An Open-Source, Extensible Spacecraft Simulation And Modeling Environment Framework

Andrew J. Turner

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Aerospace Engineering

August 13, 2003
Blacksburg, Virginia

Keywords:Spacecraft Simulation, Astrodynamics, Object-Oriented Design

Abstract

An Open-Source, extensible spacecraft simulation and modeling (Open-SESSAME) framework was developed with the aim of providing to researchers the ability to quickly test satellite algorithms while allowing them the ability to view and extend the underlying code. The software is distributed under the GPL (General Public License) and the package's extensibility allows users to implement their own components into the libraries, investigate new algorithms, or tie in existing software or hardware components for algorithm and flight component testing. This thesis presents the purpose behind the development of the framework, the software design architecture and implementation, and a roadmap of the future for the software package.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview of Problem

Researchers in the field of spacecraft dynamics and control are concerned with creating accurate and understandable physical models of satellites. These simulations help to verify principles of motion and to test control designs for both attitude and orbit control. Usually a student or researcher performing this modeling and analysis is forced to create a simulation from the ground-up for each new spacecraft. The alternatives are either to update and adapt the researcher's previous simulation software or to use another engineer's simulation code. This adaptation can be time consuming, tedious, and prone to errors that may not be noticed during operations, and as such may invalidate results obtained from the simulation.

Fortunately, there are numerous freeware and commercial spacecraft simulation packages available. However, these packages vary in their functionality, usefulness, and flexibility. Furthermore, they can be expensive and the researcher may be unaware of the internal operation of the simulation and must rely on the documented verification of the code.

The goal of this research is to develop an Open-Source, Extensible Spacecraft Simulation And Modeling Environment (Open-SESSAME) framework that can serve as a basis for satellite modeling and analysis. The entire collection of code provides most of the tools, libraries, and structure necessary for simulating a wide range of spacecraft while also allowing easy extension for any further desired functionality. The open-source nature of the packages means that users are able to investigate the design and operation of the code to reassure themselves of the validity of the simulator. The Open-SESSAME framework is also an active project within the large and rapidly growing open-source community.

This membership allows new functionality to be disseminated to all current and future users of the framework as it continues to grow and mature.

1.2 Spacecraft Simulators

Spacecraft Simulators are software tools that can be used by researchers, engineers, students, or managers to analyze and evaluate satellite operations and to answer questions regarding a project or product. These simulators are developed either from very specialized algorithms for a specific spacecraft (*e.g.* Earth Observer I), or for a more generalized class of satellites (*i.e.* non-rigid, tethered, *etc.*). There are usually associated tools that assist with coding operations such as linear algebra libraries, numerical integrators, or orbit packages. Together, the associated tools and derived algorithms make up the simulator code that is run with specified initial and operating conditions during the required time frame to analyze a desired characteristic. The pertinent data is then displayed through either graphical or text-based programs to give useful information to the user.

There are numerous spacecraft simulator packages that perform a wide-range of functions that are useful to satellite engineers and scientists. Such functionality includes orbit analysis, attitude analysis, formation flying, hardware-in-the-loop testing, or controller verification. For example, one could analyze the orbit of a spacecraft about a central body over long time periods to evaluate its ground track, calculate access to ground stations, and determe the visibility of other celestial bodies.

Commercial packages are available that have been thoroughly verified and generally accepted by the satellite community. The degree of functionality varies greatly between the packages, from simple two-body propagation of a point mass, to full three-dimensional simulations of constellations of satellites communicating with air and ground based assets. Within this array of packages, there are specialized applications that may appeal to different aspects of spacecraft modeling, including power or communications.

1.3 Rationale

Many students and researchers of satellite dynamics and control must independently develop software simulations each time a new research project begins. These simulations are typically built for the research task at hand and are not easily adaptable to future projects. Furthermore, many students have little experience developing simulations, or may not know where to begin, where to focus, and how to best implement components so they can be resuable between projects and for other students and engineers. The Open-SESSAME framework addresses these issues by providing a common groundwork upon which students can learn how simulators are implemented and develop their own components for use in the framework for their own research.

The Space Systems Simulation Laboratory (SSSL) at Virginia Tech in Blacksburg, Virginia is working on a number of projects that work to develop new methodologies for the simulation and analysis of spacecraft and their associated systems.[1] These projects include both hardware and software simulation techniques used in tandem to better understand the interplay of satellite dynamics with novel control and sensing strategies. As a result of the unique requirements of many of the projects, a single commercial software package has not yet fulfilled the needs of the lab. An open-source and extensible simulation framework creates a reusable basis for future simulation projects while also allowing the students and researchers to configure the simulation to their unique specifications. Furthermore, users are able to interface the simulation software with other analysis packages that may be required for their research.

1.4 Scope and Method of Development

This thesis aims to provide the reader with an overview of the physics, equations, and algorithms involved in spacecraft analysis and modeling, while leading them into the design aspects of bringing these principles together into a usable software framework. An introduction to object-oriented design strategies and their application to the framework layout is given. Most importantly, the reader will be introduced to, and learn, the internal operations of the framework, the interactions between components, and methods of using the framework for research projects. A treatment of the verification and validation is presented to reassure the reader of the accuracy of the current simulation framework and how to maintain this accuracy while implementing new algorithms. Finally, the goal of this thesis is to give the reader the ability to begin using the framework while understanding its underlying operation and design.

1.5 Outline of Thesis

The rest of this thesis document is organized as follows. Chapter 2 covers the development of spacecraft simulation software packages as well as a brief introduction to object-

[1]http://www.aoe.vt.edu/research/groups/sssl/

oriented programming. Chapters 3 and 4 give in-depth discussions of attitude and orbit dynamic equations that form the basis of the Open-SESSAME framework tools and libraries. Numerical methods of integration, interpolation, and calculation are covered in Chapter 5. Chapter 6 introduces the framework software design, operation and use. Chapter 7 discusses verification and validation efforts that demonstrate the accuracy of the simulation and methods to ensure that accuracy is maintained, as well as why the framework fulfills its design requirements. Finally, Chapter 9 covers the conclusions reached in developing this framework and suggests features that could be implemented by users of Open-SESSAME.

Chapter 2

Background

This chapter discusses the work preceding the development of the Open-SESSAME framework. Its purpose is to provide the context for the development of such a framework. A brief introduction to simulation research is presented followed by a survey of previous and current spacecraft simulation packages that are available and how the Open-SESSAME framework fits within this group. Finally, a brief introduction to Object-Oriented Design is covered to help in understanding the methodologies used in the design and implementation of the Open-SESSAME framework.

2.1 Prior Work in Simulation

There is a large body of research that is concerned with the development of simulation concepts and methods. The most applicable history dates back to the mid-1960's as computers became prevalent and better analysis and modeling software was developed.

In 1979, Cellier developed a numerically sound methodology for simulating hybrid continuous and discrete time models using digital computers [1]. Following this work, there were various efforts to implement simulation specific programming languages such as Dymola [2, 3], Desire [4], and Mathwork's Simulink [5]. Dymola and Desire were specialized solutions that do not find wide use in the industry today. Simulink (and by association MatLab) are widely used as analysis and modeling tools for engineering research.

Cubert and Fishwick [6] developed MOOSE (Multimodeling Object-Oriented Simulation Environment), which focused on developing a framework for modeling multiple body objects and, more importantly, creating a basis for sharing models through the MOOSE Model Repository (MMR). The design allowed development of models in any number of programming languages. These models could then be shared over the internet. MOOSE is one of the first significant efforts in creating a shared development of simulation materials that can be quickly and readily disseminated to users.

MOOSE provides the common interface to which these models from the MMR were attached and simulated. There was reuse in the interface of MOOSE as well as in the development of the models, as any model could be built up from smaller models. This building of complex models from simple models follows one of Booch's [7] principles of a complex system: "A complex system that works is invariably found to have evolved from a simpler system that works .... A complex system designed from scratch never works and cannot be patched up to make work."

Another important aspect of MOOSE was the ability to distribute the processing and operation of the simulation. Using a model similar to CORBA (Common Object Request Broker Architecture), components were either local or distributed, which did not alter the operation of the simulation. This premise was also greatly assisted by the advent of the internet and the new ease of interconnectedness between remote computers and facilities.

The future of simulation software is focused on distributed computing and employing the power of the internet to share data, models, and computing power. Object-oriented design is just one paradigm that has been leveraged to develop powerful applications that are usable and maintainable by users. As the user base grows, so does the power of the application and the ability to reuse code in developing newer software with more power in less time [8].

2.2 Prior Work in Spacecraft Simulation

There are numerous implementations of spacecraft simulation packages; most applications are proprietary or are not maintained. However, a handful of software codes are currently available as options to spacecraft engineers. These can be broadly grouped into two categories: free-of-cost/freeware and commercial packages.

2.2.1 Freeware Packages

WinOrbit is a Microsoft Windows freeware application that was developed in Visual Basic by Carl Gregory at the University of Illinois in Urbana-Champaign. WinOrbit can graphically display satellite positions in real-time. The software can also generate tracking data and ephemerides information for a number of Earth-based satellites. WinOrbit appears to have stopped development in 1998 and the code is not open-source [9].

SaVi (short for Satellite Visualization) is an open-source satellite constellation visualization that is being hosted on the Sourceforge repository[1][10], which promotes development amongst worldwide programmers. *SaVi* is being developed by Lloyd Wood, a student at the University of Surrey, UK. It has been used for a variety of applications in networking among satellites in a constellation [11, 12, 13].

Another open-source toolkit is *ORSA*: Orbit Reconstruction, Simulation and Analysis.[14]. The software is still under development by students at Padova University in Italy and does not currently have many of its features implemented. The primary goal is the simulation and analysis of celestial bodies, but because ORSA is open-source, the software could be used as a basis for a broader space simulation package.

NASA Jet Propulsion Laboratory (JPL) engineers have been developing several spacecraft simulation tools that form the Autonomy Testbed Environment (ATBE) which is built on LIBSIM and DARTS / DSHELL (DARTS Shell) [15]. The ATBE was created to test and verify autonomous spacecraft flight software on the ground. DSHELL is a library of C++ simulation routines that provides the basic framework to develop such packages as the ATBE and other spacecraft simulators. DARTS is a flexible multi-body dynamics computational engine, which also includes libraries of hardware models. DARTS is interfaced through DSHELL. Furthermore, DSHELL is portable from desktop systems to hardware-in-the-loop environments [16].

DARTS/DSHELL has been used for several NASA JPL projects such as Cassini, Galileo, Mars Pathfinder, and Stardust [15]. LIBSIM was used by the New Millenium Project's Deep Space 1. The software package is available free-of-charge to qualifying academic institutions.

Princeton Satellite Systems (PSS) has developed MultiSatSim (MSS), which can simulate up to 8 satellites as well as control them from a remote computer [17]. Unlike most other satellite simulation tools, MSS is not restricted to modeling systems orbiting about Earth. While the gravity model and control can be customized, the simulation only models a rigid body using the quaternion kinematic representation. Furthermore, MSS is not open-source, and binaries are only available for Apple brand computers.

[1]http://sourceforge.net/projects/savi

Table 2.1: Summary of spacecraft simulation software libraries and applications.

Package	**Manufacturer**	**Benefits**	**Negatives**
AutoCon	NASA Goddard	Heritage, Assists mission planning	Not maintained, not available
DSHELL	NASA JPL	Free to academic institutions	Not available, new language
Formation Flying Testbed	NASA Goddard	Well supported, ties in with hardware	Expensive, limited availability, aimed towards formations
FreeFlyer	a.i. solutions	Scriptable, good user interface, heritage, 2D/3D visualization	Expensive, limited integration with existing software
MultiSatSim	Princeton Satellite Systems	Good graphics, easy to use interface, scriptable	Apple hardware only, limited to 8 satellites, limited expandability
ORSA	Open-Source	multiple platforms, active development	Not complete, limited functionality, orbits only
Open-SESSAME	Open-Source	multiple platforms, extensible, well documented, active development, orbit and attitude, seperate libraries	No graphical user interface, requires knowledge of C++ programming
SATCOS	SAIC	Heritage	Orbit and constellations only
SaVi	Open-Source	Good user interface, in development	Only models orbits, made for constellations, single developer
SC Modeler	AVM Dynamics	-	No longer supported
Spacecraft Control Toolbox	Princeton Satellite Systems	Well developed, documented	MatLab only, attitude only
Satellite ToolKit	Analytical Graphics	Easy to use, 2D/3D visualization, heritage, large number of modules, good support, external communcations, variety of operating systems	Expensive, not extensible, complex, proprietary
Swingby	Computer Sciences Corporation	–	Not Available, now exists in STK Astrogator
WinOrbit	Carl Gregory, Univ. Illinois	Free of cost, graphical user interface	Windows only, not extensible, user interface difficult

2.2.2 Commercial Packages

Princeton Satellite Systems (PSS) has also developed the *Spacecraft Control Toolbox*, a collection of MatLab scripts that assist in the development and simulation of spacecraft attitude control systems [18]. The cost of the toolbox is about $1000 for academic users and upwards of $3000 for the full, commercial license of Spacecraft Control Toolbox.

The first of the surveyed commercial packages, *SC Modeler*[2] developed by AVM Dynamics is a collection of software tools for the design, visualization and analysis of satellite constellations. While the application is primarily used for communication constellations, it also includes tools for analyzing ground-space operations. SC Modeler is closed-source and has a high purchase cost.

SATCOS, or the Satellite Constellation Synthesis code, was developed by SAIC to assist in designing satellite constellations for telephone and Internet communication applications. First contracted by the U.S. Air Force for the space-based laser defense program, the software now optimizes global coverage and network constraints of satellite constellations for clients. [3]

AutoCon was developed by NASA Goddard Space Flight Center (GSFC) and A.I. Solutions as a satellite autonomous maneuver planning software. There are two principal components, AutoCon-F and AutoCon-G, which are used for in-flight operation and ground simulation respectively. This sharing of parts enables use of the same code on the ground and in flight, which reduces complexity and increases reliability. AutoCon was used on the Landsat-7/EO-1 formation mission to coordinate the tight formation of the spacecraft orbits. It is currently planned for use on Global Precipitation Measurement (GPM) constellation [19, 20]

FreeFlyer is another Windows based application [21] developed by a.i. solutions. Features include both limited orbit and attitude simulation, a highly customizable environment, and external scripting for control and operations. While the application has extensive functionality, the user interface is difficult to navigate, which hinders the usefulness of some operations.

Another formation simulation software package currently in development at NASA GSFC is the *Formation Flying TestBed* (FFTB). It too is meant for a real-time modeling system for providing simulated positions of formations of spacecraft. It is implemented in MatLab with extensions for external hardware interfacing [22].

Satellite ToolKit (STK) is a commercial package developed by Analytical Graphics Inc.

[2]http://www.avmdynamics.com/index1.htm
[3]http://www.saic.com/cover-archive/space/satcos.html

(AGI)[4] and includes numerous modules that include communications, visualization, coverage, and a complex orbit analysis tool, Astrogator. It is a comprehensive suite that is quickly gaining acceptance in the aerospace industry and has been used for several high visibility missions such as MAP, NEAR[23], Sirius Satellite Radio, Loral's GlobalStar, and Hughes AsiaSat3 satellite rescue [24, 25]. STK provides a Graphical User Interface (GUI) through which users perform simulation tasks or network programs for communication between remote machines.

Two shortcomings of STK are its high cost (up to $10,000 per module) and its closed source. While the basic STK program is free of charge, the modules are expensive. Although there are educational discounts offered to institutions, the add-on modules are not readily available to interested researchers and students, especially for students who are not part of an established research group, or engineers who cannot afford the cost. Furthermore, the program is produced as a commercial product and must maintain simulation accuracy and speed. However, closed-source software prevents student and engineers from understanding STK's internal operation and using the developed tools for specific tailored applications. STK does provide a basis for a good satellite modeling program, and interaction with it for further analysis is recommended if possible.

The *Swingby* program was developed in 1989 at Computer Sciences Corporation (CSC) for NASA GSFC. In January 1994, *Swingby* was used operationally for the Clementine mission. Later that same year, CSC worked with AGI to enhance this program and commercially sell it as a product called Navigator [26].

Swingby continued to be used operationally for the WIND launch in 1994 and the SOHO launch in 1995. In early 1997 at the request of GSFC, Analytical Graphics Inc. began the conversion of *Swingby* into a new product, *Astrogator*, with a prototype delivered in late 1997. Following its release, it was used to plan the lunar gravity swingby which rescued Hughes' AsiaSat3 from a useless orbit. In March 1998, GSFC began beta testing *Astrogator* and in January 1999 they began using it for MAP mission analysis. *Astrogator* was brought to the commercial market in November 1999.[5]

Table 2.1 gives an overview of the previously discussed software packages. The reader is encouraged to learn more about these applications and how they work. Certain packages offer benefits over others, and individual users may have different operating requirements.

This survey of available spacecraft simulation packages may not be complete; however it does offer a understanding of the current open-source, free-of-charge, and commercial software options for satellite engineers. The Open-SESSAME framework, as mentioned

[4]http://www.stk.com

[5]http://www.stk.com/resources/download/astrogator/about_astrogator.cfm

previously, is released under an open-source license. The next section presents some of the concepts of object-oriented design which make open-source software easier to understand and reuse in new applications.

2.3 Object-Oriented Design

Object-Oriented Design (OOD) is a relatively recent design approach to developing software. Its primary purpose is to model digital data and algorithms using real-world analogies. Objects, which are defined by *classes*, are encapsulations of data and methods that affect or are affected by that associated data.

Objects can be conceptualized in a manner similar to real world objects, or things. A car is an object that has data associated with it, such as the number of wheels and doors, mileage, color, and size. A car also has operations that can be performed, like start, stop, go, turn left, turn right, or paint. Data is usually hidden (not directly accessible) from the user but accessed using operations. We can change the internal representation without affecting how the user interfaces with the object by encapsulating the data within a class.

To illustrate OOD with an example, assume there is a **Car** class which stores the speed of the car in "Miles Per Hour" (MPH). There is an operation, *GetSpeedMPH()*, much like a function, we can call that returns the speed in miles per hour. However, requirements are changed to state that the speed should internally be stored as Kilometers Per Hour (KPH). The class operation *GetSpeedMPH()* is internally changed to convert the internal mileage from KPH to MPH. A user of the **Car** class does not need to know about this internal change, since the user still calls *GetSpeedMPH()*, which returns the speed of the car in MPH.

The practice of programming using OOD is called Object-Oriented Programming (OOP). The purpose is to design the classes in such a way as to make a simple, usable interface while preventing the users from being affected by eventual changes to the internal operations of the class. Furthermore, OOP helps design software that is extendable. Object-oriented designs lend themselves to hotspots, or extension points where new programmers can add new functionality to existing software with a minimal of effort and reusing as much software as possible.

The Open-SESSAME framework makes extensive use of object-oriented paradigms to assist in the code's understandability and reusability. It is imperative that the user understand basic software programming principles, and preferably more advanced design practices. For a full introduction and discussion of Object-Oriented Programming and other design paradigms refer to Cohoon and Davidson[27] or Stroustrup [28].

2.4 Summary

This chapter presented an in-depth coverage of past and current spacecraft simulation codes, as well as some general simulation research as it applies to object-oriented simulation. These codes range from open-source and beginning development to full-fledged commercial software applications used by major corporations for high-visibility spacecraft missions. None of them, however, currently fill the need for an open-source, extensible spacecraft simulation and modeling environment framework that can also be used for hardware-in-the-loop testing. Lastly, a brief overview of object-oriented programming was given to familiarize the reader with the general concepts. The next chapters discuss the technical details of the physics that are the basis of the software framework.

Chapter 3

Attitude Dynamics

Attitude is used to describe the orientation of one reference frame to another reference frame, such as a spacecraft body-fixed frame with respect to an Earth-fixed frame. To fully describe an attitude, a set of reference frames and the methods for representing the orientation of these frames with respect to one another are defined. The kinematics and dynamics of these frame rotations are also defined. This chapter also discusses the various environmental disturbance torques, as well as internal and control torques. Lastly, the methods of analytically and numerically calculating attitude dynamics are presented.

3.1 Reference Frames

A reference frame is a set of three orthogonal vectors in space that are used to describe a set of coordinates. To define a frame, one of the vector directions must be specified, a preferred, or desired, direction for a second vector is chosen, and the third direction is determined by right-handed orthogonality. There are numerous reference frames to be used when describing spacecraft attitude. These frames can be highly dependent on the mission scenario, operating characteristics, or project standards. Most attitude simulations and analyses are done with respect to spacecraft-fixed coordinates (origin moving with the spacecraft), but may include non-spacecraft reference frames for further analysis. The following subsections define frequently used examples of attitude frames.

3.1.1 Inertial Frame

An inertial frame is a non-rotating reference frame in fixed space. A common representation is Earth-Centered Inertial (ECI) frame which is illustrated in Figure 3.1. The $\hat{x}^i$ direction points from the center of the Earth to the vernal equinox, Υ, the $\hat{z}^i$ direction is in the Earth's orbital angular velocity direction, and $\hat{y}^i$ completes the orthonormal triad to $\hat{x}^i$ and $\hat{z}^i$. However, inertial frames can be defined with respect to any celestial body or arbitrary point in space. The inertial frame is used as a reference to describe an attitude that is independent of spatial position or mission operation.

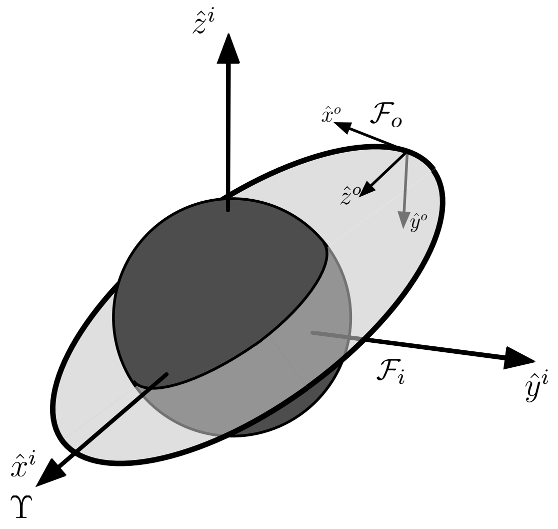

Figure 3.1: Illustration of the orbital, $\mathcal{F}_o$, and inertial, $\mathcal{F}_i$, reference frames in an orbit about a central body.

3.1.2 Orbital Frame

The orbital frame is a non-inertial, rotating frame that moves with a body in orbit. As illustrated in Figure 3.1, the origin is fixed at the spacecraft's mass center with the $\hat{z}^o$ axis in the direction from the spacecraft to the Earth (nadir direction). The $\hat{y}^o$ axis is the direction opposite to the orbit normal, and $\hat{x}^o$ completes the orthonormal triad to $\hat{z}^o$ and $\hat{y}^o$. Note that this frame is non-inertial because of orbital acceleration and the rotation of the reference frame. The orbital reference frame can used as a reference for relating a spacecraft's attitude relative to the local orbit horizon (shaded area in Figure 3.1).

3.1.3 Body Frame

A body frame is a set of axes that has a fixed origin at a point in, or on, the spacecraft. The axes are then permanently described within the spacecraft as specified by the spacecraft engineers. Body frames are useful for relating objects on a spacecraft relative to one another, or for defining how a spacecraft is oriented with respect to an external frame (such as the orbital or inertial frames).

3.1.4 Principal Axes

Sometimes it is helpful in modeling spacecraft dynamics to describe the system in the principal axes frame. This frame is a specific body-fixed reference frame with the axes aligned such that the moment of inertia matrix is diagonal. These moments of inertia are then called the principal moments of inertia.

3.2 Kinematics

Kinematics describes the orientation, or rotation, of one reference frame to another. The following are common ways of defining a rotation: Euler Axis and Angle, Euler Angles, Direction Cosine Matrix, Modified Rodriguez Parameters and Quaternions.

3.2.1 Euler Axis and Angle

The simplest description relating the orientation of two reference frames is that of the unit principal axis, or Euler axis, $\hat{\mathbf{e}}$, and angle, Φ. The axis is a single vector about which the first frame can rotate through the angle to align with the second frame as shown in Figure 3.2. The axis can be represented as the principal rotation vector, γ:

$$\gamma = \Phi\hat{\mathbf{e}} \tag{3.1}$$

This equation is the formulation of Euler's theorem. As is discussed below, $\hat{\mathbf{e}}$ is the eigenaxis, or eigenvector associated with the eigenvalue of 1 from the transformation matrix corresponding to the rotation.

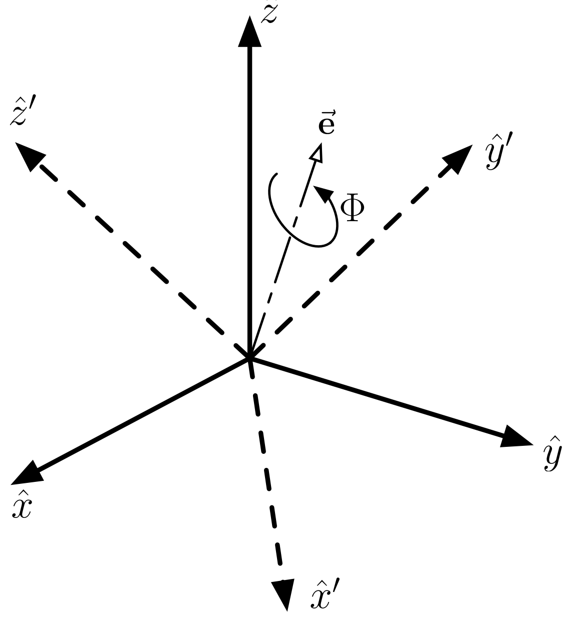

Figure 3.2: Rotation about an Euler axis, $\hat{\mathbf{e}}$, of the euler angle, Φ that transforms $\mathcal{F}$ to $\mathcal{F}'$.

3.2.2 Euler Angles

Another way of describing the orientation of two axes with respect to one another is by a sequence of angular rotations about the individual x, y, and z axes. The rotations described are those necessary to reorient axis sytem 1 with axis system 2. For example, the description of a rotation from the Earth Centered Inertial frame to the perifocal frame requires a rotation about the **z**-axis by the longitude of the ascending node, Ω, then a rotation about the new **x**-axis by the inclination, i, and finally a rotation about the new **z**-axis of the argument of perigee, ω. This set of transformations is described as a “3-1-3” rotation, which describes the order of the rotations.

There is an infinite number of rotation sequences that can be used to describe coordinate frame transformations. For most transformations, 3 rotations are sufficient, which results in 12 possible successive rotation combinations. However, specific transformations may require more or fewer successive rotations. Another important aspect to consider is that the transformation from one reference frame to another is non-unique. There may be several rotation sequences that achieve a transformation. With Euler angles there are singularities for each of the 12 combinations of three rotations. These singularities make the representation inconvenient for simulation. However, Euler angles are useful for visualization because a human can more easily perceive three rotations about axes than by using a mathematical transformation construct.

3.2.3 Direction Cosine Matrix

A simple way to describe and represent transformation is by the use of a Direction Cosine Matrix (DCM). A DCM is a 3×3 matrix of values, a rotation matrix, that represents the transformation of a vector from one coordinate frame to another:

$$\mathbf{v}^b = \mathbf{R}^{ba}\mathbf{v}^a \tag{3.2}$$

where $\mathbf{v}^a$ and $\mathbf{v}^b$ are the components of $\hat{\mathbf{v}}$ vector in $\mathcal{F}_a$ (Frame a) and $\mathcal{F}_b$, respectively, $\mathbf{R}^{ba}$ is the DCM describing the rotation from $\mathcal{F}_a$ to $\mathcal{F}_b$.

The direction cosine matrix is constructed by the angles between the frame axes:

$$\mathbf{R}^{ba} = \begin{bmatrix} \cos\theta_{x_b x_a} & \cos\theta_{x_b y_a} & \cos\theta_{x_b z_a} \\ \cos\theta_{y_b x_a} & \cos\theta_{y_b y_a} & \cos\theta_{y_b z_a} \\ \cos\theta_{z_b x_a} & \cos\theta_{z_b y_a} & \cos\theta_{z_b z_a} \end{bmatrix} \tag{3.3}$$

where $\cos\theta_{x_b x_a}$ is the cosine of the angle between the x axis of the first frame and the x axis of the second frame.

For determining successive rotations (say from $\mathcal{F}_a$ to $\mathcal{F}_b$ to $\mathcal{F}_c$), we can combine the rotation matrices by multiplying them together:

$$\mathbf{R}^{ca} = \mathbf{R}^{cb}\mathbf{R}^{ba} \tag{3.4}$$

When creating the rotation matrix using Euler angles, it is possible to combine the principal rotations. These rotations are the individual rotations through an angle θ about one of the primary axes. The principal rotations are as follows:

$$R_1(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \tag{3.5}$$

$$R_2(\theta) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \tag{3.6}$$

$$R_3(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.7}$$

Therefore, using the principal rotations, the "3-1-3" sequence described before could be determined as follows:

$$\mathbf{R}^{pi} = R_3(\omega)R_1(i)R_3(\Omega) \tag{3.8}$$

where $\mathbf{R}^{pi}$ is the rotation from the inertial frame to the perifocal coordinate system.

3.2.4 Quaternions

The 4-element quaternion set, $\bar{\mathbf{q}}$ exhibits no singularities. The quaternion, $\bar{\mathbf{q}} = [\mathbf{q}^T, q_4]^T$, can be determined from the Euler-axis parameters set ($\hat{\mathbf{e}}$, Φ) as follows:

$$\mathbf{q} = \hat{\mathbf{e}} \sin\frac{\Phi}{2} \tag{3.9}$$

$$q_4 = \cos\frac{\Phi}{2} \tag{3.10}$$

The quaternion representation vector has unit length, which is a useful characteristic. Therefore, the quaternion can be normalized during computations to maintain accuracy, $\bar{\mathbf{q}}_{new} = \frac{\bar{\mathbf{q}}}{|\bar{\mathbf{q}}|}$. Also, because the quaternion is not a unique transformation, the negative, $\bar{\mathbf{q}} = -\bar{\mathbf{q}}$, is an equivalent rotation.

