An Open-Source, Extensible Spacecraft Simulation 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

Dr. Christopher Hall, Committee Chair
Dr. Fred Lutze, Committee Member
Dr. Craig Woolsey, Committee Member

June 3, 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 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 int 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, its underlying 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 constantly concerned with creating accurate and understandable models of the satellite being analyzed. These simulations help to verify principles of motion, and 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 every new spacecraft. The alternatives are either to adapt the researcher's own previous simulation, or to use someone else'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 any 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 high-cost 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 operating 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 analyse 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 for a 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 output 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. An example may be analyzing the orbit of a spacecraft about a central body over long time periods to evaluate its ground track, calculating access to ground stations, and determining 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 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 specifically 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, what to be concerned with, and how to best

implement components so they can be resuable between projects and 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.* These projects include both hardware and software simulation techniques that are used in conjuction to better understand the interplay of satellite dynamics with novel control and sensing strategies. As a result of the various 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 the capacity to configure the simulation to their unique specifications. Furthermore, they are able to interface the simulation software with other analysis packages they may require 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 analsysis and modeling, while leading them into the design aspects of bringing these principles together into a usable software framework. An introduction is given to introduce the reader to object-oriented design strategies and their application to the framework layout. Most importantly, the reader will be introduced to and learn the internal operation of the framework, the interation between components, and ways to use the framework for research projects. A treatment of the verification and validation is presented to reassure the reader as to the accuracy of the current simulation framework and how to maintain this accuracy while implementing new algorithms. Finally, it is the goal of this thesis 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-

*http://www.aoe.vt.edu/research/groups/sssl/

oriented programming. Chapters 3 and 4 give an in-depth discussion 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 make sure the accuracy is maintained. Chapter 8 presents several simulation examples, and suggests features that could be implemented by users of Open-Sessame. Finally, Chapter 9 covers the conclusions reached in developing this framework as well as the future of research in the area of modeling and simulation.

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 the 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 community 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 software was developed that the average engineer had access to for analysis and modeling.

In 1979, Cellier developed a numerically sound methodology for simulating hybrid models using digital computers [8]. Following this work, there were various efforts to implement simulation specific programming languages such as Dymola[12, 28], Desire [23], and Mathwork's Simulink [**?**]. In the case of the first two languages, they 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 [10] developed MOOSE (Multimodeling Object-Oriented Simulation Environment) which focused on both developing a framework for modeling multiple body objects, but more importantly on 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 big 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 to 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 [5].

2.2 Prior Work in Spacecraft Simulation

There are numerous implementations of spacecraft simulation packages, most of which 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 these 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. It can graphically display

satellite positions in real-time and simulation modes as well as generate tracking and ephemerides information for a number of Earth-based satellites. It appears to have stopped development in 1998 and is not open source [3].

SaVi (short for Satellite Visualization) is an open-source satellite constellation visualization that is being hosted on the Sourceforge repository*[2], 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 [16, 24, 40].

Another open-source toolkit is *ORSA*: Orbit Reconstruction, Simulation and Analysis.[1]. It is still under development by students at Padova University in Italy and does not currently have many of its features implemented. Its main goal is the simulation and analysis of celestial bodies, but because it is open-source the software could be used as a basis for a broader space simulation package.

NASA JPL engineers have been developing several spacecraft simulation tools that form the Autonomy Testbed Environment (ATBE) which is built on LIBSIM and DARTS / DSHELL [6]. 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 (DARTS Shell). Furthermore, DSHELL is portable from desktop systems to hardware-in-the-loop environments [18].

DARTS/DSHELL has been used for several NASA JPL projects such as Cassini, Galileo, Mars Pathfinder, and Stardust [6]. LIBSIM was used by the New Millenium Project's Deep Space 1. The software package available free-of-charge to qualifying academic institutions. However, a copy of the code was not made available for testing and evaluating.

Princeton Satellite Systems (PSS) has developed MultiSatSim, which can simulate up to 8 satellites as well as control them from a remote computer [30]. Unlike most other satellite simulation tools, MSS is not restricted to modeling systems about the Earth. Therefore, while the gravity model and control can be customized, the simulation only models a rigid body with quaternions. Furthermore, MSS is not open-source, and binaries are only available for Apple brand computers.

*http://sourceforge.net/projects/savi

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 [31]. 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*[†] developed by AVM Dynamics is a collection of software tools for the design, visualization and analysis of satellite constellations. It is primarily used for communication constellations, and also includes tools for analyzing ground-space operations. SC Modeler is closed-source and has a high purchase cost.

FreeFlyer is another Windows based application [34]

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. [‡]

AutoCon was developed by NASA GSFC and AI Solutions as a satellite autonomous maneuver planning software. There two principle parts, AutoCon-F and AutoCon-G, are used for Flight and Ground simulation respectively. This sharing of parts enables use of the same code on 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 [14, 21]

Another formation simulation software packaged currently in development at NASA GSFC is the *Formation Flying Testbed*. 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 [29]. *Satellite ToolKit* (STK) is a commercial package developed by Analytical Graphics Inc. (AGI)[§] consists of 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[19], Sirius Satellite Radio, Loral's GlobalStar, and Hughes Asi-

[†]http://www.avmdynamics.com/index1.htm
[‡]http://www.saic.com/cover-archive/space/satcos.html
[§]http://www.stk.com

aSat3 satellite rescue [4, 22], just to name a few. STK a Graphical User Interface (GUI) through which users perform the simulation tasks, as well as means for communicating via network protocols between remote machines.

A big shortcoming of STK is its high cost (up to $10,000 per module) and closed source. While the basic STK program is free of charge, the modules are relatively 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 not part of an established research group or engineers that cannot afford the cost. Furthermore, the program is closed source in order to produce the commercial product and also maintain its accuracy and speed. This closed-source prevents student and engineers from understanding STK's internal operation and using the developed tools for specific tailored applications. STK definitely 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 Goddard Space Flight Center (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 [**?**].

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.[¶]

Table **??** 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.

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.

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

Table 2.1: Summary of spacecraft simulation software libraries and applications

Package	Manufacturer	Benefits	Negatives	Cost	Websi
AutoCon	NASA Goddard	Heritage	Not maintained, not readily available	Cost	–
DSHELL	NASA JPL	Free to academic institutions	Not readily available	Academic: "Free"	dshell.jpl.n
Formation Flying Testbed	NASA Goddard	Benefits	Negatives	Cost	??
FreeFlyer	a.i. solutions	Scriptable, good user interface, heritage, 2D/3D visualization	Expensive, closed-source, limited integration with existing software	Cost	www.ai-solut
MultiSatSim	Princeton Satellite Systems	Good graphics, easy to use interface, scriptable	Apple hardware only, limited to 8 satellites, limited expandability	$0	www.psatel
ORSA	Open-Source	Open-Source, multiple platforms, active development	Not complete, limited functionality, orbits only	$0 (open-source)	orsa.sourcef
Open-SESSAME	Open-Source	Open-source, multiple platforms, extensible, well documented, active development, orbit and attitude (variable coupling), seperate libraries	No graphical user interface, requires knowledge of C++ programming	$0 (open-source)	spacecraft.sour
SATCOS	SAIC	Benefits	Orbit and constellations only	Cost	www.saic.cor
SaVi	Open-Source	Good user interface, open-source, in development	Only models orbits, made for constellations, single developer	$0 (open-source)	savi.sourcef
SC Mod-	AVM Dy-	Benefits	Negatives	Cost	www.avmdyna

These objects can be thought of in much the same way we would think of a real world object. A car is an object that has data associated with it: number of wheels and doors, mileage, color, and size, as well as operations that can be done with it: 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 illustrade 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 doesn't need to know about this internal change, since the user still calls *GetSpeedMPH()* and gets 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. For a full introduction and discussion of Object-Oriented programming and other design paradigms refer to Cohoon and Davidson[9] or Stroustrup [35], the developer of the C++ programming language.

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 with respect to an Earth-fixed frame. In order to fully describe an attitude, a set of reference frames are defined as well as the methods used for representing the orientation of these frames with respect to one another. The kinematics and dynamics of these frame rotations are 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 the dynamics of the attitude 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 explicity specified, a preferred direction for a second vector (preferred in that a desired direction is determined, but orthogonality is a priority) is chosen, and the third direction is determined by 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 frames are some prevalent examples.

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). 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 $\hat{y}^i$ 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.

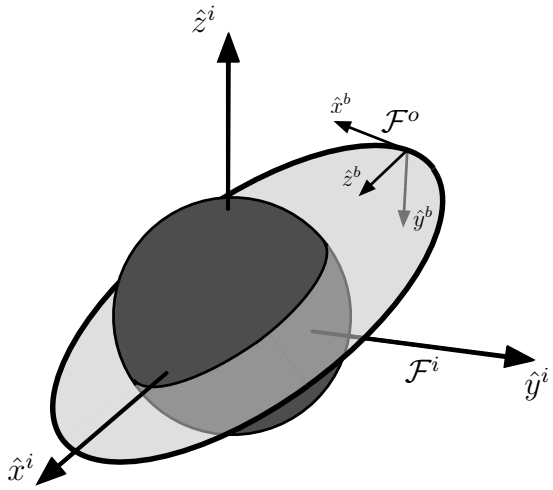

Figure 3.1: *Illustration of the orbital, $\mathcal{F}^o$, and inertial, $\mathcal{F}^i$, reference frames in an orbit about a central body.*

3.1.2 Orbital Frame

The Orbital Frame is a non-inertial, rotating, frame that moves with a body in orbit and describes the orientation of the local orbit position. As illustrated in Figure 3.1, the origin is fixed at the spacecraft's mass center with the $\hat{\mathbf{o}}_\mathbf{3}$ axis (corresponding to $\hat{z}^o$) in the direction from the spacecraft to the Earth (nadir direction). The $\hat{\mathbf{o}}_\mathbf{2}$ axis ($\hat{y}^o$) is the direction opposite to the orbit normal, and $\hat{\mathbf{o}}_\mathbf{1}$ ($\hat{z}^o$) completes the orthonormal triad to $\hat{\mathbf{o}}_\mathbf{2}$ and $\hat{\mathbf{o}}_\mathbf{3}$. 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.

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 the spacecraft dynamics to describe the system in the principal axes frame. This frame is a specific body-fixed reference frame in which the axis system has its origin at the mass center 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. In order to fully define an orientation with no ambiguity at least 4 parameters must be defined. There are several common ways of defining a rotation: Euler Axis, Euler Angles, Direction Cosine Matrix, Modified Rodriguez Parameters and Quaternions.

3.2.1 Euler Axis & Angle

The simplest description relating 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 also be represented as the principal rotation vector:

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

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

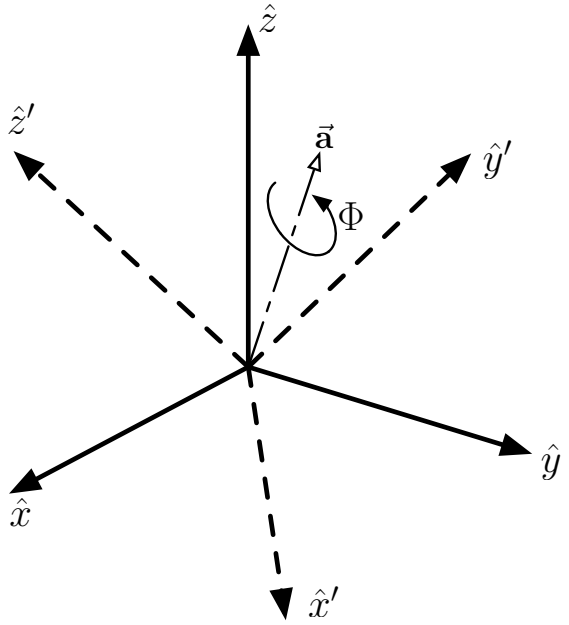

Figure 3.2: *Rotation about an Euler axis,* $\hat{\mathbf{e}}$*, of the euler angle,* Φ *that transforms* $\mathcal{F}$ *to* $\mathcal{F}'$

3.2.2 Euler Angles

A transformation can also be described by a set of angular rotations about each of the axes as the frame is rotated. For example, to describe a rotation from the Earth Centered Inertial frame to the orbital 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 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 tranformations, 3 rotations are sufficient which results in 12 possible successive rotation combinations. However, specific transformations may require more or less. 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 that prevent error free rotations to any transformation. These singularities make the representation inconvenient for simulation, but useful for visualization.

3.2.3 Direction Cosine Matrix

A simple way to describe and represent an Euler Angle sequence 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 $\hat{\mathbf{v}}$ vectors 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 components of 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.

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

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

To create the rotation matrix using euler angles, it is simplest 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:

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

3.2.4 Quaternions

The 4-element quaternion set, $\bar{\mathbf{q}}$ contains 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 has the useful characteristic that it should have unit length. 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 as follows:

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

Like the quaternions, the MRP is not a unique solution 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 is a listing of the 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-symmetic* 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 & 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:

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

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

DCM to Quaternion

The conversion from Direction Cosine Matrix to quaternion is:

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

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

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:

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

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

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

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:

$$\dot{\bar{\mathbf{q}}} = \mathbf{Q}\left(\bar{\mathbf{q}}\right)\omega \tag{3.28}$$

$$= \frac{1}{2}\begin{bmatrix} \mathbf{q}^{\times} + q_4\mathbf{1} \\ -\mathbf{q}^T \end{bmatrix}\omega \tag{3.29}$$

Modified Rodriguez Parameters Kinematic Equations of Motion

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

$$\dot{\sigma} = \mathbf{S}^{-1}(\sigma)\omega \tag{3.30}$$

where

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

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 **??** lays out the 12 possible kinematic equations, one for each type of Euler Angle sequence.

