A Multi-Vehicle Testbed and Interface Framework for the Development and Verification of Separated Spacecraft Control Algorithms

by

Mark Ole Hilstad

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

Author .....
Department of Aeronautics and Astronautics
24 May, 2002

Certified by ....
David W. Miller
Associate Professor
Thesis Supervisor

Accepted by ....
Wallace E. Vander Velde
Chairman, Department Committee on Graduate Students

A Multi-Vehicle Testbed and Interface Framework for the Development and Verification of Separated Spacecraft Control Algorithms

by

Mark Ole Hilstad

Submitted to the Department of Aeronautics and Astronautics
on 24 May, 2002, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

To reduce mission cost and improve spacecraft performance, the National Aeronautics and Space Administration and the United States military are considering the use of distributed spacecraft architectures in several future missions. Precise relative control of separated spacecraft position and attitude is an enabling technology for many science and defense applications that require distributed measurements, such as long-baseline interferometric arrays. The SPHERES testbed provides a low-risk, representative dynamic environment for the interactive development and verification of formation flight control and autonomy algorithms. The testbed is described, and properties relevant to control formulation such as thruster placement geometry and actuation non-linearity are discussed. A hybrid state determination methodology utilizing a memoryless attitude update and a Kalman filter for position and velocity is presented. State updates are performed based on range measurements to each vehicle from known positions on the periphery of the test volume. A high-level, modular control interface facilitates rapid test development and the efficient reuse of old code, while maintaining freedom in the design of new algorithms. A simulation created to facilitate the development of new maneuvers, tests, and control algorithms is described.

Thesis Supervisor: David W. Miller
Title: Associate Professor

Acknowledgments

This work is dedicated to my parents, Gordon and Mary Hilstad. Their love, care, and guidance have been of central importance to everything I've ever accomplished.

The SPHERES project began in a three-semester capstone design course, as part of the MIT Department of Aeronautics and Astronautics Conceive, Design, Implement, Operate (CDIO) educational initiative. The project is the result of dedicated effort by students, faculty, and staff of MIT, and staff of Payload Systems, Inc:

- MIT undergraduates: Stephanie Chen, Allen Chen, Julie Wertz, Stuart Jackson, Shannon Cheng, Fernando Perez, David Carpenter, Jason Szuminski, George Berkowski, Chad Brodel, Sarah Carlson, Bradley Pitts, and Daniel Feller.
- MIT graduates: Alvar Saenz-Otero, Allen Chen, Alice Liu, Simon Nolet, and Mitch Ingham.
- MIT faculty and staff: Prof. David W. Miller, Paul Bauer, Dr. Ray Sedwick, Dr. Edmund Kong, Dr. Jeffrey Hoffman, Prof. Jonathan How, Sharon Leah Brown, and Margaret Edwards.
- PSI staff: Steve Sell, Stephanie Chen, Dr. Javier deLuis, and Edison Guerra.

I would also like to express my appreciation to

- My advisor, Prof. David W. Miller, for granting me the opportunity to study at MIT and for trusting me with an important role in a flight project.
- My family and friends, both at MIT and far away, for their encouragement and support over the years.
- Russell Carpenter and the NASA Goddard Space Flight Center for sponsoring KC-135 flight tests.
- God, for creating this wonderous universe and giving us the drive and capability to explore.

Contents

List of Figures

List of Tables

Nomenclature

DARPA	Defense Advanced Research Projects Agency
DSP	Digital signal processor
GFLOPS	Generalized FLight Operations Processing Simulator
GPS	Global Positioning System
GSP	Guest Scientist Program
IR	Infrared
ISS	International Space Station
MIT	Massachusetts Institute of Technology
NASA	National Aeronautics and Space Administration
PADS	Position and attitude determination subsystem
PD	Proportional-derivative
PWM	Pulse-width modulation
SCI	Standard control interface
SPHERES	Synchronized Position Hold, Engage, Reorient Experimental Satellites
SSL	Space Systems Laboratory
STL	Sphere-to-laptop (radio communications channel)
STS	Sphere-to-sphere (radio communications channel)
US	Ultrasonic
USAF	United States Air Force

Mathematical Notation

- Scalar variables are represented in an italic typeface (e.g. x).
- Vector variables are represented in bold italic (e.g. $\boldsymbol{x}$).
- Matrices and tensors of second and higher order are represented as capital letters, and are not italicized (e.g. A).
- Units, such as km (kilometers) and s (seconds), are not italicized.
- Representations of the attitude quaternion appear bold, but not italicized, to distinguish them from standard vectors. A tilde is used to distinguish the hyper-imaginary representation of the quaternion $\tilde{\mathbf{q}}$ from the real representation $\mathbf{q}$.
- The symbol $\equiv$ is used to signify "is identically equal to," or "is defined as."
- The over-hat, $\hat{}$, signifies that a quantity is an estimate.

