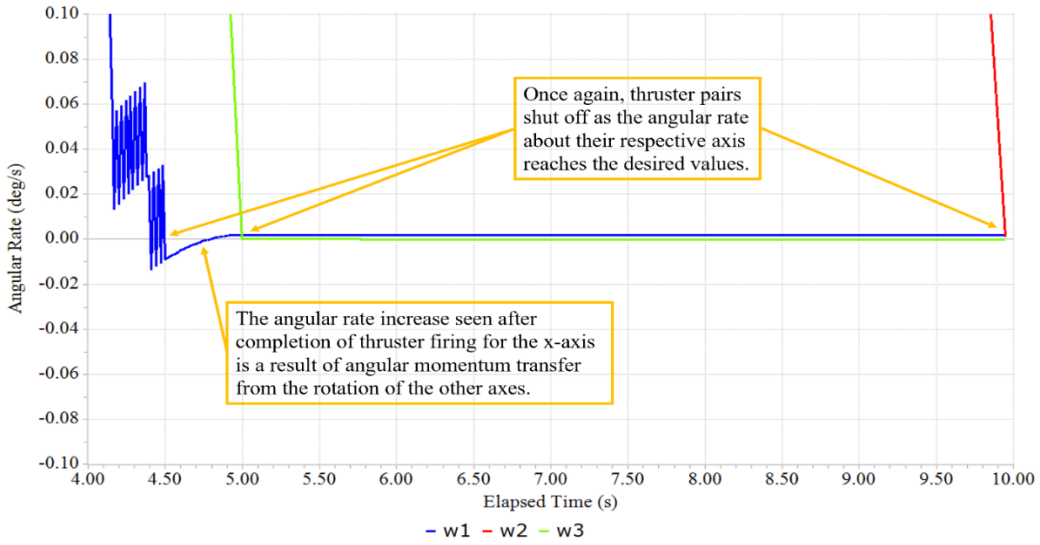


Figure 25. An enhanced view of the thruster control applied close to the desired angular velocity values is provided.

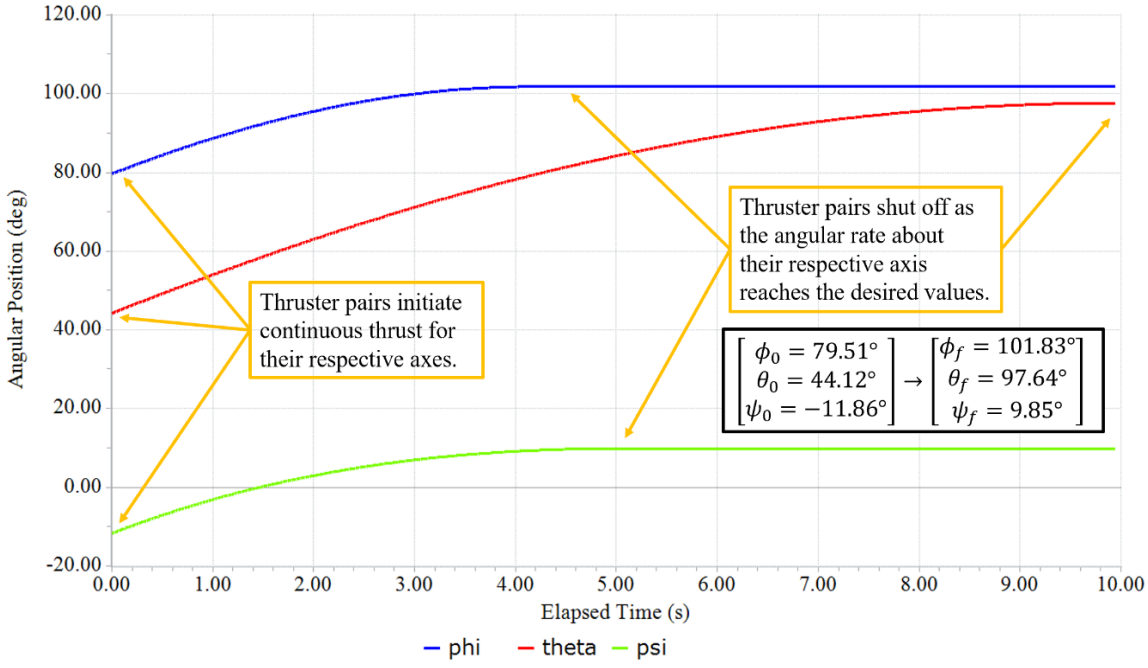


Figure 26. Euler angles are plotted with respect to time to visualize the change of the host spacecraft's orientation during the second validation case of the detumble maneuver.

3.3.5 Slew Maneuver: Validation Case 1

Validation of NEIGHBOR's slew maneuver is also subdivided into different cases for analysis. For the first validation case, the initial orientation of the host spacecraft is set to the orientation of the host spacecraft at the conclusion of the first validation case of the detumble maneuver. This also means that the initial Euler angles are the same as the final Euler angles of the first validation case of the detumble maneuver. Likewise, the initial angular velocity for the first validation case of the slew maneuver is the same as the final angular velocity of the detumble maneuver's first validation case. The initial quaternion, Euler angles, and angular velocity for the first validation case of the slew maneuver are shown in Equation 39. The initial Euler angles are propagated as before.

$$\boldsymbol{q}_0 = \begin{bmatrix} 0.1460 \\ 0.4400 \\ 0.2256 \\ 0.8573 \end{bmatrix}, \qquad \begin{bmatrix} \phi_0 \\ \theta_0 \\ \psi_0 \end{bmatrix} = \begin{bmatrix} 22.32° \\ 53.52° \\ 21.71° \end{bmatrix}, \qquad \boldsymbol{\omega}_0 = \begin{bmatrix} 0.0019°/s \\ 0.0013°/s \\ -0.0003°/s \end{bmatrix} \tag{39}$$

The host spacecraft is rotated until it comes to a stop at its original orientation for the first validation case of the detumble maneuver, which is the identity quaternion. The desired quaternion and angular velocity of the first validation case of the slew maneuver are captured in Equation 40.

$$\boldsymbol{q}_d = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \qquad \boldsymbol{\omega}_d = \begin{bmatrix} 0°/s \\ 0°/s \\ 0°/s \end{bmatrix} \tag{40}$$

NEIGHBOR's slew maneuvers also feature a PD controller to drive the host spacecraft from the initial orientation to the desired burn orientation. The expressions used in the error terms of the controller come from attitude regulation theory, where a spacecraft is slewed to a

desired orientation from an initial orientation such that it also comes to a rest at the end of the maneuver [41]. The proportional error term is composed of the product of the sign of the fourth term of the current quaternion and the vector containing the first three elements of the error quaternion, $\boldsymbol{\delta q}$, calculated in Equation 41. The operator "(x)" represents the circle cross product of two quaternions and the operator "$\cdot$" symbolizes the dot product of two arrays or vectors. The circle cross product of two arbitrary quaternions $\boldsymbol{p}$ and $\boldsymbol{q}$ is defined in Equation 42. The definition of the inverse of a quaternion is provided in Equation 43, which is used to calculate the inverse of the desired quaternion, $\boldsymbol{q}_d^{-1}$. Equation 44 defines the conjugate of a quaternion, used in the definition of the inverse of a quaternion.

$$\boldsymbol{\delta q} = \boldsymbol{q}\ (x)\ \boldsymbol{q}_d^{-1} = \begin{bmatrix} (\boldsymbol{q}_d^{-1})_4 \boldsymbol{q}_{1:3} + q_4 (\boldsymbol{q}_d^{-1})_{1:3} - \boldsymbol{q}_{1:3} \times (\boldsymbol{q}_d^{-1})_{1:3} \\ q_4 (\boldsymbol{q}_d^{-1})_4 - \boldsymbol{q}_{1:3} \cdot (\boldsymbol{q}_d^{-1})_{1:3} \end{bmatrix} \tag{41}$$

$$\boldsymbol{p}\ (x)\ \boldsymbol{q} = \begin{bmatrix} p_4 \boldsymbol{q}_{1:3} + q_4 \boldsymbol{p}_{1:3} - \boldsymbol{p}_{1:3} \times \boldsymbol{q}_{1:3} \\ p_4 q_4 - \boldsymbol{p}_{1:3} \cdot \boldsymbol{q}_{1:3} \end{bmatrix} \tag{42}$$

$$\boldsymbol{q}^{-1} = \frac{\boldsymbol{q}^*}{\|\boldsymbol{q}\|^2} \tag{43}$$

$$\boldsymbol{q}^* = \begin{bmatrix} -\boldsymbol{q}_{1:3} \\ q_4 \end{bmatrix} \tag{44}$$

The derivative error term is composed of the difference between the desired angular velocity and the current angular velocity. The error terms are weighted by the associated gains K_p and K_d, set in Equation 45. The gains are optimized across several trial simulations so that the quaternion state parameters are critically damped and driven towards the desired burn orientation. The output controller torque is given in Equation 46.

$$K_p = \begin{bmatrix} 10 \\ 10 \\ 10 \end{bmatrix}, \qquad K_d = \begin{bmatrix} 17.5 \\ 17.5 \\ 17.5 \end{bmatrix} \tag{45}$$

$$\boldsymbol{L} = -K_p sign(\delta q_4) \boldsymbol{\delta q}_{1:3} + K_d (\boldsymbol{\omega}_d - \boldsymbol{\omega}) \tag{46}$$

As with the detumble maneuver controller, the output torques are saturated at the nominal thrust delivered by the RCS thrusters or reduced to zero if the magnitude of the commanded torque is less than the nominal torque value. The threshold for the quaternion values is set to 0.01 and the tolerance for the angular velocity values is set to 0.01 degrees per second. When the current quaternion and angular velocity fall within the associated tolerances with respect to the desired quaternion and angular velocity, the simulation terminates.

The elements of the quaternion state are plotted with respect to time in Figure 27 to show that the controller drives the initial condition toward the desired condition. The first, second, third, and fourth element of the quaternion state are denoted by "q1," "q2," "q3," and "q4," respectively. Euler angles are provided in Figure 28 to accompany the quaternion state for understanding of how the host spacecraft's attitude changes with time in the simulation. Angular rates are shown in Figure 29 to understand how the attitude state is being changed in time with the commanded output torques. The maneuver lasts 24.84 seconds. The final quaternion, Euler angles, and angular velocity are reported in Equation 47.

$$\boldsymbol{q}_f = \begin{bmatrix} 0.00017 \\ 0.00041 \\ 0.00043 \\ 1.00060 \end{bmatrix}, \qquad \begin{bmatrix} \phi_f \\ \theta_f \\ \psi_f \end{bmatrix} = \begin{bmatrix} 8.63° \\ 2.19° \\ -9.74° \end{bmatrix}, \qquad \boldsymbol{\omega}_f = \begin{bmatrix} -0.0053°/s \\ -0.0054°/s \\ -0.0021°/s \end{bmatrix} \tag{47}$$

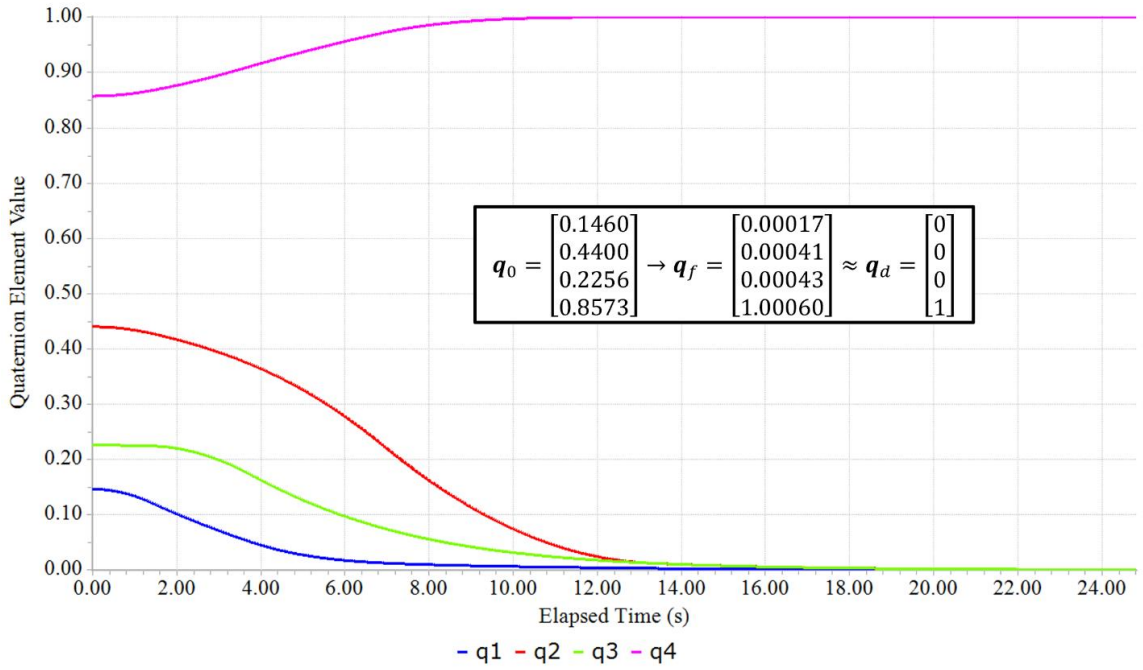


Figure 27. Quaternion elements are plotted for the duration of the first validation case of the slew maneuver to show the controller drives the system from the initial quaternion to the desired quaternion.

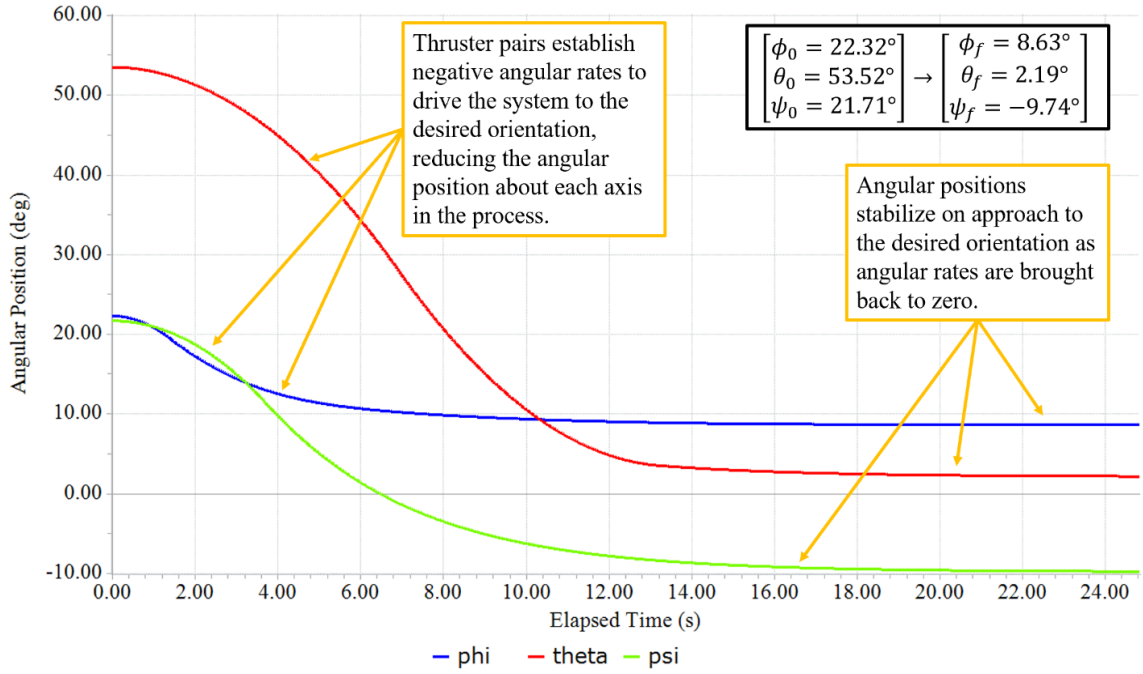


Figure 28. Euler angles are plotted with respect to time to accompany the quaternion results of the first validation case of the slew maneuver.

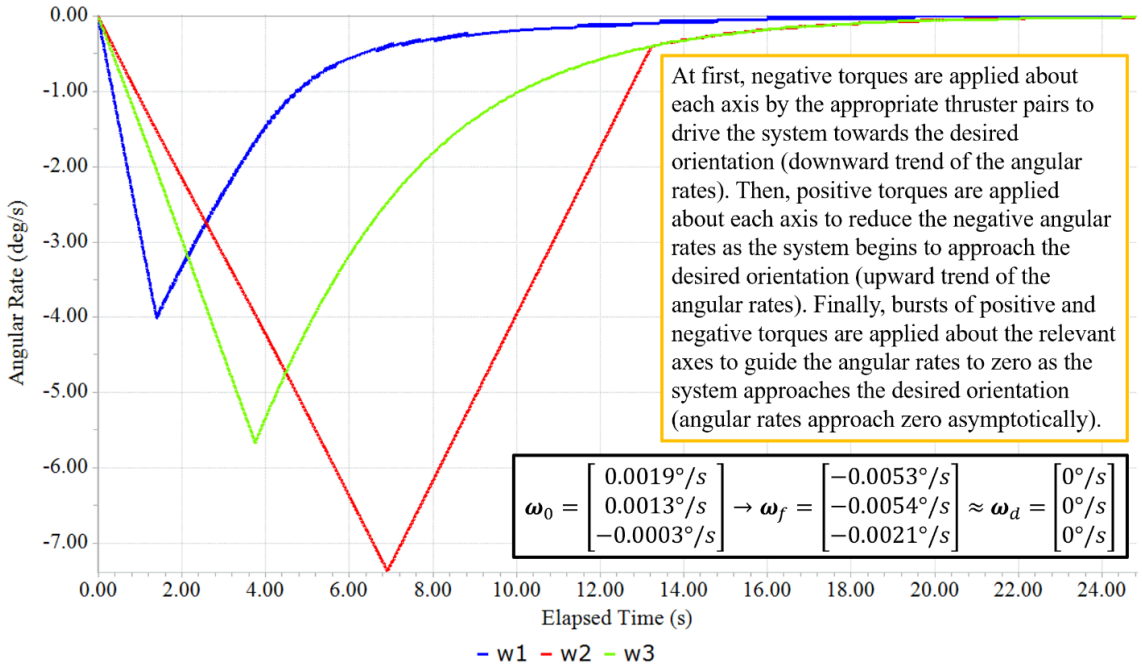


Figure 29. Angular velocity components are plotted across time for the first validation case of the slew maneuver.

3.3.6 Slew Maneuver: Validation Case 2

The second validation case of the slew maneuver uses the same controller as the first validation case, with a different initial condition and same applied gains. The initial quaternion, Euler angles, and angular velocity of the first validation case of the slew maneuver are the same as the final quaternion, Euler angles, and angular velocity of the second validation case of the detumble maneuver. The initial quaternion, Euler angles, and angular velocity of the second validation case of the slew maneuver are expressed in Equation 48. The simulation for this case still targets the identity quaternion as the desired quaternion and an angular velocity of zero degrees per second about each axis, described by Equation 40.

$$\boldsymbol{q}_0 = \begin{bmatrix} 0.5928 \\ 0.5193 \\ 0.5026 \\ 0.3566 \end{bmatrix}, \qquad \begin{bmatrix} \phi_0 \\ \theta_0 \\ \psi_0 \end{bmatrix} = \begin{bmatrix} 101.83° \\ 97.64° \\ 9.85° \end{bmatrix}, \qquad \boldsymbol{\omega}_0 = \begin{bmatrix} 0.0019°/s \\ 0.0013°/s \\ -0.0003°/s \end{bmatrix} \tag{48}$$

As before, the error terms are weighted by the associated gains K_p and K_d, set in Equation 49. The gains are optimized across several trial simulations so that the quaternion state parameters are damped and driven towards the desired burn orientation.

$$K_p = \begin{bmatrix} 10 \\ 10 \\ 10 \end{bmatrix}, \qquad K_d = \begin{bmatrix} 22.5 \\ 22.5 \\ 22.5 \end{bmatrix} \tag{49}$$

The tolerance for the desired quaternion values remains at 0.01 and the tolerance for the angular velocity values also remains at 0.01 degrees per second. The maneuver lasts about 38.25 seconds. The quaternion state, associated Euler angles, and angular velocity are plotted for the duration of the simulation in Figure 30, Figure 31, and Figure 32, respectively. The final quaternion, Euler angles, and angular velocity are reported in Equation 50.

$$\boldsymbol{q}_f = \begin{bmatrix} 0.00027 \\ 0.00039 \\ -0.00012 \\ 1.00125 \end{bmatrix}, \qquad \begin{bmatrix} \phi_f \\ \theta_f \\ \psi_f \end{bmatrix} = \begin{bmatrix} 20.72° \\ 33.33° \\ -79.84° \end{bmatrix}, \qquad \boldsymbol{\omega}_f = \begin{bmatrix} -0.0028°/s \\ -0.0038°/s \\ -0.0100°/s \end{bmatrix} \tag{50}$$

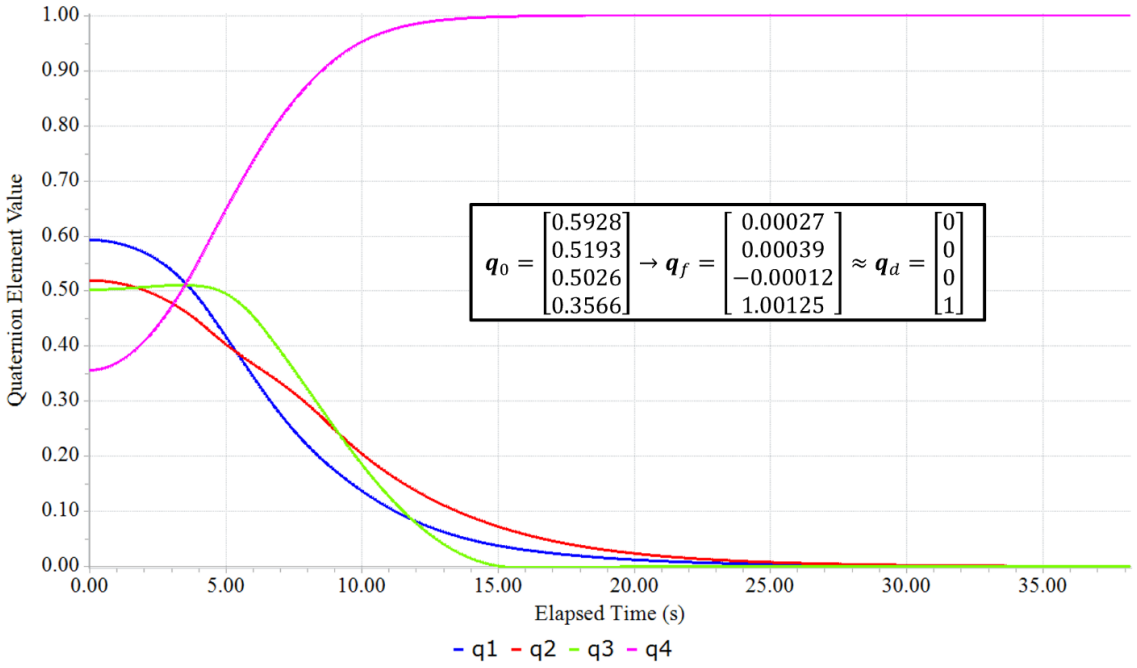


Figure 30. Quaternion elements are plotted for the duration of the second validation case of the slew maneuver to show the controller drives the system from the initial quaternion to the desired quaternion.

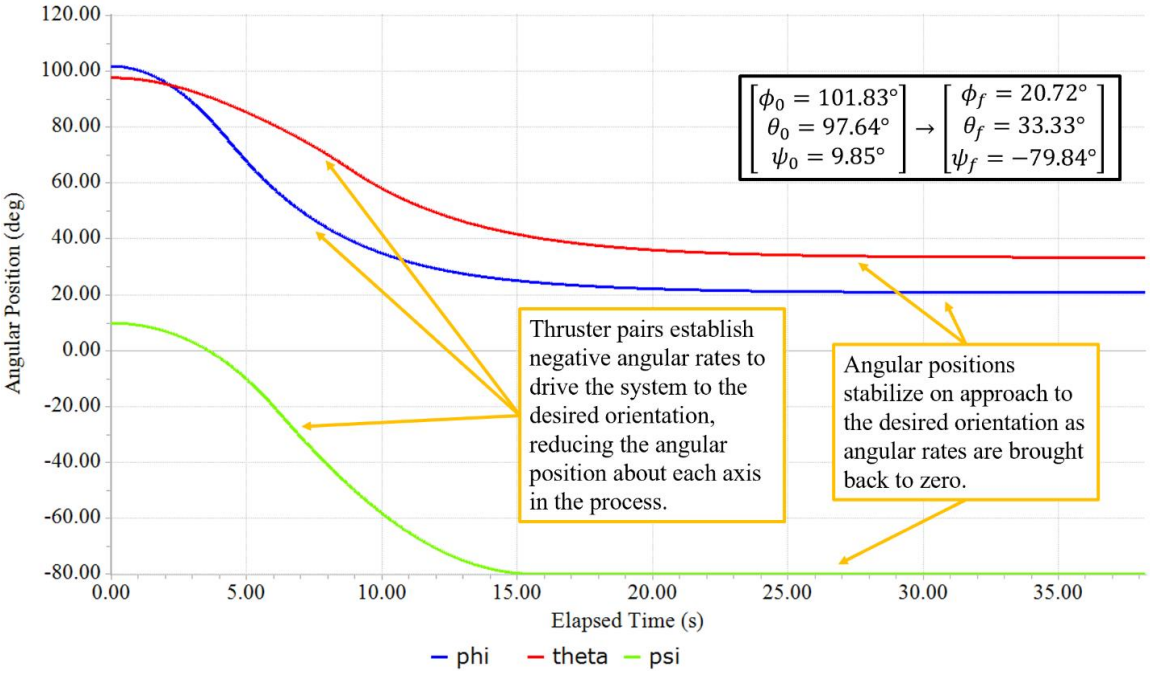


Figure 31. The Euler angles associated with the quaternion state are plotted for the duration of the second validation case of the slew maneuver.

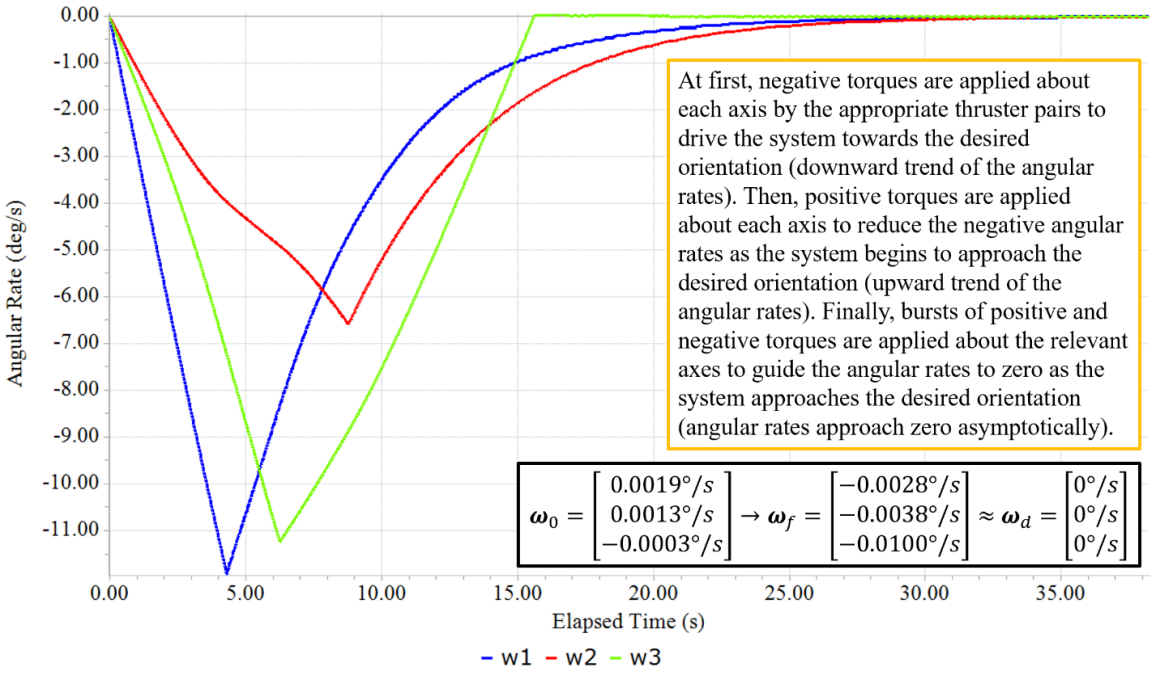


Figure 32. Angular velocity components are plotted for the duration of the second validation case of the slew maneuver.

3.3.7 Slew Maneuver: Validation Case 3

Validation case three of the slew maneuver sizes the worst-case slew maneuver needed to re-orient the host spacecraft to the desired burn orientation. To size the worst-case slew maneuver, the RCS thrusters successively rotate the host spacecraft 180 degrees about each body axis and the required propellant mass is summed. First, the x-axis is the axis of rotation, then the y-axis, and finally the z-axis. This establishes an upper bound on the propellant mass required for an arbitrary slew maneuver by considering the maximum necessary rotation about any given axis and combining the resources needed to execute them.