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

3.3.3 Dynamic Equations of Motion

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

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

where $\vec{\mathbf{h}}$ is the angular momentum vector and $\vec{\mathbf{g}}$ is the applied torque. The differential equations for the angular velocity in the body frame are based on 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.

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:*Adapted from Wertz [39]

We must find a relation between the body-inertial, ω^{bi}, and body-orbital, ω^{bo}, angular velocities to propagate the dynamics in the rotating reference frame:

$$\omega^{bi} = \omega^{bo} + \omega^{oi} = \omega^{bo} - \omega_c \mathbf{o_2} \tag{3.35}$$

therefore,

$$\dot{\omega}^{bi} = \dot{\omega}^{bo} - \omega_c \dot{\mathbf{o}}_\mathbf{2} \tag{3.36}$$

$$= \dot{\omega}^{bo} + \omega_c {\omega^{bo}}^{\times} \mathbf{o_2} \tag{3.37}$$

where ω_c is the orbital angular velocity, and $\mathbf{o_2}$ is the 2nd 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:

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

3.3.4 Flexible Body Dynamics

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 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 effect on the attitude dynamics.

3.4.1 Aerodynamics

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 most likely with much 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 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 400km the aerodynamic torque is the dominant environmental disturbance torque [39].

3.4.2 Magnetic

Spacecraft busses 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. $\vec{\mathbf{B}}$ is the central body's magnetic flux density at the spacecraft's location [39].

The magnetic disturbance can also be very useful for applying active control using magnetic torquers [25].

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. 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:

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

and $\mathbf{o_2}$ and $\mathbf{o_3}$ are the 2^{nd} and 3^{rd} columns, respectively, of the body-orbital rotation (Equation **??**).

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 [39].

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

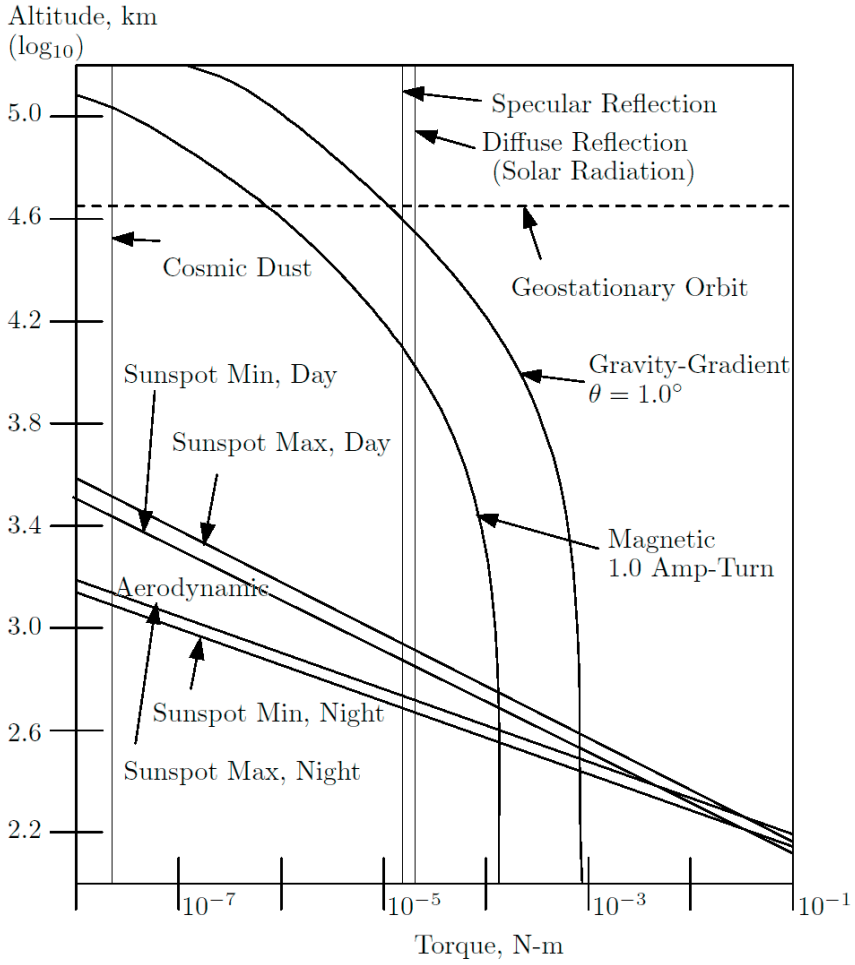

Figure 3.3: *Relative effect of various example environmental disturbance torques. (adapted from [17])*

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

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. However, 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 [**?**].

To assist in verifying the modeling solution, the closed-form approximations of the equations of motion can be used. The simple case is that of an axisymmetric body, i.e. two of the principal moments of inertia are equal, say $I_T = I_1 = I_2$, and no externally applied torques. This transverse moment of inertia, I_T, simplifies Euler's Equations (Equation **??**) to:

$$\dot{\omega}_1 = \frac{I_T - I_3}{I_T}\omega_2\omega_3 \tag{3.41}$$

$$\dot{\omega}_2 = \frac{I_3 - I_T}{I_T}\omega_1\omega_3 \tag{3.42}$$

$$\dot{\omega}_3 = 0 \tag{3.43}$$

where 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:

$$\ddot{\omega}_1 = \left(\frac{I_T - I_3}{I_T}\right)^2 \omega_1\omega_3^2 \tag{3.44}$$

$$\Rightarrow \omega_1 = \omega_T \cos\omega_p(t - t_1) \tag{3.45}$$

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:

$$\omega_p = \left(1 - \frac{I_3}{I_T}\right)\omega_3 \tag{3.46}$$

These equations can be summarized as follows:

$$\omega_1 = \omega_{01}\cos\omega_p t + \omega_{02}\sin\omega_p t \tag{3.47}$$

$$\omega_2 = \omega_{02}\cos\omega_p t - \omega_{01}\sin\omega_p t \tag{3.48}$$

$$\omega_3 = \omega_{03} \tag{3.49}$$

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

This closed-form solution can now 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, $\mathbf{g} = \mathbf{g}(t)$ [17]:

$$\omega_1 = \omega_{01}\cos\omega_p t + \omega_{02}\sin\omega_p t + \frac{1}{I_T}\int_0^t [g_1(\tau)\cos(\omega_p(t-\tau)) + g_2(\tau)\sin(\omega_p(t-\tau))]d\tau \quad (3.50)$$

$$\omega_2 = -\omega_{01}\sin\omega_p t + \omega_{02}\cos\omega_p t + \frac{1}{I_T}\int_0^t [-g_1(\tau)\sin(\omega_p(t-\tau)) + g_2(\tau)\cos(\omega_p(t-\tau))]d\tau \quad (3.51)$$

$$\omega_3 = \omega_{03} + \frac{1}{I_a}\int_0^t g_3(\tau)d\tau \quad (3.52)$$

More in-depth derivations can produce the closed-form solutions of the non-axisymmetric case using Jacobi elliptic functions [39].

3.6.2 Coupling with Orbit Maneuvers

Most of the torque disturbances become dependent on the spacecraft position as the fidelity is increased. For instance, it is required to know the position to calculate the local magnetic field, or 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 [20].

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 the ability to add refined models as required.

Chapter 4

Orbit Dynamics

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

4.1 Time

Normally, time is considered a trivial issue, measured with a clock, maybe even 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*, or instant in time, in mean solar time. There are, however, many problems associated with the measurement of time based on the 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 is measured from a nominal longitude line, and is the time required for an observer on the 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 the observers longitude. It is important to note that the definition of the hour angle is *left-handed*, so 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 inclination 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 teh 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, which uses a vernal equinox with only secular motions, and is based upon Greenwich Sidereal Time (GST), which is discussed in the next Section.

Sidereal Time

Sidereal Time is similar to solar time, which is the time it takes for the observer to revolve once and observe a celestial object at the same location. However, sidereal time uses a defined set of stars that are outside our solar system at a much greater distance, and therefore have less change over the course of a year (unlike our Sun).

To formally define sidereal time, we need to define the set of axes that the observations are taken from. The rotation axis is through the north pole of the Earth (or 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 (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 the Earth (or 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 which 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 will not be discussed further here.

Coordinated Universal Time

Atomic Time is the measurement of time based on the specific quantum transitions of electrons in a cesium-133 atom. The transition causes the emission of photons of a known frequency that can be counted. AT forms the basis of 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.*

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 [38].

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

$$\begin{aligned} \text{Julian Date} \;=\; & 367(year) - floor\left(\frac{7\left[year + \text{floor}\left(\frac{month+9}{12}\right)\right]}{4}\right) && (4.7)\\ & + \text{floor}\left(\frac{275month}{9}\right) + day + 1,721,013.5 && (4.8)\\ & + \frac{\left(\frac{seconds}{60} + minute\right)}{60} + hour}{24} && (4.9) \end{aligned}$$

where the *floor* function is truncation ($\text{floor}\,(4.587) = 4$) 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}, JD_{TDT}, JD_{TAI}, etc.

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.

*http://tycho.usno.navy.mil/gps_datafiles.html

Therefore, a set of Dynamics Times was developed based on more stable references.

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

$$UTC = UT1 - \delta UT1 \tag{4.10}$$
$$TAI = UTC + \delta AT \tag{4.11}$$
$$TDT = TAI + 32.184s \tag{4.12}$$

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:

$$T_{TDB} = T_{TDT} + 0.001\,658^S \sin M_\oplus + 0.000\,013\,85 \sin 2M_\oplus + lunar/planetaryterms + dailyterms \tag{4.13}$$
$$M_\oplus \approx 357.527\,723\,3^\circ + 35,999.050\,34 T_{TDB} \tag{4.14}$$

Furthermore, Equation **??** requires iterations to solve, while the following appoximate solution[15] neglects lunar and planetary terms (order 1×10^{-5} s) and does not require iteration:

$$T_{TDB} \approx T_{TDT} + 0.001\,658\,s \sin M_\oplus + 0.000\,013\,85 \sin 2M_\oplus \tag{4.15}$$

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:

$$EQ_{time} = -1.914\,666\,471^\circ \sin(M_\odot) - 0.019\,994\,643 \sin(2M_\odot) \tag{4.16}$$
$$+2.466 \sin(\lambda_{ecliptic}) - 0.0053 \sin(4\lambda_{ecliptic}) \tag{4.17}$$

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:

$$1 \text{ solar day} = 1.002\,737\,909\,350\,795 \text{ sidereal day} \tag{4.18}$$

$$1 \text{ sidereal day} = 0.997\,269\,566\,329\,084 \text{ solar day} \tag{4.19}$$

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

$$1 \text{ solar day} = 1.002\,737\,909\,350\,795 + 5.9006 \times 10^{-11} T_{UT1} \tag{4.20}$$

$$-5.9 \times 10^{-15} T_{UT1}^2 \text{ sidereal day} \tag{4.21}$$

where T_{UT1} is the number of Julian centures (UT1) since the J2000 epoch [38].

Julian Date and Universal Time

Because several time conversions measure the Julian Date from a certain epoch, it is necessary to derive a relation between the Julian Date and a particular type of time:

$$T_{xxx} = \frac{JD_{xxx} - 2,451,545.0}{36,525} \tag{4.22}$$

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 to be $1DU_{\oplus}$ and the time unit, $1TU_{\oplus}$ such that the velocity of the satellite in the reference frame is $1\frac{DU_{\oplus}}{TU_{\oplus}}$. The gravitational parameter is then $\mu = 1\frac{DU_{\oplus}^3}{TU_{\oplus}^2}$ which does not have to change with increasingly better measurement techniques.

TABLE OF OTHER SYMBOLS

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

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

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	IJK	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-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:*Adapted from Vallado [38, pg. 46]

Position and Velocity

The most common, and arguably the most useful, representation of the orbital state is by using the 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 values in the specified reference frame which specifies the origin and axes. As mentioned in Section 3.2.3, to transform between one reference frame and another, we must define rotation matrices that can be used to calculate the new components, though the vector itself remains invariant under coordinate transformations.

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

$$\vec{\mathbf{r}}^{inertial} = \mathbf{R}^{ir}\vec{\mathbf{r}}^{rot} \tag{4.23}$$

$$\vec{\mathbf{v}}^{inertial} = \vec{\mathbf{v}}^{rot} + \vec{\omega}^{r/i} \times \vec{\mathbf{r}}^{rot} + \vec{\mathbf{v}}_{origin} \tag{4.24}$$

$$\begin{aligned} \vec{\mathbf{a}}^{inertial} = {} & \vec{\mathbf{a}}^{rot} + \dot{\vec{\omega}}^{\mathbf{r/i}} \times \tilde{\mathbf{r}}^{\mathbf{rot}} + \tilde{\omega}^{\mathbf{r/i}} \times \left(\tilde{\omega}^{\mathbf{r/i}} \times \tilde{\mathbf{r}}^{\mathbf{rot}}\right) \\ & + 2\vec{\omega}^{r/i} \times \vec{\mathbf{v}}^{rot} + \vec{\mathbf{a}}_{origin} \end{aligned} \tag{4.25}$$

where $\vec{\omega}^{r/i}$ 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:

$$\mathbf{r}^{\mathbf{b}} = \mathbf{R}^{ba}\mathbf{r}^{\mathbf{a}} \tag{4.26}$$

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

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	$R_2(\pi)$

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 isϕ_{gd} and f_r is a retrograde factor, which is+1 when $0^\circ \leq i \leq 90^\circ$ or -1 when $90^\circ < i \leq 180^\circ$. 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 true anomaly (ν). These values are shown in figure 4.1

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 nodes. 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:

$$a = \left(\frac{2}{r} - \frac{v^2}{\mu}\right)^{-1} = \frac{r_a - r_p}{2}. \tag{4.27}$$

Conversely, the radii of apoapsis and periapsis can be determined from the semimajor

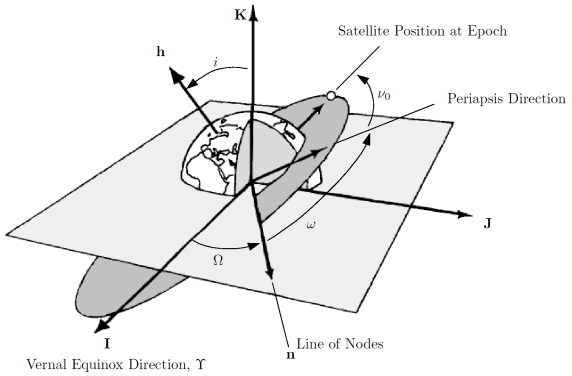

Figure 4.1:
Keplerian orbital elements in the Earth-Center Earth-Fixed frame

axis, and eccentricy, which is discussed below:

$$r_p = \frac{a\left(1-e^2\right)}{1+e} = a\left(1-e\right) \tag{4.28}$$

$$r_a = \frac{a\left(1-e^2\right)}{1-e} = a\left(1+e\right) \tag{4.29}$$

where r is the spacecraft distance from the central body's center of mass, v is the velocity, μ is the gravitational parameter, and r_a, r_p are the radii of apoapsis and periapsis respectively.