Chapter 1

Introduction

1.1 Motivation

To reduce mission cost and improve spacecraft performance, the National Aeronautics and Space Administration (NASA) and the United States military are considering the use of distributed spacecraft architectures in several future missions. Precise relative control of separated spacecraft position and attitude is an enabling technology for many science and defense applications that require distributed measurements, such as the long-baseline interferometric array proposed for the Terrestrial Planet Finder mission [2]. The Defense Advanced Research Projects Agency (DARPA) Orbital Express program will develop a fleet of satellites with the capability to dock with and re-supply or upgrade aging or damaged satellites. Routine autonomous formation flight is essential for the success of these missions.

The SPHERES testbed is intended to mitigate the risk associated with attempting autonomous separated spacecraft control, by providing a risk-tolerant medium for the development and maturation of formation flight and docking algorithms. The testbed is manifested for launch to the International Space Station (ISS) on service flight 12A.1, nominally scheduled for May, 2003. Results from this research are applicable to separated spacecraft telescopes, *in-situ* distributed space science, and autonomous docking.

1.2 Background

Relative control of separated spacecraft is termed formation flight, and may be distinguished from the concept of a satellite constellation. The difference lies in active control of the relative states of the formation flying spacecraft. As defined by NASA's Goddard Space Flight Center [11], a constellation is comprised of "two or more spacecraft in similar orbits with no active control by either [spacecraft] to maintain a relative position." Station-keeping and orbit maintenance are performed based on geocentric states, so groups of Global Positioning System (GPS) satellites or communication satellites are considered constellations. In contrast, "formation flight involves the use of an active control scheme to maintain the relative positions of the spacecraft."

A further distinction may be made between formation-keeping and formation-changing, both subsets of formation flight. Formation-keeping refers to the maintenance of a specified relative state between spacecraft in the presence of disturbances or undesired dynamic effects, while formation-changing involves the execution of a planned modification of the desired relative state, possibly leading to significant changes in the relative spacecraft dynamics. Rendezvous and docking maneuvers are another subset of formation flight, in which the distance between two or more separate vehicles is reduced until the vehicles make physical contact.

Formation flight experiments have been performed in limited degrees of freedom in ground laboratories, and several university groups are in the process of developing formation flying satellite missions. Examples of such missions are ION-F and ORION/Emerald.

- The University of Washington Dawgstar nanosatellite will perform formation flight maneuvers with nanosatellites built by Utah State University and Virginia Tech. The three satellites are collectively named the Ionospheric Observation Nanosatellite Formation (ION-F). The primary ION-F formation flight objectives are to demonstrate inter-satellite communications, autonomous formation keeping, and autonomous formation maneuvering [24].

The Dawgstar has eight micro-pulsed plasma thrusters (μPPTs), arranged to provide direct control in five degrees of freedom and indirect control in the remaining degree of freedom. The Virginia Tech HokieSat uses four μPPTs and the Utah State University USUSat uses differential drag to maintain relative position. The ION-F satellites will demonstrate leader-follower, same ground track, and formation-keeping maneuvers with separation distances ranging from three to 20 km [24].

- The primary goal of the Stanford University/Santa Clara University Emerald mission is to promote robust distributed space systems. Part of this goal is to "demonstrate and validate space-based formation flying" through "a thorough investigation of cluster management and control issues on-orbit using a simple satellite formation that can autonomously perform relative navigation as well as research into operational issues such as high-level mission specification and anomaly management [23]." The Emerald hardware consists of two 15 kg nanosatellites, to be launched from the Space Shuttle in 2003. Position determination is achieved using GPS, and formation flight maneuvers will be performed using actively-controlled drag panels [23].

  In addition, collaborative maneuvers are planned with the Stanford University/MIT ORION mission. "The Emerald satellites will provide a surrogate constellation for ORION's formation maneuvers. Together, the three satellites will demonstrate the capabilities of a multi-satellite fleet [23]." The ORION/Emerald team will demonstrate an in-track follower formation, with a separation distance between 100 and 300 m. Relative tolerance as small as 2 m will be attempted [10].

Micro-gravity dynamics cannot be reproduced in a ground laboratory, and physical and logistical constraints limit the interaction between investigators on the ground and actual satellites in orbit. Both the ground and orbital environments have fundamental limitations that impede the cost-effective development and verification of separated spacecraft control algorithms. The SPHERES testbed is designed to op-

erate inside the International Space Station, where the benefits of the controlled laboratory environment are added to those of the representative micro-gravity environment [20]. The SPHERES testbed therefore presents a unique opportunity for researchers to interactively test formation flight, rendezvous, and docking algorithms in a low-risk, representative dynamic environment.

1.3 Research Objectives

The primary objectives of the research described herein are summarized below.