3.2.5 Modified Rodriguez Parameters

Another method of specifying a rigid body attitude is through the use of Modified Rodriguez Parameters (MRP). The 3-element set is defined by using the 4-elements of the Euler axis and angle as follows:

$$\sigma = \hat{\mathbf{e}} \tan\frac{\Phi}{4} \tag{3.11}$$

Like the quaternions, the MRP is not a unique representation to the transformation, but also has a shadow set, σ^S:

$$\sigma^S = -\frac{1}{|\sigma|^2}\sigma \tag{3.12}$$

The shadow set should nominally be evaluated whenever $|\sigma| > 1$ since the shadow set will be a shorter rotational distance back to the original frame. However, this threshold can be whatever the user may desire to prevent unnecessary switching in the case where the dynamics remain close to $|\sigma| > 1$.

3.2.6 Conversions

Kinematic and dynamic equations can be formulated in different transformation representations, and sometimes it is necessary to change representations for singularities or visualization. Therefore, conversion algorithms are defined to allow the user the ability to switch between transformation representations. The following sections include common conversions between the described kinematic representations.

Quaternion to MRP
The conversion from quaternion to Modified Rodriguez Parameters ($[q_1, q_2, q_3, q_4]^T = [\sigma_1, \sigma_2, \sigma_3]^T$):

$$\sigma_i = \frac{q_i}{1+q_4} \qquad \text{for i=1,2,3} \tag{3.13}$$

When $q_4 = -1$, there is a singularity. Therefore, the equivalent quaternion should be used ($\bar{\mathbf{q}} = -\bar{\mathbf{q}}$).

MRP to Quaternion
The conversion from Modified Rodriguez Parameters to quaternion is:

$$\bar{\mathbf{q}} = \begin{bmatrix} 2\sigma_1 \\ 2\sigma_2 \\ 2\sigma_3 \\ 1-\sigma_1^2-\sigma_2^2-\sigma_3^2 \end{bmatrix} \left(1+\sigma^2\right) \tag{3.14}$$

Euler Axis to DCM
The conversion from Euler Axis and Angle to Direction Cosine Matrix is:

$$\mathbf{R} = \hat{\mathbf{e}}\hat{\mathbf{e}}^T(1-\cos\Phi) - \hat{\mathbf{e}}^\times \sin\Phi + \mathbf{1}\cos\Phi \tag{3.15}$$

$$= \begin{bmatrix} e_1^2\Sigma + \cos\Phi & e_1e_2\Sigma + e_3\sin\Phi & e_1e_3\Sigma - e_2\sin\Phi \\ e_2e_1\Sigma - e_3\sin\Phi & e_2^2\Sigma + \cos\Phi & e_2e_3\Sigma + e_1\sin\Phi \\ e_3e_1\Sigma + e_2\sin\Phi & e_3e_2\Sigma - e_1\sin\Phi & e_3^2\Sigma + \cos\Phi \end{bmatrix} \tag{3.16}$$

where $\Sigma = 1 - \cos\Phi$. It is also necessary to define the *skew-symmetric* operation, which represents the vector component version of a cross-product:

$$\mathbf{v}^\times = \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix} \tag{3.17}$$

This matrix has the property $(\mathbf{v}^\times)^T = -\mathbf{v}^\times$.

DCM to Euler Axis and Angle

The conversion from Direction Cosine Matrix to Euler axis and angle is:

$$\Phi = \cos^{-1}\left(\frac{trace(\mathbf{R})-1}{2}\right) \tag{3.18}$$

$$\hat{\mathbf{e}} = \frac{1}{2\sin\Phi}\left(\mathbf{R}^T-\mathbf{R}\right) \tag{3.19}$$

$$= \frac{1}{2\sin\Phi}\begin{bmatrix} R_{23}-R_{32} \\ R_{31}-R_{13} \\ R_{12}-R_{21} \end{bmatrix} \tag{3.20}$$

$\Phi = 0$ when $\mathbf{R} = \mathbf{1}$, and therefore the rotations are aligned. The Euler axis is undefined and can be any unit vector. There is also a singularity when $\Phi = \pi$.

Quaternion to DCM

The conversion from quaternion to Direction Cosine Matrix is:

$$\mathbf{R}\left(\bar{\mathbf{q}}\right) = \left(q_4 - \mathbf{q}^T\mathbf{q}\right)\mathbf{1} + 2\mathbf{q}\mathbf{q}^T - 2q_4\mathbf{q}^\times \tag{3.21}$$

$$= \begin{bmatrix} 1-2(q_2^2+q_3^2) & 2\left(q_1q_2+q_4q_3\right) & 2\left(q_1q_3-q_4q_2\right) \\ 2\left(q_1q_2-q_4q_3\right) & 1-2(q_1^2+q_3^2) & 2\left(q_2q_3+q_4q_1\right) \\ 2\left(q_1q_3+q_4q_2\right) & 2\left(q_2q_3-q_4q_1\right) & 1-2(q_1^2+q_2^2) \end{bmatrix} \tag{3.22}$$

DCM to Quaternion

The conversion from Direction Cosine Matrix to quaternion is:

$$q_4 = \pm\frac{1}{2}\sqrt{1+trace(\mathbf{R})} \tag{3.23}$$

$$\mathbf{q} = \frac{1}{4q_4}\begin{bmatrix} R_{23}-R_{32} \\ R_{31}-R_{13} \\ R_{12}-R_{21} \end{bmatrix} \tag{3.24}$$

However, if $q_4 = 0$, then $\mathbf{q} = \hat{\mathbf{e}}$.

MRP to DCM

The conversion from Modified Rodriguez Parameters to Direction Cosine Matrix is:

$$\mathbf{R}(\sigma) = \frac{\mathbf{1}}{(\mathbf{1}+\sigma^{\mathbf{T}}\sigma)^{\mathbf{2}}}\left[(\mathbf{1}-(\sigma^{\mathbf{T}}\sigma)^{\mathbf{2}})\mathbf{1} + \mathbf{2}\sigma\sigma^{\mathbf{T}} - \mathbf{2}(\mathbf{1}-(\sigma^{\mathbf{T}}\sigma)^{\mathbf{2}})\sigma^{\times}\right] \tag{3.25}$$

$$= \frac{1}{\left(1+\sigma^2\right)^2} \tag{3.26}$$

$$\begin{bmatrix} 4\left(\sigma_1^2-\sigma_2^2-\sigma_3^2\right)+\Sigma^2 & 8\sigma_1\sigma_2+4\sigma_3\Sigma & 8\sigma_1\sigma_3-4\sigma_2\Sigma \\ 8\sigma_2\sigma_1-4\sigma_3\Sigma & 4\left(-\sigma_1^2+\sigma_2^2-\sigma_3^2\right)+\Sigma^2 & 8\sigma_2\sigma_3+4\sigma_1\Sigma \\ 8\sigma_3\sigma_1+4\sigma_2\Sigma & 8\sigma_3\sigma_2-4\sigma_1\Sigma & 4\left(-\sigma_1^2-\sigma_2^2+\sigma_3^2\right)+\Sigma^2 \end{bmatrix}$$

where $\Sigma = 1-\sigma^2$, and $\sigma^2 = \sigma^{\mathbf{T}}\sigma$.

DCM to MRP
The conversion from Direction Cosine Matrix to Modified Rodriguez Parameters is:

$$\sigma \;=\; \frac{1}{4\Gamma(1+\Gamma)}\begin{bmatrix} R_{23}-R_{32} \\ R_{31}-R_{13} \\ R_{12}-R_{21} \end{bmatrix} \tag{3.27}$$

where $\Gamma = \pm\frac{1}{2}\sqrt{1+trace(\mathbf{R})}$, which is equivalent to q_4.

3.3 Attitude Dynamics

Attitude dynamics are the time-variation of the spacecraft attitude with respect to another reference frame due to external forces and torques. In this section, we develop the dynamic equations describing the motion of a rigid and non-rigid body in an environment subject to external disturbances. We first develop the simple rigid body dynamics before deriving the full non-rigid flexible dynamic equations.

3.3.1 Equations of Motion

The rotation of a rigid body is described by the kinematic equations of motion and the kinetic equations of motion. As discussed above, the kinematics specifically model the current attitude of the body with respect to time. The dynamics are characterized by the absolute angular velocity vector, ω.

3.3.2 Kinematic Equations of Motion

Each attitude representation discussed above has a set of equations that describe its time rate of change due to the dynamics of the rigid body.

Quaternion Kinematic Equations of Motion

The propagation of the kinematics is defined as:

$$\begin{aligned} \dot{\mathbf{q}} &= \mathbf{Q}\left(\bar{\mathbf{q}}\right)\omega && (3.28) \\ &= \frac{1}{2}\begin{bmatrix} \mathbf{q}^{\times} + q_4\mathbf{1} \\ -\mathbf{q}^T \end{bmatrix}\omega && (3.29) \end{aligned}$$

Modified Rodriguez Parameters Kinematic Equations of Motion

The MRP kinematics are defined to propagate the rigid body attitude:

$$\dot{\sigma} = \mathbf{F}(\sigma)\omega \tag{3.30}$$

where

$$\mathbf{F}(\sigma) = \frac{1}{2}\left(\mathbf{1} - \sigma^{\times} + \sigma\sigma^T - \frac{1+\sigma^T\sigma}{2}\mathbf{1}\right) \tag{3.31}$$

Euler Angle Kinematic Equations of Motion

Sometimes it is useful or required to directly integrate the Euler angles and deal with the possibility of singularities. Table 3.1 presents the 12 possible kinematic equations, one for each type of Euler angle sequence. The kinematic equation is summarized as:

$$\dot{\theta} = \mathbf{S}^{-1}(\theta)\omega \tag{3.32}$$

3.3.3 Dynamic Equations of Motion

Euler's Law defines the formulation of the angular momentum of a body in an inertial frame:

$$\dot{\vec{\mathbf{h}}} = \tilde{\mathbf{g}} \tag{3.33}$$

Table 3.1: Summary of Euler kinematic equations for various successive rotations.

Axis Sequence	**Kinematic Equation, $\mathbf{S}^{-1}$**
1-2-3, 2-3-1, 3-1-2	$\begin{bmatrix} \cos\theta_3\sec\theta_2 & -\sin\theta_3\sec\theta_2 & 0 \\ \sin\theta_3 & \cos\theta_3 & 0 \\ -\cos\theta_3\tan\theta_2 & \sin\theta_3\tan\theta_2 & 1 \end{bmatrix}$
1-3-2, 3-2-1, 2-1-3	$\begin{bmatrix} \cos\theta_3\sec\theta_2 & \sin\theta_3\sec\theta_2 & 0 \\ -\sin\theta_3 & \cos\theta_3 & 0 \\ \cos\theta_3\tan\theta_2 & \sin\theta_3\tan\theta_2 & 1 \end{bmatrix}$
1-2-1, 2-3-2, 3-1-3	$\begin{bmatrix} 0 & \sin\theta_3\sec\theta_2 & \cos\theta_3\csc\theta_2 \\ 0 & \cos\theta_3 & -\sin\theta_3 \\ 1 & -\sin\theta_3\cot\theta_2 & -\cos\theta_3\cot\theta_2 \end{bmatrix}$
1-3-1, 3-2-3, 2-1-2	$\begin{bmatrix} 0 & \sin\theta_3\sec\theta_2 & -\cos\theta_3\csc\theta_2 \\ 0 & \cos\theta_3 & \sin\theta_3 \\ 1 & -\sin\theta_3\cot\theta_2 & \cos\theta_3\cot\theta_2 \end{bmatrix}$

*Source:*From Hughes [29]

where $\vec{\mathbf{h}}$ is the angular momentum vector referenced to teh center of mass and $\vec{\mathbf{g}}$ is the applied torque. The differential equations for the angular velocity in the body frame are derived from Euler's equation:

$$\mathbf{I}\dot{\omega} = \mathbf{g} - \omega \times \mathbf{I}\omega \tag{3.34}$$

where $\mathbf{I}$ is the spacecraft moment of inertia matrix, ω is the body angular velocity, and $\mathbf{g}$ are the spacecraft torques.

To propagate the dynamics in the rotating reference frame, a relation between the body-inertial, ω^{bi}, and body-orbital, ω^{bo}, angular velocities is required:

$$\omega^{bi} = \omega^{bo} + \omega^{oi} = \omega^{bo} - \omega_c\mathbf{o}_2 \tag{3.35}$$

therefore,

$$\begin{aligned} \dot{\omega}^{bi} &= \dot{\omega}^{bo} - \omega_c\dot{\mathbf{o}}_2 - \dot{\omega}_c\mathbf{o}_2 && (3.36) \\ &= \dot{\omega}^{bo} + \omega_c{\omega^{bo}}^{\times}\mathbf{o_2} - \dot{\omega}_c\mathbf{o}_2 && (3.37) \end{aligned}$$

where ω_c is the orbital angular velocity, and $\mathbf{o}_2$ is the second column of the body-orbital rotation matrix $\mathbf{R}^{bo}$ as defined in Equation 3.21. This derivation leads to the equation of the angular velocity in body-orbital coordinates:

$$\dot{\omega}^{bo} = \mathbf{I}^{-1}\left[\mathbf{g} - \left(\omega^{bo} - \omega^{oi}\right) \times \mathbf{I}\left(\omega^{bo} - \omega^{oi}\right)\right] - \omega_c{\omega^{bo}}^{\times}\mathbf{o_2} + \dot{\omega}_c\mathbf{o}_2 \tag{3.38}$$

3.4 Disturbance Torques

The equations of motion above were derived assuming generalized torques. In a perfect environment, these torques would be limited to applied control torques. However, a real-world spacecraft is subject to many other disturbance torques from the environment. These torques can include aerodynamic, magnetic dipole, or gravity-gradient torques.

The following sections introduce and describe some of these disturbance torques and simple models for determining their effects on the attitude dynamics.

3.4.1 Aerodynamic

Most simulations model satellites in orbits about a central body with an atmosphere. The atmosphere creates disturbance torques and forces the same way it would for an aerodynamic body, but with less density due to the relatively high altitude of satellite operations.

The change in momentum of onrushing air particles imparts a force on visible sections of the spacecraft. Therefore, the spacecraft's cross-section distribution with respect to the relative atmospheric velocity vector must be calculated. Also, as the satellite's altitude decreases, the force due to the atmosphere increases because the density increases. Below 400 km the aerodynamic torque is the dominant environmental disturbance torque [30].

3.4.2 Magnetic

Spacecraft structures are typically constructed out of magnetic materials and contain numerous amounts of electronic wiring. These materials and wiring carrying electrical current produce ambient magnetic fields within and around the spacecraft. All of the magnetic fields interact with a central body's magnetic field much the way a compass behaves on the Earth. The local fields attempt to align themselves, applying a torque about the body center:

$$\vec{\mathbf{T}}_{mag} = \vec{\mathbf{m}} \times \vec{\mathbf{B}} \tag{3.39}$$

where $\vec{\mathbf{m}}$ is the spacecraft's magnetic moment due to eddy currents, hysteresis, permament and induced magnetism, or electronical current loops, and $\vec{\mathbf{B}}$ is the central body's magnetic flux density at the spacecraft's location [30].

The magnetic disturbance can also be useful for applying active control using magnetic torquers [31].

3.4.3 Gravity Gradient

The spacecraft body is subject to a non-uniform gravity field which can cause external torques about the body center of mass. This non-uniformity is due to the inverse-square relation of the force field and the distance from the mass center, as well as a non-spherical, non-homogenous central body (such as the Earth, but especially true for asteroid or irregularly shaped central bodies).

The gravity gradient torque about the body principal axes is:

$$\mathbf{T}_{gg} = 3\omega_c^2 \mathbf{o_3}^{\times} \mathbf{I} \mathbf{o_3} \tag{3.40}$$

where $\mathbf{o_3}$ is the third column of the body-orbital rotation matrix and ω_c is the orbit angular velocity of the spacecraft. The orbit angular velocity can be calculated by using the gravitational parameter, μ, and the semi-major axis, a.

$$\omega_c = \sqrt{\frac{\mu}{a^3}} \tag{3.41}$$

For enhanced accuracy, a better model would include a higher order gravity field that is dependent on the spacecraft's position and the central body's orientation. Furthermore, it is useful to analyze the spacecraft's moment of inertia matrix to evaluate its stability due to the gravity gradient disturbance torque.

3.4.4 Solar Radiation

Spacecraft are not nominally spherical, perfect bodies, but are instead a collection of flat or curved surfaces of different coloring and material. This mismatch of surfaces can create disturbance torques due to the unbalanced applied force from light particles from the sun, reflection from the central body or other nearby bodies, or radiation emitted by the central body and its atmosphere. This radiation pressure is equal to the vector difference between the incident and reflected momentum flux [30].

3.4.5 Other Disturbance Torques

There are many other disturbance torques that could be included for a more accurate attitude dynamics model. The modeler should be aware of the satellite's operating conditions and understand the pertinent disturbances to include in the model, as well as the inconsequential terms that can be neglected.

Some examples include micrometeorites, propulsion torques, propellant slosh, crew motion, or moving hardware (booms, optics, sensors).

The effects of the environmental torque disturbances discussed above are illustrated in Figure 3.3. This figure demonstrates when certain disturbances should be modeled, and when they are negligible.

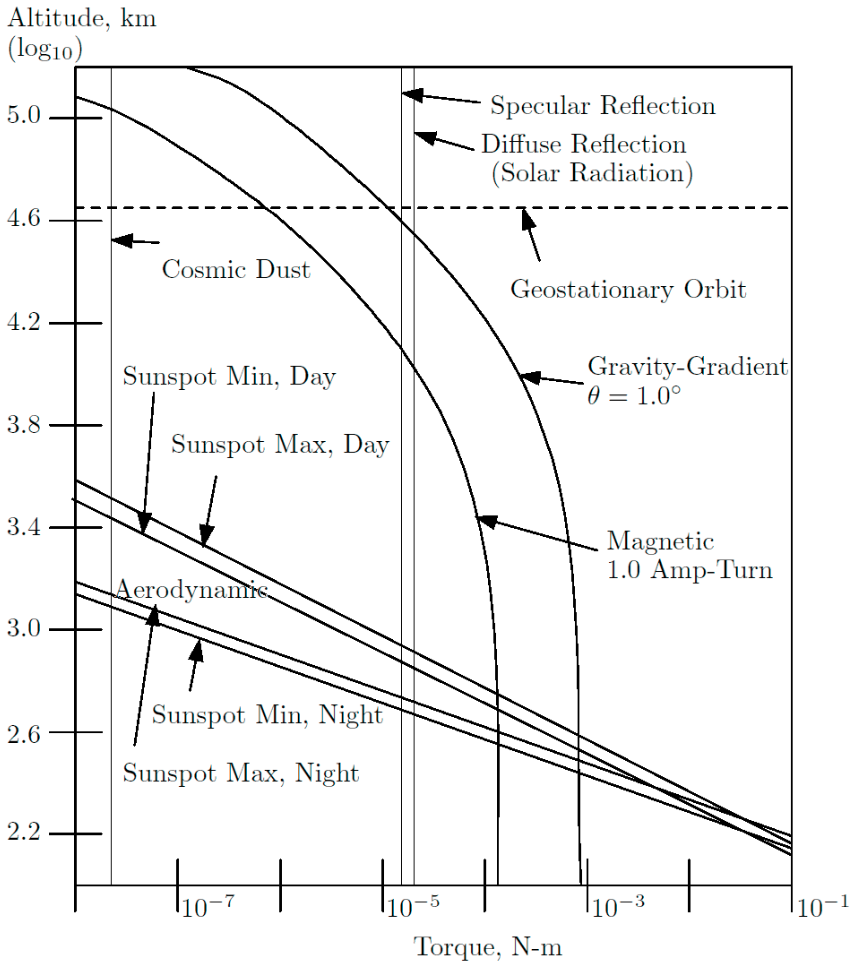

Figure 3.3: Relative effects of various example environmental disturbance torques. (From Hughes [29], adapted by Makovec[31]) It is important to know the appropriate effects to model during a simulation. Including relatively small effects add unnecessary computation time. Neglecting to include relatively large disturbances can cause the simulation results to be inaccurate.

3.5 Attitude Control

The requirements for most satellite missions specify a desired attitude with respect to some reference frame, such as earth-pointing, sun-pointing, spin-stabilized, or pointing thrusters for orbital maneuvering. Therefore, a torque needs to be applied to attain the desired attitude from the current attitude, which requires the development of suitable control algorithms for determining the requisite torque. This torque is then applied to the spacecraft for a calculated time, and then the required torque is calculated again depending on the current attitude.

There are many different methods for determining the required torque, and these methods usually must consider the method of control (momentum wheels, thrusters, torque rods, etc.). Satellite attitude control is an active area of research. For simulation purposes the end result is a set of torques being applied about the body axes. These torques are then accounted for in the dynamics equations to modify the spacecraft's attitude.

3.6 Attitude Propagation

In order to simulate a spacecraft's operation, its attitude must be propagated forward through time. Propagation requires evaluating the dynamic equations at each time step and integrating through the simulation time.

There are two primary methods of evaluating and integrating the equations of motion: dynamic modeling and gyro modeling. Dynamic modeling integrates both the kinematic and dynamic equations, while gyro modeling uses rate sensors or gyroscopes to provide the dynamics information and integrates only the kinematics equations. For both of these methods of propagation, any and all degrees of freedom must be included in the integrated state vector (*e.g.* momentum wheels, angular momentum, non-rigidity).

Choosing the kinematics representation to be propagated requires consideration. Many simulations use the quaternion equations of motion (Equation 3.28) due to its lack of troublesome singularities, but the Modified Rodriguez Parameters (Equation 3.30) are also an adequate choice.

The rest of the state vector is chosen as required by the simulation. If rate information is being supplied by sensors, then gyro modeling can be used, and only the kinematics must be integrated.

3.6.1 Methods

Chapter 5 discusses suitable techniques for integrating the equations of motion. Unlike most orbit propagation simulations, attitude simulations occur on a much smaller timescale and usually with tighter constraints. Therefore, an appropriate integration timestep or error tolerance must be chosen to ensure accurate modeling [32].

The closed-form approximations of the equations of motion can be used to assist in verifying the modeling solution. The simple case is that of an axisymmetric body, where two of the principal moments of inertia are equal, say $I_T = I_1 = I_2$, and with no externally applied torques. Euler's equations (Equation 3.34) simplify to:

$$\dot{\omega}_1 = \frac{I_T - I_3}{I_T}\omega_2\omega_3 \tag{3.42}$$

$$\dot{\omega}_2 = \frac{I_3 - I_T}{I_T}\omega_1\omega_3 \tag{3.43}$$

$$\dot{\omega}_3 = 0 \tag{3.44}$$

hence ω_3 is constant. This system of differential equations can be solved by differentiating the first with respect to t, multiplying by I_T, and substituting into the second equation to give:

$$\ddot{\omega}_1 = -\left(\frac{I_T - I_3}{I_T}\right)^2 \omega_1\omega_3^2 \tag{3.45}$$

$$\Rightarrow \omega_1 = \omega_T \cos\omega_p(t - t_1) \tag{3.46}$$

where $\omega_T = \sqrt{\omega_1^2 + \omega_2^2}$ and is the maximum value of ω_1. The variable t_1 is the time at which ω_1 first reaches ω_T and ω_p is the body nutation rate:

$$\omega_p = \left(1 - \frac{I_3}{I_T}\right)\omega_3 \tag{3.47}$$

These equations can be summarized as follows:

$$\omega_1 = \omega_{01}\cos\omega_p t + \omega_{02}\sin\omega_p t \tag{3.48}$$

$$\omega_2 = \omega_{02}\cos\omega_p t - \omega_{01}\sin\omega_p t \tag{3.49}$$

$$\omega_3 = \omega_{03} \tag{3.50}$$

where ω_{01}, ω_{02}, ω_{03} are the components of the initial angular velocity vector ω_0.

This closed-form solution can be used to verify a numerically integrated solution to verify the operation of the simulation. It is also useful to derive the closed-form solutions to the equations with time-varying torque, $\mathbf{g} = \mathbf{g}(t)$ and assuming $g_3 = 0$ [29]:

$$\begin{aligned} \omega_1 &= \omega_{01}\cos\omega_p t + \omega_{02}\sin\omega_p t \\ &\quad + \frac{1}{I_T}\int_0^t [g_1(\tau)\cos(\omega_p(t-\tau)) + g_2(\tau)\sin(\omega_p(t-\tau))]d\tau \end{aligned} \tag{3.51}$$

$$\begin{aligned} \omega_2 &= -\omega_{01}\sin\omega_p t + \omega_{02}\cos\omega_p t \\ &\quad + \frac{1}{I_T}\int_0^t [-g_1(\tau)\sin(\omega_p(t-\tau)) + g_2(\tau)\cos(\omega_p(t-\tau))]d\tau \end{aligned} \tag{3.52}$$

$$\omega_3 = \omega_{03} \tag{3.53}$$

Therefore, to calculate the angular velocities at a future point in time, the equations above can be integrated using the known control input, initial conditions, and body parameters.

More in-depth derivations can produce the closed-form solutions of the asymmetric, torque-free case using Jacobi elliptic functions [30].

3.6.2 Coupling with Orbit Maneuvers

Most of the torque disturbances become dependent on the spacecraft position as the fidelity of the environment model is increased. For instance, it is required to know the position to calculate the local magnetic field. Orbital position is also required to determine if the spacecraft is in eclipse and the solar radiation pressure should or should not be applied. For these reasons, attitude and orbit propagation should occur in tandem.

It is useful to also discuss how to model the attitude and orbit dynamics equations since they are typically on very different timescales. It would be computationally wasteful to integrate the orbit dynamics on the same small timescale as the attitude. Therefore, as is discussed in the next chapter, it is more useful to integrate the orbit at larger timescales and interpolate between these integration mesh points to evaluate the environmental torque disturbances that are dependent on position [33].

3.7 Summary

This chapter discussed the main points of interest for modeling and simulating attitude dynamics. It presented the important concepts of attitude reference frames and corresponding kinematics, as well as the dynamic equations of motion. An introduction to environmental disturbance torques was shown, as well as resources for a more in-depth coverage. The principles covered in this chapter form the basis for the spacecraft simulation framework's attitude toolkit, while also allowing the user to add refined models as required.

Chapter 4

Orbit Dynamics

This chapter covers spacecraft orbital dynamics. This survey of orbital dynamics provides an understanding of the principles behind the spacecraft simulation framework acting as a springboard for developing better models. First, time frames and conversions are presented followed by an in-depth coverage of spatial coordinate systems and state representations. Next, the equations of motion and the pertubations to the ideal orbits are developed using several different models. Finally, propagation methods are discussed.

4.1 Time

Normally, time is considered a trivial issue and measured with a clock, maybe a precise one, but with little extra consideration. When someone specifies a time, such as "11:30PM on September 13, 1998" they are defining an *epoch*, an instant in time, in mean solar time. There are, however, many problems associated with the measurement of time based on the choice reference object, the accumulation of leap seconds, or the rotation of the reference frame. The differences between these time frames may be small, but because space objects move with such a high velocity, these small time differences can account for large differences in position. Therefore, it is necessary to define and relate the different time frames that are used in astrodynamics.

4.1.1 Time Frames

Solar Time

Solar Time, measured from a nominal longitude line, is the time required for an observer on Earth at the meridian to revolve once and observe the sun at the same location. Greenwich Mean Time (GMT) is therefore the measurement of solar time from the Greenwich meridian at 0° longitude.

The means of describing these observations and movements is through the use of *hour angles*. An hour angle is the elapsed time since the object was overhead of the observers longitude. It is important to note that the definition of the hour angle is *left-handed*, and therefore it is measured positive westward. The two common measurements are the Greenwich Hour Angle (GHA) and the Local Hour Angle (LHA), and can be measured in hours or degrees, as long as they are consistent.

The Earth's orbit, however, does have a small eccentricity and Earth has an inclination with respect to its orbital plane which causes the length of each day to vary by a small amount. Apparent solar time is simply defined as

$$\text{local apparent solar time} = LHA_{\odot} + 12\,h \tag{4.1}$$

and

$$\text{Greenwich apparent solar time} = GHA_{\odot} - \alpha_{\odot} + 12\,h \tag{4.2}$$

which are defined for the Earth-Sun system, and $\alpha_{\odot}$ is the right ascension of the Sun as measured positive to the east in the equator's plane from the vernal equinox direction.

To help correct for the known errors in assuming to rotation or changes in orbit, the U.S. Naval Observatory has defined Mean solar time, and is based upon Greenwich Sidereal Time (GST), which is discussed in the next Section.

Sidereal Time

Sidereal Time is similar to solar time. However, sidereal time uses a defined set of objects (*e.g. stars*) that are outside our solar system at a much greater distance than the Sun, and therefore, the objects have less change over the course of a year.

The set of axes that the observations are taken from must be specified to formally define sidereal time. The rotation axis is through the north pole of Earth (or the central body) and is positive counter-clockwise. The time is then measured from a specified longitude line to the reference axis, which for sidereal time is the vernal equinox. Similar to solar time, the sidereal time measured at the 0° longitude line is Greenwich Sidereal Time, θ_{GST}. The sidereal time at any defined longitude line is Local Sidereal Time, θ_{LST}. To convert between the two sidereal times, a simple equation is required:

$$\theta_{LST} = \theta_{GST} + \lambda \tag{4.3}$$

where λ is the specified longitude to measure as local, and is positive for east longitudes and negative for west longitudes.

Another important consideration is that the reference, the vernal equinox, is defined as the intersection of the Earth's equator with the orbit ecliptic, both of which are moving. Therefore, an even more detailed distinction must be made. Mean Sidereal Time is defined by the mean motion of the equinox with only secular terms (time varying terms, such as precession), while Apparent Sidereal Time is defined by both the secular and periodic terms of the motion.

