The Thesis committee for Andrew John Fear
Certifies that this is the approved version of the following thesis:

CubeSat Autonomous Rendezvous and Docking Software

APPROVED BY

SUPERVISING COMMITTEE:

E. Glenn Lightsey, Supervisor

Wallace Fowler

CubeSat Autonomous Rendezvous and Docking Software

by

Andrew John Fear, B.S.As.E

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2014

To my parents and family for always supporting me.

Acknowledgments

Dr. E. Glenn Lightsey for giving me the opportunity to work in the TSL and helping me grow as an engineer.

Leslie Roberts, my best friend, who has always been there for me when I am struggling with her support.

Jenna Gates, for continually pushing me forward and keeping me on track.

Henri Kjellberg, for all of his encouragement and creating the TSL into what it is today.

Katharine Brumbaugh Gamble, for always listening to my concerns and her advice.

Parker Francis, Karl McDonald, and Kenny Donahue, for being with me through late nights and tough times.

Terry Stevenson, for reading through my papers and appreciating my bad jokes.

CubeSat Autonomous Rendezvous and Docking Software

Andrew John Fear, M.S.E.
The University of Texas at Austin, 2014

Supervisor: E. Glenn Lightsey

An autonomous mission manager is being developed for use on CubeSats to perform proximity operations with other vehicles. The mission manager software is designed to run in real-time on a microprocessor used on a CubeSat. A simulation tool was developed that provides orbital dynamics and sensor measurements to test the mission manager software. A scenario was developed to demonstrate the control of a spacecraft from 1 km to 1 m to a target vehicle. Two small satellites were simulated in near-circular orbits around Earth at an approximate 400 km altitude. Each satellite is incorporated with simulated sensors and a Kalman filter. The simulation tool includes models for accelerometers and Global Positioning System receivers. Noise corruption is added to the modeled sensors to simulate imperfect knowledge. The simulation environment is capable of modeling Earth as a spherical or non-spherical body with spherical gravitational harmonics. Simulation parameters, such as the vehicle's initial states, Earth gravity model, and sensor noise are easily changed without recompiling the program through a simulation input file.

Table of Contents

List of Figures

Chapter 1

Introduction

Utilizing small satellites for performing research or commercial missions has become an increasingly prevalent part of the space industry within the last decade. With this rise in popularity, these small satellites, known as CubeSats, need technological advancements to meet increasing mission demands and requirements. For example, previous research into CubeSat control systems has resulted in 3D cold-gas thrusters for translational and rotational motion on interplanetary CubeSats [1] and full 6 degree-of-freedom constrained attitude guidance and control systems [2].

One area of interest is enabling CubeSats to perform safe proximity maneuvers relative to other vehicles autonomously. Performing autonomous relative maneuvers for small satellites has been recognized as a critical enabling technology by the NASA Technology Roadmap (TA04 and TA05) [3]. The CubeSat Autonomous Rendezvous and Docking Software (CARDS) project is being developed as an onboard software guidance manager to run on a CubeSat's flight computer. CARDS provides onboard guidance and failure identification, taking any necessary corrective actions autonomously. The focus of this research is performing a guidance maneuver that brings a controlled satellite, called the chaser, from a 1 km distance to 1 m of the desired rendezvous vehicle, known as the target. However, the goal of this research is not to re-invent algorithms suitable for proximity maneuvers. Instead, established algorithms have been selected and implemented for autonomous rendezvous operations.

The CARDS project consists of two parts: an environmental simulation and the mission manager software. The environmental simulation is an analytical setting to test the mission manager software. A rendezvous scenario with chaser and target satellites 1 km apart in orbit around Earth has been developed using the environmental simulation. The mission manager can be considered the core of the CARDS project, as this is where the autonomy and guidance portion of the system resides.

1.1 The CubeSat Standard

A CubeSat is any small satellite that adheres to the CubeSat standard established at the California Polytechnic State University (Cal Poly). CubeSats are composed of cubes where each cube is denoted as 1-unit or 1U. According to the CubeSat standard, each unit has 10 cm sides with a maximum weight of 1.33 kg [4]. CubeSat sizes range from 1U, 3U, and even up to 6U satellites. CubeSats have the advantage of being relatively small and cheap compared to large satellites. Typically, CubeSats are launched as groups of secondary payloads instead of on their own rocket. As a result, it can be difficult to obtain an orbit with desired parameters, as the orbit achieved is dependent on the primary payload's orbit requirements. A 3U CubeSat developed by the TSL in conjunction with the Jet Propulsion Laboratory (JPL) is shown in Fig. 1.1.

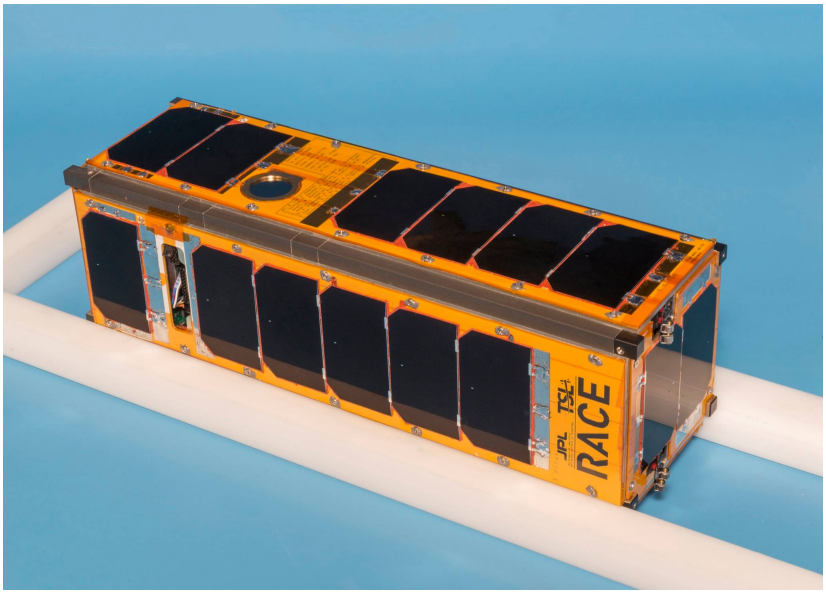

Figure 1.1: The RACE 3U CubeSat [Photo Credit: JPL]

1.2 Contribution

Due to CubeSats being small, cheap, and easy to develop, one attractive option for use is multi-vehicle constellations. These constellations would require the constellation satellites to perform relative proximity maneuvers with respect to each other. The development of CARDS would provide a software mission manager to perform these maneuvers autonomously. Eventually, the mission manager software will become part of NASA's "Autonomous Rendezvous and Docking (AR&D) Warehouse." [5] Operational proximity maneuvers would generate more uses available to CubeSats, such as on-orbit satellite inspection, servicing, and repair. An assembly of large structures using CubeSats as building blocks is also a possibility through this research.

1.3 Thesis Organization

This thesis is divided into 6 chapters, each focusing on a different part of the CARDS project. Chapter 2 is an overview of the CARDS project's motivation in relation to the satellites developed at the TSL. Chapter 3 discusses the details of the models used for the environmental simulation. In addition, the Trick and JEOD software tools used to create the environmental simulation are introduced. A description of the JEOD classes used in the simulation are provided. Chapter 4 is dedicated to the mission manager software. An explanation of the mission manager design and testing plans are presented. Results from running the environmental simulation are shown in Chapter 5. Finally, Chapter 6 concludes this thesis with an explanation of future work.

Chapter 2

Motivation

Considering that the CubeSat industry is in its infant stage, it is an exciting area of space vehicle research and development. The advancement of CubeSat technology would culminate in more accessibility to space technology. CubeSats are a great way for universities and companies to begin their own space vehicle development within their respective programs, due to their small size and relatively cheap cost. The growth of the CubeSat industry within educational and commercial settings will result in further development of space technology, and more research opportunities for CubeSats will arise.

2.1 Texas Spacecraft Laboratory

The Texas Spacecraft Laboratory is a funded research laboratory that is a part of the Aerospace Engineering and Engineering Mechanics Department at The University of Texas at Austin (UT Austin). The TSL currently has a focus on small satellite development, utilizing an undergraduate and graduate student workforce. Four satellites have been completed by the TSL. The Bevo-1 satellite was part of the Low Earth Orbiting Navigation Experiment for Spacecraft Testing Autonomous Rendezvous and docking (LONESTAR) program in conjunction with NASA's Johnson Space Center (JSC) and Texas A&M University. Bevo-1 was a cube with 5 inch sides that was launched in May 2009 aboard the Space Shuttle Endeavor. Two spacecraft collectively known as FASTRAC (Formation Autonomy Spacecraft with Thrust, Relnav, Attitude, and Crosslink) were launched on the STP-S26 in

Fall 2010. The fourth satellite built by the TSL was a collaboration with JPL called the Radiometer Atmospheric CubeSat Experiment (RACE) satellite, which was a 3U CubeSat that was lost on the failed Cygnus Orb-3 launch in October 2014. The TSL designed and built the RACE structure and bus to fit with the radiometer instrument provided by JPL.

Two satellites are in currently in development at the TSL. Bevo-2 is a part of the second mission of the LONESTAR program. It is a 3U CubeSat that is equipped with a six degree-of-freedom guidance navigation and control (GNC) system and features a 3D printed-cold gas thruster. Bevo-2 is expected to launch in the year 2015. The ARMADILLO (Atmosphere Related Measurements And Detection of submILLimeter Objects) satellite is a 3U CubeSat that won the Air Force Research Laboratory's University Nanosat Program 7 competition in December 2012. The primary payload of ARMADILLO is a piezo-dust detector (PDD), developed by Baylor University, that will measure impacts from small space debris less than 1 mm in diameter. ARMADILLO also features a dual frequency GPS receiver designed by the Radionavigation Laboratory at UT Austin which will perform GPS radio occultation experiments. The ARMADILLO satellite is expected to launch in early 2016.

2.2 Bevo-2 and LONESTAR

One goal of the Bevo-2 satellite for the LONESTAR-2 mission is to demonstrate the use of a full 6 degree-of-freedom GNC module on a CubeSat. The attitude determination and control (ADC) system consists of three reaction wheels, three gyroscopes, two magnetorquers, two sun sensors, one magnetometer, a GPS receiver, a star tracker camera, and a 3D printed cold-gas thruster. LONESTAR-3's mission is the demonstration of an AR&D operation between to-be-developed satellites by UT Austin and Texas A&M.

The mission manager software developed through CARDS will be flown on the Bevo-2 satellite in preparation for the LONESTAR-3 mission. The mission manager software will perform the AR&D maneuver onboard UT Austin's LONESTAR-3 satellite. If successful, this mission will be a critical step forward in advancement of CubeSat proximity operations.

