©Copyright 2020
Devan Tormey

Guidance Navigation & Control System Architecture for the SOCi CubeSat

Satellite for **O**ptimal **C**ontrol and **i**maging

Devan Tormey

A thesis
submitted in partial fulfillment of the
requirements for the degree for

Master of Science in Aeronautics & Astronautics

University of Washington

2020

Reading Committee:

Mehran Mesbahi, Chair

Behçet Açıkmeşe

Kristi Morgansen

Program Authorized to Offer Degree:
Astronautical and Aeronautical Engineering

University of Washington

Abstract

Guidance Navigation & Control System Architecture for the SOCi CubeSat
Satellite for **O**ptimal **C**ontrol and **i**maging

Devan Tormey

Chair of the Supervisory Committee:
Graduate Advising Comittee Mehran Mesbahi
Aeronautics & Astronautics

This thesis describes the design testing and implementation of a Guidance Navigation and Control system for the 2U CubeSat, SOCi. This thesis is intended to cover the full scope of the design process behind the satellite as well as introduce some of the novel concepts used.

ACKNOWLEDGEMENTS

Thank you to my family for supporting me through all of this and for being my guides.

Thank you to my friends for putting up with me and for making me laugh. Especially Eliza, for proof reading all of this.

Thank you Mehran, Behçet, and Kristi being my committee and my teachers!

Big thanks to the whole AACT Team, Especially my Co-Lead Cole!

And thank you to Taylor Reynolds for being one of the best teachers I've ever had, I wouldn't be here without you!

TABLE OF CONTENTS

Page

List of Figures iii

Chapter 1: Introduction 1
- 1.1 The Mission Concept: SOCi 2
- 1.2 Mission Requirements 3
- 1.3 The Payload 8

Chapter 2: Mathematical Modeling 9
- 2.1 Kinematic and Dynamic Model 10
- 2.2 Environmental Modeling 14
- 2.3 Controller Concept 26

Chapter 3: Hardware 46
- 3.1 Sensor Models 46
- 3.2 Actuator Models 53

Chapter 4: Software 60
- 4.1 Estimation 60
- 4.2 Operating Modes 68

Chapter 5: Implementation 80
- 5.1 Autocoding Pipeline 80

Chapter 6: Closing Remarks 91

Bibliography 94

LIST OF FIGURES

Figure Number Page

1.1 A render of SOCi using the current CAD model 1
1.2 Pointing Accuracy Requirement 5
1.3 Plot showing point errorl. 6
1.4 Error split between pointing and estimation. 7

2.1 A visual representation of the Code 9
2.2 This block performs the propogation of our attitude quaternion. . . . 12
2.3 This block performs the propogation of our Rigid Body Dynamics. . . 13
2.4 This block performs the propagation of orbital parameters r (radius) and v (orbital velocity). 14
2.5 Orbit around the earth 23
2.6 Orbital radius over time. 24
2.7 Total disturbance torques over time. 25
2.8 PD controller behavior. 35
2.9 PD vs SlewRate Controllers. 40
2.10 Speed difference PD and Slew Rate control. 41
2.11 Block diagram showing interaction with SOAR 42
2.12 Inputs/outputs of SOAR 43

3.1 Block diagram of sensor library 47
3.2 Image of sun sensor 48
3.3 Conversion from sensor measurements to angles 49
3.4 Going from angles to sun vector. 50
3.5 This is the SimuLink block diagram for the Magnetometer simulation library. 52

3.6 This is the SimuLink block diagram for the gyroscope simulation block. 53
3.7 This diagram depicts the copper trace that makes up the air-core magnetorquer board. 55
3.8 Magnetorquer library block. 56
3.9 This library only models one wheel. We repeat this four times for our RWA. 57

4.1 MEKF library block inputs/outputs 67
4.2 MEKF library expanded. 68
4.3 Power Mode diagram 69
4.4 Operating modes diagram. 70
4.5 Operating mode component diagram. 71
4.6 Operating modes libary block 72
4.7 Target generation library block. 74
4.8 Showing the global operating modes. Flow Chart. 75
4.9 Commission Phase diagram. 76
4.10 Mission Phase diagram. 77
4.11 Imaging Phase diagram. 78
4.12 Experimental Phase diagram. 79

5.1 SOCi testing flowchart. 81
5.2 Autocoding Pipeline flowchart. 82
5.3 This example folder shows the template for the folder layout. 84
5.4 In this image of an example input text file, the last line has been cropped off. 87
5.5 AACT Autocoder flowchart/explanation. 88
5.6 AACT autocoder example output. 89

6.1 version 1 of the FlatSat test plan 93

Common Notation

Acronym	Name
GNC	Guidance, Navigation, Control
CDH	Command and Data Handling subsystem
ECI	Earth-Centered Inertial
FSW	Flight Software
MEKF	Multiplicative Extended Kalman Filter
RWA	Reaction Wheel Assembly
SOAR	SOC-i's Optimal Attitude Reorientation (payload)
TLE	Two Line Element
h	Angular momentum
J	Inertia Matrix
q	Attitude quaternion
r	Radial coordinate in ECI
v	Orbital velocity in ECI
τ	A torque
$C_{B\leftarrow A}$	Rotation from frame A to B
e	Sun vector body frame [1]
b	Magnetic field body frame
E	Capitol version of vector implies inertial
$\hat{x}$	Represents an estimate of a vector
$\tilde{x}$	Represents a measurement

[1]The vector symbol is dropped for simplicity. Included when context is necessary.

Chapter 1

INTRODUCTION

Figure 1.1: A render of SOCi using the current CAD model. Given for use with permission by Arnela Grebovic

Since their inception in 1999, CubeSats have quickly become one of the most powerful tools universities have for the testing and development of new space technologies. The simplicity of the CubeSat design allows for the rapid development of a space-ready platform with which a wide variety of experiments can be carried out. Their relatively low power consumption and light weight allows for the use of mechanically simple actuators. This in turn allows us to test out novel methods for

the control and guidance of these machines. This thesis will focus on the design of the Guidance Navigation and Control (GNC) system for a 2U CubeSat designed at the University of Washington as part of the Aerospace and Aeronautics CubeSat Team. Specifically, it will focus mainly on my contributions to the project while also providing a fairly in depth look at the overall GNC system. While the main goal is to express our novel contributions, this thesis should also allow any new member of the team to dive into the code and pick up where we left off as well as allow future projects to have a running start when it comes to GNC.

1.1 The Mission Concept: SOCi

The mission, in the broadest scope, is to point a camera at Seattle from space and take a picture. Now obviously this itself isn't novel but as with many things in engineering, it's not what you do but how you do it. SOCi is the Satellite for Optimal Control and Imaging. Our goal is to use this simple pointing objective as a means to test out a novel guidance system, specifically one that uses state-of-the-art convex optimization techniques to compute the trajectory for an attitude maneuver. Furthermore we wish to build out a fully autonomous system structure that would need only limited interaction with the ground station to perform specific tasks. This mission has been designed to last six months in space, after which it will continue to operate until orbit-decay causes the spacecraft to de-orbit. From a GNC perspective there are two components: the primary GNC system and the payload. The primary GNC system will be a robust design that will perform the necessary duties for the mission to succeed. The second system, the payload, is a convex optimization based guidance system that will work in tandem with the primary GNC system to allow for power optimal point to point reorientation maneuvers. Of course, in addition to these systems, we will have a fully functional CubeSat. This requires a Power

System, a Communication System, a Command And Data Handling System and a Structural Support System.

1.2 Mission Requirements

One of the first bodies of work associated with this project was the design of the mission requirements. These requirements were then verified with a panel of industry experts and professors here at the University of Washington. Further information on this review can be found within internal documents for each subsystem team. In this thesis we will go over the requirements specific to the GNC System.

The GNC system shall maintain functionality for six months. - As stated earlier this will be a six month mission so any subsequent requirement must be met by the mission length.

The GNC system shall be designed to manage the angular momentum of the spacecraft. Specifically, the system should be able to stabilize the angular momentum of the spacecraft about any arbitrary inertial position. In order to do this the system must be able to absorb at least 10mNms of angular momentum from the chassis. This was found by looking at the torque generated by the disturbances and balancing that with how often we would want to desaturate the wheels given our conops. The system must also be able to maintain the set speed of the reaction wheels. This is reference to the fact that a reaction wheel works by setting the speed to a constant RPM instead of an instantaneous burst.

The GNC system shall provide CDH with the Specified telemetry at the frequency and in the format specified in SOCI-SYS-004 The minimum

telemetry needed by the Command And Data Handling team will be,

- A full state vector of the spacecraft (position, velocity, attitude, angular rates)
- The GNC operating mode
- The target attitude and an indication that the current pointing objective is being met.
- Indication that the spacecraft is above the horizon of the ground station
- All measured actuator states and commands sent to them
- All sensor measurements

All of these serve not only other necessary function within the operation of the spacecraft but also allow us to diagnose any potential issues on board.

The GNC system shall be able to point the spacecraft bus with an accuracy of 7 degrees boresight (3-sigma) and 7 degrees roll-angle This requirement is dictated by the abilities of our camera. Our camera has a 28° half-angle view cone, Figure 1.2 depicts how this information can be used to get an idea of how much of Earth can be seen at any one time.

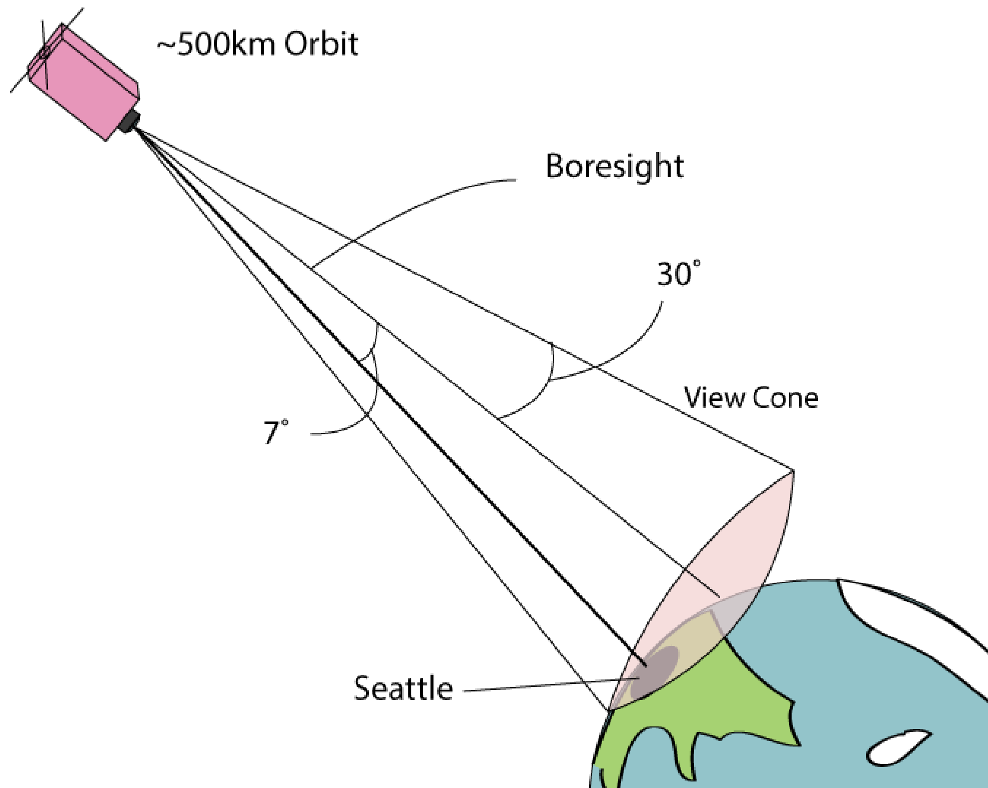

Figure 1.2: Determining the pointing accuracy requirement by studying the cameras "view cone".

By projecting this cone down onto the surface of earth from an altitude of 500 Km this creates a radius of,

$$500 * \sin(30) = 250Km \tag{1.1}$$

If we consider Seattle to be around 100 Km will begin to leave the field of view of the camera once the angle between it and the bore-sight of the camera exceeds 10 degrees.

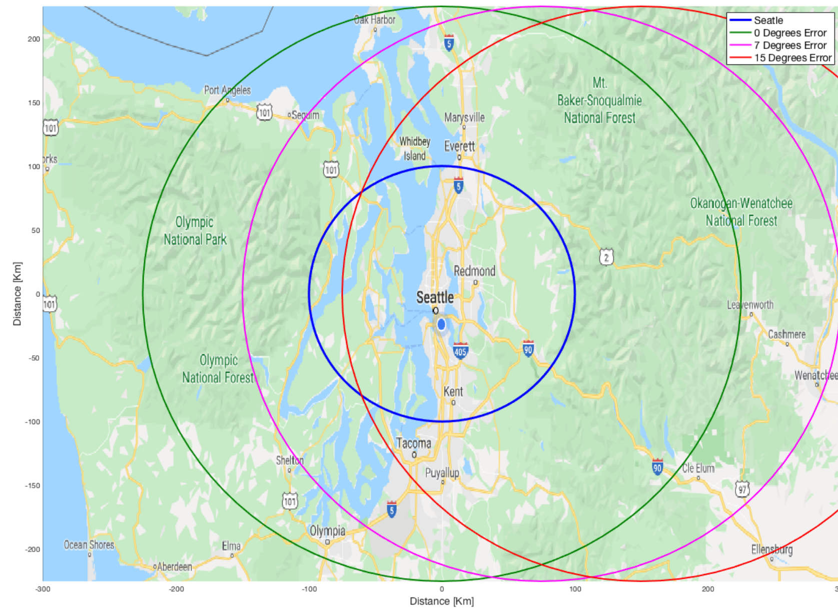

Figure 1.3: The small blue circle represents what we are considering Seattle, while the larger circular areas represent what the camera can see.

As seen in Figure 1.3, if we only allow a maximum error of 7 degrees this should ensure a considerable margin of safety for the image to be taken correctly. We then split this 7 degree requirement between the attitude estimation accuracy and the control accuracy. This is best explained using Figure 1.4

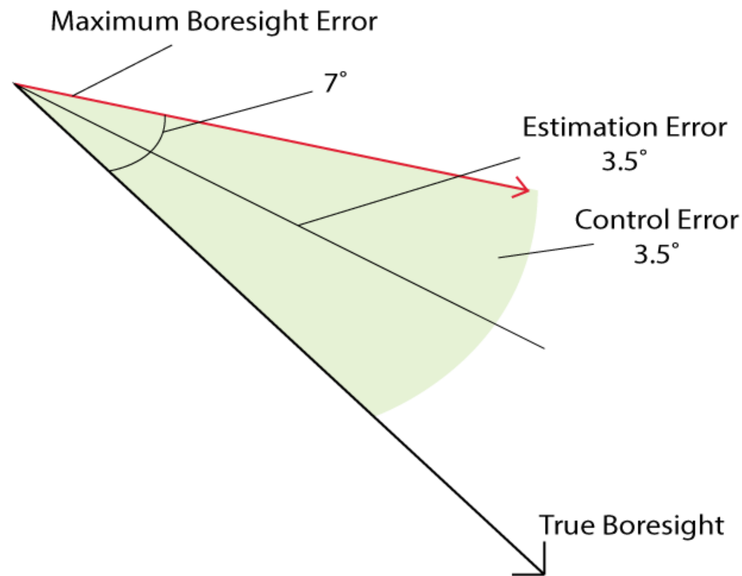

Figure 1.4: Determining the pointing accuracy requirement by studying the cameras "view cone".

Should both of these quantities remain within 3.5 degrees error their sum will never exceed 7 degrees and thus the superior requirement is met. Keep in mind that directly adding these quantities allows us to look at the worst case scenario and hence is appropriate for the requirement. In addition we also restrict the roll error to 7 degrees as this will have a similar effect to the image.

The GNC system shall maintain Attitude Knowledge Estimation during periods of eclipse (max. 35 minutes) when attitude is propagated solely using gyroscope measurements. Attitude estimation routines rely on the measurement of external vectors. During eclipse some of these vectors will become unavailable. This requirement ensures that we will design some method (it will later be described as the Kalman Filter) to allow our attitude estimation to continue

during an eclipse.

1.3 The Payload

SOCi's Optimal Attitude Reorientation (SOAR) is the name given to the feed-forward guidance system on board SOCi. It uses convex optimization techniques to provide the GNC system with guidance trajectories that will serve to reorient the spacecraft subject to some constraints. The most information on SOAR can be found in [12] as well as in Section [12]. This algorithm is powered by the ECOS, Embedded Conic Solver [10] and has been developed by Taylor P. Reynolds of RAIN lab here at UW. This thesis is mostly concerned with the implementation and testing of SOAR as well as how it fits into the overall system.

Chapter 2
MATHEMATICAL MODELING

This chapter will review the mathematical modeling within both the simulation and the flight software. It may be important to introduce the following graphic,

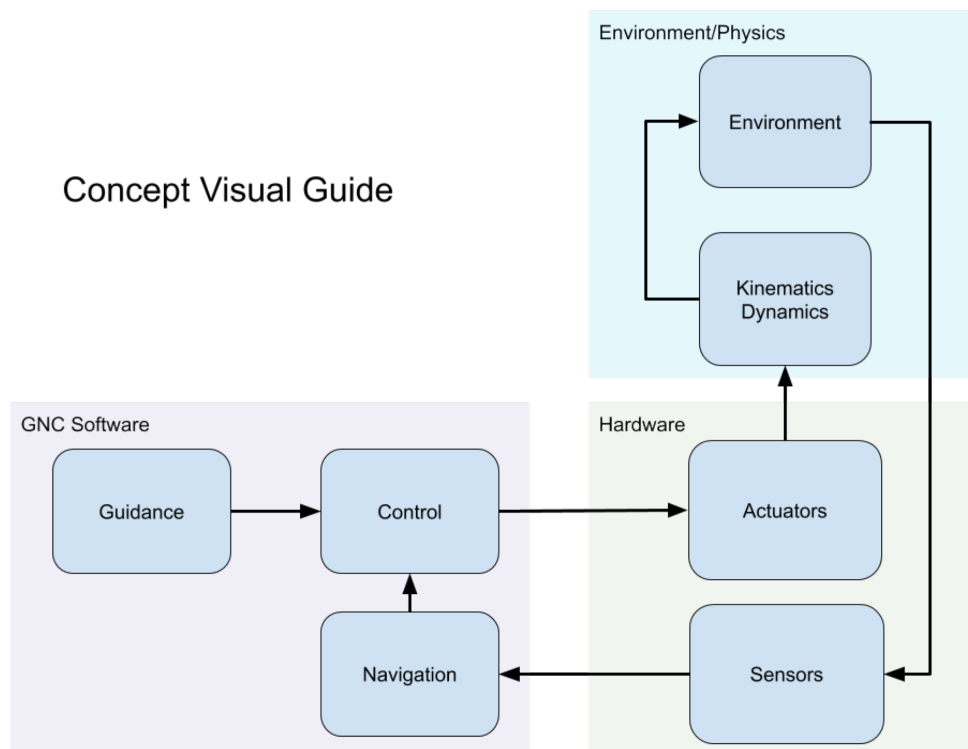

Figure 2.1: This diagram shows each section of the simulink code. It is an important distinction to not theat FSW is the only portion of code to go to space.