Universal Time

Universal Time (UT) is defined as the mean solar time at the Greenwich meridian. There are inherent errors in the measure of unversal time due to inaccuracies in the measurement of the sun's motion. Therefore, a different method is used that measures the locations of radio galaxies with higher precision to determine the solar time. This time reference is known as UT0 and is observed at a particular Earth location:

$$UT0 = 12\,h + GHA_{\odot} = 12\,h + LHA_{\odot} - \lambda \tag{4.4}$$

where λ is the longitude of the observer.

More precise modifications account for polar motion of Earth (or the central body) and is used to calculate UT1:

$$UT1 = UT0 - (x_p \sin(\lambda) + y_p \cos(\lambda)) \tan(\phi_{gc}) \tag{4.5}$$

where ϕ_{gc} is the geocentric latitude of the observing location, and x_p and y_p are coefficients of the instantaneous positions of the central body's pole.

Finally, there is UT2, a highly accurate Universal Time measurement that accounts for seasonal variations. This time is used for very accurate orbit determination and modeling such as spacecraft that have precise requirements for observation or formation flying. However, this time reference is beyond the current scope and is not discussed further here.

Coordinated Universal Time

Atomic Time (AT) is the measurement of time based on the specific quantum transitions of electrons in a cesium-133 atom. The transition causes photons of a known frequency to be emitted and can be counted. AT forms the basis of the coordinated universal time, UTC which follows $UT1$ within $\pm 0.9\,s$ and is calculated:

$$UTC = UT1 - \delta UT1 \tag{4.6}$$

where $\delta UT1$ is a correction that includes leap seconds that are added by the U.S. Naval Observatory every couple of years to account for variations in the Earth's rotation.[1]

Julian Date

The Julian date is a measure of time that combines the date and time into a succinct representation. It is the amount of time, in days, since the epoch of January 1, 4713 B.C. at 12:00. A Julian period is 7980 Julian years, which are each 365.25 days. The epoch was determined from the combination of three calendars that were combined to form the Julian date that all shared the common year 4713 B.C [34].

To calculate the Julian Date within the time perdiod March 1, 1900 to February 28, 2100:

$$\begin{aligned}\text{Julian Date} &= 367(year) - floor\left(\frac{7\left[year + \text{floor}\left(\frac{month+9}{12}\right)\right]}{4}\right)\\ &\quad + \text{floor}\left(\frac{275month}{9}\right) + day + 1,721,013.5 \\ &\quad + \frac{\frac{\left(\frac{seconds}{60} + minute\right)}{60} + hour}{24}\end{aligned} \tag{4.7}$$

where the *floor* function is truncation (floor (4.587) = 4) and the year (all four digits), month, day, hour, minute, seconds are the known date and time to be converted. Furthermore, it is important to specify the time used to calculate the Julian Date: JD_{UT1}.

Dynamic Time

Many of the time representations discussed still do not take into account many variations of the Earth and the respective frames such as variable rotation and relativistic effects. Therefore, a set of dynamic times were developed based on more stable references.

[1]http://tycho.usno.navy.mil/gps_datafiles.html

Terrestrial dynamical time, TDT, is independent of equations of motion and derived directly from the International Atomic Time, TAI:

$$
\begin{aligned}
UTC &= UT1 - \delta UT1 && (4.8)\\
TAI &= UTC + \delta AT && (4.9)\\
TDT &= TAI + 32.184s && (4.10)
\end{aligned}
$$

where $\delta UT1$ and δAT are accumulated measurements of time corrections for the given time frame that are published by the US Naval Observatory and other references.

Barycentric dynamical time, TDB, is measured from the solar system's barycenter and depends on dynamical theory which includes relativistic effecs. The full relation is as follows:

$$
\begin{aligned}
T_{TDB} &= T_{TDT} + 0.001\,658 \sin M_\oplus + 0.000\,013\,85 \sin 2M_\oplus && (4.11)\\
&\quad + \text{lunar/planetary terms} + \text{daily terms} &&\\
M_\oplus &\approx 357.527\,723\,3^\circ + 35,999.050\,34 T_{TDB} && (4.12)
\end{aligned}
$$

4.1.2 Time Conversions

Apparent and Mean Solar Time

The difference between the apparent and mean solar time is defined as the *equation of time*. This correction comes about because of the difference between the true Sun's right ascension and the mean motion fictitious Sun's right ascension:

$$
\begin{aligned}
EQ_{time} &= -1.914\,666\,471^\circ \sin(M_\odot) - 0.019\,994\,643 \sin(2M_\odot) && (4.13)\\
&\quad + 2.466 \sin(\lambda_{ecliptic}) - 0.0053 \sin(4\lambda_{ecliptic}) && (4.14)
\end{aligned}
$$

where $M_\odot$ is the mean anomaly of the Sun.

Solar Time and Sidereal Time

There is a difference in the measurement of solar time versus sidereal time since one sidereal day is 24 sidereal hours where:

$$
\begin{aligned}
1 \text{ solar day} &= 1.002\,737\,909\,350\,795 \text{ sidereal day} && (4.15)\\
1 \text{ sidereal day} &= 0.997\,269\,566\,329\,084 \text{ solar day} && (4.16)
\end{aligned}
$$

However, to introduce the motion of the equinox and the variation of the relation due to this motion, we must define a new equation:

$$\begin{aligned} 1 \text{ solar day} &= 1.002\,737\,909\,350\,795 + 5.9006 \times 10^{-11} T_{UT1} \qquad (4.17) \\ &\quad -5.9 \times 10^{-15} T_{UT1}^2 \text{ sidereal day} \qquad (4.18) \end{aligned}$$

where T_{UT1} is the number of Julian centuries (UT1) since the J2000 epoch [34].

Julian Date and Universal Time

Because several time conversions measure the Julian Date from a certain epoch, it is necessary to calculate a relation between the Julian Date and a particular type of time, in this case the year 2000:

$$T_{xxx} = \frac{JD_{xxx} - 2,451,545.0}{36,525} \qquad (4.19)$$

The time definitions are necessary to specify observations and satellite states with high precision due to the high velocities of spacecraft. The choice of time representation varies depending on the application and information accessible. The following sections present the orbit states that describe a spacecraft at an instant in time.

4.2 Orbital State

The orbital state is the description of the current trajectory of the spacecraft relative to a defined reference frame or coordinate system. The following sections define the standard units and representations, as well as a brief overview of some of the more commonly used coordinate systems.

4.2.1 Canonical Units

Since the specific values of various astronomical parameters vary due to small pertubations as well as improvements in our ability to accurately measure them, there can be some confusion as to the “correct value” of a fundamental quantity. One means of addressing this issue is to use *Canonical Units*. These units are normalized quantities of astronomical values based upon the representation of a value, rather than the value itself. For example, the distance between the earth and the sun can be one “distance unit” and the mass of the sun as one “mass unit.”

An specific example may help illustrate the point. Define the radius of a hypothetical reference orbit that is circular about the earth (denoted by the subscripted astrological symbol, $\oplus$) to be $1DU_\oplus$ and the time unit, $1TU_\oplus$ such that the velocity of the satellite in the reference frame is $1DU_\oplus/TU_\oplus$. The gravitational parameter is then $\mu = 1DU_\oplus^3/TU_\oplus^2$ and does not have to change with increasingly better measurement techniques.

4.2.2 Coordinate Systems

Table 4.1 presents some common orbit reference frames as well as their definitions of the primary axes directions. It is meant as a general overview of available frames but is not a complete listing of all possible orbit frames.

4.2.3 State Representations

At least six elements must be included to fully define a three-dimensional trajectory. However, depending on the application and algorithms involved, there are many representations that can define a state using six elements.

Position and Velocity

The most common representation of the orbital state is through position and velocity vectors:

$$\begin{aligned}\vec{\mathbf{r}} &= x\hat{\imath} + y\hat{\jmath} + z\hat{k} \\ \mathbf{r} &= [r_1, r_2, r_3]^T\end{aligned}$$

$$\begin{aligned}\vec{\mathbf{v}} = \dot{\vec{\mathbf{r}}} &= \dot{x}\hat{\imath} + \dot{y}\hat{\jmath} + \dot{z}\hat{k} \\ \mathbf{v} = \dot{\mathbf{r}} &= [v_1, v_2, v_3]^T\end{aligned}$$

where the components are the vector component values in some specific reference frame. As mentioned in Section 3.2.3, to transform from the representation of a specific vector in one frame to teh same vector represented in another frame, one must define rotation matrices that can be used to calculate the new components.

Table 4.1: Common orbit reference frames and their definitions.

System	**Symbol**	**Origin**	**Fundamental Plane**	**Principal Direction**	**Example Use**
			Interplanetary Systems		
Heliocentric	XYZ	Sun	Ecliptic	Vernal equinox	Patched conic
Solar system	$X_B Y_B Z_B$	Barycenter	Invariable plane	Vernal equinox	Planetary motion
			Earth-based Systems		
Geocentric	ECi	Earth	Earth equator	Vernal equinox	General
Earth-Moon	$I_S J_S K_S$	Barycenter	Invariable plane	Earth	Restricted three-body
Earth-Centered Earth-Fixed	ECEF	Earth	Earth Equator	Local meridian	Observations
Topocentric Horizon	SEZ	Site	Local horizon	South	Radar observations
Topocentric Equitorial	$I_t J_t K_t$	Site	Parallel to Earth equator	Vernal equinox	Optical observations
			Satellite- or Orbit-based Systems		
Perifocal	PQW	Earth	Satellite orbit	Periapsis	Processing
Satellite radial	RSW	Satellite	Satellite orbit	Radial vector	Relative motion, Pertubations
Satellite Normal	NTW	Satellite	Satellite orbit	Normal to velocity vector	Pertubations
Equinoctial	EQW	Satellite	Satellite orbit	Calculated vector	Pertubations
Roll-Pitch-Yaw	RPY	Satellite	Satellite orbit	Radial vector	Attitude maneuvers

*Source:*From Vallado [34, pg. 46]

Another important point arises when relating an inertial reference frame to a rotating reference frame:

$$\vec{\mathbf{r}}^{inertial} = \mathbf{R}^{ir}\vec{\mathbf{r}}^{rot} \tag{4.20}$$

$$\vec{\mathbf{v}}^{inertial} = \vec{\mathbf{v}}^{rot} + \vec{\omega}^{ri} \times \vec{\mathbf{r}}^{rot} + \vec{\mathbf{v}}_{origin} \tag{4.21}$$

$$\begin{aligned}\vec{\mathbf{a}}^{inertial} &= \vec{\mathbf{a}}^{rot} + \dot{\vec{\omega}}^{ri} \times \vec{\mathbf{r}}^{rot} + \vec{\omega}^{ri} \times \left(\vec{\omega}^{ri} \times \vec{\mathbf{r}}^{rot}\right) \\ &\quad +2\vec{\omega}^{ri} \times \vec{\mathbf{v}}^{rot} + \vec{\mathbf{a}}_{origin}\end{aligned} \tag{4.22}$$

where $\vec{\omega}^{ri}$ is the angular rate of the rotating reference frame with respect to the inertial system, $\mathbf{R}^{ir}$ is the transformation matrix from the rotating to inertial frame, and $\vec{\mathbf{v}}_{origin}$ and $\vec{\mathbf{a}}_{origin}$ are the velocity and acceleration of the rotating reference frame's origin with respect to the inertial frame.

Reference Frame Transformations

When using the position and velocity vectors in component form, care must be taken to ensure the vectors are being represented in the same frame. If this is not the case, or if the vectors are required in a different frame, then they must be transformed to the new frame using a transformation:

$$\mathbf{r}^{\mathbf{b}} = \mathbf{R}^{ba}\mathbf{r}^{a} \tag{4.23}$$

where $\mathbf{R}^{ba}$ can be successive rotations, *e.g.* $\mathbf{R}^{ba} = R_3(\theta_1) R_1(\theta_2) R_2(\theta_3)$.

Table 4.2.3 provides a summation of standard reference frame transformations.

The symbol θ_{LST} is the angle at local standard time (measured from the vernal equinox). The geodetic latitude is ϕ_{gd} and f_r is a retrograde factor, which is +1 when $0° \leq i \leq 90°$ or -1 when $90° < i \leq 180°$. The orbital parameters i, ω, Ω, and u are discussed in the next section.

Classical Orbital Elements

The classical orbital elements, or Keplerian elements, consist of the standard 6 values associated with an orbit: semimajor axis (a), eccentricity (e), inclination (i), longitude of the ascending node (Ω), argument of perigee (ω), and the parameter true anomaly (ν), which can be used in place of a measure of time. These values are shown in figure 4.1.

Table 4.2: Common orbit frame coordinate transformations.

Coordinate Transformation	Successive Rotations
SEZ → IJK	$R_3(-\theta_{LST})R_2(-(90^\circ - \phi_{gd}))$
IJK → SEZ	$R_2(90^\circ - \phi_{gd})R_3(\theta_{LST})$
PQW → IJK	$R_3(-\Omega)R_1(-i)R_3(-\omega)$
IJK → PQW	$R_3(\omega)R_1(i)R_3(\Omega)$
PQW → RSW	$R_3(\nu)$
RSW → PQW	$R_3(-\nu)$
EQW → IJK	$R_3(\Omega)R_1(i)R_3(-f_r\Omega)$
IJK → EQW	$R_3(-f_r\Omega)R_1(-i)R_3(\Omega)$
RSW → IJK	$R_3(-\Omega)R_1(-i)R_3(-u)$
NTW → IJK	$R_3(-\Omega)R_1(-i)R_3(-u)R_3(-\phi_{fpa})$
PQW → SEZ	$R_2(90^\circ - \phi_{gd})R_3(\theta_{LST})R_3(-\Omega)R_1(-i)R_3(-\omega)$
RSW → RPY	$R_2(\pi)$

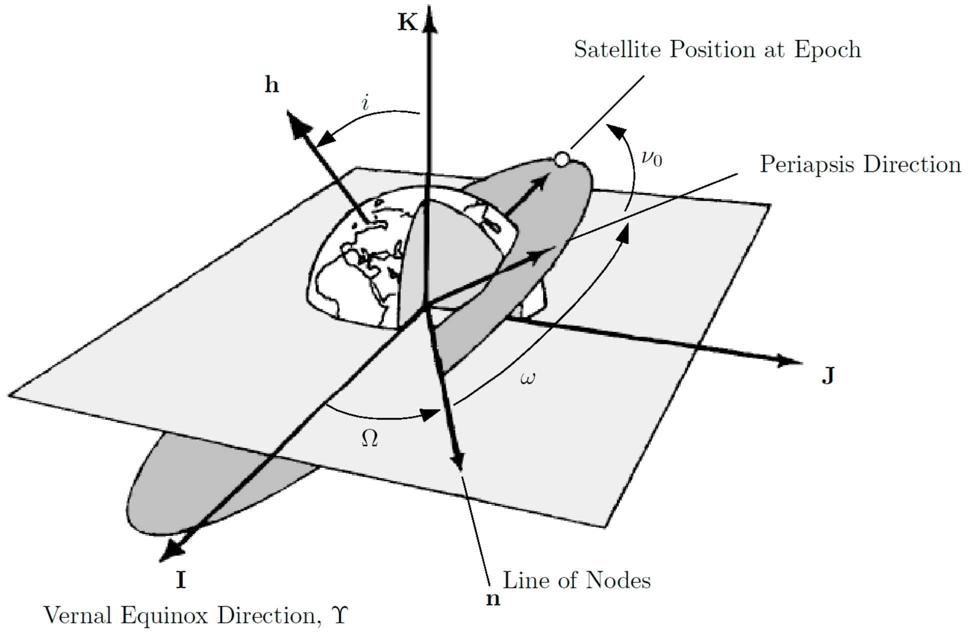

Figure 4.1: Keplerian orbital elements in the Earth-Center Earth-Fixed frame. From Bate, Mueller, and White[35, pg. 59]

The semimajor axis, a, is the distance from the center of an ellipse to the farthest end of the ellipse, defined by the intersection with the line that passes through both foci. The semimajor axis can be determined from the energy of the orbit, or by using the geometric relation of half the length of the entire ellipse:

$$a = \left(\frac{2}{r} - \frac{v^2}{\mu}\right)^{-1} = \frac{r_a + r_p}{2}. \tag{4.24}$$

The gravitational parameter, μ, is used to simplify the equation. As is discussed later, $\mu = GM$, where G is the gravitational constant, and M is the mass of the central body.

Conversely, the radii of apoapsis and periapsis can be determined from the semimajor axis, and eccentricity, e. The radius, r, of a spacecraft's position from the central body's center of mass is:

$$r = \frac{a(1-e^2)}{1 + e\cos\nu} \tag{4.25}$$

This equation can be used to formulate the radii at the closest and farthest points, since the tru anomaly, nu, is 0 at periapsis (closest), and π radians at apoapsis (farthest):

$$r_p = \frac{a\,(1-e^2)}{1+e} = a\,(1-e) \tag{4.26}$$

$$r_a = \frac{a\,(1-e^2)}{1-e} = a\,(1+e) \tag{4.27}$$

Here r_a and r_p are the radii of apoapsis and periapsis respectively.

The eccentricity, e, defines the shape of the orbit ellipse. Eccentricity is equal to the proportion of the distance from the center of the orbit to a focus on the semimajor axis, a. For example, $e = 0$ for a circular orbit since the distance from the center of the circle to a focus is 0. Eccentricity can be calculated using position and velocity:

$$\vec{\mathbf{e}} = \left(v^2 - \frac{\mu}{r}\right)\vec{\mathbf{r}} - (\vec{\mathbf{r}} \cdot \vec{\mathbf{v}})\,\vec{\mathbf{v}} \tag{4.28}$$

And therefore the eccentricity is $e = |\vec{\mathbf{e}}$.

The inclination, i, is defined as the tilt of the orbital plane with respect to the central body's equatorial plane. It is measured by the angle between the unit vector, $\hat{k}$ of the reference frame and the angular momentum vector, $\vec{\mathbf{h}}$, of the orbit:

$$\cos i = \frac{\hat{\mathbf{k}} \cdot \vec{\mathbf{h}}}{\left|\hat{\mathbf{k}}\right| \left|\vec{\mathbf{h}}\right|} \tag{4.29}$$

and can range between 0° and 180°, where 0° and 180° are equatorial orbits, inclinations of 0° to 90° are prograde, or direct, orbits, and 90° to 180° are retrograde orbits since they are orbiting in the opposite direction of the spin direction of the central body.

The longitude of the ascending node, Ω, is the angle in the equatorial plane measured positively from the $\hat{\imath}$ unit vector of the inertial reference frame (ECI) to the location of the orbit's ascending node, where the ascending node is the point on the equatorial plane that the satellite crosses from the southern hemisphere into the northern hemisphere. The line connecting the ascending node and the point at which the satellite crosses the equator going from north to south, or descending node, is the line of nodes, which is defined by the following:

$$\vec{\mathbf{n}} = \hat{k} \times \vec{\mathbf{h}} \tag{4.30}$$

From equation 4.30, one can calculate the longitude of the ascending node:

$$\cos\Omega = \frac{\hat{\imath} \cdot \vec{\mathbf{n}}}{|\hat{\imath}|\,|\vec{\mathbf{n}}|} \tag{4.31}$$

$$\text{if } (n_j < 0) \text{ then } \Omega = 360° - \Omega \tag{4.32}$$

To determine whether Ω is within the correct quadrant, one inspects the sign of the **j**-component of $\vec{\mathbf{n}}$. If it is negative then Ω lies in Quadrant III-IV. In the case of an equatorial orbit, Ω is undefined.

The argument of periapsis, ω, is the angle between the ascending node and the point of periapsis, where the satellite is at the closest approach to the central body:

$$\cos\omega = \frac{\vec{\mathbf{n}} \cdot \vec{\mathbf{e}}}{|\vec{\mathbf{n}}|\,|\vec{\mathbf{e}}|} \tag{4.33}$$

$$\text{if } (e_k < 0) \text{ then } \omega = 360° - \omega \tag{4.34}$$

The argument of periapsis is undefined for a circular orbit, since there is no periapsis.

The true anomaly, ν, is the angle between periapsis and the satellite's current position in the orbit:

$$\cos\nu = \frac{\vec{\mathbf{e}} \cdot \vec{\mathbf{r}}}{|\vec{\mathbf{e}}|\,|\vec{\mathbf{r}}|} \tag{4.35}$$

$$\text{if } (\vec{\mathbf{r}} \cdot \vec{\mathbf{v}} < 0) \text{ then } \nu = 360° - \nu \tag{4.36}$$

If the orbit is circular, the true anomaly is undefined. See Section 4.3.3 for alternate representations to true anomaly.

Special Cases

A few special cases arise in the case of circular, and/or equatorial orbits. For the case of an elliptic, uninclined orbit, the true longitude of periapsis, $\tilde{\omega}_{true}$, is used:

$$\begin{aligned} \cos\tilde{\omega}_{true} &= \frac{\hat{\imath}\cdot\vec{\mathbf{e}}}{|\hat{\imath}|\,|\vec{\mathbf{e}}|} && (4.37) \\ \text{if } (e_j < 0) \text{ then } \tilde{\omega}_{true} &= 360^\circ - \tilde{\omega}_{true} && (4.38) \end{aligned}$$

Another special case is a circular inclined orbit. There is no periapsis from which to measure the argument of periapsis in a circular orbit. Subsequently the argument of latitude, u, is used and is measured between the ascending node and the satellite's position:

$$\begin{aligned} \cos u &= \frac{\vec{\mathbf{n}}\cdot\vec{\mathbf{r}}}{|\vec{\mathbf{n}}|\,|\vec{\mathbf{r}}|} && (4.39) \\ \text{if } (r_k < 0) \text{ then } u &= 360^\circ - u && (4.40) \end{aligned}$$

The last special orbit case is that of circular equatorial orbits. These orbits use the true longitude, λ_{true}, which is the angle measured from the $\imath$-axis to the satellite's position:

$$\begin{aligned} \cos\lambda_{true} &= \frac{\hat{\imath}\cdot\vec{\mathbf{r}}}{|\hat{\imath}|\,|\vec{\mathbf{r}}|} && (4.41) \\ \text{if } (r_j < 0) \text{ then } \lambda_{true} &= 360^\circ - \lambda_{true} && (4.42) \end{aligned}$$

Orbital Elements to Position and Velocity

It is useful to have a conversion from orbital elements to position and velocity vectors. Defining some orbital parameters if they are undefined due to the special cases mentioned above is also necessary. The following rules are used:

1. If circular equatorial, set(ω,Ω) $= 0$ and $\nu = \lambda_{true}$,
2. If circular inclined, set $\omega = 0$ and $\nu = u$, and
3. If elliptical equatorial, set $\Omega = 0$ and $\omega = \tilde{\omega}_{true}$.

Then the conversions to the vectors in perifocal coordinates can be evaluated:

$$\mathbf{r}^p = \begin{bmatrix} \frac{p\cos\nu}{1+e\cos\nu} \\ \frac{p\sin\nu}{1+e\cos\nu} \\ 0 \end{bmatrix} \qquad (4.43)$$

$$\mathbf{v}^p = \begin{bmatrix} -\frac{\mu}{p}\sin\nu \\ \frac{\mu}{p}(e+\cos\nu) \\ 0 \end{bmatrix} \qquad (4.44)$$

where $p = a\,(1-e^2)$ and is known as the semi-latus rectum.

Equinoctial Elements

Another set of orbital parameters, based closely on the classical orbit elements, are the equinoctial elements. These parameters are attractive since they are not prone to the special geometry cases (*i.e.* singularities) mentioned such as for circular and equatorial orbits.

$$\begin{aligned}
k_e &= e\cos(\omega + f_r\Omega) \\
h_e &= e\sin(\omega + f_r\Omega) \\
a & \quad \text{semi-major axis} \\
\lambda_M &= M + \omega + f_r\Omega \\
p_e &= \tan f_r \frac{i}{2}\sin\Omega \\
q_e &= \tan f_r \frac{i}{2}\cos\Omega
\end{aligned} \tag{4.45}$$

where f_r is a retrograde factor and is +1 for prograde orbits, and -1 for retrograde orbits.

2-line Element Sets

Since the satellite orbital elements are widely used and must be stored, it is necessary to define a common syntax to compile and transmit the parameters. Historically, computers and communications were limited in computing power and bandwidth, and therefore a compact representation was developed called "2-line Element Sets" or TLE. Refer to Celestrak for more information and current TLEs for satellites. [2]

4.3 Equations of Motion

The equations of motion used in modern orbital dynamics are based on Newton's Laws of Motion. Newton's second law, "The rate of change of momentum is proportional to the force impressed and is in the same direction." [35] is expressed as follows:

$$\sum \mathbf{F} = m\ddot{\mathbf{r}} \tag{4.46}$$

where $\sum \mathbf{F}$ is the vector sum of all the forces acting on a mass m, and $\ddot{\mathbf{r}}$ is the vector acceleration of the mass measured relative to an inertial reference frame. These forces

[2]http://celestrak.com/NORAD/elements/

are the summation of an infinite number of external disturbances, foremost the gravity of the central body but also the solar-radiation pressure, atmospheric drag, and so on.

The following sections derive the basic equations that are typically used to develop the full equations of motion.

4.3.1 Two-Body Equation

Newton also formulated the simplified two-body equation, or Law of Universal Gravitation. This formulation is a simplified model because it only accounts for two bodies, the central body and the spacecraft. In general it can be applied to any two massive bodies that have a gravitational attraction with the following assumptions:

1. The bodies are spherically symmetric;
2. There are no external or internal forces acting on the system other than the gravitational forces that act along the line joining the centers of the two bodies.

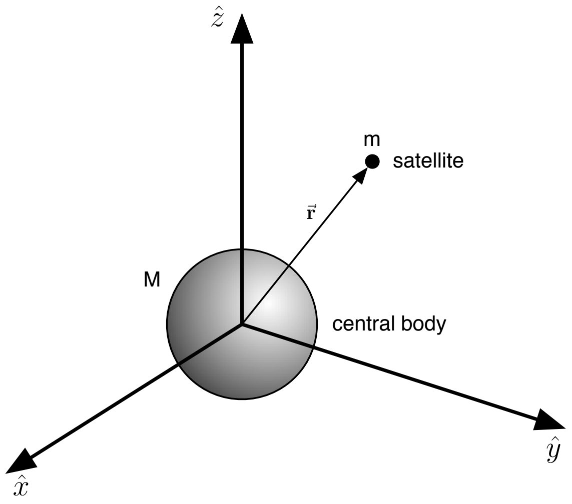

Figure 4.2: Two body gravity diagram.

Newton's Law of Universal Gravitation states that the force of gravity between two bodies as shown in Figure 4.2 is proportional to the product of their masses and inversely proprtional to the square of the distance between them:

$$\mathbf{F}_g = -\frac{GMm}{r^2}\frac{\mathbf{r}}{r} \tag{4.47}$$

where $\mathbf{F}_g$ is the force of gravity acting on mass M and m, and the vector between the two masses is $\vec{\mathbf{r}} = \vec{\mathbf{r}}_M - \vec{\mathbf{r}}_m$. The parameter G is the universal gravitational contant, which is usually measured by observing the quantity $Gm_{\oplus}$ since the mass of the earth is large and more easily measured. This gravitational parameter, $\mu = Gm_{\oplus}$, has a modern (most recent, accurate) value of $3.986\,004\,415 \times 10^5 km^3/s^2$ for Earth.

Based on equations 4.46 and 4.47, the following equations:

$$M\ddot{\mathbf{r}}_M = \frac{GMm}{r^2}\frac{\mathbf{r}}{r} \tag{4.48}$$

$$m\ddot{\mathbf{r}}_m = -\frac{GMm}{r^2}\frac{\mathbf{r}}{r} \tag{4.49}$$

(note the lack of minus sign on the equation for M due to the directions of $\vec{\mathbf{r}}$) can be combined to derive:

$$\ddot{\mathbf{r}} = -\frac{G(M+m)}{r^3}\mathbf{r} \tag{4.50}$$

A common assumption for simulating satellites orbiting central bodies is that the mass of the central body (M) is much larger than the mass of the satellite (m) and therefore, $G(M+m) \approx GM$. The gravitation parameter μ, simplifies the equations of motion to:

$$\ddot{\mathbf{r}} + \frac{\mu}{r^3}\mathbf{r} = 0 \tag{4.51}$$

4.3.2 N-body Equations

As mentioned previously, Equation 4.51 is for motion between only two bodies. However, there are multiple bodies that affect gravity on the modeled satellite. The two-body assumption is incorrect, and to fully model the dynamics of a body in space, all of the force effects must be accounted. For a system of n bodies, each of which has an inverse gravitational field, is defined. Consider the system of masses $m_1, m_2, m_3 ... m_n$ where we are determining the force on mass m_i. We use Equation 4.47 to calculate the force of gravity, $\mathbf{F}_g$, from one mass, m_j, on the observed mass, m_i:

$$\mathbf{F}_g = -\frac{Gm_im_j}{r_{ji}^3}\mathbf{r}_{ji} \tag{4.52}$$

where

$$\mathbf{r}_{ji} = \mathbf{r}_i - \mathbf{r}_j \tag{4.53}$$

Therefore, as determined by Newton's Second Law, the force of all the masses on the observed mass, m_i, is:

$$\mathbf{F}_g = -Gm_i \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{m_j}{r_{ji}^3}\mathbf{r_{ji}} \tag{4.54}$$

4.3.3 Kepler's Equation and Problem

A different formulation of orbit equations can be derived using the geometry of the orbit (be it circular, elliptical, parabolic, or hyperbolic). This was first approached by Johannes Kepler to predict the motion of planets and the moon. Kepler's equation is used to determine the relation of time and angular displacement in an orbit. It introduces the idea of the mean anomaly, M, which corresponds to the uniform angular motion on a circle:

$$M = E - e\sin E = \sqrt{\frac{\mu}{a^3}}(t - t_p) \tag{4.55}$$

where E is the eccentric anomaly, t_p is the time of periapsis passage (closest approach to the central body), and t is the time of flight. Equation 4.55 can be simplified by calculating the mean motion, n, which is the mean angular rate of the orbital motion:

$$n = \sqrt{\frac{\mu}{a^3}} \tag{4.56}$$

$$\Rightarrow M = n\,(t - t_p) \tag{4.57}$$

It is useful to determine a general form of Kepler's equation:

$$\frac{M - M_0}{n} = t - t_0 \tag{4.58}$$

$$= \sqrt{\frac{a^3}{\mu}}\left[2\pi k + E - e\sin E - (E_0 - e\sin E_0)\right] \tag{4.59}$$

$$\sin E = \frac{\sqrt{1-e^2}\sin\nu}{1 + e\cos\nu} \tag{4.60}$$

$$\cos E = \frac{e + \cos\nu}{1 + e\cos\nu} \tag{4.61}$$

However, to calculate the eccentric anomaly, given $t - t_0$, an iterative process is required. There are several methods for finding the solution, such as Newton-Raphson, that approximate the solution by using a root-finding iterative process until a desired convergence tolerance is reached. To perform the iterations, we must derive the formula for the iteration and the subsequent equations with appropriate derivatives.