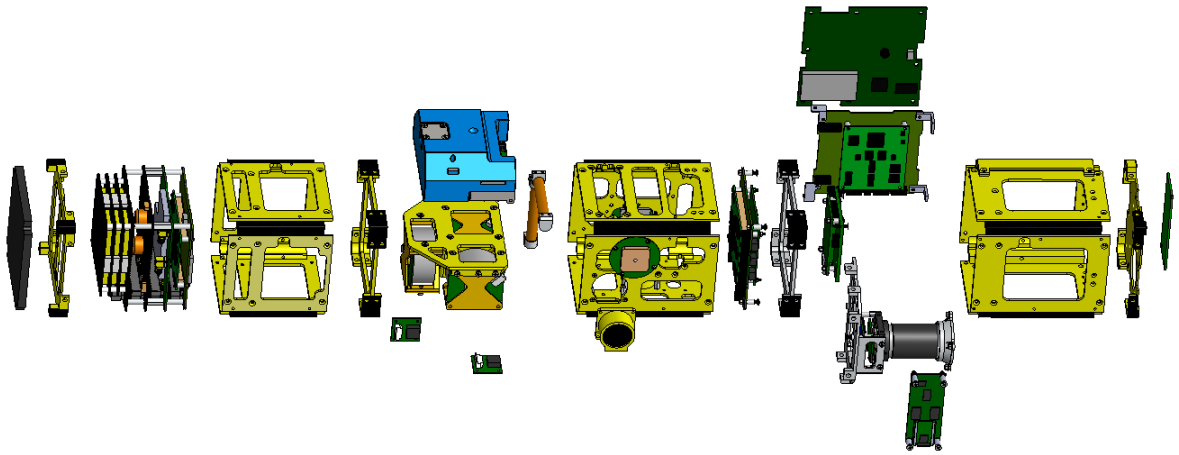

Figure 2.1: Exploded view of the Bevo-2 CAD model

An exploded view of the Bevo-2 CAD model is shown in Fig. 2.1. The middle module is the ADC system that contains all of the sensors and actuators. The 3D printed cold-gas thruster is the light blue object above the reaction wheel assembly.

Chapter 3

Environmental Simulation

The purpose of the environmental simulation is to provide a way to test the mission manager software. A rendezvous scenario with two satellites, the chaser and target, was developed that would provide the sensor and navigation measurement data to the mission manager software.

The target and chaser satellites in the simulation are assumed to be identical in mass and instrumentation. Each satellite is equipped with an accelerometer and a global positioning system (GPS) receiver. These sensor units give simulated measurements from which a navigation solution is estimated with an extended Kalman filter. The simulated measurements are obtained by taking the true satellite state and adding in noise. It is assumed that periodic communication exists between the target and chaser satellites; the chaser satellite is given the target's estimated position to perform relative navigation maneuvers. Another assumption that is that the sensors on both the satellites obtain measurements at the same time epochs.

3.1 Tools

Two software tools were used to create the simulations for CARDS. A C++ linear algebra library was used within the model software. This section provides a description of these tools and how they were implemented to create the simulation.

3.1.1 Armadillo Linear Algebra Library

A C++ linear algebra library, called Armadillo, was used for matrix calculations within the environment simulation models [6]. The Armadillo library has an easy-to-use syntax that is similar to MATLAB, making it very user-friendly and easy to read. Armadillo is open-source and licensed under the Mozilla Public License 2.0 (MPL).

3.1.2 Trick

All of the simulations for CARDS were created using a software package called Trick that was developed at NASA's Johnson Space Center (JSC) in Houston, Texas. Trick is a useful tool for creating and running dynamic simulations, which is why it was chosen for use in this project. An advantage that Trick brings is its ability to easily separate simulation and real-time execution for the user. Under the hood, Trick schedules the simulation functions to run at the appropriate time intervals while also monitoring the real-time clock, and will attempt to take action if it begins to fall behind schedule. Unfortunately, this may result in some skipped simulation execution frames as it will attempt to start the new appropriate execution frame immediately to catch up.

Functions and objects are written in C/C++ to be used by Trick. Simulation objects are defined in a syntax similar to C++ in a file named the S_define file. The S_define file is how Trick knows what objects should exist in the simulation as well as information about the functions that act on the objects. The functions and their function specifications are declared inside the definition of the simulation object. Function specifications are identifiers on functions that explain to Trick the purpose of that particular function, and therefore where and how it should be called during execution. There are numerous types of function specifications including, but not limited to, initialization, default data, scheduled, derivative,

and integration. The initialization specification means that the function should be called as soon as the simulation is started. The default data specification is typically responsible for setting any parameters that may have been included in a written simulation; this occurs after initialization so that a user can change the parameters of the simulation without worrying about a variable not being initialized. Unlike the initialization and default data functions that are usually run only once per simulation, scheduled functions are run by Trick at a user specified interval. A derivative function is used to calculate any derivatives needed for integration. Integration functions are used to perform Trick's integration. Both derivative and integration functions are called at every simulation time step.

In addition to the function specifiers, Trick has priority numbers that can be attached to the functions to specify the order in which functions should be called within each function specification group. The default order that functions are called within a function specification group is the order in which they are declared in the simulation object class definition. For example, suppose two functions with the initialization function specifier are defined in a simulation object. Each function also has a priority number in front of the function specification. The function with the higher priority will be called at simulation initialization before the function with lower priority even if the higher priority function is listed second in the simulation object definition.

Another advantage of Trick is that it utilizes user-made input files which define the parameters inside a simulation that are read into the simulation at runtime. This allows simulations with differing parameters to run without the need to recompile the whole simulation; only the input file needs to change. These input files are scripts written in the Python language that are processed at execution [7]. For example, a specification needed by the simulation executable is the desired simulation runtime. The line below sets the Trick

simulation to run for exactly 28800 seconds.

```
trick.sim_services.exec_set_terminate_time(28800.0)
```

3.1.3 JEOD

A module extension of Trick, known as the JSC Engineering Orbital Dynamics (JEOD) module, is also used. This module "is a collection of computational mathematical models that provide vehicle or vehicles trajectory generation by the solution of a set of dynamics models represented as differential equations." [8] It contains models that are useful for simulating planets and their gravitation, as well as orbit perturbations including atmospheric drag. The planet models have the option to be spherical or non-spherical bodies with spherical harmonics. The atmospheric drag effects can be easily turned on or off before running the simulation. JEOD also has the built-in capability for coordinate transformations, utilizing planet-centered inertial, planet-fixed, and local-vertical local-horizontal (LVLH) frames. It also includes relative frames that can be defined in relation to a specified target frame, such as the target satellite. This makes it easier to obtain relative navigation data for the chaser satellite. JEOD splits all of its models into four categories: Dynamics, Environment, Interactions, and Utilities. Fig. 3.1 shows a diagram for the JEOD classes that are used within Trick. More information regarding JEOD can be found in the JEOD User's Guide provided by NASA JSC.

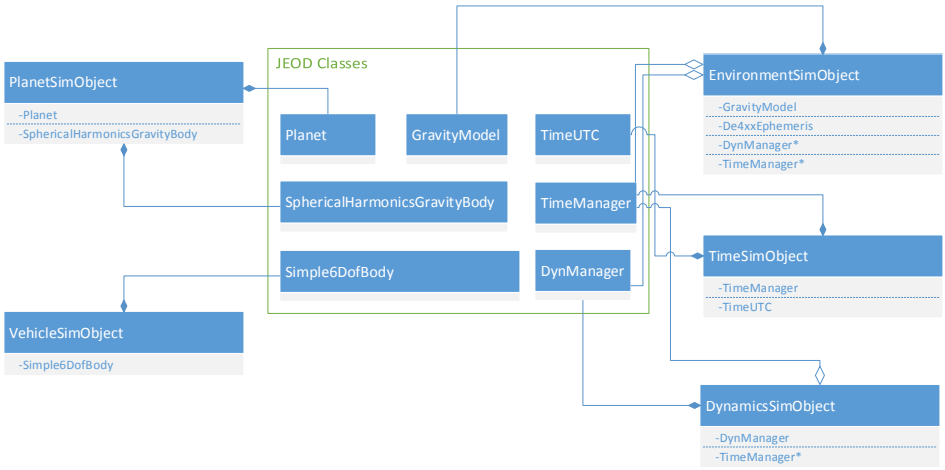

Figure 3.1: Environmental simulation class diagram showing basic JEOD class implementation

3.2 Dynamics

The Dynamics category of JEOD is related to the motion and properties of a vehicle, containing mathematical models for the equations of motion and kinematics. JEOD performs the numerical integration of both the translational and rotational equations of motion. In the simulation's S_define file, a DynamicsSimObject object that inherits from the Trick::SimObject class is created to track the dynamics for the simulation. Inside the DynamicsSimObject class, an object of JEOD's DynamicsManager class is created. The DynamicsManager class has functions necessary for integrating the simulation and vehicles' states. It also collects the various forces and torques that may be acting on a vehicle for use in the integration.

The DynManager object needs a centralized reference point for integration. This centralized reference point is defined in the Python input file for the simulation. In this rendezvous scenario, the centralized reference point is defined to be Earth. Additionally,

the propagator to use for dynamics integration is defined in the input file. The Runge-Kutta 4-5 integration method was used for the environmental simulation. The input file lines for setting the central reference point and integration method are given below.

```
dynamics.manager_init.central_point_name = "Earth"
dynamics.manager_init.sim_integ_opt
  = trick.sim_services.Runge_Kutta_Fehlberg_45
```

3.3 Environment

JEOD models that fall under the Environment category include "time, gravity, the atmosphere, and the solar system." [8] Simulation objects are created to manage each environmental model. Sample code declarations for each simulation object are available in the appendices.

3.3.1 Time

Time-keeping in space is a tricky endeavor; multiple standards for timekeeping exist. For instance, two standards in JEOD are the dynamics time used for integration and Universal Coordinated Time (UTC). The default and system time for JEOD is the dynamics integration time, which is not very useful for real-world scenarios. Fortunately, JEOD has a TimeManager class that is dedicated to tracking multiple standards for time. As with the Dynamics category, a simulation object, TimeSimObject, was created in the S_define file. Inside of the TimeSimObject, an instance of the TimeManager class is instantiated. To use a time standard, an object of that time standard class and a separate object to convert time standards is declared within the TimeSimObject class. For instance, to track the time according to the International Atomic Time (TAI) standard, an object of the classes

TimeTAI and TimeConverter_Dyn_TAI are created. This specific time converter is used to convert the dynamic time standard (used for integration) to TAI.

It should be noted that the DynamicsSimObject class contains a pointer to an object of type TimeManager. A reference to the TimeManager object contained in the TimeSimObject is passed into the constructor for the DynamicsSimObject to populate the pointer reference. This reference is needed by the DynManager object to access the current dynamics time needed for integration.