Each block in Figure 2.1 represents code within SimuLink, the broader highlighted

areas represent the purpose this code fulfills. In this section we will describe the mathematics used within the environmental and physical models, as well as the mathematics behind the attitude control system.

2.1 Kinematic and Dynamic Model

In this section the underlying mathematics determining the physics of our satellite are discussed. These mathematics inform not only the simulation model but will later be shown to be quite useful when developing a controller. At a glance the kinematic and dynamic model we use is as follows:

$$\dot{q}(t) = \frac{1}{2}q(t) \otimes \omega(t) \tag{2.1}$$

$$J\dot{\omega}(t) = \tau_c(t) + \tau_d(t) - \omega^x(J\omega(t) + h_\omega(t)) \tag{2.2}$$

The derivation of these mathematical models are detailed in [14]. For now I will seek to explain them within the context of our mission and provide some insight into how they are implemented within SimuLink.

2.1.1 Kinematics

Kinematics describe the motion of a spinning body absent any forces, relating velocities to positions. The kinematic equations we use can be described by 2.1, where q is the unit quaternion, ω is our angular rate, and $\otimes$ describes "quaternion multiplication". The quaternion is a paremeterization of the possibly more familiar Euler rotation axis-angle pair. This 4x1 vector encapsulates all the information you would find within a Directional Cosine Matrix (DCM) by embedding that information within a

higher dimension. This relationship can be directly defined using,

$$C_{B\leftarrow I} = (2q_0^2 - 1)I_{3\times 3} - 2q_0 q_v^\times + 2q_v w_v^T \tag{2.3}$$

The quaternion can be broken into the vector and scalar components as,

$$q = \begin{bmatrix} q_0 \\ q_v \end{bmatrix} = \begin{bmatrix} \cos(\frac{\phi}{2}) \\ \boldsymbol{a}\sin(\frac{\phi}{2}) \end{bmatrix} \tag{2.4}$$

Where ϕ is Euler angle and $\boldsymbol{a}$ is the Euler axis. This is what is known as scalar first notation [14]. Lastly the $\otimes$ operator works as following, given two quaternions q and p:

$$q \otimes p = \begin{bmatrix} q_0 p_0 - q_v^T p_v \\ p_0 q_v + q_0 p_v + p_v^\times q_v \end{bmatrix} \tag{2.5}$$

It's important to note here that the superscript $^\times$ is used to denote the matrix multiplication formulation of cross product. Secondly for the multiplication to work between q, our quaternion, and ω our angular rate, ω must be formulated as what is called a "pure quaternion" [5] where we set the scalar part to zero thus leaving only the vector component. Effectively this multiplication creates a skew symmetric matrix which in some literature [3] can be written,

$$\dot{q} = \Xi(\omega) q \tag{2.6}$$

This reduces the computation down to a matrix multiply. In addition for the kinematics to work out we require a normalization step of the resulting quaternion. When implemented in our software this can be found in "satelliteDynamics_lib/quat_propogation" and appears,

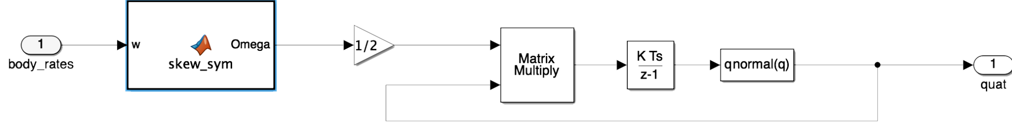

Figure 2.2: This block performs the propogation of our attitude quaternion.

2.1.2 Dynamics

The dynamics of our spacecraft are described by,

$$J\dot{\omega}(t) = \tau_c(t) + \tau_d(t) - \omega^{\times}(J\omega(t) + h_w(t)) \tag{2.7}$$

These equations are formally known as Euler rigid body dynamics [14]. J is the positive definite inertia matrix of our CubeSat found by,

$$J = \begin{bmatrix} \frac{1}{12}m(h^2+d^2)^2 & 0 & 0 \\ 0 & \frac{1}{12}m(w^2+d^2)^2 & 0 \\ 0 & 0 & \frac{1}{12}m(2^2+h^2)^2 \end{bmatrix}. \tag{2.8}$$

where m is mass and w, h, d are the dimension of the cuboid. ω describes our angular rates relative to spacecraft inertial frame. τ_c is the control torque (in this case generated from the reaction wheels),and τ_d is the disturbance torques. $h_w(t)$ is the momentum of our onboard momentum storage devices, in this case it is the reaction wheel momentum. These dynamics can be found within “satelliteDynamics_lib/RigidBodyDynamics/dynamics” and appears as,

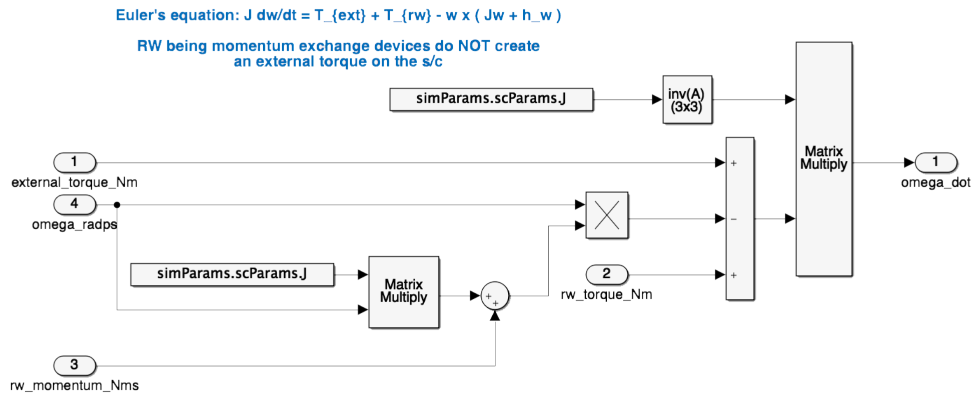

Figure 2.3: This block performs the propogation of our Rigid Body Dynamics.

To complete our dynamics the equation that governs our orbital dynamics can be written,

$$\ddot{r} = a_g(t) + a_d(t) \tag{2.9}$$

Which describes the fact that our acceleration is dictated by a_g, our gravitational acceleration, and a_d, acceleration due to environmental disturbances. These disturbances will be further detailed in the following section. For now as long as these accelerations are accurately determined they can be represented in SimuLink by,

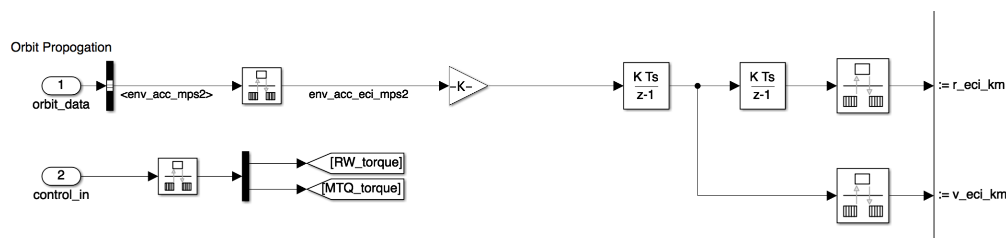

Figure 2.4: This block performs the propagation of orbital parameters r (radius) and v (orbital velocity).

2.2 Environmental Modeling

In order to properly simulate and test our flight controller we must be able to accurately simulate the external forces (disturbances) that act on our spacecraft. There are several possible sources of disturbance torques on the spacecraft which we have boiled down into three main components: atmospheric drag, gravity gradients, magnetic field residual torques, and solar pressure.

2.2.1 Time

One of the first and most important task when setting up an environmental simulation is to establish the time frame in which our system will operate. Once common solution is to run things on GPS Time as you can receive updates from GPS satellites, however, we will not be using a GPS onboard SOCi. This means we will be using one of the other standard time systems. Some algorithms and coordinate rotation require other time standards be present so we have selected several other time standards to be computed within the simulation. Spacecraft mission elapsed time (MET) is assumed to be kept in UTC seconds since J2000. When required, the time value may be mapped to the UT1, TT, and TAI time frames for use in flight code or ground simulation software. Below is a short summary of each specified time frame

as defined in [1].

- **Universal Time 1 (UT1)** closely matches UTC and is based on the fictitious mean sun exhibiting uniform motion in right ascension along the equator.

- **Terrestrial Time (TT)** is the theoretical timescale of apparent geocentric ephemerides of bodies in the solar system [1]. It is independent of equation of motion theories and utilizes the SI seconds as the fundamental interval.

- **Temps Atomique Internationale (TAI)** is the international atomic time which is independent of the average rotation of the Earth. It is based on counting the cycles of a high frequency electrical circuit maintained in resonance with a cesium-133 atomic transition.

- **Greenwich Mean Sidreal Time (GMST)** is the measure of the earth's rotation with respect to distant celestial objects.

- **Greanwich Apparent Sidreal Time (GAST)** is GMST but with a correction for the notation of Earth's poles.

In addition to these frames here is a list of other time systems usied within the calculation of the coordinate transformations,

- Julian Date

- Julian Century

- Greenwich Mean Sidreal Time (GMST) is the measure of the earth's rotation with respect to distant celestial objects.

- Greanwich Apparent Sidreal Time is GMST but with a correction for the nutation of Earth's poles.

Time needs to be tracked carefully in order to maintain correct knowledge of the satellites position. This will become clearer as we look into the coordinate systems used within the simulation and onboard the satellite.

2.2.2 Coordinate System

Dynamic equations only make sense in an inertial reference frame. As such, the inertial coordinate frame chosen for our simulation is the Earth Centered Inertial (ECI) convention. In order to move vectors from one coordinate system to another, we require coordinate transformations. These can also be described as rotations. Three other key Coordinate frames are the Earth Centered Earth Fixed (ECEF), Latitude/Longitude, and topocentric-horizon (SEZ) frames. Any rotation can be described by a matrix resulting from the dot product of a set of principle axis and that same set after a rotation. Take for instance the unit vector representing a principle axis $\vec{x}$. If we apply a rotation to this coordinate system $\vec{x}$ will have translated some distance, lets call this "new" vector $\vec{x}'$. To calculate the change in angle between $\vec{x}$ and $\vec{x}'$ we can simply take the dot product,

$$\vec{x} \cdot \vec{x}' = \|\vec{x}\| \|\vec{x}'\| \cos(\theta) = \cos(\theta). \tag{2.10}$$

If by repeating the process for all of the axis and all combinations we will esentially have a set of all information describing the angles by which this transformation has rotated us. This information (when stored in matrix form) is known as Directional Cosine Matrix (DCM). Any rotation in 3D can be decomposed into a set of three distinct rotations about an inertial frames principle axis. The three angles within

these matrices are formally known as the Euler angles [1] and are commonly referred to as ϕ, θ, ψ. The rotation matrices corresponding to these principle rotation angles are,

$$ROT1(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix} \tag{2.11}$$

$$ROT2(\alpha) = \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix} \tag{2.12}$$

$$ROT3(\alpha) = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.13}$$

$$\tag{2.14}$$

.

These matrices will be used throughout the following descriptions of the coordinate rotations used within our simulation.

The satellites orbital propagator provides the orbital radius and velocity in the ECI coordinate frame and requires a coordinate transformation to determine longitude and latitude. We use Borkowski's non-iterative method to calculate the geodetic latitude (gc) and ellipsoidal latitude (gd) which we then use to find the geocentric latitude [1]. We determine this using the equation below and the constant eccentricity of the Earth (e).

$$\tan(\phi_{gd}) = \frac{\tan(\phi_{gc})}{1 - e_{\oplus}^2} \tag{2.15}$$

$$\tan(\phi_{gc}) = (1 - e_{\oplus}^2)\tan(\phi_{gd}) \tag{2.16}$$

To convert from ECI to ECEF we must determine three transformation matrices from the Greenwich apparent sidereal time, the nutation, and precession theory models. When these three are combined they allow us to move between inertial and rotating reference frames. Rotation is the exact reason why precise measurement of time is so important. Sidereal time specifically allows us to go back and forth between rotating and non-rotating time frames[1]. Greenwich sidereal time is calculated by adding the Greenwich mean sidereal time (GMST) to the equation of the equinoxes which can be found in [1].The next transformation takes into account the periodic

effects contributed primarily by the Moon based on the 1980 IAU theory of nutation [1]. We determined the primary variation from a trigonometric series of 106 terms to find the nutation in longitude and the nutation in obliquity and add then Earth Orientation Parameters (EOP) corrections. Once again using our ROT matrices and angles gathered from the longitude, obliquity, and true obliquity we determine our second transformation matrix. The IAU 2000 precession theory uses angles taken

from the mean equator, mean equator of date and ecliptic of epoch. The equations used to determine these angles can be found in [1] and use Julian centuries (T_{TT}) from the base epoch which is J2000. We combine these three transformation matrices by

multiplying their transposes together to determine the final transformation matrix to go from ECI to ECEF. For the full mathematical details of this see [1] within the

chapter for time and coordinate systems.

2.2.3 Solar Model

Solar pressure torque is caused by the momentum imparted by incoming photons when they interact with the surface of the spacecraft. There are three basic interactions that each photon can perform,

- Absorption
- Specular Reflection
- Diffuse Reflection

All three of these will impart slightly different momentum changes. In order to really nail down the number you would have to model each one independently, however, because other people have gone to space before us we don't have to! The torque experienced by the spacecraft is not constant. It is a result of the combination of the geometry and orientation of a particular spacecraft, but the force per area (pressure) is constant. So all we need is this constant force and we can use it to inform our model. In order to properly calculate the force as vector we used the following equation,

$$F_i = P_{sp} \cos(\theta) A\boldsymbol{e} \tag{2.17}$$

Where each F_i is the force experienced in each axis. In order to calculate we must have the sun vector, $\boldsymbol{e}$. This vector is calculated outside of the solar pressure calculation as part of our environmental modeling. Once we obtain this vector we simply take a dot product between the sun vector and our body friend to find the

incident angle, . This $cos(\theta)A$ term gives us our projected area and when multiplied by the pressure constant P_{sp} will give us the resulting force. This force is then crossed with the distance of our center of pressure (a distance provided to us by the structures team) and thus we have the torque generated by the solar pressure,

$$\tau_{sp} = F_i \times r \tag{2.18}$$

where r is a vector pointing to the center of pressure.

2.2.4 Magnetic Field Model

Typically we would be able to refer to the SimuLink Aerospace Toolkit to model the magnetic field. However, in 2018 a significant shift in the magnetic field caused this toolkit to become invalid. We will also need to simulate Earth's magnetic field onboard the spacecraft for our estimation routine. With this in mind we have elected to write our own magnetic field model using the International Geomagnetic Reference Field (IGRF) as a guide. The IGRF is a large magnetic field using a large set of coefficients created by data points taken and sent in from around the world. The IGRF has been maintained and produced by an international team of scientists under the auspices of the International Association of Geomagnetism and Aeronomy (IAGA) since 1965 (Zmuda 1971) [6]. The model appears,

$$V(r,\theta,\phi,t) = a\sum_{n=1}^{N}\sum_{m=0}^{n}\left(\frac{a}{r}\right)^{n+1} \times [g_n^m(t)\cos(m\phi) + h_n^m(t)\sin(m\phi)P_n^m(\cos(\theta))]. \tag{2.19}$$

Where r is the radial distance from the center of the Earth, a = 6,371.2 km is the geomagnetic conventional Earth's mean reference spherical radius, θ is the

geocentric co-latitude, and ϕ denoting east longitude. $P_n^m(\cos(\theta))$ are the Schmidt quasi-normalized associated Legendre functions of degree n and order m. g_n^m and h_n^m are the Gauss Coefficients and are functions of time and are of the units nT. Typically these coefficients are assumed to be linear over five year periods. So for us these are pre-calculated and stored in lookup tables. From V we can calculate the magnetic field through,

$$\vec{B}(r, \theta, \phi, t) = -\Delta V. \tag{2.20}$$

This calculation becomes increasingly accurate the larger you make N and n. This is used inside our Environmental Simulation block as well as inside flight software, and exists as a MatLab function block.

2.2.5 Gravitational Model

Despite the small size, the difference in the gravitational force across the spacecraft will impart a torque. This is succinctly called the gravity gradient. Any non symmetric rigid body in orbit will be subject to a gravity gradient due to Earth's gravitational field. If we discretize the rigid body into a group of point masses the force acting on the ith mass appears,

$$F_i = m_i G r_i \tag{2.21}$$

Where G is the gravitational constant and m_i is the ith mass and r_s is the radial distance of that mass from the COM. This combined with the dynamics of the space spacecraft lead to the following equation for the torque,

$$\tau_{gg} = 3n^2[(J_z - J_y), (J_z - J_x), 0]^T \tag{2.22}$$

Where J_i are the principal axis of inertia and n is the mean motion. This is derived from taking a first order approximation of the Euler angle version of the dynamics. It is zero in the Z direction as this is the line of action for our Keplerian gravitational force.

2.2.6 Atmospheric Model

Despite being in space, objects in low Earth orbit still experience some interaction with atmospheric particles. Each interaction with these particles will impart some small force on the craft and, given enough time, this will impart a torque on the spacecraft. In order to calculate this torque, we first need to model the force imparted by a collision. The model we use is,

$$F_{drag} = \frac{1}{2}\frac{C_D A}{m}\rho\|v\|^2\frac{v}{\|v\|} \tag{2.23}$$

The force is acting opposite of the spacecrafts motion proportional to the drag coefficient, C_D, the area, A, the mass, m, the atmospheric density ρ, and $\vec{v}$ is our orbital velocity resolved in the body frame. Within the software the parameters for C_D and ρ are generated from the geometry and the atmospheric model provided by SimuLink within the aerospace toolkit. Otherwise, the torque is calculated by,

$$\tau_{atmo} = F_i \times r \tag{2.24}$$

where r is again a vector representing the point of pressure.

2.2.7 Verifying Orbital Dynamics and Disturbances

In order to assess our ability to simulate the spacecraft environment we have to perform data analysis and trade studies on the results of the simulation. The first task was to verify that our orbital dynamics were sensible, this verifies the gravity model as well as ensures our dynamics are handled properly. For this test we simulated an orbit for a long period of time an analyzed some of the characteristics. I chose the ISS orbit as it has a lot of documentation of the disturbances experienced as well as a wealth of available scientific data. With that in mind, data taken from the ISS must be scaled appropriately when compared to CubeSat data. At first glance this appears,

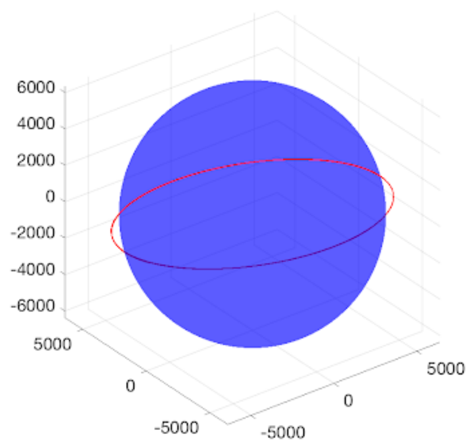

Figure 2.5: This was our first orbital graph showing what seemed to be a successful test of the orbital dynamics.

It was then decided to track the orbit's individual components as Figure 2.5 seemed to be growing over time during longer trials.