We define an approximation of the function, $f(x)$, as:

$$f(x) = P_n(x) + R_n(x) \tag{4.62}$$

where P_n is the function approximation, and R_n is the remainding error of the approximation. By using a Taylor's series expansion, the approximation function is defined as:

$$P_n(x) = f(x_0) + f'(x_0)\delta + \frac{f``(x_0)\delta^2}{2!} + ... + \frac{f^{(n)}(x_0)\delta^n}{n!} \tag{4.63}$$

$$\text{define} \qquad \delta = x - x_0 \approx -\frac{f(x_0)}{f'(x_0)} \tag{4.64}$$

Using only the first order term of the expansion, we define an iterative algorithm to calculate the term, x_n, and apply it to finding the eccentric anomaly:

$$x_{n+1} = x_n + \delta_n = x_n - \frac{f(x_n)}{f'(x_n)} \tag{4.65}$$

$$\Rightarrow E_{n+1} = E_n + \frac{M - E_n + e \sin E_n}{1 - e \cos E_n} \tag{4.66}$$

Equation 4.66 is used for solving Kepler's equation. In most cases, only several iterations should be required to converge to an acceptable range. The iterative approximation can then be used with the following relations to determine the true anomaly or distance to the satellite:

$$\cos \nu = \frac{\cos E - e}{1 - e \cos E} \tag{4.67}$$

$$r = a\,(1 - e \cos E) \tag{4.68}$$

Kepler's problem uses the solution to Kepler's equation to propagate a satellite through an orbit. The problem solution requires one to first to determine the orbital elements from the position and velocity vectors as developed in Section 4.2.3. These elements are used to determine the eccentric anomaly (using eccentricity and true anomaly). Kepler's equation (4.55) can be used to determine the original mean anomaly, M_0, and the new mean anomaly can be determined from:

$$M = M_0 + n(t - t_0) \tag{4.69}$$

Thus Kepler's equation is solved for the new eccentric anomaly that is then converted back to true anomaly for the orbit, and determine the new position and velocity vectors (Equations 4.43 and 4.44).

4.3.4 Constants of Motion

It is useful to know and understand some of the constants of motion for a satellite in orbit for the 2-body problem. These constants can be used for derivation and verification of different principles for future applications.

The specific angular momentum is conserved through an orbit:

$$\vec{\mathbf{h}} = \vec{\mathbf{r}} \times \vec{\mathbf{v}} = constant \tag{4.70}$$

The specific mechanical energy is also conserved and is the sum of mechanical energy, $v^2/2$, and potential energy, $-\mu/r$ as shown below:

$$\mathcal{E} = \frac{v^2}{2} - \frac{\mu}{r} = -\frac{\mu}{2a} \tag{4.71}$$

4.4 Pertubations

Previously, the equations of motion of a body with idealized forces, such as gravity, with no other disturbances have been derived and shown. This is not an accurate model of the actual dynamics of a body in space. To create a more accurate simulation one must evaluate and include deviations, or pertubations, from the idealized model.

Pertubations come in many forms such as deviations of the gravity model, atmospheric drag, solar radiation pressure, or controlled thrust. Furthermore, the derivation of the pertubations can come from an analytical formulation, known as general pertubations, or through numerical analysis, as in special pertubations.

We will discuss several of these pertubations and their effect on the equations of motion. This survey is only meant as an introduction to the subject, and the reader is encouraged to pursue a more in-depth discussion to develop better pertubation models as is necessary.[34, 35]

The disturbances' effects on the dynamics is summed up by Cowell's Formulation:

$$\ddot{\vec{\mathbf{r}}} = -\frac{\mu}{r^3}\vec{\mathbf{r}} + \vec{a}_p \tag{4.72}$$

where $\vec{a}_p$ is the vector of accelerations due to pertubances.

This formulation for simulations is very convenient since the pertubations can be added linearly. Our discussion of how the calculation is carried out during simulation is discussed in Chapter 5.

4.4.1 Gravity Field of the Central Body

In Equation 4.47 we assumed a uniform gravity field created by a point mass. This assumption is not accurate, and can lead to gross errors in simulation by not accounting for variations in the gravity field due to a non-homogenous, or nonsperical central body about which the satellite is orbiting.

The derivation is not important to the implementation or use of the spacecraft simulation framework, but it is highly encouraged that the reader understand the underlying physics. Vallado[34] presents a good derivation, the results of which are shown here.

The components of the nonspherical acceleration ($\mathbf{a}_{nonspherical} = [a_i, a_j, a_k]^T$) in inertial coordinates (ECI) are:

$$a_i = \left[\frac{1}{r}\frac{\partial U}{\partial r} - \frac{r_k}{r^2\sqrt{r_i^2+r_j^2}}\frac{\partial U}{\partial \phi_{gc}}\right] r_i - \left[\frac{1}{r_i^2+r_j^2}\frac{\partial U}{\partial \lambda}\right] r_j \tag{4.73}$$

$$a_j = \left[\frac{1}{r}\frac{\partial U}{\partial r} - \frac{r_k}{r^2\sqrt{r_i^2+r_j^2}}\frac{\partial U}{\partial \phi_{gc}}\right] r_j + \left[\frac{1}{r_i^2+r_j^2}\frac{\partial U}{\partial \lambda}\right] r_i \tag{4.74}$$

$$a_k = \frac{1}{r}\frac{\partial U}{\partial r} r_k + \frac{\sqrt{r_i^2+r_j^2}}{r^2}\frac{\partial U}{\partial \phi_{gc}} \tag{4.75}$$

where the partial derivatives of the potential function are:

$$\frac{\partial U}{\partial r} = -\frac{\mu}{r^2}\sum_{\ell=2}^{\infty}\sum_{m=0}^{\ell}\left(\frac{R_\oplus}{r}\right)^{\ell}(\ell+1)P_{\ell m}\sin\phi_{gc}\left(C_{\ell m}\cos m\lambda + S_{\ell m}\sin m\lambda\right) \tag{4.76}$$

$$\begin{aligned}\frac{\partial U}{\partial \phi_{gc}} &= \frac{\mu}{r}\sum_{\ell=2}^{\infty}\sum_{m=0}^{\ell}\left(\frac{R_\oplus}{r}\right)^{\ell}\left(P_{\ell,m+1}\sin\phi_{gc} - m\tan(\phi_{gc})P_{\ell m}\sin\phi_{gc}\right)\\ &\quad\times\left(C_{\ell m}\cos m\lambda + S_{\ell m}\right)\end{aligned} \tag{4.77}$$

$$\frac{\partial U}{\partial \lambda} = \frac{\mu}{r}\sum_{\ell=2}^{\infty}\sum_{m=0}^{\ell}\left(\frac{R_\oplus}{r}\right)^{\ell} mP_{\ell m}\sin\phi_{gc}\left(S_{\ell m}\cos m\lambda - C_{\ell m}\sin m\lambda\right) \tag{4.78}$$

Calculation of these equations requires the use of the Legendre Functions ($P_{\ell m}$), which can be found in a table in Vallado [34, pg. 491], and the formulation of the gravitational coefficients ($C_{\ell m}$ and $S_{\ell m}$)[34].

The order of the summations is determined by the desired degree of accuracy of the model. However, increase accuracy costs more computation time. The user should be careful to choose an order of accuracy congruant with the modeling requirements. For most applications, $\mathcal{J}2$ pertubations ($\ell = 2$) are sufficient; however, more refined models may use $\mathcal{J}4$ or $\mathcal{J}6$ pertubations.

4.4.2 Atmospheric Drag

There is a pertubation force due to the drag of the spacecraft as it moves through the atmosphere of the central body it is orbiting. The disturbance magnitude varies according to the size and drag surface of the satellites, position, wind velocity and direction, atmospheric density, season, and numerous other factors that may be considered but will not be addressed here. The effect of drag is to remove energy from the spacecraft's orbit, thereby decreasing altitude.

The simplest model calculates the acceleration due to drag based on a computed drag coefficient, c_D, that takes into account the shape and surface of the spacecraft. Nominal parameters are 1.0 for a sphere, and 2 for normal satellites. The entire term $m/c_d A$ is usually referred to as the ballistic coefficient, BC. A high BC denotes a low drag effect, and vice versa. Another factor is the cross-sectional area of the spacecraft, and can vary based on the attitude of the satellite with respect to the velocity direction. The simple model is formulated as follows:

$$\vec{\mathbf{a}}_{drag} = -\frac{1}{2}\frac{c_D A_D}{m}\rho v_{rel}^2 \frac{\vec{\mathbf{v}}_{rel}}{|\vec{\mathbf{v}}_{rel}|} \tag{4.79}$$

$$\vec{\mathbf{v}}_{rel} = \frac{d\vec{\mathbf{r}}}{dt} - \vec{\omega}_\oplus \times \vec{\mathbf{r}} = \begin{bmatrix} \frac{dx}{dt} + \omega_\oplus y \\ \frac{dy}{dt} + \omega_\oplus x \\ \frac{dz}{dt} \end{bmatrix} \tag{4.80}$$

where ρ is the atmospheric density, m is the mass of the satellite, A_D is the projected cross-sectional area perpendicular to the velocity direction, and ω is the angular velocity of the central body. A more advanced model may include wind speed variations in the atmosphere (in inertial reference frame, ECI):

$$\begin{aligned} \vec{\mathbf{v}}_{rel} &= \vec{\mathbf{v}}_{ECI} - \vec{\omega} \times \vec{\mathbf{r}}_{ECI} + \vec{\mathbf{v}}_{wind} \\ &= \begin{bmatrix} \frac{dx}{dt} + \omega_\oplus y + v_w\left(\cos\alpha\sin\delta\cos\beta_w + \sin\alpha\sin\beta_w\right) \\ \frac{dy}{dt} + \omega_\oplus x + v_w\left(\sin\alpha\sin\delta\cos\beta_w + \cos\alpha\sin\beta_w\right) \\ \frac{dz}{dt} + v_w\left(\cos\delta\cos\beta_w\right) \end{bmatrix} \end{aligned} \tag{4.81}$$

where v_w is the wind's speed, β_w is the wind's azimuth, α is the satellite's right ascension, and δ is the declination as measured from Earth-Centered, Earth-Fixed (ECEF) coordinate frame.

These atmospheric models can become as complicated as necessary and are effective for a simulation considering the expected ballistic coefficient, altitude, and mission lifetime. Refer to Vallado [34], Bate Mueller & White [35] or Battin [36] for a more in-depth discussion of atmospheric models.

4.4.3 Solar-Radiation Pressure

Another non-conservative disturbance force, like atmospheric drag, is due to the fact that light photons can impart a force on an absorbing or reflecting body. The force of these photons is usually very small but can vary largely between eclipse and during solar storms. The solar-radiation pressure is even the basis for spacecraft propulsion designs such as solar sails.

The solar pressure, p_{SR}, is the main parameter in determining the force of the solar-radiation pressure. For Earth p_{SR} has a nominal value of $4.51 \times 10^{-6} \frac{N}{m^2}$, where more precise values can be calculated depending on the time of year and position from the Sun. The effect of solar-radiation pressure also varies due to the reflectivity, c_R, of the spacecraft, where 0.0 indicates no effect (translucent), 1.0 is a completely absorbing body, and 2.0 is an absorbing and reflecting body.

The combined force of the solar radiation pressure is found to be:

$$\vec{\mathbf{a}}_{radiation} = -\frac{p_{SR} c_R A_S}{m} \frac{\vec{\mathbf{r}}_{\odot sat}}{|\vec{\mathbf{r}}_{\odot sat}|} \tag{4.82}$$

where $\vec{\mathbf{r}}_{\odot sat}$ is the distance from the satellite to the sun (or light-emitting body), and A_S is the spacecraft's exposed area to the sun. This value of area is important for calculating the disturbance difference as the spacecraft passes from full sunlight, into eclipse, or when being shadowed by another body (moon or another spacecraft).

Using basic geometry, it can be shown that simple conditions for determining whether a satellite is in sunlight are [34]:

$$\begin{aligned} \tau_{min} &= \frac{|\vec{\mathbf{r}}_{sat}|^2 - \vec{\mathbf{r}}_{sat} \cdot \vec{\mathbf{r}}_{\odot}}{|\vec{\mathbf{r}}_{sat}|^2 + |\vec{\mathbf{r}}_{\oplus}|^2 - 2\vec{\mathbf{r}}_{sat} \cdot \vec{\mathbf{r}}_{\odot}} \\ &= \frac{\vec{\mathbf{r}}_{sat} \cdot \vec{\mathbf{r}}_{sat} - \vec{\mathbf{r}}_{\odot}}{|\vec{\mathbf{r}}_{sat} - \vec{\mathbf{r}}_{\odot}|^2} \\ \text{Sunlight if} \quad & \tau_{min} < 0 \text{ or } \tau_{min} > 1 \\ \text{or} \quad & |\vec{c}\,(\tau_{min})|^2 = (1 - \tau_{min})\,|\vec{\mathbf{r}}_{sat}|^2 + (\vec{\mathbf{r}}_{sat} \cdot \vec{\mathbf{r}}_{\odot})\,\tau_{min} \geq 1.0 \end{aligned} \tag{4.83}$$

where $\vec{\mathbf{r}}_{sat}$ is the radius vector from the earth to the the satellite, $\vec{\mathbf{r}}_{\odot}$ is the radius vector from the earth to the sun, and $\vec{\mathbf{r}}_{\oplus}$ is the radius vector from the earth's center to the horizon.

4.4.4 Third-Body Pertubations

In section 4.3.2 we discussed the generalized effect of multiple bodies on the dynamics of an orbiting spacecraft. The generalized form is acceptable when the seperations between the bodies are the same order of magnitude; however, as is the case for Earth-based and other central body based satellites, the distance to the primary central body is much less than the distance to any other disturbing bodies, such as the sun or the moon. Therefore, it is necessary to develop equations that rely on the small and sometimes inaccurate distance difference when talking about the distance from Earth to the sun and the satellite to the sun. The parameter B_k is the ratio of distance from the central body to the third body over the distance from the satellite to the third body:

$$B_k = \frac{r_{\oplus k}}{r_{satk}} - 1 \tag{4.84}$$

$$\begin{aligned} \ddot{\vec{\mathbf{r}}}_{\oplus sat} &= -\frac{Gm_{\oplus}}{r^3_{\oplus sat}}\vec{\mathbf{r}}_{\oplus sat} - \sum_{k=1}^{n} \frac{Gm_k}{r^3_{\oplus k}} \left(\vec{\mathbf{r}}_{\oplus k} - \beta_k \vec{\mathbf{r}}_{satk}\right) \qquad (4.85) \\ \beta_k &= 3B_k + 3B_k^2 + B_k^3 \text{ with } B_k = B(\zeta_k) \end{aligned}$$

where ζ_k is the angle between the third body and satellite as seen from Earth.

4.4.5 Other Pertubations

The described pertubations are just an introduction to the availability and accuracy of models that are available for increased simulation accuracy. Some other examples of disturbances are the force due to thrust, as well as the mass variation over time due to propellant loss, tides, higher resolution gravity models, or solar-radiation reflection from other bodies like the moon.

4.5 Propagation

Propagation is concerned with evaluating the position of the satellite through a series of timesteps using a specified orbital model and pertubations. Propagation is also the crux of simulation. There are two main methods of propagating a satellite, analytically, or numerically. Analytical propagation uses a set of equations to evaluate the discrete solution of a satellite's position at a given time. Numerical propagation evaluates the motion of the satellite over many small timesteps, integrating the solution, to find the final solution of the satellite's position after a given time span.

4.5.1 Analytic Propagation

f and g functions

One example of analytic propagation is the use of the f and g functions. These equations are derived from a preset condition of pertubations and models. The f and g functions provide a solution that linearly combine the intial position and velocity vectors to determine the new position and velocity vectors:

$$\vec{\mathbf{r}} = f\vec{\mathbf{r}}_0 + g\vec{\mathbf{v}}_0 \tag{4.86}$$
$$\vec{\mathbf{v}} = \dot{f}\vec{\mathbf{r}}_0 + \dot{g}\vec{\mathbf{v}}_0 \tag{4.87}$$

The f and g are transformations using the position and velocity components in the orbital plane to find the position at a future point in time. This method requires solving the system of differential equations that can be formulated from the above equations. The system of differential equations is:

$$f = \frac{x\dot{y}_0 - \dot{x}_0 y}{h} \tag{4.88}$$
$$\dot{f} = \frac{\dot{x}\dot{y}_0 - \dot{x}_0\dot{y}}{h} \tag{4.89}$$
$$g = \frac{x_0 y - x y_0}{h} \tag{4.90}$$
$$\dot{g} = \frac{x_0\dot{y} - \dot{x}y_0}{h} \tag{4.91}$$

where $h = ||\vec{h}||$ is the angular momentum of the orbit, $\vec{\mathbf{h}} = \vec{\mathbf{r}} \times \vec{\mathbf{v}}$, and $\vec{\mathbf{r}}$ and $\vec{\mathbf{v}}$ are in the orbital frame. It is also convenient to verify that the correct functions were derived by using the relation:

$$1 = f\dot{g} - \dot{f}g \tag{4.92}$$

The true anomaly, ν, is assumed to be known to demonstrate how to obtain the f and g functions. The components of the position vector must be determined in perifocal coordinates as well as the time derivatives:

$$x = r\cos\nu \tag{4.93}$$
$$y = r\sin\nu \tag{4.94}$$
$$\dot{x} = -\sqrt{\frac{\mu}{p}}\sin\nu \tag{4.95}$$
$$\dot{y} = \sqrt{\frac{\mu}{p}}(e+\cos\nu) \tag{4.96}$$

The components in perifocal coordinates can be used in equation 4.88 to determine the following functions:

$$f = 1-\frac{r}{p}(1-\cos\delta\nu) \tag{4.97}$$
$$g = \frac{rr_0\sin\delta\nu}{\sqrt{\mu p}} \tag{4.98}$$
$$\dot{f} = \sqrt{\frac{\mu}{p}}\tan\left(\frac{\delta\nu}{2}\right)\left(\frac{1-\cos\delta\nu}{p}-\frac{1}{r}-\frac{1}{r_0}\right) \tag{4.99}$$
$$\dot{g} = 1-\frac{r_0}{p}(1-\cos\delta\nu) \tag{4.100}$$

These equations can be derived for different element sets or for included pertubations and control thrust.

4.5.2 Numerical

Numerical propagation uses a numerical solution to the orbit model that is integrated over sufficiently small time-steps from epoch to the end of the desired simulation time. The state is defined as the position and velocity vectors. Therefore the time derivative of the state is defined as the following:

$$\dot{\mathbf{x}} = \begin{bmatrix}\dot{\mathbf{r}}\\ \dot{\mathbf{v}}\end{bmatrix} = \begin{bmatrix}\mathbf{v}\\ -\frac{\mu}{r^3}\mathbf{r}+\mathbf{a}_{disturbance}\end{bmatrix}. \tag{4.101}$$

The disturbance accelerations can be linearly summed:

$$\mathbf{a}_{distrubance} = \mathbf{a}_{nonspherical}+\mathbf{a}_{drag}+\mathbf{a}_{3-body}+\mathbf{a}_{solar-radiation} \tag{4.102}$$

Chapter 5 discusses how to numerically integrate this equation and propagate the satellite through time.

4.6 Summary

This chapter presented the main aspects of orbital dynamics and modeling. It has served to introduce the reader to the important topics, their formulation, and suggested to the reader more in-depth resources for understanding and analysis. The spacecraft simulation framework is built upon the principles presented here, but due to the nature of the design the framework allows users to implement more advanced and accurate models as they require with little to no modification of the underlying program structure.

Chapter 5

Numerical Methods

Modeling and simulation on a digital computer requires development of numerical methods that have known accuracies and can increase the speed of operation. This chapter presents the basic methods of digital representations of numeric values and the consequences to simulation. Following this, an introduction to numeric integrators and associated interpolators is given. Lastly, a discussion of relevant computation issues are covered, including rounding error, execution speed, and accuracy.

5.1 Integration

Integration is the calculation of the future state based on the current state and its derivatives. There are both single-step and multi-step integrators; both types of integrators can be either variable or fixed step-size. Single-step integrators use the state at time t_0 and the time derivatives to calculate the state at time future time $t_0 + h$, where h is the integration step-size. Examples of single-step integrators include Euler's Method and Runge-Kutta. Multi-step integrators attempt to predict initial conditions, solve forward through time, and correct backwards in time. Examples of multi-step integrators include Adams-Bashforth and Gauss-Jackson.

5.1.1 Euler

The simplest single-step integrator is Euler's Method, which uses a first-order Taylor series expansion to calculate the new state. The state is defined by calculating the time derivative of the state multiplied by a suitable time-step, added to the initial conditions:

$$y(t) \cong y(t_0) + \dot{y}(t_0)(t - t_0) \tag{5.1}$$

The state update is very simple to calculate and requires the time derivative of the state. The derivative is normally available (especially for dynamics equations), but is not necessarily accurate when calculated digitally. Analysis is required to determine an adequate time-step, $\delta t = t - t_0$, of the state update.

5.1.2 Runge-Kutta

A more advanced single-step integrator is the Runge-Kutta integrator. It operates by evaluating the time derivative of the equations at several different time-steps over the integrated interval and combines these derivatives to form a more accurate estimate of the new state. The fourth-order Runge-Kutta integrator is based on of a fourth-order Taylor series and is formulated as follows:

$$\begin{aligned}
\dot{y}_1 &= f\left(t_0, y_0\right) \\
\dot{y}_2 &= f\left(t_0 + \frac{h}{2}, y_0 + \frac{h}{2}\dot{y}_1\right) \\
\dot{y}_3 &= f\left(t_0 + \frac{h}{2}, y_0 + \frac{h}{2}\dot{y}_2\right) \\
\dot{y}_4 &= f\left(t_0 + h, y_0 + h\dot{y}_3\right) \\
y(t) &= y(t_0) + \frac{h}{6}\left(\dot{y}_1 + 2\dot{y}_2 + 2\dot{y}_3 + \dot{y}_4\right) + O\left(h^5\right)
\end{aligned} \tag{5.2}$$

where h is the integration step-size, and $O(h^5)$ is the truncation error due to high order terms.

If necessary higher-order Runge-Kutta integrators can be derived.[34] A general formulation for an ith-order integration and using j-iterations over the time interval is:

$$\begin{aligned}
& & y(t) &= y_0 + h\sum_{i=0}^{j-1} b_i \dot{y}_i + O\left(h^{n+1}\right) & & n = 0, 1, 2, ..., n-1 \\
\dot{y}_0 f(t_n, y_n) & & c_{i0} &= p_i \sum j = 1^{i-1} c_{ij} & & \\
& & \dot{y}_i &= f\left(t_n + p_i h, y_n + h\sum_{j=0}^{i-1} c_{ij}\dot{y}_j\right) & & i = 1, 2, ..., j-1
\end{aligned} \tag{5.3}$$

where b_i, c_{ij}, and p_i are user-defined parameters. Refer to Fehlberg[37] and Der[38] for discussion of the calculation of these parameters.

5.1.3 Runge-Kutta-Fehlberg

A different implementation of the Runge-Kutta integration is the Runge-Kutta-Fehlberg, or embedded Runge Kutta method. This method uses a variable single step-size that is determined based on the error of the calculation. The step-size between the integration time-steps is then varied to achieve the specified tolerance.

$$\begin{aligned}
k_1 &= hf(t_0, y_0) \\
k_2 &= hf\left(t_0 + \frac{1}{4}h, y_0 + \frac{1}{4}k_1\right) \\
k_3 &= hf\left(t_0 + \frac{3}{8}h, y_0 + \frac{3}{32}k_1 + \frac{9}{32}k_2\right) \\
k_4 &= hf\left(t_0 + \frac{12}{13}h, y_0 + \frac{1932}{2197}k_1 - \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3\right) \\
k_5 &= hf\left(t_0 + h, y_0 + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4\right) \\
k_6 &= hf\left(t_0 + \frac{1}{2}h, y_0 - \frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5\right)
\end{aligned} \tag{5.4}$$

The error must be evaluated to determine if it is within the specified tolerance by calculating a measure of the error, s, and determining if it is possible to optimize calculation by increasing or decreasing the step-size:

$$error = \frac{1}{360}k_1 - \frac{128}{4275}k_2 - \frac{2197}{75240}k_4 + \frac{1}{50}k_5 + \frac{2}{55}k_6 \tag{5.5}$$

$$s \cong 0.8408 \left[\frac{1 \times 10^{-8}h}{error}\right]^{1/4} \tag{5.6}$$

The conditions for varying the step-size h are:

$$\text{if } s < 0.75 \text{ and } h > 2h_{min} \text{ then } h = h/2$$
$$\text{if } s > 1.5 \text{ and } 2h < 2h_{max} \text{ then } h = 2h$$

This decision method is dependent on user-specified variables for h_{min} and h_{max}. Bounds on the step-size are typically based on the initial step-size: $h_{min} = h/64$ and $h_{max} = 64h$.

If the step-size is varied, the time step must be recalculated and h varied until the error tolerance is met:

$$\text{else } y = y_0 + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5$$

5.1.4 Adams-Bashforth

As mentioned previously, there are both single-step and multi-step integration techniques. Adams-Bashforth is a multi-step integration technique. A good discussion of the differences can be found in Burden, et al. [39].

For the multi-step integrator, several of the initial fordward states must be determined, most likely using a single-step integrator, such as Runge-Kutta, of the same order as the multi-step integrator. The multi-step integrator then takes over and calculates explicit methods for determining the forward state. Implicit methods are then used to correct these values. This combination of explicit and implicit methods is known as the predictor-corrector method.

The following is a method of integration based on the fourth-order Adams-Bashforth method as predictor and one iteration of the Adams-Moulton method as corrector, with the starting values obtained from the fourth-order Runge-Kutta method [39].

Given the initial conditions, calculate the first four time-steps using the Runge-Kutta 4th order integrator. Then, using a predictor and corrector as shown below, the future states can be integrated:

$$\begin{aligned}
&\text{predictor:} \\
\mathbf{x}_{i+1} &= \mathbf{x}_i + \frac{h}{24}\left[55f(t_i,\mathbf{x}_i) - 59f(t_{i-1},\mathbf{x}_{i-1}) + 37f(t_{i-2},\mathbf{x}_{i-2}) - 9f(t_{i-3},\mathbf{x}_{i-3})\right] \\
&\text{corrector:} \\
\mathbf{x}_{i+1} &= \mathbf{x}_i + \frac{h}{24}\left[9f(t_i,\mathbf{x}_i) + 19f(t_{i-1},\mathbf{x}_{i-1}) - 5f(t_{i-2},\mathbf{x}_{i-2}) + f(t_{i-3},\mathbf{x}_{i-3})\right]
\end{aligned}$$

where $\dot{\mathbf{x}} = f(t,\mathbf{x})$. First the predictor is calculated. The result is then augmented by the corrector equation.

5.2 Interpolation

A user simulating dynamic equations may request the state at non-mesh points, or points that were not specifically calculated in the integration algorithm. Such a calculation requires the use of interpolation to find states in between the mesh points. For example, a set of dynamic equations were integrated from 0 to 100 seconds with 5 second intervals. However, it may be necessary to know the state at 56 seconds. An interpolation between the 55 and 60 second mesh points is required to determine an approximate value for the state at 56 seconds.

5.2.1 Lagrange Interpolation

The simplest method of interpolation is the "sample and hold," where the state is held at each mesh point until the next update. For example, using such a simple interpolation would cause the state at 55 seconds to be the same for all times until the new state at 60 seconds. Therefore, sample and hold is typically a poor estimate. The next best step would be to use a linear interpolation:

$$P(t) = \frac{t - t_1}{t_0 - t_1} x_0 + \frac{t - t_0}{t_1 - t_0} x_1 \tag{5.7}$$

where the points (t_0, x_0) and (t_1, x_1) are the mesh points and the function $P(t)$ is the linear interpolation function between the mesh points.

For variable order, the interpolation equation can be generalized to:

$$P(t) = \sum_{k=0}^{n} x_k L_{n,k}(t) \tag{5.8}$$

$$L_{n,k}(t) = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{t - t_i}{t_k - t_i} \tag{5.9}$$

where $L_{n,k}$ is the nth Lagrange interpolating polynomial [39].

5.2.2 Cubic Spline Interpolation

The Lagrange interpolating polynomial is based upon a continuous, evaluated polynomial from all of the mesh points. Another alternative is to derive a piecewise-polynomial approximation using a collection of subintervals. Dividing the polynomial into subsections has the benefit of permitting a choice of the set of mesh points to include in the interpolation, as well as setting the boundary conditions at the end meshpoints.

A cubic spline interpolation uses cubic polynomials between the nodes to attempt to model a smooth and continuous interpolant. The algorithm is derived in Burden, et al.