- Formulate models to describe the SPHERES vehicle components, for use in control system design.
- Develop and implement a state estimator for real-time determination of the position, velocity, attitude, and angular rate of the SPHERES vehicles.
- Develop a simple, flexible interface to the SPHERES vehicle onboard hardware and software, to facilitate the use of the testbed by guest scientists who don't have immediate access to the flight hardware. This interface should:
  - allow freedom in algorithm design;
  - be simple to learn and use;
  - facilitate rapid recognition of test contents;
  - increase productivity by ensuring code reusability;
  - satisfy ISS operational and safety requirements;
  - incorporate built-in support for important tasks; and
  - hide low-level background tasks.
- Create a simulation to be used for the development of new SPHERES algorithms and maneuvers. This simulation should:
  - use actual SPHERES flight code, except in the case of functions that directly access hardware;

- simulate three-vehicle dynamics in 0-g and 1-g environments;
- simulate radio frequency communications; and
- provide feedback to be used for determination of algorithm performance.

1.4 Outline

This thesis focuses primarily on four subjects.

- Chapter 2 briefly describes the SPHERES testbed, and discusses the physical properties of the testbed that affect control law formulation, such as thruster geometry and non-linearity.
- Chapter 3 describes the approach used to determine the position, velocity, attitude, and angular rate of each sphere with respect to the testbed reference frame. The algorithms currently in use are described, and suggestions are made for future improvements.
- Chapter 4 describes a powerful high-level, modular interface that can be used to rapidly create maneuvers and tests. Examples are used to clarify the presentation.
- Chapter 5 briefly describes a simulation that will be delivered to guest scientists, for use in the development of SPHERES control algorithms.

The relationship between the thesis chapters is depicted in Figure 1-1, in the form of a control system block diagram. Chapters 2 and 4 may be thought of as producing the desired trajectory signal and the feedback gain, and Chapter 3 contributes the system dynamics and sensor feedback. The simulation described in Chapter 5 encompasses the entire system.

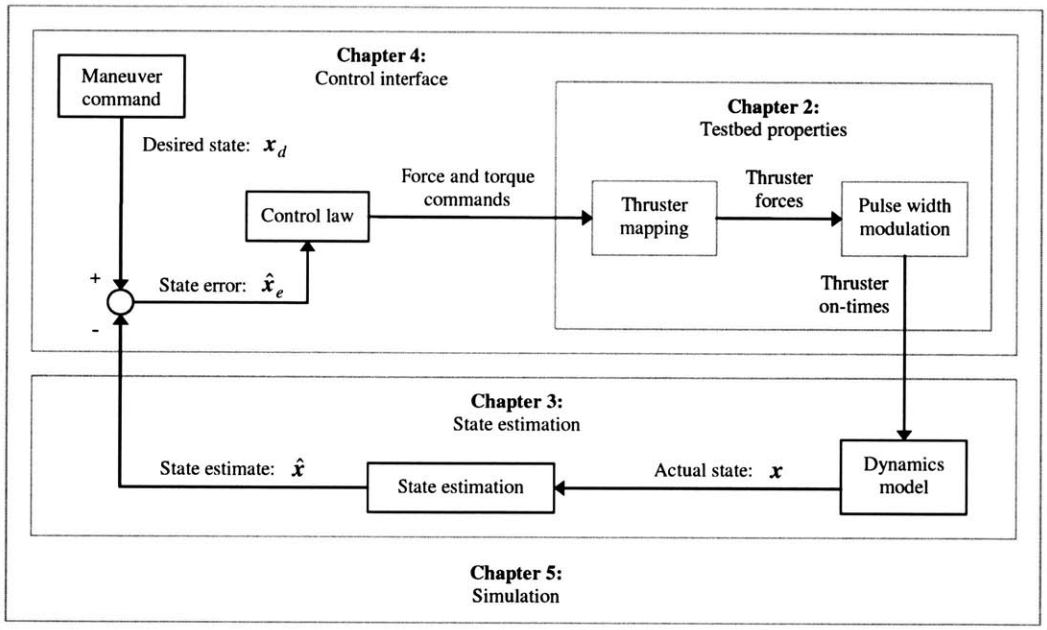


Figure 1-1: Thesis road map, showing the relationship between the thesis chapters in the form of a closed-loop control block diagram.

Chapter 2

SPHERES Testbed

2.1 Testbed Overview

The SPHERES (Synchronized Position Hold, Engage, Reorient Experimental Satellites) testbed provides a fault-tolerant development and verification environment for high-risk formation flying, rendezvous, docking, and autonomy algorithms. The testbed consists of three self-contained free-flyer vehicles ("spheres"), a laptop control station, and five small beacons that are used for position and attitude determination. The testbed is designed to test algorithms that may be used in future space missions; consequently, each individual sphere is self-contained and designed to mimic the functionality of a true satellite [21]. A CAD model of an individual flight sphere with key external features identified is shown in Figure 2-1.