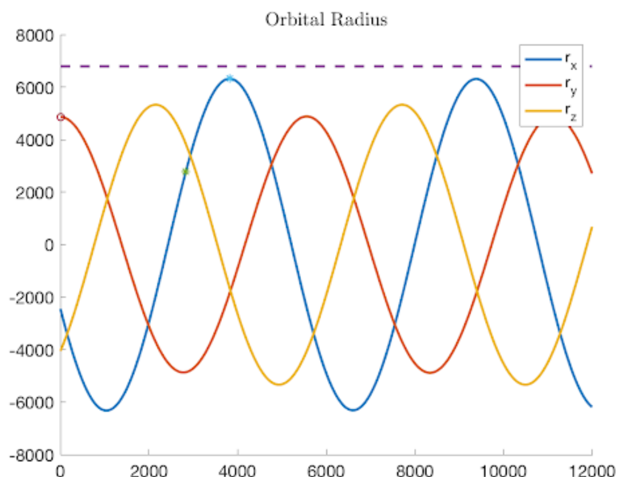

Figure 2.6: This test began to show the orbit's growth over time.

Analysis of this data showed us that we need to run the dynamics propagation with a maximum step size of 0.005 in order to produce accurate results.

Lastly, by plugging in values into our various torque models, we would expect the disturbance torques to be somewhere in the range of $10^{-9} - 10^{-8} Nm$. We can verify this by looking at data taken from previous CubeSat missions to confirm this range [13]. When running the simulation over long periods of time we found the disturbance torques to appear,

Torque Source	Magnitude [Nm]	Orbit Avg. Momentum [Nms]
Gravity Gradients	1.2E-8	0.011
Solar Pressure	2.6E-9	0.015
Magnetic Moments	4.5E-7	0.41
Atmospheric Torques	6.0E-8	0.055

This falls in line with both predictions and data found in [13]. It may be important for the reader to see how these torques change over time, so I will include a time series of the total disturbance torque.

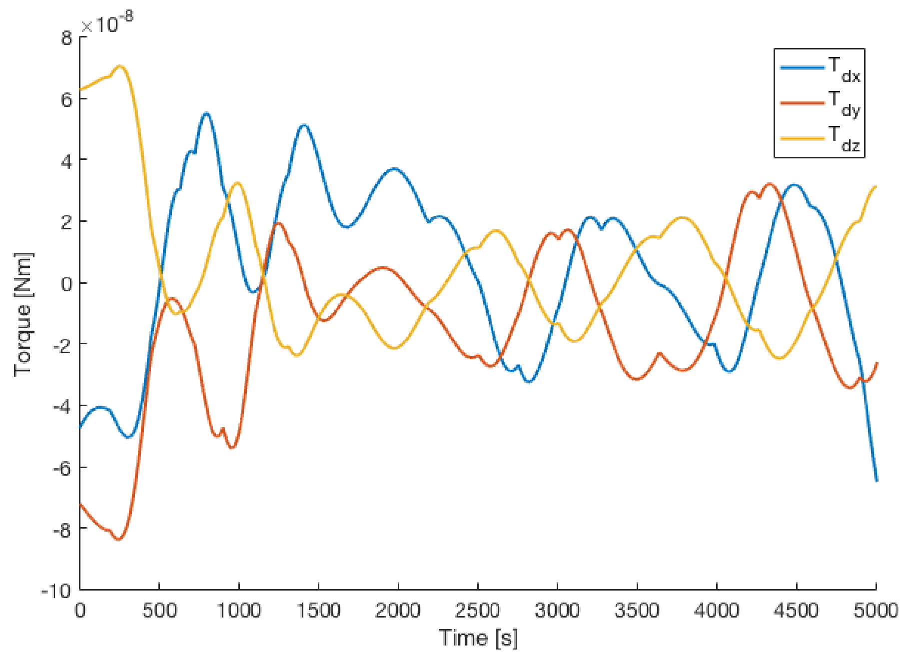

Figure 2.7: This shows us the sum of all of our disturbance torques over 1 orbit.

2.3 Controller Concept

In its most general form the Control side of our project boils down to two objectives:

1. Reduce Angular Rate to 0.

2. Reorient the craft to a desired attitude

We refer to these as Detumbling and Point to Point Reorientation, respectively. These two objectives are not mutually excusive and can be met simultaneously. As with any control system we use a feedback loop to drive the state of our craft. For the SOCi satellite our state is,

$$x = \begin{bmatrix} r \\ v \\ q \\ \omega \end{bmatrix} \tag{2.25}$$

Although the radius r and the orbital velocity v are within the state, we have no control over them, by design. Within the detumbling goal we try only to drive the angular rate, ω, to 0. During the reorientation maneuvers our goal is to drive $q \rightarrow q_{desired}$ as well as $\omega \rightarrow 0$. In this section we won't discuss where and how we get our desired attitude, but rather how we get to a desired attitude after it is provided as a reference.

2.3.1 Detumbling

The detumbling controller is fairly simple. All we need to drive our angular rate to zero is a proportional feedback law.

$$g(\omega) = -K\omega \tag{2.26}$$

In order for us to feel confident in a given control law there is one very key property to keep in mind. In order for a control law to be dependable we need it to be stable. Consider the following definitions of stability:

Def: Consider $\dot{x} = f(t, x)$ and some equilibrium point x_e. Then the dynamics are:

1. Stable if $\forall\ \epsilon > 0$ there exists some $\delta > 0$ such that $||x(t_0) - x_e|| < \delta \implies ||x(t) - x_e|| < \epsilon, \forall t \geq 0$.

2. Asymptotically stable if it is stable and there exists a $\delta > 0$ such that: $||x(t_0) - x_e|| < \delta \implies \lim_{t\to\infty} x(t) = x_e$.

3. Globally Asymptotically stable if it is stable and $\lim_{t\to\infty} x(t) = x_e$ for any $x(t_0)$.

4. Unstable if it is not stable.

Proving that our control in eq 2.26 leads our dynamics adhere to either 2 or 3 would mean we can tune K to get any desired transient effects and not have to worry about whether or not our system would be driven unstable. However, proving this directly can sometimes be fairly difficult and often not straightforward. In this instance we can turn to the Lyapunov Direct Method.

Theorem: (Lyapunov Direct Method) Consider the dynamics $\dot{x} = f(x, t) \equiv f(x)$ and some open set $D \subseteq \mathbb{R}^n$ Let $x = 0$ be an equilibrium point where $0 \in D$. Consider a locally positive definite function $V : D \to \mathbb{R}$, if along solution $x(t) \in D$ we have,

1. $\dot{V}(x) \leq 0$, then $x_e = 0$ is stable

2. $\dot{V}(x) < 0$, for $x \neq 0$, then $x_e = 0$ is asymptotically stable.

3. if $D = \mathbb{R}^n$ and $||x|| \to \infty$ implies that $V(x) \to \infty$ then $x_e = 0$ is globally asymptotically stable.

An easier way of looking at the Lyapunov Direct Method is to think about it in terms of energy. We should choose some $V(x)$ as to encapsulate the energy of the system. Let's specifically choose,

$$V(x) = \omega^T J \omega \tag{2.27}$$

Where J is our positive definite inertia matrix. This function expresses our energy in a similar fashion as the classic $E = \frac{1}{2}mv^2$. Now should we find that the energy of this system is always decreasing, given some initial conditions. We should be able to say that we will always reach a point of minimum energy, i.e. a stable equilibrium. In fact this is just what Lyapunovs theory allows us to say. We know first off that,

$$v(0) = 0 \tag{2.28}$$

$$V(\omega) > 0 \forall \omega \neq 0.\text{since J is always positive definite} \tag{2.29}$$

The function V is referred to as a "Candidate Lyapunov Function". In order to show stability we first take the derivative of this function,

$$\dot{V}(\omega) = \omega^T J \dot{\omega} \tag{2.30}$$

$$= \omega^T J(J^{-1}(g(\omega) - \omega^{\times} J\omega)). \tag{2.31}$$

$$= \omega^T g(\omega) \tag{2.32}$$

Where $\dot{\omega}$ is taken from our dynamics and $g(\omega)$ represents our control law (in 2.2 it is just called τ_c). It is important to note that in actual novel control analysis, you take the derivitive of your candidate function and use this to derive the control law. That is to say if we know that the derivative must be negative, we can find some control law g that satisfies that condition. In this case the control is known to us so we can use this to go backwards and show stability. So, by substituting Equation 2.26 into the above we get,

$$\dot{V}(\omega) = \omega^T(-k\omega) = -\omega^T K\omega \tag{2.33}$$

Which we know $\omega^T K\omega$ will remain positive definite as long as K is positive definite. So, as long as K remains positive definite,

$$\dot{V}(\omega) \leq 0. \tag{2.34}$$

Therefore, at the very least the closed loop system is stable! Furthermore, since for $\omega \neq 0$ we have $\dot{V}(\omega) < 0$, so the system is asymptotically stable and the result is global. As long as we keep our feedback gain matrix K positive definite, we can tune it to have whatever transient response we wish and the system will be remain stable, eventually driving the rotational speed to zero. Now, we can take this same analysis and extend it to the point-to-point reorientation.

2.3.2 Point-to-Point Reorientation

There are two point-to-point controllers within flight software. The reason for this will become clearer as you read further. The first controller, a proportional derivative controller (PD), is probably more familiar. A newcomer to the project should familiarize themselves with this controller first and foremost.

The PD Controller

Recall that we have the coupled dynamic/kinematic set of equations,

$$\dot{q}(t) = \frac{1}{2}q(t) \otimes \omega(t) \tag{2.35}$$

$$J\dot{\omega}(t) = \tau_c(t) + \tau_d(t) - \omega^{\times}(J\omega(t) + h_{\omega}(t)) \tag{2.36}$$

If we are interested in controlling the attitude to some desired orientation, q_d then we will need a coupled control law,

$$u = g(q, \omega). \tag{2.37}$$

For this we consider the error state,

$$\delta q(t) = q_d^* \otimes q(t) \tag{2.38}$$

Where $\otimes$ is essentially the quaternion version of subtraction. The result is a quaternion describing the difference in these two orientations. This means the dynamics are described by,

$$\delta\dot{q} = q_d^* \otimes \dot{q} = q_d^* \otimes (\frac{1}{2}q \otimes \omega) = \frac{1}{2}\delta q \otimes \omega \tag{2.39}$$

These are known as the error dynamics. The goal in the reorientation case is to drive the error quaternion, δq to the attitude equivalent of zero, which is,

$$\delta q \to \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{2.40}$$

You may also see this described in the literature as the quaternion "origin". The $[1000]^T$ error describes zero difference in attitude. Similar to the previous section we can start by suggesting a possible control law. In this case we can choose proportional control on the attitude with derivitive feedback on the angular velocity. This is somewhat intuitive due to the the fact that angular velocity describes the way an attitude will change over time. This is similar to the relationship between position an velocity but with more angles and spinning bodies. With this in mind our proposed control law is,

$$g(q, \omega) = -K\delta q_v - K_d\omega. \tag{2.41}$$

Where q_v is the vector portion of the quaternion. If you would like a more detailed derivation of this control law please see [4]. Now Consider the Lyapunov Candidate,

$$V(q, \omega) = \frac{1}{2}\omega^T J\omega + \frac{1}{2}K\delta q_v^T \delta q_v + \frac{1}{2}K(1 - \delta q_0)^2 \tag{2.42}$$

Where we assert that $K \succ 0$. When $(q, \omega) = (q_d, 0)$ we have $V(q, \omega) = 0$ and

when $(q,\omega) \neq (q_d, 0)$ we have $V(q,\omega) \succ 0$ since J and K are both positive definite. So once again we start by taking a derivative,

$$\dot{V}(q,\omega) = \frac{1}{2}\omega^T J\dot{\omega} + K\delta q_v^T \delta\dot{q}_v - K(1-\delta q_0)\delta\dot{q}_0 \tag{2.43}$$
$$= \frac{1}{2}\omega^T g(q,\omega) + K\delta q_v^T(\frac{1}{2}(\delta q_0 I_3 + \delta q_v^{\times})\omega) \tag{2.44}$$
$$- K(1-\delta q_0)(-\frac{1}{2}\delta q_0^T \omega) \tag{2.45}$$
$$= \omega^T g(q,\omega) + \frac{1}{2}K\delta q_0 \delta q_v^T \omega + \frac{1}{2}K\delta q_v^T \omega \tag{2.46}$$
$$- \frac{1}{2}K\delta q_0 \delta q_v^T \omega \tag{2.47}$$
$$= \frac{1}{2}\omega^T g(q,\omega) + \frac{1}{2}K\delta q_v^T \omega. \tag{2.48}$$
$$\tag{2.49}$$

Then by substituting in equation 2.41 into the above we get,

$$\dot{V}(q,\omega) = \frac{1}{2}\omega^T(-K\delta q_v - K_D\omega) + \frac{1}{2}K\delta q_v^T \omega. \tag{2.50}$$
$$= -\frac{1}{2}K_D||\omega||^2 \leq 0. \tag{2.51}$$

Which means that the closed loop system is stable! But what about asymptotic stability? In order to prove that we need to use another tool. That tool is called the La Salle Invariance Principle.

Theorem: (La Salle's) Consider the dynamics $\dot{x} = f(x)$ and some open set $D \subseteq \mathbb{R}^n$ (with $0 \in D$). If for a solution $x(t)$ there is some set $\Omega \subseteq D$ such that $x(t) \in \Omega \; \forall \; t \geq 0$ and $V : D \to \mathbb{R}$ is a continuously differentiable function such that along $x(t)$ we

have $\dot{V}(x) \leq 0$. Then,

$x(t) \to M$ as $t \to \infty$.

Where M is the largest invariant set contaned in,

$E = \{x \in \Omega | \dot{V}(x) = 0\}$.

If $M = 0$, then this is asymptotic stability.

So for us this means we need to show that, $\dot{V} = 0$ cannot persist if we don't have $\delta q_v = 0$. The set E is then,

$$E = \{(q, \omega) | \omega = 0\} \tag{2.52}$$

For trajectories contained in E, we have $\dot{\omega} = 0$ as well. The closed loop dynamics are in this case,

$$\delta \dot{q}_v = 0 \tag{2.53}$$

$$\delta \dot{q}_0 = 0 \tag{2.54}$$

$$0 = J^{-1}(-K\delta q_v)) = -KJ^{-1}\delta q_v \tag{2.55}$$

Since J is a positive definite matrix, these equation imply that $\delta q_v = 0$. Thus,

$$M = \{(q, \omega) | \delta q_v = 0 \text{ and } \omega = 0\} \tag{2.56}$$

and La Salle's invariance principle tells us that we are asymptotically stable. Furthermore, the set Ω can be found by using V to bound the angular velocity.

$$\frac{1}{4}\omega^T J\omega = V(q, \omega) - K\delta q_v^T \delta q_v - K(q - \delta q_0)^2$$

Since $\lambda_{min}(J)||\omega||^2 \leq \omega^T J\omega \leq \lambda_{max}(J)||\omega||^2$ we have,

$$\frac{1}{4}J_{min}||\omega||^2 \leq V(q,\omega) - K\delta q_v^T \delta q_v - K(q - \delta q_0)^2 \tag{2.57}$$

$$\leq V(q,\omega) \tag{2.58}$$

$$\implies ||\omega|| \leq \sqrt{\frac{4V(q,\omega)}{J_{min}}} \tag{2.59}$$

Since $\dot{V} \leq 0$ we know that $V(q(t),\omega(t)) \leq V(q(0),\omega(0))$ Thus,

$$||\omega|| \leq \sqrt{\frac{4V(q(0),\omega(0)}{J_{min}}}$$

So,

$$\Omega = \{(q,\omega)|||q|| \leq 1 \text{ and } ||\omega|| \leq \sqrt{\frac{4V(q(0),\omega(0)}{J_{min}}}\}$$

Where the initial condition is arbitrary here, so the result is global. Therefore, we have shown our system to be globally asymptotically stable. Some very important things to note from this are,

1. The result is **almost** global, since if we start at $-q_d$ we will stay there. If perturbed you will rotate a full 360° rather the expected delta.

2. The PD control law is a linear feedback law that stabilizes and nonlinear spacecraft.

3. The control law does not depend on the physical characteristics of the spacecraft, namely J. This means it is a "model free control law".

Possibly the most important point here is 3. Model Free Control means that while the dynamics we use to arrive at this are nonlinear we don't rely on them for tuning our controller. Furthermore this means we can use a fully linearized system to tune our control to the desired transient effects and we know that the system **must** converge to the desired attitude. So, as long as we don't mind some deviation in transient behavior far away from the desired attitude, we know that it will behave exactly as we wish once we inevitably come close enough to q_d so that linearization becomes accurate. Below we can see an example of this controller in action,

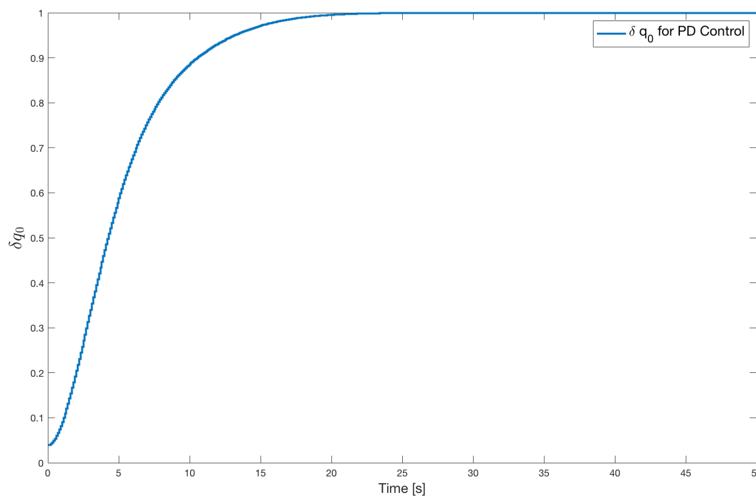

Figure 2.8: Here we are looking at the scalar component of the error quaternion. This controller was tuned to maximize the speed and minimize the overshoot.

Slew Rate-Limited Controller

Although the other controller will meet all requirements and achieve the mission objectives, it may be pertinent to employ some slightly more advanced methods to tweak the performance to gain some improved transient effects. Specifically, we are interested in limiting the slew rate of our point-to-point maneuvers so we don't enter the linear region with too much angular velocity. This improves the accuracy of the linear analysis, as the slower the movement, the closer to linear the dynamics become. The description here will be fairly limited as, once we understand the origin and implementation of the previous PD controller, this one is fairly straight forward. The controller comes from a paper by Bong Wie and Jian Lu and can be found in [7]. This controller uses command saturation to limit the command control torques in such a way as to limit the slew rate. Mathematically a saturation function can be defined as,

$$sat(x_i) = \begin{cases} x_i^+ & \text{if } x_i > x_i^+ \\ x_i & \text{if } x_i^- \leq x_i \leq x_i^+ \\ x_i^+ & \text{if } x_i < x_i^- \end{cases} . \tag{2.60}$$

A common example of a saturation function would be the sign function,

$$sign(x_i) = \begin{cases} +1 & \text{if } x_i > 0 \\ 0 & \text{if } x_i = 0 \\ -1 & \text{if } x_i < 0 \end{cases} . \tag{2.61}$$

One important thing that is thoroughly explored in [7] is the fact that it does not suffice just to introduce a saturation limit, to do this would not ensure stability.