The initial simulation time is set in the Python input file. The time standard used for time initialization is first specified, otherwise JEOD will be unsure which time standard to use as the base time when initializing the other time standards. Then, each time standard (except the dynamics integration time) object needs a specifier of the time standard to use for initialization. The last piece of information needed in the input file for each time standard object is the desired time standard to use for updating. This requires that a corresponding time converting object was declared in the TimeSimObject. It is not required for the initialization and update time standard to be the same. For instance, if UTC is used as the base time standard for the simulation, then a TAI time standard object uses UTC as the initializing time standard, but uses the dynamics integration time to update for the rest of the simulation. Below is a sample from an input file that intializes the simulation time.

```
sim_time.manager_init.initializer = "UTC"
sim_time.manager_init.sim_start_format
  = trick.TimeEnum.calendar

sim_time.utc.calendar_year   = 2014
sim_time.utc.calendar_month  =   02
sim_time.utc.calendar_day    =   12
sim_time.utc.calendar_hour   =    0
sim_time.utc.calendar_minute =    0
sim_time.utc.calendar_second =  0.0

sim_time.tai.initialize_from_name  = "UTC"

sim_time.tai.update_from_name  = "Dyn"
sim_time.utc.update_from_name  = "TAI"
```

The above lines are a sample of initializing a simulations time. This example sets the time to midnight on February 12, 2014. Since the time was specified in UTC, the TAI time standard object initializes from UTC. Conversion objects from the dynamics integration time to TAI and from TAI to UTC are needed for the updates shown. This is reflected in the TimeSimObject example in Appendix C.

3.3.2 Gravity and Ephemeris

Another simulation object, called EnvSimObject, is created in the S_define file to define the gravitational environment models. Inside this simulation object are JEOD class objects that model gravity and the ephemeris data: the GravityModel and De4xxEphemeris classes, respectively. The EnvSimObject class has pointers to TimeManager and DynamicsManager objects. References to these objects are passed in from the other previously described simulation objects into the EnvSimObject constructor. The GravityModel and De4xxEphemeris initialization functions called inside the EnvSimObject class require these pointers to be populated prior to initialization. Within the EnvSymObject class the DynamicsManager pointer calls a function to update the ephemeris data at a regularly scheduled

time interval.

3.3.3 Planet

The CARDS environment requires that two satellites, the target and chaser, are in orbit around Earth. The JEOD classes Planet and SphericalHarmonicsGravityBody are used to create an Earth object. Instances of both of these objects are inside an EarthSimObject class object that inherits from the Trick::SimObject class. The initialization functions for the Planet and SphericalHarmonicsGravityBody are defined with the Trick initialization function specification. In addition, the Planet::register_model function is specified as a Trick initialization function, which registers the SphericalHarmonicsGravityBody object with the JEOD dynamics manager. The Planet class by default has no data for which planet is being modeled. It is necessary to define the planet data within the simulation object. In the case of Earth, an object of the JEOD class Planet_earth_default_data is used to set the variables such as the name ("Earth") for the Planet object. The gravitational constant for the Planet object is defined in the simulation's Python input file. For example, an EarthSimObject class object named "earth" has the following line in the input file to define the gravitational constant:

```
earth.planet.mu = trick.attach_units("M3/s2",3.98600436e14)
```

3.4 Vehicles

The CARDS environment is specified for a rendezvous guidance situation. Therefore, there are two vehicles that need to be simulated, the target and the chaser. For preliminary results, the target and chaser have been assumed to be identical in mass and

instrumentation. Each vehicle is a separate object of a vehicle simulation class, named SvDynSimObject, defined in the S_define file. The vehicle simulation class contains an instances of a JEOD class, Simple6DofDynBody, that models the mass properties as well as the translational and rotational state of the vehicle. The SVDynSimObject class also has JEOD class objects for various reference frame states such as PlanetaryDerivedState, OrbElemDerivedState, LvlhDerivedState, and RelativeDerivedState. These reference frame objects are defined in section 3.5. The vehicle class also has objects of the accelerometer and GPS sensor classes. An object of the Kalman filter class is also defined in the vehicle object. The Kalman filter object's update function is called at a regularly scheduled interval to estimate the state of the vehicle object.

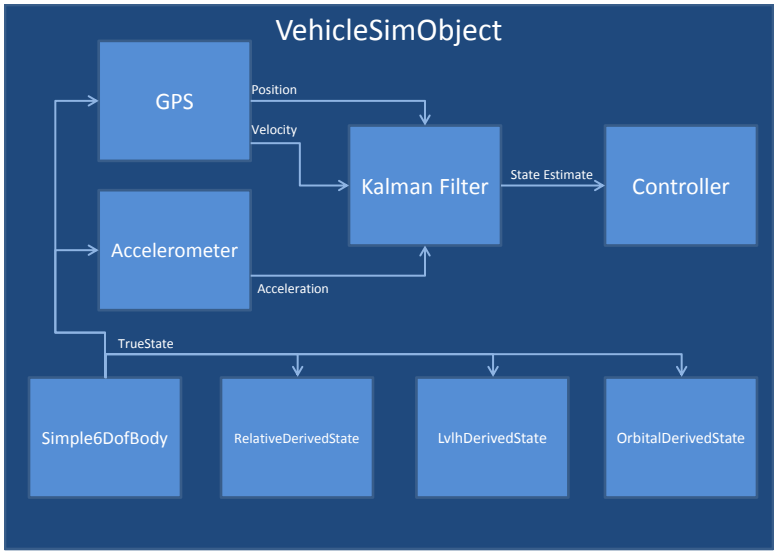

Figure 3.2: Vehicle simulation state block diagram

The Simple6DofBody object within the SVDynSimObject class needs a reference to the frame in which it is integrated. This is defined in the simulation's input file. The integration of the translation and rotational dynamics for the body can also be turned on

or off. The following lines set a SVDynSimObject object named “chaser” to use Earth’s inertial frame as the integration frame and turns off rotational dynamics for the body.

```
chaser.body.integ_frame_name        = "Earth.inertial"
chaser.body.translational_dynamics = True
chaser.body.rotational_dynamics     = False
```

Additionally, the SVDynSimObject class has a SphericalHarmonicsGravityControls object. The SphericalHarmonicsGravityControls class controls how a planet’s gravity affects the SVDynSimObject’s body. Parameters for the SphericalHarmonicsGravityControls object are set in the simulation’s input file.

```
chaser.earth_grav_control.planet_name = "Earth"
chaser.earth_grav_control.active      = True
chaser.earth_grav_control.spherical   = True
chaser.earth_grav_control.gradient    = False
chaser.body.grav_interaction.add_control(chaser.earth_grav_control)
```

The first line above specifies the planet that the SphericalHarmonicsGravityControls object controls. The next line allows the effect of the planet’s gravity on the SVDynSimObject to be toggled on and off. When set to false, the planet’s gravity is “off” for the vehicle and its dynamics are unaffected by the planet. The next two lines specify if the planet should be modeled as a spherical or non-spherical body when acting on the vehicle object. Finally, the last line is necessary for the Simple6DofBody object defined within the SVDynSimObject to obtain information about the SphericalHarmonicsGravityControls object. Without this, the vehicle object would be unaffected by the planet’s gravity.

3.5 Reference Frames

A number of reference frame classes exist within JEOD to help the user view a body’s state in different coordinates without having to perform the transformations manu-

ally.

3.5.1 Planet-Fixed

The PlanetaryDerivedState class tracks the state of the vehicle body in terms of the latitude, longitude, and altitude in reference to a planet with the origin at the geometric planet center. A PlanetaryDerivedState object is initialized with references to Simple6DofBody and DynManager objects. The PlanetaryDerivedState update function computes the latitude, longitude, and altitude of the vehicle from the vehicle's current state. The following lines show the use of a PlanetaryDerivedState object's initialization and update functions inside of a simulation object class in the S_define file. In this example, the PlanetaryDerivedState object is called pfix.

```
P_DYN ("initialization") pfix.initialize(body, *dyn_manager);
(dyn_cycle, "environment") pfix.update();
```

In the above lines, the initialize function of the pfix object has the Trick function specifier ("initialization") in front, telling Trick that this function is to be run at simulation initialization. The P_DYN keyword is a predefined JEOD priority number that to group any dynamics initialization functions to be called at the same initialization priority. The second line defines the pfix object's update function with an "environment" function specifier, which is a scheduled function that is called every "dyn_cycle" seconds in simulation time.

3.5.2 Orbital Elements

Another JEOD class, OrbElemDerivedState, is used to compute a body's state in terms of the Keplerian orbital elements. As with a PlanetaryDerivedState object, an OrbElemDerivedState object is initialized with Simple6DofBody and DynManager object

references. The OrbElemDerivedState update function is called from the object instance to update the orbital elements with the current body state.

```
P_DYN ("initialization") orb_elem.initialize(body, *dyn_manager);
(dyn_cycle, "environment") orb_elem.update();
```

3.5.3 Local-Vertical Local-Horizontal

The LvlhDerivedState class defines a LVLH reference frame for a body. This class does not compute the body's state as the reference frame is centered at the body's geometric origin. Instead, the purpose of this class is to be used to calculate a separate body's state relative to the current body's state in the current body's LVLH reference frame. The LVLH reference frame used by JEOD is characterized in the following manner. The Z-axis is in the radial direction from the body's origin to the planet's geometric center. The Y-axis is perpendicular to the body's orbital plane in the opposite direction of the angular momentum vector. Finally, the X-axis completes a right-handed coordinate system. Figure 3.3 depicts this LVLH reference frame definition.

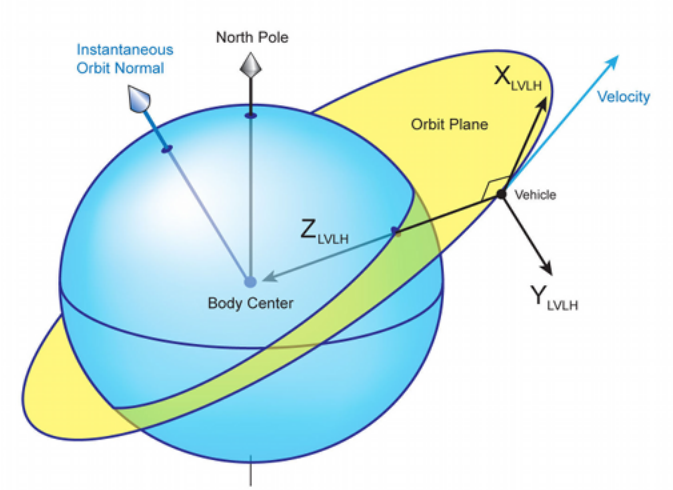

Figure 3.3: JEOD LVLH reference frame [8]

3.5.4 Relative

A body's state relative to a defined reference frame is computed using the RelativeDerivedState class. A RelativeDerivedState object is initialized and updated in the same manner as the other reference frame classes. However, unlike the other objects, the RelativeDerivedState object needs a target reference frame in which to define the relative state. The target reference frame is declared by the user in the simulations input file. In addition, the direction of the relative state must be specified, either subject to target or target to subject. The following lines show the chaser vehicle's relative frame being set to use the target's LVLH frame with the target to subject direction.

```
chaser.rel.target_frame_name = "target.Earth.lvlh"
chaser.rel.direction_sense =
  trick.RelativeDerivedState.ComputeSubjectStateinTarget
```

3.6 Sensor Models

As previously mentioned, the vehicle simulation objects possess child objects that model the various onboard sensors. The currently modeled sensors are accelerometers and a Global Positioning System (GPS) receiver. This sensor list is preliminary and will be updated as more aspects of the CubeSat hardware are included in the analysis. This section describes the models used to simulate these onboard sensors.