The eccentricity, e, defines the shape of the ellipse and is equal to the proportion of the distance from the center of the orbit to a focus to the semimajor axis. 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:

$$\vec{\mathbf{e}} = \frac{1}{\mu}\left(v^2 - \frac{\mu}{r}\right)\vec{\mathbf{r}} - \left(\vec{\mathbf{r}}\cdot\vec{\mathbf{v}}\right)\vec{\mathbf{v}} \tag{4.30}$$

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:

$$\cos i = \frac{\hat{k}\cdot\vec{\mathbf{h}}}{\left|\hat{k}\right|\left|\vec{\mathbf{h}}\right|} \tag{4.31}$$

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 $\hat{i}$ unit vector of the reference frame to the location of the orbit's ascending node, where the ascending node is the point on the equatorial plane which the satellite crosses from south to north. 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,:

$$\vec{\mathbf{n}} = \hat{k} \times \vec{\mathbf{h}} \tag{4.32}$$

which allows us to calculate the longitude of the ascending node:

$$\cos\Omega = \frac{\hat{\imath} \cdot \vec{\mathbf{n}}}{|\hat{\imath}|\,|\vec{\mathbf{n}}|} \tag{4.33}$$

$$\text{if } (n_j < 0) \text{ then } \Omega = 360^\circ - \Omega \tag{4.34}$$

which is checked for the correct quadrant by inspecting the sign of the $\jmath$-component of $\vec{\mathbf{n}}$, which if negative, means that Ω lies in Quadrant III-IV. In the case of an uninclined orbit Ω is undefined.

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

$$\cos\omega = \frac{\vec{\mathbf{n}} \cdot \vec{\mathbf{e}}}{|\vec{\mathbf{n}}|\,|\vec{\mathbf{e}}|} \tag{4.35}$$

$$\text{if } (e_k < 0) \text{ then } \omega = 360^\circ - \omega \tag{4.36}$$

and is undefined for a circular orbit (since there is not periapsis).

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

$$\cos\nu = \frac{\vec{\mathbf{e}} \cdot \vec{\mathbf{r}}}{|\vec{\mathbf{e}}|\,|\vec{\mathbf{r}}|} \tag{4.37}$$

$$\text{if } (\vec{\mathbf{r}} \cdot \vec{\mathbf{v}} < 0) \text{ then } \nu = 360^\circ - \nu \tag{4.38}$$

See Section 4.3.3 for alternate representations to true anomaly.

Special Cases

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

$$\cos\tilde{\omega}_{true} = \frac{\hat{\imath} \cdot \vec{\mathbf{e}}}{|\hat{\imath}|\,|\vec{\mathbf{e}}|} \tag{4.39}$$

$$\text{if } (e_j < 0) \text{ then } \tilde{\omega}_{true} = 360^\circ - \tilde{\omega}_{true} \tag{4.40}$$

Another special case is a circular inclined orbit. A circular orbit means there is not a periapsis from which to measure the argument of perigee and subsequently the true anomaly. Therefore, the argument of latitude, u, is used, which is measured between the ascending node and the satellite's position:

$$\cos u = \frac{\vec{\mathbf{n}} \cdot \vec{\mathbf{r}}}{|\vec{\mathbf{n}}|\,|\vec{\mathbf{r}}|} \tag{4.41}$$

$$\text{if } (r_k < 0) \text{ then } u = 360^\circ - u \tag{4.42}$$

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:

$$\cos \lambda_{true} = \frac{\hat{\imath} \cdot \vec{\mathbf{r}}}{|\hat{\imath}|\,|\vec{\mathbf{r}}|} \tag{4.43}$$

$$\text{if } (r_j < 0) \text{ then } \lambda_{true} = 360^\circ - \lambda_{true} \tag{4.44}$$

Orbital Elements to Position and Velocity

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

1. If circular equatorial, set(ω,Ω)=0.0 and $\nu = \lambda_{true}$.
2. If circular inclined, set $\omega = 0.0$ and $\nu = u$.
3. If elliptical equitorial, set $\Omega = 0.0$ and $\omega = \tilde{\omega}_{true}$.

Then the conversions can be evaluated:

$$\mathbf{r^i} = \begin{bmatrix} \frac{p\cos\nu}{1+e\cos\nu} \\ \frac{p\sin\nu}{1+e\cos\nu} \\ 0 \end{bmatrix} \tag{4.45}$$

$$\mathbf{v^i} = \begin{bmatrix} -\frac{\mu}{p}\sin\nu \\ \frac{\mu}{p}(e+\cos\nu) \\ 0 \end{bmatrix} \tag{4.46}$$

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 mentioned such as for circular and equatorial orbits.

$$\begin{aligned} k_e &= e\cos(\omega + f_r\Omega) \\ h_e &= e\sin(\omega + f_r\Omega) \\ a & \\ \lambda_M &= M + \omega + f_r\Omega \\ p_e &= \tan f_r\frac{i}{2}\sin\Omega \\ q_e &= \tan f_r\frac{i}{2}\cos\Omega \end{aligned} \tag{4.47}$$

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

Canonical Variables

Canonical variables are a formulation of orbital elements that allow the 6×6 state matrix:

$$\begin{aligned} \dot{\mathbf{x}} &= S(\mathbf{x})\,\mathbf{x} \\ \left[6\times1\right] &= \left[6\times6\right]\left[6\times1\right] \end{aligned}$$

to be completely diagonal, with only zeros in the off diagonal elements. Two such formulations are Delaunay variables:

$$\begin{aligned} M & \\ \omega & \\ \Omega & \\ h & \end{aligned} \tag{4.48}$$

$$\begin{aligned} L_d &= \sqrt{\mu a} \\ H_d &= \sqrt{\mu a\left(1-e^2\right)}\cos i \end{aligned} \tag{4.49}$$

and Poincaré variables:

$$\begin{aligned} \lambda_M &= M + \omega + \Omega \\ g_p &= \sqrt{-2\sqrt{\mu a}\left(1-\sqrt{1-e^2}\right)\sin(\omega+\Omega)\cos(\omega+\Omega)} \\ h_p &= \sqrt{-2\sqrt{\mu a\left(1-e^2\right)}\left(\cos i - 1\right)\sin\Omega\cos\Omega} \\ L_p &= \sqrt{\mu a} \\ G_p &= \sqrt{-2\sqrt{\mu a}\left(1-\sqrt{1-e^2}\right)\sin(\omega+\Omega)} \\ H_p &= \sqrt{-2\sqrt{\mu a\left(1-e^2\right)}\left(\cos i - 1\right)\sin\Omega} \end{aligned} \tag{4.50}$$

(4.51)

where the orbital elements are those discussed above.

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 confusing, but compact representation was developed called "2-line Element Sets" or TLE.

INSERT TABLE SHOWING TLE

$$BC = \frac{R_\oplus \rho_0}{2B^*} \tag{4.52}$$

where ρ_0 is the atmospheric density at periapsis. Other parameters are defined by n being the mean velocity of the orbit (rev/day), and $\bar{n}$ being the Kozai mean.

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." [33] is expressed as follows:

$$\sum \mathbf{F} = m\ddot{\mathbf{r}} \tag{4.53}$$

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 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 will 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 which 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 which act along the line joining the centers of the two bodies.

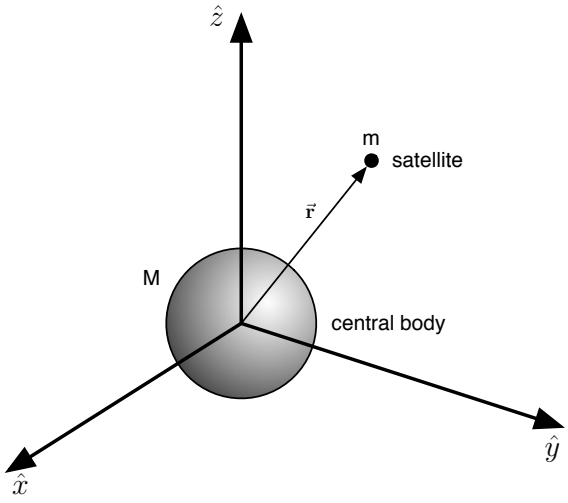

Figure 4.2: *Two body gravity diagram*

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:

$$\mathbf{F}_g = -\frac{GMm}{r^2}\frac{\mathbf{r}}{r} \tag{4.54}$$

where $\mathbf{F}_g$ is the force of gravity acting on mass M and m, and the vector between the two masses is $\mathbf{r} = \mathbf{r}_M - \mathbf{r}_m$. The parameter G is the universal 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, μ has a modern (most recent, accurate) value of $3.986\,004\,415 \times 10^5 \frac{km^3}{s^2}$. It is important to note that the vector $\mathbf{r}$ is measured with respect to inertial axes.

Based on equations 4.53 and 4.54 the following equations:

$$M\ddot{\mathbf{r}}_M = \frac{GMm}{r^2}\frac{\mathbf{r}}{r} \tag{4.55}$$

$$m\ddot{\mathbf{r}}_m = -\frac{GMm}{r^2}\frac{\mathbf{r}}{r} \tag{4.56}$$

(note the lack of minus sign on the equation for M due to the coordinate frame definition) can be combined to derive:

$$\ddot{\mathbf{r}} = -\frac{G(M+m)}{r^3}\mathbf{r}. \tag{4.57}$$

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 $G(M+m) \approx GM$. A common variable definition $\mu \cong GM$ known as the gravitation parameter (and is different for every central body) simplifies the equations of motion to:

$$\ddot{\mathbf{r}} + \frac{\mu}{r^3}\mathbf{r} = 0. \tag{4.58}$$

4.3.2 Three-body and n-body Equations

As mentioned previously, Equation 4.58 is for motion between only two bodies. However, as is readily apparent, 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. Therefore, a system of n bodies which affect gravity on one another 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 4.54 to calculate the force of gravity, $\mathbf{F}_g$, from one mass, m_j, on the observed mass, m_i:

$$\mathbf{F}_g = -\frac{Gm_im_j}{r_{ji}^3}\mathbf{r}_{ji} \tag{4.59}$$

where

$$\mathbf{r}_{ji} = \mathbf{r}_i - \mathbf{r}_j. \tag{4.60}$$

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

$$\mathbf{F}_g = -Gm_i \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{m_j}{r_{ji}^3} \mathbf{r_{ji}}. \tag{4.61}$$

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 mean anomaly, M, which corresponds to the uniform angular motion on a circle:

$$M = E - e \sin E = \sqrt{\frac{\mu}{a^3}}(t - T) \tag{4.62}$$

where E is the eccentric anomaly, T is the period of the orbit, and t is the time of flight. Equation 4.62 can be simplified by calculating the mean motion, n, which is the mean angular rate of the orbital motion:

$$n = \sqrt{\frac{\mu}{a^3}} \tag{4.63}$$

$$\Rightarrow M = n\,(t - T) \tag{4.64}$$

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

$$\frac{M - M_0}{n} = t - t_0 \tag{4.65}$$

$$= \sqrt{\frac{a^3}{\mu}} \left[2\pi k + E - e \sin E - (E_0 - e \sin E_0)\right] \tag{4.66}$$

$$\sin E = \frac{\sqrt{1 - e^2} \sin \nu}{1 + e \cos \nu} \tag{4.67}$$

$$\cos E = \frac{e + \cos \nu}{1 + e \cos \nu} \tag{4.68}$$

However, to find the eccentric anomaly, an iterative process is required. There are several methods for performing this solution finding, 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:

$$f(x) = P_n(x) + R_n(x) \tag{4.69}$$

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:

$$P_n(x) = f(x_0) + f'(x_0)\delta + \frac{f''(x_0)\delta^2}{2!} + \frac{f^{(n)}(x_0)\delta^n}{n!} \tag{4.70}$$

$$\text{define} \quad \delta = x - x_0 \approx -\frac{f(x_0)}{f'(x_0)} \tag{4.71}$$

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:

$$x_{n+1} = x_n + \delta_n = x_n - \frac{f(x_n)}{f'(x_n)} \tag{4.72}$$

$$\Rightarrow E_{n+1} = E_n + \frac{M - E_n + e\sin E_n}{1 - e\cos E_n} \tag{4.73}$$

Equation 4.73 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:

$$\cos\nu = \frac{\cos E - e}{1 - e\cos E} \tag{4.74}$$

$$r = a\left(1 - e\cos E\right) \tag{4.75}$$

Kepler's problem uses the solution to Kepler's equation to propagate a satellite through an orbit. To solve the problem, we must first determine the orbital elements from the position and velocity vectors as developed in Section . We then use these elements to determine the eccentric anomaly (using eccentricity and true anomaly), then use Kepler's equation (4.62) to determine the new mean anomaly after a prescribed amount of time. We then convert the eccentric anomaly back to true anomaly for the orbit, and determine the new position and velocity vectors (Equations 4.45 and 4.46).