Instead, the paper goes through the process of revisiting the analysis we did earlier in the context of the altered control law. This new control law appears,

$$u = -Ksat(Pq) - C\omega + \omega \times J\omega \tag{2.62}$$

where,

$$K = diag(k_1, k_2, k_3)J$$
$$P = diag(p_1, p_2, p_3)$$
$$C = cJ.$$

In this k_i,p_i, and c are all positive scalar constants serving as the controller gains. They can be tuned to achieve the required transient behavior. To determine k_i we can use the following formulas,

$$k_i = c\frac{|q_i(0)|}{||q(0)||}\dot{\theta}_{max} \tag{2.63}$$

and

$$KP = kJ \tag{2.64}$$

where $\dot{\theta}_{max}$ is the maximum allowable slew rate and $k \equiv k_i p_i$. This means that k_i and p_i can be determined by setting your max slew rate and some scalar c. It is important to remember here that $q(0)$ is the eigen axis about which the rigid body will rotate. The scalar c is what determines your transient effects such as settling time. If we look at the entire length of this maneuver as three segments of time over the range $[0, +\infty)$ we can see it as three segments,

$$[0, t_s] \text{ Acceleration Phase} \tag{2.65}$$

$$[t_s, t^*] \text{ Coast Phase} \tag{2.66}$$

$$[t^*, +\infty) \text{ Deceleration Phase} \tag{2.67}$$

$$\tag{2.68}$$

Where t_s is the settling time. These times relate to c via,

$$t_s \simeq \frac{4}{c} \tag{2.69}$$

$$t^* \simeq t_s + \frac{2}{\dot{\theta}_{max}} tan^{-1}\left(\frac{||q(0)||}{q_4 t_s}\right). \tag{2.70}$$

We actually employ a specific instance of this controller designed to not only perform a slew maneuver as fast as possible within the limits of some maximum rate, but within the saturation limits of rate gyros as well as reaction wheels [7]. For this we employ a second type of saturation function which set our commanded torque as,

$$\tau = \underset{\sigma}{sat}(\tau_c) = \begin{cases} \tau_c & if \sigma(q,\omega) \leq 1 \\ \tau_c/\sigma(q,\omega) & if \sigma(q,\omega) > 1 \end{cases} \tag{2.71}$$

where the σ function is,

$$\sigma(q,\omega) = ||T\tau_c||_\infty = \max_i |(T\tau_c)_i|, \tag{2.72}$$

and,

$$T = diag(\frac{1}{\tau_{c1}}, \frac{1}{\tau_{c2}}, ...). \tag{2.73}$$

The result is a control law that appears,

$$u = -\underset{\sigma}{sat}[Ksat(Pq) + C\omega]. \tag{2.74}$$

In summary, by employing a few saturation functions and shuffling the derivation of the control gains, we have created a globally stable version of our control law that allows us to perform a slew maneuver as fast as possible without going above a safe slew rate. This can reduce some wobble and actually improves the efficiency of our overall system. Another interesting fallout of this configuration is the fact that it converges to the previous control law once angle error and angular velocity get sufficiently small. The convergence also coincides with the linear approximation becoming valid. This makes the tuning of this controller a combination of tuning for slew rate as well as using the analysis from our previous controller to synthesize gains. Now let's look at how the PD controller and this new controller perform side-by-side when given the same objective,

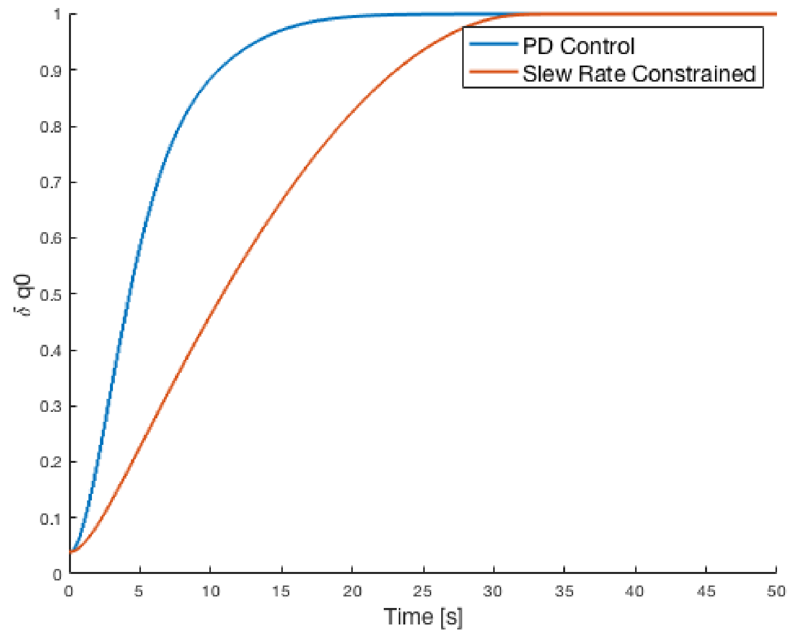

Figure 2.9: Notice that due to the slew rate constraint this controller will typically be slower. We are looking at the error quaternions scalar component.

A second set of data that may be interesting is to look at the actual speed at which these things move. Specifically let's look at the $\frac{deg}{sec}$,

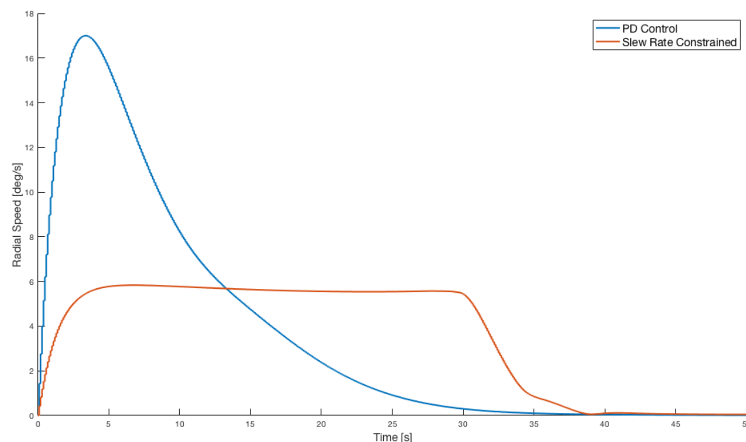

Figure 2.10: Here we compare the difference in the rotational speeds of the slew rate and PD controller.

The reduced slew rate means once we reach the desired attitude we can come to a more controlled stop and avoid the initial spike of the PD controller.

2.3.3 Payload: SOAR

The Satellite for Optimal Control and Imaging without the optimal part would just be SCI, thus losing the cool salmon pun and therefore all reason to exist. Jokes aside, the goal of our mission is to take pictures and test out the Convex Optimization-based trajectory planning algorithm. This section seeks to explain the payload from our systems perspective. For a more in depth explanation, please reach out to Taylor Reynolds. We are essentially solving a constrained optimal control problem. In simplest terms, this means you have some function that describes the cost of movement (dollars per gallon of gas, for example) constrained by some conditions (must drive slower than 80 mph) and you must find a set of inputs to keep that cost to a

minimum. The method of detailing and solving these problems mathematically is known as optimal control. We are interested in one very specific form of optimization, Convex Optimization.

Systems Perspective

From the perspective of the controller, the payload is simply a trajectory generation tool. While the payload itself requires it's own dynamic model and solver, really, the only output we care about is the resulting desired state and control output. How this interplay will work is, the solver will output a series of "set points". These set points will be a series of $[q_d, w_d, u]^T$, which will be fed to the PD control loop. The subsequent control loop works the same as before. A block diagram can be seen here,

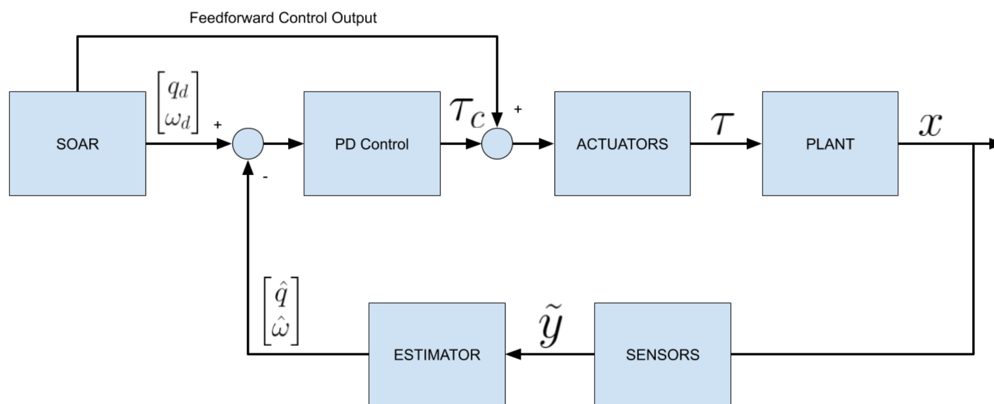

Figure 2.11: This Block diagram displays the interplay between the payload target generation and the existing feedback control loop.

The second adding/subtracting junction will allow us to alter the control output out of the PD Controller to more closely match the optimal one. In addition, the payload will require clearance from CDH and GNC to operate. This ensures that

it will only be allowed to run under the most ideal circumstances. This means that in order to run the payload will require a direct command from CDH as well as the necessary environmental measurements. This figure details all of inputs/outputs of the SOAR library,

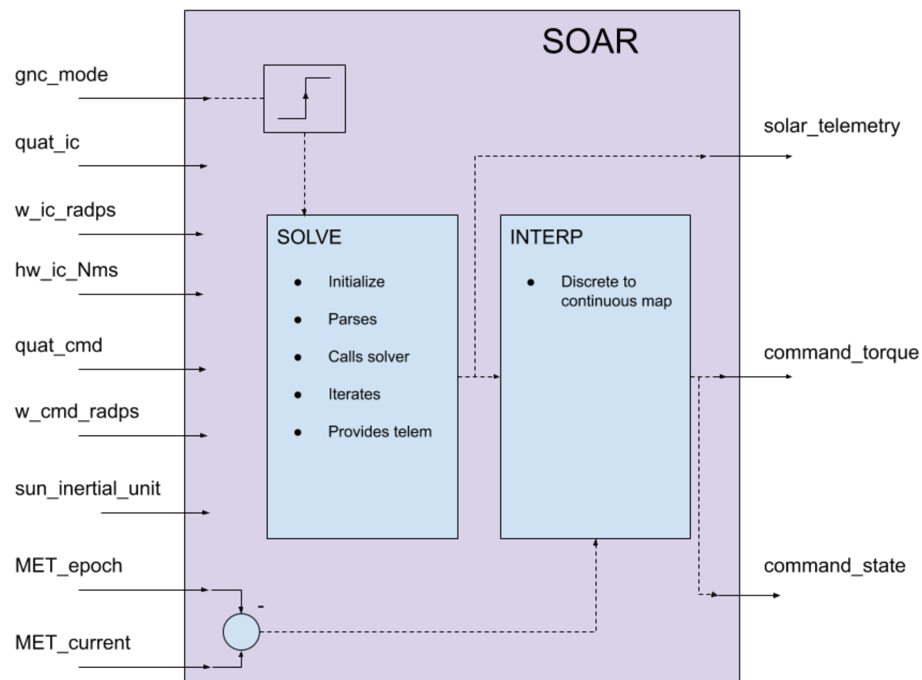

Figure 2.12: This figure details the inputs/outputs of the SOAR system as well as an overview of the inner workings.

An important takeaway from Figure 2.12 is the overview of the inner workings. Here we see that the payload consists of a solver and interpreter. The solver uses discrete time mathematics to create an optimal trajectory for the spacecraft. However, in order to ensure the same optimal trajectory is realized in continuous time

(the real world), we require a second mapping, in this case an interpreter, to move things back to the continuous time domain.

Problem Statement

For completeness I have chosen to include the mathematical problem statement, although for a full derivation and explanation of the algorithm you will need to see [12]. Similar to the previous section or state and control vectors are described as,

$$\boldsymbol{x}(t) = \begin{bmatrix} \boldsymbol{q}(t) \\ \boldsymbol{h}_{\mathcal{B}}(t) \\ \boldsymbol{h}_w(t) \end{bmatrix}_{10\times 1} \quad \text{and} \quad \boldsymbol{\tau}(t) = \begin{bmatrix} \tau_x(t) \\ \tau_y(t) \\ \tau_z(t) \end{bmatrix} \in \mathbb{R}^3 \tag{2.75}$$

In the same vein the dynamics and kinematics can be described by,

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{\boldsymbol{q}} \\ \dot{\boldsymbol{h}}_{\mathcal{B}} \\ \dot{\boldsymbol{h}}_w \end{bmatrix} = \begin{bmatrix} \frac{1}{2}\boldsymbol{q} \otimes (J^{-1}\boldsymbol{h}_{\mathcal{B}}) \\ -\boldsymbol{\tau} + (\boldsymbol{h}_{\mathcal{B}} + \boldsymbol{h}_w)^{\times} (J^{-1}\boldsymbol{h}_{\mathcal{B}}) \\ \boldsymbol{\tau} \end{bmatrix}. \tag{2.76}$$

With these in hand, the minimization problem can be formulated as follows,

$$
\begin{aligned}
\min_{t_f,\boldsymbol{\tau}(\cdot)} \quad & \int_{t_0}^{t_f} \boldsymbol{\tau}^\top \boldsymbol{\tau}(t) dt \\
\text{subject to:} \quad & (2.76) \\
& \|J^{-1}\boldsymbol{h}_{\mathcal{B}}\|_\infty \leq \omega_{\max} \\
& \|\boldsymbol{\tau}\|_\infty \leq \tau_{\max} \\
& \boldsymbol{q}^T M_E \boldsymbol{q} \leq 2 \\
& \boldsymbol{q}^T M_I \boldsymbol{q} \leq 2 \\
& t_{f,\min} \leq t_f \leq t_{f,\max} \\
& \boldsymbol{q}(t_0) = \boldsymbol{q}_{ic}, \quad \boldsymbol{h}_{\mathcal{B}}(t_0) = \boldsymbol{h}_{\mathcal{B},ic}, \quad \boldsymbol{h}_w(t_0) = \boldsymbol{h}_{w,ic}, \\
& \boldsymbol{q}(t_f) = \boldsymbol{q}_f, \quad \boldsymbol{h}_{\mathcal{B}}(t_f) = 0.
\end{aligned}
\tag{2.77}
$$

This problem seeks to minimize the $\mathcal{L}_2$ of the net control torque signal. Subject to the constraints: rotational limits of the RWA, maximum possible torque, time constraints, and desired attitude. The two constraints that may not be as straightforward are the third and fourth constraint which refer to [8] and are the excluded attitude constraints. That is to say they detail regions in which the spacecraft cannot be oriented (for example: pointing the camera at the sun). The solver is driven by the Embedded Conic Solver or ECOS under the hood [10]. If the user would like to learn more about this, I suggest reading about “second order cone problems” before jumping into the ECOS solver.

Chapter 3

HARDWARE

In this chapter we will review the hardware chosen for SOCi, as well as how we chose to model this hardware mathematically within the Sim. The ICD provides a more technical hardware breakdown for each component.

3.1 Sensor Models

In order to properly estimate our attitude, we need the measurement of at least two vectors. These measurements can then be passed on to our estimator and resolved in the attitude quaternion. This will be further discussed in Chapter 4. Many small satellites make use of a star tracker in order to obtain measurements using only one instrument. However these sensors can be quite expensive and require the storage of star information onboard the satellite. To avoid this we use a Sun Sensor and a magnetometer. In addition, because of our estimation routine, we also require a gyroscope for measuring the body rates. For a more in depth explanation of the hardware take a look at the Interface Control Document (ICD) [11]. In order to properly test the estimation scheme, we need to create a simulation of the underlying hardware. This provides us with realistic "measurements" that allow us to properly tune the estimation scheme. This all happens within the sensor lib,

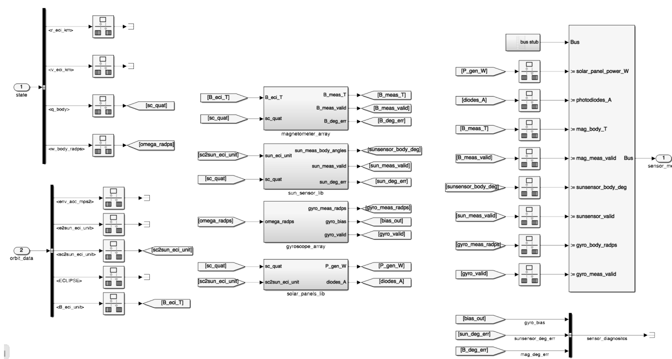

Figure 3.1: This is the Sensory Library Block in which we simulate each of various sensors. This is done by taking simulated environmental parameters and converting them into sensor measurements.

3.1.1 Sun Sensor

We have selected the following sun sensor: SolarMems nanoSSOC-D60 (digital). An ilustration of this sensor can be seen in Figure 3.2

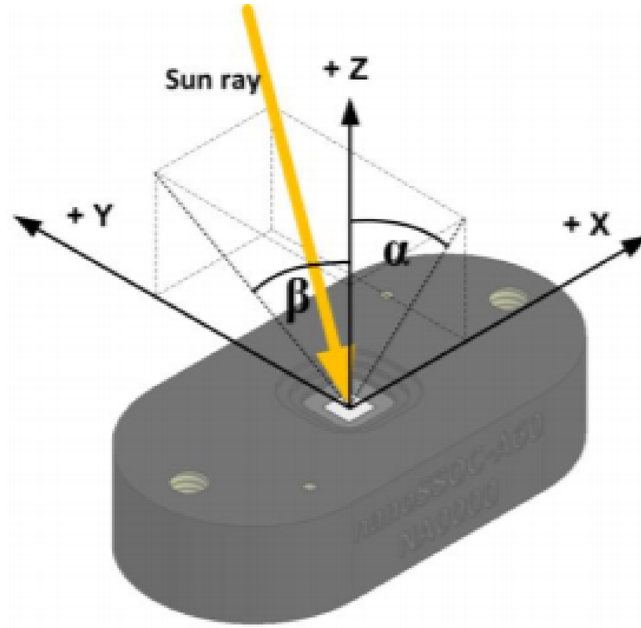

Figure 3.2: This diagram depicts the sun sensor onboard SOCi as well as the two angles that make up the sun measurement.