The first validation case of the detumble maneuver is selected to precede the third validation case of the slew maneuver. Therefore, the initial quaternion, Euler angles, and angular velocity of the third validation case of the slew maneuver are the same as the final quaternion, Euler angles, and angular velocity of the first validation case of the detumble maneuver,

expressed in Equation 51. The simulation for this case still targets a desired angular velocity of zero degrees per second about each axis.

$$\boldsymbol{q}_0 = \begin{bmatrix} 0.1460 \\ 0.4400 \\ 0.2256 \\ 0.8573 \end{bmatrix}, \qquad \begin{bmatrix} \phi_0 \\ \theta_0 \\ \psi_0 \end{bmatrix} = \begin{bmatrix} 22.32° \\ 53.52° \\ 21.71° \end{bmatrix}, \qquad \boldsymbol{\omega}_0 = \begin{bmatrix} 0.0019°/s \\ 0.0013°/s \\ -0.0003°/s \end{bmatrix} \tag{51}$$

To calculate the desired final quaternion for each rotation, Equation 52 is first utilized to determine the quaternion representing the transformation from the initial quaternion to the final quaternion. In the expression, $\boldsymbol{e}$ is the axis of rotation vector and ϑ is the angle of rotation about the axis.

$$\boldsymbol{q} = \begin{bmatrix} \boldsymbol{e} \sin\left(\frac{\vartheta}{2}\right) \\ \cos\left(\frac{\vartheta}{2}\right) \end{bmatrix} \tag{52}$$

Equation 53 shows the parameters specifying the first rotation. The axis of rotation is the x-axis, represented by $\boldsymbol{e}_x$, and the angle of rotation is 180 degrees, represented by ϑ_x.

$$\boldsymbol{e}_x = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \qquad \vartheta_x = 180° \tag{53}$$

The quaternion representing the rotation about the x-axis by 180 degrees is calculated in Equation 54.

$$\boldsymbol{q}_x = \begin{bmatrix} \boldsymbol{e}_x \sin\left(\frac{\vartheta_x}{2}\right) \\ \cos\left(\frac{\vartheta_x}{2}\right) \end{bmatrix} = \begin{bmatrix} \boldsymbol{e}_x \sin\left(\frac{180°}{2}\right) \\ \cos\left(\frac{180°}{2}\right) \end{bmatrix} = \begin{bmatrix} \boldsymbol{e}_x \sin(90°) \\ \cos(90°) \end{bmatrix} = \begin{bmatrix} \boldsymbol{e}_x \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{54}$$

The quaternions representing the rotations about the y-axis and z-axis by 180 degrees are similarly determined from the inputs displayed in Equation 55 and reported in Equation 56.

$$\boldsymbol{e}_y = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \qquad \vartheta_y = 180°, \qquad \boldsymbol{e}_z = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \qquad \vartheta_z = 180° \tag{55}$$

$$\boldsymbol{q}_y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \qquad \boldsymbol{q}_z = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \tag{56}$$

The desired final quaternions for each rotation can now be computed with the circle cross product operation that unites two sequential rotations to represent a single rotation. In Equation 57, the rotation of 180 degrees about the x-axis combined with the initial quaternion, $\boldsymbol{q}_0$, produces the quaternion $\boldsymbol{q}_{d,x}$, the desired quaternion after the rotation about the x-axis. From there, the desired quaternion after the rotation about the y-axis, $\boldsymbol{q}_{d,y}$, is produced in Equation 58 by applying the quaternion representing the rotation by 180 degrees about the y-axis, $\boldsymbol{q}_y$, to the desired quaternion following the x-axis rotation, $\boldsymbol{q}_{d,x}$. The desired quaternion following the rotation about the z-axis by 180 degrees is then calculated by applying $\boldsymbol{q}_z$ to $\boldsymbol{q}_{d,y}$ in Equation 59. The desired final quaternions are listed in Equation 60.

$$\boldsymbol{q}_{d,x} = \boldsymbol{q}_x \,(x)\, \boldsymbol{q}_0 = \begin{bmatrix} (\boldsymbol{q}_x)_4(\boldsymbol{q}_0)_{1:3} + (\boldsymbol{q}_0)_4(\boldsymbol{q}_x)_{1:3} - (\boldsymbol{q}_x)_{1:3} \times (\boldsymbol{q}_0)_{1:3} \\ (\boldsymbol{q}_x)_4(\boldsymbol{q}_0)_4 - (\boldsymbol{q}_x)_{1:3} \cdot (\boldsymbol{q}_0)_{1:3} \end{bmatrix} \tag{57}$$

$$\boldsymbol{q}_{d,y} = \boldsymbol{q}_y \,(x)\, \boldsymbol{q}_{d,x} \tag{58}$$

$$\boldsymbol{q}_{d,z} = \boldsymbol{q}_z \,(x)\, \boldsymbol{q}_{d,y} \tag{59}$$

$$\boldsymbol{q}_{d,x} = \begin{bmatrix} 0.8573 \\ 0.2256 \\ -0.4400 \\ -0.1460 \end{bmatrix}, \qquad \boldsymbol{q}_{d,y} = \begin{bmatrix} 0.4400 \\ -0.1460 \\ 0.8573 \\ -0.2256 \end{bmatrix}, \qquad \boldsymbol{q}_{d,z} = \begin{bmatrix} -0.1460 \\ -0.4400 \\ -0.2256 \\ -0.8573 \end{bmatrix} \tag{60}$$

The control law of Equation 46 is implemented to drive the system from the initial conditions to the final conditions following each rotation about an axis. Gains are optimized for each rotation through iterative simulation. The tolerance for the desired quaternion values is still 0.01 and the tolerance for the angular velocity values is still 0.01 degrees per second. The gains used as input for the controller during the x-axis rotation are displayed in Equation 61.

$$K_{p,x} = \begin{bmatrix} 10 \\ 10 \\ 10 \end{bmatrix}, \qquad K_{d,x} = \begin{bmatrix} 20 \\ 20 \\ 20 \end{bmatrix} \tag{61}$$

The first rotation lasts 32.81 seconds. The Euler angles, associated quaternion state, and angular velocity are plotted for the duration of the rotation in Figure 33, Figure 34, and Figure 35, respectively. The final quaternion and angular velocity fall within tolerance and are reported in Equation 62 with the final Euler angles.

$$\boldsymbol{q}_{f,x} = \begin{bmatrix} 0.8585 \\ 0.2267 \\ -0.4405 \\ -0.1451 \end{bmatrix}, \qquad \begin{bmatrix} \phi_{f,x} \\ \theta_{f,x} \\ \psi_{f,x} \end{bmatrix} = \begin{bmatrix} 202.17° \\ 53.53° \\ 21.68° \end{bmatrix}, \qquad \boldsymbol{\omega}_{f,x} = \begin{bmatrix} 0.0019°/s \\ -0.0004°/s \\ -0.0018°/s \end{bmatrix} \tag{62}$$

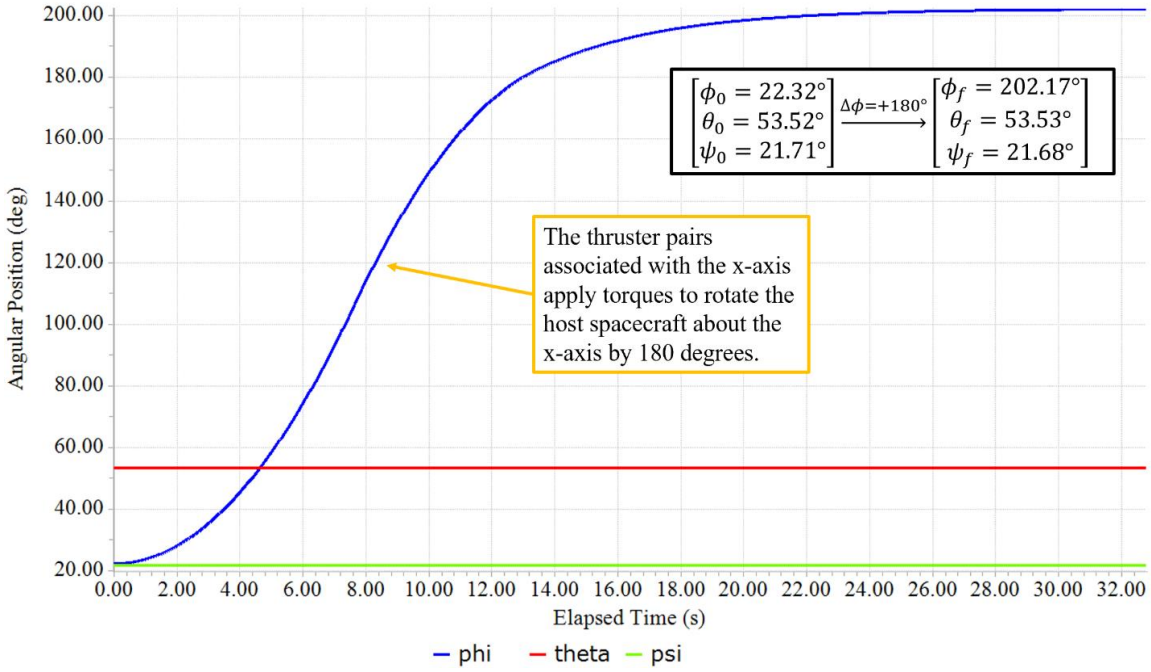


Figure 33. The Euler angles are plotted for the x-axis rotation during the third validation case of the slew maneuver to show a 180-degree rotation about the x-axis.

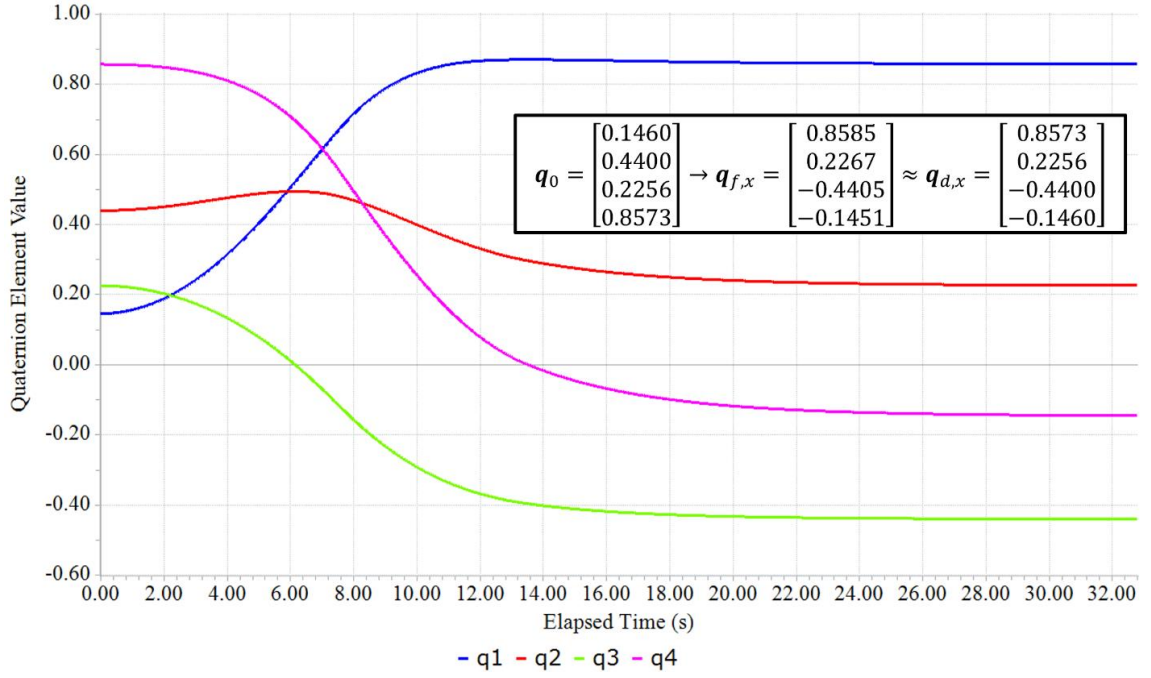


Figure 34. Quaternion elements are plotted for the x-axis rotation during the third validation case of the slew maneuver to show the controller drives the system from the initial quaternion to the desired quaternion.

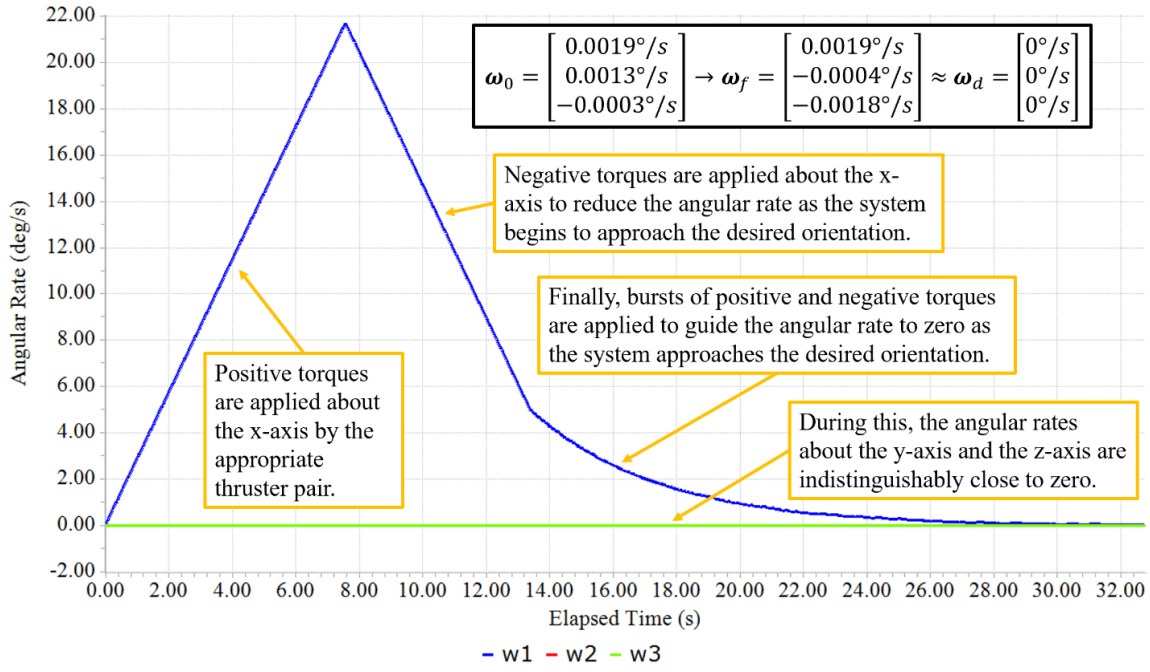


Figure 35. Angular velocity components are plotted for the x-axis rotation during the third validation case of the slew maneuver.

The final conditions of the rotation about the x-axis are the initial conditions for the rotation about the y-axis. The gains used as input for the controller during the y-axis rotation are displayed in Equation 63 and are a result of iterative simulation. The second rotation lasts about 82.5 seconds. The Euler angles, associated quaternion state, and angular velocity are plotted for the duration of the rotation in Figure 36, Figure 37, and Figure 38, respectively. The final quaternion and angular velocity fall within tolerance and are reported in Equation 64 with the final Euler angles.

$$K_{p,y} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \qquad K_{d,y} = \begin{bmatrix} 5 \\ 5 \\ 5 \end{bmatrix} \tag{63}$$

$$\boldsymbol{q}_{f,y} = \begin{bmatrix} -0.4412 \\ 0.1453 \\ -0.8585 \\ 0.2282 \end{bmatrix}, \qquad \begin{bmatrix} \phi_{f,y} \\ \theta_{f,y} \\ \psi_{f,y} \end{bmatrix} = \begin{bmatrix} 202.24° \\ -126.57° \\ 21.85° \end{bmatrix}, \qquad \boldsymbol{\omega}_{f,y} = \begin{bmatrix} 0.0001°/s \\ -0.0004°/s \\ 0.0028°/s \end{bmatrix} \tag{64}$$

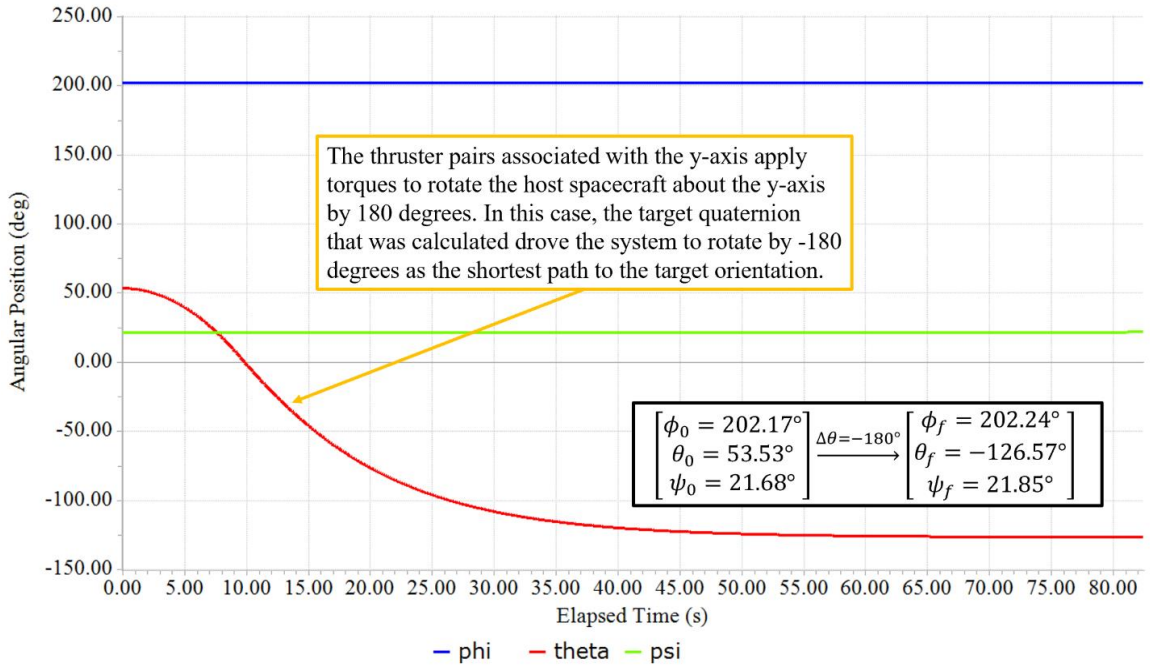


Figure 36. The Euler angles are plotted for the y-axis rotation during the third validation case of the slew maneuver to show a 180-degree rotation about the y-axis.

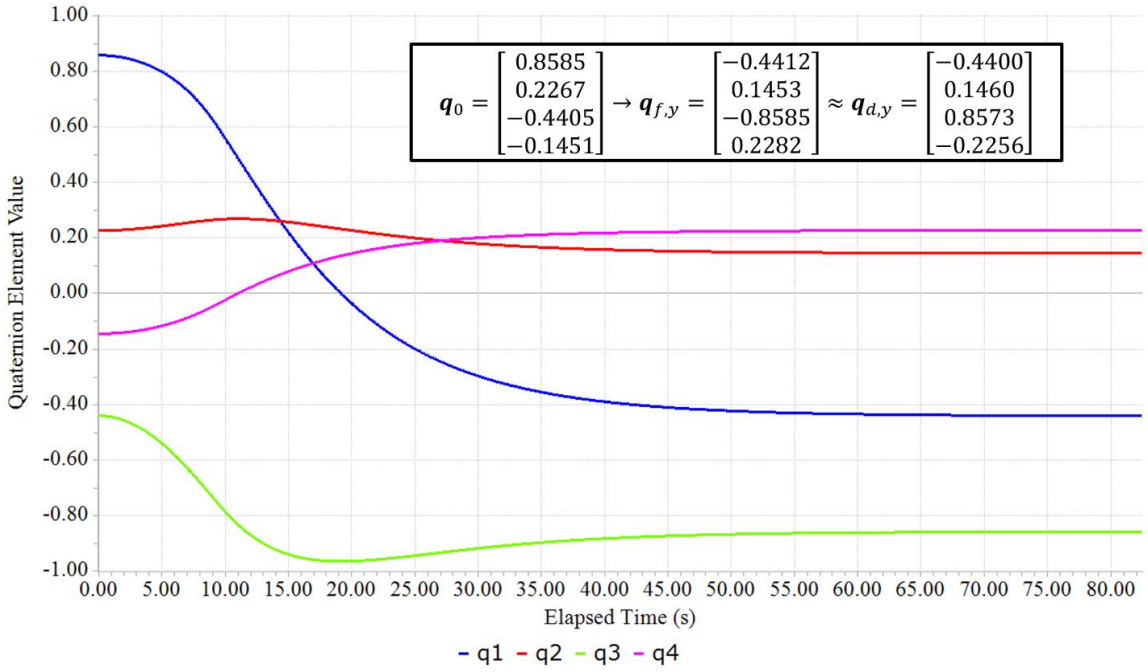


Figure 37. Quaternion elements are plotted for the y-axis rotation during the third validation case of the slew maneuver to show the controller drives the system from the initial quaternion to the desired quaternion.

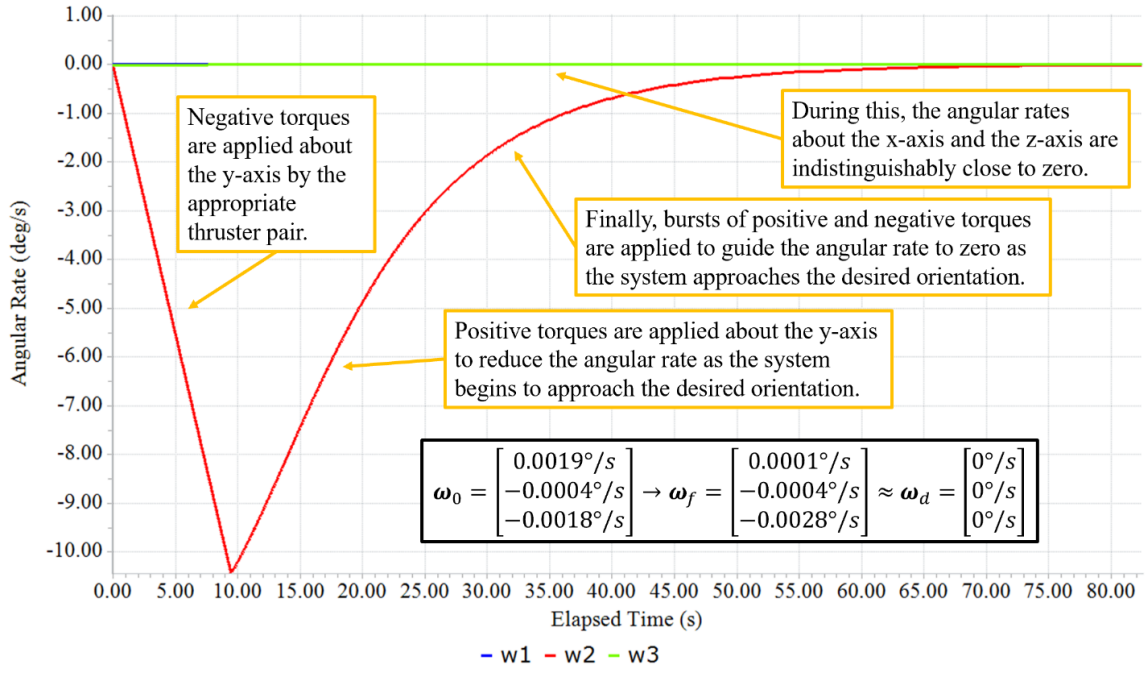


Figure 38. Angular velocity components are plotted for the y-axis rotation during the third validation case of the slew maneuver.

The final conditions of the rotation about the x-axis are the initial conditions for the rotation about the y-axis. The gains used as input for the controller during the y-axis rotation are displayed in Equation 65 and are a result of iterative simulation. The second rotation lasts 50.96 seconds. The Euler angles, associated quaternion state, and angular velocity are plotted for the duration of the rotation in Figure 39, Figure 40, and Figure 41, respectively. The final quaternion and angular velocity fall within tolerance and are reported in Equation 66 with the final Euler angles. In total, the rotations require 166.26 seconds to execute.

$$K_{p,z} = \begin{bmatrix} 10 \\ 10 \\ 10 \end{bmatrix}, \qquad K_{d,z} = \begin{bmatrix} 30 \\ 30 \\ 30 \end{bmatrix} \tag{65}$$

$$\boldsymbol{q}_{f,z} = \begin{bmatrix} 0.1460 \\ 0.4413 \\ 0.2254 \\ 0.8601 \end{bmatrix}, \qquad \begin{bmatrix} \phi_{f,z} \\ \theta_{f,z} \\ \psi_{f,z} \end{bmatrix} = \begin{bmatrix} 202.15° \\ -126.08° \\ 201.56° \end{bmatrix}, \qquad \boldsymbol{\omega}_{f,z} = \begin{bmatrix} -0.0034°/s \\ -0.0083°/s \\ 0.0029°/s \end{bmatrix} \tag{66}$$

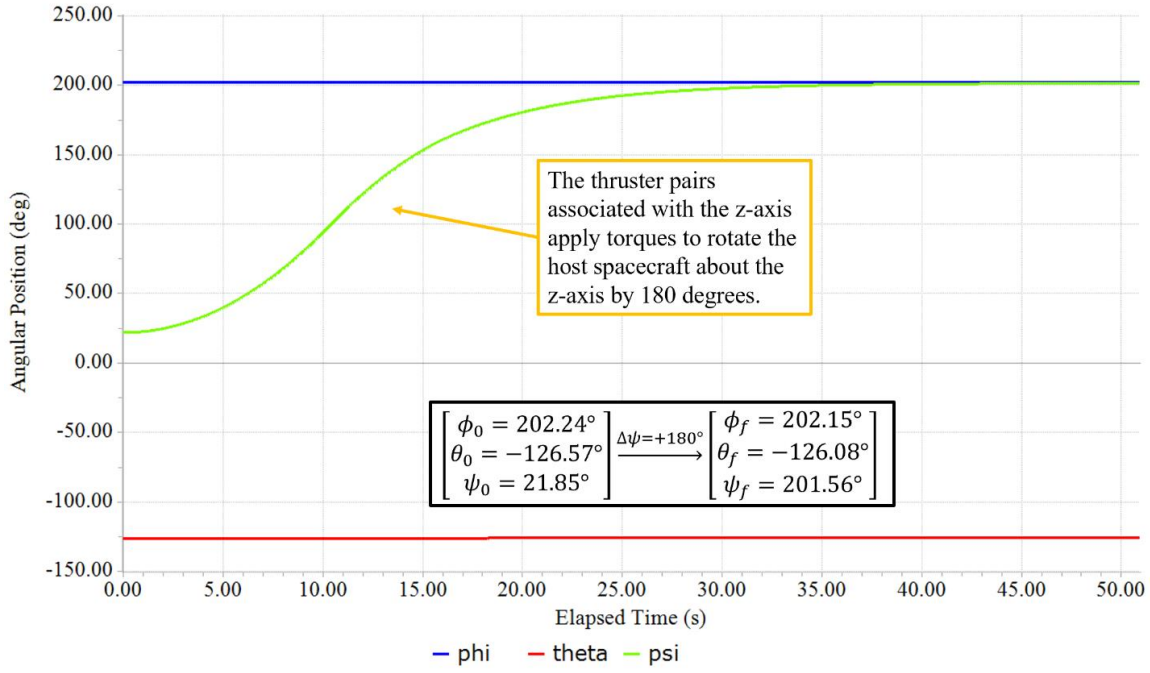


Figure 39. The Euler angles are plotted for the z-axis rotation during the third validation case of the slew maneuver to show a 180-degree rotation about the z-axis.

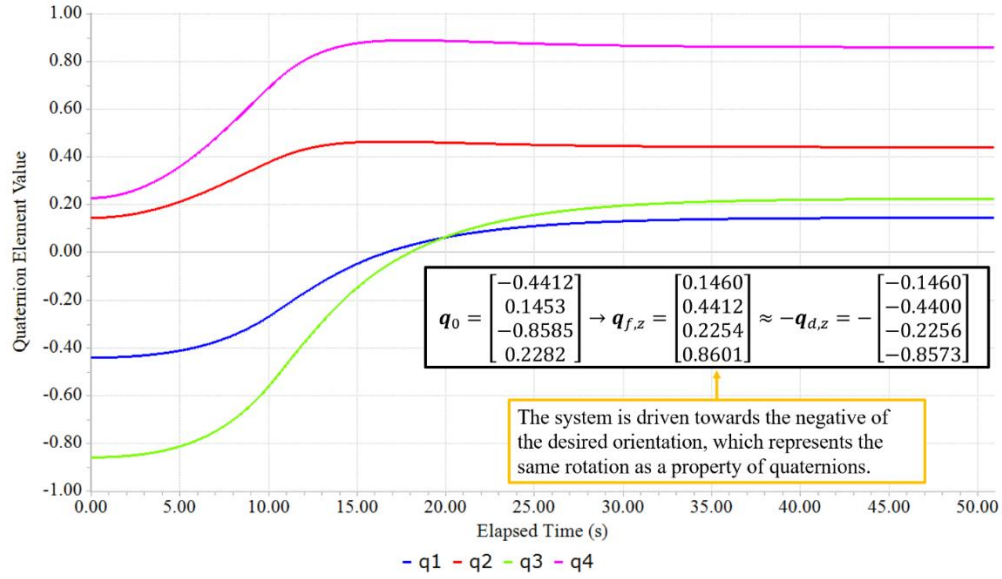


Figure 40. Quaternion elements are plotted for the z-axis rotation during the third validation case of the slew maneuver to show the controller drives the system from the initial quaternion to the desired quaternion.

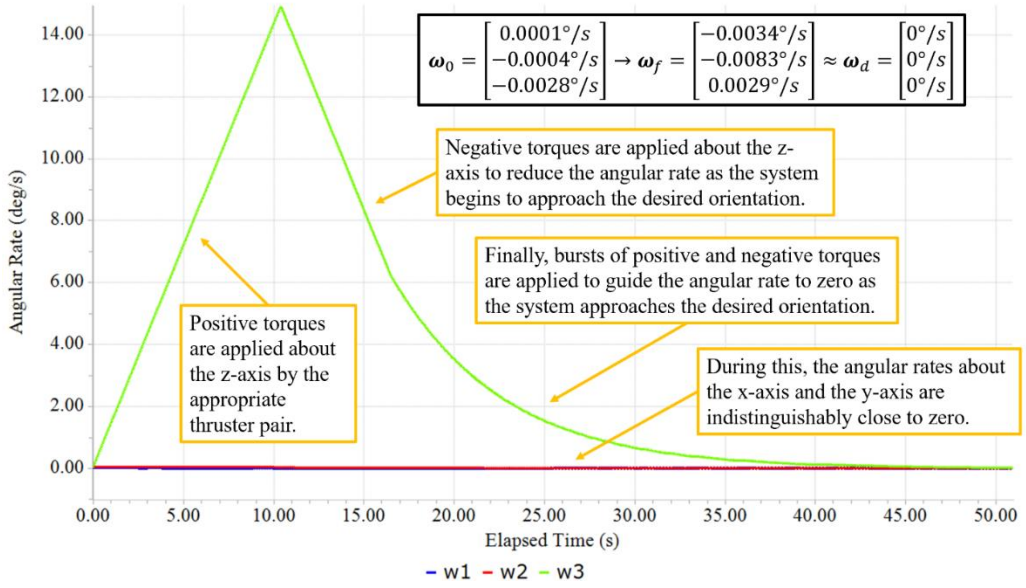


Figure 41. Angular velocity components are plotted for the z-axis rotation during the third validation case of the slew maneuver.