[39] and shown here:

$$\begin{aligned} \text{for } i=0,1,...,n\text{-}1 \text{ set} \quad & h_i = x_{i+1} - x_i \\ \text{for } i=1,2,...,n\text{-}1 \text{ set} \quad & \alpha_i = \frac{3}{h_i}(x_{i+1} - x_i) - \frac{3}{h_{i-1}}(x_i - x_{i-1}) \end{aligned} \tag{5.10}$$

$$\tag{5.11}$$

$$\begin{aligned} \text{set} \quad & l_0 = 1 \quad \mu_0 = 0 \quad z_0 = 0 \\ \text{for } i=1,2,...,n\text{-}1 \text{ set} \quad & l_i = 2(t_{i+1} - t_{i-1}) - (h_{i-1})(\mu_{i-1}) \\ & \mu_i = \frac{h_i}{l_i} \qquad i = 1, 2, ...n-1 \\ & z_i = \frac{\alpha_i - (h_{i-1})(z_{i-1})}{l_i} \end{aligned} \tag{5.12}$$

$$\begin{aligned} \text{set} \quad & l_n = 1 \quad \mu_n = 0 \quad z_n = 0 \\ \text{for } j=\text{n-1,n-2,...,0 set} \quad & \\ & c_j = z_j - \mu_j c_{j+1} \\ & b_j = \frac{x_{j+1} - x_j}{h_j} - \frac{h_j}{3}(c_{j+1} + 2c_j) \\ & d_j = \frac{c_{j+1} - c_j}{3h_j} \end{aligned} \tag{5.13}$$

The result is a set of coefficients that can be used to calculate the interpolation spline at any point t in the spline interval j:

$$S_j(t) \quad = \quad x_j + b_j(t - t_j) + c_j(t - t_j)^2 + d_j(t - t_j)^3 \tag{5.14}$$

There is also a similar derivation of a cubic spline interpolation with clamped endpoints shown in Burden, et al. [39, pg. 148].

5.3 Computation

Analog computation can be exact, such as calculating $\sqrt{2}$, or $\sin 4\pi$. However, when these computations are carried out on a computer, they must be represented digitally and are therefore subject to errors due to the finite nature of their digital representation (from before the analog computations can be calculated using some appropriate method such as a series exansion to find approximations: $\sqrt{2} = 1.414213...$ and $\sin 4\pi = 0.054803665...$). Furthermore there needs to be consideration for the computational effort and speed required to perform the large number of operations in a simulation. This section discusses these issues, their effects on simulation, and ways to minimize these effects.

5.3.1 Rounding Error

Rounding error is the effect of a digital representation of a real number. Most computers have a common implementation of the IEEE *Binary Floating Point Arithmetic Standard 754-1985* which uses a 64-bit (binary digit) representation for the real numbers. This *long real* allows at least 16 decimal digits of precision [39].

The term *roundoff error* is the error that results from replacing an exact number with its floating-point form. The error can be measured using *absolute error*: $|p - p^*|$, or *relative error*: $\frac{|p-p^*|}{|p|}$, if $p \neq 0$, and where p^* is the approximation of p.

Whereas the floating-point digital representation of a number reduces the accuracy of an analog value, arithmetic operations on these numbers introduce different errors. For example, subtracting nearly equal numbers leads to the cancellation of significant digits and therefore introduction of error. Dividing by a number of small magnitude, or multiplying a number of large magnitude also enlarges the error. Therefore, it is sometimes necessary to rearrange algorithms to minimize these arithmetic errors.

Another useful tip in numerical calculation is nesting of polynomials. Calculating a polynomial in its nested form reduces the number of arithmetic operations and can greatly decrease the error [39].

In the following example:

$$\begin{aligned} f(y) &= x^2 - 3x + 2 \\ \Rightarrow f(y) &= x(x-3) + 2 \end{aligned}$$

the first equation has a total of three multiplications and two additions (where subtractions are additions that are merely sign compliments), whereas the second equation has 2 additions and 1 multiplication. As discussed below, nesting not only decreases error, but it can also increase operational speed.

5.3.2 Execution Speed

The optimization of code to increase execution speed is challenging subject. Depending on the programming language, there are numerous methods for decreasing program running times such as the following:

1. Ordering arrays and vectors according to the layout of the array in memory to promote sequential accessing;

2. Using pointers to memory rather than producing expensive copies of data for simple function calls;
3. Using predictive operation of the processor by knowing that *if* statements are assumed to be true;
4. Using low-level, or machine code for small, quick functions;
5. Allowing compilers to optimize code, and possibly using a higher order optimization scheme; and
6. Postponing calculation until necessary rather than merely at a function call;

While these tricks are useful for increasing running speed, they can also induce errors if not used correctly. Furthermore, optimizations can be computer architecture dependent, thereby stifling cross-platform compatability. Programming specific optimizations also often produces illegible code, making maintenance and reuse difficult or impossible. The goal of the user and programmer should be to first produce correctly functioning code, then seek to optimize. Advanced optimization may require the developer to also do research into the appropriate language, architecture, compiler and operating system. Good resources include Meyers [40, 41], Stroustrup [28], Sutter [42, 43].

5.4 Summary

This chapter presented some numerical computation concepts that are important to the implementation of a dynamic simulation application. The included examples of integrators and interpolators form the mathematic library of Open-SESSAME, but they may also be used for other software integration. This chapter also included some discussion of digital errors that are introduced when modeling analog systems. The implementations of the mathematical components, and previously discussed attitude and orbit algorithms are presented in the next chapter.

Chapter 6

Software Design

In this chapter the development, layout, and implementation of the Open-SESSAME Framework is presented. An overview of the object-oriented design is covered followed by an in-depth discussion of each of the libraries and toolkits. Finally, a demonstration of the use of Open-SESSAME is given.

To begin, a definition of the Open-SESSAME acronym is given:

1. *Open-Source* - the source is free and available for users and developers. Changes are propagated back to the community via the publicly hosted repository so the software continues to grow and mature.

2. *Extensible* - the framework is designed with the premise that functionality will be added as necessary by new users and developers. The code is plainly written and well-documented to ease understanding of the source, and also encourages development via hotspots, points which assist in adding functionality (*i.e.* environmental disturbance functions, dynamic equations, kinematic representations, *etc.*).

3. *Spacecraft* - Open-SESSAME is developed with the target of simulating spacecraft and satellites in outer space. While there are generic mathematical and operational toolboxes (matrix, rotation, XML storage) as part of the software package, these libraries were developed or interfaced with the given goal in mind.

4. *Simulation And Modeling* - Simulation is providing a user with a non-real, but approximated environment that accurately corresponds to the real-world. Modeling is the creation of the physical dynamics and characteristics of this simulated world. Open-SESSAME is meant to be used as both a stand-alone model of spacecraft,

and for use in creating simulations that interact with hardware and other software programs.

5. *Environment* - the environment is the entire collection of dynamics, disturbances, data handling operations and interfaces that allow the user to interact with the simulation.

6. *Framework* - Open-SESSAME is a framework. It provides the tools and libraries that are combined together to create a simulation environment. By themselves they do not constitute an application, but must be joined by the user/developer in a meaningful way to simulate and analyze their particular problem.

These terms form the basis of design and requirements for Open-SESSAME. The architecture presented below should fulfill these requirements and provide the user with an easy to use and powerful tool for modeling spacecraft.

6.1 Framework Layout

The framework consists of a collection of libraries and toolkits that can be used in conjunction to build a spacecraft simulation application of varying complexity, from simple attitude kinematics analysis, to full orbit and attitude integrated propagation of a satellite with multi-body dynamics and control mechanisms. A goal of the system is to provide for this functionality in a modularized design such that specific components can be inserted, replaced or taken out easily and accurately as desired by the user.

The following sections outline all of the components that make up the Open-SESSAME framework. Their purpose and general use is defined, as well as typical operation of the implemented aspects of the libraries. Furthermore, suggested extension points are pointed out for future users and maintainers of the system that may require added functionality. A diagram of the entire framework is shown in Figure 6.1. The specific toolkits include UML (Unified Modeling Language) diagrams of their architecture and some interfaces to the primary classes. For more information, refer to the Open-SESSAME User's & Maintainer's Guide [44], or the Sourceforge Repository [45].

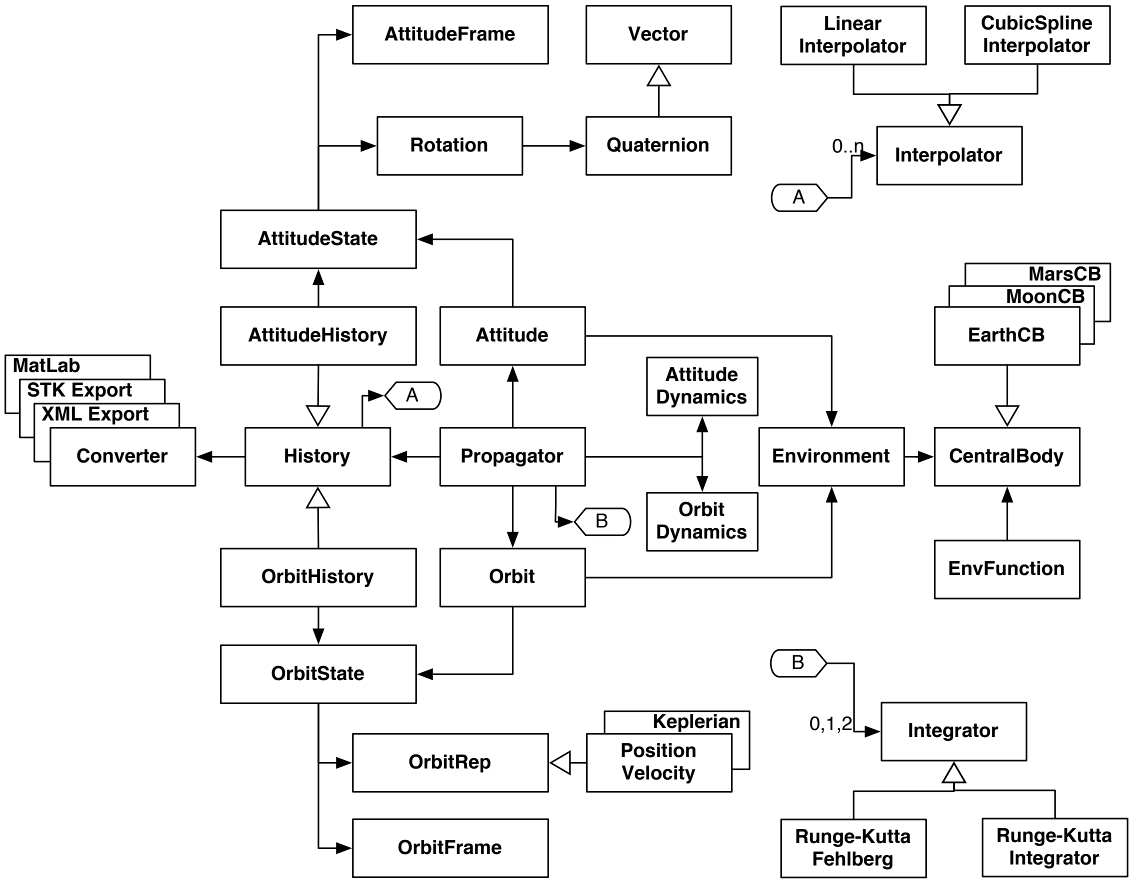

Figure 6.1: UML diagram of spacecraft simulation application software components. This diagram illustrates how an entire spacecraft simulation is built using Open-SESSAME framework components. Open arrowheads denote a derived, or "is-a" relationship, where the module from the base of the arrow inherits and extends the functionality of the module at the end of the arrow (*e.g.* PositionVelocity is-a OrbitStateRepresentation). Solid arrowheads are a composition, or "has-a" relationship(*e.g.* AttitudeState has-a Rotation). Blocks that are stacked on top of one another can be interchanged as necessary (*e.g.* replacing Position/Velocity orbital components with Keplerian elements).

6.1.1 Rotation Library

Description

The *Rotation Library* is a collection of kinematic representations and operations used to represent coordinate transformations. Attitude orientations or orbit relative reference frames require a transformation to describe their axes relative to a specified reference frame. For instance, a rotation describes the transformation required to determine the orbit reference frame from the Earth-Centered Inertial (ECI) reference frame: $\mathbf{R}^{oi}$. Refer to Section 3.2 for an in-depth discussion of attitude kinematics.

The current representations are: *Quaternion*, *Modified Rodriguez Parameters*, and *Direction Cosine Matrix*. The functionality is included , but not specifically implemented in a seperate class, for *Euler Angles* and *Euler Axis & Angle*.

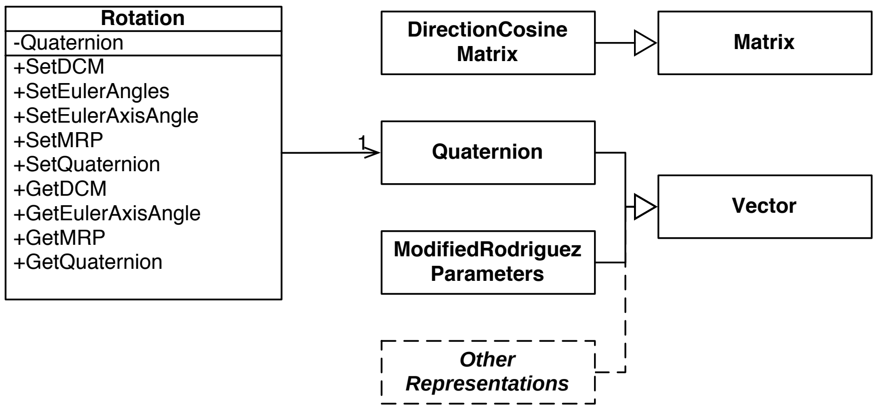

Figure 6.2: Rotation Library UML diagram. Rotation is an abstract class that provides an interface functionality for any type of rotation. Internally Rotation stores a Quaternion, which is derived from, and has the functionality of, a Vector. There are also classes for ModifiedRodriguezParameters and DirectionCosineMatrix. Future representations could include Gibbs vector or an EulerAngle class.

Extension Point: there are other kinematic representations that could be implemented as necessary, *Gibb's Vector* or *Euler-Rodriguez Parameters*. These classes would be derived from the *Vector* class and include the conversion algorithms to and from each of the other existing representations. The user could extend the existing representations to convert to the new representations, though it is suggested to try and minimize the alteration of the existing classes.

The main class of the *Rotation Library* is the *Rotation* class. The *Rotation* class is an abstract representation (different from a Abstract Data Type, ADT) in that it is not a specific *type* of representation but rather a general rotation concept between reference frames. For this reason, the *Rotation* class may be set, and can be output in, any of the kinematic representation types. The class can also be used to determine successive and relative rotations.

Implementation

Each of the major kinematic representations is encapsulated in a class, all of which are derived from the *Vector* or *Matrix* classes, and therefore, the representations include all the functionality of these classes while extending more functionality to include appropriate conversion and transformation operations. The added funtionality includes getting and setting the representation from any other representation as well as determining successive and relative rotations between like kinematic types.

The *Rotation* class has a *Quaternion* as a private data member, which is not directly accessible from outside of the class. The user accesses and manipulates the data using the provided public member functions. The quaternion representation was chosen since it does not exhibit difficulties due to singularities like other representations and also only contains four elements, saving a small amount of data over larger representations, like a Direction Cosine Matrix.

Usage

The following code is an example of using the *Rotation Library* to create various kinematic rotations and output these values to console.

```
// create a DCM with successive rotations of [30,-10,5] degs
// in a 123 rotation order
DirectionCosineMatrix dcm1(deg2rad(30),deg2rad(-10),deg2rad(5), 123);
// create a quaternion that is the same attitude transformation as dcm1
Quaternion q1(dcm1);
// create a second quaternion from the transpose of dcm1 (~dcm1)
Quaternion q2(~dcm1);
// create a rotation from the successive rotation of q1 and q2
Rotation rot1(q1 * q2);
// output rot1 to the standard stream (usually the screen)
```

```
cout << rot1;

Vector eulerAxis;
double eulerAngle;
// convert rot1 to the Euler Axis and Angle
//   returned by reference
rot1.GetEulerAxisAngle(eulerAxis, eulerAngle);
// output the axis and angle to the console
cout << eulerAxis << eulerAngle;
```

6.1.2 Attitude Toolkit

Description

The *Attitude Toolkit* is the collection of classes and tools that provide for analyzing, modeling, and simulating the attitude of a spacecraft. This collection includes state representations, kinematic and dynamic equations of motion, and the general spaceraft attitude as discussed in Chapter 3.

Implementation

The *AttitudeState* class represents the actual attitude measurement of a spacecraft at an instant in time. It contains a reference to both the appropriate rotation and relative reference frame of the rotation. Therefore, the *AttitudeState* class encapsulates this combined data into a single, succinct class with methods.

The kinematic and dynamic equations of motion are the physical algorithms that describe the motion of the spacecraft due to torques, disturbances, and any other desired modeling characteristic.

$$\dot{\mathbf{x}} = f(t, \mathbf{x}, pOrbit, pAttitude, Parameters, DisturbanceFunction) \tag{6.1}$$

These functions follow a generalized prototype, **odeFunctor**, which allows any kinematic and kinetic equation to be used, as long as it follows this function prototype. This prototype is shown in Equation 6.1. The state, $\mathbf{x}$, is the vector of states values at time t. The *pOrbit* and *pAttitude* inputs are instances of the current **Orbit** and **Attitude** classes at the evaluation time, *Parameters* are any constants, and *DisturbanceFunction* is a reference to the disturbance function (such as torque disturbances). Currently there are kinematic and dynamic equations that make use of the quaternion kinematic representation and angular velocities.

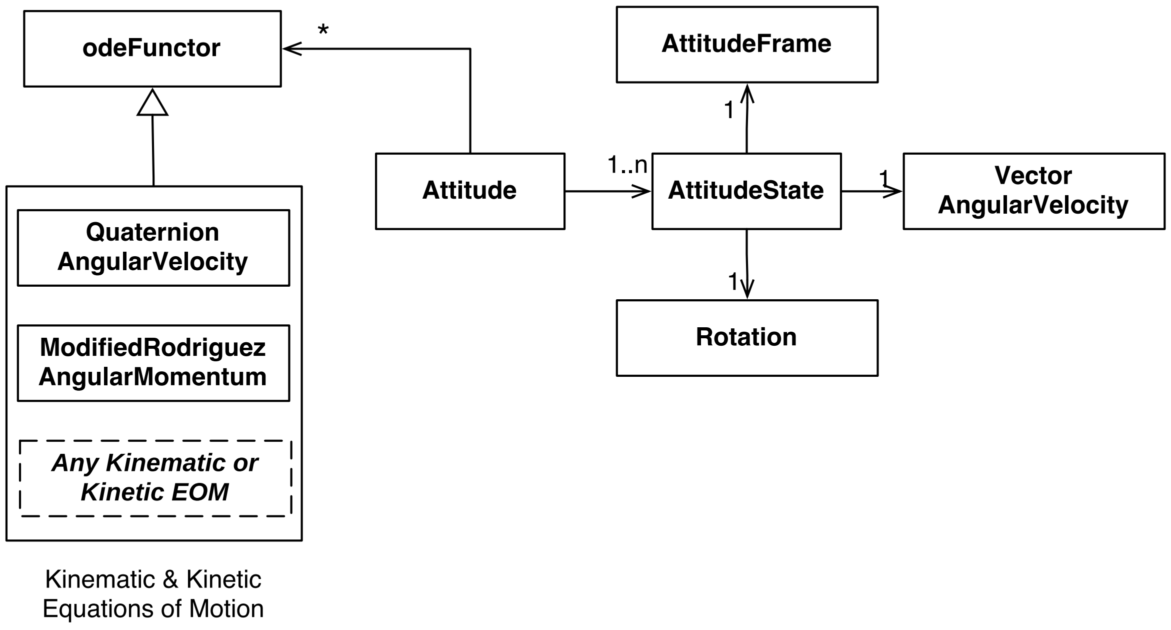
odeFunctor
*
AttitudeFrame
1
Attitude
1..n
AttitudeState
1
Vector AngularVelocity
1
Rotation
Quaternion AngularVelocity
ModifiedRodriguez AngularMomentum
Any Kinematic or Kinetic EOM
Kinematic & Kinetic Equations of Motion

Figure 6.3: Attitude toolkit UML diagram. The Attitude class stores a history of one or many AttitudeStates. Each AttitudeState stores a frame, Rotation, and Angular Velocity. Attitude also has a function pointer (denoted by the * on the line connecting Attitude to odeFunctor) to a dynamics equation.

Extension Point: It is apparent that there are other kinematic representations that may be used in modeling the attitude dynamics. Furthermore, the engineer may desire to integrate and model any number of attitude related characteristics such as momentum wheels, or kinetic energy. These algorithms can be implemented and verified by the user as necessary using the existing algorithms as a model.

Another necessary part of the *Attitude Toolkit* is the collection of attitude state conversion functions. The standard state output vector, such as may be returned from an integration timestep, consists of the simulation time and state components. The state conversion function translates this vector to an **AttitudeState** object. These functions are required since the simulator does not know what the state vector components are but requires knowledge of the attitude state from the integrated dynamics.

All of the attitude information is contained in the **Attitude** object. This general class contains the current attitude state, a history of attitude states (see Section 6.1.7), and a reference to the equation of motion to use for integration.

Usage

The following code fragment creates an attitude state and sets the rotation and angular velocity vector.

```
// Create the initial attitude state
AttitudeState myAttitudeState;
myAttitudeState.SetRotation(Rotation(Quaternion(0,0,0,1)));
Vector initAngVelVector(3); // elements initially all [0,0,0]
    initAngVelVector(1) = 0.1;
myAttitudeState.SetAngularVelocity(initAngVelVector);
```

6.1.3 Orbit Toolkit

Description

The *Orbit Toolkit* includes all the functionality to represent and simulate a spacecraft orbit. This toolkit includes, similar to the *Attitude Toolkit*, state representations, kinematic and dynamic equations of motion, and the general spacraft orbit as discussed in Chapter 4.

Implementation

The orbit state is representated by a conglomeration of classes. **OrbitStateRepresentation** is an abstract interface definition for storing and converting the state parameters of an orbit. The two primary representations are **Keplerian** and **PositionVelocity**. Each class stores a vector of its respective parameters and provides conversion functions for changing between the parameter types. Future representations could include canonical units, Delauney, or Poincaré variables.

State representations are accompanied by **OrbitFrame**, which stores the information regarding the frame from which the **OrbitStateRepresentation** is measured. Example frames could be Earth-Centered Inertial, Moon-Centered Moon-Fixed, or other pertinent representations. By associating a frame with a state representation, consistency is promoted to prevent comparing, for example, position vectors in ECI versus ECEF frames. Therefore, the **OrbitState** class contains both an **OrbitStateRepresentation** and an **OrbitFrame**. This concise class ensures that orbit state information that is passed through functions has a representation in a specified frame.

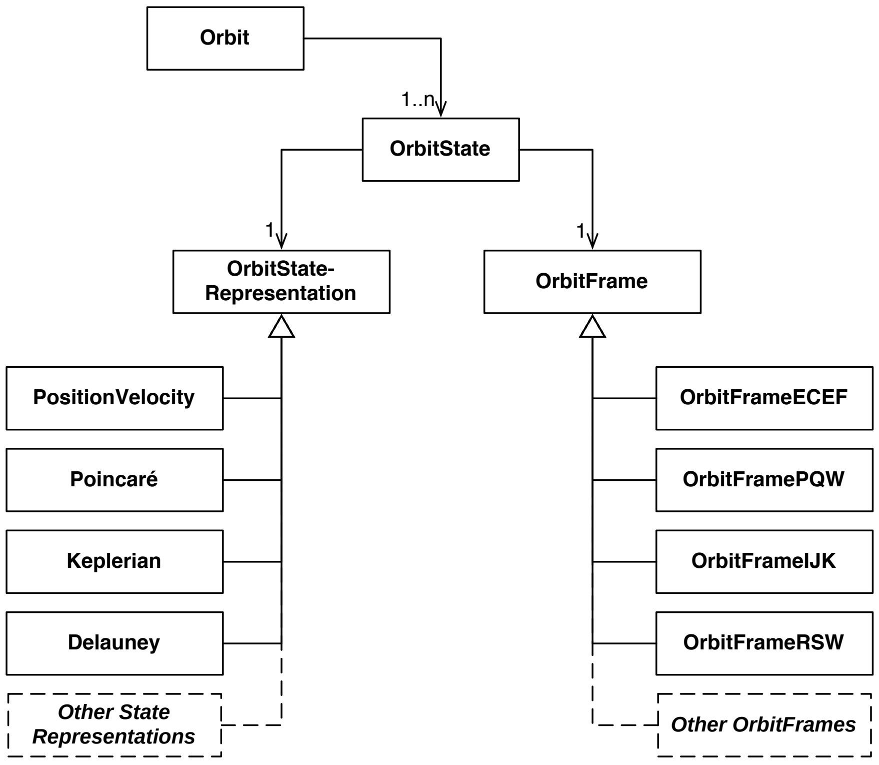

Figure 6.4: Orbit toolkit UML diagram. The Orbit class stores the current OrbitState as well as a history of the states. Each OrbitState stores a representation and the reference frame. Orbit also has a dynamics equation pointer which is not shown in the diagram.

The **Orbit** class, much like the **Attitude** class, is a general encapsulation of the current orbit state, orbit history, associated environment, and reference to the current propagator and dynamic equations.

Usage

The following code example creates an orbit state with a position and velocity representation in the inertial frame.

```
// create an instance of OrbitState
OrbitState myOrbit;
```

```
// specify the state rep to be position, velocity
myOrbit.SetStateRepresentation(new PositionVelocity);
// set the refererence frame to earth inertial
myOrbit.SetOrbitFrame(new OrbitFrameIJK);

// specify the individual components of the vector
Vector initPV(6);
    initPV(1) = -5776.6; // km
    initPV(2) = -157; // km
    initPV(3) = 3496.9; // km
    initPV(4) = -2.595;  // km/s
    initPV(5) = -5.651;  // km/s
    initPV(6) = -4.513; // km/s
// store the vector in the OrbitState object
myOrbit.GetStateRepresentation()->SetPositionVelocity(initPV);
```

6.1.4 Environment

Description

The *Environment Toolkit* is the collection of central bodies, external force and torque disturbance functions, and method of calculating the effect of the environment on a spacecraft. The primary class, **Environment**, encapsulates all of the environment data that is usually referenced by the **Orbit** and **Attitude** objects.

Implementation

The user can create an instance of a **CentralBody**, a representation of the Earth, Moon, or whichever celestial body about which the spacecraft is situated. This **CentralBody** object contains information regarding the radius, atmosphere, angular velocity, mass, and any other data that is pertinent to spacecraft modeling. The **EarthCentralBody** and other planets and moons are derived from the **CentralBody** class and, therefore, have the general functionality of the **CentralBody**.

The **Environment** object contains a reference to the central body and a list of the applied disturbance functions. These disturbance functions have a generalized but specified interface (time, orbit state, and attitude state) that is used to calculate the specific torque or force on the spacecraft during the integration of the dynamics. The user creates these functions, assigns constants that may be used (spacecraft mass, altitude, reflectivity), and stores them in the *Environment* instance that is used for the spacecraft. In each integration step, the attitude or orbit dynamics may call this function, at the instantaneous state, to obtain the torques and forces upon the spacecraft.

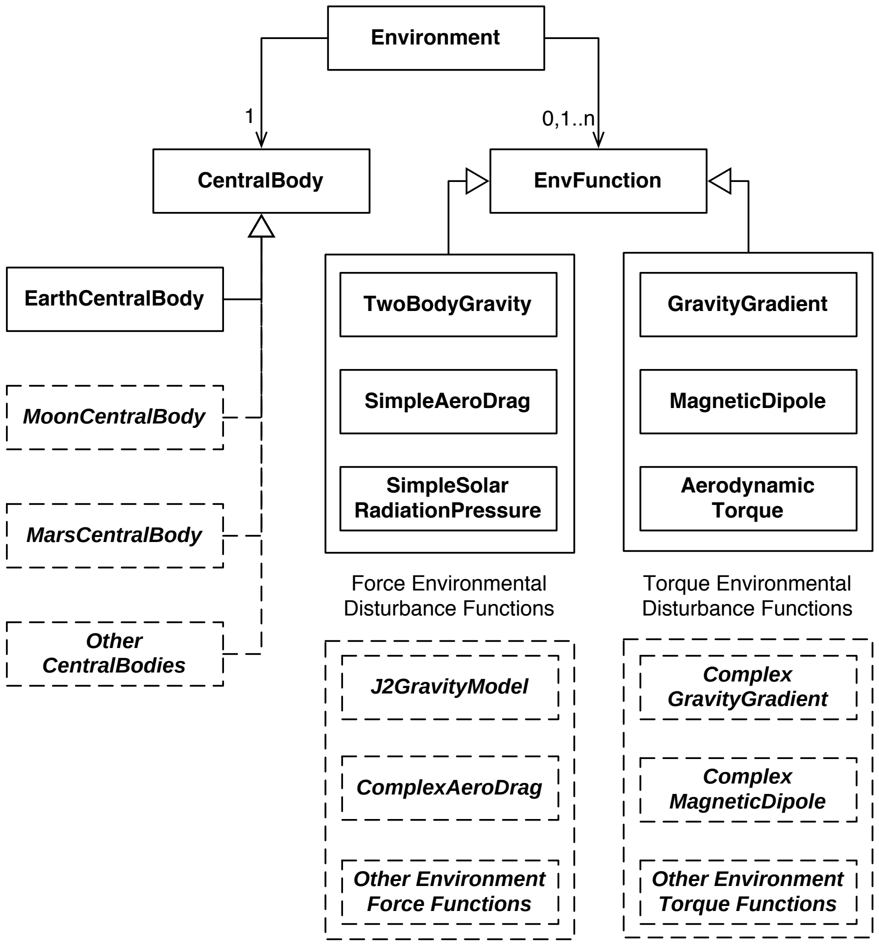

Figure 6.5: Environment toolkit UML diagram. Environment consists of a CentralBody and any number of force or torque EnvFunctions that contribute a disturbance during propagation of an orbit or attitude.