2.1.1 Sphere subsystems

The elements of the sphere hardware and software are organized by function into subsystems [4, 20, 18]. The propulsion, communications, control, and position and attitude determination subsystems are directly relevant to the ability of the spheres to perform coordinated maneuvers.

The propulsion subsystem hardware consists of twelve cold-gas thrusters, a tank containing liquid CO_2 propellant, and the regulator, piping, manifolds, and valves

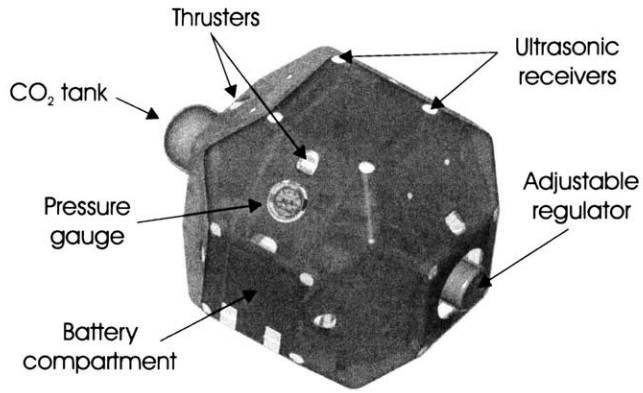


Figure 2-1: CAD model of flight sphere with important external features identified. The sphere shell measures approximately 21 cm in diameter, and the vehicle wet mass is approximately 3.6 kg [18].

required to connect the tank to the thrusters. The propulsion hardware is serviced by flight software running in a dedicated high-frequency timed interrupt process. The SPHERES propulsion system is described and characterized in detail by Chen [5].

The communications subsystem consists of two independent radio frequency channels. The sphere-to-sphere (STS) channel is used for communication between the spheres, enabling cooperative and coordinated maneuvers. The sphere-to-laptop (STL) channel is used to send command and telemetry data between the spheres and the laptop control station. The design and implementation of the SPHERES communications subsystem is described in detail by Saenz-Otero [21].

The SPHERES position and attitude determination system (PADS) provides real-time position, velocity, attitude, and angular rate information to each sphere. Inertial measurements are used to propagate the state estimate, and range measurements are used to update the state estimate with respect to the laboratory reference frame. The PADS hardware and software are detailed in Chapter 3.

The control subsystem is serviced by flight software running in a dedicated timed-interrupt process. The control subsystem produces thruster on-time commands based on a specified maneuver profile, a control law, and the thruster geometry and actuation properties. A high-level, modular interface to the control subsystem is presented in Chapter 4. The effects of thruster non-linearity are discussed in Section 2.2, and

the thruster geometry is discussed in Section 2.3.

2.2 Thruster Pulse Modulation

Most widely-used modern control design methodologies assume linear dynamics and linear, continuous, unbounded actuation. Linear, continuous momentum-transfer actuators such as reaction wheels are widely used for control of spacecraft attitude, but there are no effective linear, continuous mass-transfer actuators available for management of spacecraft position. Nonlinear, discontinuous actuators such as on-off thrusters are therefore necessary both for translation control, and for periodic desaturation of reaction wheels. The spheres rely on on-off thrusters for management of both position and attitude.

The on-off thrusters used on the spheres exhibit nonlinear, discontinuous, bounded behavior. Each thruster consists of a solenoid valve and a nozzle. When a thruster is commanded on, a voltage spike and hold circuit activates and holds open the solenoid valve. The thruster output force increases rapidly (< 1 ms rise time) from zero to the steady-state thrust following a delay, $\tau \approx 5$ ms, due to solenoid actuation dynamics [5]. The force returns to zero when the thruster is commanded off and the solenoid valve closes. A typical time-history force profile for a SPHERES cold-gas thruster is shown in Figure 2-2. Neglecting opening and closing transients, the control authority of an on-off thruster may be considered zero when closed, and equal to the nominal thrust when open. The twelve thrusters are arranged in six back-to-back pairs, allowing for both positive and negative actuation. The thruster geometry is detailed in Section 2.3.

The response of a system to unmodulated on-off actuation may be seen through the following two examples. Using on-off thrusters with a zero deadband leads to bang-bang control, in which any deviation from the nominal state is counteracted with the full available control authority. Delays in the system and the finite size of the thruster impulse bit result in over-actuation, and the system oscillates in the phase plane about the switching line. Bang-bang control results in a 100% duty cycle