These sensors work by using a set of high sensitivity photo resistors and some onboard circuitry to convert this to two measurement angles, α and β. These can be converted to a sun unit vector using the following formula,

$$e_z = [\tan(\alpha)^2 + \tan(\beta)^2 + 1]^{-1/2} \tag{3.1}$$

$$e_x = \tan(\alpha) \cdot e_Z \tag{3.2}$$

$$e_y = \tan(\beta) \cdot e_Z \tag{3.3}$$

Where e is the unit sun vector (e_i being a component). This is implemented within our code in Flight Software as the following block within the voting scheme

lib,

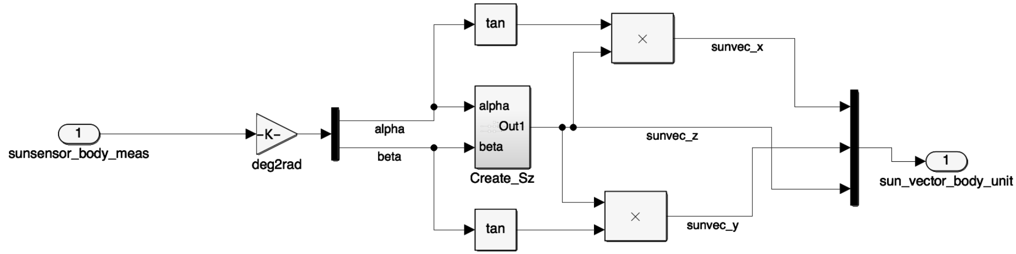

Figure 3.3: This block diagram shows how we implemented the conversion from sun sensor measurements to sun vector angle.

This is basically inverted within our Sensor Library Block. The incoming inertial unit sun vector is first rotated to body frame using a Directional Cosine matrix acquired from from converting the attitude quaternion to rotation matrix. This is done using a SimuLink library block. The body frame unit sun vector is described by,

$$e_b = [e_{bx} e_{by} e_{bz}]^T \tag{3.4}$$

This is then converted to the measurement angles via,

$$\alpha = \tan^{-1} \frac{e_{bx}}{e_{bz}} \tag{3.5}$$

$$\beta = \tan^{-1} \frac{e_{by}}{e_{bz}} \tag{3.6}$$

In reality these measurements will not be perfect so we can assume that whatever

error or noise they have will be Gaussian. With that, we introduce noise into the system as,

$$\tilde{\alpha} = \alpha + \epsilon_\alpha \tag{3.7}$$

$$\tilde{\beta} = \alpha + \epsilon_\beta \tag{3.8}$$

With the distribution of these two Gaussian terms determined by the sun sensor data sheet. In addition, we also limit the output of this library to ±60° as this is the field of view of the actual sensor. To account for the fact that this limiter will still allow impossible measurements to be generated (like sensing through the spacecraft), we added a logical AND to the output comparing it to the possible range of measurements. This can be found in our sun sensor library block inside sensor lib,

Figure 3.4: In addition to simulating the sun sensor output we have also included the degree error as a sanity check and debugging value.

3.1.2 Magnetometer

The second vector we are measuring in the Earth's magnetic field. To do this we use magnetometers with triple redundancy. These sensors are quite cheap, so the redundancy serves two purposes: one as a backup in case of failure, and the second so we can impose a voting scheme should the accuracy of any one sensor drift over time. The sensor we have chosen is NXP FXAS21002C. Magnetometers are subject to what are called 'soft' and 'hard' iron biases. These are distortions in the surrounding magnetic field and are usually a product of something on the craft itself. A hard iron bias is typically created by electronics such as an inductor or some other device that generates a magnetic field as it operates. This results in a shift in the measured magnetic field. A soft iron bias is one caused by a distortion in the surrounding magnetic field. Certain ferrous metals, such as nickel and iron, cause Earth's magnetic field to warp within the area surrounding the spacecraft and can distort the measurements. From [5] we have the hard iron biases modelled as the Gaussian element, $\epsilon_h \in \mathbb{R}^3$. The soft iron biases are modelled as a distortion via the directional cosine matrix C_s. Finally, there is one more Gaussian term ϵ_B which represents the specific electronic sensor noise. These are all pulled together to form the measurement through,

$$\hat{b}_{mag} = C_{mag \leftarrow B}(C_s C_{B \leftarrow I}(q) b_I + \epsilon_h) + \epsilon_B \tag{3.9}$$

Where $C_{B \leftarrow I}(q)$ is our directional cosine matrix generated from the current attitude quaternion, b_I is the simulated magnetic field, and $C_{mag \leftarrow B}$ rotates from Body frame to the coordinate system of the magnetometer. Similar to the Sun Sensor Lib we also use the sensors known range to generate a validity boolean as well as a truth vs simulated degree error. This can be seen in our Sensor Lib as,

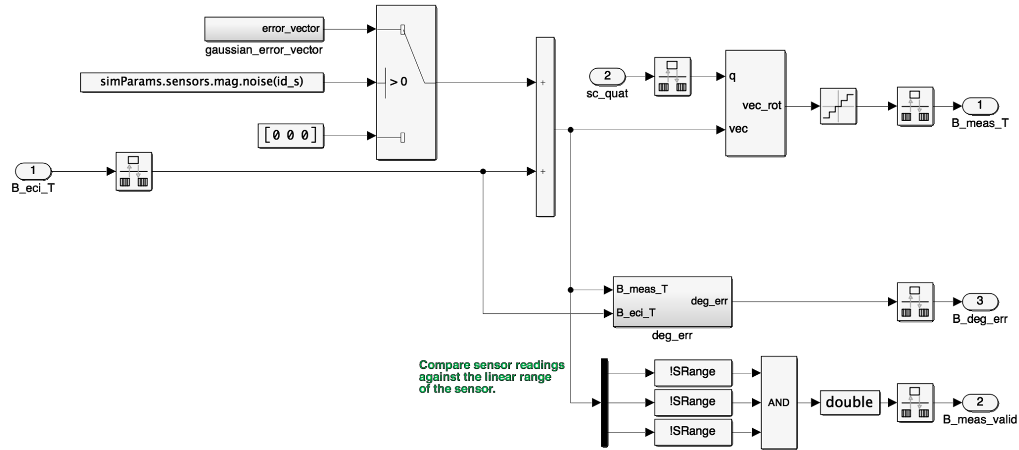

Figure 3.5: This is the SimuLink block diagram for the Magnetometer simulation library.

3.1.3 Gyroscope

Gyroscopes have a tendency to asymmetrically drift over time. Rather than just white noise, they have an error that grows linearly over time appearing as a biased random walk. This is known as the gyro bias. We model this by first applying a linear bias and adding white noise to that. This is then added to the measurement as the following [5],

$$\tilde{\omega} = \omega + \epsilon + \chi \tag{3.10}$$

$$\dot{\chi} = \epsilon \tag{3.11}$$

Where ϵ_a is known as the Angle Random Walk (ARW) and ϵ_r is known as the Rate Random Walk (RRW) [5]. By integrating the RRW and adding that to the

ARW you get the expected behavior of the gyroscope random walk. As before, ω represents our true spacecraft angular velocity with the $\tilde{}$ denoting a measurement of the vector. This can be found in our Sensor Lib and appears,

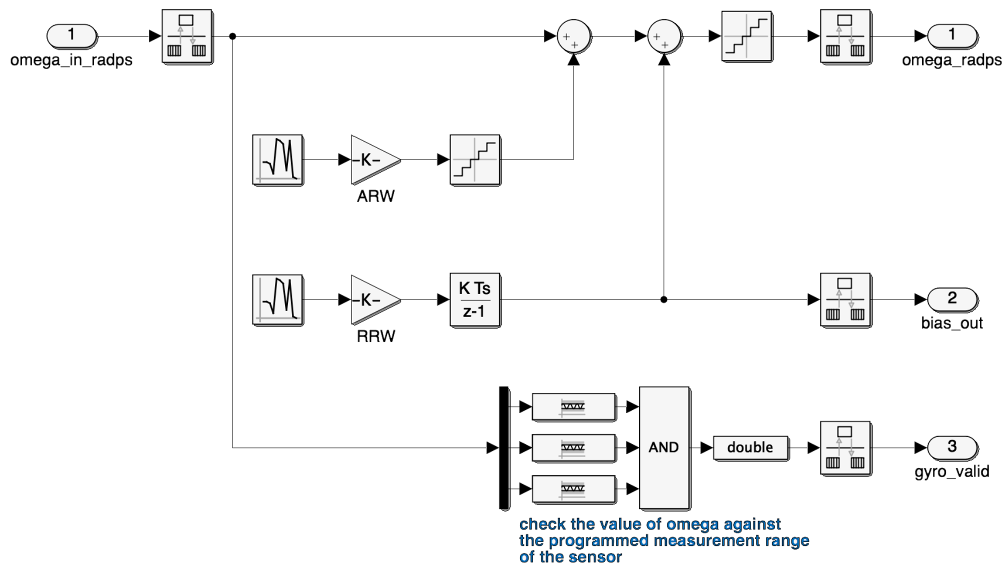

Figure 3.6: This is the SimuLink block diagram for the gyroscope simulation block.

There are also outputs to check the validity of the gyroscope as well as the current bias of the sensor due to the random walk.

3.2 Actuator Models

Onboard the satellite we have two sets of actuators: the Reaction Wheel Assembly (RWA), and the Magnetorquers (MTA). The RWA is in the category of actuators known as momentum exchange devices. These devices use the conservation of momentum in order to create internal torques that rotate the spacecraft. They work by spinning heavy (relatively) metal wheels very fast and creating a torque vector. This

comes at a price, however, for once you have spun the wheel up to generate the torque you must maintain that speed or else you will experience a torque in the opposite direction. This is due to the fact that you have merely exchanged momentum from the wheel to the body rather than generated an external torque. As the wheel spins back down that momentum must be transferred again. This causes a problem known as momentum saturation. Essentially these devices have some max speed (RPM) at which they can rotate. Once this maximum rotation rate is achieved, they cannot spin any faster. However, you can get around this by creating an external torque to "absorb" the stored momentum. This is where the magnetorquer assembly comes into place. Using the principles of a solenoid magnet, we create a magnetic field in a direction counter to Earth's magnetic field. These two fields will attempt to realign with one another and this in turn generates our external torque. The MTA are a set of magnetorquers contained within the solar panels themselves. This means, due to the fact that one face is taken up by the camera, we have five magnetorques present in the Assembly.

3.2.1 The Magnetorquers

As stated, the magnetorquers themselves are essentially solenoid magnets. We have what are specifically referred to as air-core solenoids (rather than a ferro-core). Behind the solar panels are five sets of PCB's with a thin copper trace in a loop pattern as seen in Figure 3.7

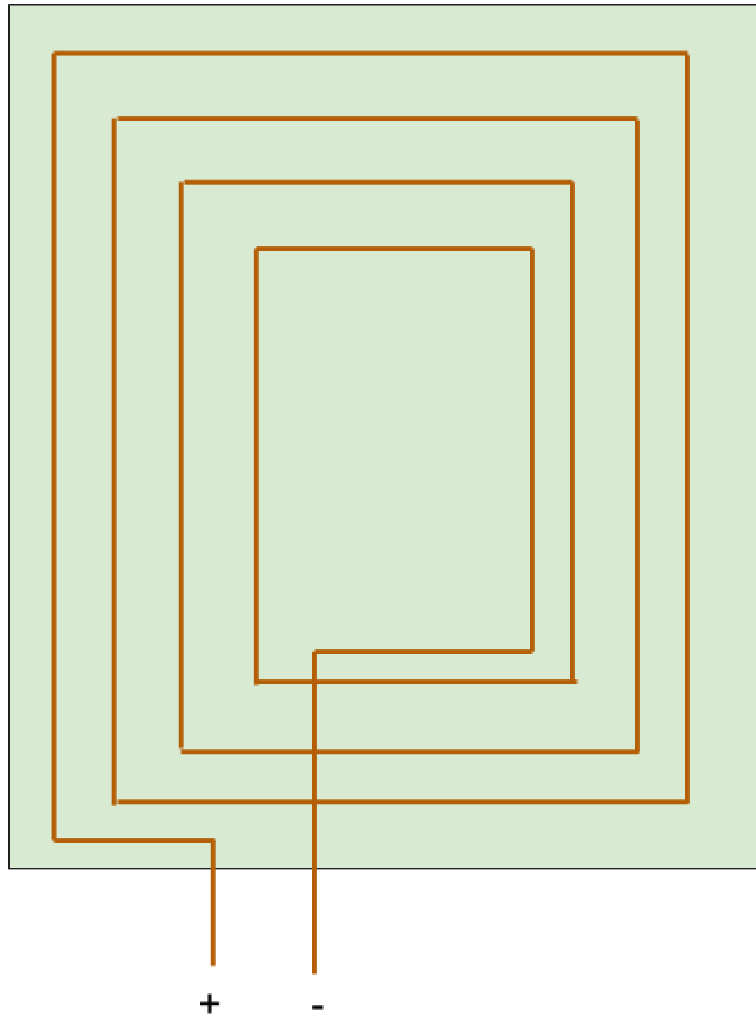

Figure 3.7: This diagram depicts the copper trace that makes up the air-core magnetorquer board.

The magnetometers are driven by a PWM signal that is generated by software onboard the OBC. The magnetic field or dipole moment generated by these devices is,

$$m_{mt} = NIA\hat{n} \tag{3.12}$$

where N is the number of loops, I is the current, A is the cross sectional area, and $\hat{n}$ is the normal vector pointing outwards. The torque generated by this dipole moment can be found through,

$$\tau_{mt} = m_{mt} \times b_B \tag{3.13}$$

where b_B is the surrounding magnetic field in the body frame. This is realized inside the actuator lib as the following,

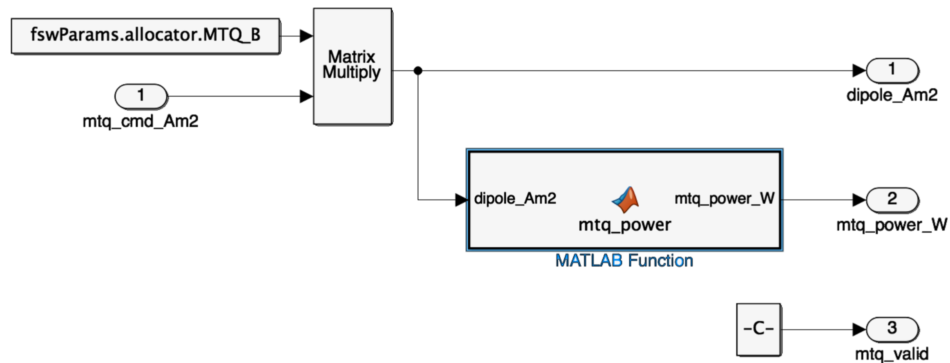

Figure 3.8: This library generates an external torque from the requested command according to the physics behind the MTA.

3.2.2 The Reaction Wheel Assembly

If you have ever done the physics demo where you sit in a swivel chair and spin a bike wheel, then you have some intuition for how the reaction wheel assembly works. If

you haven't had that experience then the previous sentence probably sounded pretty weird. From [5] and [2] we assume that the torque generated by the reaction wheel assembly is,

$$\tau_{RWA} = J_w\dot{\Omega} + c\Omega \tag{3.14}$$

Where J_w is the inertia of the wheel, Ω is the angular velocity of the wheel, and c is the coefficient of viscous friction. The RWA comes with an electronic speed controller (ESC) for each of the four wheels. These act essentially as PID controllers where an RPM is input and that speed is maintained by the ESC. We have modelled each wheel separately with individual PIDs and internal dynamics. These can be found in Acutator_Lib in the reaction_wheel_assembly_lib_disc. The library for each wheel appears,

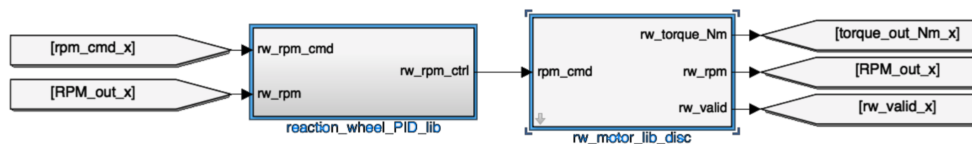

Figure 3.9: This library only models one wheel. We repeat this four times for our RWA.

In addition to modelling the physics of the wheels, this library also does some basic power modelling. The electrical power is calculated by,

$$\frac{\tau_{RWA} \cdot RPM}{\eta} = EP \tag{3.15}$$

where η is the mechanical efficiency of the wheels. In all, this library outputs: RPM, torque, power (W), and a validity check (to ensure it's within allowable RPM

range).

3.2.3 Allocation

The controllers generated commanded torques in three axes. However, due to the fact that we have 4 reaction wheels in our assembly, we need to project the 3-axis vector onto a 4-D actuator space. This process is known as actuator allocation. Essentially,

$$\tau_{rwa} = A\tau_{cmd} \tag{3.16}$$

Where A is some matrix that projects our commanded torque on the actuator space. One way of doing this projection is to simply collect the unit vectors corresponding to each of the reaction wheels and compile these into a matrix that will effectively dot product the torque and project the components into each direction. However, with a 4-wheel configuration there are some issues with this definition. This way of defining allocation inherently limits the summed potential of the wheels within the assembly. This is a result of the geometric space that this layout enforces, more details on this can be found in [9]. We have imposed a L_∞ based approach to reaction wheel allocation that solves this problem along with the second issue, desaturation. Due to the fact that there is overlap between the directions that these wheels face this transformation has a non-zero nullspace. That is to say that, when considering the momentum of the wheels, there are multiple speed configurations that would achieve the same net momentum. This can be a problem when we wish to desaturate the wheels as this may leave us with arbitrarily high RPM values while satisfying the “no net momentum” criteria. The L_∞ allocation scheme manages to separate the individual momenta of the wheels allowing us to drive them all to a

lower speed. As a bonus, this will also avoid the issue of possibly entering a “zero net momentum” configuration with one wheel having an RPM lower than the minimum operating speed of 1000 RPM. These details and more can be found in [9].

Chapter 4
SOFTWARE

This chapter will serve as an overview of the software which is not directly responsible for the control of the spacecraft. Specifically, we will be reviewing the attitude estimation schemes as well as how the overall system operates.

4.1 Estimation

In order to properly determine our attitude we must use some estimation routine. Because attitude cannot be directly measured, this method needs to remove noise from measurements as well as use a series of attitude estimation algorithms to provide us with accurate attitude knowledge. Keep in mind that the requirements we established in section 1.2 dictate our need to estimate our attitude to within 3.5° accuracy. To do this, we will employ two main algorithms,

1. TRIAD Filter

2. Multiplicative Extended Kalman Filter (MEKF).

As a note it is common to refer to estimation routines as Filters.

4.1.1 TRIAD Filter

The purpose of the TRIAD filter is to perform a rough estimation in order to provide a good initial guess for the MEKF. The TRIAD cannot satisfy the requirements on

its own, but in combination with our MEKF they can satisfy the 3.5° requirement as well as the time to converge requirement. Should you be unfamiliar with attitude estimation routine, the TRIAD filter is a very good starting place. It is fairly straightforward and performs the operations that underlie all attitude determination routines. Consider two sets of vectors R and r. Where R is a set of three vectors in some inertial frame and r is those same vectors but resolved in a body frame (ex: spacecraft coordinate frame). These two sets of vectors are related by the following equation,

$$\vec{R} = A\vec{r}. \tag{4.1}$$

Where A is a rotation matrix that transforms between the two coordinate systems. This matrix *is* our attitude. That is to say that attitude is a description of how we move from one reference frame to another, in this case a matrix transformation. In order to solve this problem, we need a minimum of three vectors on each side of the equation. However, by taking only two measurements we can use cross product to create a third. In our case, the two vectors we measure are magnetic field, $\vec{b}$, and sun vector, $\vec{e}$. This means our sets of column vectors are in the body frame,

$$\vec{r} = \left[\vec{b} \;\vdots\; \vec{e} \;\vdots\; (\vec{b} \times \vec{e})\right] \tag{4.2}$$

and likewise in the inertial frame,

$$\vec{R} = \left[\vec{B} \;\vdots\; \vec{E} \;\vdots\; (\vec{B} \times \vec{E})\right] \tag{4.3}$$

where $\vdots$ represents a concatenation of column vectors.