3.3.8 De-orbit Attitude Tracking

NEIGHBOR uses the RCS thrusters to hold the desired burn orientation for the duration of the de-orbit maneuver. The de-orbit control simulation models attitude control of the host spacecraft during the de-orbit maneuver. The desired burn orientation is targeted by calculating a desired angular velocity that allows the host spacecraft to continuously point in the desired direction as time passes. The host spacecraft must be controlled so that it rotates at a rate of one revolution per orbit period to track the proper direction during the de-orbit maneuver. The x-axis must point in the ram direction, i.e. the direction of motion, so that the axial thrusters are able to deliver the retrograde burn. The axial thrusters used for the de-orbit maneuver are depicted in Figure 42.

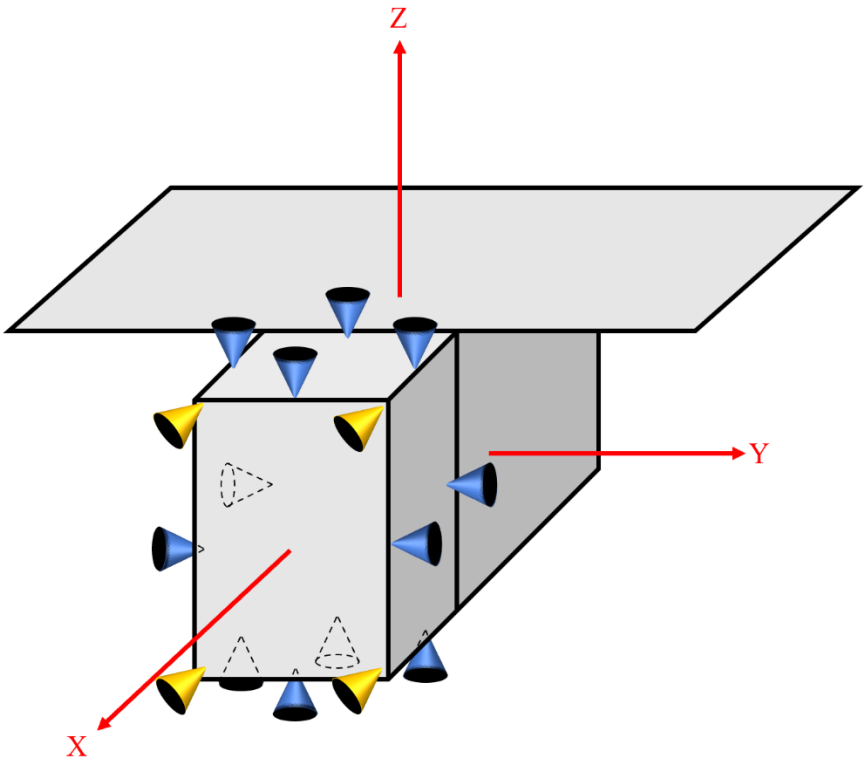


Figure 42. The thrusters fired in the de-orbit maneuver to de-orbit the spacecraft are highlighted in yellow.

The axis of rotation to track this direction is assumed to be the y-axis. The y-axis has the largest moment of inertia and therefore represents the largest propellant requirement for tracking control of the burn direction. To calculate the desired rate of rotation, the period of the spacecraft's orbit is first computed with Equation 67. The semi-major axis, *A*, is the sum of the equatorial radius of the Earth, 6378.1363 km, and the average orbit altitude during the maneuver, which is 400 km for the first de-orbit scenario. The average semi-major axis of the orbit during the maneuver is therefore 6778.1363 km. The orbital period is 92.56 minutes. One revolution is equivalent to 360 degrees, so the desired angular rate of the y-axis, $\omega_{y,d}$, is found with Equation 68. The new desired angular rate for the x-axis and z-axis are zero degrees per second.

$$T = 2\pi\sqrt{\frac{A^3}{\mu}} \tag{67}$$

$$\omega_{y,d} = \frac{360°}{T} = 0.0648°/s \tag{68}$$

The desired angular velocity is used to drive the quaternion kinematics so that a desired quaternion is computed, integrated across time, and used as input for the controller. Equations 19 and 20 are utilized with the current desired quaternion and desired angular velocity as input, yielding the time rate of change of the desired quaternion, which is then integrated with Equation 21. The error quaternion is calculated as before with Equation 41.

The initial conditions of the de-orbit control simulation are the final conditions of the first validation case of the slew maneuver, described in Equation 69.

$$\boldsymbol{q}_0 = \begin{bmatrix} 0.00017 \\ 0.00041 \\ 0.00043 \\ 1.00060 \end{bmatrix}, \qquad \begin{bmatrix} \phi_0 \\ \theta_0 \\ \psi_0 \end{bmatrix} = \begin{bmatrix} 8.63° \\ 2.19° \\ -9.74° \end{bmatrix}, \qquad \boldsymbol{\omega}_0 = \begin{bmatrix} -0.0053°/s \\ -0.0054°/s \\ -0.0021°/s \end{bmatrix} \tag{69}$$

Sliding vector control is used to keep the host spacecraft tracking the changing desired orientation [41]. First, the sliding surface vector is computed with Equation 70, where k is a positive scalar constant that serves as a control gain.

$$\boldsymbol{s} = (\boldsymbol{\omega} - \boldsymbol{\omega}_d) + k * sign(\delta q_4)\boldsymbol{\delta q}_{1:3} \tag{70}$$

Second, a saturation function is applied to the sliding vector in Equation 71 to calculate $\overline{\boldsymbol{s}}$, which targets the desired orientation. The components ϵ_i are positive quantities that act as saturation thresholds. The output torque is calculated with Equation 72 where G is a positive definite matrix. The parameters k, epsilon, and G are optimized via iterative simulation. Their values are reported in Equation 73.

$$\overline{s}_i = sat(s_i, \epsilon_i) = \begin{cases} 1 & for \quad s_i > \epsilon_i \\ 0 & for \quad |s_i| \leq \epsilon_i \\ -1 & for \quad s_i < -\epsilon_i \end{cases} \tag{71}$$

$$\boldsymbol{L} = I_B \left\{ \frac{k}{2} [|\delta q_4|(\boldsymbol{\omega}_d - \boldsymbol{\omega}) - sign(\delta q_4)\boldsymbol{\delta q}_{1:3} \times (\boldsymbol{\omega} + \boldsymbol{\omega}_d)] + \dot{\boldsymbol{\omega}}_d - G\overline{\boldsymbol{s}} \right\} + \boldsymbol{\omega} \times I_B \boldsymbol{\omega} \tag{72}$$

$$k = 1, \quad G = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix}, \quad \varepsilon = \begin{bmatrix} 0.001 \\ 0.001 \\ 0.001 \end{bmatrix} \tag{73}$$

Thruster output torques are checked and re-set to saturated nominal torque values once more. Euler angle and angular velocity results are graphed in Figure 43 and Figure 44, respectively. An enhanced view of the angular velocity components is provided in Figure 44 to show the cyclical nature of using the RCS thrusters to hold the desired angular velocity values for the maneuver. The angular velocity components are difficult to distinguish when they are plotted for the entire maneuver. The data is relevant for the first de-orbit scenario using an initial altitude of 420 km. It takes 38.74 minutes for the de-orbit maneuver to terminate in this scenario. NEIGHBOR drives the host spacecraft to rotate at the desired rate about the y-axis while angular positions are held with respect to the x-axis and the z-axis.

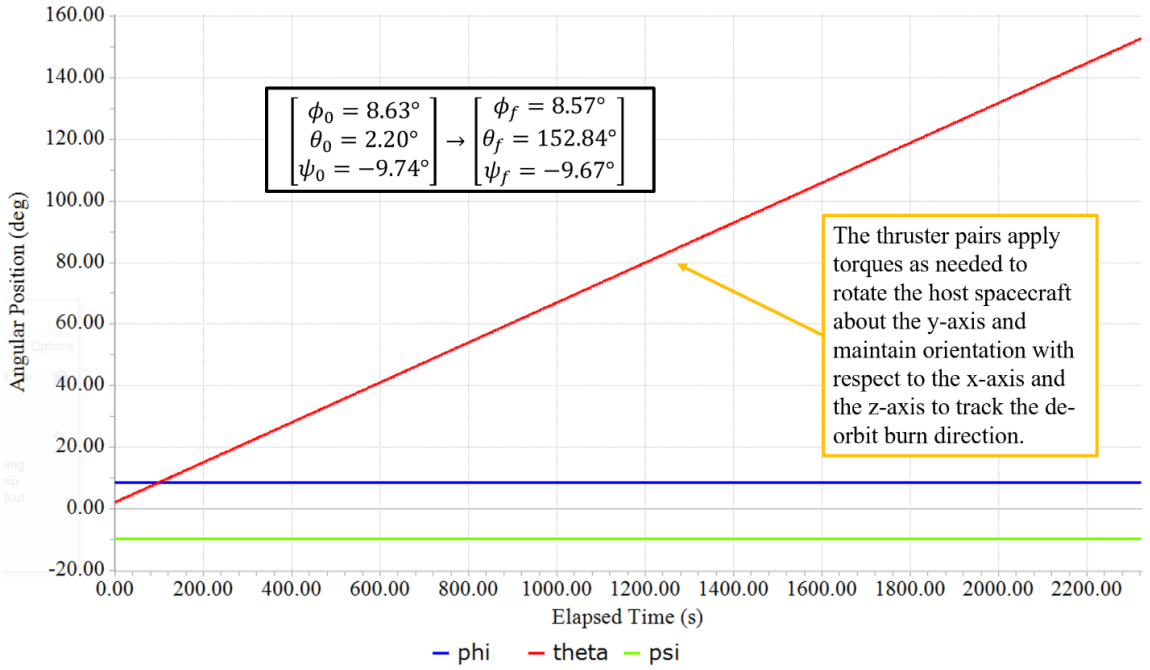


Figure 43. Euler angles are plotted as a function of time for the de-orbit control simulation.

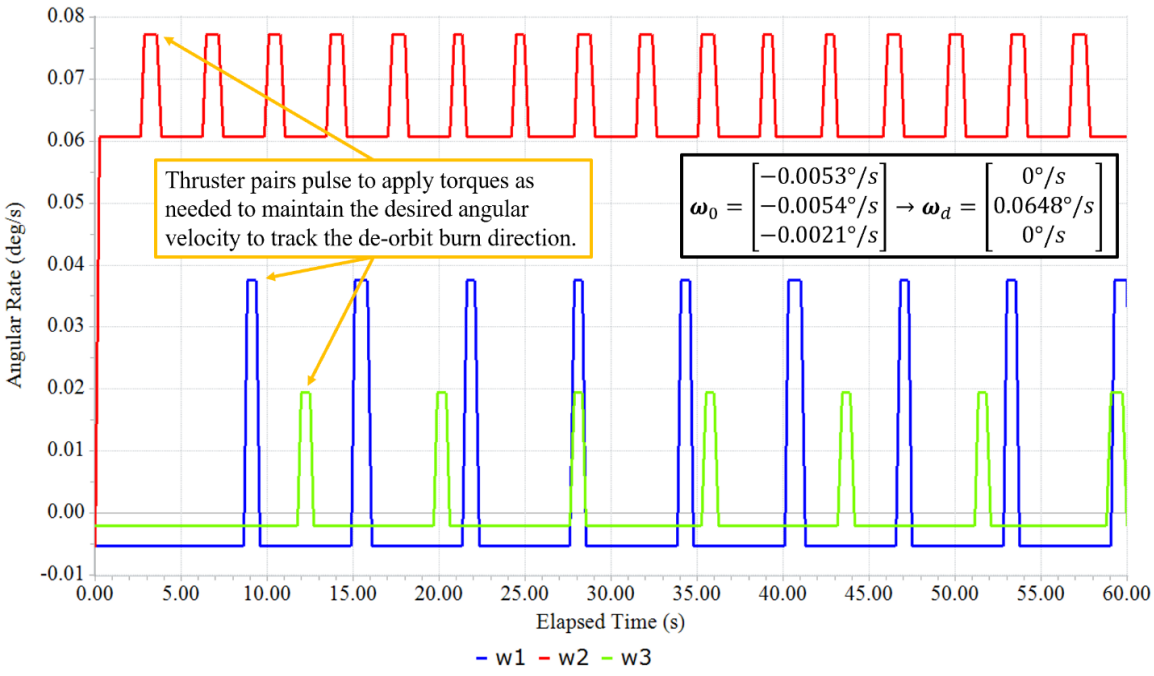


Figure 44. An enhanced view of the angular velocity components is shown for the de-orbit control simulation.

3.3.9 Propellant Tank Sizing

The detumble, slew, and de-orbit control simulations track the time that the thrusters are active and multiply this by the force delivered by two thrusters (since thrusters are always operated as pairs) to calculate total impulse. Equation 74 yields the required propellant mass for the maneuvers, where I_t is total impulse, I_{sp} is specific impulse, and g_0 is the acceleration due to gravity at Earth's surface.

$$m_p = \frac{I_t}{I_{sp} g_0} \tag{74}$$

The required propellant mass is 2.36 g for both validation cases of the detumble maneuver. Validation cases one, two, and three of the slew maneuver require 4.51 g, 9.96 g, and 16.03 g, respectively, to then maneuver the satellite to the burn orientation. The total required propellant mass for attitude control during the de-orbit maneuver for an initial altitude of 420 km is 4.75 g. Summing the mass required for the detumble maneuver, the third validation case (worst case) of the slew maneuver, and the attitude control during the de-orbit maneuver, results in a worst case required propellant mass of 23.14 g. When combining this with the propellant required for the first de-orbit scenario, the total required propellant mass is 0.587 kg.

NEIGHBOR's propulsion system is assumed to use R-236FA as a propellant. The propellant behaves as an ideal gas in storage, and so the ideal gas law is used to determine the volume needed to store the propellant in Equation 75, where V is the volume of the propellant, n is the amount in moles of the propellant, R is the ideal gas constant (8.314 J/K-mol), T is the temperature of the gas, and P is the pressure of the gas. The propellant is assumed to be stored at a pressure of 25 MPa and an operating temperature of 20°C and has a molar mass of 152.04 g/mol [48]. The first de-orbit scenario requires a propellant mass of 0.587 kg for attitude control and de-orbiting, which is equivalent to 3.86 moles. The required volume is 376.4 cm^3 as a result.

$$V = \frac{nRT}{P} \tag{75}$$

NEIGHBOR is designed to use a spherical tank to store propellant. The volume of a sphere is given by V_s in Equation 76, where r_s is the radius of the sphere. A spherical volume of 376.4 cm^3 corresponds to a radius of 4.48 cm.

$$V_s = \frac{4}{3}\pi r_s^3 \tag{76}$$

An expression to determine wall thickness of a thin-walled spherical pressure vessel is provided in Equation 77, where t_w is the spherical tank wall thickness, P is the allowable pressure, r_s is the spherical tank inner radius, σ_y is the yield strength of the tank's wall material, and SF is an applied safety factor [49]. The spherical tank's wall is assumed to be made of Aluminum 6061-T6, which has a tensile yield strength of 276 MPa [50]. A safety factor of two is applied to the spherical tank. The propellant is assumed to be stored at a pressure of 25 MPa. The tank's wall thickness is then 0.41 cm.

$$t_w = \frac{Pr_s}{2\sigma_y}(SF) \tag{77}$$

The outer radius of the spherical tank is 4.88 cm as a result. The outer volume of the sphere is calculated to be 488.2 cm^3 with Equation 76. The volume that the tank shell occupies is the difference between the outer volume and the inner volume of the sphere. This difference is 111.8 cm^3. The tank shell's material, Aluminum 6061-T6, has a density of 2.7 g/cm^3 [50]. The tank shell has a mass of 301.9 g. The total combined mass of the propellant tank shell and the required propellant for the mission is 888.9 g.

Propellant feed lines must be sized to deliver the propellant to the RCS thruster nozzles. The material used for the feed lines is 304 stainless steel, with a yield strength of 205 MPa and a

density of 8 g/cm^3 [51] [52]. The wall thickness of the cylindrical pipes is determined by Equation 78, where D_p is the outer diameter of the pipe [53]. The pressure the pipe needs to withstand is conservatively considered to be 25 MPa. The pipes are constrained with a safety factor of two once more. CubeSat propellant feed systems commonly use one quarter inch outer diameter stainless steel feed lines, which is equivalent to 6.35 mm [54]. The wall thickness of the feed lines is then 0.77 mm. The feed lines therefore have an inner radius of 2.4 mm and an outer radius of 3.18 mm.

$$t_w = \frac{PD_p}{2\sigma_y}(SF) \tag{78}$$

NEIGHBOR's spherical propellant tank is designed to reside in the center of its anticipated 2U volume. Feed lines begin at NEIGHBOR's outer wall below the spherical tank and continue in a straight line to the thrusters at their locations. Lengths of feed lines for each thruster are listed in Table 4. The volumes of the feed lines are determined by using the formula for the volume of a cylinder, V_c, in Equation 79, where r_c is the radius of the cylinder and l_c is the length of the cylinder. The mass of the feed lines is determined by using the density of 304 stainless steel.

$$V_c = \pi r_c^2 l_c \tag{79}$$

Table 4. Thruster feed lines are sized to determine required mass and volume for NEIGHBOR's design. The total required mass is 210.21 g and the required volume is 61.35 cm^3.

Thruster Number	Feed Line Length (cm)	Feed Line Material Volume (cm^3)	Mass (g)	Feed Line Outer Volume (cm^3)
1	11.18	1.52	12.13	3.54
2	11.18	1.52	12.13	3.54
3	11.18	1.52	12.13	3.54
4	11.18	1.52	12.13	3.54
5	10	1.36	10.85	3.17
6	5	0.68	5.43	1.58
7	10	1.36	10.85	3.17
8	5	0.68	5.43	1.58
9	12.25	1.66	13.29	3.88
10	12.25	1.66	13.29	3.88
11	12.25	1.66	13.29	3.88
12	12.25	1.66	13.29	3.88
13	20	2.71	21.70	6.33
14	15	2.03	16.28	4.75
15	20	2.71	21.70	6.33
16	15	2.03	16.28	4.75
		Total	**210.21**	**61.35**

The thruster nozzles have a mass of 0.04 kg each [55]. With sixteen RCS thrusters, the total mass is 0.64 kg. Summing the mass of the propellant tank, the required propellant, the propellant feed lines, and the thruster nozzles, the propulsion assembly comes to 1739.11 g. The propulsion assembly requires a manifold to control the flow of propellant to the feed lines and the thruster nozzles. Its mass is estimated to require 20% of the propulsion assembly, which comes to 347.82 g. The total mass required for the propulsion assembly to support NEIGHBOR's mission is 2086.93 g.

3.4 Command and Data Handling

NEIGHBOR's command and control functionality is provided by an embedded microprocessor executing flight control software. A STM32L432KC processor is a representative device used by the Laboratory for Advanced Space Systems at Illinois in many of its CubeSat missions [56]. It features 256 kB of flash memory and interfaces to most of the common data buses in use today. NEIGHBOR's design includes a UHF transceiver and a deployable antenna system to manage the flow of data and commands between the host spacecraft and the ground station [57] [58].

NEIGHBOR's flight software command functions are organized by operation modes. The first operation mode is the "idle" mode. In this configuration, NEIGHBOR's microprocessor is intermittently powered on by its internal clock to monitor the host spacecraft's heartbeat signal. All other components remain quiescent to conserve power. Between heartbeat signal checks, the microprocessor's clock uses negligible power as it tracks time to decide when the microprocessor activates and deactivates. The idle mode occurs between host spacecraft deployment and the point at which the spacecraft fails or completes its mission. This is when the host spacecraft heartbeat signal is lost.

When NEIGHBOR no longer detects the heartbeat signal, it enters the "grace period" mode. It remains in this mode for a fixed time period that is decided by host spacecraft operators before the main de-orbit mission starts. For the duration of this operational mode, ground station operators may attempt maintenance to try to restore spacecraft functionality. Host spacecraft operators are also able to manually turn off the heartbeat signal once a mission is complete to initiate the de-orbit process. At the end of the grace period, NEIGHBOR's antennas deploy to communicate with the ground station and receive final confirmation to de-orbit.

When the grace period ends with a final confirmation to de-orbit, NEIGHBOR transitions to the "de-orbit" mode. In this mode, NEIGHBOR powers on its attitude sensing equipment, determines the spacecraft's attitude, detumbles it if necessary, and slews it to the burn attitude for the de-orbit maneuver. NEIGHBOR then executes and verifies the de-orbit maneuver.

With the de-orbit maneuver complete, NEIGHBOR enters a “monitoring” mode, periodically beaconing GPS information when in contact with the ground station during its descent from orbit. An operation modes flow diagram is provided in Figure 45 to illustrate this operations concept over the course of NEIGHBOR’s mission and the criteria for entering and exiting each mode.

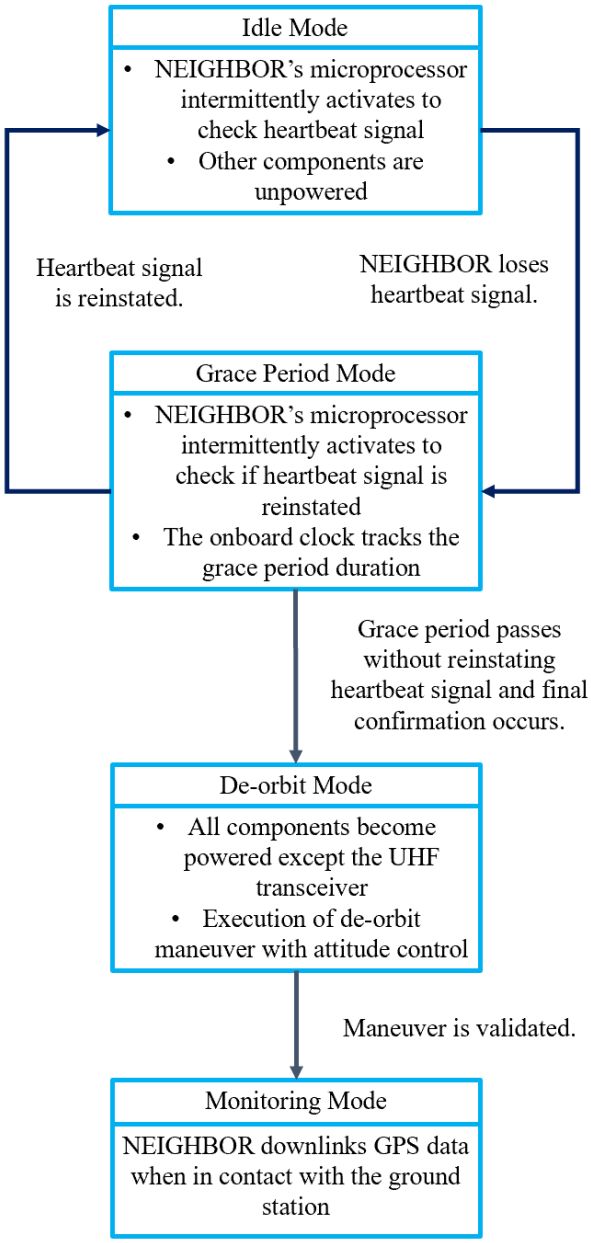


Figure 45. NEIGHBOR operates in four different modes during its mission.

3.5 Power Supply

Before designing NEIGHBOR as a stand-alone, independent system, consideration was given to a more invasive integration approach with a host spacecraft. Taking advantage of highly reliable host spacecraft systems to support NEIGHBOR's functions could decrease the mass and volume impact of incorporating a de-orbit capability like NEIGHBOR into an existing satellite design. One of the more obvious opportunities would be to utilize a host spacecraft power system to energize NEIGHBOR. Unfortunately, it also increases the number of possible points of failure for achieving NEIGHBOR's principal objective. As a result, only a self-contained configuration of NEIGHBOR was considered to further assess feasibility.

NEIGHBOR's own power subsystem consists of the required power storage and distribution hardware. It features regulatory and inhibitory mechanisms to safeguard the host launch vehicle and spacecraft from premature NEIGHBOR activation.

Before launch, NEIGHBOR is inhibited from operation by a hardwire switch that is closed only after the host satellite is deployed in orbit. Once enabled, in the idle mode, NEIGHBOR powers on its microprocessor periodically to check the status of the host's heartbeat signal. The microprocessor is active for a period of five seconds per day to check the signal. During this time, it requires 0.02 W of nominal power. When not checking the signal, the microprocessor operates in a quiescent state, which requires 92.4 nW. This is enough for the internal clock to periodically awaken NEIGHBOR to check the heartbeat signal from the host spacecraft.

In the grace period mode, NEIGHBOR's microprocessor will switch back to its quiescent state and return to the idle mode if a heartbeat signal returns. If the heartbeat does not come back at the end of the grace period, the microprocessor operates with a nominal power requirement and turns on the UHF transceiver to receive confirmation from the ground station to de-orbit. The UHF transceiver requires 2.82 W to operate.

In the de-orbit mode, the microprocessor remains powered on and the UHF transceiver is unpowered. The horizon sensor, sun sensor, gyroscope, GPS, and propulsion system are powered on to determine attitude in preparation for the de-orbit maneuver. The horizon sensor requires

0.15 W, the sun sensor requires 0.12 W, the gyroscope requires 0.02 W, and the GPS requires 1.36 W. After ten minutes of sensing attitude, NEIGHBOR detumbles and slews the host spacecraft to the de-orbit maneuver attitude. The propulsion system requires 0.5 W to operate. The attitude components and the propulsion system remain powered to hold orientation and deliver the required ΔV to de-orbit the host spacecraft.

Once the microprocessor validates the maneuver, it goes back into a quiescent state to monitor the host spacecraft's descent as part of the monitoring mode. NEIGHBOR periodically powers on the GPS (to obtain position data) and the UHF transceiver (to actively downlink spacecraft latitude, longitude, and altitude). The power requirements for each of NEIGHBOR's components are provided in Table 5. The power required for each action in each operation mode is provided in Table 6. The detumble and slew maneuvers are referred to as the attitude control maneuvers in Table 6.

Table 5. The operational power requirements for NEIGHBOR's components are listed.

Component	Power (W)
Propulsion System	0.5
Horizon Sensor	0.15
Sun Sensor	0.115
Gyroscope	0.018
GPS Receiver	1.300
GPS Antenna	0.066
Microprocessor (Nominal)	0.0221
Microprocessor (Quiescent)	$9.24 \cdot 10^{-8}$
UHF Rx	0.1815
UHF Tx	2.64

Table 6. The power requirements for the actions of each operation mode are summarized.

Mode	Action	Power Required (W)
Idle	Checking Heartbeat	0.0221
Idle	Not Checking Heartbeat	$9.24 \cdot 10^{-8}$
Grace Period	Checking Heartbeat	0.0221
Grace Period	Not Checking Heartbeat	$9.24 \cdot 10^{-8}$
Grace Period	Awaiting Final Contact	2.84
Grace Period	Final Contact	1.67
De-orbit	Attitude Determination	2.17
De-orbit	Attitude Control Maneuvers	2.17
De-orbit	De-orbit Maneuver	$9.24 \cdot 10^{-8}$
Quiescent	Microprocessor Quiescent State	4.21
Quiescent	Downlink	0.0221

The power requirements for the actions of each operation mode are scripted in a FreeFlyer simulation according to their operational duty cycles to determine energy used so that it may be subtracted over time from a total battery stored energy. The spacecraft is assumed to remain in idle mode for five years (equal to 1855 days) before the host spacecraft's heartbeat signal is lost. The grace period is set to last thirty days. From the results provided in Section 3.3, the attitude control maneuvers require 176.23 seconds as a maximum (summing the first validation case of the detumble maneuver and the third validation case of the slew maneuver) and the de-orbit maneuver lasts 38.74 minutes for the first de-orbit scenario.

A single LG Chem HG2 INR Lithium-Ion battery is sufficient for the mission [59]. It has a maximum stored energy of 3000 mAh at a voltage of 3.6 V. This corresponds to a maximum stored energy of 10.8 Wh.

A detailed power profile illustrating the end of the grace period, the final confirmation to de-orbit, the attitude determination phase, the attitude control maneuvers, and the de-orbit maneuver is provided in Figure 46. NEIGHBOR's battery drops to a level of 79.65% charge following the de-orbit maneuver, which corresponds to a stored energy requirement of 2.2 Wh perform the necessary actions. After completing the de-orbit burn, the battery still has 8.6 Wh of

energy available. Given this state of charge, the GPS and UHF transmitter could be used infrequently over time to provide occasional altitude updates as atmospheric drag completes the de-orbit process. For instance, NEIGHBOR takes 2.2 years until re-entry for the first de-orbit scenario while the battery is drained following the de-orbit maneuver in a matter of two weeks.