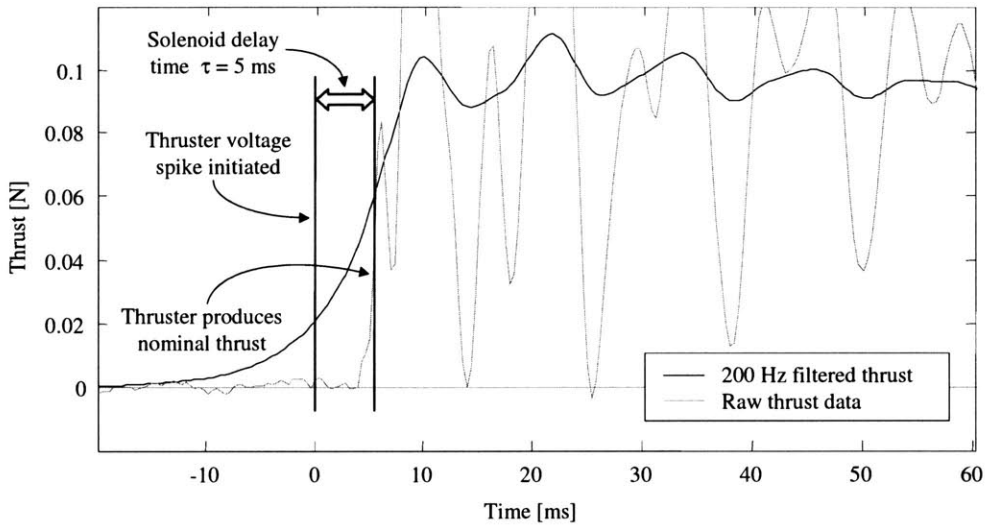


Figure 2-2: The force profile during the opening transient of a typical SPHERES cold-gas thruster. High-magnitude oscillation appears in the raw force data because of natural frequencies in the test stand at approximately 30, 90, and 150 Hz. A bi-directional zero-phase filter at 200 Hz was used to produce the dark line. A feed pressure of 32.25 psig was used for this test, resulting in a steady-state thruster force of 0.097 N. The maximum feed pressure of 55 psig results in a steady-state thruster force of approximately 1.7 N [5].

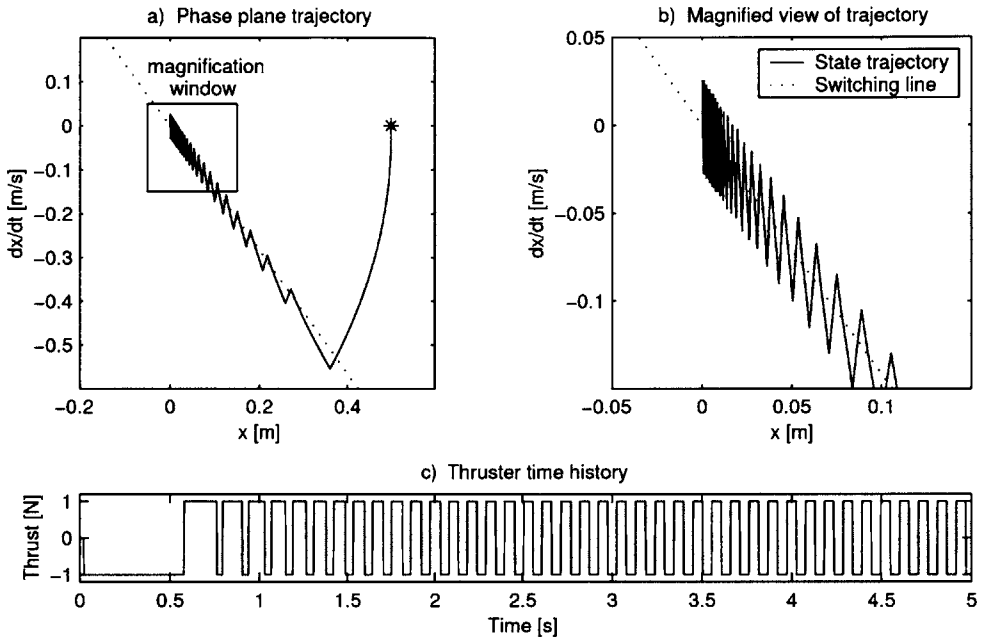


Figure 2-3: Bang-bang switching (zero deadband). a) Zero deadband leads to rapid oscillation in the phase plane about the switching line. b) Magnified view of oscillation. c) Thruster time history shows 100% duty cycle for a thruster pair.

in each thruster pair (i.e. six thrusters are open at any given time). This chattering behavior may be shown for a double integrator plant with proportional-derivative feedback control and bang-bang actuators using the equation of motion

$$\begin{aligned} m\ddot{x}(t) &= u(t) \\ &= -\text{sign}(\zeta\omega_n\dot{x}(t-\tau_0) + \omega_n^2 x(t-\tau_0)) \end{aligned} \tag{2.1}$$

with damping ratio $\zeta = \sqrt{2}/2$, natural frequency $\omega_n = 2\,\text{rad/s}$, mass $m = 1\,\text{kg}$, and feedback time delay $\tau_0 = 25\,\text{ms}$. Initial conditions are $x_0 = 0.5\,\text{m}$, $\dot{x}_0 = 0\,\text{m/s}$. The chattering behavior of this system is shown in Figure 2-3