Remark 4.1.1. *Due to the fact that we are only able to measure the vectors in the body frame, we will be running simulation on board the satellite to generate the body frame vectors.*

These sets of vectors are enough to solve equation 4.1, but only in the noise-free case. We can improve the accuracy of this filter by using the following parametrizations:

$$\hat{S} = \frac{\vec{B}}{||\vec{B}||} \tag{4.4}$$

$$\hat{s} = \frac{\vec{b}}{||\vec{b}||} \tag{4.5}$$

and

$$\hat{M} = \frac{\vec{B} \times \vec{E}}{||\vec{B} \times \vec{E}||} \tag{4.6}$$

$$\hat{m} = \frac{\vec{b} \times \vec{e}}{||\vec{b} \times \vec{e}||}. \tag{4.7}$$

Thus, our system is now given by,

$$\left[\vec{S} \ \vdots \ \vec{M} \ \vdots \ (\vec{S} \times \vec{M})\right] = A \left[\vec{s} \ \vdots \ \vec{m} \ \vdots \ (\vec{s} \times \vec{m})\right] \tag{4.8}$$

Therefore the attitude matrix can be obtained by,

$$\hat{A} = \left[\vec{S} \ \vdots \ \vec{M} \ \vdots \ (\vec{S} \times \vec{M})\right] \left[\vec{s} \ \vdots \ \vec{m} \ \vdots \ (\vec{s} \times \vec{m})\right]^T \tag{4.9}$$

In this case $\hat{A}$ is our estimate of the attitude matrix. That concludes the TRIAD filter. From here we can convert the rotation matrix to a quaternion and this can be used as the initial guess for the MEKF.

4.1.2 Multiplicative Extended Kalman Filter (MEKF)

In this section I will layout the fundamental underlying equations of the MEKF in order to give a good understanding of the concepts. For a full derivation of the filter, reference [3], as well as the GNC-ICD written by the codes author, Kylle Ashton. The reader may be familiar with a Kalman Filter or an extended Kalman filter and in fact the MEKF is really no different. The differences here lie in the formulation of our components. One key difference is that the quaternion (or rotation matrix) world subtraction, which plays a key role in a Kalman Filter, is defined by multiplication. This is where the multiplicative term comes from in MEKF. In general a Kalman Filter is defined by the following set of equations: the noisy dynamics

$$x_k = A_k x_{k-1} + B_k u_k + w_k \tag{4.10}$$

$$y_k = H_k x_k + v_k \tag{4.11}$$

and the estimate,

$$\hat{x}_k^+ = \hat{x}_k^- + K_k(y_k - H_k \hat{x}_k^-). \tag{4.12}$$

The dynamics and measurements are effected Gaussian white noise, w_k and v_k, and we form some estimate $\hat{x}$ which is updated and corrected using the Kalman Gain

K_k. Here pluses and minuses define before and after the correction. The attitude determination scheme is, at it's core, no different. Within the whole of flight software, our state is considered to be $[r\ v\ q\ \omega]^T$, however, for the purposes of our estimator the estimated state will be $\hat{x} = [\hat{q}\ \hat{\omega}]$. The equations in this section are based on [3] as well as the breakdown in [5].

Our dynamics are nonlinear so we need to per approximation within the estimation itself, for this we have chosen a first order approximation of the error dynamics. We will be using $\delta\hat{\alpha}$ to denote the vector part of the error quaternion, this will be used as a state in the MEKF in place of $\hat{q}$ [5]. The MEKF does not estimate the angular velocity directly, instead we estimate the gyroscopic bias described in Section 3.1. For consistency with 3.1 we will use $\hat{\chi}$ to denote the estimate gyro bias. Similar to the result in [5] the discrete time 'k' subscript.

The full error model is given in continuous time by,

$$\Delta\dot{x} = F\Delta x + G\omega, \tag{4.13}$$

Where

$$\Delta x = \begin{bmatrix} \delta\hat{\alpha} \\ \delta\hat{\chi} \end{bmatrix}, \omega = \begin{bmatrix} \epsilon_a \\ \epsilon_r \end{bmatrix}, F = \begin{bmatrix} -\hat{\omega}^\times & -I_{3\times3} \\ 0_{3\times3} & 0_{3\times3} \end{bmatrix}, G = \begin{bmatrix} -I_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & I_{3\times3} \end{bmatrix}, Q = \begin{bmatrix} \sigma_a^2 I_{3\times3} & 0_{3\times3} \\ 0_{3\times3} & \sigma_r^2 I_{3\times3} \end{bmatrix}.$$

Where $\epsilon_a, \epsilon_r, \sigma_a, \sigma_r$ refer to the error quantities described in the gyroscope error modeling section of Section 3.1. Those quantities being the angle random walk and the rate random walk. Our measurement model is described by,

$$\tilde{y} = \begin{bmatrix} C(\hat{q}^-)b \\ C(\hat{q}^-)e \end{bmatrix} + \begin{bmatrix} \nu_1 \\ \nu_2 \end{bmatrix} := h(\hat{x}^-) + \nu, \tag{4.14}$$

Where $C(q)$ is the attitude matrix generated by the given quaternion , b is the magnetic field vector in the body frame, and e is sun vector in the body frame (both from measurements). Our sensitivity matrix becomes,

$$H(\hat{x}^-) = \begin{bmatrix} (C(\hat{q}^-)b)^\times & 0_{3\times3} \\ (C(\hat{q}^-)e)^\times & 0_{3\times3} \end{bmatrix}. \tag{4.15}$$

Given this formulation the Kalman update equations become,

$$K = P^- H^T(\hat{x}^-)[H(\hat{x}^-)P^- H^T(\hat{x}^-) + R]^{-1} \tag{4.16}$$

$$P^+ = (I - KH(\hat{x}^-))P^- \tag{4.17}$$

$$\Delta\hat{x}^+ = K(\tilde{y} - h(\hat{x}^-)), \tag{4.18}$$

With the main equations setting these apart from the standard EKF being,

$$\hat{q}^+ = (\hat{q}^- + \Xi(\hat{q}^-)\delta\hat{\alpha}^+)/||\hat{q}^- + \Xi(\hat{q}^-)\delta\hat{\alpha}^+|| \tag{4.19}$$

$$\hat{\chi}^+ = \hat{\chi}^- + \delta\hat{\chi}^+. \tag{4.20}$$

K is the Kalman gain and P is the covariance matrix. In addition, quaternion "subtraction" is defined by the multiplication of,

$$\Xi(q) = \begin{bmatrix} q_0 I_{3\times3} + q_v^\times \\ -q_v^T \end{bmatrix}. \tag{4.21}$$

After the previous update equations have been calculated, we then propagate the following quantities through a discrete time step,

$$P^- = \Phi P^+ \Phi^T + \Gamma Q \Gamma^T \tag{4.22}$$

$$\hat{\omega}^+ = \bar{\omega} - \hat{\chi}^+ \tag{4.23}$$

$$\hat{q}^- = \Omega(\hat{\omega}^+)\hat{q}^+. \tag{4.24}$$

The matrices Φ, Γ, Q, Ω are defined in both the appendix of [5] as well as Chapter 7 of [3]. The estimator can be found within flight software in "MEKF_lib" and appears,

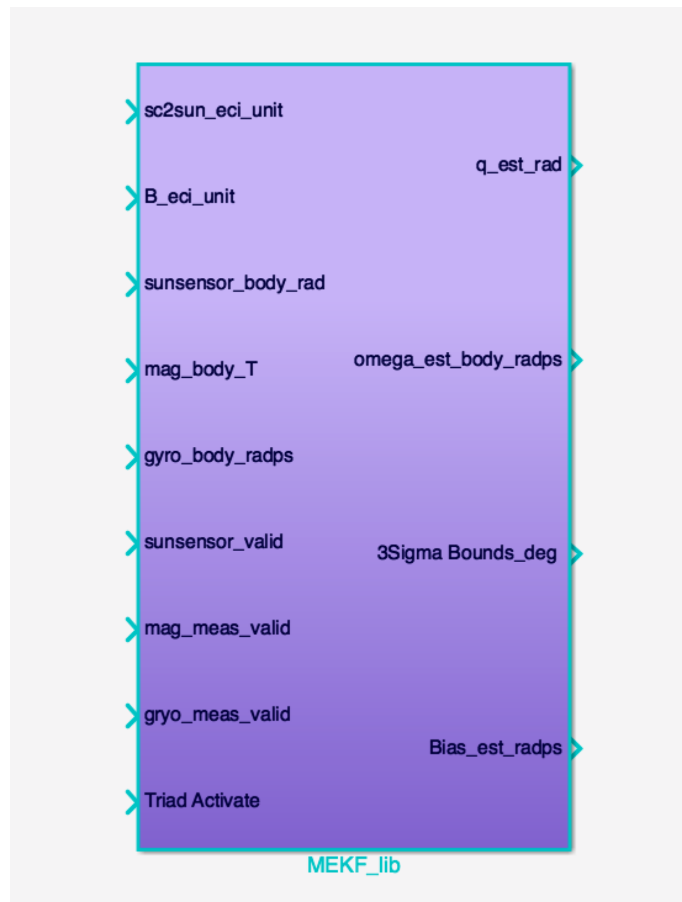

Figure 4.1: This figure shows the inputs and outputs of the MEKF library inside FSW. (with permission from Kylle Ashton)

Upon expansion we can see the implementation of TRIAD filter within the MEKF_lib,

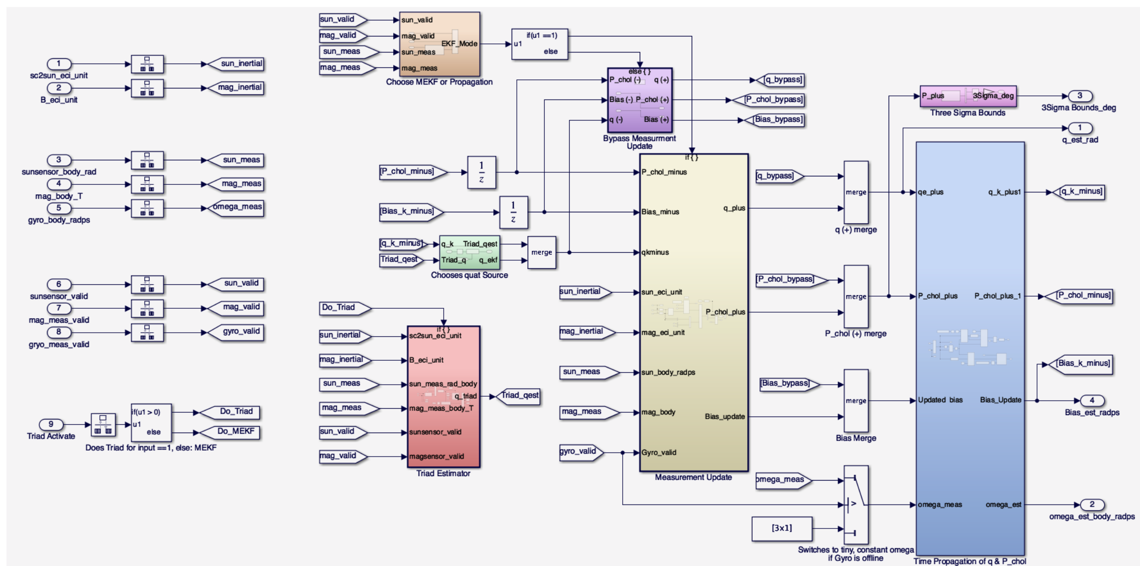

Figure 4.2: This library alternates between propagation and Kalman updates. Also see TRIAD in red. (with permission from Kylle Ashton)

4.2 *Operating Modes*

The complexity of this mission requires that we have designated various operating modes within which the spacecraft will be enabled to perform certain tasks. These modes are dictated by power, environmental circumstance, and directly by the ground station. Aside from the those operating modes dictated by the ground station these operating modes will be arrived at and transitioned from autonomously. This section will seek to lay out all of the modes as well as the transition triggers. For a more detailed description of this, one should reference GNC ICD where many of these images and tables were pulled from, courtesy of the one and only Cole Morgan [11].

4.2.1 Mission Mode Flow

In General there are three main operating regimes,

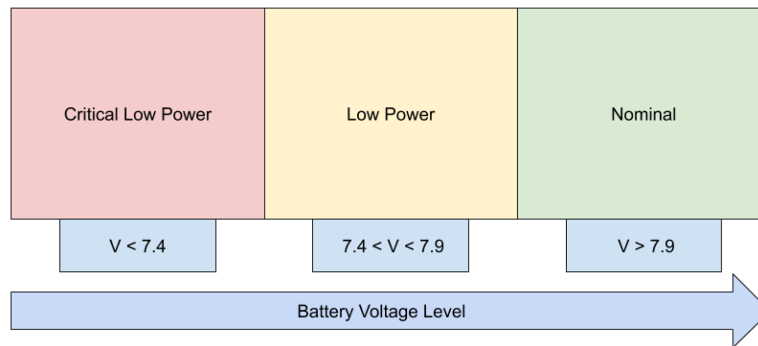

Figure 4.3: These three power modes dictate which parts of the system are allowed to activate and perform tasks.

While there are subdividing modes within each of these regimes, understanding these three is key to understanding the workflow of the mission. These modes are dictated primarily by the power condition of the battery. This is why we have a distinction of Critical Low Power and Low power. That distinction come directly from our specific EPS manufacturer. Figure 4.4 should provide a better understanding with the mode flow,

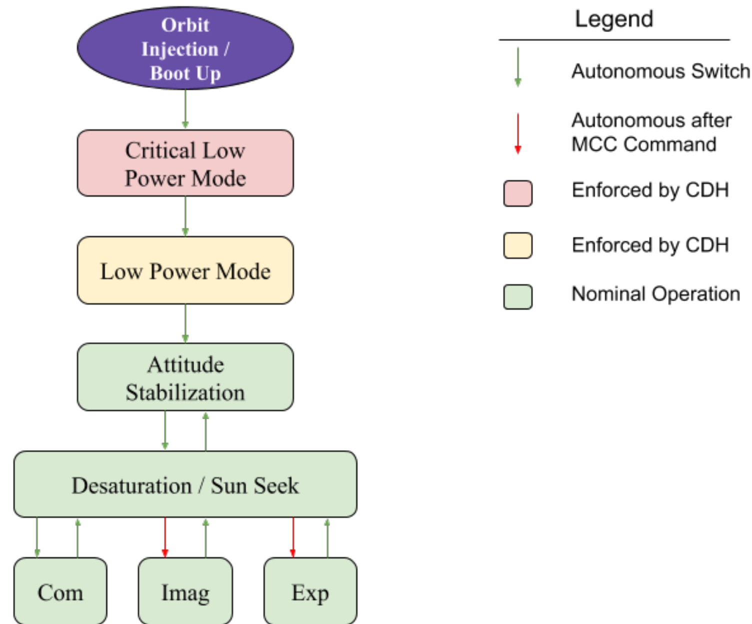

Figure 4.4: The arrows in this diagram represent the direction in which modes will switch autonomously.

Notice that the direction is always moving the system towards nominal mode. Any traversal up this tree will require a direct command from CDH.

Once the satellite boots up we enter into Critical Low Power Mode (CLPM). At this stage the craft does not have the power to run any of the GNC-specific tasks and therefore, from GNC's perspective, CLPM is essentially our "idle". The only way GNC leaves CLPM is when CDH issues a command to do so. Once the battery has been sufficiently charged EPS/CDH will issue the command to enter Low Power Mode (LPM). GNC begins attitude estimation once we have entered LPM. The reason why we want to estimate during LPM is to ensure we have some attitude knowledge when we enter Nominal Mode. Again, the transition from LPM

to Nominal will be autonomous and dictated by CDH. Once in Nominal Mode, the GNC software will be fully active and will begin to switch between all of the subdividing modes and tasks. For a quick breakdown we can reference Figure 4.5.

CDH	CLPM	LPM	Nominal					
GNC	CLPM	LPM	ASM	Desat	Sun Seek	Comm	Image	GNC-2
RWA	x	x	✓	✓	✓	✓	✓	✓
MTQ	x	x	x	✓	x	x	x	x
Sensors	x	✓	✓	✓	✓	✓	✓	✓

Figure 4.5: A red square with an x represents hardware or software being inactive as opposed to a green check meaning this software is allowed or on.

If we breakdown the various tasks within Nominal Mode we get

1. **Attitude Stabilization Mode (ASM)** - Prevents from spinning chaotically.

2. **Desaturation** - Unloads momentum stored in the reaction wheels.

3. **Sun Seek** - Keeps solar panels pointed optimally at the sun.

4. **COMM** - Orients the antenna at the ground station.

5. **Image** - Points the camera at a target.

6. **GNC-2** - Hands over trajectory generation to the payload.

The decision logic for these operating modes is housed within flight software in,

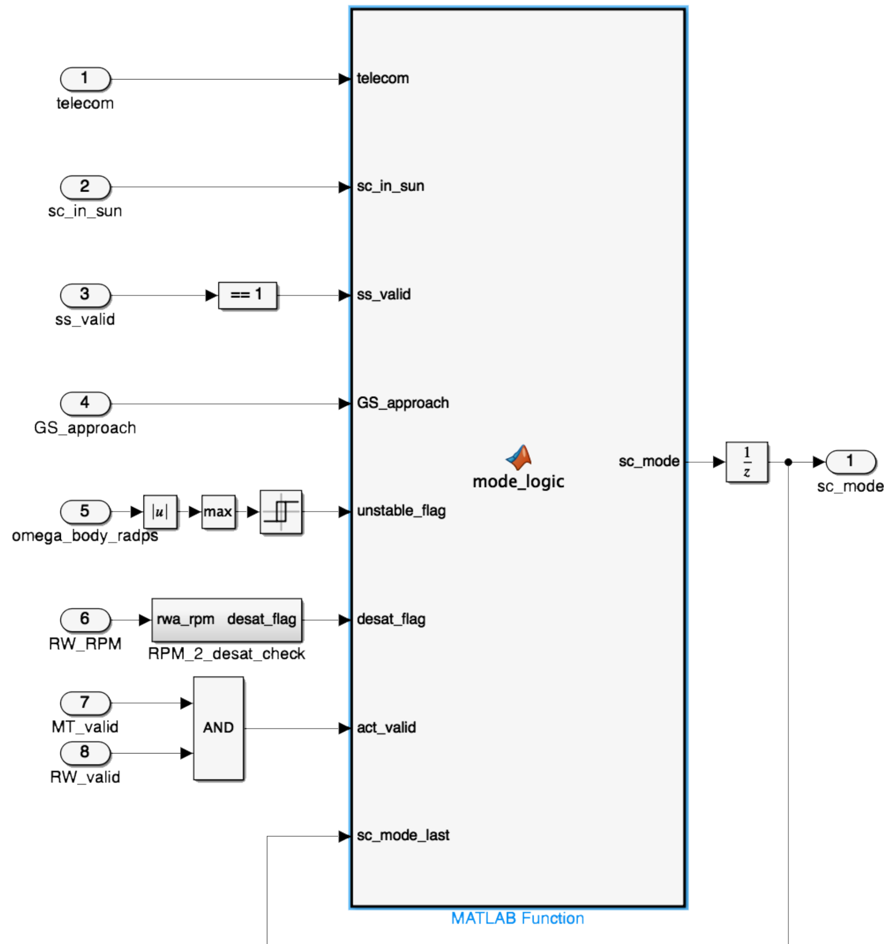

Figure 4.6: This should give you some idea of what goes in to computing the current mode.