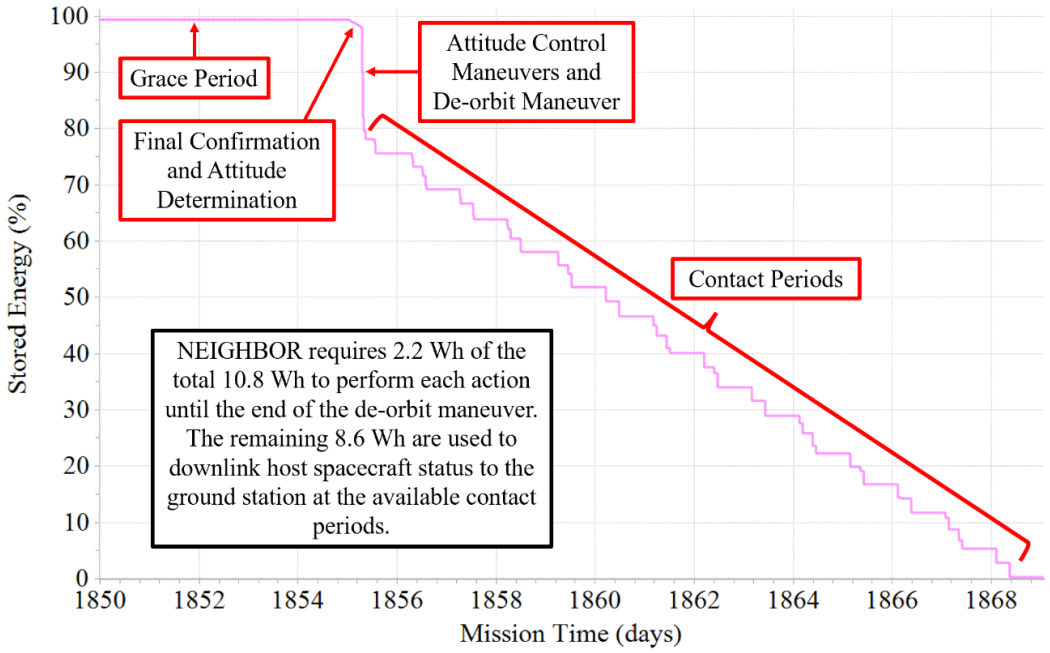


Figure 46. Battery state is tracked as final confirmation, attitude determination, the attitude control maneuvers, the de-orbit maneuver, and several contact periods occur.

3.6 Data Collection and Downlink

The status data that NEIGHBOR collects allows the ground to assess the progress of the de-orbit process of the host spacecraft. Spacecraft position, power status, and propulsion system status are among the data that provide useful information. Status data is sensed in real time and stored with different cadences. The real time cadence for the recording of attitude data is anticipated to be on the order of 50 milliseconds to sufficiently diagnose host spacecraft orientation. Supporting data from the horizon sensor, sun sensor, gyroscope, and GPS are also downlinked intermittently.

Power data is stored once per minute. Tank pressure and valve status are also stored once per minute. Burn start and burn durations are recorded once for the propulsion device's detumbling and de-orbit maneuvers. NEIGHBOR measures a dozen temperatures associated with its subsystems and stores them once each minute. Table 7 provides notional data management information. The average data recording rate is 30 bytes per second.

Table 7. A data budget is provided to organize the required data rates of each component.

Component	Data Field	Data Format	Data Size (bytes)	Storage Cadence (s)
Horizon/Sun Sensor	Attitude Pitch (deg)	Double	4	1
Horizon/Sun Sensor	Attitude Roll (deg)	Double	4	1
Horizon/Sun Sensor	Attitude Yaw (deg)	Double	4	1
Gyroscope	Attitude Pitch Rate (deg/s)	Double	4	1
Gyroscope	Attitude Roll Rate (deg/s)	Double	4	1
Gyroscope	Attitude Yaw Rate (deg/s)	Double	4	1
GPS	Altitude (km)	Double	4	10
GPS	Latitude (deg)	Double	4	10
GPS	Longitude (deg)	Double	4	10
GPS	Time (unix time)	Unsigned Long Int	4	1
ADCS	ADCS Voltage (V)	Double	4	60
ADCS	ADCS Current (A)	Double	4	60
Propulsion Module	Thruster Start (unix time)	Unsigned Long Int	4	Once
Propulsion Module	Burn Duration (unix time)	Unsigned Long Int	4	Once
Propulsion Module	Tank Pressure (Pa)	Float	4	60
Propulsion Module	Valve Status	Boolean	1	60
Thermocouples	Temperatures (K)	Float	48	60
Battery	Battery Voltage (V)	Float	4	60

NEIGHBOR stores all of the information in Table 7 only during the detumbling and de-orbit modes. When in monitoring mode, no data is stored, but NEIGHBOR powers on to continuously beacon GPS data during contact periods with the ground station.

A FreeFlyer script calculates the data storage profile required for NEIGHBOR's mission. The simulation uses the specifications for a STM32L432KC, which has 256 kB of available flash data storage. The UHF transceiver downlinks information at a standard 9600 baud rate, which is equivalent to 1200 bytes per second. The data storage and downlink that occurs in the de-orbit mode is shown in Figure 47. Host spacecraft operators can decide frequency of GPS downlink in the monitoring mode while the power system remains able to support it.

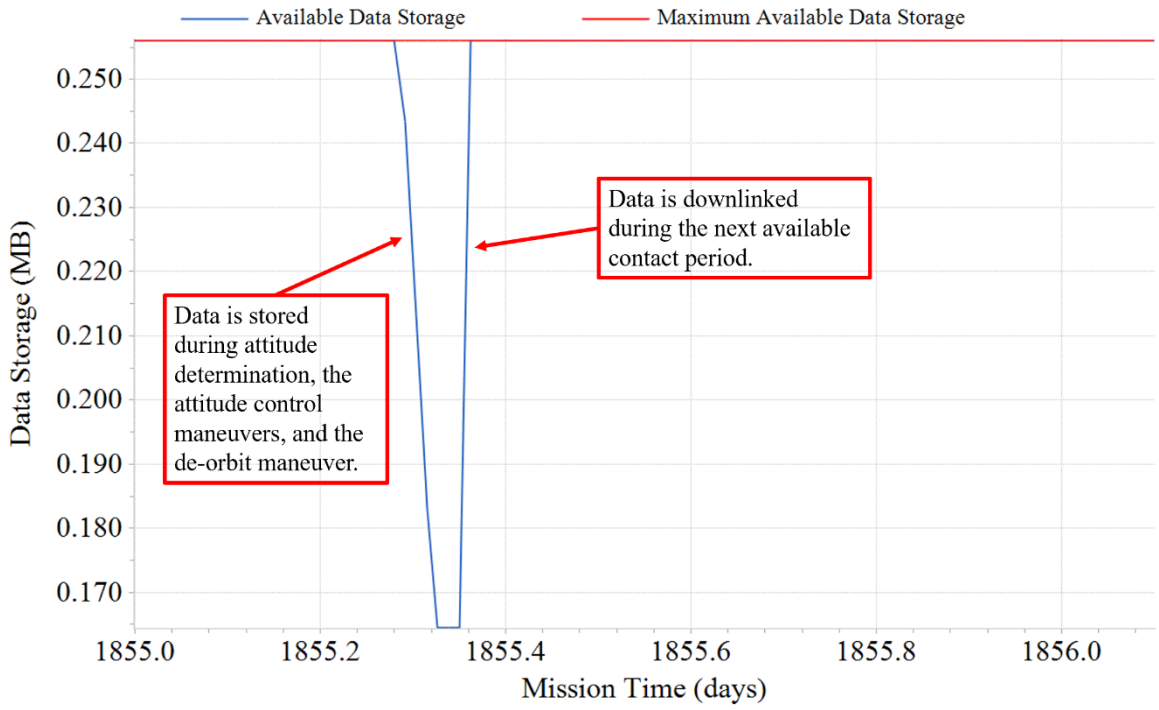


Figure 47. NEIGHBOR's data management is simulated during the de-orbit mode, showing the capability to downlink stored attitude, GPS, propulsion, and host spacecraft health during this critical part of NEIGHBOR's mission.

Available data storage decreases as the host spacecraft stores status information. After NEIGHBOR downlinks the information, the data memory is released and available storage is

recovered. Depending on the geometric relationship between NEIGHBOR's orbit and a representative ground station at the University of Illinois at Urbana-Champaign, several opportunities for contact can occur throughout the day. In this scenario, NEIGHBOR never exceeds a total of 0.1 MB for data storage. Figure 47 shows that two contact periods after the de-orbit maneuver are sufficient to downlink the data captured during the de-orbit mode.

3.7 Interfaces

For NEIGHBOR to function properly, its components must work together in concert to provide the necessary capabilities. For design, this involves a description and illustration of the power and data interfaces between NEIGHBOR's components and the host spacecraft.

With respect to data interfaces, NEIGHBOR uses the heartbeat signal provided by the host spacecraft's watchdog timer hardware. This allows for the detection of host spacecraft failure and the beginning of NEIGHBOR's responsibilities. The attitude determination components send attitude information to the microprocessor, which in turn sends control commands to the attitude control hardware. The attitude determination and control components send maneuver data to the microprocessor to validate the de-orbit maneuver. Altogether, the microprocessor relays this data via the UHF transceiver and antenna to the host spacecraft operator's ground station, which helps the operators understand the state of the spacecraft following its moment of failure.

The components and their interfaces are annotated in a system block diagram divided between Figure 48 and Figure 49 [60]. Red arrows indicate components that are part of the microprocessor's configuration and black lines indicate data interfaces between the microprocessor and NEIGHBOR's other components. Boxes sitting on the periphery indicate types of data interfaces while words near lines indicate signal type. Data interfaces between the microprocessor's board and other components or the host spacecraft are indicated with blue lines and power interfaces are indicated with gold lines.

As for power, the battery and power distribution unit of the host spacecraft route power to the each of NEIGHBOR's components. This is accomplished with the appropriate connections

and harnessing. Through their use, NEIGHBOR's components will generate heat. NEIGHBOR's heat generating hardware is physically attached to the baseplate interfacing with the host satellite that, in its inoperable state, provides a heat sink for NEIGHBOR to dissipate into.

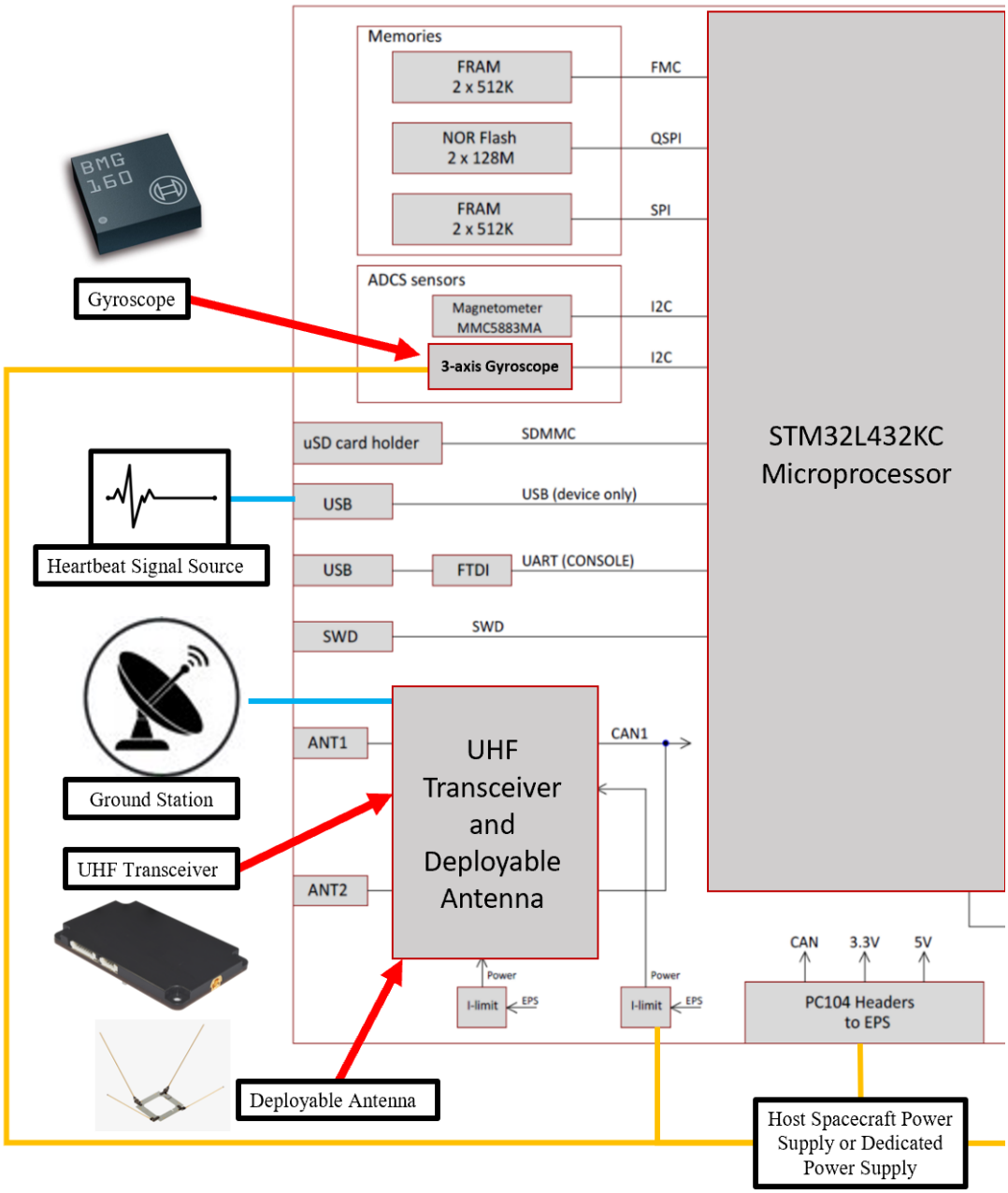


Figure 48. The data and power interfaces are visualized on the left side of the system block diagram. Gold lines indicate power, blue lines indicate data, and red arrows indicate components installed to the microprocessor's board.

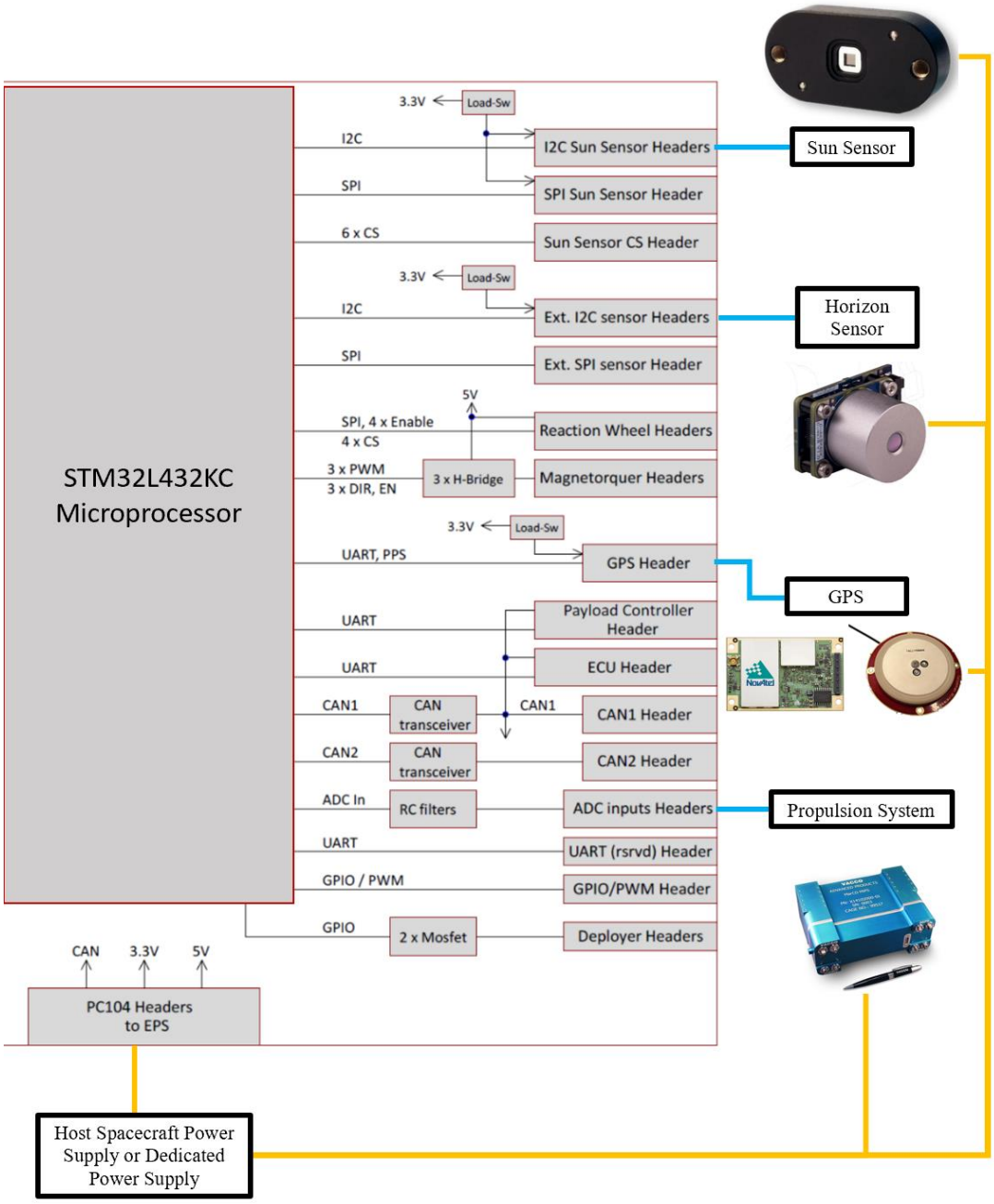


Figure 49. The data and power interfaces are visualized using the right side of the system block diagram. Gold lines indicate power and blue lines indicate data.

3.8 Mass and Volume

A list of the components and their masses for NEIGHBOR's notional self-contained configuration is provided in Table 8. Host spacecraft operators specify the required de-orbit propellant mass that supports their specific mission requirements (i.e., orbital altitude). The mass of the electrical harnessing needed to connect the components and deliver power is assumed to be less than 0.2 kg. The total mass of the NEIGHBOR system is 2652.13 g.

Table 8. The masses of NEIGHBOR's components are listed for its self-contained system configuration.

Component	Mass (g)
Required Propellant	587
Propellant Tank	301.9
Propellant Feed Lines	210.21
Thruster Nozzles	640
Miscellaneous Propulsion Assembly	347.82
Horizon Sensor	120
Sun Sensor	6.5
Gyroscope	0.2
GPS Receiver	31
GPS Antenna	100
Microprocessor	3
UHF Transceiver	24.5
UHF Antenna	33
Battery	47
Harness	200
Total	**2652.13**

A list of the components and their dimensions and volumes for NEIGHBOR's notional self-contained configuration are provided in Table 9. The electrical harnessing is assumed to take up minimal volume. Altogether, NEIGHBOR takes up 1271.84 cm^3, equivalent to 1.27 U of space in the host spacecraft. This leaves some leftover space in the notional 2 U of space that was expected.

Table 9. The volume of each of the components in NEIGHBOR's self-contained configuration is determined from representative component specifications.

Component	Dimensions (mm)	Volume (cm^3)
Propellant Tank	53.10 (radius)	627.3
Propellant Feed Lines	See Section 3.3.9	61.35
Horizon Sensor	90.00 x 92.00 x 50.00	414
Sun Sensor	43.00 x 14.00 x 5.90	3.55
Gyroscope	3.00 x 3.00 x 0.95	0.00855
GPS Receiver	46.00 x 71.00 x 8.00	26.13
GPS Antenna	56.00 x 56.00 x 7.80	24.46
Microprocessor	5.30 x 5.30 x 0.50	0.0140
UHF Transceiver	65.00 x 40.00 x 6.50	16.90
UHF Antenna	99.60 x 99.60 x 7.70	76.39
Battery	65.00 x 18.29 x 18.29	21.74
	Total	**1271.84**

3.9 Risk Analysis and Mitigation

With the capabilities and requirements relevant to NEIGHBOR's mission addressed by its design, some risks still require monitoring. Risks are captured in Table 10 and sorted by their associated likelihood and consequence in Figure 50. For each risk, a mitigation plan is developed.

The first source of risk stems from the possibility of NEIGHBOR causing the host spacecraft to collide with another satellite or space debris during the de-orbit process. To mitigate this risk, host spacecraft operators can make use of available space surveillance information to identify an optimal time to send a de-orbit confirmation signal to NEIGHBOR.

For the second risk, NEIGHBOR faces the chance of losing partial or total functionality due to space weather events like heightened solar activity. Radiation could affect the electronics or the memory and processing performance of NEIGHBOR's microprocessor. This may trigger a premature de-orbit or cause function loss. For mitigation of this risk, NEIGHBOR's components are selected to be radiation tolerant and screened to provide the most reliable parts for its systems.

As a third risk, NEIGHBOR could fail to adequately detumble the host spacecraft, resulting in an inability to execute the de-orbit maneuver. This could occur, for example, if NEIGHBOR fails to properly sense and control attitude as a consequence of software or hardware failure. Mitigation of this risk lies in pre-mission simulation and test for each unique host spacecraft and the usage of reliable commercial-off-the-shelf components to provide NEIGHBOR's capabilities in each case.

The de-orbit maneuver requires NEIGHBOR to simultaneously sense attitude, control attitude, and use its propulsion system to de-orbit the host spacecraft. For a fourth source of risk, a random failure during the de-orbit maneuver could cause its premature termination. This would result in a prolonged stay in orbit and a possible violation of the Federal Communications Commission's five-year rule to de-orbit. This also increases the chance that the host spacecraft collides with other satellites or space debris. This risk is addressed with the same mitigation plan as the previous one: proper simulation, test, and selection of reliable commercial-off-the-shelf components.

Table 10. Risks for NEIGHBOR's mission are summarized.

Risk #	Risk
1	Given that NEIGHBOR has no knowledge of the positions of other satellites and orbital debris, there is a moderate risk of the de-orbit process causing NEIGHBOR to collide with other space objects, potentially causing the generation of space debris.
2	Given that NEIGHBOR's components will be screened for radiation tolerance, they may still be vulnerable to space weather events, such that there is a moderate risk of cosmic radiation adversely impacting NEIGHBOR's subsystems, resulting in partial or temporary loss of function.
3	Given that NEIGHBOR's de-orbit maneuver relies on sufficient detumbling of the host spacecraft, there is a moderate risk of insufficient detumbling due to software or hardware failure, resulting in partial or complete inability to de-orbit the host spacecraft.
4	Given that NEIGHBOR must successfully sense attitude, control attitude, and fully execute a propulsive maneuver to complete its mission, there is a moderate risk of random failure impeding or preventing application of one of the necessary functions, causing a sub-optimal de-orbit maneuver that prolongs the stay of the host spacecraft in orbit.

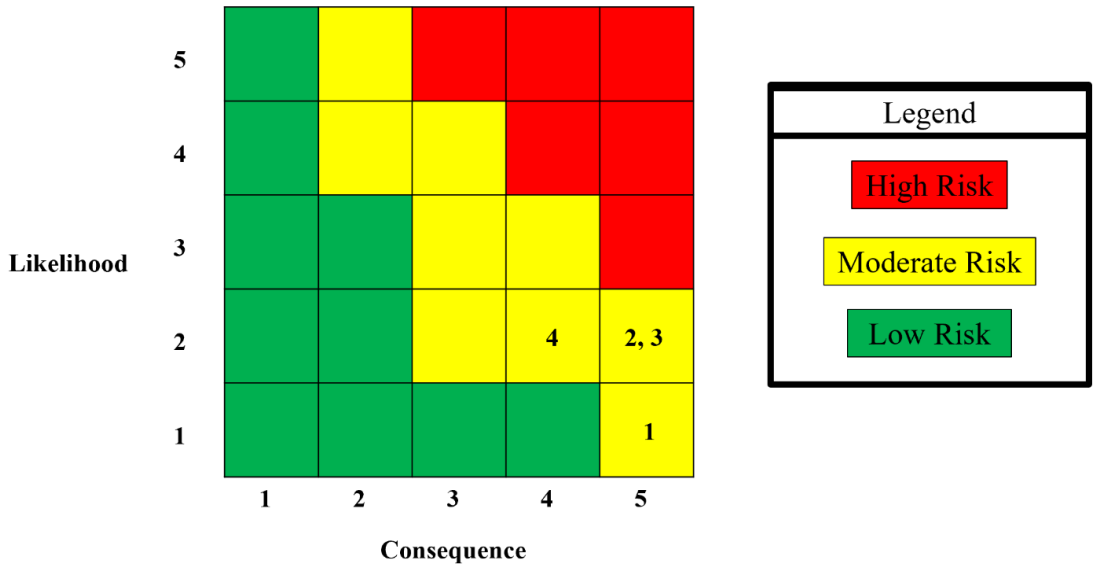


Figure 50. Risks for NEIGHBOR's mission are categorized by likelihood and consequence.

CHAPTER 4: CONCLUSION AND FUTURE WORK

The current space debris environment poses significant challenges for safe spacecraft operations. Debris removal technologies are needed to address this issue. NEIGHBOR's core purpose is to de-orbit small satellites, namely CubeSats, in compliance with the current standard of five years.

For future work, this design can be implemented with varying levels of invasiveness to the host spacecraft. Spacecraft operators can tailor each of the components in this design for their unique mission requirements. It can also be scaled up for larger satellites and modified to supplement the capabilities that a host spacecraft may already feature in its design. To enhance NEIGHBOR's capability to prevent space debris generation, spacecraft operators might incorporate a system for aggregating and sending space surveillance data to NEIGHBOR in its final confirmation phase to avoid de-orbiting along a trajectory that is likely to intercept with nearby debris. NEIGHBOR-like systems on larger spacecraft may have enough propellant to conduct collision avoidance maneuvers with this information as well. NEIGHBOR's stored propellant could be increased to provide the capability for station keeping in addition to de-orbiting.

Addtional research could examine the trade space for reducing time to re-enter depending on the nature of the de-orbit maneuver. The de-orbit maneuvers can vary depending on whether or not re-circularization of the orbit occurs in the high drag zone, whether or not re-entry follows an elliptical trajectory, and the amount of time the spacecraft spends in the high drag zone on a re-circularized orbit (like that resulting from a Hohmann Transfer). The rate of descent has an impact on the change in eccentricity of the orbit, which will affect the time to re-entry. See the work of Toidjanov et al. for similar discussions [3].

This preliminary design analysis demonstrates the feasibility of the proposed debris removal approach. NEIGHBOR controls host spacecraft attitude, prepares the spacecraft for a de-orbit burn, communicates with the ground for maneuver confirmation, executes the de-orbit maneuver, and downlinks the spacecraft's status as it descends. With this novel design,

spacecraft operators can implement an embedded or self-contained system for their CubeSat missions that provides this critical de-orbiting capability.

REFERENCES

[1] IADC Steering Group and Working Group 4, "IADC Space Debris Mitigation Guidelines," Inter-Agency Space Debris Coordination Committee, 2021.

[2] NASA Office of Inspector General, "NASA's Efforts to Mitigate the Risks Posed by Orbital Debris," NASA, 2021.

[3] Toidjanov, A., et al., "End of Life Deorbiting Module for Small Satellites," in *AIAA SciTech Forum*, San Diego, CA & Virtual, 2022.

[4] Kessler, D. J., et al., "The Kessler Syndrome: Implications to Future Space Operations," *Advances in the Astronautical Sciences,* vol. 137, Jan. 2010.

[5] Wiquist, W., "FCC Adopts New '5-Year Rule' for Deorbiting Satellites," Federal Communications Commission, 29 September 2022. https://www.fcc.gov/document/fcc-adopts-new-5-year-rule-deorbiting-satellites.

[6] "ESA's Annual Space Environment Report," ESA Space Debris Office, 2022.

[7] Cowardin, H., and Johnson, A., *NASA Orbital Debris Quarterly News,* vol. 27, no. 1, March 2023.

[8] Anz-Meador, P. D., et al., "History of On-orbit Satellite Fragmentations 15th Edition," NASA Orbital Debris Program Office, 2018.

[9] Cowardin, H., and Shoots, D., *NASA Orbital Debris Quarterly News,* vol. 26, no. 2, June 2022.

[10] Anz-Meador, P., and Shoots, D., *NASA Orbital Debris Quarterly News,* vol. 18, no. 4, Oct. 2014.

[11] Weeden, B., "2007 Chinese Anti-satellite Test Fact Sheet," 23 November 2010. https://swfound.org/media/9550/chinese_asat_fact_sheet_updated_2012.pdf.

[12] Undseth, M., Jolly, C., and Olivari, M., "Space Sustainability: The Economics of Space Debris in Perspective," OECD Publishing, Paris, 2020.

[13] Cowardin, H., and Miller, R., *NASA Orbital Debris Quarterly News,* vol. 26, no. 1, March 2022.

[14] Nicholas, J., "The Collision of Iridium 33 and Cosmos 2251: The Shape of Things to Come," in *International Astronautical Congress*, Daejeon, 2009.

[15] Kessler, D. J., and Cour-Palais, B. G., "Collision Frequency of Artificial Satellites: The Creation of a Debris Belt," *Journal of Geophysical Research,* vol. 83, no. A6, pp. 2637-2646, 1 June 1978.

[16] United States National Research Council, "Limiting Future Collision Risk to Spacecraft: An Assessment of NASA's Meteoroid and Orbital Debris Programs," National Academies Press, Washigton, DC, 2011.

[17] Kurt, J., "Triumph of the Space Commons: Addressing the Impending Space Debris Crisis Without an International Treaty," *William & Mary Environmental Law and Policy Review,* vol. 40, no. 1, 2015.

[18] Adilov, N., Alexander, P. J., and Cunningham, B. M., "An Economic 'Kessler Syndrome': A Dynamic Model of Earth Orbit Debris," *Economics Letters,* vol. 166, pp. 79-82, 2018.

[19] Hearey, C., "When You Wish Upon a 'Starlink': Evaluating the FCC's Actions to Mitigate the Risk of Orbital Debris in the Age of Satellite 'Mega-constellations'," *Administrative Law Review,* vol. 72, no. 4, pp. 751-779, 2020.