Usage

The following large section of code does the following tasks: creates an environment (allocate memory), creates and assigns the Earth central body, adds the two body force function to the environment, and adds a drag force function that requires a ballistic coefficient and air density variable.

```
// create a new environment and allocate memory
Environment* pEarthEnv = new Environment;
// create and allocate memory for the earth representation
EarthCentralBody *pCBEarth = new EarthCentralBody;
// set the earth CB to the environment object
pEarthEnv->SetCentralBody(pCBEarth);

// add Gravity force function
cout << "Filling Parameters" << endl;
EnvFunction TwoBodyGravity(&GravityForceFunction);
pEarthEnv->AddForceFunction(TwoBodyGravity);

// add Drag Force Function
EnvFunction DragForce(&DragForceFunction);

// setup the ballistic coefficient parameter
double *BC = new double(200);
DragForce.AddParameter(reinterpret_cast<void*>(BC), 1);
// setup the atmospheric density parameter
double *rho = new double(1.13 * pow(10., -12.)); // kg/m^3
DragForce.AddParameter(reinterpret_cast<void*>(rho), 2);

// add the force function to the environment
pEarthEnv->AddForceFunction(DragForce);
```

The reinterpret_cast ¡void*¿ code is necessary to maintain the generality of the input variables. Using this casting allows the user to pass in parameters of any type without changing the environment's force and torque calculating functions.

6.1.5 Integrator

Description

The **Integrator** library is part of the math library, but requires some specific explanation. Integration in Open-SESSAME is modeled after integration use in MatLab. The user must specify a dynamic equation of the following form to be integrated :

$$\dot{\mathbf{x}} = f(t, \mathbf{x}) \tag{6.2}$$

where $\mathbf{x}$ is the state vector, and $\dot{\mathbf{x}}$ is the time derivative of the state vector. The function $f(t, \mathbf{x})$ is referred to as the "Right-Hand Side," or RHS. The state vector can be any components the user may wish to integrate, whether it is a quaternion and angular velocities, or position, kinetic energy, and momentum. This dynamic equation is then integrated using an implemented integrator, which is described in Section 5.1, such as **RungeKuttaIntegrator** or **AdamsBashforthIntegrator**.

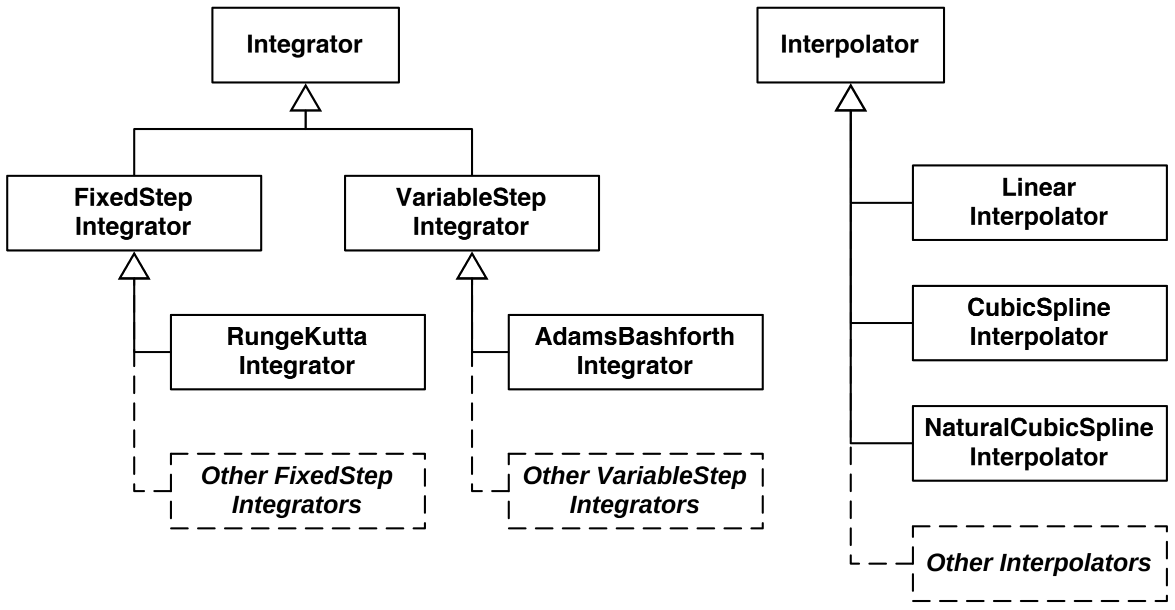

Figure 6.6: Math toolkit UML diagram. The Integrator and Interpolator classes provide consistent interfaces as base classes. The derived classes (open arrowheads) implement the particular algorithm.

Implementation

As mentioned above there are several integrators available, all of which are derived from **Integrator**. This superclass defines that the integrators must all include an *Integrate()* function that takes the current time, integrating state vector, initial conditions, reference to an orbit and attitude (if required), constants for calculation, and a reference to an external force function.

The reference to an orbit or attitude is used only when some coupling is required. These references are passed directly to the dynamics equation (RHS), and may be queried for environmental information, state parameters, or other constants. If no coupling occurs, or the information is not needed, the user is not required to send these references. The constants for calculation are a matrix of any other parameters needed for the dynamics equation, and are held constant during the integration. Finally, the external force function is a **Functor**, a type of call-back function that the user may specify for evaluating the force or torque disturbance function. A **Functor** permits the user to use the member function of a class, such as the **Environment** *GetForces()* function.

The output of the integration is a vector of times and states at each of these times. The step-size between the state outputs is dependent on the integration method employed as well as the specific parameters set by the user in the case of a multi-step or variable-step integration scheme.

Usage

The code example creates a Runge-Kutta 4th order integrator, sets the integration time to 20 seconds, and then integrates an AttitudeDynamics equation with no orbit or attitude coupling, and also passes in a matrix of the moments of inertia (Parameters) and the disturbance force function (AttitudeForcesFunctor):

```
// setup an integrator and any special parameters
RungeKuttaIntegrator myIntegrator;
myIntegrator.SetNumSteps(1000);
// integration times
vector<ssfTime> integrationTimes;
ssfTime begin(0);
ssfTime end(begin + 20);
integrationTimes.push_back(begin);
integrationTimes.push_back(end);

// call the integrator
    Matrix history = myIntegrator.Integrate(
                        integrationTimes, // seconds
                        &AttituteDynamics,
                        myAttitudeState.GetState(),
                        NULL,
                        NULL,
                        Parameters,
                        AttitudeForcesFunctor
                    );
```

6.1.6 Propagator

Description

The *Propagator Toolkit* provides the functionality necessary to simplify the entire simulation process by encapsulating the simulatenous operation of many of the other toolkits, such as *Integration*, *Orbit*, *Attitude*, and *Environment*. Furthermore, it also is useful for coupling orbit and attitude dynamics. There are several existing schemes, as well as extension points for any new algorithms, for varying degrees of coupling. Currently, there are independent, weak (attitude dependent on orbit, or orbit dependent on attiude), strong (orbit and attitude interdependent), and joined (fully coupled dynamic equations) propagation schemes. A **Propagator** can also be used when the attitude or orbit is available from an external source (*e.g.* file, hardware, other software package).

Implementation

The **Propagator** class provides a defined interface to the library of propagators. The two derived classes, **NumericPropagator** and **AnalyticPropagator**, each implement the respective method of propagation. Specifically, a **NumericPropagator** requires an **Orbit** and/or **Attitude** class with dynamic equations or populated history. When the user assigns an orbit with a dynamics equation, the propagator will integrate the orbit, according to the propagation scheme, which is the same for attitude.

If the user does not include an orbit or attitude object, then the class will not attempt to integrate it. If a orbit or attitude is supplied (via external methods such as from file or hardware), then the other motion may use the assigned orbit or attitude history to calculate the dynamics due to coupling. For example, if only an orbit dynamics equation is supplied to a coupled propagator, then the propagator could use an attitude history file to use when calculating the orbit dynamics.

An **EnckeCombinedPropagator** is a unique scheme that applies Encke corrections to the orbit propagation during attitude propagation. The user may inherit from the **NumericPropagator** or **AnalyticPropagator** to implement new propagation schemes.

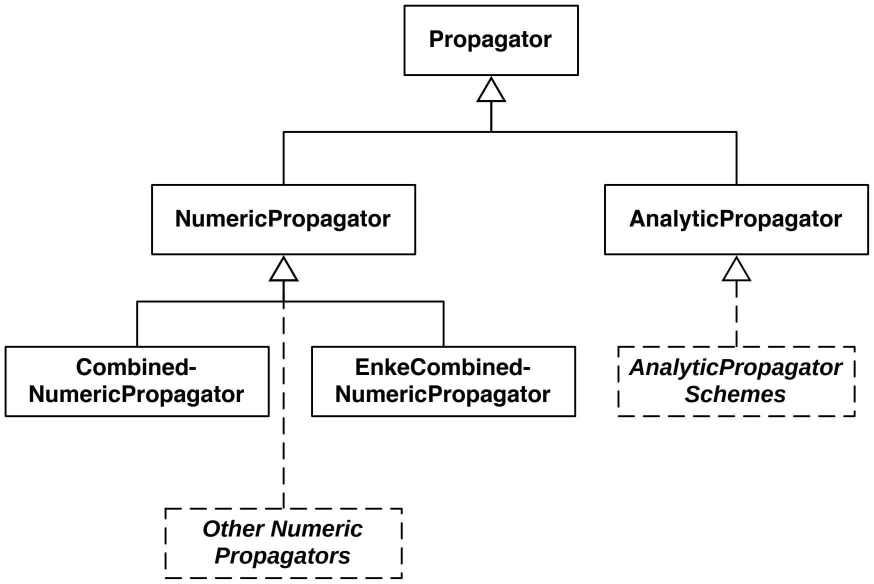
Propagator
NumericPropagator
AnalyticPropagator
Combined-NumericPropagator
EnkeCombined-NumericPropagator
AnalyticPropagator Schemes
Other Numeric Propagators

Figure 6.7: Dynamics library UML diagram. Like the Integrator class, Propagator provides an interface that is consistent for each of the derived classes. NumericPropagators use integrators to numerically compute the state at future times. AnalyticPropagators use closed-form solutions to evaluate the state at a future time.

Usage

The example code creates a new **CombinedNumericPropagator**, adds Runge-Kutta integrators, and sets the state conversion functions, as mentioned previously. The propagator is then assigned to the appropriate orbit and attitude object, and propagated for a set amount of time given the initial conditions.

```
// create and allocate memory for the propagators and integrators
CombinedNumericPropagator* myProp = new CombinedNumericPropagator;
RungeKuttaIntegrator* orbitIntegrator = new RungeKuttaIntegrator;
RungeKuttaIntegrator* attitudeIntegrator = new RungeKuttaIntegrator;

// specify the number of integration steps for the integrators
orbitIntegrator->SetNumSteps(100);
myProp->SetOrbitIntegrator(orbitIntegrator);
attitudeIntegrator->SetNumSteps(1000);
myProp->SetAttitudeIntegrator(attitudeIntegrator);

// specify the conversion functions from mesh points to states
myProp->SetOrbitStateConversion(&myOrbitStateConvFunc);
myProp->SetAttitudeStateConversion(&myAttitudeStateConvFunc);

// specify the propagator being used
myOrbit->SetPropagator(myProp);
myAttitude->SetPropagator(myProp);

// propagate
myProp->Propagate(integrationTimes,
            myOrbit->GetStateObject().GetState(),
            myAttitude->GetStateObject().GetState());
```

6.1.7 Data Handling

The *Data Handling* library is a collection of classes and functions for interacting with large sets of data as well as the external system environment. The **History** class and associated subclasses are used for storing the states of the spacecraft's orbit, attitude, or other parameters during simulation. The **Converter** collection of classes is used for saving and restoring the spacecraft states and parameters from a variety of forms, such as comma-separated value ASCII, MatLab, Satellite ToolKit (STK), or XML. Lastly, the communications software is included that allows an Open-SESSAME application to connect to networked machines to retrieve state, send state, or control multiple simulations or external software packages.

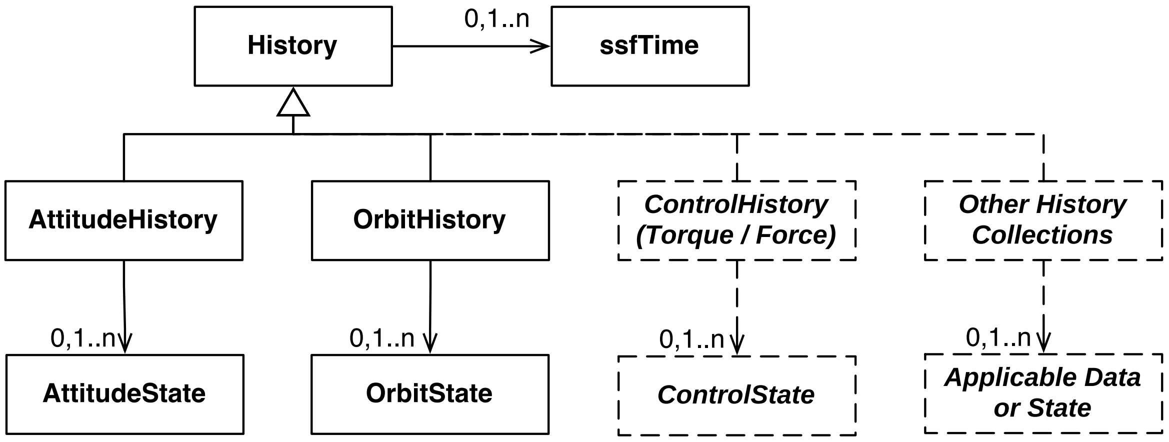

Figure 6.8: Data Handling toolkit UML diagram. History is a base class that provides a consistent interface and also stores the corresponding time value for each stored state. AttitudeHistory and OrbitHistory are derived classes that store AttitudeState and OrbitState respectively.

History

To succinctly store any number of states of the spacecraft, the **History** class provides a dynamically resizable vector of times, and derived classes add state variables that can be stored at associated times. For example, **OrbitStateHistory** and **AttitudeStateHistory** each respectively store the **OrbitState** and **AttitudeState** of the spacecraft after integration. These can then be retrieved, stored, or erased.

Furthermore, because integration only produces discrete meshpoints, the history objects include an **Interpolator** which is used to calculate requested states in between meshpoints. The current interpolators include **LinearInterpolator** and **CubicSplineInterpolator**, but developers are free to implement new interpolators as necessary.

```
History myHistory;   // create a history with an empty collection
myHistory.AppendHistory(0);  // add 0 seconds to the history
myHistory.AppendHistory(10);// add 10 seconds to the history
```

```
// Get a matrix of the stored times and output at t=3 s
cout << myHistory.GetHistory(3) << endl;
```

6.1.8 Time

The class **ssfTime** encapsulates simulation time, and allows for conversion to different time formats (*i.e.* UTC or Julian Date). Each time object is associated with a stored time, and an epoch time. Therefore, all time instances have a reference time they are measured from, such as the time since launch or a system clock time. The *Time* library also includes tools for using time objects, such as getting the current time, or measuring the operation time of a calculation (*tick* and *tock*).

```
// create an instance of the ssfTime object
ssfTime simTime;
// create a new time object and set it to 10 seconds
ssfSeconds integrationTime = 10;
// set the simTime instance to the value of intTime
simTime.Set(integrationTime);
// create a nowTime instance and set it to the clock time
ssfTime nowTime(Now());
```

6.2 Using the Framework

Because Open-SESSAME is a framework, there is no prescribed method for creating an application. An application is a stand-alone program that carries out a specified purpose, such as simulating a spacecraft, while the framework provides the tools necessary to build an application.

However, Open-SESSAME, by design, has suggested methods of implementing various applications, including attitude or orbit integration, coupled propagation, hardware-in-the-loop testing, or using libraries in flight code. The following sections present suggested architectures for these example applications. These designs should serve as a model for users developing their own applications.

6.2.1 Attitude Simulation

An attitude simulation is a stand-alone integration of the spacecraft attitude dynamics equation with a possible disturbance torque function. The following steps could be followed to simulate a spacecraft's attitude:

1. Code the attitude dynamics equation
2. Code the disturbance torque function
3. Assign function parameters
4. Create an initial attitude state
5. Create and initialize integrator
6. Integrate the equations
7. Graph or output the state history

These steps are shown in the code snippets blow and the module interconnections are illustrated in Figure 6.9.

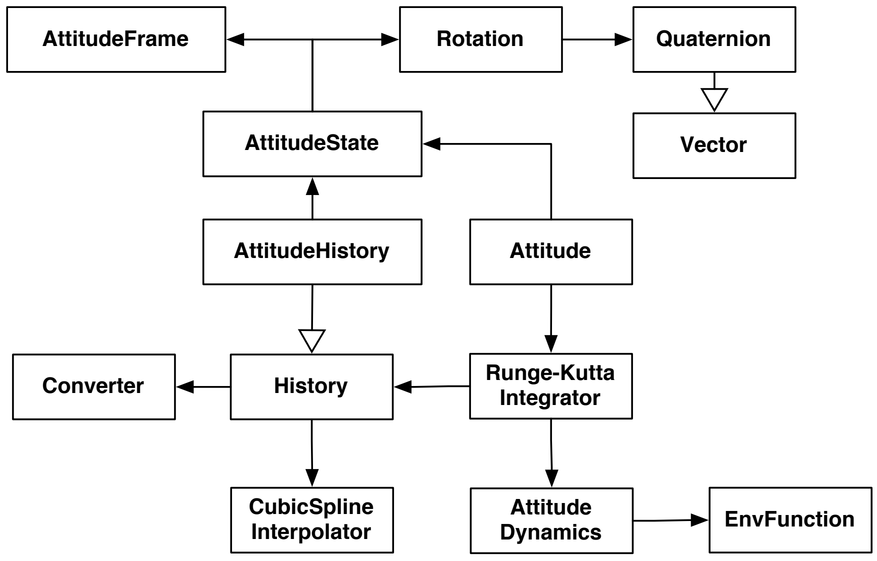

Figure 6.9: Attitude integration using Open-SESSAME.

Code the attitude dynamics equation

The dynamics equation is the right-hand side time derivative of the attitude dynamics. The function requires the current integration time, integrating state, spacecraft attitude and orbit (if applicable), a matrix of parameters, and a function pointer to the disturbance function.

```
static Vector AttituteDynamics(const ssfTime &_time,
                                    const Vector& _integratingState,
                                    Orbit *_Orbit, Attitude *_Attitude,
                                    const Matrix &_parameters,
                                    const Functor &_forceFunctorPtr)
{
   // initialize the variables, static to save memory allocation
    static Vector stateDot(7);
    static Matrix bMOI(3,3);
    static Matrix qtemp(4,3);
    static Matrix Tmoment(3,1);
    static Vector qIn(4);
    static Vector qDot(4);
    static Vector wIn(3);
    static Vector wDot(3);
    // get the state variables from the input integrating state
    qIn = _integratingState(_(1, 4));
    wIn = _integratingState(_(5,7));
    // normalize the quaternion
    qIn /= norm2(qIn);

    // calculate qDot
    qtemp(_(1,3),_(1,3)) = skew(qIn(_(1,3))) + qIn(4) * eye(3);
    qtemp(4, _(1,3)) = -(~qIn(_(1,3)));
    qDot = 0.5 * qtemp * wIn;

    // get the moments of inertia from the input parameters
    bMOI = _parameters(_(1,3),_(1,3));
    // calculate the disturbance torques
    Tmoment(_,_) = (_forceFunctorPtr.Call(_time, _Orbit->GetStateObject(),
                              _Attitude->GetStateObject()))(_);
    // calculate omegaDot
    wDot = (bMOI.inverse() * (Tmoment - skew(wIn) * (bMOI * wIn)));

    // setup the time derivate return state vector
    stateDot(_(1,3)) = qDot;
    stateDot(_(5,7)) = wDot;

    return stateDot;
}
```

Code the disturbance torque function

The disturbance function would contain any modeled torque disturbances. This example models no disturbance torques.

```
    Vector NullFunctor(const ssfTime& _pSSFTime,
                                 const OrbitState& _pOrbitState,
                                 const AttitudeState& _pAttitudeState)
    {
        return Vector(3);
    }
```

Assign function parameters

Function parameters are used to pass information into the dynamics equation. In this example, the moments of inertia are used in the right-hand side. The disturbance torque function is set to the empty modeling function from above.

```
    Matrix I(3,3); //I=[100 0 0;0 200 0;0 0 150];
        I(1,1) = 100;
        I(2,2) = 200;
        I(3,3) = 150;

    SpecificFunctor AttitudeTorquesFunctor(&NullFunctor);
```

Create an initial attitude state

The initial attitude is setup with a quaternion corresponding to no rotation and an angular velocity of 0.1 m/s about the x-axis.

```
AttitudeState myAttitudeState;
myAttitudeState.SetRotation(Rotation(Quaternion(0,0,0,1)));
Vector initAngVelVector(3);
    initAngVelVector(1) = 0.1;
myAttitudeState.SetAngularVelocity(initAngVelVector);
```

Create and initialize integrator

The Runge-Kutta integrator is configured with 1000 steps for 20 seconds.

```
RungeKuttaIntegrator myIntegrator;
myIntegrator.SetNumSteps(1000);

// Integration times
vector<ssfTime> integrationTimes;
ssfTime begin(0);
ssfTime end(begin + 20);
integrationTimes.push_back(begin);
integrationTimes.push_back(end);
```

Integrate the equations

The integration function is called within a “tick()” and “tock()” to calculate the operating time.

```
cout << "PropTime = " << begin.GetSeconds() << " s -> "
            << end.GetSeconds() << " s" << endl;
cout << "Attitude State: " << ~myAttitudeState.GetState() << endl;

tick();
Matrix history = myIntegrator.Integrate(
                        integrationTimes, // seconds
                        &AttituteDynamics,
                        myAttitudeState.GetState(),
                        NULL,
                        NULL,
                        I,
                        AttitudeTorquesFunctor
                    );
cout << "finished propagating in " << tock() << " seconds." << endl;
```

Graph or output the state history

Figure **??** shows an example attitude plot from Open-SESSAME using gnuplot. The formatting is plain, and allows the user to change the format as needed.

```
    cout << history;
    Matrix plotting = history(_,_(1,5));
```

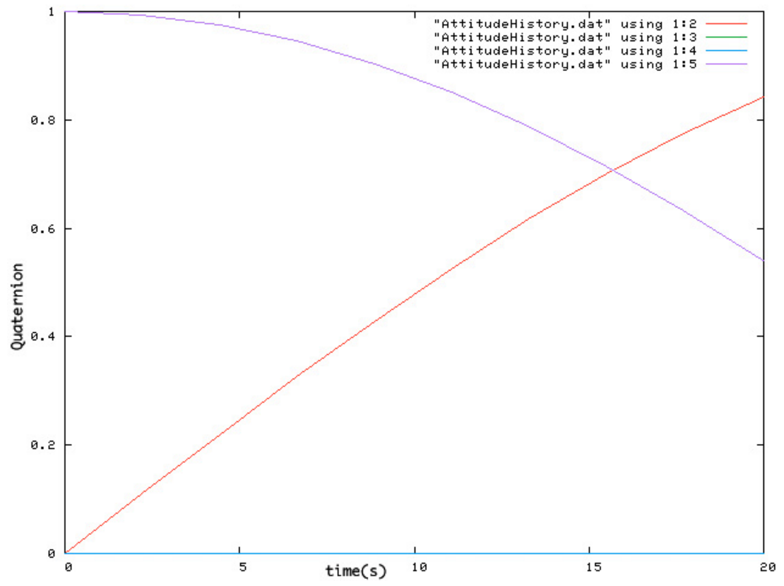

Figure 6.10: An example gnuplot output from Open-SESSAME of an attitude integration. A **Plot** class is provided that encapsulates formatting, labeling, and saving.

6.2.2 Orbit Simulation

An orbit simulation is a stand-alone integration of the spacecraft orbit dynamics equation with a possible disturbance function (such as gravity). The following steps could be followed to simulate a spacecraft orbit:

1. Code the orbit dynamics equation

2. Code the disturbance force function
3. Assign function parameters
4. Create an initial orbit state
5. Create and initialize integrator
6. Integrate the equations
7. Graph or output the state history

The component interaction is similar to attitude integration's design. The orbit simulation code snippets are shown below.

Code the orbit dynamics equation

The orbit dynamics equation is the right-hand side describing the orbit time derivative. It is similar to the attitude dynamics equation and uses the same parameters.

```
static Vector TwoBodyDynamics
(const ssfTime &_time, const Vector& _integratingState,
  Orbit *_pOrbit, Attitude *_pAttitude,
  const Matrix &_parameters,
  const Functor &_forceFunctorPtr)
{
    // setup the initial variables
    static Vector Forces(3);
    static Vector Velocity(3);
    static Vector stateDot(6);
    static AttitudeState tempAttState;
    static OrbitState orbState(new PositionVelocity);

    orbState.GetStateRepresentation()
                    ->SetPositionVelocity(_integratingState);

    // if the orbit is dependent on the attitude and
    // an attitude representation exists, calculate the
    // force function with the attitude
    if(_pAttitude)
```

```
        Forces = _forceFunctorPtr.Call(_time, orbState,
                        _pAttitude->GetStateObject());
    else
        Forces = _forceFunctorPtr.Call(_time, orbState, tempAttState);

    Velocity(_) = _integratingState(_(4,6));

    // form the time derivative vector
    stateDot(_(1, 3)) = Velocity(_);
    stateDot(_(4, 6)) = Forces(_);
    return stateDot;
}
```

Code the disturbance force function

This force function models two body gravity using the parameters passed into the function.

```
Vector GravityForceFunction(const ssfTime &_currentTime,
           const OrbitState  &_currentOrbitState,
           const AttitudeState &_currentAttitudeState,
           const EnvFuncParamaterType &_parameterList)
{
    static Vector Forces(3);
    static Vector Position(3);
    Position(_) = _currentOrbitState.GetState()(_(1,3));
    Forces = *(reinterpret_cast<double*>(_parameterList[0]))
                    / pow(norm2(Position),3) * Position;
    return Forces;
}
```

Assign function parameters

```
    Matrix Parameters(1,1);
    Parameters(1,1) = 398600.4418; //km / s^2
```

Create an initial orbit state

An initial orbit state is setup using position and velocity in the earth-centered inertial frame.

```
OrbitState myOrbit;
myOrbit.SetStateRepresentation(new PositionVelocity);
myOrbit.SetOrbitFrame(new OrbitFrameIJK);
Vector initPV(6);
    initPV(1) = -5776.6; // km
    initPV(2) = -157; // km
    initPV(3) = 3496.9; // km
    initPV(4) = -2.595;  // km/s
    initPV(5) = -5.651;  // km/s
    initPV(6) = -4.513; // km/s
myOrbit.GetStateRepresentation()->SetPositionVelocity(initPV);
```

Create and initialize integrator

```
 RungeKuttaIntegrator myIntegrator;
myIntegrator.SetNumSteps(100);

vector<ssfTime> integrationTimes;
ssfTime begin(0);
ssfTime end(begin + 100);
integrationTimes.push_back(begin);
integrationTimes.push_back(end);
```

Integrate the equations

```
cout << "PropTime = " << begin.GetSeconds() << " s -> "
          << end.GetSeconds() << " s" << endl;
cout << "Orbit State: "
       << ~myOrbit.GetStateRepresentation()->GetPositionVelocity()
       << endl;
tick();
Matrix history = myIntegrator.Integrate(
        integrationTimes, // seconds
        &TwoBodyDynamics,
        myOrbit.GetStateRepresentation()->GetPositionVelocity(),
        NULL,
        NULL,
        Parameters,
        OrbitForcesFunctor
        );

cout << "finished propagating in " << tock() << " seconds." << endl;
```

Graph or output the state history

Figure 6.11 shows an example output from this example case. However, since the simulation time is short (100 seconds), an extended length output is shown in Figure 6.12. Currently, Open-SESSAME only uses a basic gnuplot output, with formatting possible by the user.

```
    Matrix plotting = history(_,_(2,4));
    Plot3D(plotting);
```

6.2.3 Coupled Simulation

Coupled simulation involves integrating the orbit and attitude dynamic equations with some dependence of one dynamic or disturbance function on the state of the other. As discussed previously, there are several different schemes for propagating coupled equations. However, each scheme would employ the same, following method. This implementation assumes the user is using the orbit and attitude dynamics, distubance functions, function parameters, and initial state from before. The entire application architecture is shown in Figure 6.13.

1. Create and populate the environment
2. Define orbit and attitude conversion functions
3. Create and initialize the propagator
4. Propagate the equations

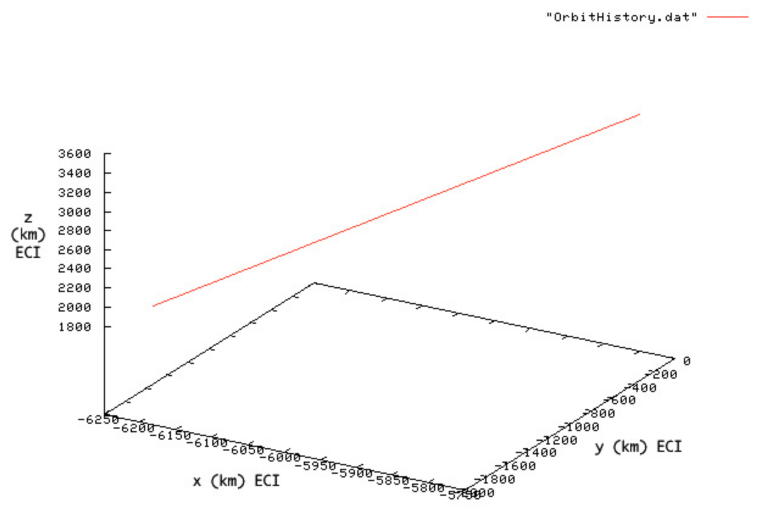

Figure 6.11: Example gnuplot output of an orbit integration. While this plot is an accurate plot of the position during teh simulation, it does not demonstrate the orbit well.