The “telecom” input at the top is a direct command from CDH. This command tells the GNC system what mode to enter. All subsequent inputs are provided to check if that command is currently possible or when to begin to act on that command. We can understand how this checks occur by digging a bit deeper.

4.2.2 Control Operating Modes and Objectives

Within in each operating mode there are specific objectives. These objectives allow us to reach a target or generate a desired state which the controller will move us towards. A breakdown of these objectives can be seen in the following table,

Table 4.1: Attitude guidance pointing objectives.

Mode	**Attitude***	**Angular Rates**
ASM	n/a	$\boldsymbol{\omega}_{\mathcal{B}} \to 0$
Desaturation	$\hat{\boldsymbol{s}}_{ss,\mathcal{B}} \to \hat{\boldsymbol{s}}_{\mathcal{I}}$ $^*\boldsymbol{z}_{\mathcal{B}} \to \hat{\boldsymbol{r}}_{uw,I}$	$J_w^{-1}\boldsymbol{h}_w = \boldsymbol{\omega}_w \to 1000$ RPM $\boldsymbol{\omega}_{\mathcal{B}} \to 0$
Sun seek (tracking)	$\hat{\boldsymbol{s}}_{ss,\mathcal{B}} \to \hat{\boldsymbol{s}}_{\mathcal{I}}$ $^*\boldsymbol{z}_{\mathcal{B}} \to \hat{\boldsymbol{r}}_{uw,I}$	$\boldsymbol{\omega}_{\mathcal{B}} \to 0$
Pointing (COM)	$\hat{\boldsymbol{s}}_{ss,\mathcal{B}} \to \hat{\boldsymbol{s}}_{\mathcal{I}}$ $^*\boldsymbol{z}_{\mathcal{B}} \to \hat{\boldsymbol{r}}_{uw,\mathcal{I}}$	$\boldsymbol{\omega}_{\mathcal{B}} \to 0$
Pointing (EXP)	$-\boldsymbol{z}_{\mathcal{B}} \to \hat{\boldsymbol{v}}_{targ,\mathcal{I}}$ $^*\hat{\boldsymbol{s}}_{ss,\mathcal{B}} \to \hat{\boldsymbol{s}}_{\mathcal{I}}$	$\boldsymbol{\omega}_{\mathcal{B}} \to 0$

*Secondary pointing objective to be fulfilled as best as possible.

We see from Table 4.1 there are two types of pointing objectives: primary and secondary. The primary objective dictates the current mission objective. The most common secondary objective is to point the sun sensor at the sun, in order to maximize the incoming power to the solar panels. In an ideal world we would always be able to achieve both objectives, however, geometry won't always allow this.

To achieve this duality, these two objectives are hierarchical. We will always meet objective 1, while objective 2 will be to minimize the error between the two vectors. For example if we are pointing the COMM system at Seattle, we will simultaneously minimize the angle between the sun sensor bore-sight and the sun vector. Additionally, we will always attempt to lower the angular rate to zero. The

trajectory that is generated from these requirements will be an attitude quaternion. The logic for this lives within target_generation_lib,

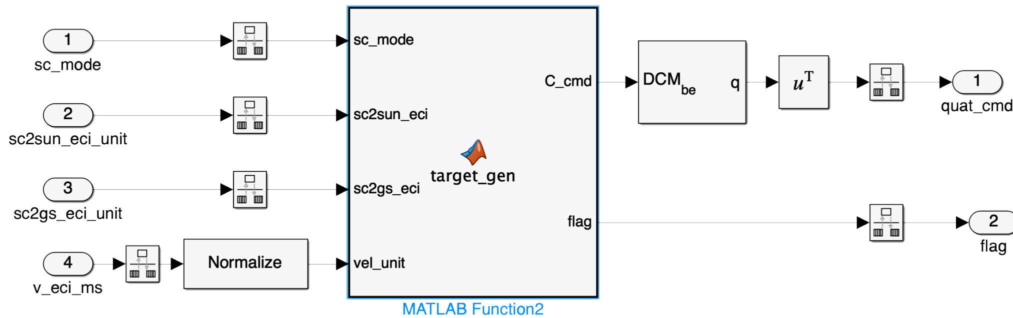

Figure 4.7: The Result that comes out of the actual Matlab function is a rotation matrix which is then converted to attitude quaternion.

The logic flow for the MATLAB function is essentially broken into two steps: first, perform a validity check on the sun vector and ground station vector (i.e. are they parallel?), and second, solve for the rotation matrix that takes you from the current set of vectors to the one satisfying the conditions. For a more detailed account see [11]. This logic is very similar to the math powering the TRIAD estimation routine, in that you use the difference in the two sets of vectors to generate a rotation matrix which is then converted to your desired quaternion.

4.2.3 Concept of Operations

For each of these operating modes there is a series of operations and modal changes that go along with each phase of the mission. In order to have these processes be both repeatable and well understood, we developed the Concept of Operations (CONOPS). The following flow chart shows the series of checks and balances that

lead to each of the global operating modes.

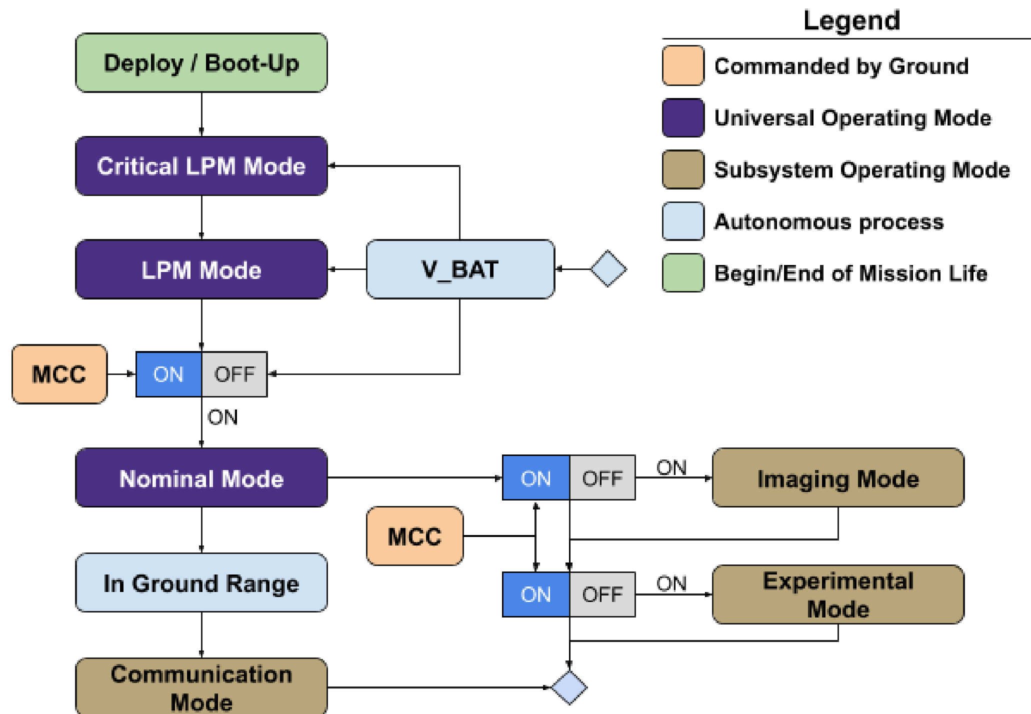

Figure 4.8: This gives slightly more detail to how we transition between universal operating modes and GNC-specific tabs. (Image given with permission by Taylor Reynolds [5])

The universal operating modes affect the CubeSat as a whole. These modes, set by CDH, dictate what GNC is allowed to do. Before we can enter nominal mode, the CubeSat EPS will need to absorb enough power to run all of the systems. Once this happens, the CubeSat will passively attempt to communicate with the ground station.

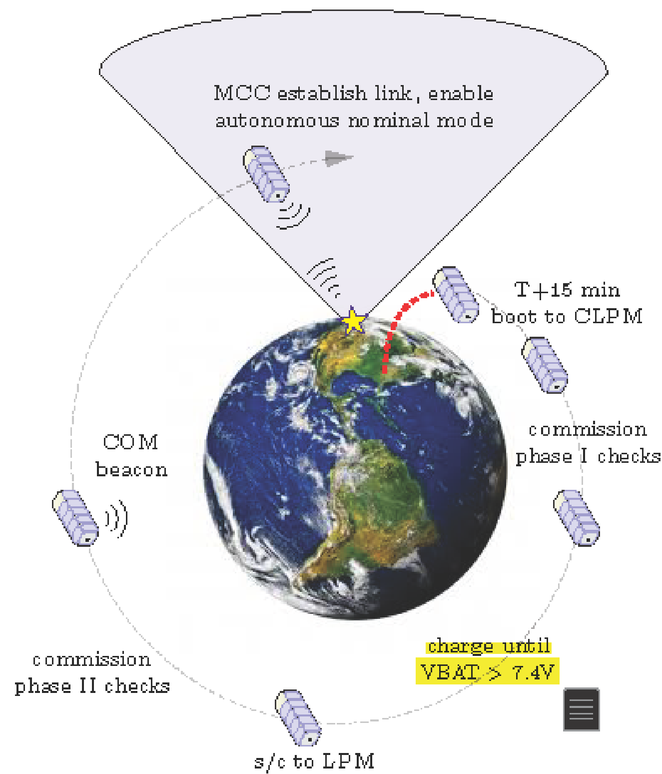

Figure 4.9: This images shows the various operations during the comission phase with time moving forwards as we move outwards on the spiral. (Image given with permission by Taylor Reynolds [5])

Most of this mission phase is passive with the only action being taken by the CubeSat being the COM beacon. The satellite will be in a tumble during this time and will only take action after a command from the MCC. Next is the the main Mission Phase.

Figure 4.10: This phase will (hopefully) be where the satellite spends the majority of its time. (Image given with permission by Taylor Reynolds [5])

In the mission phase, we will perform active detumbling, attitude determination, and attitude control, the key objective being the active uplink/downlink communication with the MCC. The data transfer requires active pointing for the link to be established. This active communication will allow us to issue a command moving the satellite into the imaging mode.

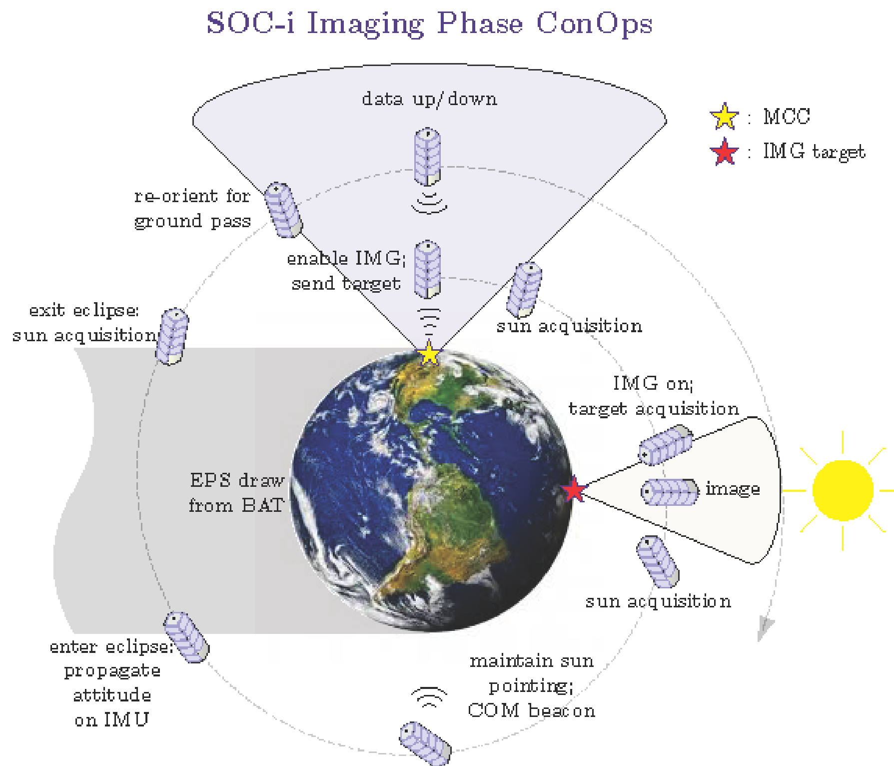

Figure 4.11: It is important to point out that the information determining the imaging orientation comes from a sim run on the MCC. (Image given with permission by Taylor Reynolds [5])

This mode will only be triggered when the environmental situation allows it. This is to say we can image when we have an upcoming ground pass with enough power and time to reorient in order to take a picture. Similarly, when circumstances allow, we will be permitted to enter the experimental phase. During the experimental phase we will activate the mission payload, which is an alternative guidance system.

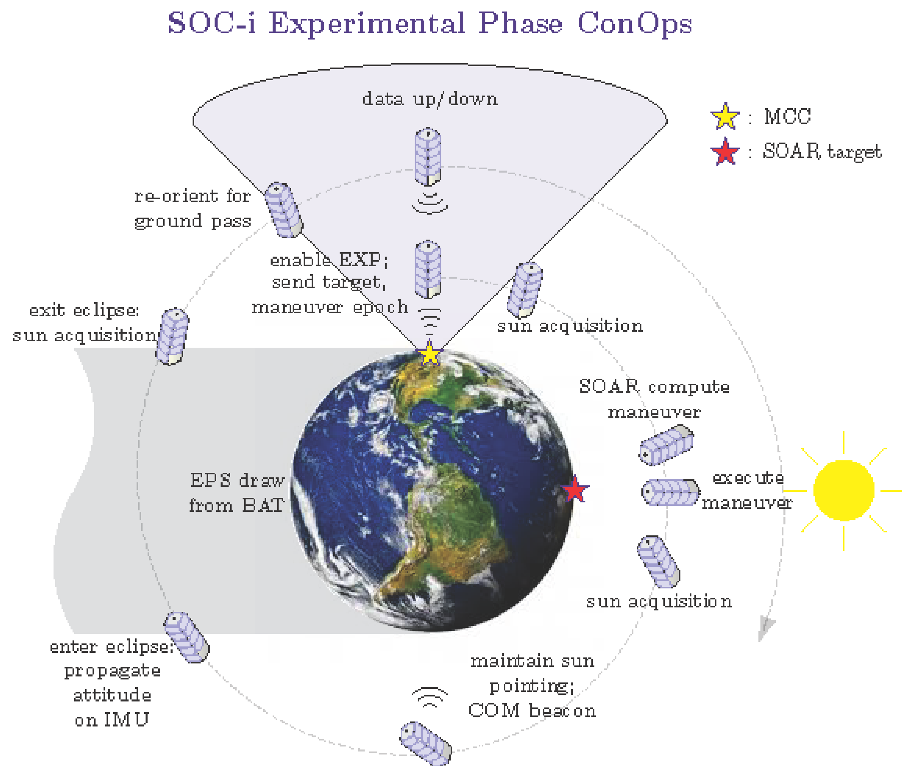

Figure 4.12: The experimental phase can be exited at anytime, should behavior become unexpected, and control will be returned to GNC-1. (Image given with permission by Taylor Reynolds [5])

Chapter 5
IMPLEMENTATION

In this chapter we will discuss some of the novel contributions I personally made for the implementation and testing of the software. The two main sections of this chapter discuss an automated system for building and testing code, as well as a discussion of the AACT Autocoder. This Autocoder was designed to generate C code based on our individual software unit tests.

5.1 Autocoding Pipeline

The purpose of the autocoding pipeline it to simplify the process of going from Simulink code to embedded c in order for it to be as user-friendly as possible. Simulink Autocode is essentially a build system within Simulink that allows the user to automatically generate a C/C++ version of the original Simulink model. Although MatLab Simulink is computationally limited, it allows one to focus on the system-to-system interaction of the code as well as the pure controls aspects, without requiring the learning of more complex C programming techniques. One large drawback of using Autocoding is that it can often create somewhat difficult to parse C libraries that, due to structure of Autocode, obscure the purpose of the C code through the generated filenames and header structure. This pipeline seeks to remove some of the need for a programmer to have to dig through this complex codebase and instead will create a more normalized structure. In addition, we have created a secondary autocoder to aid in the development of embedded unit tests.

In order to better understand where this fits in within our flight code verification, please reference our testing plan,

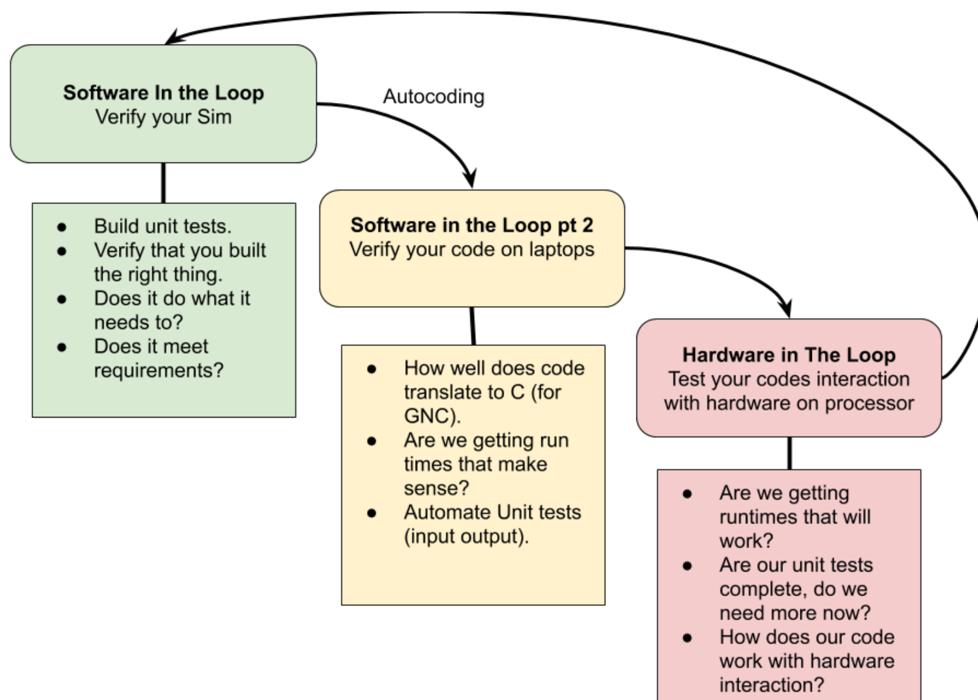

Figure 5.1: This process is iterative and may require several "Loops" through this flowchart.

At the core of our testing scheme is the unit test, which is a standalone implementation of the library that allows us to verify each bit of code in isolation. These unit tests are what will initially be autocoded, following that we will have C code verified first on laptops before moving to an embedded target. This should allow us to catch the majority of bugs before we move to the more difficult testing that needs to be done on the flight hardware. The final stage of testing, Hardware in the Loop (HIL), will require significant interactions with CDH, thereby requiring that

the code be thoroughly tested before progressing to this stage.