4.3.4 Constants of Motion

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

The specific angular momentum is conserved through an orbit:

$$\vec{\mathbf{h}} = \vec{\mathbf{r}} \times \vec{\mathbf{v}} = constant \tag{4.76}$$

The specific mechanical energy is also conserved:

$$\xi = \frac{v^2}{2} - \frac{\mu}{r} = -\frac{\mu}{2a} \tag{4.77}$$

which is the sum of mechanical energy, $\frac{v^2}{2}$, and potential energy, $-\frac{\mu}{r}$.

4.4 Pertubations

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

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 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 more in-depth discussion on the subject in order to develop better pertubation models as is necessary.[38][33]

These disturbances effect on the dynamics is summed up by Cowell's Formulation:

$$\ddot{\vec{\mathbf{r}}} = -\frac{\mu}{r^3}\vec{\mathbf{r}} + \vec{a}_{perturbed} \tag{4.78}$$

which is very nice for simulations 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 4.54 we assumed a uniform gravity field created by a point mass. However, this is not a completely accurate assumption to make, and can even 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 the reader pursue it for full understanding of the underlying physics. A good presentation may be found in Vallado[38], and the results will be shown here.

The components of the nonspherical acceleration ($\mathbf{a_{nonspherical}} = [\mathbf{a_i}, \mathbf{a_j}, \mathbf{a_k}]^\mathbf{T}$) are:

$$a_i = \left[\frac{1}{r}\frac{\partial U}{\partial r} - \frac{r_k}{r^2\sqrt{r_i^2 + r_j^2}}\frac{\partial U}{\partial \phi_{gc}}\right] r_i - \left[\frac{1}{r_i^2 + r_j^2}\frac{\partial U}{\partial \lambda}\right] r_j \tag{4.79}$$

$$a_j = \left[\frac{1}{r}\frac{\partial U}{\partial r} - \frac{r_k}{r^2\sqrt{r_i^2 + r_j^2}}\frac{\partial U}{\partial \phi_{gc}}\right] r_j + \left[\frac{1}{r_i^2 + r_j^2}\frac{\partial U}{\partial \lambda}\right] r_i \tag{4.80}$$

$$a_k = \frac{1}{r}\frac{\partial U}{\partial r} r_k + \frac{\sqrt{r_i^2 + r_j^2}}{r^2}\frac{\partial U}{\partial \phi_{gc}} \tag{4.81}$$

where the partial derivatives of the potential function are:

$$\frac{\partial U}{\partial r} = -\frac{\mu}{r^2}\sum_{\ell=2}^{\infty}\sum_{m=0}^{\ell}\left(\frac{R_\oplus}{r}\right)^\ell (\ell+1) P_{\ell m}\sin\phi_{gc}\left(C_{\ell m}\cos m\lambda + S_{\ell m}\sin m\lambda\right) \tag{4.82}$$

$$\frac{\partial U}{\partial \phi_{gc}} = \frac{\mu}{r}\sum_{\ell=2}^{\infty}\sum_{m=0}^{\ell}\left(\frac{R_\oplus}{r}\right)^\ell \left(P_{\ell,m+1}\sin\phi_{gc} - m\tan(\phi_{gc}) P_{\ell m}\sin\phi_{gc}\right) \times \left(C_{\ell m}\cos m\lambda + S_{\ell m}\right) \tag{4.83}$$

$$\frac{\partial U}{\partial \lambda} = \frac{\mu}{r}\sum_{\ell=2}^{\infty}\sum_{m=0}^{\ell}\left(\frac{R_\oplus}{r}\right)^\ell m P_{\ell m}\sin\phi_{gc}\left(S_{\ell m}\cos m\lambda - C_{\ell m}\sin m\lambda\right) \tag{4.84}$$

Calculation of these equations requires the use of the Legendre Functions ($P_{\ell m}$), a table of which can be found in Vallado[38, pg. 491], as well as the formulation of the gravitational coefficients ($C_{\ell m}$ and $S_{\ell m}$).

The order of the summations is determined by the desired degree of accuracy of the model. However, this comes at the cost of computation time. So the user should be careful to

only choose an order of accuracy congruant with her modeling requirements. For most applications, J2 pertubations ($\ell = 2$) are sufficient, however, more refined models may use J4 or J6 pertubations.

4.4.2 Atmospheric Drag

There is a force pertubation due to the drag of the spacecraft through the atmosphere of the central body it is orbiting. This varies based on the size and drag surface of the satellites, as well as the position, wind velocity, and direction, atmospheric density, season, and numerous other factors that may be considered but won't 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:

$$\vec{a}_{drag} = -\frac{1}{2}\frac{c_D A_D}{m}\rho v_{rel}^2 \frac{\vec{v}_{rel}}{|\vec{v}_{rel}|} \tag{4.85}$$

$$\vec{v}_{rel} = \frac{d\vec{\mathbf{r}}}{dt} - \vec{\omega}_\oplus \times \vec{\mathbf{r}} = \begin{bmatrix} \frac{dx}{dt} + \omega_\oplus y \\ \frac{dy}{dt} + \omega_\oplus x \\ \frac{dz}{dt} \end{bmatrix} \tag{4.86}$$

where ρ is the atmospheric denstiy, m is the mass of the satellite, A_D is the cross-sectional area in 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:

$$\begin{aligned} \vec{v}_{rel} &= \vec{\mathbf{v}}_{ECI} - \vec{\omega} \times \vec{\mathbf{r}}_{ECI} + \vec{\mathbf{v}}_{wind} \\ &= \begin{bmatrix} \frac{dx}{dt} + \omega_\oplus y + v_w\left(\cos\alpha \sin\delta \cos\beta_w + \sin\alpha \sin\beta_w\right) \\ \frac{dy}{dt} + \omega_\oplus x + v_w\left(\sin\alpha \sin\delta \cos\beta_w + \cos\alpha \sin\beta_w\right) \\ \frac{dz}{dt} + v_w\left(\cos\delta \cos\beta_w\right) \end{bmatrix} \end{aligned} \tag{4.87}$$

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.

These atmospheric models can become as complicated as necessary and is effective for a simulation considering the expected ballistic coefficient, altitude, and mission lifetime. Refer to Vallado[38], Bate Mueller & White[33] or Battin[**?**] for a more in-depth discussion of atmosphere 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 low, but can vary largely between eclipse, based on the body, and during solar storms. The solar-radiation pressure is even the basis for such spacecraft propulsion designs as solar sails, and so should be used for accurate models

The solar pressure, p_{SR}, or change in momentum, is the main parameter in determing the force of the solar-radiation pressure. For the Earth, this 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, as well as 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, 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:

$$\vec{a}_{radiation} = -\frac{p_{SR} c_R A_S}{m} \frac{\vec{\mathbf{r}}_{\odot sat}}{|\vec{\mathbf{r}}_{\odot sat}|} \tag{4.88}$$

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 very 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 if a satellite is in sunlight are[38]:

$$\tau_{min} = \frac{|\vec{\mathbf{r}}_{sat}|^2 - \vec{\mathbf{r}}_{sat} \cdot \vec{\mathbf{r}}_{\odot}}{|\vec{\mathbf{r}}_{sat}|^2 + |\vec{\mathbf{r}}_{\oplus}|^2 - 2\vec{\mathbf{r}}_{sat} \cdot \vec{\mathbf{r}}_{\odot}} \tag{4.89}$$

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

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 for the same order of magnitude seperations between the bodies, 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 Moon. Therefore, it is necessary to develop equations that do rely on the small, and sometimes inaccurate distance difference when talking about the distance from the Earth to the Sun and the satellite to the Sun.

$$\ddot{\vec{\mathbf{r}}}_{\oplus\mathbf{sat}} = -\frac{Gm_{\oplus}}{r^3_{\oplus sat}}\vec{\mathbf{r}}_{\oplus sat} - \sum_{k=1}^{n}\frac{Gm_k}{r^3_{\oplus k}}\left(\vec{\mathbf{r}}_{\oplus k} - \beta_k\vec{\mathbf{r}}_{satk}\right) \tag{4.90}$$

$$\beta_k = 3B_k + 3B_k^2 + B_k^3 \text{ with } B_k = B(\zeta_k)$$

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

4.4.5 Other Pertubations

These pertubations are just an introduction to the number, and accuracy of models that are available for increased simulation accuracy. Some other examples may be 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. It is also the crux of simulation. There are 2 main methods of propagating a satellite, analytically, or numerically. Analytical propagation uses a set of equations to evaluate the discrete solution to 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 to the satellite's position after a given time span.

4.5.1 Analytic

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 combines the intial position and velocity vectors to determine the new position and velocity vectors:

$$\vec{\mathbf{r}} = f\vec{\mathbf{r}}_0 + g\vec{\mathbf{v}}_0 \tag{4.91}$$
$$\vec{\mathbf{v}} = \dot{f}\vec{\mathbf{r}}_0 + \dot{g}\vec{\mathbf{v}}_0 \tag{4.92}$$

where

$$f = \frac{x\dot{y}_0 - \dot{x}_0 y}{h} \tag{4.93}$$
$$\dot{f} = \frac{\dot{x}\dot{y}_0 - \dot{x}_0\dot{y}}{h} \tag{4.94}$$
$$g = \frac{x_0 y - x\dot{y}_0}{h} \tag{4.95}$$
$$\dot{g} = \frac{x_0\dot{y} - \dot{x}y_0}{h} \tag{4.96}$$

with 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 the functions using the relation:

$$1 = f\dot{g} - \dot{f}g. \tag{4.97}$$

To demonstrate how to obtain the f and g functions we need to assume we know the true anomaly, ν. The components of the position vector must be determined in perifocal coordinates, as well as the time derivatives:

$$x = r\cos\nu \tag{4.98}$$
$$y = r\sin\nu \tag{4.99}$$
$$\dot{x} = -\sqrt{\frac{\mu}{p}}\sin\nu \tag{4.100}$$
$$\dot{y} = \sqrt{\frac{\mu}{p}}\left(e + \cos\nu\right) \tag{4.101}$$

which can be used in equation 4.93 to determine the functions:

$$f = 1 - \frac{r}{p}(1 - \cos\delta\nu) \tag{4.102}$$

$$g = \frac{rr_0 \sin\delta\nu}{\sqrt{\mu p}} \tag{4.103}$$

$$\dot{f} = \sqrt{\frac{\mu}{p}} \tan\left(\frac{\delta\nu}{2}\right)\left(\frac{1-\cos\delta\nu}{p} - \frac{1}{r} - \frac{1}{r_0}\right) \tag{4.104}$$

$$\dot{g} = 1 - \frac{r_0}{p}(1 - \cos\delta\nu) \tag{4.105}$$

$$\tag{4.106}$$

These equations can be derived for different element sets or when including 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. We define our state as the position and velocity vectors. Then, the time derivative of the state is:

$$\dot{\mathbf{x}} = \begin{bmatrix}\dot{\mathbf{r}} \\ \dot{\mathbf{v}}\end{bmatrix} = \begin{bmatrix}\mathbf{v} \\ -\frac{\mu}{r^3}\mathbf{r} + \mathbf{a}_{disturbance}\end{bmatrix}. \tag{4.107}$$

We then need to determine our disturbance accelerations, which are linearly addable:

$$\mathbf{a}_{distrubance} = \mathbf{a}_{nonspherical} + \mathbf{a}_{drag} + \mathbf{a}_{3-body} + \mathbf{a}_{solar-radiation} \tag{4.108}$$

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 lead them to 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, allows the users the ability to implement more advance 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 is given, as well as associated interpolators. 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 state and its derivatives. There are both single-step and multi-step integrators, both of which can be either variable of fixed step size. Single-step integrators use the state at time t_0 and the time derivatives to calculate the state at time $t_o + h$. 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 then correct backwards in time. Examples of multi-step integrators include Adams-Bashforth and Gauss-Jackson.

5.1.1 Euler

The simplest integrator is that of the Euler integrator 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:

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

$$\tag{5.2}$$

This is very simple to calculate, even though it requires the time derivative of the state. However, this is normally available (especially for dynamics equations), but is not very accurate, and also requires analysis to determine an adequate time-step $t - t_0$.

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 then combines these to form a more accurate estimate of the new state. The fourth-order Runge-Kutta integrator is based off of a fourth-order Taylor series and is formulated as follows:

$$\begin{aligned} \dot{y}_1 &= f\left(t_0, y_0\right) \\ \dot{y}_2 &= f\left(t_0 + \frac{h}{2}, y_0 + \frac{h}{2}\dot{y}_1\right) \\ \dot{y}_3 &= f\left(t_0 + \frac{h}{2}, y_0 + \frac{h}{2}\dot{y}_2\right) \\ \dot{y}_4 &= f\left(t_0 + h, y_0 + h\dot{y}_3\right) \\ y(t) &= y(t_0) + \frac{h}{6}\left(\dot{y}_1 + 2\dot{y}_2 + 2\dot{y}_3 + \dot{y}_4\right) + O\left(h^5\right) \end{aligned} \tag{5.3}$$

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

As is necessary, higher-order Runge-Kutta integrators can be derived.[38] A general formulation for ith-order and j-iterations is:

$$y(t) = y_0 + h\sum_{i=0}^{j-1} b_i\dot{y}_i + O\left(h^{n+1}\right) \qquad n = 0, 1, 2, ..., n-1$$

$$\dot{y}_0 f(t_n, y_n) \qquad c_{i0} = p_i \sum j = 1^{i-1} c_{ij} \tag{5.4}$$

$$\dot{y}_i = f\left(t_n + p_i h, y_n + h\sum_{j=0}^{i-1} c_{ij}\dot{y}_j\right) \qquad i = 1, 2, ..., j-1$$

$$\tag{5.5}$$

where b_i, c_{ij}, and p_i are user defined parameters. Refer to Fehlberg[13] and Der[11] for discussion of their calculation.