3.6.1 Accelerometer

The model for each accelerometer sensor measurement is given by

$$\tilde{a} = a + \epsilon_a + \eta_{w,a} \tag{3.1}$$

$$\dot{\epsilon}_a = -\epsilon_a/\tau_a + \eta_{d,a} \tag{3.2}$$

where tilde represents a measured sensor value. In other words, the measured acceleration is the truth with an added Gauss-Markov noise term, ϵ_a, and white noise measurement error, $\eta_{w,a}$, with standard deviation, $\sigma_{w,a}$. Eq. (3.2) represents a first-order Gauss-Markov noise model for the random walk measurement error. The values for the error correlation time constant, τ_a, and the standard deviation, $\sigma_{d,a}$, for the zero-mean Gaussian distribution terms are parameters associated with a physical accelerometer. A manufacturer may give the accelerometer random walk noise, $\sigma_{r,a}$, in units of $\mathrm{m/s^2}/\sqrt{\mathrm{Hz}}$. This is related to the standard deviation for the Gauss-Markov white noise error, $\sigma_{d,a}$, by [9]

$$\sigma_{d,a} = \sigma_{r,a}\sqrt{2/\tau_a} \tag{3.3}$$

A C++ class to model an accelerometer was defined for use with Trick. The accelerometer class has variables pertaining to the white noise standard deviation, $\sigma_{w,a}$, the time constant, τ_a, and the drift bias, $\sigma_{r,a}$. At every simulation time step interval, the time-correlated noise term ϵ_a is integrated. The accelerometer class has an update function that calculates new measurement values. The rate for the update function to be called is set in the S_define file. At every time step where the update function is called, the accelerometer class object obtains the true acceleration for the vehicle body to which it is attached and adds in the current value for the noise, ϵ_a, as well as a value from the distribution, $\eta_{w,a}$; the result is stored in a variable for the current acceleration measurement.

The parameters for each accelerometer ($\sigma_{w,a}$, τ_a, $\sigma_{r,a}$) are set in the simulations input file. It should be noted that the user inputs $\sigma_{r,a}$ and the model uses Eq. (3.3) to calculate the value for $\sigma_{d,a}$. Thus, it is easy to perform simulations for modeling different accelerometer devices. The accelerometer noise can also be turned off in the input file by setting the standard deviations for the noise terms to zero and a flag can be set so that the integration of ϵ_a does not occur. This allows for testing using true acceleration values if desired. The following lines are an example setting the time correlation, τ_a, to 10 min and the accelerometer sensor bias to 2 m/s^2 in the simulation's input file. In this example, the vehicle simulation and accelerometer sensor objects are called "chaser" and "imu," respectively.

```
chaser.imu.accel_gm_time
  = trick.sim_services.attach_units("min", 10.0)
chaser.imu.accel_bias
  = trick.sim_services.attach_units("m/s2", 2.0)
```

The accelerometer class (called the IMU class as gyroscopes will also be added) is defined with a reference to a Simple6DofBody object. A pointer to the SVDynSimObject's

Simple6DofBody object is passed into the initialization function of the IMU class. The Simple6DofBody object reference is how the accelerometer model obtains the true acceleration of the body to create its acceleration measurement. The IMU::update_noise function calculates the derivative of the random walk noise, $\dot{\boldsymbol{\epsilon}}_a$ as defined in Eq. (3.2). This function is given a derivative function specifier to ensure that it is called before every integration, resulting in the random walk noise, $\boldsymbol{\epsilon}_p$, at each integration step.

3.6.2 GPS

The GPS sensor model uses the same measurement and noise format as the accelerometer sensor model.

$$\tilde{\mathbf{r}} = \mathbf{r} + \boldsymbol{\epsilon}_p + \boldsymbol{\eta}_{w,p} \tag{3.4}$$

$$\dot{\boldsymbol{\epsilon}}_p = -\boldsymbol{\epsilon}_p/\tau_p + \boldsymbol{\eta}_{d,p} \tag{3.5}$$

Similar to the accelerometer model, the GPS sensor was written as a C++ class. The GPS class has variables that store the current measured GPS position in the Earth-Centered Inertial reference frame (ECI), as that is the default reference frame for the Trick simulation. However, the reference frame for the GPS measurements can be easily changed to store the Earth-Centered Earth-Fixed (ECEF) position using the built in JEOD reference frame transformations for a more realistic simulation. Like the accelerometer model, the GPS class has an update function that is set to be called at a regular time interval in the S_define file. When the simulation calls a GPS object's update function, the measurement for the current position is calculated. First, the GPS object obtains from JEOD the true satellite position in the simulation (in the default ECI frame). Then, the value for the drift

and random walk noise, $\boldsymbol{\epsilon}_p$, at the current simulation time is added to the true position, as well as a generated random white noise value from the $\boldsymbol{\eta}_{w,p}$ distribution. The drift and random walk noise is integrated at the simulation's set integration rate and occurs before the GPS update function is called so that the most up-to-date noise value is used in the measurement calculation. In addition, it is assumed that the GPS receiver is also estimating a solution for the spacecraft velocity. The model for the velocity is comparable to the position model, with different values for the noise and time constant variables.

The GPS sensor class, like the accelerometer class, has a reference to a Simple6DofBody object. The reference to the vehicle simulation's Simple6DofBody object is passed as a parameter into the GPS object's initialization function. This reference is needed for the GPS sensor class to access the true position and velocity states of the body for creating measurements. The GPS::update_noise function calculates the derivative of the random walk noise, $\dot{\boldsymbol{\epsilon}}_p$. This function is given a derivative function specifier to ensure that it is called before every integration, resulting in the random walk noise, $\boldsymbol{\epsilon}_p$, at each integration step.

3.7 Kalman Filter

A Kalman filter class was created to perform an extended Kalman filter (EKF) technique to estimate a satellite's position, velocity, and accelerometer bias using the accelerometer and GPS position and velocity measurements. Additionally, the covariance of the state is estimated. First, the standard procedure for finding the estimate of a state and covariance from a process is shown. Then, the model for the chaser and target vehicle state estimation is described.

A standard process has the form shown by Eq. (3.6) where $\boldsymbol{X}$ represents the $n \times 1$

state vector with n states, $\boldsymbol{u}$ is the $q \times 1$ control vector, and $\mathbf{f}$ is an $n \times 1$ vector-valued function representing the dynamics of the system[10]. The second term, $\mathbf{G}\boldsymbol{w}(t)$, represents uncertainty in the process model, where $\boldsymbol{w}$ is an $n \times 1$ vector of white noise whose standard deviation is a result of the process model and must be tuned.

$$\dot{\boldsymbol{X}}(t) = \mathbf{f}(\boldsymbol{X}, \boldsymbol{u}, t) + \mathbf{G}\boldsymbol{w}(t) \tag{3.6}$$

$$\boldsymbol{Y}(t) = \mathbf{g}(\boldsymbol{X}, t) + \boldsymbol{v}(t) \tag{3.7}$$

Equation (3.7) is the measurement model, where $\boldsymbol{Y}(t)$ is an $m \times 1$ measurement vector and $\boldsymbol{v}(t)$ is an $m \times 1$ white noise vector. The process and measurement noise terms, $\boldsymbol{w}$ and $\boldsymbol{v}$, have zero correlation with covariances $\mathbf{Q}$ and $\mathbf{R}$, respectively.

$$E[\boldsymbol{w}\boldsymbol{v}^T] = \mathbf{0} \tag{3.8}$$

$$E[\boldsymbol{w}\boldsymbol{w}^T] = \mathbf{Q} \tag{3.9}$$

$$E[\boldsymbol{v}\boldsymbol{v}^T] = \mathbf{R} \tag{3.10}$$

It is assumed that there is no cross-correlation between the elements in each noise vector and as a result $\mathbf{Q}$ and $\mathbf{R}$ are diagonal matrices. This model can be linearized by taking the first-order Taylor series expansion of Eq. (3.6) about the current state estimate $\hat{\boldsymbol{x}}$ with $\boldsymbol{X}(t) = \hat{\boldsymbol{x}}(t) + \boldsymbol{x}(t)$ and $\boldsymbol{Y}(t) = \mathbf{g}(\hat{\boldsymbol{x}}, t) + \boldsymbol{y}(t)$.

$$\dot{\boldsymbol{x}}(t) = \mathbf{A}\boldsymbol{x}(t) + \mathbf{B}\boldsymbol{u}(\boldsymbol{x}, t) \tag{3.11}$$

$$\boldsymbol{y}(t) = \mathbf{H}\boldsymbol{x}(t) \tag{3.12}$$

where the matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{H}$ have sizes $n \times n$, $n \times q$, $m \times n$, respectively, and are defined as

$$\mathbf{A} \equiv \partial \mathbf{f} / \partial \boldsymbol{x}|_{\boldsymbol{x}=\hat{\boldsymbol{x}}} \tag{3.13}$$

$$\mathbf{B} \equiv \partial \mathbf{f} / \partial \boldsymbol{u}|_{\boldsymbol{x}=\hat{\boldsymbol{x}}} \tag{3.14}$$

$$\mathbf{H} \equiv \partial \mathbf{g} / \partial \boldsymbol{x}|_{\boldsymbol{x}=\hat{\boldsymbol{x}}} \tag{3.15}$$

The covariance propagation is performed with the following model.

$$\mathbf{P}(t) = E[\boldsymbol{x}(t)\boldsymbol{x}(t)^T] \tag{3.16}$$

$$\dot{\mathbf{P}}(t) = \mathbf{AP} + \mathbf{PA}^T - \mathbf{PH}^T\mathbf{R}^{-1}\mathbf{HP} + \mathbf{GQG}^T \tag{3.17}$$

Assuming an a priori estimate for the state estimate and covariance, it is now possible to update these estimates using the models above at each measurement epoch. If there is no a priori estimate, $\bar{\boldsymbol{x}}(0) = \mathbf{0}$ and $\bar{\mathbf{P}}(0) = \boldsymbol{I}_{n\times n}$ are used. The procedure for updating the state estimate is to first propagate the a priori estimate and covariance to the next measurement epoch. Then, the measurement model is calculated at the current state estimate. The Kalman gain is then found and used to find the new estimated state $\hat{\boldsymbol{x}}$ and covariance $\mathbf{P}$. This procedure is outlined below.

1. Propagate a priori estimates $\bar{\boldsymbol{x}}$ and $\bar{\mathbf{P}}$ using eqs. (3.18) and (3.17) to next measurement epoch to obtain $\hat{\boldsymbol{x}}$ and $\mathbf{P}$.