[20] Deepak, R. A., and Twiggs, R. J., "Thinking Out of the Box: Space Science Beyond the CubeSat," *Journal of Small Satellites,* vol. 1, no. 1, pp. 3-7, 2012.
[21] Mabrouk, E., "What are SmallSats and CubeSats?," NASA, 7 August 2017. https://www.nasa.gov/content/what-are-smallsats-and-cubesats.
[22] Abdulaziz, A., and Straub, J., "Statistical Analysis of CubeSat Mission Failure," in *Small Satellite Conference*, Logan, UT, 2018.
[23] Kulu, E., "Nanosats Database," https://www.nanosats.eu/#figures.
[24] Swartwout, M., "Reliving 24 Years in the Next 12 Minutes: A Statistical and Personal History of University-Class Satellites," in *Small Satellite Conference*, Logan, UT, 2018.
[25] Swartwout, M., and Jane, C., "University-Class Spacecraft by the Numbers: Success, Failure, Debris. (But Mostly Success.)," in *Small Satellite Conference*, Logan, UT, 2017.
[26] Bouwmeester, J., Menicucci, A., and Gill, E. K. A., "Improving CubeSat Reliability: Subsystem Redundancy or Improved Testing?," *Reliability Engineering and System Safety,* vol. 220, April 2022.
[27] Guo, J., Monas, L., and Gill, E., "Statistical Analysis and Modelling of Small Satellite Reliability," *Acta Astronautica,* vol. 98, pp. 97-110, 2014.
[28] Thyrso, V., et al., "Towards the Thousandth CubeSat: A Statistical Overview," *International Journal of Aerospace Engineering,* vol. 2019, pp. 1-13, 2019.
[29] Swartwout, M., "The First One Hundred CubeSats: A Statistical Look," *Journal of Small Satellites,* vol. 2, no. 2, pp. 213-233, 2013.
[30] "Public Conjunctions," space-track.org, 20 June 2022. https://www.space-track.org/#/conjunctions.
[31] Liou, J.-C., et al., "Stability of the Future LEO Environment - An IADC Comparison Study," Inter-Agency Debris Coordination Committee, 2013.
[32] NASA, "State-of-the-Art Small Spacecraft Technology," National Aeronautics and Space Administration, Moffett Field, CA, 2021.
[33] Lemoine, F. G., et al., "The Development of the Joint NASA GSFC and the National Imagery and Mapping Agency (NIMA) Geopotential Model EGM96," NASA, Greenbelt, MD, 1998.
[34] "Atmospheric Forces," a.i. Solutions, https://ai-solutions.com/_help_Files/atmospheric_forces.htm.
[35] "Jacchia Roberts Density Model," a.i. Solutions, https://ai-solutions.com/_help_Files/jacchia_roberts_density_model.htm.
[36] "RK89 Object," a.i. Solutions, https://ai-solutions.com/_help_Files/rk89_nanosecond.htm.
[37] VACCO Industries, "JPL MarCO Micro CubeSat Propulsion System," https://cubesat-propulsion.com/jpl-marco-micro-propulsion-system/.
[38] SatCatalog, "MarCo MiPs," https://www.satcatalog.com/component/marco-mips/.
[39] "Electron," Rocket Labs, November 2020. https://www.rocketlabusa.com/assets/Uploads/Payload-User-Guide-LAUNCH-V6.6.pdf.
[40] CubeSpace, "CubeIR," CubeSpace, https://www.cubespace.co.za/products/adcs-components/cube-ir/#cube-ir-downloads.
[41] Markley, F. L., and Crassidis, J. L., Fundamentals of Spacecraft Attitude Determination and Control, Microcosm Press and Springer, 2014.
[42] Wertz, J. R., Spacecraft Attitude Determination and Control, Dordrecht: Kluwer Academic Publishers, 1978.

[43] Solar MEMS Technologies, “nanoSSOC-D60 - Digital Sun Sensor for Nanosatellites,” Solar MEMS Technologies, https://www.solar-mems.com/wp-content/uploads/2020/04/Solar-MEMS-nanoSSOC-D60.pdf.
[44] “BMG160 3-axis Gyroscope Sensor,” https://www.mouser.com/datasheet/2/783/BST-BMG160-FL000-01_2012-09-786479.pdf.
[45] NovAtel, “OEM7700 Multi-Frequency GNSS Receiver,” 2022. https://novatel.com/products/receivers/gnss-gps-receiver-boards/oem7700.
[46] Tallysman, “Tallysman TW2106 Embedded Single Band GNSS Antenna,” 2010. http://www.canalgeomatics.com/wp-content/uploads/2020/10/tallysman-tw2106-tw2108-datasheet.pdf.
[47] NanoRacks, “NanoRacks External CubeSat Deployer Interface Definition Document,” 31 August 2018. https://nanoracks.com/wp-content/uploads/Nanoracks-External-Cygnus-Deployer-E-NRCSD-IDD.pdf.
[48] Climalife, “R-236fa,” https://climalife.dehon.com/uploads/media/3/241/241_1746_r236fa-fd-en-13.pdf.
[49] University of Washington, “Pressure Vessels: Combined Stresses,” http://courses.washington.edu/me354a/chap12.pdf.
[50] Aerospace Specification Metals Inc., “Aluminum 6061-T6; 6061-T651,” https://asm.matweb.com/search/SpecificMaterial.asp?bassnum=ma6061t6.
[51] Thomas, “All About 304 Steel (Properties, Strength, and Uses),” https://www.thomasnet.com/articles/metals-metal-products/all-about-304-steel-properties-strength-and-uses/.
[52] Azo Network, “What is 304 Stainless Steel?,” Masteel, https://masteel.co.uk/news/what-304-stainless-steel/.
[53] American Piping Products, “Barlow’s Formula,” https://amerpipe.com/reference/charts-calculators/barlows-formula/.
[54] Polzin, K. A., et al., “The iodine Satellite (iSat) Propellant Feed System - Design and Development,” in *International Electric Propulsion Conference*, Atlanta, 2017.
[55] “100 mN HPGP Thruster,” SatSearch, https://satsearch.co/products/ecaps-100-m-n-hpgp-thruster.
[56] STMicroelectronics, “STM32L432KC,” https://www.st.com/en/microcontrollers-microprocessors/stm32l432kc.html.
[57] GomSpace, “NanoCom AX100,” https://gomspace.com/shop/subsystems/communication-systems/nanocom-ax100.aspx.
[58] NanoAvionics, “CubeSat UHF Antenna System,” https://nanoavionics.com/cubesat-components/cubesat-uhf-antenna/.
[59] LG Chem, “Rechargeable Lithium Ion Battery Model: INR18650HG2 3000mAh,” https://files.batteryjunction.com/frontend/files/lg/datasheet/LG-HG2-18650-INR-Datasheet.pdf.
[60] NanoAvionics, “SatBus 3C2,” 2022. https://satsearch.co/products/nanoavionics-satbus-3c2.

APPENDIX A: ATTITUDE CONTROL AND DE-ORBIT OBJECTS CODE

```
// Mass
Variable m0 = 12; // initial spacecraft total mass [kg]

// Host spacecraft side lengths [m]
Variable a = 0.1;
Variable b = 0.2;
Variable c = 0.3;

// Moments of Inertia [kg-m^2]
Variable Ix = 1/12 * m0 * (a^2 + b^2);
Variable Iy = 1/12 * m0 * (b^2 + c^2);
Variable Iz = 1/12 * m0 * (a^2 + c^2);

Matrix I = [Ix, 0, 0;
            0, Iy, 0;
            0, 0, Iz];

// Thruster Properties
Variable F_MarCO = 0.025; // thrust of one cold gas RCS thruster [N]
Variable g0 = 9.81; // acceleration due to gravity on Earth [m/s^2]

// NEIGHBOR thruster properties (based off MarCO)
NEIGHBOR.Thrusters[0].ThrusterC1 = 4*F_MarCO; // de-orbit maneuver thrust [N]
NEIGHBOR.Thrusters[0].ThrusterK1 = 42; // specific impulse [s]
Variable Isp_MarCO = NEIGHBOR.Thrusters[0].ThrusterK1; // store specific
impulse in variable [s]

// Attitude Dynamics
Matrix I_q = [0; 0; 0; 1]; // identity quaternion
Matrix q; // quaternion representation of spacecraft attitude
Variable q1; // first quaternion element
Variable q2; // second quaternion element
Variable q3; // third quaternion element
Variable q4; // fourth quaternion element
Matrix q0; // initial quaternion
Matrix qf; // final quaternion

Matrix q_d; // desired quaternion
Matrix q_tol; // quaternion element tolerance

Matrix w; // angular velocity of the host spacecraft in the body-fixed frame
[rad/s]
Matrix w0; // initial angular velocity [rad/s]
Matrix wf; // final angular velocity [rad/s]
Matrix w_d; // desired angular velocity [rad/s]
Matrix w_deg; // angular velocity [deg/s]

Matrix w_dot; // time rate of change of angular velocity [rad/s^2]
```

```
Matrix w_d_dot; // desired time rate of change of angular velocity

Matrix w_tol; // desired angular rate tolerance

//// Angular Positions
Variable phi;
Variable theta;
Variable psi;

Variable phi_deg; // conversion to degrees for plotting
Variable theta_deg;
Variable psi_deg;

// Thruster Signals
Array ThrustFlags = {0, 0, 0}; // array to store thruster signals
Variable ThrustFlagX; // thruster operation signal for x-axis
Variable ThrustFlagY; // thruster operation signal for y-axis
Variable ThrustFlagZ; // thruster operation signal for z-axis

// Control
Variable kp; // proportional gain for arbitrary axis
Variable kd; // derivative gain for arbitrary axis

Matrix Kp; // proportional gains
Matrix Kd; // derivative gains

Variable k; // de-orbit tracking scalar gain
Matrix G; // de-orbit tracking gain matrix
Matrix e; // de-orbit tracking saturation function thresholds

// Torques
Variable Lx = ( (a/2)*(2*F_MarCO) ); // torque applied by thruster pair about
x-axis
Variable Ly = ( (c/2)*F_MarCO - ( (c/2) - (c/3) )*F_MarCO ); // torque
applied by thruster pair about y-axis
Variable Lz = ( (c/2)*F_MarCO - ( (c/2) - (c/3) )*F_MarCO ); // torque
applied by thruster pair about z-axis

Matrix L_thrusters = [Lx; Ly; Lz]; // torques the thrusters can apply
Matrix L = [0; 0; 0]; // actual torque applied to the host spacecraft

// Thrust Times
Variable DetumbleThrustTime = 0; // detumble thrust time
Variable SlewThrustTime = 0; // slew thrust time
Variable DeorbitControlThrustTime = 0; // de-orbit control thrust time

// Simulation Time Conditions
Variable t_step = 15e-3; // standard time step [s]

// Mission Time Objects (for capturing times and time differences)
Variable t0; // initial time
Variable tf; // final time
Variable dt; // time step
Variable MissionTime; // mission time
TimeSpan InitialEpoch; // starting epoch
TimeSpan FinalEpoch; // end epoch
String InitialEpochText; // starting epoch text
```

```
String FinalEpochText; // end epoch text

// Calculate the required delta-v value based on the initial altitude
Variable h0 = 550; // initial altitude [km]
Variable hTarget = 300; // target altitude [km]
Variable hf; // final altitude

Variable r1 = Earth.Radius + h0; // initial orbit radius [km]
Variable r2 = Earth.Radius + hTarget; // final orbit radius [km]
Variable DeltaV = -sqrt(Earth.Mu/r1)*(sqrt(2*r2/(r1 + r2)) - 1)*1e3; // [m/s]

// Propellant Analysis
Variable mp = m0*(1 - 1/exp(DeltaV/(Isp_MarCO*g0))); // required propellant
mass [kg]
Variable mf; // final mass
Variable mBurned; // currently utilized propellant mass

// Set Fuel Mass and Dry Mass
(NEIGHBOR.Tanks[0] AsType SphericalTank).TankMass = mp;
NEIGHBOR.VehicleDryMass = m0 - mp;

// De-orbit Maneuver Time Objects
Variable BurnStart; // start of burn
Variable BurnEnd; // end of burn
Variable BurnTime; // burn duration

// Mission scripting and error recovery
Variable NEIGHBORstate; // flag for spacecraft with and without NEIGHBOR
Variable numError = 0; // error count object to terminate some loops
```