5.1.3 Runge-Kutta-Fehlberg

A rather different implementation of the Runge-Kutta integration is the Runge-Kutta-Fehlberg, or embedded Runge Kutta method that uses a variable single step-size that is determined based on the error of the calculation, which then sizes the step-size between the integration time-steps.

$$\begin{aligned}
k_1 &= hf(t_0, y_0) \\
k_2 &= hf\left(t_0 + \frac{1}{4}h, y_0 + \frac{1}{4}k_1\right) \\
k_3 &= hf\left(t_0 + \frac{3}{8}h, y_0 + \frac{3}{32}k_1 + \frac{9}{32}k_2\right) \qquad (5.6) \\
k_4 &= hf\left(t_0 + \frac{12}{13}h, y_0 + \frac{1932}{2197}k_1 - \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3\right) \\
k_5 &= hf\left(t_0 + h, y_0 + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4\right) \\
k_6 &= hf\left(t_0 + \frac{1}{2}h, y_0 - \frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5\right)
\end{aligned}$$

(5.7)

To determine the calculation step-size h we must calculate an error and evaluate if it is within the specified tolerance:

$$error = \frac{1}{360}k_1 - \frac{128}{4275}k_2 - \frac{2197}{75240}k_4 + \frac{1}{50}k_5 + \frac{2}{55}k_6 \qquad (5.8)$$

$$s \cong 0.8408\left[\frac{1 \times 10^{-8}h}{error}\right]^{1/4} \qquad (5.9)$$

$$\begin{aligned}
&\text{if } s < 0.75 \text{ and } h > 2h_{min} \text{ then } h = h/2 \\
&\text{if } s > 1.5 \text{ and } 2h < 2h_{max} \text{ then } h = 2h \\
&\text{recalculate the time step until these conditions are met} \\
&\text{else } y = y_0 + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5 \qquad (5.10)
\end{aligned}$$

where $h_{min} = h/64$ and $h_{max} = 64h$.

5.1.4 Adams-Bashforth

As mentioned previously, there are both single-step and multi-step integration techniques. up to this point, we have only presented single-step integrators. Adams-Bashforth, how-

ever, is a multi-step integration technique. A good discussion of the differences can be found in [32].

For the multi-step integrator, initial conditions 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-Kutte method[32]:

Given the initial conditions, calculate the first 4 time-steps using the Runge-Kutta 4th order integrator:

$$t = t_0 + ih$$

predictor:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \frac{h}{24}\left[55f(t_i, \mathbf{x}_i) - 59f(t_{i-1}, \mathbf{x}_{i-1}) + 37f(t_{i-2}, \mathbf{x}_{i-2}) - 9f(t_{i-3}, \mathbf{x}_{i-3})\right]$$

corrector:

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \frac{h}{24}\left[9f(t, \mathbf{x}) + 19f(t_{i-1}, \mathbf{x}_{i-1}) - 5f(t_{i-2}, \mathbf{x}_{i-2}) + f(t_{i-3}, \mathbf{x}_{i-3})\right]$$

where $\dot{\mathbf{x}} = f(t, \mathbf{x})$.

5.2 Interpolation

As may be apparent in the discussion of integration methods, it may occur where a user simulating dynamic equations may request the state at non-mesh points, or points that were not specifically calculated in the integration algorithm. This requires the use of interpolation to find states in between the mesh points. For example, dynamic equations were integrated from 0 to 100 seconds with 5 second intervals. However, we may be required to know the state at 56 seconds. This would require us to interpolate between the 55 and 60 second mesh points 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 calculating time-step until the next update. This would cause, for example, the state at 55 seconds to be the same for all times up until the new state at 60 seconds. This is obviously a very poor estimate. The next best step would be to use a linear interpolation:

$$P(t) = \frac{t - t_1}{t_0 - t_1} x_0 + \frac{t - t_0}{t_1 - t_0} x_1 \tag{5.11}$$

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, this can be generalized to:

$$P(t) = \sum_{k=0}^{n} x_k L_{n,k}(t) \tag{5.12}$$

$$L_{n,k}(t) = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{t - t_i}{t_k - t_i} \tag{5.13}$$

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

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. This has the benefit of allowing us to choose the set of mesh points we want to include in our 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[32] and

shown here:

$$\begin{aligned}
&\text{For } i\text{=0,1,...,}n\text{-1 set} && h_i = x_{i+1} - x_i \\
&\text{For } i\text{=1,2,...,}n\text{-1 set} && \alpha_i = \frac{3}{h_i}(x_{i+1} - x_i) - \frac{3}{h_{i-1}}(x_i - x_{i-1}) && (5.14)\\
& && && (5.15)\\
&\text{Set} && l_0 = 1 \quad \mu_0 = 0 \quad z_0 = 0 \\
&\text{For } i\text{=1,2,...,}n\text{-1 set} && l_i = 2(t_{i+1} - t_{i-1}) - h_{i-1}\mu_{i-1} \\
& && \mu_i = \frac{h_i}{l_i} \qquad i = 1, 2, ...n-1 && (5.16)\\
& && z_i = \frac{\alpha_i - h_{i-1}z_{i-1}}{l_i} \\
& && && (5.17)\\
&\text{Set} && l_n = 1 \quad \mu_n = 0 \quad z_n = 0 \\
&\text{For } j\text{=n-1,n-2,...,0 set} && && (5.18)\\
& && c_j = z_j - \mu_j c_{j+1} \\
& && b_j = \frac{x_{j+1} - x_j}{h_j} - \frac{h_j}{3}(c_{j+1} + 2c_j) && (5.19)\\
& && d_j = \frac{c_{j+1} - c_j}{3h_j} \\
& && && (5.20)
\end{aligned}$$

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:

$$S_j(t) \quad = \quad x_j + b_j(t - t_j) + c_j(t - t_j)^2 + d_j(t - t_j)^3 \tag{5.21}$$

There is also a very similar derivation of the cubic spline with clamped endpoints shown in Burden[32, pg. 148].

5.3 Computation

Normal computation that is carried out by humans can usually be exact, such as saying $\sqrt{2}$, or $\sin 4\pi$. However, when these computations are carried out on a computer, they must be represented digitally and as such are subject to errors due to the finite nature of their digital representation (from before, 1.414213... and 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 concerns, their effect on simulation, and ways to minimize this effect.

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* that has a coprocessor 64-bit (binary digit) representation for the real numbers. This *long real* allows us to assume atleast 16 decimal digits of precision.[32]

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.

While 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 in order 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 also greatly decrease the error. [32]

In the following example:

$$\begin{aligned} f(y) &= x^2 - 3x + 2 \\ \Rightarrow f(y) &= (x-2)(x-1) \end{aligned}$$

the first equation has a total of 3 multiplications and 2 additions (where subtractions are additions that are merely sign compliments), whereas the second equation has 2 additions and 1 multiplication. As will be discussed below, this does not only decrease error, it can increase operational speed.

5.3.2 Execution Speed

The optimization of code to decrease execution speed is the subject of unending challenge to most programmers. Depending on the language, there are numerous methods for speeding up running programs such:

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

2. The use of 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. Allow for compilers to optimize code, and possibly use a higher order optimization scheme.
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, can be computer architecture dependent, thereby stifling cross-platform compatability, and often produces illegible code. The goal of the user and programmer should be to first produce correctly functioning code, then seek to optimize. This may require the developer to also do research into the appropriate language, architecture, compiler and operating system. Good resources include Meyers[27][26], Stroustrup[35], Sutter[36][37]

5.3.3 Comparison of Accuracy

Not sure how to define.

Chapter 6

Software Design

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

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 attitude attitude integrated propagation of a satellite with multi-body dynamics and control mechanisms. As specified in the requirements, a goal of the system is to provide for this functionality in a piece-meal fashion such that specific components can be put in, 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 called out for future users and maintainers of the system that may require added functionality. Together, a cartoon of the entire framework is shown in Figure 6.1. The specific toolkits include UML diagrams of their architecture and some interfaces to the primary classes. For more information, refer to the Open-SESSAME User's & Maintainer's Guide[**?**]

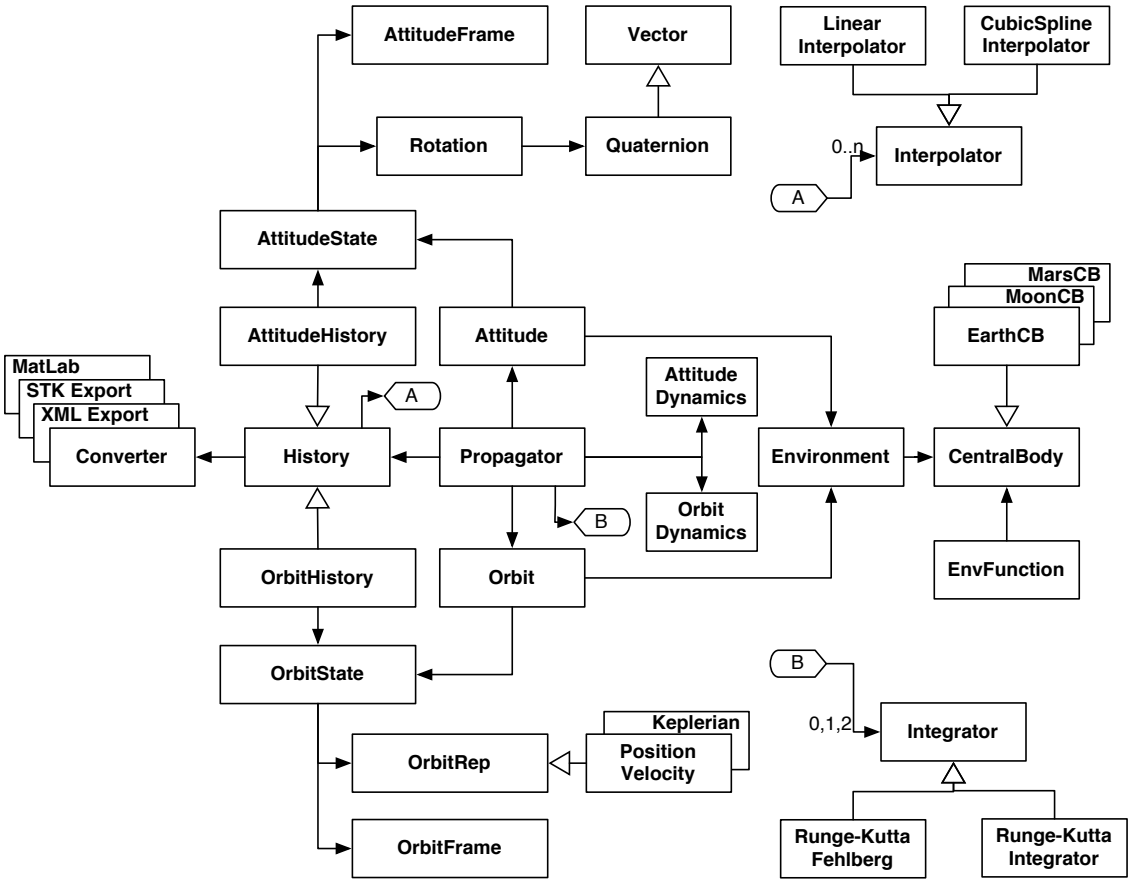

Figure 6.1: *UML Diagram of Spacecraft simulation and control software components*

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*, with the functionality included for, but not specifically implemented in a seperate class, for: *Euler Angles*, and *Euler Axis & Angle*.

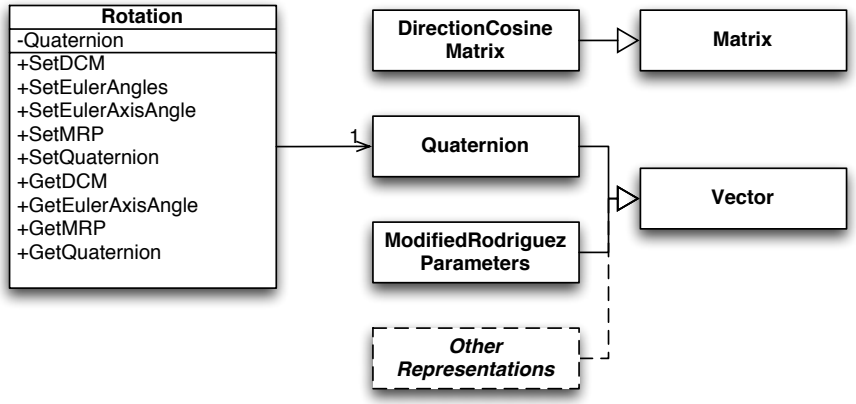

Figure 6.2: *Rotation Library UML diagram*

Extension Point: there are other kinematic representations that could be implemented as necessary: *Gibb's Vector* or *Euler-Rodriguez Parameters*. These classes would also be derived from the *Vector* class and include the conversion algorithms to and from each of the other existing representations. Furthermore, 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 an Abstract Data Type, ADT) in that it is no specific *type* of representation, but rather a general rotation concept between reference

frames. For this reason, the *Rotation* class may be set and output in any of the kinematic representation types, as well as 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 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. Instead, 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 some of the other representations and also only contains 4-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 (cout).

```
// 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 that is the transpose of dcm1 (~dcm1)
Quaternion q2(~dcm1);
// create a rotation that is 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;
rot1.GetEulerAxisAngle(eulerAxis, eulerAngle);
```

```
cout << eulerAxis << eulerAngle;
```

6.1.2 Attitude Toolkit

Description

The *Attitude Toolkit* is the collection of classes and tools that provide for analysing, modeling, and simulating the attitude of a spacecraft. This includes state representations, kinematic and dynamic equations of motion, and the general spacraft attitude.