With a non-zero deadband, two switching lines border a negatively-sloped band in the phase plane in which actuation ceases and the spacecraft trajectory is determined by the unforced equations of motion. This behavior, termed bang-off-bang control,

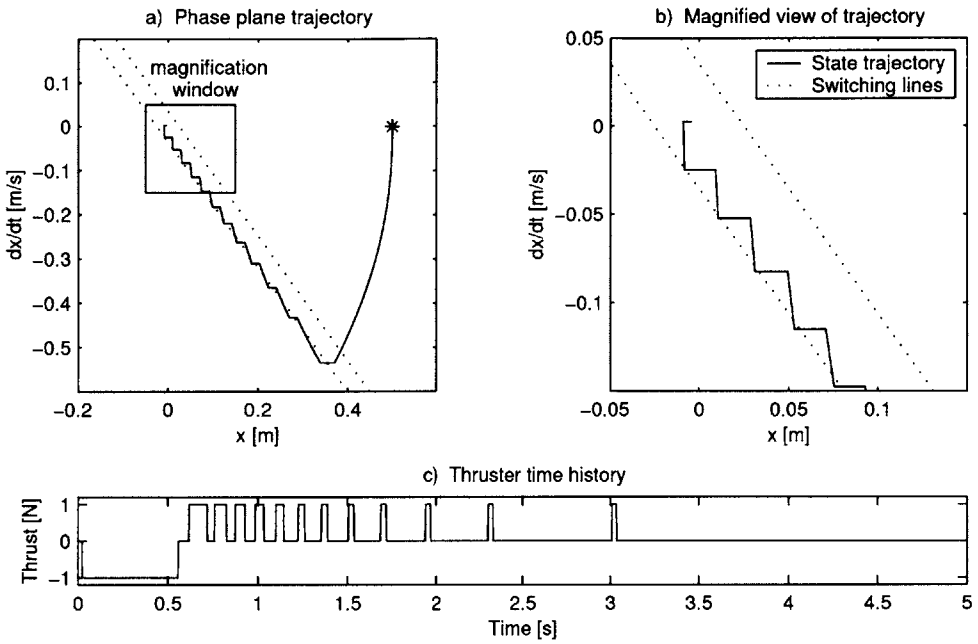


Figure 2-4: Bang-off-bang switching (non-zero deadband). a) State dynamics alternate between forced and unforced equations of motion. b) Magnified view of switching line behavior shows forced motion and free drift. c) Thruster time history shows about 22% duty cycle over the simulation time.

can be demonstrated with the equation of motion

$$m\ddot{x}(t) = \begin{cases} -\text{sign}(\zeta\omega_n\dot{x}(t-\tau_0) + \omega_n^2 x(t-\tau_0)) & , \ |\zeta\omega_n\dot{x}(t-\tau_0) + \omega_n^2 x(t-\tau_0)| \geq 0.1 \\ 0 & , \ \text{otherwise} \end{cases} \tag{2.2}$$

The behavior of this system is shown in the phase-plane diagram of Figure 2-4. With a non-zero deadband, oscillation in the phase plane results in a duty cycle of less than one, and the duty cycle decreases with increasing deadband in most disturbance environments.

Bang-bang and bang-off-bang actuation are neither linear nor continuous, and the control authority in each case is equal to and therefore bounded by the force of the open thruster. Several methods of overcoming the problem of thruster nonlinearity are used in practice, such as pulse width modulation (PWM), pulse frequency modulation, pulse width pulse frequency modulation, and derived rate modulation [7].

These four methods are based on the assumption that nonlinearity and discontinuity occurring on small time scales may be averaged out, leading to effectively linear, continuous actuation over longer time scales. In pulse width modulation, the pulse frequency is held constant, and pulses of varying width (temporal duration) begin at fixed time intervals. The amplitude of the input to the modulator determines the width of each pulse. In pulse frequency modulation, the pulse width is constant, and the amplitude of the input determines the pulse frequency. Pulse width pulse frequency and derived rate modulation incorporate elements of both pulse width and pulse frequency modulation [7]. The fixed-frequency nature of pulse width modulation makes it a natural choice for use with a clocked digital control computer such as that used in the SPHERES vehicles, so therefore only pulse width modulation is given further consideration here.