APPENDIX B: ATTITUDE CONTROL AND DE-ORBIT PLOT SETTINGS CODE

```
// DETUMBLE PLOTS ------------------------------------------------------------

// Detumble Angular Velocity
PlotWindow DetumbleEulerAngles({NEIGHBOR.ElapsedTime.ToSeconds, phi_deg,
theta_deg, psi_deg});

//DetumbleEulerAngles.PlotTitle.Text = 'Detumble Euler Angles vs. Time';
DetumbleEulerAngles.PlotTitle.Text = ' ';
DetumbleEulerAngles.PlotTitle.Font.Size = 18;
DetumbleEulerAngles.PlotTitle.Visible = 1;

DetumbleEulerAngles.PlotSubTitle.Visible = 0;

DetumbleEulerAngles.XAxis.Title.Text = 'Elapsed Time (s)';
DetumbleEulerAngles.XAxis.Title.Font.Typeface = "Times New Roman";
DetumbleEulerAngles.XAxis.Title.Font.Size = 18;
DetumbleEulerAngles.XAxis.LabelsFont.Typeface = "Times New Roman";
DetumbleEulerAngles.XAxis.LabelsFont.Size = 18;
DetumbleEulerAngles.XAxis.LabelsFormat = "0.00";
DetumbleEulerAngles.XAxis.MaximumValue = 10;

DetumbleEulerAngles.YAxis.Title.Text = 'Angular Position (deg)';
DetumbleEulerAngles.YAxis.Title.Font.Typeface = "Times New Roman";
DetumbleEulerAngles.YAxis.Title.Font.Size = 18;
DetumbleEulerAngles.YAxis.LabelsFont.Typeface = "Times New Roman";
DetumbleEulerAngles.YAxis.LabelsFont.Size = 18;
DetumbleEulerAngles.YAxis.LabelsFormat = "0.00";

DetumbleEulerAngles.Series[0].Label = 'phi';
DetumbleEulerAngles.Series[0].LineWidth = 3;
DetumbleEulerAngles.Series[0].LineStyle = 0;
DetumbleEulerAngles.Series[0].LineColor = ColorTools.Blue;

DetumbleEulerAngles.Series[1].Label = 'theta';
DetumbleEulerAngles.Series[1].LineWidth = 3;
DetumbleEulerAngles.Series[1].LineStyle = 0;
DetumbleEulerAngles.Series[1].LineColor = ColorTools.Red;

DetumbleEulerAngles.Series[2].Label = 'psi';
DetumbleEulerAngles.Series[2].LineWidth = 3;
DetumbleEulerAngles.Series[2].LineStyle = 0;
DetumbleEulerAngles.Series[2].LineColor = ColorTools.LawnGreen;

DetumbleEulerAngles.Legend.Font.Size = 18;

// Detumble Angular Velocity
PlotWindow DetumbleAngularVelocity({NEIGHBOR.ElapsedTime.ToSeconds,
w_deg[0,0], w_deg[1,0], w_deg[2,0]});
```

```
//DetumbleAngularVelocity.PlotTitle.Text = 'Detumble Angular Velocity vs.
Time';
DetumbleAngularVelocity.PlotTitle.Text = ' ';
DetumbleAngularVelocity.PlotTitle.Font.Size = 18;
DetumbleAngularVelocity.PlotTitle.Visible = 1;

DetumbleAngularVelocity.PlotSubTitle.Visible = 0;

DetumbleAngularVelocity.XAxis.Title.Text = 'Elapsed Time (s)';
DetumbleAngularVelocity.XAxis.Title.Font.Typeface = "Times New Roman";
DetumbleAngularVelocity.XAxis.Title.Font.Size = 18;
DetumbleAngularVelocity.XAxis.LabelsFont.Typeface = "Times New Roman";
DetumbleAngularVelocity.XAxis.LabelsFont.Size = 18;
DetumbleAngularVelocity.XAxis.LabelsFormat = "0.00";
DetumbleAngularVelocity.XAxis.MaximumValue = 10;

DetumbleAngularVelocity.YAxis.Title.Text = 'Angular Rate (deg/s)';
DetumbleAngularVelocity.YAxis.Title.Font.Typeface = "Times New Roman";
DetumbleAngularVelocity.YAxis.Title.Font.Size = 18;
DetumbleAngularVelocity.YAxis.LabelsFont.Typeface = "Times New Roman";
DetumbleAngularVelocity.YAxis.LabelsFont.Size = 18;
DetumbleAngularVelocity.YAxis.LabelsFormat = "0.00";
DetumbleAngularVelocity.YAxis.MinimumValue = -1;

DetumbleAngularVelocity.Series[0].Label = 'w1';
DetumbleAngularVelocity.Series[0].LineWidth = 3;
DetumbleAngularVelocity.Series[0].LineStyle = 0;
DetumbleAngularVelocity.Series[0].LineColor = ColorTools.Blue;

DetumbleAngularVelocity.Series[1].Label = 'w2';
DetumbleAngularVelocity.Series[1].LineWidth = 3;
DetumbleAngularVelocity.Series[1].LineStyle = 0;
DetumbleAngularVelocity.Series[1].LineColor = ColorTools.Red;

DetumbleAngularVelocity.Series[2].Label = 'w3';
DetumbleAngularVelocity.Series[2].LineWidth = 3;
DetumbleAngularVelocity.Series[2].LineStyle = 0;
DetumbleAngularVelocity.Series[2].LineColor = ColorTools.LawnGreen;

DetumbleAngularVelocity.Legend.Font.Size = 18;

// Detumble X-axis Pulsing
PlotWindow DetumbleThrustX({NEIGHBOR.ElapsedTime.ToSeconds, ThrustFlags[0]});

//DetumbleThrustX.PlotTitle.Text = 'Detumble X-axis Thrust Signal vs. Time';
DetumbleThrustX.PlotTitle.Text = ' ';
DetumbleThrustX.PlotTitle.Font.Size = 18;
DetumbleThrustX.PlotTitle.Visible = 1;

DetumbleThrustX.PlotSubTitle.Visible = 0;

DetumbleThrustX.Series[0].LineWidth = 1;
DetumbleThrustX.Series[0].LineColor = ColorTools.Blue;

DetumbleThrustX.XAxis.Title.Text = 'Elapsed Time (s)';
DetumbleThrustX.XAxis.Title.Font.Typeface = "Times New Roman";
```

```
DetumbleThrustX.XAxis.Title.Font.Size = 18;
DetumbleThrustX.XAxis.LabelsFont.Typeface = "Times New Roman";
DetumbleThrustX.XAxis.LabelsFont.Size = 18;
DetumbleThrustX.XAxis.LabelsFormat = "0.00";
DetumbleThrustX.XAxis.MaximumValue = 10;

DetumbleThrustX.XAxis.Title.Visible = 0;

DetumbleThrustX.YAxis.Title.Text = 'X-Axis Thrust Signal';
DetumbleThrustX.YAxis.Title.Font.Typeface = "Times New Roman";
DetumbleThrustX.YAxis.Title.Font.Size = 18;
DetumbleThrustX.YAxis.LabelsFont.Typeface = "Times New Roman";
DetumbleThrustX.YAxis.LabelsFont.Size = 18;
DetumbleThrustX.YAxis.LabelsSpacing = 1;
DetumbleThrustX.YAxis.MaximumValue = 1.5;
DetumbleThrustX.YAxis.MinimumValue = -1.5;
DetumbleThrustX.YAxis.LabelsFormat = "0";

DetumbleThrustX.Legend.Visible = 0;

// Detumble Y-axis Pulsing
PlotWindow DetumbleThrustY({NEIGHBOR.ElapsedTime.ToSeconds, ThrustFlags[1]});

//DetumbleThrustY.PlotTitle.Text = 'Detumble Y-axis Thrust Signal vs. Time';
DetumbleThrustY.PlotTitle.Text = ' ';
DetumbleThrustY.PlotTitle.Font.Size = 18;
DetumbleThrustY.PlotTitle.Visible = 1;

DetumbleThrustY.PlotSubTitle.Visible = 0;

DetumbleThrustY.Series[0].LineWidth = 1;
DetumbleThrustY.Series[0].LineColor = ColorTools.Red;

DetumbleThrustY.XAxis.Title.Text = 'Elapsed Time (s)';
DetumbleThrustY.XAxis.Title.Font.Typeface = "Times New Roman";
DetumbleThrustY.XAxis.Title.Font.Size = 18;
DetumbleThrustY.XAxis.LabelsFont.Typeface = "Times New Roman";
DetumbleThrustY.XAxis.LabelsFont.Size = 18;
DetumbleThrustY.XAxis.LabelsFormat = "0.00";
DetumbleThrustY.XAxis.MaximumValue = 10;

DetumbleThrustY.XAxis.Title.Visible = 0;

DetumbleThrustY.YAxis.Title.Text = 'Y-Axis Thrust Signal';
DetumbleThrustY.YAxis.Title.Font.Typeface = "Times New Roman";
DetumbleThrustY.YAxis.Title.Font.Size = 18;
DetumbleThrustY.YAxis.LabelsFont.Typeface = "Times New Roman";
DetumbleThrustY.YAxis.LabelsFont.Size = 18;
DetumbleThrustY.YAxis.LabelsSpacing = 1;
DetumbleThrustY.YAxis.MaximumValue = 1.5;
DetumbleThrustY.YAxis.MinimumValue = -1.5;
DetumbleThrustY.YAxis.LabelsFormat = "0";

DetumbleThrustY.Legend.Visible = 0;

// Detumble Z-axis Pulsing
PlotWindow DetumbleThrustZ({NEIGHBOR.ElapsedTime.ToSeconds, ThrustFlags[2]});
```

```
//DetumbleThrustZ.PlotTitle.Text = 'Detumble Z-axis Thrust Signal vs. Time';
DetumbleThrustZ.PlotTitle.Text = ' ';
DetumbleThrustZ.PlotTitle.Font.Size = 18;
DetumbleThrustZ.PlotTitle.Visible = 1;

DetumbleThrustZ.PlotSubTitle.Visible = 0;

DetumbleThrustZ.Series[0].LineWidth = 1;
DetumbleThrustZ.Series[0].LineColor = ColorTools.LawnGreen;

DetumbleThrustZ.XAxis.Title.Text = 'Elapsed Time (s)';
DetumbleThrustZ.XAxis.Title.Font.Typeface = "Times New Roman";
DetumbleThrustZ.XAxis.Title.Font.Size = 18;
DetumbleThrustZ.XAxis.LabelsFont.Typeface = "Times New Roman";
DetumbleThrustZ.XAxis.LabelsFont.Size = 18;
DetumbleThrustZ.XAxis.LabelsFormat = "0.00";
DetumbleThrustZ.XAxis.MaximumValue = 10;

DetumbleThrustZ.YAxis.Title.Text = 'Z-Axis Thrust Signal';
DetumbleThrustZ.YAxis.Title.Font.Typeface = "Times New Roman";
DetumbleThrustZ.YAxis.Title.Font.Size = 18;
DetumbleThrustZ.YAxis.LabelsFont.Typeface = "Times New Roman";
DetumbleThrustZ.YAxis.LabelsFont.Size = 18;
DetumbleThrustZ.YAxis.LabelsSpacing = 1;
DetumbleThrustZ.YAxis.MaximumValue = 1.5;
DetumbleThrustZ.YAxis.MinimumValue = -1.5;
DetumbleThrustZ.YAxis.LabelsFormat = "0";

DetumbleThrustZ.Legend.Visible = 0;

// SLEW PLOTS ------------------------------------------------------------------

PlotWindow SlewEulerAngles({NEIGHBOR.ElapsedTime.ToSeconds, phi_deg,
theta_deg, psi_deg});

//SlewEulerAngles.PlotTitle.Text = 'Slew Euler Angles vs. Time';
SlewEulerAngles.PlotTitle.Text = ' ';
SlewEulerAngles.PlotTitle.Font.Size = 18;
SlewEulerAngles.PlotTitle.Visible = 1;

SlewEulerAngles.PlotSubTitle.Visible = 0;

SlewEulerAngles.XAxis.Title.Text = 'Elapsed Time (s)';
SlewEulerAngles.XAxis.Title.Font.Typeface = "Times New Roman";
SlewEulerAngles.XAxis.Title.Font.Size = 18;
SlewEulerAngles.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewEulerAngles.XAxis.LabelsFont.Size = 18;
SlewEulerAngles.XAxis.LabelsFormat = "0.00";

SlewEulerAngles.YAxis.Title.Text = 'Angular Position (deg)';
SlewEulerAngles.YAxis.Title.Font.Typeface = "Times New Roman";
SlewEulerAngles.YAxis.Title.Font.Size = 18;
SlewEulerAngles.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewEulerAngles.YAxis.LabelsFont.Size = 18;
SlewEulerAngles.YAxis.LabelsFormat = "0.00";
```

```
SlewEulerAngles.Series[0].Label = 'phi';
SlewEulerAngles.Series[0].LineWidth = 3;
SlewEulerAngles.Series[0].LineStyle = 0;
SlewEulerAngles.Series[0].LineColor = ColorTools.Blue;

SlewEulerAngles.Series[1].Label = 'theta';
SlewEulerAngles.Series[1].LineWidth = 3;
SlewEulerAngles.Series[1].LineStyle = 0;
SlewEulerAngles.Series[1].LineColor = ColorTools.Red;

SlewEulerAngles.Series[2].Label = 'psi';
SlewEulerAngles.Series[2].LineWidth = 3;
SlewEulerAngles.Series[2].LineStyle = 0;
SlewEulerAngles.Series[2].LineColor = ColorTools.LawnGreen;

SlewEulerAngles.Legend.Font.Size = 18;

// Slew Angular Velocity
PlotWindow SlewAngularVelocity({NEIGHBOR.ElapsedTime.ToSeconds, w_deg[0,0],
w_deg[1,0], w_deg[2,0]});

//SlewAngularVelocity.PlotTitle.Text = 'Slew Angular Velocity vs. Time';
SlewAngularVelocity.PlotTitle.Text = ' ';
SlewAngularVelocity.PlotTitle.Font.Size = 18;
SlewAngularVelocity.PlotTitle.Visible = 1;

SlewAngularVelocity.PlotSubTitle.Visible = 0;

SlewAngularVelocity.XAxis.Title.Text = 'Elapsed Time (s)';
SlewAngularVelocity.XAxis.Title.Font.Typeface = "Times New Roman";
SlewAngularVelocity.XAxis.Title.Font.Size = 18;
SlewAngularVelocity.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewAngularVelocity.XAxis.LabelsFont.Size = 18;
SlewAngularVelocity.XAxis.LabelsFormat = "0.00";

SlewAngularVelocity.YAxis.Title.Text = 'Angular Rate (deg/s)';
SlewAngularVelocity.YAxis.Title.Font.Typeface = "Times New Roman";
SlewAngularVelocity.YAxis.Title.Font.Size = 18;
SlewAngularVelocity.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewAngularVelocity.YAxis.LabelsFont.Size = 18;
SlewAngularVelocity.YAxis.LabelsFormat = "0.00";

SlewAngularVelocity.Series[0].Label = 'w1';
SlewAngularVelocity.Series[0].LineWidth = 3;
SlewAngularVelocity.Series[0].LineStyle = 0;
SlewAngularVelocity.Series[0].LineColor = ColorTools.Blue;

SlewAngularVelocity.Series[1].Label = 'w2';
SlewAngularVelocity.Series[1].LineWidth = 3;
SlewAngularVelocity.Series[1].LineStyle = 0;
SlewAngularVelocity.Series[1].LineColor = ColorTools.Red;

SlewAngularVelocity.Series[2].Label = 'w3';
SlewAngularVelocity.Series[2].LineWidth = 3;
SlewAngularVelocity.Series[2].LineStyle = 0;
SlewAngularVelocity.Series[2].LineColor = ColorTools.LawnGreen;
```

```
SlewAngularVelocity.Legend.Font.Size = 18;

SlewAngularVelocity.MaxPoints = 10000;

// Slew Attitude
PlotWindow SlewQuaternion({NEIGHBOR.ElapsedTime.ToSeconds, q[0,0], q[1,0],
q[2,0], q[3,0]});

//SlewQuaternion.PlotTitle.Text = 'Slew Quaternion vs. Time';
SlewQuaternion.PlotTitle.Text = ' ';
SlewQuaternion.PlotTitle.Font.Size = 18;
SlewQuaternion.PlotTitle.Visible = 1;

SlewQuaternion.PlotSubTitle.Visible = 0;

SlewQuaternion.XAxis.Title.Text = 'Elapsed Time (s)';
SlewQuaternion.XAxis.Title.Font.Typeface = "Times New Roman";
SlewQuaternion.XAxis.Title.Font.Size = 18;
SlewQuaternion.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewQuaternion.XAxis.LabelsFont.Size = 18;
SlewQuaternion.XAxis.LabelsFormat = "0.00";

SlewQuaternion.YAxis.Title.Text = 'Quaternion Element Value';
SlewQuaternion.YAxis.Title.Font.Typeface = "Times New Roman";
SlewQuaternion.YAxis.Title.Font.Size = 18;
SlewQuaternion.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewQuaternion.YAxis.LabelsFont.Size = 18;
SlewQuaternion.YAxis.LabelsFormat = "0.00";

SlewQuaternion.Series[0].Label = 'q1';
SlewQuaternion.Series[0].LineWidth = 3;
SlewQuaternion.Series[0].LineStyle = 0;
SlewQuaternion.Series[0].LineColor = ColorTools.Blue;

SlewQuaternion.Series[1].Label = 'q2';
SlewQuaternion.Series[1].LineWidth = 3;
SlewQuaternion.Series[1].LineStyle = 0;
SlewQuaternion.Series[1].LineColor = ColorTools.Red;

SlewQuaternion.Series[2].Label = 'q3';
SlewQuaternion.Series[2].LineWidth = 3;
SlewQuaternion.Series[2].LineStyle = 0;
SlewQuaternion.Series[2].LineColor = ColorTools.LawnGreen;

SlewQuaternion.Series[3].Label = 'q4';
SlewQuaternion.Series[3].LineWidth = 3;
SlewQuaternion.Series[3].LineStyle = 0;
SlewQuaternion.Series[3].LineColor = ColorTools.Magenta;

SlewQuaternion.Legend.Font.Size = 18;

SlewQuaternion.MaxPoints = 10000;

// Slew X-axis Pulsing
PlotWindow SlewThrustX({NEIGHBOR.ElapsedTime.ToSeconds, ThrustFlags[0]});

SlewThrustX.PlotTitle.Text = 'Slew X-axis Thrust Signal vs. Time';
```

```
SlewThrustX.PlotTitle.Font.Size = 18;
SlewThrustX.PlotSubTitle.Visible = 0;

SlewThrustX.XAxis.Title.Text = 'Elapsed Time (s)';
SlewThrustX.XAxis.Title.Font.Typeface = "Times New Roman";
SlewThrustX.XAxis.Title.Font.Size = 18;
SlewThrustX.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewThrustX.XAxis.LabelsFont.Size = 18;
SlewThrustX.XAxis.LabelsFormat = "0.00";

SlewThrustX.YAxis.Title.Text = 'X-Axis Thrust Signal';
SlewThrustX.YAxis.Title.Font.Typeface = "Times New Roman";
SlewThrustX.YAxis.Title.Font.Size = 18;
SlewThrustX.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewThrustX.YAxis.LabelsFont.Size = 18;
SlewThrustX.YAxis.LabelsSpacing = 1;
SlewThrustX.YAxis.MaximumValue = 1.5;
SlewThrustX.YAxis.MinimumValue = -1.5;
SlewThrustX.YAxis.LabelsFormat = "0";

SlewThrustX.Legend.Visible = 0;

SlewThrustX.MaxPoints = 10000;

// Slew Y-axis Pulsing
PlotWindow SlewThrustY({NEIGHBOR.ElapsedTime.ToSeconds, ThrustFlags[1]});

SlewThrustY.PlotTitle.Text = 'Slew Y-axis Thrust Signal vs. Time';
SlewThrustY.PlotTitle.Font.Size = 18;
SlewThrustY.PlotSubTitle.Visible = 0;

SlewThrustY.XAxis.Title.Text = 'Elapsed Time (s)';
SlewThrustY.XAxis.Title.Font.Typeface = "Times New Roman";
SlewThrustY.XAxis.Title.Font.Size = 18;
SlewThrustY.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewThrustY.XAxis.LabelsFont.Size = 18;
SlewThrustY.XAxis.LabelsFormat = "0.00";

SlewThrustY.YAxis.Title.Text = 'Y-Axis Thrust Signal';
SlewThrustY.YAxis.Title.Font.Typeface = "Times New Roman";
SlewThrustY.YAxis.Title.Font.Size = 18;
SlewThrustY.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewThrustY.YAxis.LabelsFont.Size = 18;
SlewThrustY.YAxis.LabelsSpacing = 1;
SlewThrustY.YAxis.MaximumValue = 1.5;
SlewThrustY.YAxis.MinimumValue = -1.5;
SlewThrustY.YAxis.LabelsFormat = "0";

SlewThrustY.Legend.Visible = 0;

SlewThrustY.MaxPoints = 10000;

// Slew Z-axis Pulsing
PlotWindow SlewThrustZ({NEIGHBOR.ElapsedTime.ToSeconds, ThrustFlags[2]});

SlewThrustZ.PlotTitle.Text = 'Slew Z-axis Thrust Signal vs. Time';
SlewThrustZ.PlotTitle.Font.Size = 18;
```

```
SlewThrustZ.PlotSubTitle.Visible = 0;

SlewThrustZ.XAxis.Title.Text = 'Elapsed Time (s)';
SlewThrustZ.XAxis.Title.Font.Typeface = "Times New Roman";
SlewThrustZ.XAxis.Title.Font.Size = 18;
SlewThrustZ.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewThrustZ.XAxis.LabelsFont.Size = 18;
SlewThrustZ.XAxis.LabelsFormat = "0.00";

SlewThrustZ.YAxis.Title.Text = 'Z-Axis Thrust Signal';
SlewThrustZ.YAxis.Title.Font.Typeface = "Times New Roman";
SlewThrustZ.YAxis.Title.Font.Size = 18;
SlewThrustZ.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewThrustZ.YAxis.LabelsFont.Size = 18;
SlewThrustZ.YAxis.LabelsSpacing = 1;
SlewThrustZ.YAxis.MaximumValue = 1.5;
SlewThrustZ.YAxis.MinimumValue = -1.5;
SlewThrustZ.YAxis.LabelsFormat = "0";

SlewThrustZ.Legend.Visible = 0;

SlewThrustZ.MaxPoints = 10000;

// SLEW UPPER BOUND X AXIS ------------------------------------------------

PlotWindow SlewEulerAnglesX({NEIGHBOR.ElapsedTime.ToSeconds, phi_deg,
theta_deg, psi_deg});

//SlewEulerAnglesX.PlotTitle.Text = 'Slew Euler Angles vs. Time for X-Axis';
SlewEulerAnglesX.PlotTitle.Text = ' ';
SlewEulerAnglesX.PlotTitle.Font.Size = 18;
SlewEulerAnglesX.PlotTitle.Visible = 1;

SlewEulerAnglesX.PlotSubTitle.Visible = 0;

SlewEulerAnglesX.XAxis.Title.Text = 'Elapsed Time (s)';
SlewEulerAnglesX.XAxis.Title.Font.Typeface = "Times New Roman";
SlewEulerAnglesX.XAxis.Title.Font.Size = 18;
SlewEulerAnglesX.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewEulerAnglesX.XAxis.LabelsFont.Size = 18;
SlewEulerAnglesX.XAxis.LabelsFormat = "0.00";

SlewEulerAnglesX.YAxis.Title.Text = 'Angular Position (deg)';
SlewEulerAnglesX.YAxis.Title.Font.Typeface = "Times New Roman";
SlewEulerAnglesX.YAxis.Title.Font.Size = 18;
SlewEulerAnglesX.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewEulerAnglesX.YAxis.LabelsFont.Size = 18;
SlewEulerAnglesX.YAxis.LabelsFormat = "0.00";

SlewEulerAnglesX.Series[0].Label = 'phi';
SlewEulerAnglesX.Series[0].LineWidth = 3;
SlewEulerAnglesX.Series[0].LineStyle = 0;
SlewEulerAnglesX.Series[0].LineColor = ColorTools.Blue;

SlewEulerAnglesX.Series[1].Label = 'theta';
SlewEulerAnglesX.Series[1].LineWidth = 3;
SlewEulerAnglesX.Series[1].LineStyle = 0;
```

```
SlewEulerAnglesX.Series[1].LineColor = ColorTools.Red;

SlewEulerAnglesX.Series[2].Label = 'psi';
SlewEulerAnglesX.Series[2].LineWidth = 3;
SlewEulerAnglesX.Series[2].LineStyle = 0;
SlewEulerAnglesX.Series[2].LineColor = ColorTools.LawnGreen;

SlewEulerAnglesX.Legend.Font.Size = 18;

// Slew Angular Velocity
PlotWindow SlewAngularVelocityX({NEIGHBOR.ElapsedTime.ToSeconds, w_deg[0,0],
w_deg[1,0], w_deg[2,0]});

//SlewAngularVelocityX.PlotTitle.Text = 'Slew Angular Velocity vs. Time for
X-Axis';
SlewAngularVelocityX.PlotTitle.Text = ' ';
SlewAngularVelocityX.PlotTitle.Font.Size = 18;
SlewAngularVelocityX.PlotTitle.Visible = 1;

SlewAngularVelocityX.PlotSubTitle.Visible = 0;

SlewAngularVelocityX.XAxis.Title.Text = 'Elapsed Time (s)';
SlewAngularVelocityX.XAxis.Title.Font.Typeface = "Times New Roman";
SlewAngularVelocityX.XAxis.Title.Font.Size = 18;
SlewAngularVelocityX.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewAngularVelocityX.XAxis.LabelsFont.Size = 18;
SlewAngularVelocityX.XAxis.LabelsFormat = "0.00";

SlewAngularVelocityX.YAxis.Title.Text = 'Angular Rate (deg/s)';
SlewAngularVelocityX.YAxis.Title.Font.Typeface = "Times New Roman";
SlewAngularVelocityX.YAxis.Title.Font.Size = 18;
SlewAngularVelocityX.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewAngularVelocityX.YAxis.LabelsFont.Size = 18;
SlewAngularVelocityX.YAxis.LabelsFormat = "0.00";

SlewAngularVelocityX.Series[0].Label = 'w1';
SlewAngularVelocityX.Series[0].LineWidth = 3;
SlewAngularVelocityX.Series[0].LineStyle = 0;
SlewAngularVelocityX.Series[0].LineColor = ColorTools.Blue;

SlewAngularVelocityX.Series[1].Label = 'w2';
SlewAngularVelocityX.Series[1].LineWidth = 3;
SlewAngularVelocityX.Series[1].LineStyle = 0;
SlewAngularVelocityX.Series[1].LineColor = ColorTools.Red;

SlewAngularVelocityX.Series[2].Label = 'w3';
SlewAngularVelocityX.Series[2].LineWidth = 3;
SlewAngularVelocityX.Series[2].LineStyle = 0;
SlewAngularVelocityX.Series[2].LineColor = ColorTools.LawnGreen;

SlewAngularVelocityX.Legend.Font.Size = 18;

SlewAngularVelocityX.MaxPoints = 10000;

// Slew Attitude
PlotWindow SlewQuaternionX({NEIGHBOR.ElapsedTime.ToSeconds, q[0,0], q[1,0],
q[2,0], q[3,0]});
```

```
//SlewQuaternionX.PlotTitle.Text = 'Slew Quaternion vs. Time for X-Axis';
SlewQuaternionX.PlotTitle.Text = ' ';
SlewQuaternionX.PlotTitle.Font.Size = 18;
SlewQuaternionX.PlotTitle.Visible = 1;

SlewQuaternionX.PlotSubTitle.Visible = 0;

SlewQuaternionX.XAxis.Title.Text = 'Elapsed Time (s)';
SlewQuaternionX.XAxis.Title.Font.Typeface = "Times New Roman";
SlewQuaternionX.XAxis.Title.Font.Size = 18;
SlewQuaternionX.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewQuaternionX.XAxis.LabelsFont.Size = 18;
SlewQuaternionX.XAxis.LabelsFormat = "0.00";

SlewQuaternionX.YAxis.Title.Text = 'Quaternion Element Value';
SlewQuaternionX.YAxis.Title.Font.Typeface = "Times New Roman";
SlewQuaternionX.YAxis.Title.Font.Size = 18;
SlewQuaternionX.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewQuaternionX.YAxis.LabelsFont.Size = 18;
SlewQuaternionX.YAxis.LabelsFormat = "0.00";

SlewQuaternionX.Series[0].Label = 'q1';
SlewQuaternionX.Series[0].LineWidth = 3;
SlewQuaternionX.Series[0].LineStyle = 0;
SlewQuaternionX.Series[0].LineColor = ColorTools.Blue;

SlewQuaternionX.Series[1].Label = 'q2';
SlewQuaternionX.Series[1].LineWidth = 3;
SlewQuaternionX.Series[1].LineStyle = 0;
SlewQuaternionX.Series[1].LineColor = ColorTools.Red;

SlewQuaternionX.Series[2].Label = 'q3';
SlewQuaternionX.Series[2].LineWidth = 3;
SlewQuaternionX.Series[2].LineStyle = 0;
SlewQuaternionX.Series[2].LineColor = ColorTools.LawnGreen;

SlewQuaternionX.Series[3].Label = 'q4';
SlewQuaternionX.Series[3].LineWidth = 3;
SlewQuaternionX.Series[3].LineStyle = 0;
SlewQuaternionX.Series[3].LineColor = ColorTools.Magenta;

SlewQuaternionX.Legend.Font.Size = 18;

SlewQuaternionX.MaxPoints = 10000;

// SLEW UPPER BOUND Y AXIS ------------------------------------------------

PlotWindow SlewEulerAnglesY({NEIGHBOR.ElapsedTime.ToSeconds, phi_deg,
theta_deg, psi_deg});

//SlewEulerAnglesY.PlotTitle.Text = 'Slew Euler Angles vs. Time for Y-Axis';
SlewEulerAnglesY.PlotTitle.Text = ' ';
SlewEulerAnglesY.PlotTitle.Font.Size = 18;
SlewEulerAnglesY.PlotTitle.Visible = 1;

SlewEulerAnglesY.PlotSubTitle.Visible = 0;
```

```
SlewEulerAnglesY.XAxis.Title.Text = 'Elapsed Time (s)';
SlewEulerAnglesY.XAxis.Title.Font.Typeface = "Times New Roman";
SlewEulerAnglesY.XAxis.Title.Font.Size = 18;
SlewEulerAnglesY.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewEulerAnglesY.XAxis.LabelsFont.Size = 18;
SlewEulerAnglesY.XAxis.LabelsFormat = "0.00";

SlewEulerAnglesY.YAxis.Title.Text = 'Angular Position (deg)';
SlewEulerAnglesY.YAxis.Title.Font.Typeface = "Times New Roman";
SlewEulerAnglesY.YAxis.Title.Font.Size = 18;
SlewEulerAnglesY.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewEulerAnglesY.YAxis.LabelsFont.Size = 18;
SlewEulerAnglesY.YAxis.LabelsFormat = "0.00";

SlewEulerAnglesY.Series[0].Label = 'phi';
SlewEulerAnglesY.Series[0].LineWidth = 3;
SlewEulerAnglesY.Series[0].LineStyle = 0;
SlewEulerAnglesY.Series[0].LineColor = ColorTools.Blue;

SlewEulerAnglesY.Series[1].Label = 'theta';
SlewEulerAnglesY.Series[1].LineWidth = 3;
SlewEulerAnglesY.Series[1].LineStyle = 0;
SlewEulerAnglesY.Series[1].LineColor = ColorTools.Red;

SlewEulerAnglesY.Series[2].Label = 'psi';
SlewEulerAnglesY.Series[2].LineWidth = 3;
SlewEulerAnglesY.Series[2].LineStyle = 0;
SlewEulerAnglesY.Series[2].LineColor = ColorTools.LawnGreen;

SlewEulerAnglesY.Legend.Font.Size = 18;

// Slew Angular Velocity
PlotWindow SlewAngularVelocityY({NEIGHBOR.ElapsedTime.ToSeconds, w_deg[0,0],
w_deg[1,0], w_deg[2,0]});

//SlewAngularVelocityY.PlotTitle.Text = 'Slew Angular Velocity vs. Time for
Y-Axis';
SlewAngularVelocityY.PlotTitle.Text = ' ';
SlewAngularVelocityY.PlotTitle.Font.Size = 18;
SlewAngularVelocityY.PlotTitle.Visible = 1;

SlewAngularVelocityY.PlotSubTitle.Visible = 0;

SlewAngularVelocityY.XAxis.Title.Text = 'Elapsed Time (s)';
SlewAngularVelocityY.XAxis.Title.Font.Typeface = "Times New Roman";
SlewAngularVelocityY.XAxis.Title.Font.Size = 18;
SlewAngularVelocityY.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewAngularVelocityY.XAxis.LabelsFont.Size = 18;
SlewAngularVelocityY.XAxis.LabelsFormat = "0.00";

SlewAngularVelocityY.YAxis.Title.Text = 'Angular Rate (deg/s)';
SlewAngularVelocityY.YAxis.Title.Font.Typeface = "Times New Roman";
SlewAngularVelocityY.YAxis.Title.Font.Size = 18;
SlewAngularVelocityY.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewAngularVelocityY.YAxis.LabelsFont.Size = 18;
SlewAngularVelocityY.YAxis.LabelsFormat = "0.00";
```

```
SlewAngularVelocityY.Series[0].Label = 'w1';
SlewAngularVelocityY.Series[0].LineWidth = 3;
SlewAngularVelocityY.Series[0].LineStyle = 0;
SlewAngularVelocityY.Series[0].LineColor = ColorTools.Blue;

SlewAngularVelocityY.Series[1].Label = 'w2';
SlewAngularVelocityY.Series[1].LineWidth = 3;
SlewAngularVelocityY.Series[1].LineStyle = 0;
SlewAngularVelocityY.Series[1].LineColor = ColorTools.Red;

SlewAngularVelocityY.Series[2].Label = 'w3';
SlewAngularVelocityY.Series[2].LineWidth = 3;
SlewAngularVelocityY.Series[2].LineStyle = 0;
SlewAngularVelocityY.Series[2].LineColor = ColorTools.LawnGreen;

SlewAngularVelocityY.Legend.Font.Size = 18;

SlewAngularVelocityY.MaxPoints = 10000;

// Slew Attitude
PlotWindow SlewQuaternionY({NEIGHBOR.ElapsedTime.ToSeconds, q[0,0], q[1,0],
q[2,0], q[3,0]});

//SlewQuaternionY.PlotTitle.Text = 'Slew Quaternion vs. Time for Y-Axis';
SlewQuaternionY.PlotTitle.Text = ' ';
SlewQuaternionY.PlotTitle.Font.Size = 18;
SlewQuaternionY.PlotTitle.Visible = 1;

SlewQuaternionY.PlotSubTitle.Visible = 0;

SlewQuaternionY.XAxis.Title.Text = 'Elapsed Time (s)';
SlewQuaternionY.XAxis.Title.Font.Typeface = "Times New Roman";
SlewQuaternionY.XAxis.Title.Font.Size = 18;
SlewQuaternionY.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewQuaternionY.XAxis.LabelsFont.Size = 18;
SlewQuaternionY.XAxis.LabelsFormat = "0.00";

SlewQuaternionY.YAxis.Title.Text = 'Quaternion Element Value';
SlewQuaternionY.YAxis.Title.Font.Typeface = "Times New Roman";
SlewQuaternionY.YAxis.Title.Font.Size = 18;
SlewQuaternionY.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewQuaternionY.YAxis.LabelsFont.Size = 18;
SlewQuaternionY.YAxis.LabelsFormat = "0.00";

SlewQuaternionY.Series[0].Label = 'q1';
SlewQuaternionY.Series[0].LineWidth = 3;
SlewQuaternionY.Series[0].LineStyle = 0;
SlewQuaternionY.Series[0].LineColor = ColorTools.Blue;

SlewQuaternionY.Series[1].Label = 'q2';
SlewQuaternionY.Series[1].LineWidth = 3;
SlewQuaternionY.Series[1].LineStyle = 0;
SlewQuaternionY.Series[1].LineColor = ColorTools.Red;

SlewQuaternionY.Series[2].Label = 'q3';
SlewQuaternionY.Series[2].LineWidth = 3;
```

```
SlewQuaternionY.Series[2].LineStyle = 0;
SlewQuaternionY.Series[2].LineColor = ColorTools.LawnGreen;

SlewQuaternionY.Series[3].Label = 'q4';
SlewQuaternionY.Series[3].LineWidth = 3;
SlewQuaternionY.Series[3].LineStyle = 0;
SlewQuaternionY.Series[3].LineColor = ColorTools.Magenta;

SlewQuaternionY.Legend.Font.Size = 18;

SlewQuaternionY.MaxPoints = 10000;

// SLEW UPPER BOUND Z AXIS ------------------------------------------------

PlotWindow SlewEulerAnglesZ({NEIGHBOR.ElapsedTime.ToSeconds, phi_deg,
theta_deg, psi_deg});

//SlewEulerAnglesZ.PlotTitle.Text = 'Slew Euler Angles vs. Time for Z-Axis';
SlewEulerAnglesZ.PlotTitle.Text = ' ';
SlewEulerAnglesZ.PlotTitle.Font.Size = 18;
SlewEulerAnglesZ.PlotSubTitle.Visible = 0;

SlewEulerAnglesZ.XAxis.Title.Text = 'Elapsed Time (s)';
SlewEulerAnglesZ.XAxis.Title.Font.Typeface = "Times New Roman";
SlewEulerAnglesZ.XAxis.Title.Font.Size = 18;
SlewEulerAnglesZ.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewEulerAnglesZ.XAxis.LabelsFont.Size = 18;
SlewEulerAnglesZ.XAxis.LabelsFormat = "0.00";

SlewEulerAnglesZ.YAxis.Title.Text = 'Angular Position (deg)';
SlewEulerAnglesZ.YAxis.Title.Font.Typeface = "Times New Roman";
SlewEulerAnglesZ.YAxis.Title.Font.Size = 18;
SlewEulerAnglesZ.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewEulerAnglesZ.YAxis.LabelsFont.Size = 18;
SlewEulerAnglesZ.YAxis.LabelsFormat = "0.00";

SlewEulerAnglesZ.Series[0].Label = 'phi';
SlewEulerAnglesZ.Series[0].LineWidth = 3;
SlewEulerAnglesZ.Series[0].LineStyle = 0;
SlewEulerAnglesZ.Series[0].LineColor = ColorTools.Blue;

SlewEulerAnglesZ.Series[1].Label = 'theta';
SlewEulerAnglesZ.Series[1].LineWidth = 3;
SlewEulerAnglesZ.Series[1].LineStyle = 0;
SlewEulerAnglesZ.Series[1].LineColor = ColorTools.Red;

SlewEulerAnglesZ.Series[2].Label = 'psi';
SlewEulerAnglesZ.Series[2].LineWidth = 3;
SlewEulerAnglesZ.Series[2].LineStyle = 0;
SlewEulerAnglesZ.Series[2].LineColor = ColorTools.LawnGreen;

SlewEulerAnglesZ.Legend.Font.Size = 18;

// Slew Angular Velocity
PlotWindow SlewAngularVelocityZ({NEIGHBOR.ElapsedTime.ToSeconds, w_deg[0,0],
w_deg[1,0], w_deg[2,0]});
```

```
//SlewAngularVelocityZ.PlotTitle.Text = 'Slew Angular Velocity vs. Time for
Z-Axis';
SlewAngularVelocityZ.PlotTitle.Text = ' ';
SlewAngularVelocityZ.PlotTitle.Font.Size = 18;
SlewAngularVelocityZ.PlotTitle.Visible = 1;

SlewAngularVelocityZ.PlotSubTitle.Visible = 0;

SlewAngularVelocityZ.XAxis.Title.Text = 'Elapsed Time (s)';
SlewAngularVelocityZ.XAxis.Title.Font.Typeface = "Times New Roman";
SlewAngularVelocityZ.XAxis.Title.Font.Size = 18;
SlewAngularVelocityZ.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewAngularVelocityZ.XAxis.LabelsFont.Size = 18;
SlewAngularVelocityZ.XAxis.LabelsFormat = "0.00";

SlewAngularVelocityZ.YAxis.Title.Text = 'Angular Rate (deg/s)';
SlewAngularVelocityZ.YAxis.Title.Font.Typeface = "Times New Roman";
SlewAngularVelocityZ.YAxis.Title.Font.Size = 18;
SlewAngularVelocityZ.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewAngularVelocityZ.YAxis.LabelsFont.Size = 18;
SlewAngularVelocityZ.YAxis.LabelsFormat = "0.00";

SlewAngularVelocityZ.Series[0].Label = 'w1';
SlewAngularVelocityZ.Series[0].LineWidth = 3;
SlewAngularVelocityZ.Series[0].LineStyle = 0;
SlewAngularVelocityZ.Series[0].LineColor = ColorTools.Blue;

SlewAngularVelocityZ.Series[1].Label = 'w2';
SlewAngularVelocityZ.Series[1].LineWidth = 3;
SlewAngularVelocityZ.Series[1].LineStyle = 0;
SlewAngularVelocityZ.Series[1].LineColor = ColorTools.Red;

SlewAngularVelocityZ.Series[2].Label = 'w3';
SlewAngularVelocityZ.Series[2].LineWidth = 3;
SlewAngularVelocityZ.Series[2].LineStyle = 0;
SlewAngularVelocityZ.Series[2].LineColor = ColorTools.LawnGreen;

SlewAngularVelocityZ.Legend.Font.Size = 18;

SlewAngularVelocityZ.MaxPoints = 10000;

// Slew Attitude
PlotWindow SlewQuaternionZ({NEIGHBOR.ElapsedTime.ToSeconds, q[0,0], q[1,0],
q[2,0], q[3,0]});

//SlewQuaternionZ.PlotTitle.Text = 'Slew Quaternion vs. Time for Z-Axis';
SlewQuaternionZ.PlotTitle.Text = ' ';
SlewQuaternionZ.PlotTitle.Font.Size = 18;
SlewQuaternionZ.PlotTitle.Visible = 1;

SlewQuaternionZ.PlotSubTitle.Visible = 0;

SlewQuaternionZ.XAxis.Title.Text = 'Elapsed Time (s)';
SlewQuaternionZ.XAxis.Title.Font.Typeface = "Times New Roman";
SlewQuaternionZ.XAxis.Title.Font.Size = 18;
SlewQuaternionZ.XAxis.LabelsFont.Typeface = "Times New Roman";
SlewQuaternionZ.XAxis.LabelsFont.Size = 18;
```

```
SlewQuaternionZ.XAxis.LabelsFormat = "0.00";

SlewQuaternionZ.YAxis.Title.Text = 'Quaternion Element Value';
SlewQuaternionZ.YAxis.Title.Font.Typeface = "Times New Roman";
SlewQuaternionZ.YAxis.Title.Font.Size = 18;
SlewQuaternionZ.YAxis.LabelsFont.Typeface = "Times New Roman";
SlewQuaternionZ.YAxis.LabelsFont.Size = 18;
SlewQuaternionZ.YAxis.LabelsFormat = "0.00";

SlewQuaternionZ.Series[0].Label = 'q1';
SlewQuaternionZ.Series[0].LineWidth = 3;
SlewQuaternionZ.Series[0].LineStyle = 0;
SlewQuaternionZ.Series[0].LineColor = ColorTools.Blue;

SlewQuaternionZ.Series[1].Label = 'q2';
SlewQuaternionZ.Series[1].LineWidth = 3;
SlewQuaternionZ.Series[1].LineStyle = 0;
SlewQuaternionZ.Series[1].LineColor = ColorTools.Red;

SlewQuaternionZ.Series[2].Label = 'q3';
SlewQuaternionZ.Series[2].LineWidth = 3;
SlewQuaternionZ.Series[2].LineStyle = 0;
SlewQuaternionZ.Series[2].LineColor = ColorTools.LawnGreen;

SlewQuaternionZ.Series[3].Label = 'q4';
SlewQuaternionZ.Series[3].LineWidth = 3;
SlewQuaternionZ.Series[3].LineStyle = 0;
SlewQuaternionZ.Series[3].LineColor = ColorTools.Magenta;

SlewQuaternionZ.Legend.Font.Size = 18;

SlewQuaternionZ.MaxPoints = 10000;

// DE-ORBIT PLOTS ------------------------------------------------------------

// Plot Altitude
PlotWindow PlotAlt;
PlotAlt.PlotTitle.Visible = 0;
PlotAlt.PlotSubTitle.Visible = 0;

PlotAlt.YAxis.Title.Text = "Altitude (km)";
PlotAlt.YAxis.Title.Font.Typeface = "Times New Roman";
PlotAlt.YAxis.Title.Font.Size = 18;
PlotAlt.YAxis.LabelsFont.Typeface = "Times New Roman";
PlotAlt.YAxis.LabelsFont.Size = 18;

PlotAlt.XAxis.Title.Text = "Elapsed Time (days)";
PlotAlt.XAxis.Title.Font.Typeface = "Times New Roman";
PlotAlt.XAxis.Title.Font.Size = 18;
PlotAlt.XAxis.LabelsFont.Typeface = "Times New Roman";
PlotAlt.XAxis.LabelsFont.Size = 18;

PlotAlt.Legend.Location = 2;
PlotAlt.Legend.Font.Typeface = "Times New Roman";
PlotAlt.Legend.Font.Size = 18;

PlotScatterSeries AltWithNEIGHBOR;
```

```
AltWithNEIGHBOR.Label = "Trajectory with NEIGHBOR";
AltWithNEIGHBOR.LineColor = ColorTools.Red;
AltWithNEIGHBOR.LineStyle = 0;
AltWithNEIGHBOR.MarkersColor = ColorTools.Red;

PlotScatterSeries AltNoNEIGHBOR;
AltNoNEIGHBOR.Label = "Trajectory without NEIGHBOR";
AltNoNEIGHBOR.LineColor = ColorTools.Blue;
AltNoNEIGHBOR.LineStyle = 0;
AltNoNEIGHBOR.MarkersColor = ColorTools.Blue;

// Data tables for reporting data in FreeFlyer output
DataTableWindow Totals({h0,hf,DeltaV,m0,mf,mBurned,BurnTime,

      InitialEpochText,FinalEpochText,MissionTime});

PlotWindow DeorbitEulerAngles({NEIGHBOR.ElapsedTime.ToSeconds, phi_deg,
theta_deg, psi_deg});

//DeorbitEulerAngles.PlotTitle.Text = 'Deorbit Euler Angles vs. Time';
DeorbitEulerAngles.PlotTitle.Text = ' ';
DeorbitEulerAngles.PlotTitle.Font.Size = 18;
DeorbitEulerAngles.PlotTitle.Visible = 1;

DeorbitEulerAngles.PlotSubTitle.Visible = 0;

DeorbitEulerAngles.XAxis.Title.Text = 'Elapsed Time (s)';
DeorbitEulerAngles.XAxis.Title.Font.Typeface = "Times New Roman";
DeorbitEulerAngles.XAxis.Title.Font.Size = 18;
DeorbitEulerAngles.XAxis.LabelsFont.Typeface = "Times New Roman";
DeorbitEulerAngles.XAxis.LabelsFont.Size = 18;
DeorbitEulerAngles.XAxis.LabelsFormat = "0.00";

DeorbitEulerAngles.YAxis.Title.Text = 'Angular Position (deg)';
DeorbitEulerAngles.YAxis.Title.Font.Typeface = "Times New Roman";
DeorbitEulerAngles.YAxis.Title.Font.Size = 18;
DeorbitEulerAngles.YAxis.LabelsFont.Typeface = "Times New Roman";
DeorbitEulerAngles.YAxis.LabelsFont.Size = 18;
DeorbitEulerAngles.YAxis.LabelsFormat = "0.00";

DeorbitEulerAngles.Series[0].Label = 'phi';
DeorbitEulerAngles.Series[0].LineWidth = 3;
DeorbitEulerAngles.Series[0].LineStyle = 0;
DeorbitEulerAngles.Series[0].LineColor = ColorTools.Blue;

DeorbitEulerAngles.Series[1].Label = 'theta';
DeorbitEulerAngles.Series[1].LineWidth = 3;
DeorbitEulerAngles.Series[1].LineStyle = 0;
DeorbitEulerAngles.Series[1].LineColor = ColorTools.Red;

DeorbitEulerAngles.Series[2].Label = 'psi';
DeorbitEulerAngles.Series[2].LineWidth = 3;
DeorbitEulerAngles.Series[2].LineStyle = 0;
DeorbitEulerAngles.Series[2].LineColor = ColorTools.LawnGreen;

DeorbitEulerAngles.Legend.Font.Size = 18;
```

```
// De-orbit Angular Velocity
PlotWindow DeorbitAngularVelocity({NEIGHBOR.ElapsedTime.ToSeconds,
w_deg[0,0], w_deg[1,0], w_deg[2,0]});

//DeorbitAngularVelocity.PlotTitle.Text = 'De-orbit Angular Velocity vs.
Time';
DeorbitAngularVelocity.PlotTitle.Text = ' ';
DeorbitAngularVelocity.PlotTitle.Font.Size = 18;
DeorbitAngularVelocity.PlotTitle.Visible = 1;

DeorbitAngularVelocity.PlotSubTitle.Visible = 0;

DeorbitAngularVelocity.XAxis.Title.Text = 'Elapsed Time (s)';
DeorbitAngularVelocity.XAxis.Title.Font.Typeface = "Times New Roman";
DeorbitAngularVelocity.XAxis.Title.Font.Size = 18;
DeorbitAngularVelocity.XAxis.LabelsFont.Typeface = "Times New Roman";
DeorbitAngularVelocity.XAxis.LabelsFont.Size = 18;
DeorbitAngularVelocity.XAxis.LabelsFormat = "0.00";

DeorbitAngularVelocity.YAxis.Title.Text = 'Angular Rate (deg/s)';
DeorbitAngularVelocity.YAxis.Title.Font.Typeface = "Times New Roman";
DeorbitAngularVelocity.YAxis.Title.Font.Size = 18;
DeorbitAngularVelocity.YAxis.LabelsFont.Typeface = "Times New Roman";
DeorbitAngularVelocity.YAxis.LabelsFont.Size = 18;
DeorbitAngularVelocity.YAxis.LabelsFormat = "0.00";

DeorbitAngularVelocity.Series[0].Label = 'w1';
DeorbitAngularVelocity.Series[0].LineWidth = 3;
DeorbitAngularVelocity.Series[0].LineStyle = 0;
DeorbitAngularVelocity.Series[0].LineColor = ColorTools.Blue;

DeorbitAngularVelocity.Series[1].Label = 'w2';
DeorbitAngularVelocity.Series[1].LineWidth = 3;
DeorbitAngularVelocity.Series[1].LineStyle = 0;
DeorbitAngularVelocity.Series[1].LineColor = ColorTools.Red;

DeorbitAngularVelocity.Series[2].Label = 'w3';
DeorbitAngularVelocity.Series[2].LineWidth = 3;
DeorbitAngularVelocity.Series[2].LineStyle = 0;
DeorbitAngularVelocity.Series[2].LineColor = ColorTools.LawnGreen;

DeorbitAngularVelocity.Legend.Font.Size = 18;

DeorbitAngularVelocity.MaxPoints = 10000;

// De-orbit Attitude
PlotWindow DeorbitQuaternion({NEIGHBOR.ElapsedTime.ToSeconds, q[0,0], q[1,0],
q[2,0], q[3,0]});

//DeorbitQuaternion.PlotTitle.Text = 'De-orbit Quaternion vs. Time';
DeorbitQuaternion.PlotTitle.Text = ' ';
DeorbitQuaternion.PlotTitle.Font.Size = 18;
DeorbitQuaternion.PlotTitle.Visible = 1;

DeorbitQuaternion.PlotSubTitle.Visible = 0;

DeorbitQuaternion.XAxis.Title.Text = 'Elapsed Time (s)';
```

```
DeorbitQuaternion.XAxis.Title.Font.Typeface = "Times New Roman";
DeorbitQuaternion.XAxis.Title.Font.Size = 18;
DeorbitQuaternion.XAxis.LabelsFont.Typeface = "Times New Roman";
DeorbitQuaternion.XAxis.LabelsFont.Size = 18;
DeorbitQuaternion.XAxis.LabelsFormat = "0.00";

DeorbitQuaternion.YAxis.Title.Text = 'Quaternion Elements';
DeorbitQuaternion.YAxis.Title.Font.Typeface = "Times New Roman";
DeorbitQuaternion.YAxis.Title.Font.Size = 18;
DeorbitQuaternion.YAxis.LabelsFont.Typeface = "Times New Roman";
DeorbitQuaternion.YAxis.LabelsFont.Size = 18;
DeorbitQuaternion.YAxis.LabelsFormat = "0.00";

DeorbitQuaternion.Series[0].Label = 'q1';
DeorbitQuaternion.Series[0].LineWidth = 3;
DeorbitQuaternion.Series[0].LineStyle = 0;
DeorbitQuaternion.Series[0].LineColor = ColorTools.Blue;

DeorbitQuaternion.Series[1].Label = 'q2';
DeorbitQuaternion.Series[1].LineWidth = 3;
DeorbitQuaternion.Series[1].LineStyle = 0;
DeorbitQuaternion.Series[0].LineColor = ColorTools.Red;

DeorbitQuaternion.Series[2].Label = 'q3';
DeorbitQuaternion.Series[2].LineWidth = 3;
DeorbitQuaternion.Series[2].LineStyle = 0;
DeorbitQuaternion.Series[2].LineColor = ColorTools.LawnGreen;

DeorbitQuaternion.Series[3].Label = 'q4';
DeorbitQuaternion.Series[3].LineWidth = 3;
DeorbitQuaternion.Series[3].LineStyle = 0;
DeorbitQuaternion.Series[3].LineColor = ColorTools.Magenta;

DeorbitQuaternion.Legend.Font.Size = 18;

DeorbitQuaternion.MaxPoints = 10000;

// De-orbit X-axis Pulsing
PlotWindow DeorbitThrustX({NEIGHBOR.ElapsedTime.ToSeconds, ThrustFlags[0]});

DeorbitThrustX.PlotTitle.Text = 'De-orbit X-axis Thrust Signal vs. Time';
DeorbitThrustX.PlotTitle.Font.Size = 18;
DeorbitThrustX.PlotSubTitle.Visible = 0;

DeorbitThrustX.XAxis.Title.Text = 'Elapsed Time (s)';
DeorbitThrustX.XAxis.Title.Font.Typeface = "Times New Roman";
DeorbitThrustX.XAxis.Title.Font.Size = 18;
DeorbitThrustX.XAxis.LabelsFont.Typeface = "Times New Roman";
DeorbitThrustX.XAxis.LabelsFont.Size = 18;
DeorbitThrustX.XAxis.LabelsFormat = "0.00";

DeorbitThrustX.YAxis.Title.Text = 'X-Axis Thrust Signal';
DeorbitThrustX.YAxis.Title.Font.Typeface = "Times New Roman";
DeorbitThrustX.YAxis.Title.Font.Size = 18;
DeorbitThrustX.YAxis.LabelsFont.Typeface = "Times New Roman";
DeorbitThrustX.YAxis.LabelsFont.Size = 18;
DeorbitThrustX.YAxis.LabelsSpacing = 1;
```

```
DeorbitThrustX.YAxis.MaximumValue = 1.5;
DeorbitThrustX.YAxis.MinimumValue = -1.5;
DeorbitThrustX.YAxis.LabelsFormat = "0";

DeorbitThrustX.Legend.Visible = 0;

DeorbitThrustX.MaxPoints = 10000;

// De-orbit Y-axis Pulsing
PlotWindow DeorbitThrustY({NEIGHBOR.ElapsedTime.ToSeconds, ThrustFlags[1]});

DeorbitThrustY.PlotTitle.Text = 'De-orbit Y-axis Thrust Signal vs. Time';
DeorbitThrustY.PlotTitle.Font.Size = 18;
DeorbitThrustY.PlotSubTitle.Visible = 0;

DeorbitThrustY.XAxis.Title.Text = 'Elapsed Time (s)';
DeorbitThrustY.XAxis.Title.Font.Typeface = "Times New Roman";
DeorbitThrustY.XAxis.Title.Font.Size = 18;
DeorbitThrustY.XAxis.LabelsFont.Typeface = "Times New Roman";
DeorbitThrustY.XAxis.LabelsFont.Size = 18;
DeorbitThrustY.XAxis.LabelsFormat = "0.00";

DeorbitThrustY.YAxis.Title.Text = 'Y-Axis Thrust Signal';
DeorbitThrustY.YAxis.Title.Font.Typeface = "Times New Roman";
DeorbitThrustY.YAxis.Title.Font.Size = 18;
DeorbitThrustY.YAxis.LabelsFont.Typeface = "Times New Roman";
DeorbitThrustY.YAxis.LabelsFont.Size = 18;
DeorbitThrustY.YAxis.LabelsSpacing = 1;
DeorbitThrustY.YAxis.MaximumValue = 1.5;
DeorbitThrustY.YAxis.MinimumValue = -1.5;
DeorbitThrustY.YAxis.LabelsFormat = "0";

DeorbitThrustY.Legend.Visible = 0;

DeorbitThrustY.MaxPoints = 10000;

// De-orbit Z-axis Pulsing
PlotWindow DeorbitThrustZ({NEIGHBOR.ElapsedTime.ToSeconds, ThrustFlags[2]});

DeorbitThrustZ.PlotTitle.Text = 'De-orbit Z-axis Thrust Signal vs. Time';
DeorbitThrustZ.PlotTitle.Font.Size = 18;
DeorbitThrustZ.PlotSubTitle.Visible = 0;

DeorbitThrustZ.XAxis.Title.Text = 'Elapsed Time (s)';
DeorbitThrustZ.XAxis.Title.Font.Typeface = "Times New Roman";
DeorbitThrustZ.XAxis.Title.Font.Size = 18;
DeorbitThrustZ.XAxis.LabelsFont.Typeface = "Times New Roman";
DeorbitThrustZ.XAxis.LabelsFont.Size = 18;
DeorbitThrustZ.XAxis.LabelsFormat = "0.00";

DeorbitThrustZ.YAxis.Title.Text = 'Z-Axis Thrust Signal';
DeorbitThrustZ.YAxis.Title.Font.Typeface = "Times New Roman";
DeorbitThrustZ.YAxis.Title.Font.Size = 18;
DeorbitThrustZ.YAxis.LabelsFont.Typeface = "Times New Roman";
DeorbitThrustZ.YAxis.LabelsFont.Size = 18;
DeorbitThrustZ.YAxis.LabelsSpacing = 1;
DeorbitThrustZ.YAxis.MaximumValue = 1.5;
```

```
DeorbitThrustZ.YAxis.MinimumValue = -1.5;
DeorbitThrustZ.YAxis.LabelsFormat = "0";

DeorbitThrustZ.Legend.Visible = 0;

DeorbitThrustZ.MaxPoints = 10000;
```