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, succint 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.

$$\dot{\mathbf{x}} = RHS(t, \mathbf{x}, Orbit, Attitude, parameters, DisturbanceFunction) \tag{6.1}$$

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 **??**. The state, $\mathbf{x}$, is the vector of states values at time t. The *Orbit* and *Attitude* inputs are the current **Orbit** and **Attitude** instances 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.

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* are the attitude state conversion functions. The standard state output vector, such as may be returned from an integration timestep,

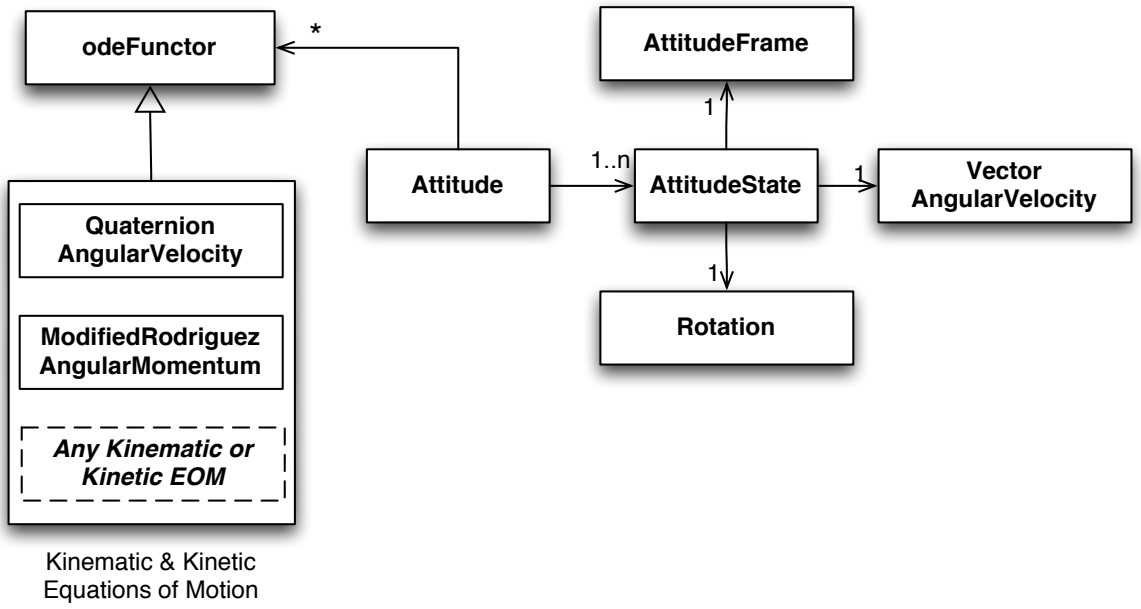

Figure 6.3: *Attitude toolkit UML diagram*

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 doesn't 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, 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);
    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 includes, similar to the *Attitude Toolkit*, state representations, kinematic and dynamic equations of motion, and the general spacraft orbit.

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 accompianied by an **OrbitFrame**, which stores the information regarding the frame from which the **OrbitStateRepresentation** is measured. This could be Earth-Centered Inertial, Moon-Centered Moon-Fixed, or other pertinent representations. By associating a frame with a state representation, consistency is promoted in hopes 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.

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

Usage

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

```
OrbitState myOrbit;
    myOrbit.SetStateRepresentation(new PositionVelocity);
    myOrbit.SetOrbitFrame(new OrbitFrameIJK);
    Vector initPV(6);
```

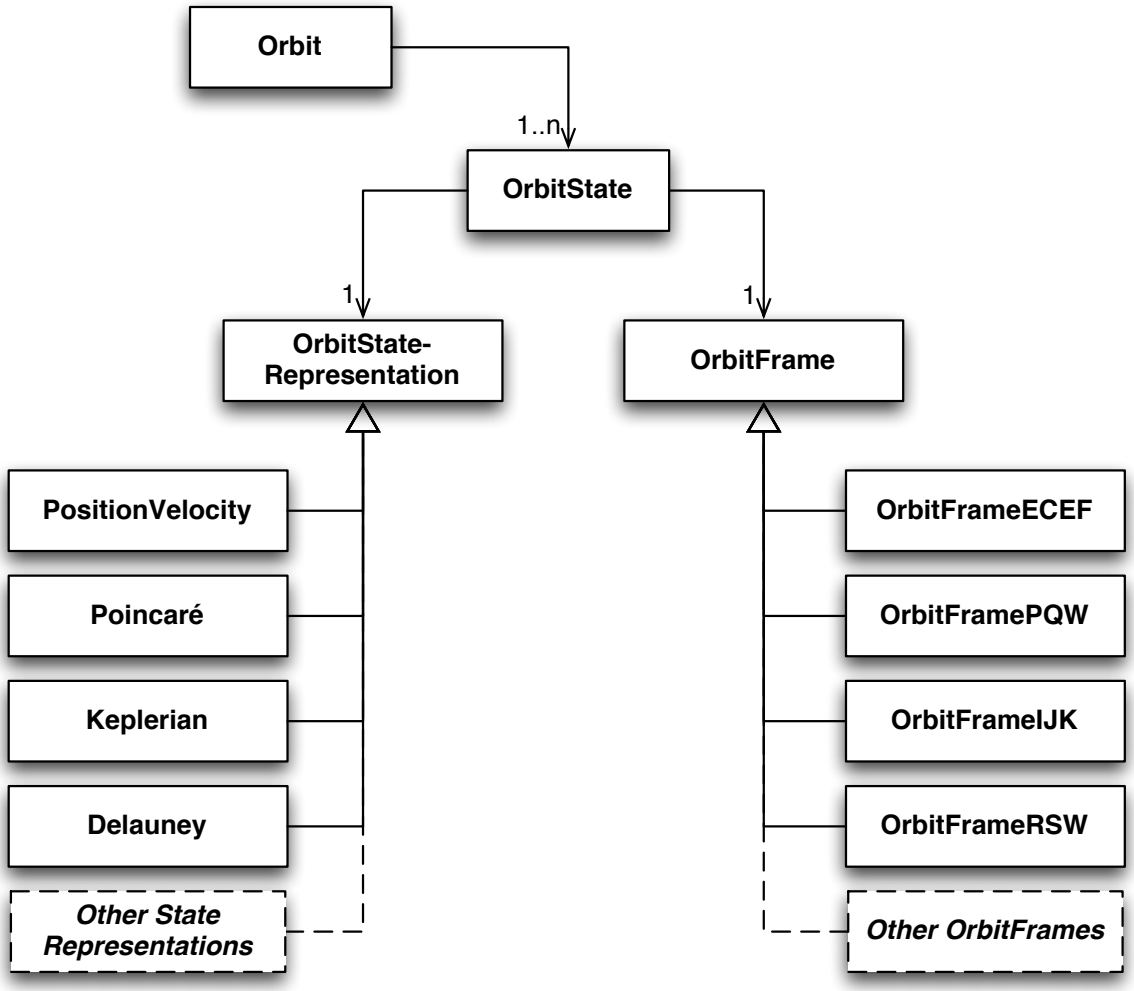

Figure 6.4: *Orbit toolkit UML diagram*

```
        initPV(1) = -5776.6; // km/s
        initPV(2) = -157; // km/s
        initPV(3) = 3496.9; // km/s
        initPV(4) = -2.595;  // km/s
        initPV(5) = -5.651;  // km/s
        initPV(VectorIndexBase+5) = -4.513; // km/s
    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 which is usually reference by the **Orbit** and **Attitude** objects.

Implementation

The user can create an instance of a **CentralBody**, which is 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 **Earth-CentralBody** 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, as well as 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. Then, 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.

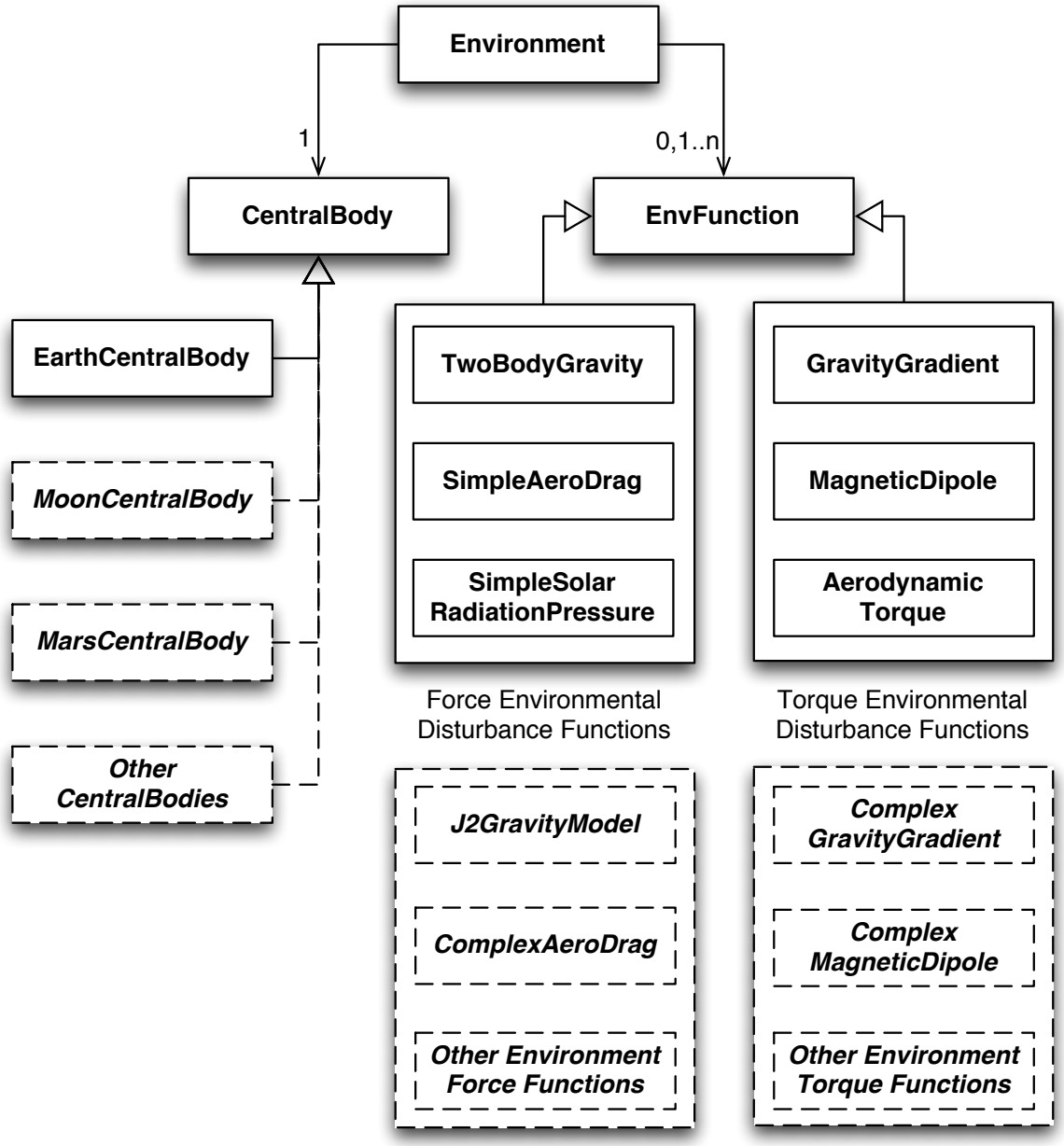

Figure 6.5: *Environment toolkit UML diagram*

Usage

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

```
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(static_cast<double>(10), static_cast<double>(
DragForce.AddParameter(reinterpret_cast<void*>(rho), 2);
pEarthEnv->AddForceFunction(DragForce);
```

The

```
reinterpret_cast<void*>
```

6.1.5 Integrator

Description

The **Integrator** library is really 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 to be integrated of the form:

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

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, equation. The state vector can

be any components the user may wish to integrate, whether it is quaternion and angular velocities, or position, kinetic energy, and momentum. This dynamic equation is then integrated using an implemented integrator described in 5.1, such as **RungeKuttaIntegrator** or **AdamsBashfourthIntegrator**.

Implementation

As mentioned above, there are several integrations 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 environment 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 is 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. By being a **Functor**, the user may even 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