Pulse modulation schemes may be tailored to meet particular performance requirements related to deadband and propellant expenditure. The SPHERES thruster pulse width modulation scheme is designed to approximate the action of a linear actuator over the linear range of the time integral of the thrust profile. The lower end of this range corresponds to the minimum useful impulse bit, and the upper end corresponds to the impulse of a thruster held open over the modulation period. Given the steady state thruster force φ_i and thruster on-time u_i of thruster i, the modulation period p, and the solenoid actuation delay time τ, a piecewise linear relationship between the resultant force f_i and the on-time u_i may be written as

$$f_i = \begin{cases} \varphi_i \left(\frac{p - \tau}{p} \right) & , \quad u_i \geq p \\ \varphi_i \left(\frac{u_i - \tau}{p} \right) & , \quad \tau < u_i < p \\ 0 & , \quad 0 < u_i < \tau \end{cases} \tag{2.3}$$

This relationship holds with constant non-zero delay time τ only when the thruster is turned off at some point during each modulation period. If the thruster is kept open throughout the period, the value of τ over the next modulation period will be zero for that thruster, as no actuation delay is incurred when the solenoid valve is

already open. Equation 2.3 may be solved for the on-time u_i to produce a mapping from commanded force to thruster on-time in the presence of actuation delay.

$$u_i = \begin{cases} p & , \quad f_i \geq \varphi_i \left(\frac{p-\tau}{p} \right) \\ \frac{f_i}{\varphi_i} p + \tau & , \quad 0 < f_i < \varphi_i \left(\frac{p-\tau}{p} \right) \\ 0 & , \quad f_i = 0 \end{cases} \tag{2.4}$$

The pulse modulation relationships of Equations 2.3 and 2.4 result in bang-bang actuation. In order to reduce the modulation factor, a deadband term may be specified on either the commanded thruster force f_i or on the on-time u_i. Specifying a force deadband f_0, the pulse modulation relationship may be written as

$$u_i = \begin{cases} p & , \quad f_i \geq \varphi_i \left(\frac{p-\tau}{p} \right) \\ \frac{f_i}{\varphi_i} p + \tau & , \quad f_0 < f_i < \varphi_i \left(\frac{p-\tau}{p} \right) \\ 0 & , \quad f_i < f_0 \end{cases} \tag{2.5}$$

The appropriate on-time u_i for each thruster i may be determined from Equation 2.5 from the commanded thruster force f_i and the actual steady-state force φ_i of that thruster. The pulse modulation curves of Equations 2.4 and 2.5 are depicted graphically in Figure 2-5. Note that the maximum equivalent force over a modulation period is reduced from that of the steady-state thrust φ_i by a factor of $\frac{p-\tau}{p}$ when the solenoid delay τ is greater than zero (i.e. when the thruster is not already open) during a given modulation period.

2.3 Thruster Geometry

Two versions exist of the sphere vehicle hardware. The prototype sphere, designed by a class of MIT undergraduate students, has been tested extensively in the laboratory and on NASA's KC-135 micro-gravity aircraft. The flight hardware sphere design differs from the prototype design in several ways that affect state estimation and control law formulation. The principle geometric changes involve the design of the

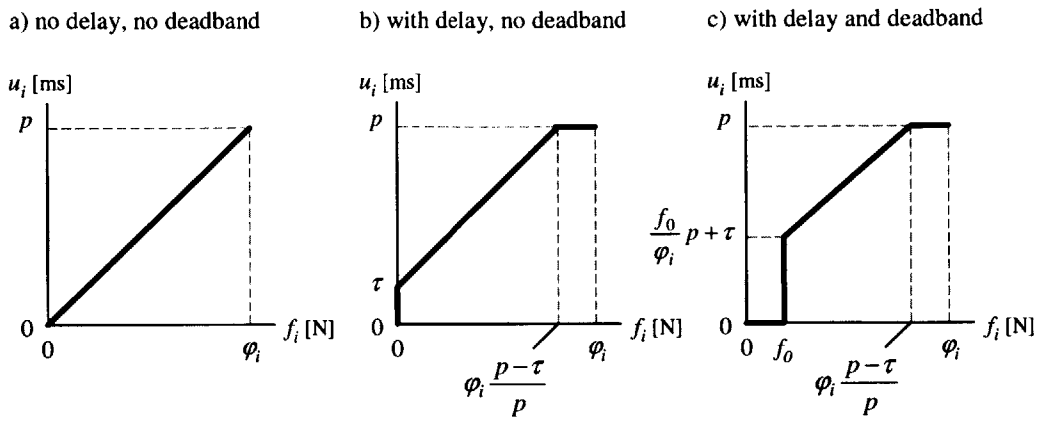


Figure 2-5: Pulse modulation curves, mapping commanded thruster force to equivalent on-time. a) Assuming thruster opens without delay. b) Accounting for solenoid delay time τ. c) With delay time τ and force deadband f_0.

load-bearing structure, the shape and construction of the outer shell, the placement of the cold-gas thrusters, and the number of ultrasonic receivers on each face.

2.3.1 Thruster placement and mixing matrix - prototype

The prototype sphere thruster configuration was designed with back-to-back mounted thruster pairs to simplify routing of internal wiring and tubing. The thruster pairs are mounted on sloped panels rather than on the six sides that are aligned with the body axes, due to placement constraints imposed by the propulsion and ultrasonic ranging systems. Figure 2-6 shows the prototype thruster geometry, and Table 2.1 lists thruster positions and resultant force and torque directions.