2. Calculate the measurement residual, $\boldsymbol{y}(t) = \boldsymbol{Y}(t) - \mathbf{g}(\hat{\boldsymbol{x}}, t)$

3. Find the Kalman gain $K = (\mathbf{PH})^{-1}(\mathbf{HPH}^T + \mathbf{R}^{-1})$

4. Calculate new best estimate $\hat{\boldsymbol{x}}_{new} = K\boldsymbol{y}$

5. Calculate new covariance $\mathbf{P}_{new} = (\boldsymbol{I}_{n\times n} - K\mathbf{H})\mathbf{P}$

6. Update a priori estimate $\bar{\boldsymbol{x}} = \hat{\boldsymbol{x}}_{new}$, $\bar{\mathbf{P}} = \mathbf{P}_{new}$

The state for a satellite in orbit around Earth is given in terms of position and velocity of the satellite in the ECI reference frame, as well as the accelerometer bias. The vehicle has some 3 degree-of-freedom acceleration control (through thrusters or another actuator). Therefore, the number of states in this case is $n = 9$ with $q = 3$ controls. The resulting state space model is

$$\dot{\boldsymbol{x}}(t) = \begin{bmatrix} \dot{\mathbf{r}} \\ \ddot{\mathbf{r}} \\ \boldsymbol{b} \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{r}} \\ -\mu\mathbf{r}/r^3 + \boldsymbol{u} \\ \mathbf{0} \end{bmatrix} \tag{3.18}$$

$$\mathbf{A} = \begin{bmatrix} \mathbf{0}_{3\times3} & \boldsymbol{I}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{J} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \end{bmatrix} \tag{3.19}$$

$$\mathbf{B} = \begin{bmatrix} \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \boldsymbol{I}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \end{bmatrix} \tag{3.20}$$

$$\mathbf{G} = \boldsymbol{I}_{9\times9} \tag{3.21}$$

Where

$$\mathbf{J} \equiv \frac{\partial\ddot{\mathbf{r}}}{\partial\mathbf{r}} = -\frac{\mu}{r^3}\boldsymbol{I}_{3\times3} + \frac{3\mu}{r^5}\mathbf{r}\mathbf{r}^T \tag{3.22}$$

The measurements being made are the position, velocity, and acceleration of the satellite. In order to account for the bias in the accelerometer measurement the accelerometer bias term is added into the model for the acceleration.

$$\boldsymbol{y}(t) = \begin{bmatrix} \mathbf{r} \\ \dot{\mathbf{r}} \\ -\mu\mathbf{r}/r^3 + \boldsymbol{b} \end{bmatrix} \tag{3.23}$$

$$\mathbf{H} = \begin{bmatrix} \boldsymbol{I}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \boldsymbol{I}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{J} & \mathbf{0}_{3\times3} & \boldsymbol{I}_{3\times3} \end{bmatrix} \tag{3.24}$$

The KalmanFilter class has a derivative function that is given the Trick derivative function specifier. The derivative function calculates the derivative of the state model and covariance, as in Eqs. (3.18) and (3.17). This function is given the current accelerometer measurement as an input parameter so that the state propagation "flys the accelerometers" between measurement epochs. The KalmanFilter::updateState function is given a function specifier of a scheduled function that is called every "dyn_cycle" seconds in simulation time. The GPS object's current position and velocity measurements as well as the accelerometer object's acceleration measurements are taken as input parameters.

```
("derivative") filter.derivatives(imu.body_accel);
(dyn_cycle, "sensor")
  filter.updateState(gps.position, gps.velocity, imu.body_accel);
```

3.8 Controller

A simple guidance law was implemented to control the chaser satellite to a 1 meter distance from the target satellite. The law implemented is described in a paper by D'Souza[11] using Hill's equations for satellites in near-circular orbits.

$$\dot{\boldsymbol{X}}(t) = \boldsymbol{A}\boldsymbol{X}(t) + \boldsymbol{B}\boldsymbol{u}(t) \tag{3.25}$$

$$\boldsymbol{X}(t) = [x \quad y \quad z \quad \dot{x} \quad \dot{y} \quad \dot{z}] \tag{3.26}$$

$$\boldsymbol{A} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 2\omega \\ 0 & -\omega^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3\omega^2 & -2\omega & 0 & 0 \end{bmatrix} \tag{3.27}$$

$$\boldsymbol{B} = \begin{bmatrix} \mathbf{0}_{3\times3} & \boldsymbol{I}_{3\times3} \end{bmatrix}^T \tag{3.28}$$

$$\boldsymbol{u}(t) = -\boldsymbol{B}^{\boldsymbol{T}}\boldsymbol{R}(t)\boldsymbol{Q}^{-1}(t)\left[\boldsymbol{\psi} - \boldsymbol{R}^T(t)\boldsymbol{X}(t)\right] \tag{3.29}$$

In Eq. (3.29) the variable $\boldsymbol{\psi} = [x_f \quad y_f \quad z_f \quad \dot{x}_f \quad \dot{y}_f \quad \dot{z}_f]^T$ is the desired state of the controlled vehicle at the given final time, t_f. It can easily be seen that the y direction can be decoupled from the x and z, and as such the control can be split into two different problems to be solved separately. The matrices $\boldsymbol{R}$ and $\boldsymbol{Q}$ are the guidance and controllability matrices whose elements will be defined for each problem. The controllability matrix, $\boldsymbol{Q}$, has the property of symmetry, $\boldsymbol{Q} = \boldsymbol{Q}^T$.

As shown by D'Souza, the coupled control for x and z can be solved using

$$R_{11} = 1 \tag{3.30}$$

$$R_{12} = 0 \tag{3.31}$$

$$R_{13} = 0 \tag{3.32}$$

$$R_{14} = 0 \tag{3.33}$$

$$R_{21} = 6(\omega t_{go} - \sin\omega t_{go}) \tag{3.34}$$

$$R_{22} = 4 - 3\cos\omega t_{go} \tag{3.35}$$

$$R_{23} = 6\omega(1-\cos\omega t_{go}) \tag{3.36}$$

$$R_{24} = 3\omega\sin\omega t_{go} \tag{3.37}$$

$$R_{31} = (4\sin\omega t_{go} - 3\omega t_{go})/\omega \tag{3.38}$$

$$R_{32} = 2(\cos\omega t_{go} - 1)/\omega \tag{3.39}$$

$$R_{33} = 4\cos\omega t_{go} - 3 \tag{3.40}$$

$$R_{34} = -2\sin\omega t_{go} \tag{3.41}$$

$$R_{41} = 2(1-\cos\omega t_{go})/\omega \tag{3.42}$$

$$R_{42} = \sin\omega t_{go}/\omega \tag{3.43}$$

$$R_{43} = 2\sin\omega t_{go} \tag{3.44}$$

$$R_{44} = \cos\omega t_{go} \tag{3.45}$$

$$Q_{11} = (3\sin 2\omega t_{go} + 32\sin\omega t_{go} - 24\omega t_{go}\cos\omega t_{go} - 3(\omega t_{go})^3 - 14\omega t_{go})/\omega^3 \tag{3.46}$$

$$Q_{12} = -3(\sin\omega t_{go} - \omega t_{go})^2/\omega^3 \tag{3.47}$$

$$Q_{13} = (6\cos 2\omega t_{go} + 8\cos\omega t_{go} + 24\omega t_{go}\sin\omega t_{go} - 9(\omega t_{go})^2 - 14)/2\omega^2 \tag{3.48}$$

$$Q_{14} = -(3\sin 2\omega t_{go} + 16\sin\omega t_{go} - 12\omega t_{go}\sin\omega t_{go} - 10\omega t_{go})/2\omega^2 \tag{3.49}$$

$$Q_{22} = -(3\sin 2\omega t_{go} - 32\sin\omega t_{go} + 26\omega t_{go})/4\omega^3 \tag{3.50}$$

$$Q_{23} = -(3\sin 2\omega t_{go} - 28\sin\omega t_{go} + 22\omega t_{go})/2\omega^2 \tag{3.51}$$

$$Q_{24} = -(3\cos 2\omega t_{go} - 16\cos\omega t_{go} + 13)/4\omega^2 \tag{3.52}$$

$$Q_{33} = -(3\sin 2\omega t_{go} - 24\sin\omega t_{go} - 19\omega t_{go})/\omega \tag{3.53}$$

$$Q_{34} = -(12\sin\frac{\omega t_{go}}{2})^4/\omega \tag{3.54}$$

$$Q_{44} = (3\sin 2\omega t_{go} - 10\omega t_{go})/4\omega \tag{3.55}$$

$$t_{go} \equiv t_f - t \tag{3.56}$$

Similarly, for the decoupled y coordinate control:

$$\boldsymbol{R} = \begin{bmatrix} \cos\omega t_{go} & -\omega \sin\omega t_{go} \\ \frac{\sin\omega t_{go}}{\omega} & \cos\omega t_{go} \end{bmatrix} \tag{3.57}$$

$$Q_{11} = (\sin 2\omega t_{go} - 2\omega t_{go})/4\omega^3 \tag{3.58}$$

$$Q_{12} = (\cos 2\omega t_{go} - 1)/4\omega^2 \tag{3.59}$$

$$Q_{22} = -(\sin 2\omega t_{go} + 2\omega t_{go})/4\omega \tag{3.60}$$

Chapter 4

Mission Manager

4.1 Purpose

The primary purpose of the mission manager is to provide a CubeSat with an autonomous guidance system. The system must be capable of monitoring the vehicle's rendezvous path and taking any necessary corrective action in case of maneuver deviation or failed hardware. Although execution of the mission manager is autonomous, human interaction is intended for higher level instructions such as go/no-go commands. The design of the mission manager software is intended to have a few modes to provide nominal functionality with guidance (path) and sensor health monitoring, failure correction, and standby for human input if necessary. The highest level of the mission manager software is a mode switcher that allows for easy transition from each mode to the next depending on events that trigger the mode switch. All of the mission manager software is written in the C++ programming language in preparation for porting to a spacecraft computer.

4.2 Design

A base Mode class exists to provide a baseline for the more specific mode classes which inherit from the base Mode. Each mode class has a central function to perform the nominal processes for that mode. The ModeManager has a pointer to a base Mode object which keeps track of the current mode of the mission manager. A main loop performs the current mode's nominal processes indefinitely or until a transition event occurs. In the

central function for each mode, checks exist for possible transition events. When an event is triggered, the ModeManager's main loop ends the functions of the current mode and enters the appropriate mode after the transition, beginning the processes for the next mode (which is now the current mode after transition). For instance, a sensor failure is a transition event that would exit the nominal guidance mode and enter standby for human input.

Another design of the mode classes is that each class is a singleton. Only one object of each mode may exist in the software. This is enforced by having a static pointer of each mode's own class type inside of its own definition. The constructor for each mode class is also private so that the mode cannot be created accidentally; a function belonging to each mode class must be called to create an instance of that mode. In addition, this creation function checks if an instance of the object exists by checking the static pointer member. If the static pointer is null, no instance of that mode has yet been created, so the constructor is called and the static pointer is set to point to the newly created mode object. Furthermore, static variables such as these mode pointers must be defined at compile-time of the software, meaning that these mode class objects will always exist from start to finish.

A block diagram of the mission manager software is shown in Fig. 4.1. Classes and threads inside the mission manager software are represented by light blue and orange boxes, respectively. The green box shows human interaction with the mission manager software.