APPENDIX C: ATTITUDE CONTROL AND DE-ORBIT PROCEDURES CODE

```
// Procedure to update attitude dynamics state parameters

Define Procedure StateChange(Matrix I, Matrix q, Matrix w, Matrix w_dot,
Matrix L, Variable dt);

      // Current Quaternion Elements
      Variable q1 = q[0,0]; // first element
      Variable q2 = q[1,0]; // second element
      Variable q3 = q[2,0]; // third element
      Variable q4 = q[3,0]; // fourth element

      // Chi Operator Definition
      Matrix Xi_q = [q4, -q3,  q2;
                      q3,  q4, -q1;
                     -q2,  q1,  q4;
                     -q1, -q2, -q3];

      // Quaternion Time Rate of Change
      Matrix q_dot = 0.5 * Xi_q * w;

      // Euler's Rotational Equation
      w_dot = I.Inverse * (L - w.CrossProduct(I*w)); // [rad/s^2]

      // Integrate State
      q = q + q_dot*dt;
      w = w + w_dot*dt; // [rad/s]

EndProcedure;


// Detumble Control Law

Define Procedure Detumble(Matrix w, Matrix w_d, Matrix w_tol, Matrix w_dot,
Matrix L, Matrix L_thruster, Matrix Kp, Matrix Kd, Array ThrustFlags,
Variable ThrustTime, Variable dt);

      Variable i; // index variable

      // Apply PD feedback
      L = Kp * (w_d - w) - Kd * w_dot; // controller output torque [N-m]

      // Compute control torque for each axis and perform analysis
      For i = 0 to 2;

            // Saturate control output to nominal torque
      If (L[i,0] >= L_thruster[i,0]);

            L[i,0] = L_thruster[i,0];
```

```
      ElseIf (L[i,0] <= -L_thruster[i,0]);

            L[i,0] = -L_thruster[i,0];

      Else;

            L[i,0] = 0;

      End;

            // Apply no control if within allowable tolerance
            If (abs(w[i,0] - w_d[i,0]) <= w_tol[i,0]);

                  L[i,0] = 0;

            End;

            // Track thruster operation time
            If (L[i,0] != 0);

                  ThrustTime += dt;

            End;

            // Thrust Flags for visual indication of thruster usage
            If (L[i,0] == L_thruster[i,0]);

                  ThrustFlags[i] = 1;

            ElseIf (L[i,0] == -L_thruster[i,0]);

                  ThrustFlags[i] = -1;

            Else;

                  ThrustFlags[i] = 0;

            End;

      End;

EndProcedure;



// Slew Control Law

Define Procedure Slew(Matrix q, Matrix q_d, Matrix q_tol, Matrix w, Matrix
w_d, Matrix w_dot, Matrix L, Matrix L_thruster, Matrix Kp, Matrix Kd, Array
ThrustFlags, Variable ThrustTime, Variable dt);

      Variable i; // index variable

      // Vector and Scalar Components of Quaternion and Desired Quaternion
      Matrix q123 = q[0:2,0]; // vector component of quaternion
      Variable q4 = q[3,0]; // scalar component of quaternion
```

```
      Matrix q123_d = q_d[0:2,0]; // vector component of desired quaternion
      Variable q4_d = q_d[3,0]; // scalar component of desired quaternion

      // Calculate Inverse of Desired Quaternion and Components
      Matrix q_d_inv = [-q123_d; q4_d]/q_d.Norm;

      Matrix q123_d_inv = q_d_inv[0:2,0];
      Variable q4_d_inv = q_d_inv[3,0];

      // Calculate Error Quaternion
      Matrix delta_q = [q4_d_inv * q123 + q4 * q123_d_inv -
q123.CrossProduct(q123_d_inv);
q4*q4_d_inv - q123.DotProduct(q123_d_inv)];

      // Apply PD feedback
      L = -Kp * sign(delta_q[3,0]) * delta_q[0:2,0] + Kd * (w_d - w);

      // Compute control output for each thruster pair
      For i = 0 to 2;

            // Saturate control output to nominal torque
      If (L[i,0] >= L_thruster[i,0]);

            L[i,0] = L_thruster[i,0];

      ElseIf (L[i,0] <= -L_thruster[i,0]);

                  L[i,0] = -L_thruster[i,0];

      Else;

            L[i,0] = 0;

      End;

            // Track thruster usage time
            If (L[i,0] != 0);

                  ThrustTime += dt;

            End;

            // Thrust Flags for visual indication of thruster usage
            If (L[i,0] == L_thruster[i,0]);

                  ThrustFlags[i] = 1;

            ElseIf (L[i,0] == -L_thruster[i,0]);

                  ThrustFlags[i] = -1;

            Else;

                  ThrustFlags[i] = 0;

            End;
```

```
      End;

EndProcedure;


// Procedure to convert quaternion to 1-2-3 Euler angles in radians

Define Procedure q_to_Euler(Matrix q, Variable phi, Variable theta, Variable
psi);

      // Parse Current Quaternion
      Variable q1 = q[0,0]; // first element of quaternion
      Variable q2 = q[1,0]; // second element
      Variable q3 = q[2,0]; // third element
      Variable q4 = q[3,0]; // fourth element
      Matrix q123 = q[0:2,0]; // vector component of current quaternion

      Matrix I3 = [1, 0, 0;
                   0, 1, 0;
                   0, 0, 1]; // identity matrix

      Matrix q123_cross = [0, -q3,  q2;
                           q3,   0, -q1;
                          -q2,  q1,  0]; // cross-product matrix of the
quaternion vector component

      // Convert Quaternion to Attitude Matrix
      Matrix A = (q4^2 - q123.Norm^2)*I3 - 2*q4*q123_cross +
2*q123*q123.Transpose;

      // Compute 1-2-3 Sequence Euler Angles
      phi = atan2(-A[2,1],A[2,2]); // first Euler angle, phi

      // Control values for input to determine theta
      If (A[2,0] > 1);

            A[2,0] = 1;

      ElseIf (A[2,0] < -1);

            A[2,0] = -1;

      End;

      theta = asin(A[2,0]); // second Euler angle, theta

      psi = atan2(-A[1,0],A[0,0]); // third Euler angle, psi

EndProcedure;


// Procedure to determine de-orbit attitude tracking target quaternion

Define Procedure TargetQuaternion(Matrix q_d, Matrix w_d, Variable dt);

      // Desired Quaternion Elements
```

```
      Variable q1_d = q_d[0,0];
      Variable q2_d = q_d[1,0];
      Variable q3_d = q_d[2,0];
      Variable q4_d = q_d[3,0];

      // Chi Operator Definition
      Matrix Xi_qd = [q4_d, -q3_d,  q2_d;
                      q3_d,  q4_d, -q1_d;
                     -q2_d,  q1_d,  q4_d;
                     -q1_d, -q2_d, -q3_d];

      // Quaternion Time Rate of Change [Equation 3.20 on page 71 (PDF page
86) in ADCS texbtook]
      Matrix q_d_dot = 0.5 * Xi_qd * w_d;

      // Integrate State
      q_d = q_d + q_d_dot*dt;

EndProcedure;


// De-orbit Control Law

Define Procedure Deorbit(Matrix I, Matrix q, Matrix q_d, Matrix w, Matrix
w_d, Matrix w_d_dot, Matrix L, Matrix L_thruster, Variable k, Matrix G,
Matrix e, Array ThrustFlags, Variable ThrustTime, Variable dt);

      Variable i; // loop index variable
      Variable j; // another loop index variable

      // Vector and Scalar Components of Current and Desired Quaternions
      Matrix q123 = q[0:2,0]; // vector component of current quaternion
      Variable q4 = q[3,0]; // scalar component of current quaternion

      Matrix q123_d = q_d[0:2,0]; // vector component of desired quaternion
      Variable q4_d = q_d[3,0]; // scalar component of desired quaternion

      // Calculate Inverse of Desired Quaternion
      Matrix q_d_inv = [-q123_d; q4_d]/q_d.Norm;

      Matrix q123_d_inv = q_d_inv[0:2,0]; // vector component
      Variable q4_d_inv = q_d_inv[3,0]; // scalar component

      // Calculate Error Quaternion
      Matrix delta_q = [q4_d_inv * q123 + q4 * q123_d_inv -
q123.CrossProduct(q123_d_inv);
                        q4*q4_d_inv - q123.DotProduct(q123_d_inv)];

      // Calculate Sliding Surface Vector
      Matrix s = (w - w_d) + k * sign(delta_q[3,0]) * delta_q[0:2,0];

      // Use saturation function to construct s_bar
      Matrix s_bar = [0; 0; 0];

      For j = 0 to 2;

            If (s[j,0] > e[j,0]);
```

```
                  s_bar[j,0] = 1;

            ElseIf (abs(s[j,0]) <= e[j,0]);

                  s_bar[j,0] = s[j,0] / e[j,0];

            ElseIf (s[j,0] < -e[j,0]);

                  s_bar[j,0] = -1;

            End;

      End;

      // Apply Sliding-mode Control
      L = I * ( (k/2)* ( (abs(delta_q[3,0])) * (w_d - w) - sign(delta_q[3,0])
* delta_q[0:2,0].CrossProduct(w + w_d) ) + w_d_dot - G*s_bar ) +
w.CrossProduct(I*w);

      For i = 0 to 2;

            // Saturate control output to nominal torque
      If (L[i,0] >= L_thruster[i,0]);

            L[i,0] = L_thruster[i,0];

      ElseIf (L[i,0] <= -L_thruster[i,0]);

                  L[i,0] = -L_thruster[i,0];

      Else;

            L[i,0] = 0;

      End;

            // Track thruster usage time
            If (L[i,0] != 0);

                  ThrustTime += dt;

            End;

            // Thrust Flags for visual indication of thruster usage
            If (L[i,0] == L_thruster[i,0]);

                  ThrustFlags[i] = 1;

            ElseIf (L[i,0] == -L_thruster[i,0]);

                  ThrustFlags[i] = -1;

            Else;

                  ThrustFlags[i] = 0;
```

```
            End;

      End;

EndProcedure;


// Procedure to initialize spacecraft time and orbit

Define Procedure InitializeSC(Spacecraft sc, Variable InitialAltitude,
TimeSpan InitialEpoch, String InitialEpochText);

      // Place spacecraft back to initial time
      sc.Epoch = "Jan 01 2023 00:00:00.000".ParseCalendarDate;
      InitialEpoch = sc.Epoch;
      InitialEpochText = sc.EpochText;

      // Reset initial orbital elements and set initial SMA based on specific
initial altitude
      sc.A = Earth.Radius + InitialAltitude; // Semi-major axis [km]
      sc.E = 0; // Eccentricity
      sc.I = 51.6; // Inclination [deg]
      sc.RAAN = 0; // Right ascension of the ascending node [deg]
      sc.W = 0; // Argument of perigee [deg]
      sc.TA = 0; // True anomaly [deg]

EndProcedure;


// Procedure for trajectory propagation following de-orbit maneuver

Define Procedure DeorbitToReEntry(Spacecraft sc, Variable NEIGHBORstate,
Variable numError, Variable m0, Variable mf, Variable mBurned, Variable
DeltaV, TimeSpan InitialEpoch, TimeSpan FinalEpoch, Variable MissionTime,
String FinalEpochText, DataTableWindow Totals, PlotWindow PlotAlt,
PlotScatterSeries AltWithNEIGHBOR, PlotScatterSeries AltNoNEIGHBOR);

      // Avoid error messages to end mission and generate output
      Try sending ErrorCount to numError;

            // Step spacecraft forward in time
            Step sc;

      End;

      // Stop simulation when an error occurs
      If (numError > 0);

            // Determine fuel usage
            mf = sc.Mass;
            mBurned = m0 - mf;

            // Mission Time
            FinalEpoch = sc.Epoch;
            FinalEpochText = sc.EpochText;
            MissionTime = (FinalEpoch.ToDays - InitialEpoch.ToDays)/365;
```

```
            If (NEIGHBORstate == 1);

                  DeltaV = 0;

            End;

            Update Totals;

            ExitProcedure;

      End;

      // Update altitude plots while mission continues
      If(NEIGHBORstate == 0);

            AltWithNEIGHBOR.AddPoints(sc.ElapsedTime, sc.A - Earth.Radius);

      Else;

            AltNoNEIGHBOR.AddPoints(sc.ElapsedTime, sc.A - Earth.Radius);

      End;

      Update PlotAlt;

EndProcedure;


// Procedure for determining circle cross product of two quaternions

Define Procedure CircleCross(Matrix p, Matrix q, Matrix p_circle_cross_q);

      // Scalar components of input quaternions
      Variable p4 = p[3,0];
      Variable q4 = q[3,0];

      // Vector components of input quaternions
      Matrix p123 = p[0:2,0];
      Matrix q123 = q[0:2,0];

      // Circle cross product of input quaternions
      p_circle_cross_q = [p4*q123 + q4*p123 - p123.CrossProduct(q123);
                          p4*q4 - p123.DotProduct(q123)];

EndProcedure;
```

APPENDIX D: DETUMBLE MANEUVER CODE

```
// Mission Initialization
Call InitializeSC(NEIGHBOR, h0, InitialEpoch, InitialEpochText);
t0 = NEIGHBOR.Epoch.ToSeconds; // initial time [s]

// VALIDATION CASE 1 INITIAL QUATERNION
q = [0; 0; 0; 1];

// VALIDATION CASE 2 INITIAL QUATERNION GENERATION
//q1 = UniformNoise(0.55,0.05);
//q2 = UniformNoise(0.35,0.05);
//q3 = UniformNoise(0.15,0.05);
//q4 = (1 - q1^2 - q2^2 - q3^2)^(1/2);
//q = [q1; q2; q3; q4];

// SELECTED VALIDATION CASE 2 QUATERNION VALUES FOR THESIS
//q1 = 0.5597;
//q2 = 0.3484;
//q3 = 0.1653;
//q4 = (1 - q1^2 - q2^2 - q3^2)^(1/2);
//q = [q1; q2; q3; q4];

// Convert Initial Quaternion to Euler Angles
Call q_to_Euler(q, phi, theta, psi);

phi_deg = deg(phi); // phi [deg]
theta_deg = deg(theta); // theta [deg]
psi_deg = deg(psi); // psi [deg]

// Other Initial Conditions
q0 = q; // set initial quaternion

w = rad([10; 10; 10]); // initial angular velocity [rad]
w_deg = deg(w); // initial angular velocity [deg]
w0 = w_deg; // set initial angular velocity [deg]

w_dot = [0; 0; 0]; // angular acceleration [rad/s^2]

// Final Conditions
w_d = rad([0; 0; 0]); // desired angular velocity [rad/s]
w_tol = rad([0.01; 0.01; 0.01]); // angular velocity tolerance [rad/s]

// Gains
Kp = 1*10^2*[1, 0, 0;
             0, 1, 0;
             0, 0, 1];

Kd = 1*10^0*[1, 0, 0;
             0, 1, 0;
             0, 0, 1];

// Set time step for maneuver
```

```
NEIGHBOR.Propagator.StepSize = TimeSpan.FromSeconds(t_step);
dt = NEIGHBOR.Propagator.StepSize.ToSeconds;

// Simulation
While (NEIGHBOR.ElapsedTime < TIMESPAN(20 seconds)); // 1000 --> arbitrarilly
large period of time

      // Update Flag Plots
      Update DetumbleThrustX; Update DetumbleThrustY; Update DetumbleThrustZ;

      // Control
      Call Detumble(w, w_d, w_tol, w_dot, L, L_thruster,
                          Kp, Kd, ThrustFlags, DetumbleThrustTime, dt);

      // State Change
      Call StateChange(I, q, w, w_dot, L, dt);

      // Convert to degrees and integrate Euler angles
      w_deg = deg(w);

      phi = phi + w[0,0] * dt;
      theta = theta + w[1,0] * dt;
      psi = psi + w[2,0] * dt;

      phi_deg = deg(phi); // [deg]
      theta_deg = deg(theta);
      psi_deg = deg(psi);

      // Update Plots
      Update DetumbleEulerAngles; Update DetumbleAngularVelocity;
      Update DetumbleThrustX; Update DetumbleThrustY; Update DetumbleThrustZ;

      // Stop Simulation if final conditions met
      If (w[0,0] - w_d[0,0] <= w_tol[0,0] and
            w[1,0] - w_d[1,0] <= w_tol[1,0] and
            w[2,0] - w_d[2,0] <= w_tol[2,0]);

            tf = NEIGHBOR.Epoch.ToSeconds;

            Break;

      End;

      // Step Spacecraft
      Step NEIGHBOR;

End;

qf = q; // final quaternion
wf = w_deg; // final angular velocity

// Post-maneuver Analysis
Variable mpDetumble = 2*F_MarCO*DetumbleThrustTime/(Isp_MarCO*g0); //
propellant mass
Variable DetumbleManeuverTime = tf - t0; // maneuver time
```

APPENDIX E: SLEW MANEUVER CODE

```
// Maneuver Conditions
q0 = q; // initial quaternion

q_d = I_q; // desired quaternion
q_tol = 0.01*[1; 1; 1; 1]; // q tolerance

w0 = w_deg; // initial angular velocity

w_d = rad([0; 0; 0]); // desired angular velocity [rad/s]
w_tol = rad([0.01; 0.01; 0.01]); // w tolerance [rad/s]

// Gains

// VALIDATION CASE 1
Kp = 1*10^1*[1, 0, 0;
             0, 1, 0;
             0, 0, 1];

Kd = 1.75*10^1*[1, 0, 0;
                0, 1, 0;
                0, 0, 1];

// VALIDATION CASE 2
//Kp = 1*10^1*[1, 0, 0;
//             0, 1, 0;
//             0, 0, 1];
//
//Kd = 2.25*10^1*[1, 0, 0;
//                0, 1, 0;
//                0, 0, 1];

// Re-set thrust flags
ThrustFlags = {0, 0, 0};

// Initial Time
t0 = NEIGHBOR.Epoch.ToSeconds;

// Set time step for maneuver
NEIGHBOR.Propagator.StepSize = TimeSpan.FromSeconds(t_step);
dt = NEIGHBOR.Propagator.StepSize.ToSeconds;

// Simulation
While (NEIGHBOR.ElapsedTime < TIMESPAN(60 seconds));

      // Update Flag Plots
      Update SlewThrustX; Update SlewThrustY; Update SlewThrustZ;

      // Control
      Call Slew(q, q_d, q_tol, w, w_d, w_dot, L, L_thruster,
                   Kp, Kd, ThrustFlags, SlewThrustTime, dt);
```

```
      // State Change
      Call StateChange(I, q, w, w_dot, L, dt);

      // Convert to degrees and integrate Euler angles
      w_deg = deg(w);

      phi = phi + w[0,0] * dt;
      theta = theta + w[1,0] * dt;
      psi = psi + w[2,0] * dt;

      phi_deg = deg(phi) % 360; // [deg]
      theta_deg = deg(theta) % 360;
      psi_deg = deg(psi) % 360;

      // Update All Plots
      Update SlewEulerAngles; Update SlewQuaternion; Update
SlewAngularVelocity;
      Update SlewThrustX; Update SlewThrustY; Update SlewThrustZ;

      // Step Spacecraft
      Step NEIGHBOR;

      // End simulation if desired condition reached
      If (abs(q_d[0,0] - q[0,0]) <= q_tol[0,0] and
            abs(q_d[1,0] - q[1,0]) <= q_tol[1,0] and
            abs(q_d[2,0] - q[2,0]) <= q_tol[2,0] and
            abs(q_d[3,0] - q[3,0]) <= q_tol[3,0] and
            abs(w[0,0] - w_d[0,0]) <= w_tol[0,0] and
            abs(w[1,0] - w_d[1,0]) <= w_tol[1,0] and
            abs(w[2,0] - w_d[2,0]) <= w_tol[2,0]);

                  tf = NEIGHBOR.Epoch.ToSeconds;

                  Break;

      End;

End;

qf = q; // final quaternion
wf = w_deg; // final angular velocity

// Post-maneuver Analysis
Variable mpSlew = 2*F_MarCO*SlewThrustTime/(Isp_MarCO*g0); // propellant mass
Variable SlewManeuverTime = tf - t0; // maneuver time

Report mpSlew, SlewManeuverTime, q0, qf, w0, wf;
```