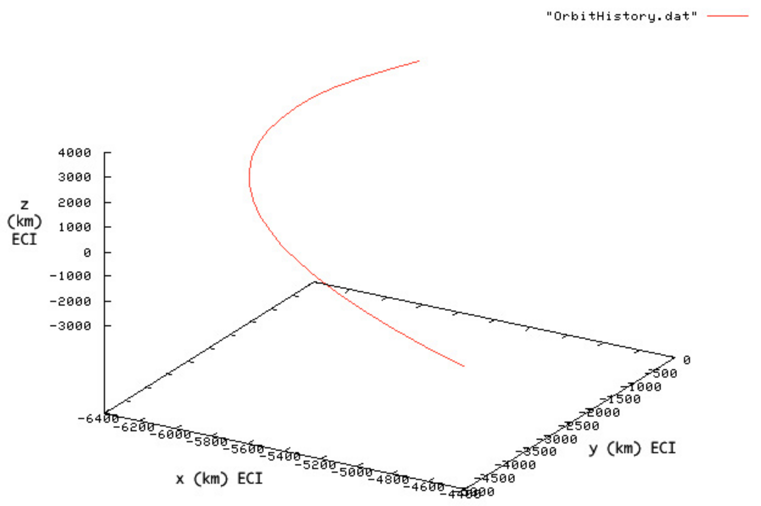

Figure 6.12: An extended length example plot of orbit integration. This plot is the using the simulation as shown in Figure 6.11 but with a longer simulation time to demonstrate the plotting of an orbit where the geometry can be seen.

5. Graph or output the state history

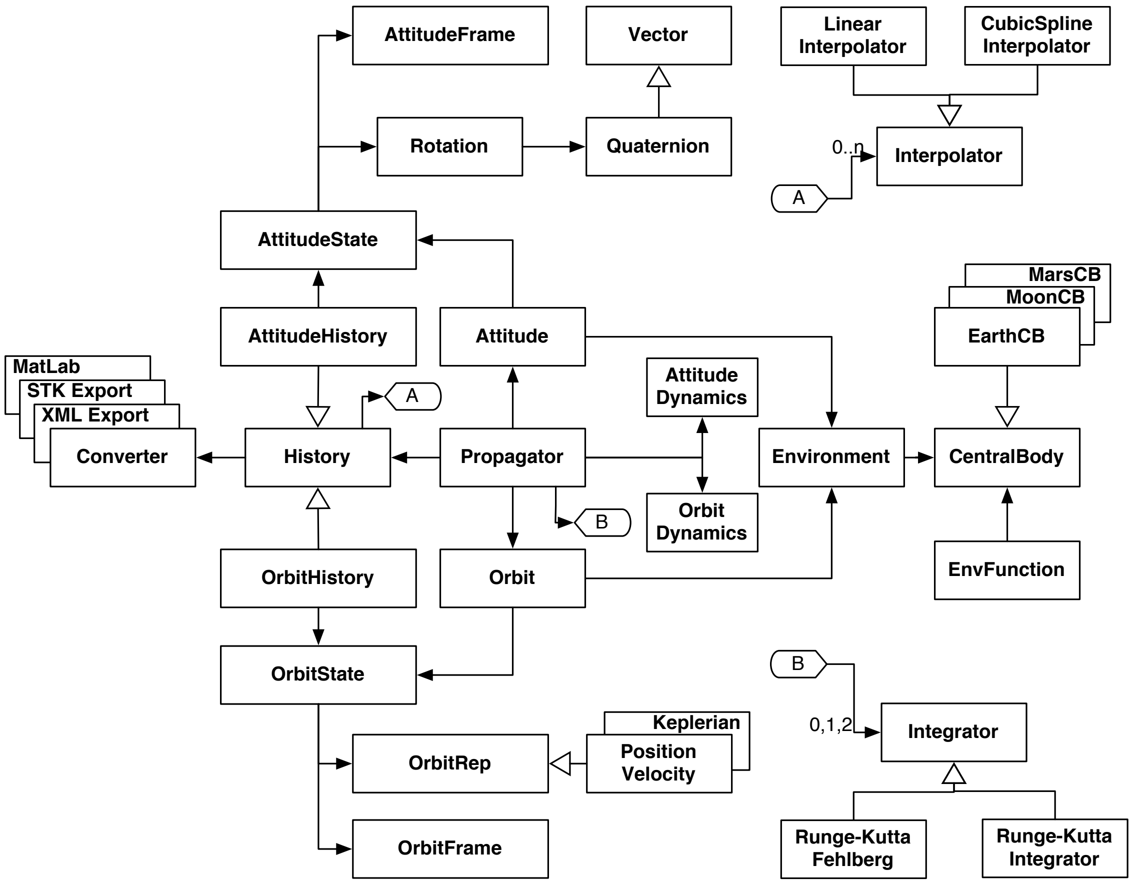

Figure 6.13: UML diagram of an Open-SESSAME coupled orbit simulation application.

Create and populate the environment

```
Environment* pEarthEnv = new Environment;
EarthCentralBody *pCBEarth = new EarthCentralBody;
pEarthEnv->SetCentralBody(pCBEarth);

// Add Gravity force function
cout << "Filling Parameters" << endl;
EnvFunction TwoBodyGravity(&GravityForceFunction);
pEarthEnv->AddForceFunction(TwoBodyGravity);
```

```
    // Add Drag Force Function
    EnvFunction DragForce(&DragForceFunction);
    double *BC = new double(200);
    DragForce.AddParameter(reinterpret_cast<void*>(BC), 1);
    double *rho = new double(1.13 * pow(10., -12.)); // kg/m^3
    DragForce.AddParameter(reinterpret_cast<void*>(rho), 2);
    pEarthEnv->AddForceFunction(DragForce);

    myOrbit->SetEnvironment(pEarthEnv);
    myAttitude->SetEnvironment(pEarthEnv);
```

Define orbit and attitude conversion functions

```
void myOrbitStateConvFunc(const Matrix &_meshPoint,
               OrbitState &_convertedOrbitState)
{
    static Vector tempVector(_meshPoint[MatrixColsIndex].getIndexBound() - 1);
    tempVector(_) =
          ~_meshPoint(_, _(2, _meshPoint[MatrixColsIndex].getIndexBound()));
    _convertedOrbitState.SetState(tempVector);
    return;
}

void myAttitudeStateConvFunc(const Matrix &_meshPoint,
            AttitudeState &_convertedAttitudeState)
{
    static Vector tempQ(4); tempQ(_) = ~_meshPoint(_,_(2, 5));
    static Vector tempVector(3); tempVector(_) = ~_meshPoint(1, _(6, 8));
    _convertedAttitudeState.SetState(Rotation(Quaternion(tempQ)), tempVector);
    return;
}
```

Create and initialize the propagator

```
     CombinedNumericPropagator* myProp = new CombinedNumericPropagator;

    // Create & setup the integator
```

```
// Setup an integrator and any special parameters
RungeKuttaIntegrator* orbitIntegrator = new RungeKuttaIntegrator;
RungeKuttaIntegrator* attitudeIntegrator = new RungeKuttaIntegrator;

orbitIntegrator->SetNumSteps(100);
myProp->SetOrbitIntegrator(orbitIntegrator);
attitudeIntegrator->SetNumSteps(1000);
myProp->SetAttitudeIntegrator(attitudeIntegrator);

myProp->SetOrbitStateConversion(&myOrbitStateConvFunc);
myProp->SetAttitudeStateConversion(&myAttitudeStateConvFunc);

myOrbit->SetPropagator(myProp);
myAttitude->SetPropagator(myProp);
```

Propagate the equations

```
int numOrbits = 5
vector<ssfTime> integrationTimes;
ssfTime begin(0);
ssfTime end(begin + 92*60*numOrbits);
integrationTimes.push_back(begin);
integrationTimes.push_back(end);
cout << "PropTime = " << begin.GetSeconds() << " s -> "
        << end.GetSeconds() << " s" << endl;
cout << "Orbit State: "
        << ~myOrbit->GetStateObject().GetState() << endl;
cout << "Attitude State: "
        << ~myAttitude->GetStateObject().GetState() << endl;

// Integrate over the desired time interval
myPropagator->Propagate(integrationTimes,
      myOrbit->GetStateObject().GetState(),
      myAttitude->GetStateObject().GetState());
```

Graph or output the state history

```
Matrix orbitHistory = myOrbit->GetHistory().GetHistory();
// plotting rx, ry, rz
Matrix orbitPlotting = orbitHistory(_,_(2,4));
Matrix attitudeHistory = myAttitude->GetHistory().GetHistory();
// plotting t:(q1,q2,q3,q4)
Matrix attitudePlotting = attitudeHistory(_,_(1,5));

Plot3D(orbitPlotting);
Plot2D(attitudePlotting);
```

6.2.4 Simulation with Flight Components

Another benefit of using the Open-SESSAME framework is the ability to configure a simulation that can integrate with flight hardware and software for simulating the space environment and flight operations. Software components interact with an Open-SESSAME simulation during testing, and are incrementally replaced with hardware components as they become available. Testing then can verify the operation of the hardware, on the bench, in the same simulation environment.

Figure 6.14 shows the application architecture and interaction between flight software and an Open-SESSAME simulation. The sensor stubs interact with the flight software like the hardware drivers, but query the simulation application for the current state information. The sensor stub then converts this simulated state information into the sensor's expected output, and adds any simulated measurement errors. Communication occurs through a socket connection, which allows the simulation server to reside and operate on a separate machine if necessary.

6.2.5 Integrating with External Programs

The Open-SESSAME framework also includes tools for communicating and exporting to external programs such as Satellite ToolKit or MatLab. The simplest method is to convert the state history data to the external program's format, and then loading the history file into the external program. As new external programs are required, new converters can be implemented.

Another means of interfacing an Open-SESSAME simulation to an external program is

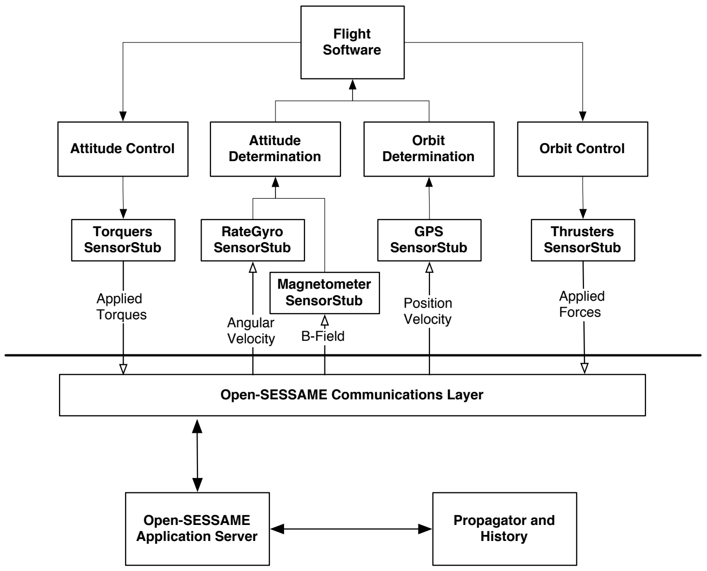

Figure 6.14: Use of Open-SESSAME for hardware-in-the-loop testing and integration. The compoents above the horizontal line are flight hardware software components. The software stubs communicate via a communications layer to an Open-SESSAME application server. These stubs then simulate hardware measurement error and output. The stubs can then be replaced as hardware components are introduced.

to create a real-time connection, such as shared memory, that provides data or controls the external program. An example of connecting to an external program is providing real-time attitude data to an STK simulation. Connections are made with socket and commands are sent as standard ASCII lines that can control the STK simulation and provide the real-time data. This type of connection would use some of the tools from the Open-SESSAME communication library, and send commands as per STK's prescribed interface [46].

6.3 Summary

This chapter presented the reader the design of the Open-SESSAME framework and all the components that make up the framework's libraries and toolkits. UML diagrams showed the interaction of modules, and layed out the development of simulation applications using Open-SESSAME. Code snippets showed specific examples of how to implement various simulation components that comprise an application. Finally, a discussion of using Open-SESSAME with hardware and flight software or external programs was presented.

Chapter 7

Verification and Validation

This chapter presents the verification and validation of the Open-SESSAME Framework and its components. The purpose and methods for performing the tests are given. Following, the results from tests and comparisons are discussed as well as an overall summary of the validity of Open-SESSAME.

7.1 Purpose

A verification and validation effort is required in order to assure the user and any clients reviewing the validity of analysis results from the Open-SESSAME Framework. Verification of the software involves checking that the numerical output of the simulation matches (or closely approximates) an accepted analytic solution, or output from other verified software programs. Validation is demonstrating that the software solution meets the purposes and requirements of the project. In the case of Open-SESSAME, the requirements stated that the framework is to provide an extensible library of software tools for analysing and modeling spacecraft.

The verification and validation of the software should serve as an assurance that the software output can be trusted. Users may rely on the provided libraries being accurate and then need only worry about their specific simulation problem. The software may also be verified by inspecting the code, since it is provided in the framework package.

7.2 Methods

Three methods are used for verifying the Open-SESSAME framework:

1. Inspection of the software code to ensure that the implementation matches the derived algorithms.
2. Comparison of the output to a closed-form analytic solution to the dynamics equations.
3. Comparison of the output to an industry standard software package using the same or similar modeling parameters.

Inspection of the code has been performed by the developers to date, but is also left to future users to verify and assure themselves that the code is implemented as documented. The code is written with verbose variable and function naming to assist in white-box inspection. White-box inspection is a means of verification by a human manually reading over the code to check the programs logic and operation. Furthermore, the embedded documentation includes the mathematical algorithm, in symbolic form, that is implemented in the function.

Simple equations of motion (assuming axisymmetric body or with a simple disturbance force) have a closed-form solution. These solutions are evaluated at any arbitrary time of the spacecraft simulation. The calculation from this solution is compared to the same simulation parameters when the equations of motion are integrated numerically.

Finally, for simulations that are complex (contain many disturbances or have a non-symmetric body), comparisons with verified programs is the easiest and fastest means of verification. Industry standard spacecraft simulation packages, such as STK or FreeFlyer, have documented verification, and have flight heritage to compare with the simulations. A comparison is made by setting up these software packages with the same configuration as an Open-SESSAME simulation application. The output is compared after propagating both the Open-SESSAME and the application's simulation.

Table 7.1: Comparison of orbit simulation with closed-form solution without forces.

$\mathbf{r}_0^i$	$[1131.34, -2282.343, 6672.423]^T km$
$\mathbf{v}_0^i$	$[-5.64305, 4.30333, 2.42879]^T km/s$
t	2400 sec
Integrator	RK-4(5), tol: 10×10^{-9}
Output	
Running Time	0.59 sec
$\mathbf{r}_{Kep}^i(t)$	$[-4219.853, 4363.116, -3958.789]^T km$
$\mathbf{v}_{Kep}^i(t)$	$[3.689736, -1.91662, -6.112528]^T km/s$
$\mathbf{r}_{Sim}^i(t)$	$[-4219.752, 4363.029, -3958.766]^T km$
$\mathbf{v}_{Sim}^i(t)$	$[3.689865, -1.916734, -6.112511]^T km/s$
RMS error	$\Delta\mathbf{r}_{RMS} = 3.1493 \times 10^{-5}$
	$\Delta\mathbf{v}_{RMS} = 6.95115 \times 10^{-5}$

7.3 Closed-Form Solution Results

7.3.1 Orbit Verification

In Section 4.3.3 we derived the equation for a closed-form solution of the orbit equation known as Kepler's Equation (Equation 4.55). By calculating the orbit state using this closed-form solution and comparing to a numerical integration with Open-SESSAME, we can verify the accuracy of the integration of a two-body point mass orbit simulation.

Table 7.1 shows the results from performing a short term (40 minutes) simulation of a satellite with only two-body forces. The simulation results are numerically close to the closed-form result with little required computation time. The tolerance of the integrator can be decreased, or the order of the integrator increased to achieve better accuracy at the cost of a higher computation time.

7.3.2 Attitude Verification

In Equations 3.48 and 3.51, the derivations of a closed-form solution for an axisymmetric body was derived. These are useful for verifying simple cases of attitude simulation. A time-varying torque is applied about an axisymmetric body. The closed-form and integrated solutions are compared. The results are shown in Table 7.2, and demonstrate

Table 7.2: Comparison of attitude simulation with closed-form solution with time-varying torques.

ω_0	$[0.3, -0.4, 0.7]^T rad/s$
I_t	100 kg m^2
I_3	150 kg m^2
$g(t)$	$\begin{bmatrix}0.2\sin t & -0.4\sin t & 0\end{bmatrix}^T Nm$
t	300 sec
Integrator	RK-4(5), tol: 10×10^{-8}
Output	
$\omega(t)$	$\begin{bmatrix}-0.46134984 & -0.194318263 & 0.7\end{bmatrix}^T rad/s$
ω_{Sim}	$\begin{bmatrix}-0.46134984100 & -0.193182629296 & 0.7\end{bmatrix}^T rad/s$
RMS error	$\sqrt{(-1.15712 \times 10^{-9})^2 + (2.942222 \times 10^{-9})^2} = 3.165158 \times 10^{-9} rad/s$

that the implementation of attitude dynamics using quaternion and angular velocities is accurate. Higher precision can be acheived by decreasing the tolerance of the variable step integrator. Also, dynamics equations of other representations (*i.e.* Modified Rodriguez Parameters, Angular Momentum) should be verified as necessary.

7.4 Software Comparison Results

Satellite ToolKit and FreeFlyer were chosen as the two baseline spacecraft simulation packages to compare with Open-SESSAME. These packages are chosen because they are the most mature, and have flight heritage. Furthemore, both STK and FreeFlyer are considered the industry leaders in satellite modeling and analysis, and have easy to use interfaces for setting up configurations that match Open-SESSAME. However, neither program provides support for advanced attitude modeling that includes environmental disturbance torques or control inputs.

Orbits were simulated independently to inspect their individual operation. Both short term and long term simulations are presented, in simple and complex configurations. Since neither of the benchmark programs contain the functionality necessary to model coupled simulation, the verification of attitude dynamics will reside in the closed-form comparison. The validation of Open-SESSAME described below demonstrates the func-

Table 7.3: Comparison of simulations of a low-earth orbit

Parameter	**Value**		
Initial Orbit State	$\mathbf{r}_0 = \begin{bmatrix}-2216.05436 & -2837.09413 & -6235.38291\end{bmatrix}^T$ $\mathbf{v}_0 = \begin{bmatrix}6.938058 & -5.419317 & 0\end{bmatrix}^T$		
Disturbances	Two-Body Gravity		
Integrator	Runge-Kutta-Fehlberg 4(5)		
Simulation Time	86,400 s		
Output	**O-SESSAME**	**STK**	**FreeFlyer**
Final Orbit State	$\begin{bmatrix}813.3068066\\9186.344286\\13406.4852\\-3.32870222\\1.370725213\\-1.678024353\end{bmatrix}$	$\begin{bmatrix}813.238901\\9186.370669\\13406.4847\\-3.328704\\1.370709\\-1.678049\end{bmatrix}$	$\begin{bmatrix}813.2919879\\9186.350424\\13406.47774\\-3.32870249\\1.370721908\\-1.678029149\end{bmatrix}$
Running Time	3 sec	1 sec	7 sec
RMS Error	-	8.57×10^{-5}	1.86×10^{-5}

tionality of this coupled simulation.

7.4.1 Orbit Simulation

Low Earth Orbit

The simple modeling of a satellite in Low-Earth is shown in Table 7.4. The errors of Open-SESSAME when compared to STK and FreeFlyer are favorable. Furthermore, Open-SESSAME is faster than FreeFlyer. When performing a long-term integration, as shown in Table 7.4, the error grows and computation takes much longer. It will be necessary to optimize the Open-SESSAME application for use in long-term simulations to cut down on run time.

Geostationary Orbit

A geostationary satellite has a slow, nearly constant angular velocity. Therefore, large and constant integration stepsizes are sufficient for accurate simulation. Table 7.5 shows the

Table 7.4: Comparison of simulations of a low-earth orbit over long-term

Parameter	Value		
Initial Orbit State	$\mathbf{r}_0 = \begin{bmatrix}-2213.110462 & -2839.391147 & -6235.382907\end{bmatrix}^T$ $\mathbf{v}_0 = \begin{bmatrix}6.943675 & -5.412118 & 0\end{bmatrix}^T$		
Disturbances	Two-Body Gravity		
Intregrator	Runge-Kutta-Fehlberg 4(5)		
Simulation Time	20 days		
Output	**O-SESSAME**	**STK**	**FreeFlyer**
Final Orbit State	$\begin{bmatrix}5541.623013\\-6308.090106\\-2716.871416\\5.55366187\\-0.602634833\\5.090185196\end{bmatrix}$	$\begin{bmatrix}5543.012599\\-6308.239404\\-2715.59583\\5.552857\\-0.601718\\5.090581\end{bmatrix}$	$\begin{bmatrix}5542.078932\\-6308.138143\\-2716.45159\\5.55339773\\-0.602332835\\5.090316517\end{bmatrix}$
Running Time	45 sec	4 sec	4 sec
RMS Error	-	1.62×10^{-3}	5.34×10^{-4}

output from an Open-SESSAME simulation of a geostationary satellite for one simulation day. The computation was relatively fast with little error when compared to STK and FreeFlyer.

High Eccentricity Orbit

The low-earth orbit and geosynchronous simulations modeled a satellite at a nearly constant angular velocity because the orbit is circular, or near circular. However, it is useful to verify the operation of Open-SESSAME with a satellite that experiences large velocity changes over an orbit. This case can be simulated with an high eccentricity orbit. At periapsis, the lowest point of the orbit, the satellite has a high velocity, and at apoapsis, the farthest point, the satellite has a low velocity. During periapsis passage, the spacecraft has an increased velocity, which can cause simulation errors due to the large distances in short time periods. Table 7.6 shows the results of this simulation. Figure 7.1 is a plot of the position RMS error versus the altitude of the spacecraft. The error is greater at a lower altitude, corresponding to a higher velocity. This error could be reduced by decreasing the tolerance. Also, the error is low at a low altitude at the beginning of the simulation since both simulations were given the same initial conditions.

Table 7.5: Comparison of simulations of a geostationary orbit

Parameter	Value		
Initial Orbit State	$\mathbf{r}_0 = \begin{bmatrix}14580.17548 & 39563.05959 & 0\end{bmatrix}^T$ $\mathbf{v}_0 = \begin{bmatrix}-2.884984 & 1.063203 & 0\end{bmatrix}^T$		
Disturbances	Two-Body Gravity		
Integrator	Runge-Kutta-Fehlberg 4(5)		
Simulation Time	86,400 s		
Output	**O-SESSAME**	**STK**	**FreeFlyer**
Final Orbit State	$\begin{bmatrix}13897.40111\\39808.0323\\0\\-2.9028477\\1.0134143\\0\end{bmatrix}$	$\begin{bmatrix}13897.45665\\39808.01301\\0\\-2.902846\\1.013419\\0\end{bmatrix}$	$\begin{bmatrix}13897.40079\\39808.03247\\0\\-2.902847711\\1.013414275\\0\end{bmatrix}$
Running Time	3.5 sec	1 sec	2 sec
RMS Error	-	6.16×10^{-6}	3.37×10^{-8}

Open-SESSAME agrees closely with STK, both of which have a fast calculation time. FreeFlyer, however, has a slow calculation time and greater error when compared to Open-SESSAME and STK.

7.5 Validation

While the Open-SESSAME framework is verified by comparing the accurate numerical output to the closed-form solution and other software packages, it is necessary to validate the framework design. Open-SESSAME should make simulation of spacecraft easier for users, rather than imposing unnecessary complications.

7.5.1 Simulation Validation

The Virginia Tech HokieSat, a nanosatellite that is being built by students to launch as a part of the Ionospheric Nanosatellite Formation (ION-F) program, is currently modeled in software using a monolithic simulation function that is tied in closely with the flight code. When a change in the simulation parameters is necessary, software developers on

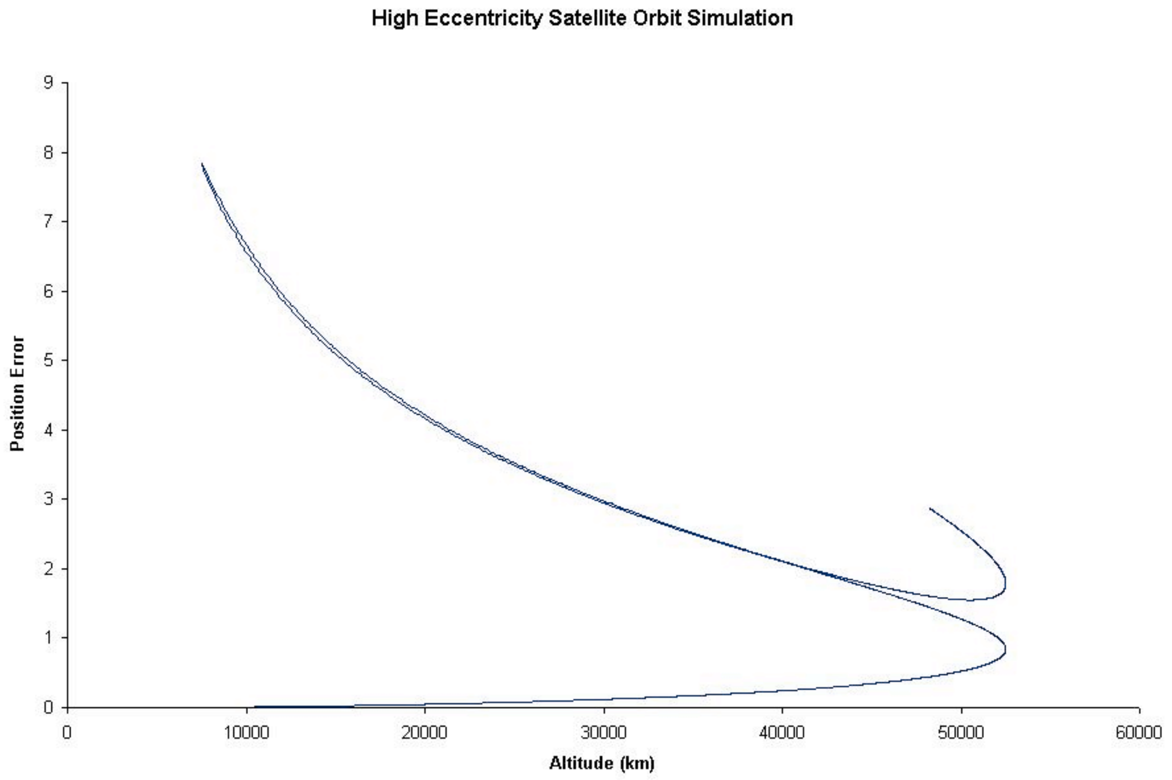

Figure 7.1: High Eccentricity Simulation Error. The altitude of the spacecraft is plotted against the position RMS error at that altitude.

Table 7.6: Comparison of simulations of a high eccentricity orbit

Parameter	**Value**		
Initial Orbit State	$\mathbf{r}_0 = \begin{bmatrix}7500 & 0 & 0\end{bmatrix}^T$ $\mathbf{v}_0 = \begin{bmatrix}0 & 6.819333903 & 6.819333903\end{bmatrix}^T$		
Disturbances	Two-Body Gravity		
Integrator	Runge-Kutta-Fehlberg 4(5)		
Simulation Time	20 days		
Output	**O-SESSAME**	**STK**	**FreeFlyer**
Final Orbit State	$\begin{bmatrix}-46724.94751\\-8276.177337\\-8276.177337\\1.339061791\\-0.85741544\\-0.85741544\end{bmatrix}$	$\begin{bmatrix}-46727.35881\\-8275.048124\\-8275.048124\\1.338824\\-0.857447\\-0.857447\end{bmatrix}$	$\begin{bmatrix}-46713.96349\\-8283.620519\\-8283.620519\\1.340490327\\-0.857151194\\-0.857151194\end{bmatrix}$
Running Time	15 sec	4 sec	48 sec
RMS Error	-	2.72×10^{-4}	1.73×10^{-3}

the project are required to modify hard-coded values in the flight code. Furthermore, the software is poorly documented, and subject to errors. To add functionality to the HokieSat simulation, the new algorithms must be implemented and tied in without disturbing the existing simulation. When software development is complete, the simulation must be extracted from the flight code.

The Open-SESSAME validation effort demonstrates the benefits of this new framework by developing a simulation application for use in testing the HokieSat flight code and hardware. The application uses Open-SESSAME framework components to provide an easy to use simulation that is easily configurable for testing the flight software in various operating conditions. The simulation is also transparent to the developer and separate from the flight code.

7.5.2 HokieSat Simulation Application

The ION-F mission requires that HokieSat fly at a relatively low altitude, approximately 400 km above the Earth. Atmospheric drag will be an important disturbance force that directly affects the operational lifetime of the satellite. However, the current simulator

Table 7.7: Summary of simulation components for HokieSat using Open-SESSAME

	Attitude	Orbit
State	Quaternion, Angular Velocity	Position, Velocity
Model	Euler	Two-Body Point Masses
Environment	Gravity-Gradient	Magnetic Field
Integrators	Runge-Kutta-Fehlberg 4(5)	Runge-Kutta-Fehlberg 4(5)
	Tol: 10×10^{-6}, Step: $0.01 \Rightarrow 0.25$	Tol: 10×10^{-6}, Step: $0.01 \Rightarrow 0.25$
Propagator	Weakly Coupled	Uncoupled

does not model atmospheric drag. The Open-SESSAME based simulation application will not include atmospheric density at first, in order to verify the switch-over from the original simulator to the new simulator, but the new application includes support for increasingly complex atmosphere models.

HokieSat uses magnetic torque coils for attitude control, and a magnetometer to measure the Earth's local magnetic field. Therefore, the simulation should include a magnetic field model of the Earth. For a first run, a simple tilted dipole model is sufficient. As the flight software program progresses, a higher-order model of the magnetic field is desired.

The new simulator also needs to include gravity-gradient disturbance torque, and allow for force and torque inputs from the orbit and attitude controllers respectively. In order to separate the flight code from the simulation code, a communications socket is established. The simulator receives requests for attitude and orbit, or setting of control torques from the flight code. The simulator is propagating in real-time, or some specified factor of real-time. As requests are made, or parameters are set, the simulator updates its data and continues simulating. This design concept is layed out in Section 6.2.4.