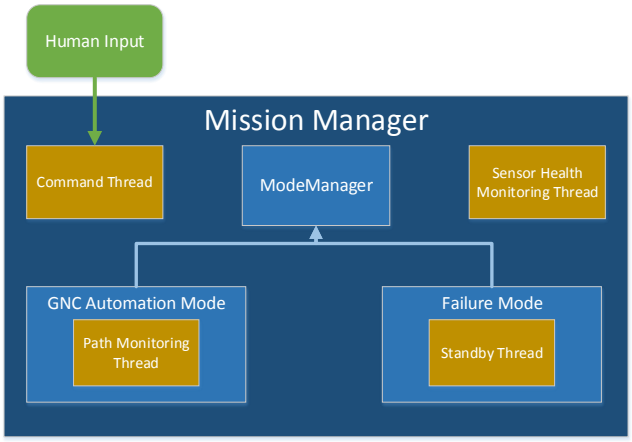

Figure 4.1: Mission manager block diagram

4.2.1 Guidance Automation

As stated, an important component of the mission manager software is to determine the guidance maneuvers for the CubeSat. It should be able to control the vehicle from 1 km to a 1 m distance to the target. The chaser satellite is assumed to be in communication with the target satellite and is receiving periodic state updates from the target. The control law in section 3.8 was implemented for testing, but this controller is intended as an example only in order to demonstrate the operation of the mission manger. The mission manager has a thread that is responsible for monitoring the current and planned path for the vehicle as part of the guidance, navigation, and control (GNC) mode. The monitoring thread constantly checks the chaser vehicle's trajectory for a possible maneuver deviation and will autonomously trigger a corrective action maneuver transition if needed. The mission manager calculates the nominal trajectory by propagating the equations of motion from the maneuver's starting state to the desired final state using the current state estimate as initial conditions. At each measurement epoch, the thread compares the updated state estimate to the calculated nominal state. A performance index is used to determine if the estimated

satellite position is too far off from the intended trajectory and a corrective maneuver is needed.

4.2.2 Sensor Health Monitoring

Separately, another thread runs alongside the navigation thread to monitor sensor health. If a sensor stops reporting data, the mission manager will take note and take an appropriate action, which may include standing by for human input if a reboot is desired. Currently, there are plans to also check the case where a sensor is reporting data which is incorrect or "false positive." This is a much harder case to diagnose than if the sensor is no longer responding and more research into determining the validity of sensor data is needed for this part of the mission manager.

4.2.3 Human Input

Another thread in the mission manager constantly listens for human input. The human input may be override or exit commands. When an instruction is received by the command thread, the message is parsed and the appropriate action is taken. For instance, if the user enters an override command to enter standby, the transition to standby event is triggered as it would have if a real failure had occurred.

The standby thread exists in case of system failure. While the standby thread is running, the mission manager waits until a continue command is received from the user. When the continue command is received by the command thread, the standby thread's exit event is triggered and the mission manager will proceed with its next function.

4.2.4 Testing

The mission manager software is currently in development. Thorough testing is planned to ensure all parts of the mission manager software perform as intended. The ModeManager and Mode classes will undergo unit testing to validate the functionality of each class. After each class has been unit tested, full functional testing will be performed on the mission manager software by testing predetermined scenarios to simulate failures.

The sensor health thread will be tested and validated by changing the value of a variable acting as a flag in each sensor object mid-simulation. If a flag that controls whether or not the sensor object is collecting data is set to false, that object's update function does not obtain new values for the sensor's measurement variable. The data collection flag is controlled with a button on a Trick created graphical user interface (GUI). When the button is pressed by the user, the value of the data collection flag toggles between off and on values. Similarly, another flag variable owned by each sensor object is responsible for the false positive data case. When this flag is set to true, the sensor's update function will provide incorrect data measurements. Alternately, scheduled simulation events such as sensor failures can be scripted to occur at designated times through a user defined input file.

A similar testing scheme will be used for the GNC mode thread. Using the same GUI method, the mission manager may receive false state estimates or erroneous measurement data, simulating that the vehicle is on a path different than the nominal. The corrective action taken by the mission manager software is then fed back into the testing GUI, affecting the true vehicle state.

An issue to be resolved is how to determine if the vehicle's sensors have failed due

to bad calibration (or other failure) or if the vehicle is off the nominal trajectory when the estimated vehicle state differs from the calculated nominal state. If only one sensor is providing measurements that are not within a to-be-determined tolerance of the other sensors, then there is a greater chance that a sensor failure has occurred rather than the vehicle being off the calculated path. However, if an unlikely situation presents itself where all sensors have failed in a similar bias, it is a possibility that the mission manager decides the vehicle is deviating from the nominal path, yielding an undesired result. The likelihood of this scenario and possible solutions will be researched examined in future work.

4.3 Embedded System

As previously mentioned, the mission manager runs on a CubeSat's embedded microprocessor system. The embedded system used in the TSL's Bevo-2 satellite is a Phytec PhyCORE®-LPC3250 running a generic Linux kernel. Fig. 4.2 shows the LPC3250 system on a module with dimensions. The LPC3250 has an ARM9 processor that runs up to 208 MHz, 128 MB NAND, and has ethernet, USB, UART, and I2C capabilities.

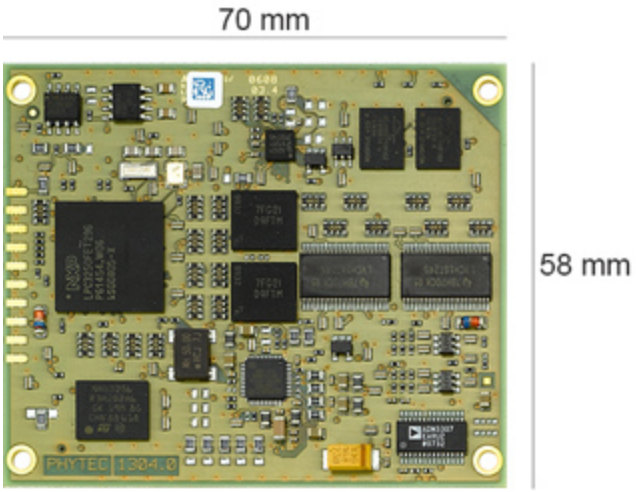

Figure 4.2: Phytec PhyCORE®-LPC3250 System on a Module [Photo Credit: Phytec]

The mission manager software will be ported to this target system for testing. Standalone testing of the mission manager will be performed on the system before real sensor measurements are provided. Successful testing of the mission manager software would result in readiness for flight operations on the Bevo-2 satellite as the testing system is identical to the flight system.

Chapter 5

Simulation Testing

The Trick environmental simulation was run for approximately 5 orbits (8 hours) with the chaser satellite in orbit around Earth at an altitude of 400 km in cases with and without the controller providing actuation. In this simulation, Earth was modeled as a spherical body and there were no atmospheric drag effects. Typical results from the simulation are presented.

5.1 Nonlinear Comparison

The use of Trick and JEOD simulations were verified by comparing the propagation of the nonlinear equations of motion for satellites in Earth orbit with a MATLAB simulation. This analysis was performed to ensure that there were no major discrepancies between the two software packages. In both Trick and MATLAB, a satellite was in an approximately 400 km altitude circular orbit around a spherical Earth without any perturbations. The Trick simulation was run for around 5 orbit periods (roughly 8 hours) and the position and velocity of the satellite was recorded every 0.1 seconds. The ode45 MATLAB function was used to propagate the nonlinear equations of motion for the same time period with the state output at 0.1 second intervals and identical initial positions. The difference between the satellite's position and velocity was calculated at every recorded time epoch (no interpolation was required because a 0.1 second time interval recording was specified for both methods).

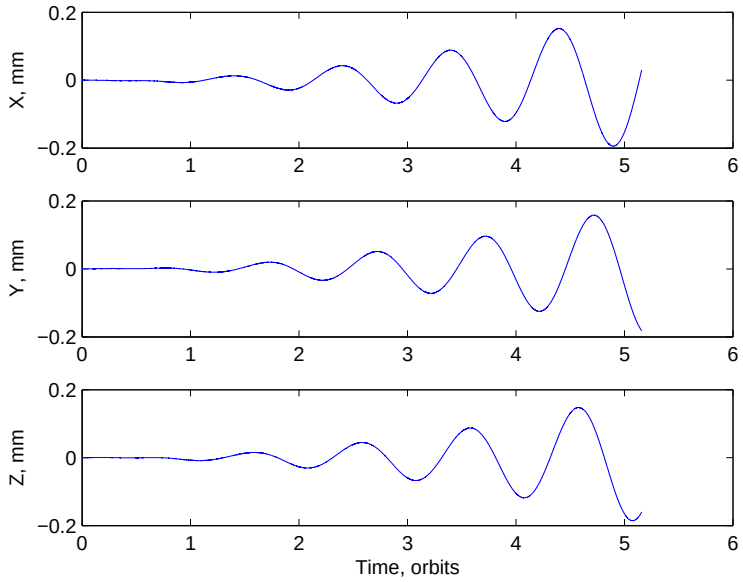

Figure 5.1: Position comparison between Trick and MATLAB nonlinear propagation in the ECI frame

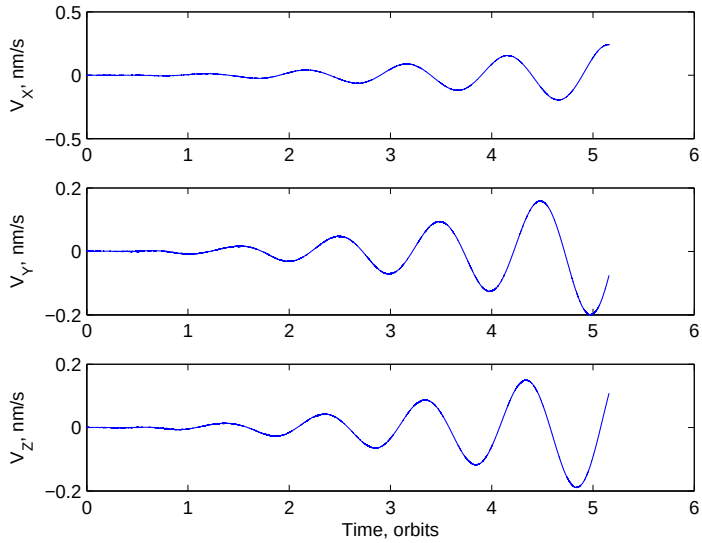

Figure 5.2: Velocity comparison between Trick and MATLAB nonlinear propagation in the ECI frame

As seen in Figs. 5.1-5.2, the difference between Trick and MATLAB is small. The position error between the two propagators is less than 1 mm, although it does increase with time. The focus of a mission manager is an AR&D maneuver which will not last long enough for the error to grow to a significant amount. The error in the velocity states is less than 1 nm/s. Again, it grows over time, but similarly it is not enough to be a significant difference.

5.2 Sensor Measurements

Values for the sensor noise parameters were chosen to demonstrate simulation functionality. Noise parameters may change depending on further research into acceptable real-world trends and capabilities. In the following sections, the error due to drift for each sensor is shown.