APPENDIX F: DE-ORBIT MANEUVER CODE

```
// Track optimal burn attitude
Variable Rmaneuver = Earth.Radius + 400; // average orbit radius [km]
Variable Period = 2*Constants.Pi*sqrt(Rmaneuver^3/Earth.Mu); // average orbit
period [s]
Variable TrackingRate = 360/Period; // burn direction tracking rate [deg/s]

// Maneuver Conditions
q_d = q; // desired orientation starts at burn direction

w_d = rad([0; TrackingRate; 0]); // insert tracking rate to angular velocity
[rad/s]

w_d_dot = [0; 0; 0]; // desired angular acceleration [rad/s^2]

// Gains
k = 1; // positive scalar constant

G = [0.1, 0, 0;
     0, 0.1, 0;
       0, 0, 0.1]; // must be positive definite matrix

e = 0.001*[1; 1; 1]; // components of e are positive quantities

// Re-set Thrust Flags
ThrustFlags = {0, 0, 0};

// Propellant Analysis
Variable mpDeorbitControl;

// Plots
PlotAlt.AddSeries(AltWithNEIGHBOR);
PlotAlt.AddSeries(AltNoNEIGHBOR);

For NEIGHBORstate = 1 to 1; // 0 = NEIGHBOR, 1 = no NEIGHBOR

      Call InitializeSC(NEIGHBOR, h0, InitialEpoch, InitialEpochText);

      NEIGHBOR.MassTotal = 12;

      // De-orbit Maneuver
      If (NEIGHBORstate == 0);

            NEIGHBOR.Propagator.StepSize = TIMESPAN(0.5 seconds); //
TimeSpan.FromSeconds(t_step);
            dt = NEIGHBOR.Propagator.StepSize.ToSeconds;

            BurnStart = NEIGHBOR.Epoch.ToMinutes;

            Try;
            WhileManeuvering NEIGHBOR using RetrogradeBurn;
```

```
                  // Update All Plots
                  Update DeorbitEulerAngles; Update DeorbitQuaternion; Update
DeorbitAngularVelocity;
                  Update DeorbitThrustX; Update DeorbitThrustY; Update
DeorbitThrustZ;

                  If (m0 - NEIGHBOR.Mass >= mp);

                        Break;

                  End;

                  // Update Flag Plots
                  Update DeorbitThrustX; Update DeorbitThrustY; Update
DeorbitThrustZ;

                  // Control
                  Call Deorbit(I, q, q_d, w, w_d, w_d_dot, L, L_thruster,
                                    k, G, e, ThrustFlags,
DeorbitControlThrustTime, dt);

                  // State Change
                  Call StateChange(I, q, w, w_dot, L, dt);

                  // Get Target Quaternion
                  Call TargetQuaternion(q_d, w_d, dt);

                  // Convert to degrees and integrate Euler angles
                  w_deg = deg(w);

                  phi = phi + w[0,0] * dt;
                  theta = theta + w[1,0] * dt;
                  psi = psi + w[2,0] * dt;

                  phi_deg = deg(phi) % 360; // [deg]
                  theta_deg = deg(theta) % 360;
                  psi_deg = deg(psi) % 360;

                  // Update All Plots
                  Update DeorbitEulerAngles; Update DeorbitQuaternion; Update
DeorbitAngularVelocity;
                  Update DeorbitThrustX; Update DeorbitThrustY; Update
DeorbitThrustZ;

                  AltWithNEIGHBOR.AddPoints(NEIGHBOR.ElapsedTime, NEIGHBOR.A
- Earth.Radius);
                  Update PlotAlt;

            End;
            End;

            // Post-maneuver Analysis
            mpDeorbitControl =
2*F_MarCO*DeorbitControlThrustTime/(Isp_MarCO*g0); // propellant mass

            BurnEnd = NEIGHBOR.Epoch.ToMinutes;
            BurnTime = BurnEnd - BurnStart; // maneuver time
```

```
            Report mpDeorbitControl, BurnTime, NEIGHBOR.Mass;

      End;

      NEIGHBOR.Propagator.StepSize = TIMESPAN(5 minutes);

      // Deorbit by drag to Re-entry
      While (NEIGHBOR.Radius > Earth.Radius);

            Call DeorbitToReEntry(NEIGHBOR, NEIGHBORstate, numError, m0, mf,
mBurned, DeltaV,
                                                InitialEpoch, FinalEpoch,
MissionTime, FinalEpochText,
                                                Totals, PlotAlt, AltWithNEIGHBOR,
AltNoNEIGHBOR);

            Report NEIGHBOR.Radius, Earth.Radius;

            If (numError > 0);

                  Break;

            End;

      End;

      numError = 0;

End;
```

APPENDIX G: SLEW MANEUVER VALIDATION CASE THREE CODE

```
// Mission Initialization
Call InitializeSC(NEIGHBOR, h0, InitialEpoch, InitialEpochText);

// Initial Conditions
q0 = q; // initial quaternion

w_deg = deg(w); // angular velocity [deg]
w0 = w_deg; // initial angular velocity

w_dot = [0; 0; 0]; // angular acceleration [rad/s^2]

// Final Conditions
q_tol = 0.01*[1; 1; 1; 1]; // quaternion tolerance

w_d = rad([0; 0; 0]); // desired angular velocity [rad/s]
w_tol = rad([0.01; 0.01; 0.01]); // tolerance [rad/s]

// Set Euler Angle Rotations
Variable phi_delta = rad(180);
Variable theta_delta = rad(180);
Variable psi_delta = rad(180);

// Calculate Quaternion Transformations
Matrix ex = [1; 0; 0]; // x-axis
Matrix ey = [0; 1; 0]; // y-axis
Matrix ez = [0; 0; 1]; // z-axis

Matrix qx = [ex*sin(phi_delta/2); // x-axis rotation quaternion
                    cos(phi_delta/2)];

Matrix qy = [ey*sin(theta_delta/2); // y-axis rotation quaternion
                    cos(theta_delta/2)];

Matrix qz = [ez*sin(psi_delta/2); // z-axis rotation quaternion
                    cos(psi_delta/2)];

// Calculate Target Quaternions
Matrix qdx; // x-axis desired quaternion
Matrix qdy; // y-axis desired quaternion
Matrix qdz; // z-axis desired quaternion

Call CircleCross(qx, q0, qdx);
Call CircleCross(qy, qdx, qdy);
Call CircleCross(qz, qdy, qdz);

Matrix q_targets = [qdx, qdy, qdz]; // collect targets

// Gains
```

```
Matrix Kpx = 1*10^1*[1, 0, 0;
                     0, 1, 0;
                     0, 0, 1];

Matrix Kdx = 2*10^1*[1, 0, 0;
                     0, 1, 0;
                     0, 0, 1];

Matrix Kpy = 1*10^0*[1, 0, 0;
                     0, 1, 0;
                     0, 0, 1];

Matrix Kdy = 5*10^0*[1, 0, 0;
                     0, 1, 0;
                     0, 0, 1];

Matrix Kpz = 1*10^1*[1, 0, 0;
                     0, 1, 0;
                     0, 0, 1];

Matrix Kdz = 3*10^1*[1, 0, 0;
                     0, 1, 0;
                     0, 0, 1];

Matrix KpMatrix = [Kpx, Kpy, Kpz]; // collect proportional gains

Matrix KdMatrix = [Kdx, Kdy, Kdz]; // collect derivative gains

// Set time step for maneuver
NEIGHBOR.Propagator.StepSize = TimeSpan.FromSeconds(t_step);
dt = NEIGHBOR.Propagator.StepSize.ToSeconds;

Variable SlewManeuverTime;
Array SlewManeuverTimeArray = {0, 0, 0};

// Simulation
Variable n;

For n = 0 to 2;

      t0 = NEIGHBOR.Epoch.ToSeconds;

      q0 = q;
      w_deg = deg(w);

      phi_deg = deg(phi);
      theta_deg = deg(theta);
      psi_deg = deg(psi);

      q_d = q_targets[0:3,n];

      Kp = KpMatrix[0:2,3*n:(3*n+2)];
      Kd = KdMatrix[0:2,3*n:(3*n+2)];

      // Simulation
      While (NEIGHBOR.ElapsedTime < TIMESPAN(1500 seconds));
```

```
            // Update Flag Plots
            //Update SlewThrustX; Update SlewThrustY; Update SlewThrustZ;

            // Control
            Call Slew(q, q_d, q_tol, w, w_d, w_dot, L, L_thruster,
                        Kp, Kd, ThrustFlags, SlewThrustTime, dt);

            // State Change
            Call StateChange(I, q, w, w_dot, L, dt);

            // Convert to degrees and integrate Euler angles
            w_deg = deg(w);

            phi = phi + w[0,0] * dt;
            theta = theta + w[1,0] * dt;
            psi = psi + w[2,0] * dt;

            phi_deg = deg(phi) % 360; // [deg]
            theta_deg = deg(theta) % 360;
            psi_deg = deg(psi) % 360;

            // Update All Plots
            If (n == 0);

                  Update SlewEulerAnglesX; Update SlewQuaternionX; Update
SlewAngularVelocityX;
                  //Update SlewThrustX; Update SlewThrustY; Update
SlewThrustZ;

            ElseIf (n == 1);

                  Update SlewEulerAnglesY; Update SlewQuaternionY; Update
SlewAngularVelocityY;
                  //Update SlewThrustX; Update SlewThrustY; Update
SlewThrustZ;

            ElseIf (n == 2);

                  Update SlewEulerAnglesZ; Update SlewQuaternionZ; Update
SlewAngularVelocityZ;
                  //Update SlewThrustX; Update SlewThrustY; Update
SlewThrustZ;

            End;

            // Step Spacecraft
            Step NEIGHBOR;

            // Target Condition Check
            If ((abs(q_d[0,0] - q[0,0]) <= q_tol[0,0] or abs(-q_d[0,0] -
q[0,0]) <= q_tol[0,0]) and
                  (abs(q_d[1,0] - q[1,0]) <= q_tol[1,0] or abs(-q_d[1,0] -
q[1,0]) <= q_tol[1,0]) and
                  (abs(q_d[2,0] - q[2,0]) <= q_tol[2,0] or abs(-q_d[2,0] -
q[2,0]) <= q_tol[2,0]) and
                  (abs(q_d[3,0] - q[3,0]) <= q_tol[3,0] or abs(-q_d[3,0] -
q[3,0]) <= q_tol[3,0]) and
```

```
                    abs(w[0,0] - w_d[0,0]) <= w_tol[0,0] and
                    abs(w[1,0] - w_d[1,0]) <= w_tol[1,0] and
                    abs(w[2,0] - w_d[2,0]) <= w_tol[2,0]);

                    tf = NEIGHBOR.Epoch.ToSeconds; // final time
                    SlewManeuverTime = tf - t0; // maneuver time
                    SlewManeuverTimeArray[n] = SlewManeuverTime;

                    qf = q; // final quaternion
                    w_deg = deg(w); // final angular velocity

                    Break;

            End;

      End;

End;

// Post-maneuver Analysis
Variable mpSlew = 2*F_MarCO*SlewThrustTime/(Isp_MarCO*g0); // propellant mass
Variable TotalSlewManeuverTime = SlewManeuverTimeArray.Sum; // total maneuver
time
```

APPENDIX H: TANK SIZING CODE

```
// Ideal Gas Law
Variable mp = 0.587; // required propellant mass [kg]
Variable MW = 0.15204; // molecular weight [kg/mol]
Variable n = mp/MW; // moles of propellant [mol]

Variable T = 273.15 + 20; // storage temperature [K]
Variable P = 25e6; // storage pressure [Pa]
Variable R = 8.314; // ideal gas constant [J/K-mol]

Variable Vm3 = n*R*T/P; // inner storage volume [m^3]

// Tank Radius
Variable Vsi = Vm3*1e6; // inner storage volume [cm^3]
Variable rsi = (3/(4*Constants.Pi)*Vsi)^(1/3); // storage tank radius [cm]

// Tank Wall Thickness
Variable sigma_al = 276e6; // yield strength of aluminum [Pa]
Variable SF = 2; // safety factor

Variable tw = P*rsi/(2*sigma_al) * SF; // wall thickness [cm]

// Outer Tank Volume
Variable rso = rsi + tw; // outer tank radius [cm]
Variable Vso = 4*Constants.Pi/3 * rso^3; // outer tank volume [cm^3]

// Tank Wall Mass
Variable V_tank = Vso - Vsi; // tank wall material volume
Variable rho_tank = 2.7; // density of aluminum [g/cm^3]
Variable m_tank = V_tank * rho_tank; // [g]

// Tank and Propellant Mass
Variable m_p_and_tank = mp * 1000 + m_tank; // mass of propellant and tank
[g]

//Report Vi, ri, tw;
//Report ro, Vo, V_tank, m_tank, m_p_and_tank;


// Feed Line Sizing
Variable Dp = 0.635; // feed line outer diameter [cm]
Variable sigma_steel = 205e6; // yield strength of steel [Pa]
Variable t_feed_line = P*Dp/(2*sigma_steel) * SF; // feed line wall thickness
[cm]

Variable rci = Dp/2 - t_feed_line; // feed line inner radius [cm]
Variable rco = Dp/2; // feed line outer radius [cm]

//Report rci, rco, t_feed_line;

Array lc = {sqrt(10^2 + 5^2), sqrt(10^2 + 5^2), sqrt(10^2 + 5^2), sqrt(10^2 +
5^2),
```

```
                    10, 5, 10, 5,
                    sqrt(10^2 + 5^2 + 5^2), sqrt(10^2 + 5^2 + 5^2), sqrt(10^2
+ 5^2 + 5^2), sqrt(10^2 + 5^2 + 5^2),
                    10 + 10, 5 + 10, 10 + 10, 5 + 10}; // feed line lengths
[cm]

Variable rho_steel = 8; // feed line density due to 304 stainless steel
[g/cm^3]

Array Vci = {};
Array Vco = {};

Variable i = 0;

For i = 0 to 15;

      Vci.PushBack(Constants.Pi * rci^2 * lc[i]);
      Vco.PushBack(Constants.Pi * rco^2 * lc[i]);

End;

Array Vc_material = Vco - Vci; // outer volume of feed lines

Array mc_material = Vc_material * rho_steel; // mass of feed lines

Variable mc_material_total = mc_material.Sum; // total feed line mass [g]
Variable Vco_total = Vco.Sum;
```

APPENDIX I: POWER AND DATA ANALYSIS CODE

```
Define Procedure CONTACT(Spacecraft sc, GroundStation gs, Variable
GS_contact_Sat);

      VisibilityCalculator VisCalc;
      VisCalc.AddSegment('GS see SC');
      VisCalc.Segments[0].SetObserver(gs);
      VisCalc.Segments[0].SetTarget(sc);

      GS_contact_Sat = VisCalc.Segments[0].Visibility(sc.Epoch);

EndProcedure;



// OBJECT DECLARATION ------------------------------------------------------

// Data
Variable DataCollected;
Variable DataStorage_GigaByte;
Variable DataStorageAvailable_GigaByte;

Variable CollectionTime;
Variable CollectionPeriod;
Variable DataFlag = 0;

// Power
Variable EnergyUsed;
Variable MaxEnergyStorage;

// Contact
Variable Contact;
Variable PreviousContact;
Variable ContactTime;

// Day Tracker
Variable n;

// Time
Variable TimeStepSec;
Variable TimeStepHrs;
Variable t0;
Variable tf;
Variable BurnTime;
Variable PostManeuverPeriod;

TimeSpan InitialEpoch = NEIGHBOR.Epoch;
Variable MissionStart = NEIGHBOR.Epoch.ToDays;
Variable CurrentTime;
```

```
// INPUTS ----------------------------------------------------------------

// Data
Variable DataStorageAvailable = 256e+3; // 256 kilobytes flash storage
[bytes]
Variable DataStorage = DataStorageAvailable; // Initialize
Variable UHFrate = 9600/8; // downlink data rate in from 9600 baud [bytes/s]
Variable Bus_data = 29.503; // data collection rate [bytes/s]

// Power (P = I * V)
MaxEnergyStorage = 3*3.6; // [Ah]*[V] = [Wh]

Variable Propulsion_pwr = 0.5;

Variable HorizonSensor_pwr = 0.15;
Variable SunSensor_pwr = 0.115;
Variable Gyroscope_pwr = 0.018;
Variable GPS_pwr = 1.366; // receiver and antenna
Variable Sensor_pwr = HorizonSensor_pwr + SunSensor_pwr + Gyroscope_pwr +
GPS_pwr;

Variable Computer_pwr = 84e-6 * 80 * 3.3; // runs at 41 mircoAmps per MHz and
up to 120 MHz with 3.3 V power supply
Variable Computer_idle_pwr = 28e-9 * 3.3; // standby mode is 28 nA [W]

Variable Radio_Rx = 55e-3 * 3.3; // standard receiver current of 55 mA and
3.3 V supply voltage [W]
Variable Radio_Tx = 800e-3 * 3.3; // standard transmitter current of 800 mA
and 3.3 supply voltage [W]
Variable Radio_pwr = Radio_Rx + Radio_Tx; // total radio power requirement

// Contact
Variable Radio_flag = 0; // for visualizing when radio is on and adding power
usage

// Maneuver
Variable m0 = NEIGHBOR.Mass;
Variable mp = 0.564; // propellant mass for maneuver from 420 km altitude

// Detumble and Slew Times
Variable SlewTime = 166.26; // [s]
Variable DetumbleTime = 9.97; // [s]

// Orbit

// Set arbitrary altitude above scenario's initial altitude to simulate idle
mode, grace period [km]
Variable ArbitraryAltitude = 500;

Variable InitialAltitude = 420; // scenario initial altitude [km]

// Idle Mode and Grace Period Mode ---------------------------------------

NEIGHBOR.A = Earth.Radius + ArbitraryAltitude; // initialize orbit
EnergyStorage = MaxEnergyStorage; // initialize battery
```

```
n = 1; // initialize day counter for grace period

While (NEIGHBOR.ElapsedTime < (TIMESPAN(1855 days))); // Idle Mode of 5 years
and Grace Period of 30 days

      NEIGHBOR.Propagator.StepSize = TIMESPAN(15 minutes);
      TimeStepHrs = NEIGHBOR.Propagator.StepSize.ToHours;
      TimeStepSec = NEIGHBOR.Propagator.StepSize.ToSeconds;

      // Power Usage
      If (n <= NEIGHBOR.ElapsedTime.ToDays);

            NEIGHBOR.Propagator.StepSize = TIMESPAN(5 seconds);
            TimeStepHrs = NEIGHBOR.Propagator.StepSize.ToHours;
            TimeStepSec = NEIGHBOR.Propagator.StepSize.ToSeconds;

            n += 1;
            PowerUsed = Computer_pwr;

      ElseIf(NEIGHBOR.ElapsedTime == TIMESPAN(0 days));

            NEIGHBOR.Propagator.StepSize = TIMESPAN(5 seconds);
            TimeStepHrs = NEIGHBOR.Propagator.StepSize.ToHours;
            TimeStepSec = NEIGHBOR.Propagator.StepSize.ToSeconds;

            PowerUsed = Computer_pwr;

      Else;

            PowerUsed = Computer_idle_pwr;

      End;

      // Update Battery
      EnergyUsed = PowerUsed * TimeStepHrs ; // [Wh]
      EnergyStorage = EnergyStorage - EnergyUsed;

      // Correct energy storage if it falls to zero
      If (EnergyStorage < 0);
            EnergyStorage = 0;
      End;

      // Enforce stored maximum energy storage
      If (EnergyStorage >= MaxEnergyStorage);
            EnergyStorage = MaxEnergyStorage;
      End;

      // Energy Profile
      EnergyStoragePercent = EnergyStorage/MaxEnergyStorage*100;

      // Data Profile
      Data_Storage = DataStorage/1000000; // [MB]
      Data_Storage_Available = DataStorageAvailable/1000000; // [MB]

      // Update Plots
      Update EnergyPlot;
      Update DataPlot;
```

```
      // Step Spacecraft
      Step NEIGHBOR;

End;

// Final Confirmation from Ground ------------------------------------------
NEIGHBOR.Propagator.StepSize = TIMESPAN(1 seconds);
TimeStepHrs = NEIGHBOR.Propagator.StepSize.ToHours;
TimeStepSec = NEIGHBOR.Propagator.StepSize.ToSeconds;

PreviousContact = 0; // Detect contact with ground for signal to de-orbit

While (NEIGHBOR.ElapsedTime < TIMESPAN(1000 days)); // Arbitrary amount of
time set until contact established

      Call CONTACT(NEIGHBOR, UIUC, Contact);

      // Proceed to next phase if no longer in contact with ground
      If (PreviousContact == 1 and Contact == 0);
            Break;
      End;

      // Check Contact and Power Usage
      If (Contact == 1);
            PowerUsed = Computer_pwr + Radio_pwr;
      Else;
            PowerUsed = Computer_pwr;
      End;

      // Update Battery
      EnergyUsed = PowerUsed * TimeStepHrs ; // [Wh]
      EnergyStorage = EnergyStorage - EnergyUsed;

      // Correct energy storage if it falls to zero
      If (EnergyStorage < 0);
            EnergyStorage = 0;
      End;

      // Enforce stored maximum energy storage
      If (EnergyStorage >= MaxEnergyStorage);
            EnergyStorage = MaxEnergyStorage;
      End;

      // Energy Profile
      EnergyStoragePercent = EnergyStorage/MaxEnergyStorage*100;

      // Data Profile
      Data_Storage = DataStorage/1000000; // [MB]
      Data_Storage_Available = DataStorageAvailable/1000000; // [MB]

      // Update Plots
      Update EnergyPlot;
      Update DataPlot;

      // Step Spacecraft
      PreviousContact = Contact;
```

```
      Step NEIGHBOR;

End;

// De-orbit Mode ------------------------------------------------------------

// Attitude Determination
NEIGHBOR.Propagator.StepSize = TIMESPAN(1 seconds);
TimeStepHrs = NEIGHBOR.Propagator.StepSize.ToHours;
TimeStepSec = NEIGHBOR.Propagator.StepSize.ToSeconds;

While(NEIGHBOR.ElapsedTime < TIMESPAN(10 minutes));

      // Power Usage
      PowerUsed = Computer_pwr + Sensor_pwr;

      // Update Battery
      EnergyUsed = PowerUsed * TimeStepHrs; // [Wh]
      EnergyStorage = EnergyStorage - EnergyUsed;

      // Correct energy storage if it falls below zero
      If (EnergyStorage < 0);
            EnergyStorage = 0;
      End;

      // Enforce stored maximum energy storage
      If (EnergyStorage >= MaxEnergyStorage);
            EnergyStorage = MaxEnergyStorage;
      End;

      // Energy Profile
      EnergyStoragePercent = EnergyStorage/MaxEnergyStorage*100;

      // Data Collection
      DataCollected = Bus_data*TimeStepSec; // [bytes]
      DataStorage = DataStorage - DataCollected;

      // Enforce minimum available data storage
      If (DataStorage <= 0);
            DataStorage = 0;
      End;

      // Enforce maximum available data storage
      If (DataStorage >= DataStorageAvailable);
            DataStorage = DataStorageAvailable;
      End;

      // Data Profile
      Data_Storage = DataStorage/1000000; // [MB]
      Data_Storage_Available = DataStorageAvailable/1000000; // [MB]

      // Update Plots
      Update EnergyPlot;
      Update DataPlot;

      // Step Spacecraft
      Step NEIGHBOR;
```

```
End;


// Control Maneuvers
NEIGHBOR.Propagator.StepSize = TIMESPAN(1 seconds);
TimeStepHrs = NEIGHBOR.Propagator.StepSize.ToHours;
TimeStepSec = NEIGHBOR.Propagator.StepSize.ToSeconds;

While(NEIGHBOR.ElapsedTime < TimeSpan.FromSeconds(DetumbleTime + SlewTime));

      // Power Usage
      PowerUsed = Computer_pwr + Sensor_pwr + Propulsion_pwr;

      // Update Battery
      EnergyUsed = PowerUsed * TimeStepHrs; // [Wh]
      EnergyStorage = EnergyStorage - EnergyUsed;

      // Correct energy storage if it falls below zero
      If (EnergyStorage < 0);
            EnergyStorage = 0;
      End;

      // Enforce stored maximum energy storage
      If (EnergyStorage >= MaxEnergyStorage);
            EnergyStorage = MaxEnergyStorage;
      End;

      // Energy Profile
      EnergyStoragePercent = EnergyStorage/MaxEnergyStorage*100;

      // Data Collection
      DataCollected = Bus_data*TimeStepSec; // Bytes, No Compress
      DataStorage = DataStorage - DataCollected;

      // Enforce minimum available data storage
      If (DataStorage <= 0);
            DataStorage = 0;
      End;

      // Enforce maximum available data storage
      If (DataStorage >= DataStorageAvailable);
            DataStorage = DataStorageAvailable;
      End;

      // Data Profile
      Data_Storage = DataStorage/1000000; // Mind the underscores [MB]
      Data_Storage_Available = DataStorageAvailable/1000000; // [MB]

      // Update Plots
      Update EnergyPlot;
      Update DataPlot;

      // Step Spacecraft
      Step NEIGHBOR;

End;
```

```
// De-orbit Maneuver
NEIGHBOR.A = Earth.Radius + InitialAltitude; // set to scenario altitude

NEIGHBOR.Propagator.StepSize = TIMESPAN(1 seconds);
TimeStepHrs = NEIGHBOR.Propagator.StepSize.ToHours;
TimeStepSec = NEIGHBOR.Propagator.StepSize.ToSeconds;

WhileManeuvering NEIGHBOR using RetrogradeBurn;

      If (m0 - NEIGHBOR.Mass > mp);

            Break; // Proceed once propellant mass necessary for maneuver is
used

      End;

      // Power Usage
      PowerUsed = Propulsion_pwr + Sensor_pwr + Computer_pwr;

      // Update Battery
      EnergyUsed = PowerUsed * TimeStepHrs; // [Wh]
      EnergyStorage = EnergyStorage - EnergyUsed;

      // Correct energy storage if it falls below zero
      If (EnergyStorage < 0);
            EnergyStorage = 0;
      End;

      // Enforce stored maximum energy storage
      If (EnergyStorage >= MaxEnergyStorage);
            EnergyStorage = MaxEnergyStorage;
      End;

      EnergyStoragePercent = EnergyStorage/MaxEnergyStorage*100;

      // DATA PROFILE
      DataCollected = Bus_data*TimeStepSec; // Bytes, No Compress
      DataStorage = DataStorage - DataCollected;

      // Enforce minimum available data storage
      If (DataStorage <= 0);
            DataStorage = 0;
      End;

      // Enforce maximum available data storage
      If (DataStorage >= DataStorageAvailable);
            DataStorage = DataStorageAvailable;
      End;

      Data_Storage = DataStorage/1000000; // Mind the underscores [MB]
      Data_Storage_Available = DataStorageAvailable/1000000; // [MB]

      // Update Plots
      Update EnergyPlot;
      Update DataPlot;
```

```
End;

Data_Storage -= 8; // collect maneuver start, burn time as data

// Quiescent Mode ------------------------------------------------------------
NEIGHBOR.Propagator.StepSize = TIMESPAN(1 seconds); // 10 seconds
TimeStepHrs = NEIGHBOR.Propagator.StepSize.ToHours;
TimeStepSec = NEIGHBOR.Propagator.StepSize.ToSeconds;

CurrentTime = NEIGHBOR.Epoch.ToDays;

CollectionTime = 0;

While (EnergyStorage > 0 and NEIGHBOR.Radius > Earth.Radius);

      Call CONTACT(NEIGHBOR, UIUC, Contact);

      // Check Contact and Power Usage
      If (Contact == 1);
            Radio_flag = 1;
            PowerUsed = Computer_pwr + Radio_pwr + GPS_pwr;
      Else;
            Radio_flag = 0;
            PowerUsed = Computer_idle_pwr;
      End;

      // Update Battery
      EnergyUsed = PowerUsed * TimeStepHrs;
      EnergyStorage = EnergyStorage - EnergyUsed;

      // Correct energy storage if it falls to zero
      If (EnergyStorage < 0);
            EnergyStorage = 0;
      End;

      // Enforce stored maximum energy storage
      If (EnergyStorage >= MaxEnergyStorage);
            EnergyStorage = MaxEnergyStorage;
      End;

      // Energy Profile
      EnergyStoragePercent = EnergyStorage/MaxEnergyStorage*100;

      // Data Downlink
      DataStorage = DataStorage + Radio_flag*UHFrate*TimeStepSec;

      // Enforce minimum available data storage
      If (DataStorage <= 0);
            DataStorage = 0;
      End;

      // Enforce maximum available data storage
      If (DataStorage >= DataStorageAvailable);
            DataStorage = DataStorageAvailable;
      End;
```

```
        // Data Profile
        Data_Storage = DataStorage/1000000; // Mind the underscores [MB]
        Data_Storage_Available = DataStorageAvailable/1000000; // [MB]

        // Update Plots
        Update EnergyPlot;
        Update DataPlot;

        // Step Spacecraft
        Try;

              Step NEIGHBOR;

        End;

End;
```