To summarize, a simulation for HokieSat includes the components presented out in Table 7.7. Initial conditions are parsed from a file to allow developers and student engineers to quickly change the operating parameters of the simulation without having to modify and recompile the Open-SESSAME simulation code.

7.5.3 HokieSat Simulation Results

The code for the *main* function of the HokieSat simulation application built using Open-SESSAME is included in Appendix A. Figures 7.2 and 7.3 are output plots of the orbit and attitude of HokieSat with no control-input. Graphs like these can be output on a regular basis to update the user as to the status of the simulation as it is operating.

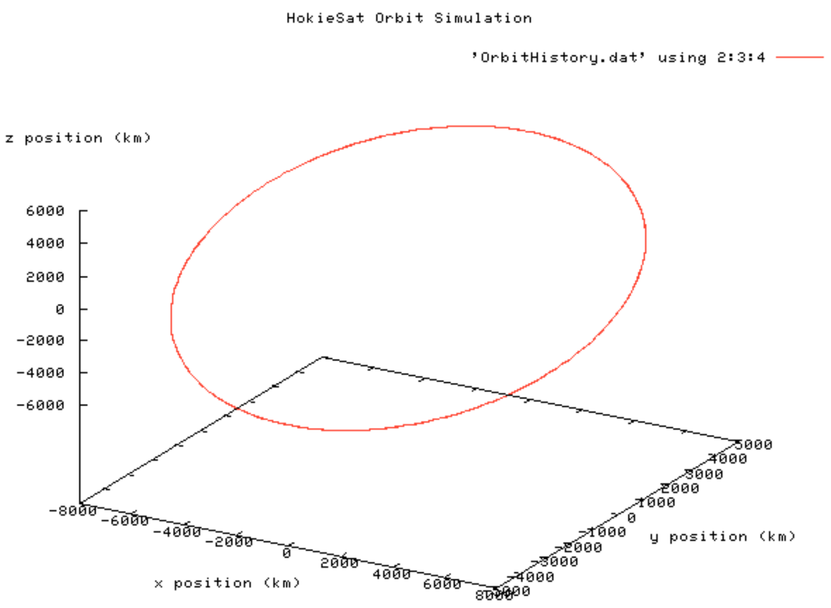

Figure 7.2: Example Gnuplot output of HokieSat orbit position.

Another benefit of developing this Open-SESSAME simulation server is that the application can be used for other engineering projects. For example, the Space Systems Simulation Laboratory is developing several spacecraft simulation tables to model attitude dynamics using various hardware and software control techniques. It would be useful to model the physical space environment and propagate the orbit of the spacecraft to provide to the control and sensing algorithms to increase the scope of the DSACSS project. The Open-SESSAME simulation server could be used with the DSACSS tables by providing a new configuration file containing the desired modeling characteristics of a DSACSS simulation. Also, any desired sensor stubs would be added to the spacecraft table computers to interface to Open-SESSAME. Therefore, the DSACSS project gains the use of a full-featured, configurable simulation server with little effort required by the team.

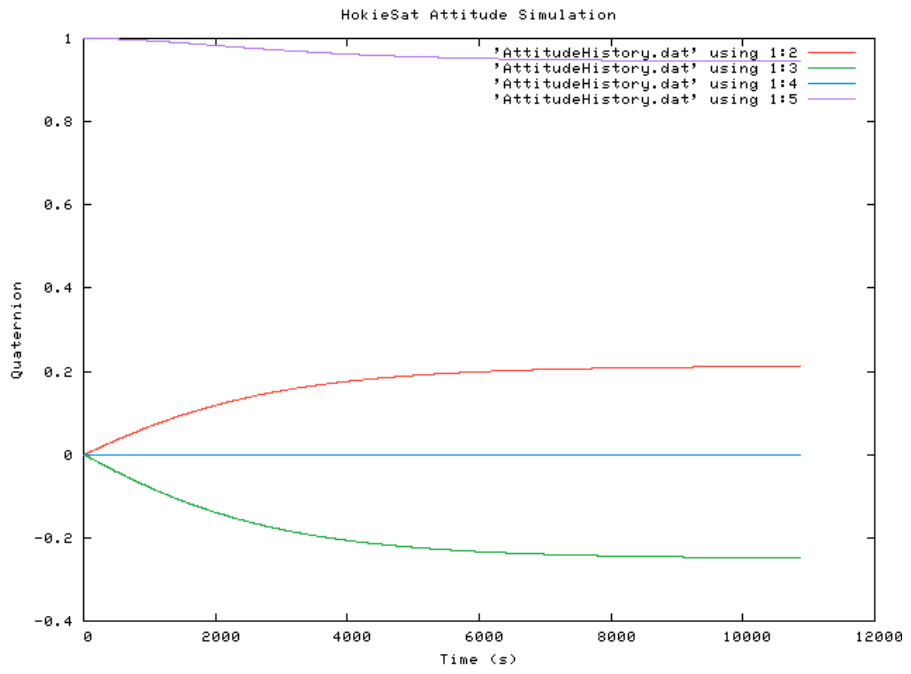

Figure 7.3: Example Gnuplot output of HokieSat attitude.

7.6 Summary

This chapter demonstrated the validity of the dynamics and modeling algorithms used in Open-SESSAME when compared against closed-form analytic solutions as well as agasint numerical integration using industry accepted spacecraft simulation packages. Furthermore, in some of the test cases Open-SESSAME was demonstrated to be faster than the simulation packages. Validation of Open-SESSAME involved implementing a simulation for HokieSat which is easier to use and understand for the HokieSat team of engineering. Open-SESSAME also provides enhanced functionality and the ability to improve the simulation server as requirements change.

Chapter 8

Conclusions

An extensible spacecraft simulation and modeling framework was developed and implemented. The software has been released as open-source and will continue to mature and grow. Results are summarized below, along with recommendations for future work with Open-SESSAME.

8.1 Summary

The need for an open-source, freely available satellite modeling tool was presented. The requirements called for developing the software the is understandable by an engineer or student who has some programming experience and wants to analyze and model a spacecraft, or test flight hardware and software.

The background concepts that form the underlying algorithms were developed and presented. A software achitecture was developed that incorporated these algorithms while also allowing future users the ability to add new algorithms with little or no required changes to the existing code. Furthermore, the architecture also allows for multiple configurations based on the needs of the users, as well as allowing the software libraries to be extracted for use in flight code and other analysis ventures.

The framework architecture was implemented in software along with documentation of the implementation, interface, and usage. This complete package is made available to the public via a popular internet repository for open-source projects. Code may be downloaded, or new code uploaded through this repository, which allows the software to remain free and continue to grow.

A verification effort demonstrated the correct implementation of the mathematical simulation algorithms. The Open-SESSAME demonstration application output matches that of analytical solutions as well as numerical solutions from industry accepted commercial satellite simulation packages. The documentation also includes the documentation and references of the algorithms implemented in the software, giving users the ability to independently verify and validate the operation of the Open-SESSAME libraries and toolkits.

8.2 Recommended Future Work

The Open-SESSAME framework is in a usable state for developing simple to moderate applications of orbit and attitude simulations. The code also includes the ability to provide a simulation server to hardware and external software that are not part of Open-SESSAME. Future work should begin with adding more models, environmental disturbances, and math algorithms (*i.e.* integration and interpolation). These algorithms can be implemented as individual components as necessary and added to the public repository.

Longer term goals should include developing automated unit testing for maintaining verification of the framework components. As new models are added, or existing code is optimized, the automated testing would alert the developer to any introduced errors. This verification goes with the future need to optimize the operation of the code to speed up calculation.

A future goal could also be to develop some form of better visualization using any number of existing open-source programs like *Celestia*, *Spacecraft Modeler*, or *GeomView*. These visualization add-ons would display a real-time, or postprocessing display of the spacecraft's attitude and orbit.

References

[1] F. E. Cellier, "Combined Continuous/Discrete System Simulation Languages - Usefulness, Experiences and Future Development," in *Methodology in Systems Modelling and Simulation*, (North-Holland, Amsterdam, the Netherlands), pp. 201–220, 1979.

[2] H. Elmqvist, F. E. Cellier, and M. Otter, "Object-oriented Modeling of Hybrid Systems," in *ESS'93, European Simulation Symposium*, (Delft, The Netherlands), pp. 25–28, 1993.

[3] M. Otter, H. Elmqvist, and F. E. Cellier, "Modeling of Multibody Systems with the Object-Oriented Modeling Language Dymola," in *Proceedings of the NATO/ASI Computer-Aided Analysis of Rigid and Flexible Mechanical Systems*, (Troia, Portugal), pp. 27–29, 1993.

[4] G. Korn, *Interactive Dynamic-System Simulation.* New York, NY: McGraw-Hill, 1989.

[5] *The Mathworks Company Website.* May 6 2003. <http://www.mathworks.com/>.

[6] R. Cubert, T. Goktekin, and P. Fishwick, "MOOSE: architecture of an object-oriented multimodeling simulation system," in *Proceedings of Enabling Technology for Simulation Science, Part of SPIE AeroSense '97 Conference*, (Orlando, FL), April 22-24 1997.

[7] G. Booch, *Object-Oriented Analysis and Design.* Boston, MA: Addison-Wesley, 1994.

[8] J. Banks, "The Future of Simulation Software: A Panel Discussion," in *Proceedings of the 1998 Winter Simulation Conference*, (Washington, DC), pp. 1681–1687, 1998.

[9] *WinOrbit Homepage.* May 5, 2003. <http://www.sat-net.com/winorbit>.

[10] *SaVi Sourceforge Website.* May 5, 2003. <http://sourceforge.net/projects/savi>.

[11] T. R. Henderson, *Networking over next-generation satellite systems.* PhD thesis, University of California at Berkeley, Fall 1999.

[12] L. Wood, G. Pavlou, and B. G. Evans, "Managing diversity with handover to provide classes of service in satellite constellation networks," in *Proceedings of the 19th AIAA International Communication Satellite Systems Conference (ICSSC '01)*, (Toulouse, France), Vol 3, Session 35, no. 194, April 2001.

[13] L. Wood, *Internetworking with satellite constellations.* PhD thesis, University of Surrey, June 2001.

[14] *ORSA Sourceforge Website.* May 5, 2003. <http://orsa.sourceforge.net>.

[15] J. Biesiadecki, A. Jain, and M. James, "Advanced simulation environment for autonomous spacecraft," in *International Symposium on Artificial Intelligence, Robotics and Automation in Space*, (Tokyo, Japan), July 1997.

[16] J. Biesiadecki, D. Henriquez, and A. Jain, "A Reusable, Real-Time Spacecraft Dynamics Simulator," in *6th Digital Avionics Systems Conference*, (Irvine, CA), October 1997.

[17] Princeton Satellite Systems, *MultiSatSim User's Guide v1.1.3.* Princeton Satellite Systems, 33 Witherspoon St. Princeton, NJ 08542, May 2002.

[18] Princeton Satellite Systems, *Spacecraft Dynamics and Control Using The Spacecraft Control Toolbox.* 33 Witherspoon St. Princeton, NJ 08542, 2nd ed., 2003.

[19] F. Bauer, J. Bristow, D. Folta, K. Hartman, D. Quinn, and J. How, "Satellite Formation Flying Using an Innovative Autonomous Control System (AUTOCON) Environment," in *Proceedings of the AIAA/AAS Astrodynamics Specialist Conference*, AIAA 97-3821, August 1997.

[20] J. Bristow and D. Folta, *AutoCon User's Guide v3.0.* NASA Goddard, Available at <http://icb.nasa.gov/swoy2002/> 1999.

[21] A.I. Solutions, *FreeFlyer User's Guide, v4.0.* 10001 Derekwood Lane, Suite 215, Lanham, MD 20706, March 1999.

[22] P. Ferguson, T. Yang, M. Tillerson, and J. How, "New Formation Flying Testbed for Analyzing Distributed Estimation and Control Architectures," in *AIAA Guidance, Navigation, and Control Conference and Exhibit*, (Monterey, CA), AIAA 02-4961, August 5-8 2002.

[23] J. Woodburn, K. Williams, and H. DeWitt, "The Customization of Satellite Tool Kit For Use On The NEAR Mission," in *Proceedings of the 1997 Space Flight Mechanics Conference*, (Huntsville, AL), AAS 97-176, American Astronautical Society (AAS), February 9-12 1997.

[24] "Hughes Using Moon For Orbit Change," *Aviation Week & Space Technologies*, May 4 1998.

[25] L. Kijewski, *Hughes and Analytical Graphics Make First Commercial Use of Moon.* 40 General Warren Blvd., Malvern, Pennsylvania 19355: Analytical Graphics Inc. press release, April 29 1998.

[26] J. Carrico, D. Conway, D. Ginn, Lanham-Seabrook, D. C. Folta, and K. V. Richon, "Operational Use of Swingby-an Interactive Trajectory Design and Maneuver Planning Tool - for Mission to the Moon and Beyond," in *Proceedings of the 1995 AAS/AIAA Astrodynamics Specialist*, (Halifax, Nova Scotia, Canada), AAS 95-323, 1995.

[27] J. P. Cohoon and J. W. Davidson, *C++ Program Design: An Introduction to Programming and Object-Oriented Design.* Boston, MA: McGraw-Hill, 2nd edition ed., 1999.

[28] B. Stroustrup, *The C++ Programming Language.* Boston, MA: Addison-Wesley, 3rd ed., 1997.

[29] P. C. Hughes, *Spacecraft Attitude Dynamics.* New York, NY: John Wiley & Sons, 1986.

[30] J. R. Wertz, ed., *Spacecraft Attitude Determination and Control.* Hingham, MA: Reidel Publishing, 1978.

[31] K. Makovec, "A Nonlinear Magnetic Controller for Three-Axis Stability of Nanosatellites," Master's thesis, Virginia Polytechnic Institute and State University, 2001.

[32] D. D. Rowlands, J. J. McCarthy, M. H. Torrence, and R. G. Williamson, "Multi-Rate Numerical integration of Satellite Orbits for Increased Computational Efficiency," *The Journal of the Astronautical Sciences*, vol. 43, pp. 89–100, Jan-Mar 1995.

[33] J. Woodburn and S. Tanygin, "Efficient Numerical Integration of Coupled Orbit and Attitude Trajectories Using An Encke Type Correction Algorithm," in *2001 Astrodynamics Specialist Conference*, (Quebec City, Canada), AAS 01-428, July 30 - August 2 2001.

[34] D. A. Vallado, *Fundamentals of Astrodynamics and Applications.* New York, NY: McGraw-Hill, 1997.

[35] R. R. Bate, D. D. Mueller, and J. E. White, *Fundamentals of Astrodynamics.* New York, NY: Dover Publications Inc., 1971.

[36] R. H. Battin, *An Introduction to the Mathematics and Methods of Astrodynamics, Revised Edition.* New York, NY: AIAA Education Series, June 1999.

[37] E. Fehlberg, "Classical Fifth-, Sixth-, Seventh-, and Eighth-Order Runge-Kutta Formulas with Stepsize Control," NASA Technical Report TR-R-287, 1968.

[38] G. Der, "Runge-Kutta Integration Methods for Trajectory Propagation Revisited," in *Proceedings of the AAS/AIAA Astrodynamics Specialist Conference*, (Halifax, Nova Scotia, Canada), AAS 95-420, 1995.

[39] R. L. Burden and J. D. Faires, *Numerical Analysis.* Pacific Grove, CA: Brooks/Cole, 7th ed., 2001.

[40] S. Meyers, *Effective C++.* Boston, MA: Addison-Wesley, 2nd edition ed., 1998.

[41] S. Meyers, *More Effective C++.* Reading, MA: Addison-Wesley, 1996.

[42] H. Sutter, *Exceptional C++.* Boston, MA: Addison-Wesley, 2000.

[43] H. Sutter, *More Exceptional C++.* Boston, MA: Addison-Wesley, 2002.

[44] A. J. Turner, *Open-SESSAME User's & Maintainer's Guide.* July 20, 2003. <http://spacecraft.sourceforge.net>.

[45] *Open-SESSAME Sourceforge Repository.* July 20, 2003 <http://sourceforge.net/projects/spacecraft>.

[46] Analytical Graphics, Inc., *STK Connect Manual.* 40 General Warren Blvd., Malvern, Pennsylvania 19355, 2003.

Appendix A

HokieSat Simulation Source Code

A.1 HokieSatSimulation.h

```
////////////////////////////////////////////////////////////////////////////////////
/*! \file HokieSatSimulation.h
*  \brief Include files and function prototypes for HokieSat attitude simulation exam
*  \author $Author: nilspace $
*  \version $Revision: 1.2 $
*  \date    $Date: 2003/06/12 23:05:55 $
*///////////////////////////////////////////////////////////////////////////////////


// Standard includes
#include "Matrix.h"
#include "Rotation.h"

// Dynamics includes
#include "CombinedNumericPropagator.h"
#include "RungeKuttaFehlbergIntegrator.h"

// Orbit Includes
#include "Orbit.h"
#include "OrbitState.h"
#include "orbitmodels/TwoBodyDynamics.h"
#include "orbitstaterep/PositionVelocity.h"
```

```
#include "orbitframes/OrbitFrameIJK.h"

// Attitude includes
#include "Attitude.h"
#include "AttitudeState.h"
#include "AttitudeModels/QuaternionAngVelDynamics.h"

// Environment Includes
#include "CentralBody/EarthCentralBody.h"
#include "CentralBody/Models/TiltedDipoleMagneticModel.h"
#include "Disturbances/GravityFunctions.h"
#include "Disturbances/SimpleAerodynamicDisturbances.h"

// Utility Includes
#include "Plot.h"
#include "MathUtils.h"

using namespace O_SESSAME;

/** @brief Sets up a combined numeric propagator, RK4(5) integrator and tolerances. *
NumericPropagator* SetupPropagator();
/** @brief Creates an Earth environment with point-mass,
                 * two-body gravity, gravity-gradient torque,
                 * and a tilted-dipole magnetic field model. */
Environment* SetupEnvironment(Attitude* pSatAttitude);

/** @brief Creates an initial orbit read in from a file. */
Orbit* SetupOrbit();

/** @brief Creates an initial attitude read in from a file. */
Attitude* SetupAttitude();

/** @brief HokieSat magnetic controller algorithm prototype.
    */
Vector ControlTorques(Matrix CurrentAttState,
                                  Matrix DesAttState,
                                  double epoch,
                                  double count);
```

A.2 HokieSatSimulation.cpp

```
////////////////////////////////////////////////////////////////////////////////////
/*! \file HokieSatSimulation.cpp
*  \brief Demonstrates the use of Open-SESSAME for simulating
*       HokieSat.
*  \author $Author: nilspace $
*  \version $Revision: 1.4 $
*  \date    $Date: 2003/06/12 23:05:55 $
*///////////////////////////////////////////////////////////////////////////////////
/*!
*/
////////////////////////////////////////////////////////////////////////////////////

#include "HokieSatSimulation.h"

/** @brief Main operating function for HokieSat simulation.
* @author Andrew Turner
*
* Breaks down all object initializations into seperate functions.
*/
int main()
{
    Orbit* pHokiesatOrbit = SetupOrbit();
    Attitude* pHokiesatAttitude = SetupAttitude();

    // Setup Propagator
        NumericPropagator* pHokiesatPropagator = SetupPropagator();
        pHokiesatOrbit->SetPropagator(pHokiesatPropagator);
        pHokiesatAttitude->SetPropagator(pHokiesatPropagator);

    // Setup external environment
        Environment* pEarthEnv = SetupEnvironment(pHokiesatAttitude);
        pHokiesatOrbit->SetEnvironment(pEarthEnv);
        pHokiesatAttitude->SetEnvironment(pEarthEnv);

    // Integration Times
        double propTime = 20; // mins
```

```
            cout << "Propagation time (mins): " << flush;
            cin >> propTime;
        double propStep = 60; // s
            cout << "Propagation step (secs): " << flush;
            cin >> propStep;
        vector<ssfTime> integrationTimes;
        ssfTime begin(0);
        ssfTime end(begin + propStep);
        integrationTimes.push_back(begin);
        integrationTimes.push_back(end);

    // Output the current state properties
        cout << "PropTime = " << begin.GetSeconds() << " s -> "
                << end.GetSeconds()
                << " s" << endl;
        cout << "Orbit State: " << ~pHokiesatOrbit->
                 GetStateObject().GetStateRepresentation()->
                 GetPositionVelocity();
        cout << "Attitude State: " << ~pHokiesatAttitude->
                GetStateObject().GetState() << endl;

    // Integrate over the desired time interval
        tick();
        pHokiesatPropagator->Propagate(integrationTimes, pHokiesatOrbit->
                 GetStateObject().GetStateRepresentation()->
                 GetPositionVelocity(), pHokiesatAttitude->
                 GetStateObject().GetState());

        for (int ii = 0; ii < propTime*60/propStep ; ++ii)
        {
            // Integrate again
            integrationTimes[0] = integrationTimes[1];
            integrationTimes[1] = integrationTimes[0] + propStep;
            //cout << integrationTimes[0] << " -> " <<
                    integrationTimes[1] << endl;
            pHokiesatPropagator->Propagate(integrationTimes,
                 pHokiesatOrbit->
                 GetStateObject().GetStateRepresentation()->
                GetPositionVelocity(), pHokiesatAttitude->
                GetStateObject().GetState());
        }
        cout << endl;
        ssfSeconds calcTime = tock();
        cout << "finished propagating " << propTime*60
                << " sim-seconds in " << calcTime
                << " real-seconds." << endl;
    // Plot the state history
        Matrix orbitHistory =
                pHokiesatOrbit->GetHistoryObject().GetHistory();
        Matrix orbitPlotting =
                orbitHistory(_,_(MatrixIndexBase+1,MatrixIndexBase+3));
        Matrix attitudeHistory =
                pHokiesatAttitude->GetHistoryObject().GetHistory();
        Matrix attitudePlotting =
                attitudeHistory(_,_(MatrixIndexBase,MatrixIndexBase+4));

        Plot3D(orbitPlotting);
        Plot2D(attitudePlotting);

    // Store the output to file
        ofstream ofile;
        ofile.open("OrbitHistory.dat");
        ofile << pHokiesatOrbit->GetHistoryObject().GetHistory();
        ofile.close();

        ofile.open("AttitudeHistory.dat");
        ofile << pHokiesatAttitude->GetHistoryObject().GetHistory();
        ofile.close();

    return 0;

}

// ************************
// ****** ENVIRONMENT ******
// ************************
```

```
Environment* SetupEnvironment(Attitude* pSatAttitude)
{
    // ENVIRONMENT TESTING
    Environment* pEarthEnv = new Environment;
    EarthCentralBody *pCBEarth = new EarthCentralBody;
    pEarthEnv->SetCentralBody(pCBEarth);

    // Add Gravity force function
        EnvFunction TwoBodyGravity(&GravityForceFunction);
        double *mu = new double(pCBEarth->GetGravitationalParameter());
        TwoBodyGravity.AddParameter(reinterpret_cast<void*>(mu), 1);
        pEarthEnv->AddForceFunction(TwoBodyGravity);

    cout << "Add Drag? (y or n): " << flush;
    char answer;
    cin >> answer;
    if(answer == 'y')
    {
        // Add Drag Force Function
        EnvFunction DragForce(&SimpleAerodynamicDragForce);
        double *BC = new double(2);
        DragForce.AddParameter(reinterpret_cast<void*>(BC), 1);
        double *rho = new double(1.13 * pow(static_cast<double>(10),
                                  static_cast<double>(-12))); // kg/m^3
        DragForce.AddParameter(reinterpret_cast<void*>(rho), 2);
        pEarthEnv->AddForceFunction(DragForce);
    }
    // Add Gravity torque function

        EnvFunction GGTorque(&GravityGradientTorque);
        Matrix *MOI = new Matrix(pSatAttitude->GetParameters()(_(1,3),_));
        GGTorque.AddParameter(reinterpret_cast<void*>(MOI), 1);
        GGTorque.AddParameter(reinterpret_cast<void*>(mu), 2);
        pEarthEnv->AddTorqueFunction(GGTorque);

    // Assign Magnetic Model
        pCBEarth->SetMagneticModel(new TiltedDipoleMagneticModel);
    return pEarthEnv;
```

```
}

// ************************
// ****** PROPAGATOR *******
// ************************
NumericPropagator* SetupPropagator()
{
    CombinedNumericPropagator* pSatProp = new CombinedNumericPropagator;

    // Create & setup the integator
        // Setup an integrator and any special parameters
    RungeKuttaFehlbergIntegrator* orbitIntegrator =
            new RungeKuttaFehlbergIntegrator;
    RungeKuttaFehlbergIntegrator* attitudeIntegrator =
            new RungeKuttaFehlbergIntegrator;

    orbitIntegrator->SetTolerance(pow(10.,-7.));
    orbitIntegrator->SetStepSizes(0.01, 300);
    pSatProp->SetOrbitIntegrator(orbitIntegrator);
    attitudeIntegrator->SetTolerance(pow(10.,-7.));
    attitudeIntegrator->SetStepSizes(0.01, 5);
    pSatProp->SetAttitudeIntegrator(attitudeIntegrator);

    return pSatProp;
}

// ************************
// ********* ORBIT *********
// ************************
Orbit* SetupOrbit()
{
    Orbit* pSatOrbit = new Orbit;

    // Create & initialize the orbit
    OrbitState SatOrbitState;
    SatOrbitState.SetStateRepresentation(new PositionVelocity);
    SatOrbitState.SetOrbitFrame(new OrbitFrameIJK);
    Vector initPV(6);
```

```
        // Space station
        initPV(VectorIndexBase+0) = -5776.6; // km
        initPV(VectorIndexBase+1) = -157; // km
        initPV(VectorIndexBase+2) = 3496.9; // km
        initPV(VectorIndexBase+3) = -2.595;  // km/s
        initPV(VectorIndexBase+4) = -5.651;  // km/s
        initPV(VectorIndexBase+5) = -4.513; // km/s
    SatOrbitState.SetState(initPV);
    pSatOrbit->SetStateObject(SatOrbitState);

    pSatOrbit->SetDynamicsEq(&TwoBodyDynamics);
    pSatOrbit->SetStateConversion(&PositionVelocityConvFunc);

    return pSatOrbit;
}


// ************************
// ******* ATTITUDE ********
// ************************
Attitude* SetupAttitude()
{
    Attitude* pSatAttitude = new Attitude;

    AttitudeState SatAttState;
    SatAttState.SetRotation(Rotation(Quaternion(0,0,0,1)));
    Vector initAngVelVector(3);
        initAngVelVector(1) = 0;
    SatAttState.SetAngularVelocity(initAngVelVector);

    pSatAttitude->SetStateObject(SatAttState);
    pSatAttitude->SetDynamicsEq(&AttituteDynamics_QuaternionAngVel);
    pSatAttitude->SetStateConversion(&QuaternionAngVelConvFunc);

    // Create the matrix of parameters needed for the RHS
    Matrix MOI(3,3);
        MOI(1,1) =  0.4084; MOI(1,2) =  0.0046; MOI(1,3) = 0.0;
        MOI(2,1) =  0.0046; MOI(2,2) =  0.3802; MOI(2,3) = 0.0;
        MOI(3,1) =  0.0;    MOI(3,2) =  0.0;    MOI(3,3) = 0.4530;

    MOI = eye(3);
        MOI(1,1) = 2; MOI(2,2) = 2; MOI(3,3) = 10;
    Matrix params(6,3);
        params(_(1,3),_) = MOI;
        params(_(4,6),_) = MOI.inverse();

    pSatAttitude->SetParameters(params);

    return pSatAttitude;

}

// Do not change the comments below - they will be added automatically by CVS
/*****************************************************
* $Log: HokieSatSimulation.cpp,v $
* Revision 1.4  2003/06/12 23:05:55  nilspace
* Works.
*
* Revision 1.3  2003/06/12 21:01:48  nilspace
* Fixed GG torque MOI parameter.
*
* Revision 1.2  2003/06/12 20:48:10  nilspace
* Asks user for times.
*
* Revision 1.1  2003/06/12 18:06:06  nilspace
* Initial submission.
*
* Revision 1.4  2003/05/27 17:47:13  nilspace
* Updated example to have seperate orbit & attitude integrators.
*
* Revision 1.3  2003/05/20 19:24:43  nilspace
* Updated.
*
* Revision 1.2  2003/05/13 18:57:32  nilspace
* Clened up to work with new Propagators.
*
```

```
* Revision 1.1  2003/05/01 02:42:47  nilspace
* New propagation test file.
*
*
************************************************************/
```

Vita

Andrew Turner was born and raised in Montgomery County, Pennsylvania. He attended the local public schools and graduated from Methacton High School in 1996. His college education began at the University of Virginia in Charlottesville, Virginia. While at UVA, Andrew was the Vice President of Engineering for the UVA Solar Airship Program, a Raven Society Fellow in 1999, and a Research Symposium Finalist for his undergraduate thesis "Semi-Autonomous Control of the UVA Solar Airship *Aztec*," which he presented at the 3^{rd} Annual AA/AIAA Airship Convention held in Friedrichshafen, Germany in July 2000. Andrew graduated in from UVA in 2000 with a Bachelor's of Science degree in Aerospace Engineering and a minor in Computer Science.

Andrew then began his post-graduate education at the Aerospace & Ocean Enginering Department at Virginia Polytechnic Institute and State University (Virginia Tech) in 2000 and graduated with his Master's degree in July, 2003. During his time as a graduate student at Virginia Tech, Andrew was a member and Systems Lead for the Virginia Tech Nanosatellite, HokieSat, and was a Virginia Space Grant Fellow in 2001 for his research in neural network control algorithms for spacecraft attitude control. Andrew also worked for 6-months at Astrium, an international space systems company based in Europe, at their Friedrichshafen, Germany office. While there, he developed simulations and models of sensor and actuator components for the GOCE spacecraft.

Andrew is now employed at Realtime Technologies, Inc. in Royal Oak, Michigan. He is developing vehicle dynamics simulation software and hardware as well as consulting for a variety of companies and the US Army.