5.2.1 Accelerometer

The following parameters were used for the accelerometer sensor:

$$\begin{array}{rcrc} \sigma_{w,a} & = & 3.162 & \mathrm{cm/s^2} \\ \sigma_{r,a} & = & 3.162 & \mathrm{cm/s^2/\sqrt{Hz}} \\ \tau_a & = & 10 & \mathrm{min} \end{array}$$

5.2.2 GPS

The parameters for the simulated GPS sensor are given below. A subscript v refers to the GPS velocity. No graph of the GPS velocity drift is shown as it is comparable to the accelerometer and GPS position graphs.

$$\begin{array}{rcrc} \sigma_{w,p} & = & 1 & \mathrm{m} \\ \sigma_{r,p} & = & 3.162 & \mathrm{m/\sqrt{Hz}} \\ \tau_p & = & 10 & \mathrm{min} \end{array} \qquad \begin{array}{rcrc} \sigma_{w,v} & = & 1 & \mathrm{km/hr} \\ \sigma_{r,v} & = & 2.24 & \mathrm{m/s/\sqrt{Hz}} \\ \tau_v & = & 10 & \mathrm{min} \end{array}$$

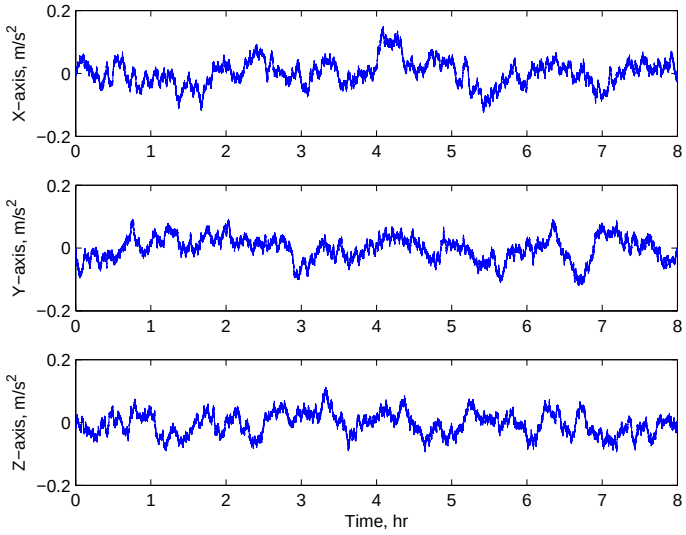

Figure 5.3: Accelerometer drift using $\sigma_{r,a} = 3.162$ cm/s$^2/\sqrt{\text{Hz}}$ and $\tau_a = 10$ min

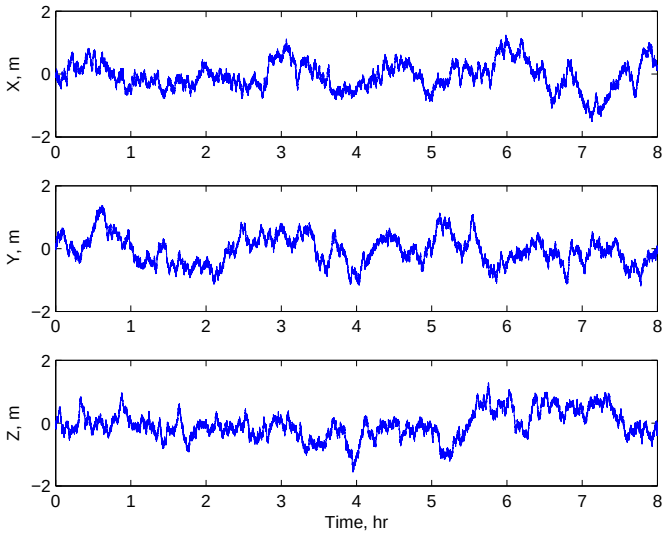

Figure 5.4: GPS position drift using $\sigma_{r,p} = 3.162$ m$/\sqrt{\text{Hz}}$ and $\tau_p = 10$ min

5.3 Kalman Filter

The GPS and accelerometer sensors were modeled as measurements with the parameters described in the previous sections. The measurements were input into the Kalman filter object's update function to perform the state estimation. At each estimate epoch, the difference between the estimated and true states was calculated. A constant 2 m/s^2 bias was added into the accelerometer measurements to demonstrate that the filter correctly estimates and removes the accelerometer bias.

It can be seen that the estimated state has an error that stays near 2 m, which is due to the noise in the GPS position measurements shown in Fig. 5.4. In Fig. 5.6 the bias estimation is centered around 2 m/s^2 as expected.

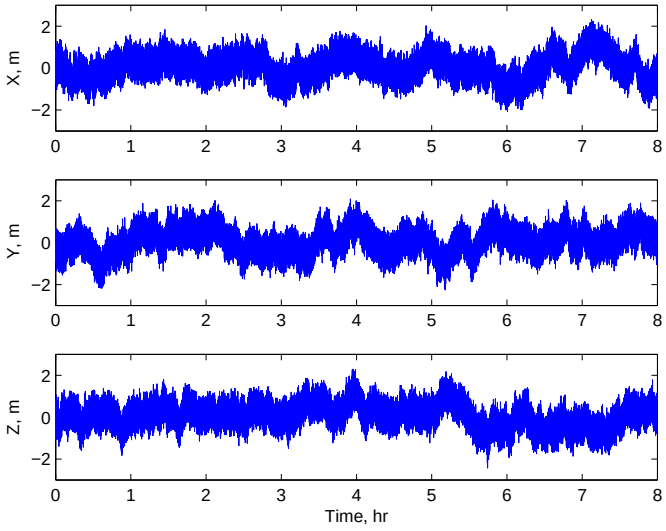

Figure 5.5: Estimated position error

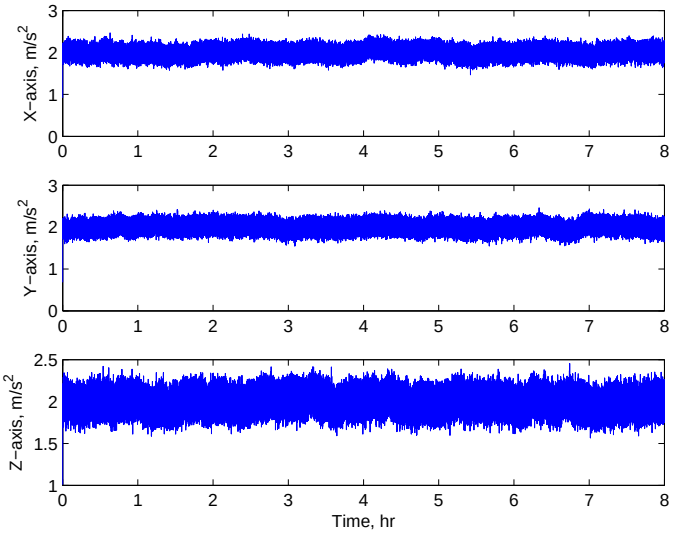

Figure 5.6: Estimated accelerometer bias

5.4 Controller

An example by D'Souza was recreated to confirm the implementation of the controller. The chaser satellite was initially at a position of 2 km ahead of the target satellite with a desired final distance of 200 m. The transfer time was defined as 2400 seconds. The results of the maneuver can be seen in Figs. 5.7-5.9. It can be seen in Fig. 5.7 that the vehicle state starts at an initial distance of 2 km in the X-axis and successfully reaches the target distance within the 2400 second transfer time. The measurements given in the testing of the controller were true measurements without noise. This was done to show that the controller was implemented correctly. The controller will operate on noisy measurements when being used in conjunction with the mission manager.

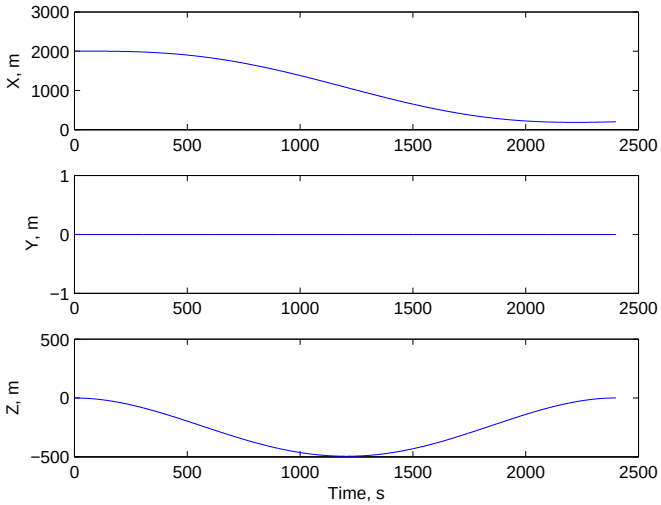

Figure 5.7: Chaser position

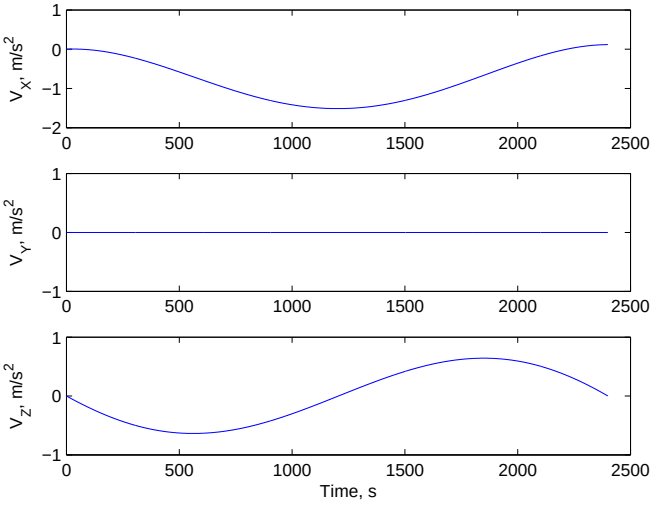

Figure 5.8: Chaser velocity

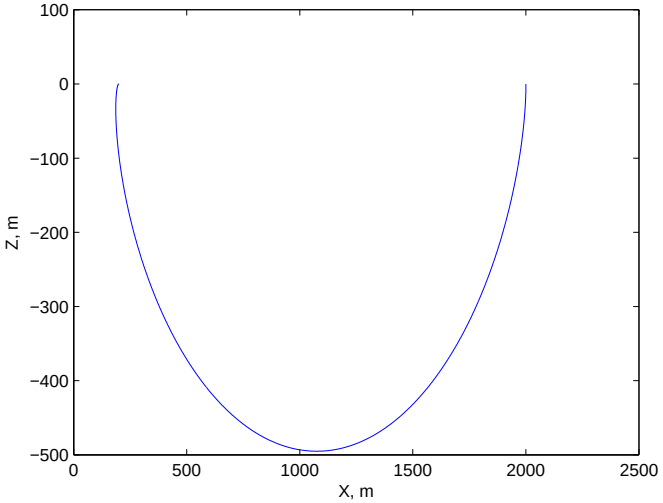

Figure 5.9: Chaser in-plane trajectory

Chapter 6

Future Work

The CARDS project is an ongoing endeavor to be completed within the next year (2015). The environmental simulation to be used for testing the mission manager has been developed and run as a closed-loop simulation. However, as this research matures, the environmental simulation will continue to improve. A non-spherical Earth and atmospheric drag has been implemented in the simulation, but were disabled for the testing scenarios in this report for an initial benchmark scenario. Further testing will implement these perturbations for a more realistic simulation.