5.1.1 Concept Overview

In short, the Pipeline will take the user from Simulink to autocode to a standardized file structure complete with makefiles and a build system. This is further explained in Figure 5.2, which depicts graphically how this process takes place.

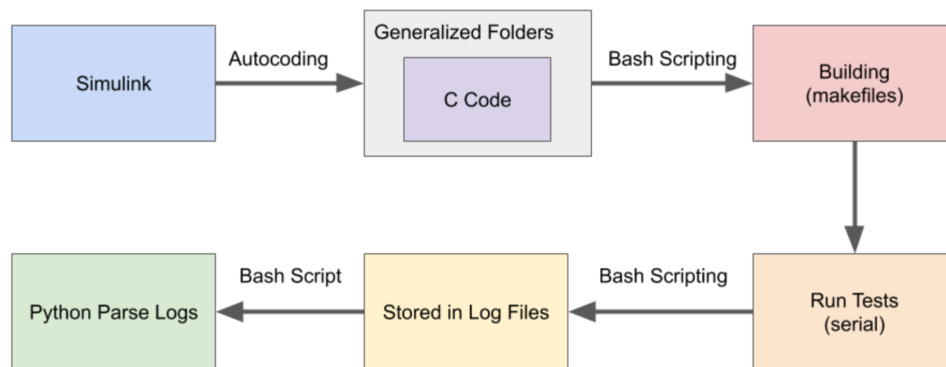

Figure 5.2: Bash scripts are utilized throughout the process as the main backbone of our automation.

Bash scripting is essentially a way of automating the commands the user would normally be entering into the command line. This means a series of commands like "cd" and "make" can be scripted to execute with only one bash command. This thesis will mostly focus on this Pipeline as way of building on a laptop. The set of bash scripts is diverse enough that, should the user only wish to run a portion of the

pipeline, the user can select which block they wish to run. The only non-automated section of the code is the Simulink Autocoding block. This will have to be set up by the user on their computer at the time of code generation.

5.1.2 Build System

Let's now go through how this works and how to use the pipeline. For a list of all of the possible commands in our toolkit run:

```
$ ./commands
```

This list should continue to grow as this tool matures. With this we can dig in to some of the specifics.

Folder Configuration

We have created a generic folder structure for any and all unit tests. This is to make it possible to automate the process of building and running by using a combination of bash scripts and generic MakeFiles. Fortunately, rather than trying to ensure every developer use the same exact setup, the process of organizing the folders has been automated. The basic structure is as follows:

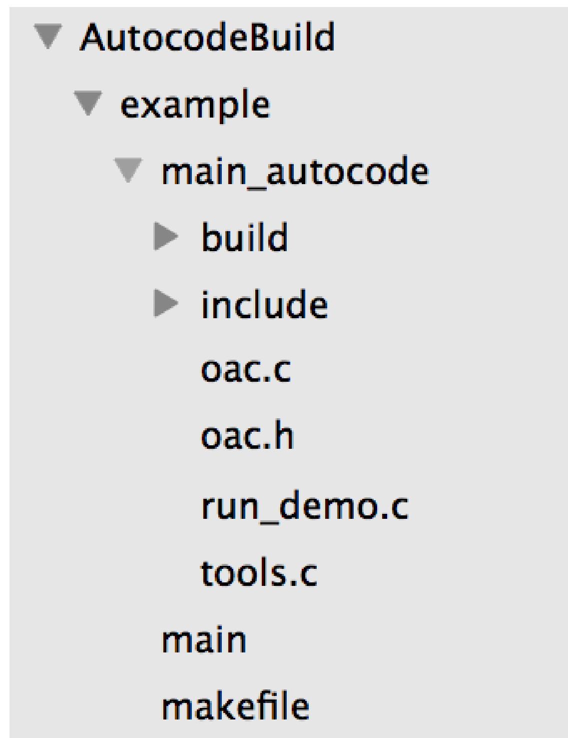

Figure 5.3: This example folder shows the template for the folder layout.

All autocoded files will be placed within the “AutocodeBuild” folder. The generated folder will naturally be quite messy so we rely on some scripts to clean things up.

To do this run the following command:

```
$ ./setup <code_folder>
```

With that in place, let’s go through what these folders are for.

- A generic **makefile** has been created to build our executable

- The **main_autocode** folder contains all of our .c and .h files

- Within **main_autocode** there are two sub-folders: **build** and **include**

- **Build** will house all of our .o files after compilation

- **Include** can house all of the header files should the user feel the need to use it. (depending on how many there are).

This folder structure should allow for any of the other tools present in the AutocodeBuild folder to work with any given code.

The AACT AutoCoder

When a Simulink unit test is Autocoded the result is a set of c files and header files. The test itself is not what is compiled, rather the underlying functions and libraries called within. However, without too much effort the user can recover the unit test as it was intended. If the Simulink Autocoder is run correctly the result will be an example main program titled, "ert_main.c". This is a boiler plate example main program which includes all of the proper header references and data types to run the test. The workhorses behind running any Simulink tests are the inputs and outputs represented by the two structs,

- **rtU** - this struct contains the inputs in the correct data format and sizes

- **rtY** - this struct contains the output in the correct data format and sizes

as well as the functions,

- **libraryname_init()** - this function initializes all of the structures and timers necessary to run the library

- **rt_Onestep()** - this function calls one step of the Simulink block.

In order for OneStep to run correctly the user will need the proper loop structure and variable population. This would involve either hard coding in all of the input values for each iteration of OneStep or a way of pulling in those inputs from a file. I elected to create a way to automate the second. To achieve this, I have written our own autocoder that uses Python to alter the generic "ert_main.c" and creates a more useful testing script. It does this by making use of a set of input and output text files that the user will need to generate and place within the autocoded folder. The input file is a CSV with the structure,

line 1: number of inputs
line 2: input names
line 3: input sizes
line 4: variable values for first iteration
line 5+: nth set of variable values

where the nth set would represent the input values at time t. For example here is the input.txt for the SGP4 autocoded test,

```
1   3,
2   JD_utc_J2000,orbit_tle,teme_to_gcrf,
3   1,9,9,
4   float,float,double,
5   7220.9952105745,19.0,7220.9945161301,0.0001027,51
```

Figure 5.4: In this image of an example input text file, the last line has been cropped off.

This is mirrored by an output text file containing the output of the **Simulink** unit test. We do this so that we can compare the output of the C version to the output of the Simulink version. This simplifies the code we have to automate because we no longer have to perform additional calculations to check the validity of the output. Instead, we can simply see if this output matches the already verified Simulink output. Think of it as the transitive property of unit testing. So, the autocoder is writing two types of code,

1. Code that reads the inputs, populates variable for this iteration of the test, and run the test

2. Code that read in the outputs and compares the results.

I have also created this diagram to help with understanding this process,

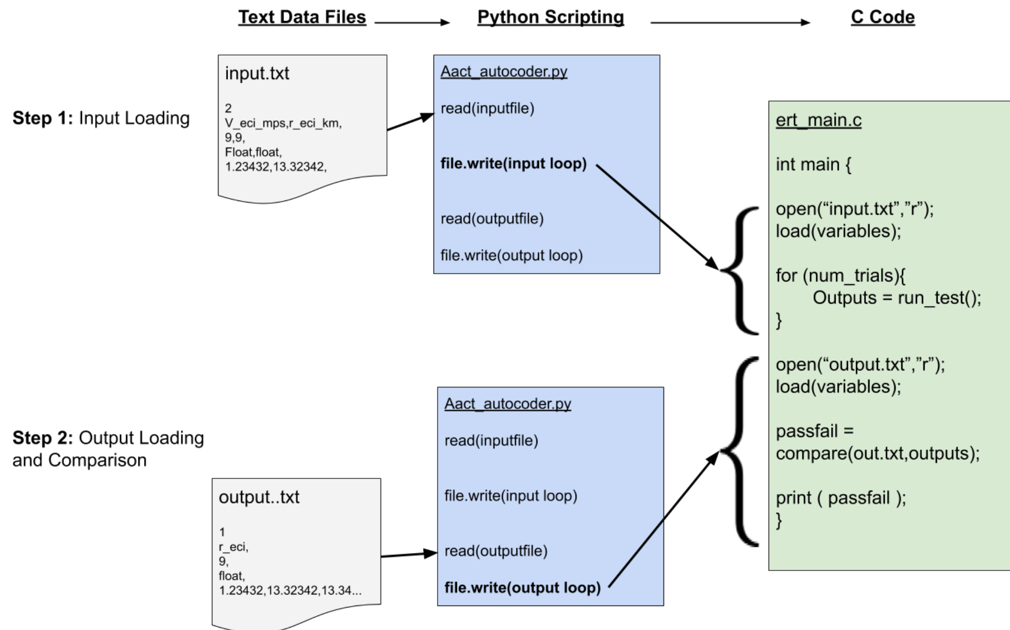

Figure 5.5: The left column represents the input/output text files, the middle is the python script, and the right is the C file getting edited.

The only argument that needs to be passed to the autocoder is the name of the folder, which should mirror the name of the initialization function. Additionally, if things are not working as they should, the user can pass a "-d" flag to get some more debugging outputs. An example output can be seen here,

```
➜ SGP4 git:(dev_dev) ✗ cd main_autocode
➜ main_autocode git:(dev_dev) ✗ python aact_autocode.py
~~~~~~~~~~~~~~~~~~~~
 Reading input.txt
~~~~~~~~~~~~~~~~~~~~
- -
Info Obtained From Input.txt:
----------------------------
Line count: 5
Variable Count: 3
Variable Names: ['JD_utc_J2000', 'orbit_tle', 'teme_to_gcrf']
Variable Size: [1, 9, 9]
Number of Trials: 1
Variable Types: ['float', 'float', 'double']
~~~~~~~~~~~~~~~~~~~~
Stripping ert_main.c
~~~~~~~~~~~~~~~~~~~~
start Strip
End strip
~~~~~~~~~~~~~~~~~~~~~~~~~
 Creating Variable Lines
~~~~~~~~~~~~~~~~~~~~~~~~~
19
the parse line is:
fscanf(fp,"%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,
it_tle[6], &rtU.orbit_tle[7], &rtU.orbit_tle[8], &rtU.teme_to_gcrf[0], &rtU
teme_to_gcrf[8]);

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 Adding Variable Population to ert_main.c
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Adding Code
➜ main_autocode git:(dev_dev) ✗ cd ..
➜ SGP4 git:(dev_dev) ✗ make
gcc -c -o main_autocode/build/ert_main.o main_autocode/ert_main.c -I../std_
gcc -o main main_autocode/build/sgp4_lib_fsw0.o main_autocode/build/ert_mai
➜ SGP4 git:(dev_dev) ✗ ./main
rtY.pos_eci_m[0] = -6229688.849031401798
rtY.pos_eci_m[1] = 2943455.957642101217
rtY.pos_eci_m[2] =  373453.523682077532
rtY.vel_eci_mps[0] =   -1573.507944604490
rtY.vel_eci_mps[1] =   -4435.787988992482
rtY.vel_eci_mps[2] =    5996.478176598727
rtY.SGP4_FLAG = 0.000000
➜ SGP4 git:(dev_dev) ✗
```

Figure 5.6: This is the autocoder run with the debugging flag and a printed output (rather than compared).

The portion of this script that generates the actual c-code is relatively straight forward so, with the help of this document, we can continue to add features as they are needed. Hopefully one day this can be helpful to the other subsystems as well.

Building and Running

In order to build a specific test within the folder, all that needs to be done is to run

```
$ make
```

and

```
$ ./main
```

For the purposes of testing, however, it may be incredibly useful to build and log all the outputs of a given autocoded test. For this we have a created a tool that will allow the user to log the data as well as build for a specific piece of hardware (for example the NXP).

For building on a laptop the user need only enter:

```
$ ./build <code_folder> <log output format (.log .csv etc)>
```

If the user omits the log output format, the result will just be a .log folder with all of the print statements dumped. Keep in mind that should the use run the test multiple times, the log will be overwritten.

All of this makes up the core of the autocoding pipeline. I'm sure as time goes on students will continue to add features and maybe we can get the to the point where we can add the aact_autocoder as a custom plug-in to the Simulink Autocoder itself. For now I will leave the user with this document, as well as the surrounding documentation within the repository. So good luck and good coding!

Chapter 6
CLOSING REMARKS

I feel it's important to take a moment to look at the time spent in development of this system. In just under two years, we were able to go from nothing to a full CubeSat complete with a GNC system capable of deploying and testing a novel guidance method. This was all done by a team of engineers who were, at the same time as developing, learning the software and tools as well as developing the theoretical background necessary to build this system. This goes to show the power of the CubeSat design envelope. With only a handful of engineers, we are able to utilize this platform to test out novel controls experiments and hopefully one day make a significant contribution to spacecraft development. This thesis should hopefully serve as a guide for future master's students who may be looking to build a system of their own. Our system architecture can serve as basis for any experiment to build on. Whether that payload be a guidance system or something even more ambitious, having a solid GNC system is critical to any spaceflight mission.

6.0.1 Future Work

As it stands now, we have been accepted as a launch applicant by NASA. Our earliest possible launch date will June 1, 2021, which leaves us a year to pull together the satellite and do all necessary testing and confirmations to ensure a successful mission. The Autocoding Pipeline seeks to facilitate part of that process, but there is still a lot of work to be done on the physical testing of our system.

A major issue facing CubeSat teams is the limitation that comes with being Earth-bound. If we were to hook up our sensors and turn on the satellite in the lab, our system would not be able to make heads or tails of the inputs. There are two schools of thought that allow us to bypass this issue. The first is to create a laboratory setting that closely mimics the environment in space. This usually involves physical simulators, such as a Helmholtz cage to simulate magnetic field, or an air-bearing table to allow free rotation. This setup, while hopefully something we can eventually work towards, is expensive and time intensive to set up. The second way to way to do this testing is to take advantage of the high fidelity software sim we have already written. I have written a full proposal of this method which can be found in our club's resources, but I will summarize a brief section of it here.

The next big step in CubeSat development is to create what we call the "FlatSat". This layout will esentially be a large PCB in which we can socket each of the CubeSat components. This means we will be able to test the hardware-software interaction between each of the subsystems. Critical to us is the interaction between CDH and GNC. The Simulink code provides us with both environmental and sensor models. These models are built to recreate the output of the sensors. The actual reading and translation of the sensors will be performed by CDH. All of our sensors use an I2C serial interface which is not modelled in Simulink. The goal is to determine a way to feed the simulations sensor outputs to CDH and have that be what informs the Software on FlatSat. This will require the use of embedded devices to translate the data coming from Simulink to the I2C data that can be interpreted by the on board computer. An illustration of this setup can be seen here,

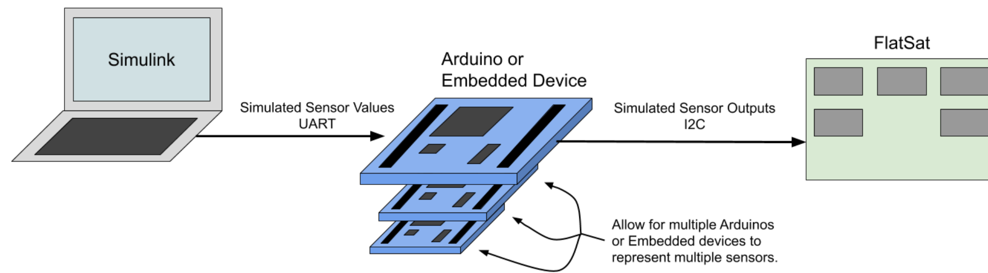

Figure 6.1: Multiple embedded devices may be needed to supply multiple I2c addressees.

This test plan should allow us to run whatever scenario we are interested in just by changing the initialization parameters. This should allow us to catch any bugs as well as iron out any issues with hardware-hardware interaction. There is still a long way to go from here, but I think the work that we have done thus far and the structures that we have put in place will allow us to not only meet our design schedule but will lead to a successful mission. This flight will be a test not only of the overall system architecture but will allow us to give flight heritage to one of the novel and exciting algorithms developed by our department here at the University of Washington. This has been a fantastic journey and I am thankful to all of those who helped me along the way.

BIBLIOGRAPHY

[1] D. A. Vallado, and W. D. McClain, *Fundamentals of Astrodynamics and Applications*. New York: McGraw-Hill Companies, Inc, 1997.

[2] M. J. Sidi, *Spacecraft Dynamics and Control: a Practical Engineering Approach* Cambridge Univ. Pr., 2006, pp. 191–193.

[3] J. L. Crassidis, J. L. Junkins, *Optimal Estimations of Dynamic Systems*, 2nd ed., Chapman and Hall/CRC Applied Mathematics and Nonlinear Science Series, Boca Raton, 2012, pp. 456–460.

[4] J.T. Wen, & K. Kreutz-Delgado, " The Attitude Control Problem " *Transactions on Automatic Control*, IEEE Transactions on 36. 10.1109/9.90228.

[5] T. P. Reynolds, K. Kaycee, B. Barzgaran, M. Hudoba de Badyn, S. Rice, E. Hansen, A. Adler, B. Acikmese and M. Mesbahi, " Development of a Generic Guidance Navigation & Control System for Small Satellites: Application to HuskySat-1", presented at *AIAA Space Forum*, Orlando, FL, 2018.

[6] E. Thébault, C.C. Finlay, C.D. Beggan, et al. "International Geomagnetic Reference Field: the 12th generation." Earth Planet Sp 67, 79 (2015). https://doi.org/10.1186/s40623-015-0228-9

[7] Wie, Bong & Lu, Jianbo. (1995). "Feedback control logic for spacecraft Eigenaxis rotations under slew rate and control constraints." *Journal of Guidance Control and Dynamics* - J GUID CONTROL DYNAM. 18. 1372-1379. 10.2514/3.21555.

[8] Y. Kim, & M. Mesbahi, " Quadratically Constrained Attitude Control via Semidefinite Programming. " *Transactions on Automatic Control*, IEEE Transactions on. 49. 731 - 735. 10.1109/TAC.2004.825959.

[9] R. Reynolds & L. Markley, " Maximum Torque and Momentum Envelopes for Reaction Wheel Arrays." *Journal of Guidance, Control, and Dynamics*. 33. 10.2514/1.47235.

[10] A. Domahidi, E. Chu and S. Boyd, "ECOS: An SOCP solver for embedded systems", presented at the *2013 European Control Conference (ECC)*, Zurich, 2013, pp. 3071-3076, doi: 10.23919/ECC.2013.6669541.

[11] C. Morgan, T. Reynolds, D. Tormey, *SOCI-GNC-001 GNC Interface Control Document*, unpublished. (can be found on AACT github)

[12] T. Reynolds, *SOAR Interface Control Document*, unpublished. (found on soci-gnc github)

[13] J. R. Wertz,, D. F. Everett, and J. J. Puschell. *Space Mission Engineering: The New Smad.* Hawthorne, CA: Microcosm Press, 2011. Print.

[14] A. H. DeReuter, C. J. Damaren, and J. R. Forbes, *Spacecraft dynamics and control: an introduction.* Chichester, West Sussex, United Kingdom: Wiley, 2013.