```
////////////////////////////////////////////////
// Setup an integrator and any special parameters
RungeKuttaIntegrator myIntegrator;
```

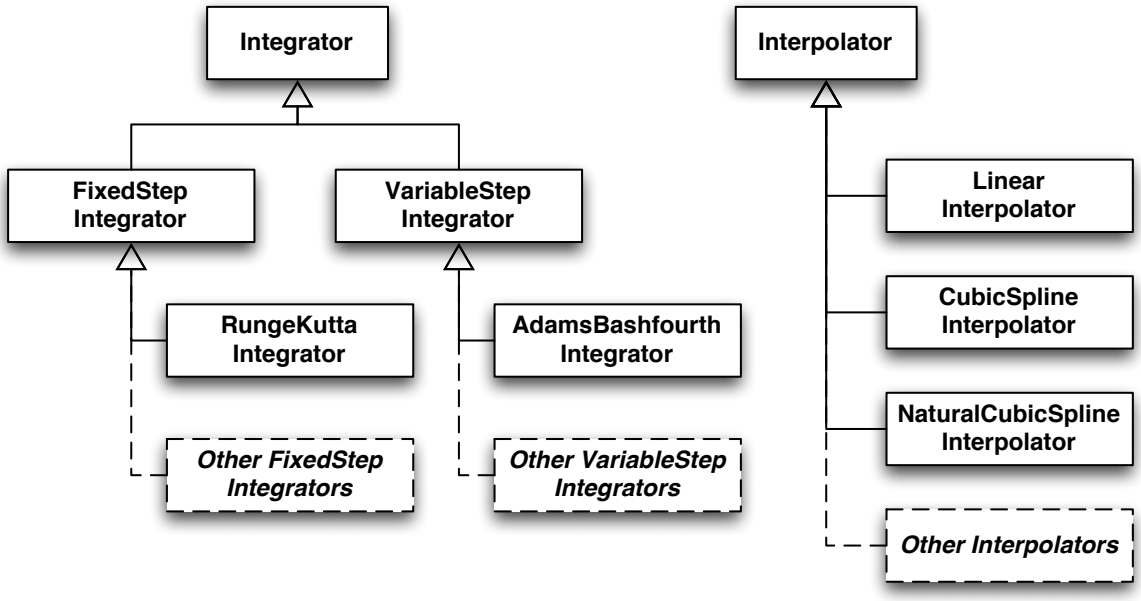

Figure 6.6: *Math toolkit UML diagram*

```
myIntegrator.SetNumSteps(1000);
// Integration times
vector<ssfTime> integrationTimes;
ssfTime begin(0);
ssfTime end(begin + 20);
integrationTimes.push_back(begin);
integrationTimes.push_back(end);

    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 atttiude), strong (orbit and attitude interdepent), and joined (fully coupled dynamic equations) propagation schemes. A **Propagator** can also be used when the attitude or orbit is availabe from an external source (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.

However, 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.

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.

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 initial conditions.

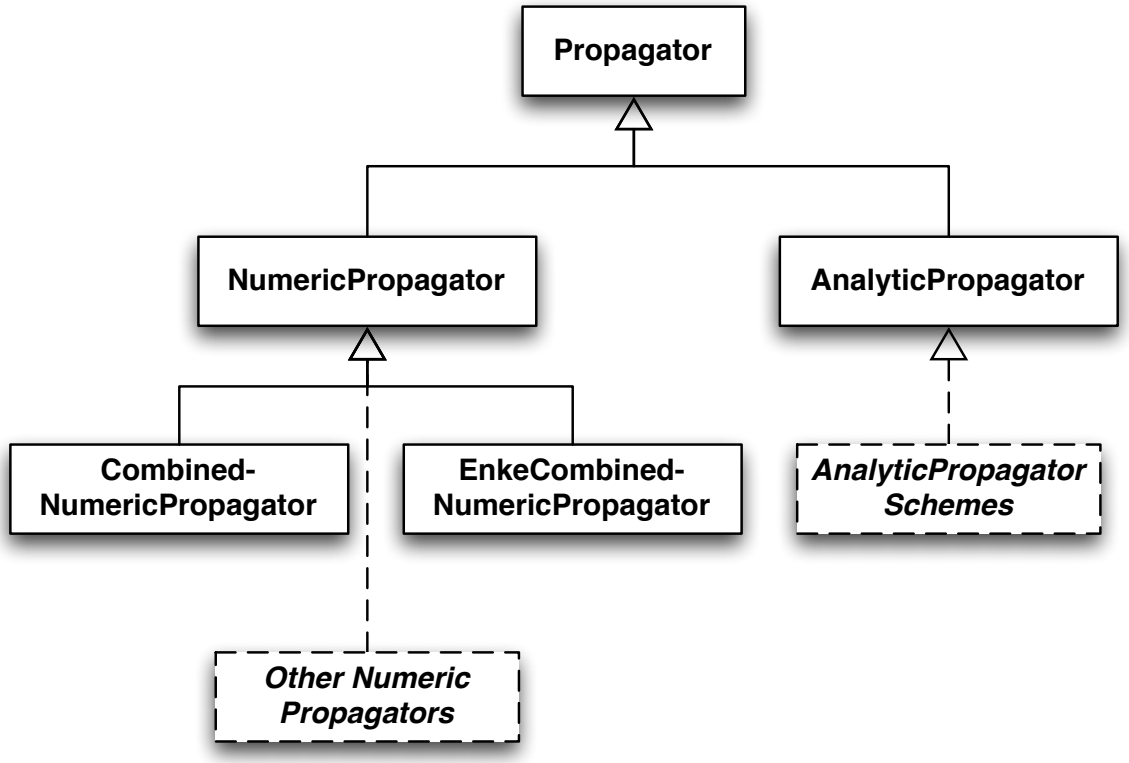

Figure 6.7: *Dynamics library UML diagram*

```
CombinedNumericPropagator* myProp = new CombinedNumericPropagator;
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);
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 of the spacecraft states and parameters from a variety of forms, such as comma-seperated 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.

History

In order 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.

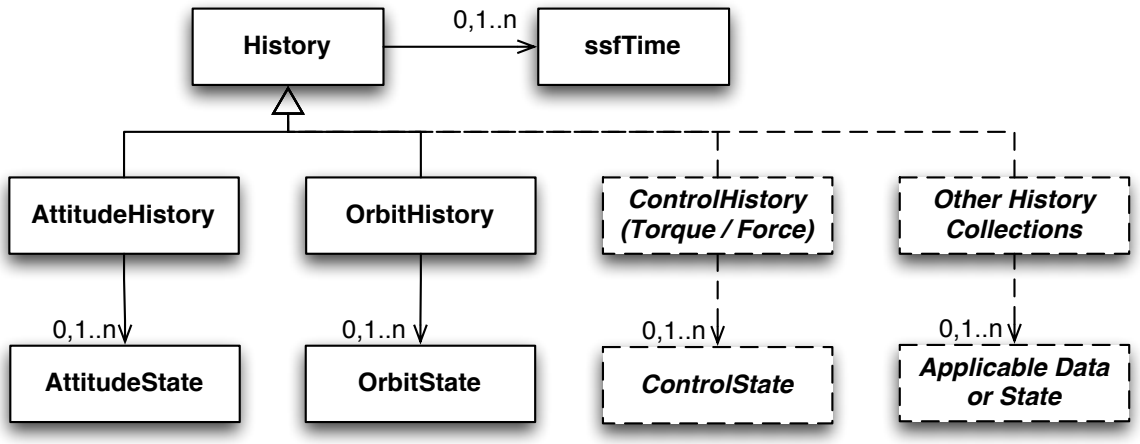

Figure 6.8: *Data Handling toolkit UML diagram*

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;
```

Import/Export

Communications

6.1.8 Visualization

2-D Plotting

3-D Graphics

6.1.9 Utility Libraries

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*).

```
ssfTime simTime;
ssfSeconds integrationTime = 10;
simTime.Set(integrationTime);
ssfTime nowTime(Now());
```

Math Utilities

6.2 Using the Framework

Because Open-SESSAME is just 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 (such as a control torque). 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.

Code the attitude dynamics equation

```
static Vector AttituteDynamics(const ssfTime &_time, const Vector& _integratingState,
{
    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 wIn(3);
    qIn = _integratingState(_(1, 4));
    wIn = _integratingState(_(5,7));
    qIn /= norm2(qIn);
```

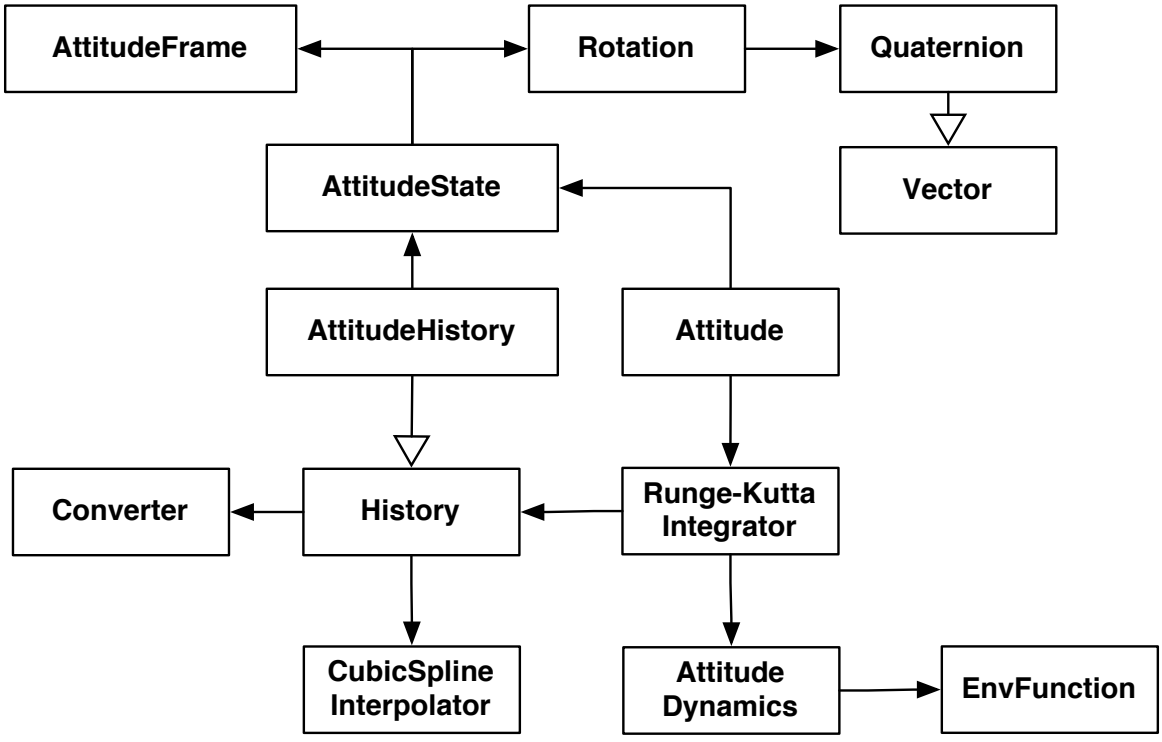

Figure 6.9: *Attitude integration using Open-SESSAME*

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

    bMOI = _parameters(_(1,3),_(1,3));
    Tmoment(_,_) = (_forceFunctorPtr.Call(_time, _Orbit->GetStateObject(),
                                  _Attitude->GetStateObject()))(_);
    Tmoment = (bMOI.inverse() * (Tmoment - skew(wIn) * (bMOI * wIn)));

    stateDot(_(1,3)) = qtemp(_,1);
    stateDot(_(5,7)) = Tmoment(_,1);

    return stateDot;
}
```

Code the disturbance torque function

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

Assign function parameters

```
    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 AttitudeForcesFunctor(&NullFunctor);
```

Create and initialize integrator

```
    AttitudeState myAttitudeState;
    myAttitudeState.SetRotation(Rotation(Quaternion(0,0,0,1)));
```

```
Vector initAngVelVector(3);
    initAngVelVector(1) = 0.1;
myAttitudeState.SetAngularVelocity(initAngVelVector);
```

Create and initialize integrator

```
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

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

Graph or output the state history

```
cout << history;
```

```
Matrix plotting = history(_,_(MatrixIndexBase,MatrixIndexBase+4));
```

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

Code the orbit dynamics equation

```
static Vector TwoBodyDynamics
(const ssfTime &_time, const Vector& _integratingState,
  Orbit *_pOrbit, Attitude *_pAttitude,
  const Matrix &_parameters,
  const Functor &_forceFunctorPtr)
{
    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(_pAttitude)
        Forces = _forceFunctorPtr.Call(_time, orbState,
                                                    _pAttitude->GetStateObject());
    else
        Forces = _forceFunctorPtr.Call(_time, orbState, tempAttState);

    Velocity(_) = _integratingState(_(VectorIndexBase+3,VectorIndexBase+5));


    stateDot(_(VectorIndexBase, VectorIndexBase+2)) = Velocity(_);
    stateDot(_(VectorIndexBase+3, VectorIndexBase+5)) = Forces(_);
    return stateDot;
}
```

Code the disturbance force 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(MatrixIndexBase,MatrixIndexBase) = 398600.4418; //km / s^2
```

Create an initial orbit state