The mission manager software, discussed in Chapter 4, is currently in development. After completion of the mission manager software, it will be tested with the Trick environmental simulation on a desktop computer. A Trick GUI will be developed to test components of the mission manager software individually. To accomplish the individual testing, the Trick GUI will implement failure modes in the simulation, such as creating false sensor masurements, so that the response of the mission manager software can be observed. Once the mission manager software has been validated on a desktop environment, it will be ported to run on an embedded system, specifically the PhyCORE®-LPC3250. Similar to the desktop testing, the Trick simulation will be used to verify the performance of the mission manager software. Finally, the mission manager software will be tested on the embedded system using measurements obtained from real hardware.

The mission manager software is planned to be flown on the Bevo-2 satellite as part of the LONESTAR-2 mission. The flight of the mission manager software on the Bevo-2 satellite is in preparation for performing AR&D operations between two small satellites as part of the LONESTAR-3 mission.

The scenario presented in this thesis was a maneuver that brought the chaser satellite from a 1 km starting distance to 1 m of the target satellite. Such a maneuver is only the initial approach for an AR&D operation. To be a fully complete AR&D operation, it will be necessary for continued development and testing of the mission manager software to perform the 1 m distance maneuver and docking procedure. This is beyond the scope of the current research for the CARDS project and is an option for expansion of the mission manager software.

Chapter 7

Conclusion

A simulation tool for testing an in development mission manager software for CubeSat AR&D operations was created. The simulation was made with the Trick software and JEOD module. The use of Trick and JEOD was verified by comparing the propagation of a satellite in Earth orbit in Trick and JEOD with an identical MATLAB simulation. The error between the two methods was determined to grow as the simulation time increased, but was determined to be negligible for short-term rendezvous scenarios. Two identical satellites were simulated in near-circular Earth orbits to demonstrate the environmental simulation. One satellite, known as the chaser, performs a control maneuver from an initial 1 km to 1 m distance to the second target satellite. The satellites were equipped with accelerometer and GPS sensors, and a Kalman filter was used to estimate the vehicle's state. The sensor models used included noise to provide imperfect knowledge for a more real-world scenario. The resulting drifts from the sensor noise models were shown. The chaser satellite was used to demonstrate the error in the estimated state. The environmental simulation will be used to test the mission manager software once the latter has been completed.

Appendix A: Assumptions

The following assumptions were made for the development of the environmental simulation.

1. The target and chaser satellites are in perfectly circular orbits around Earth.
2. The target and chaser satellites are identical in mass and instrumentation.
3. The measurement and state estimation epochs for the target and chaser satellites are perfectly aligned.
4. The target and chaser satellites are in periodic communication.

Appendix B: Dynamics Simulation Object

```
class DynamicsSimObject : public Trick::SimObject
{
public:
  DynManager       manager;
  DynManagerInit   manager_init;
  BodyAction*      body_action_ptr;
  TimeManager*     time_manager;

  DynamicsSimObject(TimeManager& ext_time)
  {
    time_manager = &ext_time;

    P_MNGR ("initialization")
      manager.initialize_model(manager_init, *time_manager);

    P_BODY ("initialization") manager.initialize_simulation();

    P_GRAV ("derivative") manager.gravitation();
    ("derivative") manager.compute_derivatives();

    ("integration", &manager.sim_integrator) trick_ret =
      manager.integrate(exec_get_sim_time(), *time_manager);
  }
};
```

Listing 1: Sample DynamicsSimObject class in an S_define file

Appendix C: Time Simulation Object

```
class TimeSimObject : public Trick::SimObject
{
public:
  TimeManager      manager;
  TimeManagerInit  manager_init;

  TimeUTC utc;
  TimeTAI tai;

  TimeConverter_Dyn_TAI conv_dyn_tai;
  TimeConverter_TAI_UTC conv_tai_utc;

  TimeConverter_TAI_UTC_tai_to_utc_default_data
    tai_utc_default_data;

  TimeSimObject(double dyn_cycle)
  {
    ("default_data")
      tai_utc_default_data.initialize(&conv_tai_utc);

    P_TIME ("initialization") manager.register_type(tai);
    P_TIME ("initialization")
      manager.register_converter(conv_dyn_tai);
    P_TIME ("initialization") manager.register_type(utc);
    P_TIME ("initialization")
      manager.register_converter(conv_tai_utc);

    P_TIME ("initialization") manager.initialize(&manager_init);
    P_TIME ("initialization")
      utc.calendar_update(exec_get_sim_time());

    (dyn_cycle, "environment")
      manager.update(exec_get_sim_time());
    (dyn_cycle, "environment")
      utc.calendar_update(exec_get_sim_time());
  }
};
```

Listing 2: Sample TimeSimObject class in an S_define file

Appendix D: Environment Simulation Object

```
class EnvSimObject : public Trick::SimObject
{
public:
  GravityModel    gravity;
  De4xxEphemeris de405;
  DynManager*     dyn_manager;
  TimeManager*    time_manager;

  De4xxEphemeris_de405_default_data de405_default_data;

  EnvSimObject(double dyn_cycle,
    DynManager& ext_dyn, TimeManager& ext_time)
  {
    dyn_manager  = &ext_dyn;
    time_manager = &ext_time;

    ("default_data") de405_default_data.initialize(&de405);

    P_ENV ("initialization")
      gravity.initialize_model(*dyn_manager);
    P_ENV ("initialization")
      de405.initialize_model(*time_manager, *dyn_manager);

    (dyn_cycle, "environment") dyn_manager->update-ephemerides();
  }
};
```

Listing 3: Sample EnvSimObject class in an S_define file

Appendix E: Earth Simulation Object

```
class EarthSimObject : public Trick::SimObject
{
public:
  Planet                              planet;
  SphericalHarmonicsGravityBody gravity_body;
  GravityModel*                       env_gravity;
  DynManager*                         dyn_manager;
  TimeUTC*                            utc;

  Planet_earth_default_data earth_planet_init;
  SphericalHarmonicsGravityBody_earth_spherical_default_data
    earth_gravity_init;

  EarthSimObject(GravityModel& ext_grav, DynManager& ext_dyn,
    TimeUTC& ext_utc)
  {
    env_gravity = &ext_grav;
    dyn_manager = &ext_dyn;
    utc         = &ext_utc;

    ("default_data") earth_gravity_init.initialize(&gravity_body);
    ("default_data") earth_planet_init.initialize(&planet);

    P_ENV ("initialization") gravity_body.initialize_body();
    P_ENV ("initialization") env_gravity->add_grav_body(gravity_body);
    P_ENV ("initialization")
      planet.register_model(gravity_body, *dyn_manager);
    P_BODY ("initialization") planet.initialize();
  }
};
```

Listing 4: Sample EarthSimObect class in an S_define file

Appendix F: Vehicle Simulation Object

```
class SVDynSimObject : public Trick::SimObject
{
public:
  Simple6DofDynBody body;
  DynBodyInitOrbit  orb_init;
  MassBodyInit      mass_init;
  DynManager*       dyn_manager;

  PlanetaryDerivedState pfix;
  LvlhDerivedState      lvlh;
  OrbElemDerivedState   orb_elem;
  RelativeDeriedState   rel;

  Force  force_extern;
  Torque torque_extern;
  SphericalHarmonicsGravityControls earth_grav_control;

  GPS          gps;
  IMU          imu;
  KalmanFilter filter;

  SVDynSimObject(double dyn_cycle, DynManager& ext_dyn)
  {
    dyn_manager = &ext_dyn;

    P_ENV ("initialization") body.initialize_model(*dyn_manager);
    P_ENV ("initialization") gps.initialize(lvlh.lvlh_frame, body);
    P_ENV ("initialization") imu.initialize(body);

    P_DYN ("initialization") pfix.initalize(body, *dyn_manager);
    P_DYN ("initialization") lvlh.initialize(body, *dyn_manager);
    P_DYN ("initialization") orb_elem.initialize(body, *dyn_manager);
    P_DYN ("initialization") rel.initialize(body, *dyn_manager);

    (dyn_cycle, "environment") pfix.update();
    (dyn_cycle, "environment") lvlh.update();
    (dyn_cycle, "environment") orb_elem.update();
    (dyn_cycle, "environment") rel.update();
```

```
    ("derivative") gps.update_noise();
    ("derivative") imu.update_noise();

    (dyn_cycle, "sensor") gps.update();
    (dyn_cycle, "sensor") imu.update();

    ("derivative") filter.derivatives(imu.body_accel);
    (dyn_cycle, "sensor")
      filter.updateState(gps.position, gps.velocity, imu.body_accel);
  }
};
```

Listing 5: Sample SVDynSimObject class in an S_define file

Bibliography

[1] Imken, T., "Design and Characterization of a Printed Spacecraft Cold Gas Thruster for Attitude Control," Master's Thesis, The University of Texas at Austin, May 2014.

[2] Kjellberg, H., "Constrained Attitude Guidance and Control for Satellites," Ph.D Dissertation, The University of Texas at Austin, December 2014.

[3] "Space Technology Roadmaps: Technology Area Breakdown Structure": http://www.nasa.gov/pdf/501627main_STR-Int-Foldout_rev11-NRCupdated.pdf [cited May 29, 2014].

[4] Mehrparvar, A., CubeSat Design Specification, Revision 13, The CubeSat Program, Cal Poly SLO, February 2014: http://www.cubesat.org/images/developers/cds_rev13_final.pdf [cited December 1, 2014].

[5] "A Proposed Strategy for the U.S. to Develop and Maintain a Mainstream Capability Suite ("Warehouse") for Automated/Autonomous Rendezvous and Docking in Low Earth Orbit and Beyond," February 2012: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20120003191_2012002775.pdf [cited May 29, 2014].

[6] Armadillo C++ linear algebra library: http://arma.sourceforge.net

[7] "Trick User's Guide," NASA Johnson Space Center, Version 13.1 dev, Houston, Texas, 2013.

[8] Jackson, A. A., and Thebeau, C. D., "JSC Engineering Orbital Dynamics (JEOD) Top Level Document," NASA Johnson Space Center, Rev. 1.3, Houston, Texas, 2013.

[9] Christian, J. A., and Lightsey, E. G., "Sequential Optimal Attitude Recursion Filter," Journal of Guidance, Control, and Dynamics, Vol. 33, No. 6, 2010.

[10] Brown, R., and Hwang, P.,Introduction to Random Signals and Applied Kalman Filtering: With MATLAB Exercises and Solutions, 3rd ed., Wiley, New York, 1997, Chaps. 2, 5, 10.

[11] D'Souza, C., "An Optimal Guidance Law for Formation Flying and Stationkeeping," AIAA Guidance, Navigation, and Control Conference, AIAA, Monterey, California, 2002.

[12] PhyCORE®-LPC3250: http://phytec.com/products/system-on-modules/phycore/lpc3250 [cited December 2, 2014].