```
OrbitState myOrbit;
myOrbit.SetStateRepresentation(new PositionVelocity);
myOrbit.SetOrbitFrame(new OrbitFrameIJK);
Vector initPV(6);
    initPV(VectorIndexBase+0) = -5776.6; // km/s
    initPV(VectorIndexBase+1) = -157; // km/s
    initPV(VectorIndexBase+2) = 3496.9; // km/s
    initPV(VectorIndexBase+3) = -2.595;  // km/s
    initPV(VectorIndexBase+4) = -5.651;  // km/s
    initPV(VectorIndexBase+5) = -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

```
Matrix plotting = history(_,_(MatrixIndexBase+1,MatrixIndexBase+3));
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, all methods would follow the 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 6.10.

1. Create and populate the environment
2. Define orbit and attitude conversion functions
3. Create and initialize the propagator
4. Propagate the equations
5. Graph or output the state history

Create and populate the environment

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

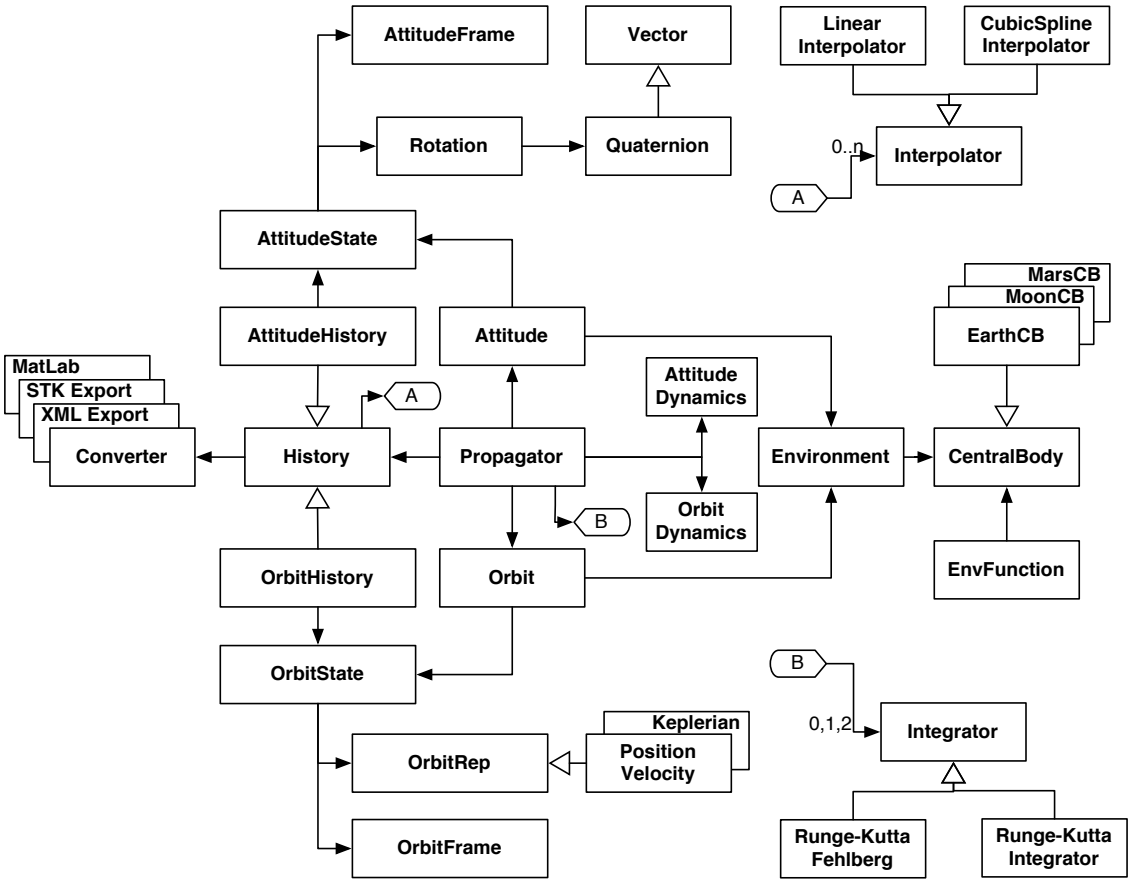

Figure 6.10: *UML Diagram of Spacecraft simulation and control software components*

```
    // 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 Hardware-in-the-Loop

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 6.11 shows the application architecture and interation 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 seperate machine if necessary.

6.2.5 Integrating with External Programs

6.3 Summary

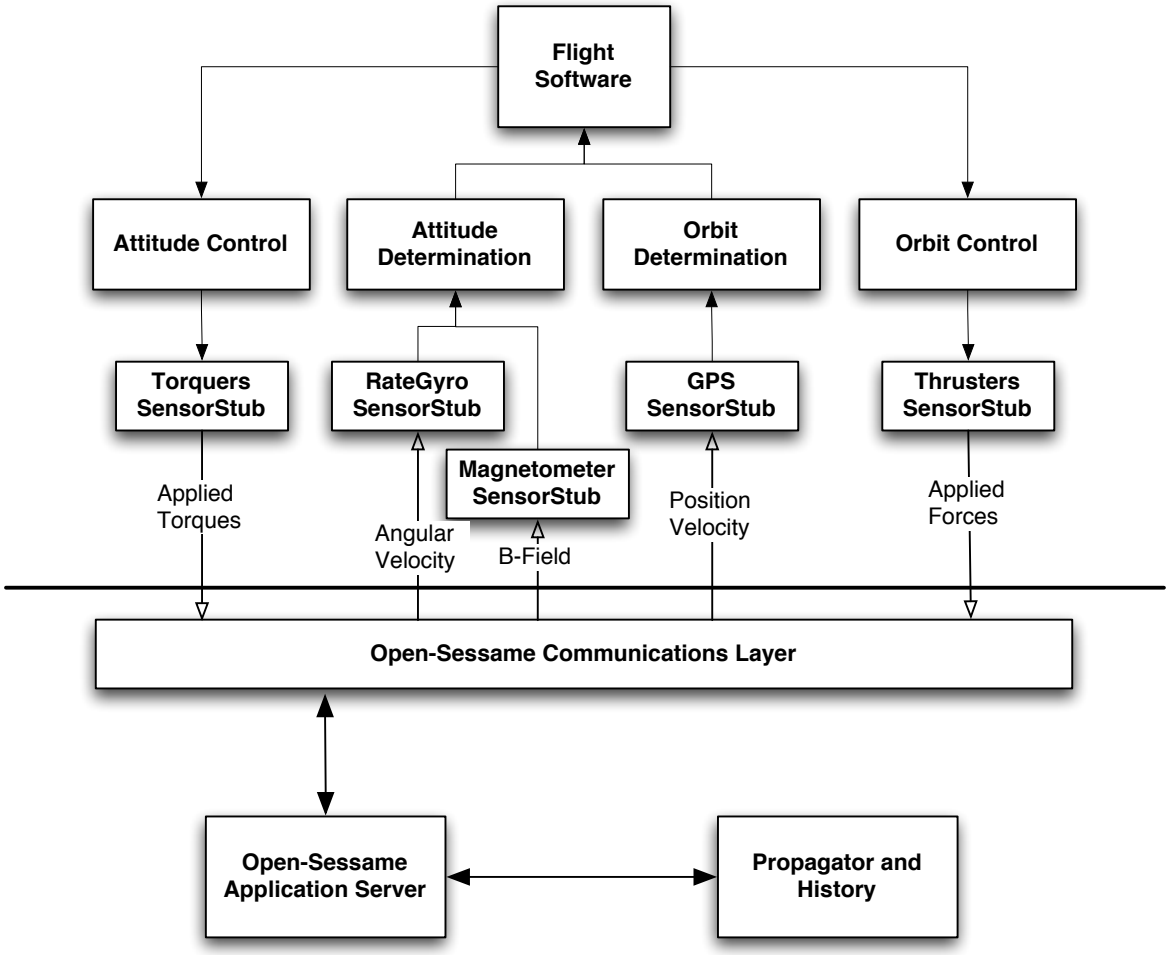

Figure 6.11: *Use of Open-Sessame for hardware-in-the-loop testing and integration.*

References

[1] *Orsa sourceforge website.*, Available at http://orsa.sourceforge.net. 7

[2] *Savi sourceforge website*, http://sourceforge.net/projects/savi. 7

[3] *Winorbit homepage*, Available at www.sat-net.com/winorbit. 7

[4] *Hughes using moon for orbit change*, Aviation Week & Space Technologies (1998). 9

[5] Jerry Banks, *The future of simulation software: A panel discussion*, Proc of the 1998 Winter Simulation Conference (Washington, DC), 1998, pp. 1681–1687. 6

[6] J. Biesiadecki, A. Jain, and M. James, *Advanced simulation environment for autonomous spacecraft*, 1997. 7

[7] Grady Booch, *Object-oriented analysis and design*, Addison-Wesley, Boston, MA, 1994. 6

[8] Francois E. Cellier, *Combined continuous/discrete system simulation languages - usefulness, experiences and future development*, Methodology in Systems Modelling and Simulation (North-Holland, Amsterdam, the Netherlands), 1979, pp. 201–220. 5

[9] James P. Cohoon and Jack W. Davidson, *C++ program design: An introduction to programming and object-oriented design*, 2nd edition ed., McGraw-Hill, Boston, MA, 1999. 11

[10] R. Cubert and P. Fishwick, *Moose: An object-oriented multimodeling and simulation application framework*, 1997. 5

[11] Gim Der, *Runge-kutta integration methods for trajectory propagation revisited*, Proc. of the AAS/AIAA Astrodynamics Specialist Conference (Halifax, Nova Scotia, Canada), AAS 95-420, 1995. 56

[12] Helding Elmqvist, Francois E. Cellier, and Martin Otter, *Object-oriented modeling of hybrid systems*, ESS'93, European Simulation Symposium (Delft, The Netherlands), 1993, pp. 25–28. 5

[13] Erwin Fehlberg, *Classical fifth-, sixth-0, seventh-, and eighth-order runge-kutta formulas with stepsize control*, NASA Technical Report TR-R-287, 1968. 56

[14] D. Folta K.Hartman D. Quinn J.P. How F.H. Bauer, J. Bristow, *Satellite formation flying using an innovative autonomous control system (autocon) environment*, Proc. of the AIAA/AAS Astrodynamics Specialis Conf., Aug 1997, pp. AIAA Paper 97–3821. 8

[15] Mark Frank, *The new FK5 Mean of J2000 time and reference frame transformations*, Dvo technical report 84-2, 1984. 33

[16] T. R. Henderson, *Networking over next-generation satellite systems*, Ph.D. thesis, University of California at Berkeley, Fall 1999. 7

[17] Peter C. Hughes, *Spacecraft attitude dynamics*, John Wiley & Sons, New York, NY, 1986. 25, 28

[18] D. Henriquez J. Biesiadecki and A. Jain, *A reusable, real-time spacecraft dynamics simulator*, 6th Digital Avionics Systems Conference (1997). 7

[19] Henry DeWitt James Woodburn, Ken Williams, *The customization of satellite tool kit for use on the near mission*, Proceedings of the 1997 Space Flight Mechanics Conference (Huntsville, AL), American Astronautical Society (AAS), February 9-12 1997, pp. AAS 97–176. 8

[20] Sergei Tanygin James Woodburn, *Efficient numerical integration of coupled orbit and attitude trajectories using an encke type correction algorithm*, 2001 Astrodynamics Specialist Conference (Quebec City, Canada), American Astronautical Society (AAS), July 30 - August 2, 2001 2001, pp. AAS 01–428. 28

[21] Dave Folta John Bristow, *Autocon user's guide*, NASA, 2 ed., 1999. 8

[22] Lori Kijewski, *Hughes and analytical graphics make first commercial use of moon*, April 29 1998. 9

[23] G.A. Korn, *Interactive dynamic-system simulation*, McGraw-Hill, New York, 1989. 5

[24] G. Pavlou L. Wood and B. G. Evans, *Managing diversity with handover to provide classes of service in satellite constellation networks*, Proceedings of the 19th AIAA International Communication Satellite Systems Conference (ICSSC '01) (Toulouse, France), Vol 3, Session 35, no. 194, April 2001. 7

[25] Kristin Makovec, *A nonlinear magnetic controller for three-axis stability of nanosatellites*, Master's thesis, Virginia Polytechnic Institute and State University, 2001. 24

[26] Scott Meyers, *More effective c++*, Addison-Wesley, Reading, MA, 1996. 62

[27] ———, *Effective c++*, 2nd edition ed., Addison-Wesley, Boston, MA, 1998. 62

[28] M. Otter, H. Elmqvist, and F. E. Cellier, *Modeling of multibody systems with the object-oriented modeling language Dymola*, Proc. NATO/ASI, Computer-Aided Analysis of Rigid and Flexible Mechanical Systems, Troia, Portugal, 1993, pp. 27–29. 5

[29] Michael Tillerson Philip Ferguson, Trent Yang and Jonathan How, *New formation flying testbed for analyzing distributed estimation and control architectures*, AIAA Guidance, Navigation, and Control Conference and Exhibit (Monterey, CA), August 5-8 2002, pp. AIAA 02–4961. 8

[30] Princeton Satellite Systems, 33 Witherspoon St. Princeton, NJ 08542, *Multisatsim user's guide*, 1.1.3 ed., May 2002. 7

[31] Princeton Satellite Systems, 33 Witherspoon St. Princeton, NJ 08542, *Spacecraft dynamics and control using the spacecraft control toolbox*, 2nd ed., 2003. 8

[32] J. Douglas Faires Richard L. Burden, *Numerical analysis*, 7th ed., Brooks/Cole, Pacific Grove, CA, 2001. 58, 59, 60, 61

[33] Jerry E. White Roger R. Bate, Donald D. Mueller, *Fundamentals of astrodynamics*, Dover Publications Inc., New York, NT, 1971. 43, 48, 51

[34] A.I. Solutions, *"freeflyer user's guide"*, vol. Version 4.0, March 1999. 8

[35] Bjarne Stroustrup, *The c++ programming language*, 3rd ed., Addison-Wesley, Boston, MA, 1997. 11, 62

[36] Herb Sutter, *Exceptional c++*, Addison-Wesley, Boston, MA, 2000. 62

[37] ———, *More exceptional c++*, Addison-Wesley, Boston, MA, 2002. 62

[38] David A. Vallado, *Fundamentals of astrodynamics and applications*, McGraw-Hill, New York, NY, 1997. 32, 34, 36, 48, 49, 51, 56

[39] James R. Wertz (ed.), *Spacecraft attitude determination and control*, Reidel Publishing, Hingham, MA, 1978. 22, 23, 24, 28

[40] L. Wood, *Internetworking with satellite constellations*, Ph.D. thesis, University of Surrey, June 2001. 7