Given the prototype sphere thruster force and torque direction properties in Table 2.1, it is possible to determine the combination of thrusters required to produce force along or torque about any body axis. Table 2.2 lists the thrusters required to produce pure body-axis actuation, assuming that all thrusters produce equal magnitude force and that all moment arms are equal.

Since a single thruster can produce force in only one direction, two axially aligned, opposing thrusters are required to enable positive and negative force along a given axis. In the prototype sphere geometry, each set of opposing thrusters constitutes one

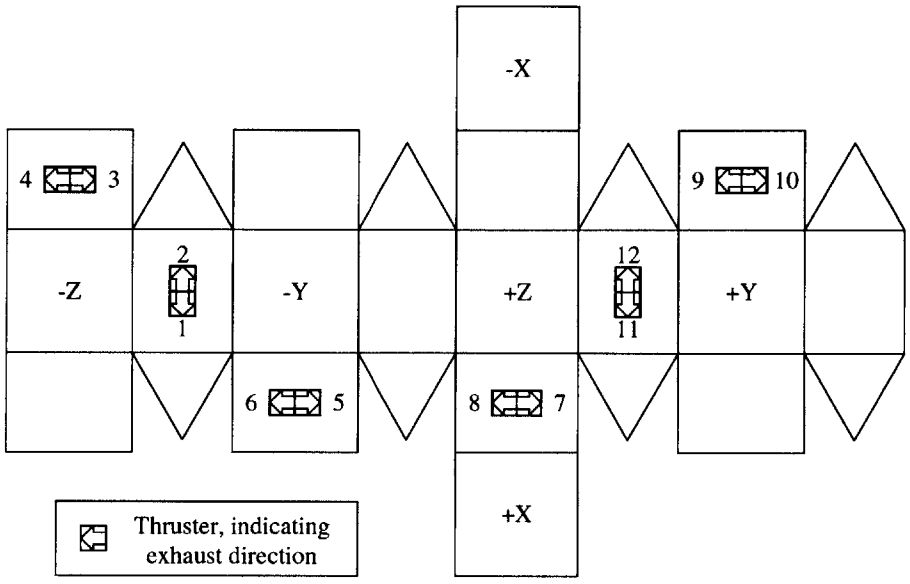


Figure 2-6: Prototype sphere thruster geometry. This geometry allows for pure body-axis force with only two thrusters, but six thrusters are required to produce pure body-axis torque.

Table 2.1: Prototype sphere thruster geometry. Thruster positions are given in units of centimeters, and force and torque directions are unitless.

Thr #	Thruster position [cm]			Resultant force direction			Resultant torque direction		
	x	y	z	x	y	z	x	y	z
1	1.9	−8.0	−8.0	−1	0	0	0	0.707	−0.707
2	−1.9	−8.0	−8.0	1	0	0	0	−0.707	0.707
3	−8.0	−1.9	−8.0	0	1	0	0.707	0	−0.707
4	−8.0	1.9	−8.0	0	−1	0	−0.707	0	0.707
5	8.0	−8.0	1.9	0	0	−1	0.707	0.707	0
6	8.0	−8.0	−1.9	0	0	1	−0.707	−0.707	0
7	8.0	1.9	8.0	0	−1	0	0.707	0	−0.707
8	8.0	−1.9	8.0	0	1	0	−0.707	0	0.707
9	−8.0	8.0	1.9	0	0	−1	−0.707	−0.707	0
10	−8.0	8.0	−1.9	0	0	1	0.707	0.707	0
11	1.9	8.0	8.0	−1	0	0	0	−0.707	0.707
12	−1.9	8.0	8.0	1	0	0	0	0.707	−0.707

Table 2.2: Prototype sphere thruster combinations to produce pure body-axis force and pure body-axis torque. To produce pure body-axis force or torque about a particular axis, fire the thrusters in the corresponding column.

Thr #	Body-axis force						Body-axis torque					
	$+x$	$-x$	$+y$	$-y$	$+z$	$-z$	$+x$	$-x$	$+y$	$-y$	$+z$	$-z$
1		×						×	×			×
2	×						×			×	×	
3			×				×			×		×
4				×				×	×		×	
5						×	×		×		×	
6					×			×		×		×
7				×			×			×		×
8			×					×	×		×	
9						×		×		×		×
10					×		×		×		×	
11		×					×			×	×	
12	×							×	×			×

of the back-to-back mounted thruster pairs. For simplicity in the following derivation, it is desirable to treat each thruster pair as a single unit, capable of positive and negative actuation. Given the force magnitude $f_{i(p)}$ of an odd-numbered thruster i in the prototype geometry, the resultant force of the thruster pair i, $i+1$ is the difference between the positive forces $f_{i(p)}$ and $f_{i+1(p)}$. The resultant forces for the six thruster pairs are

$$f_{1,2(p)} \equiv f_{1(p)} - f_{2(p)} \tag{2.6}$$

$$f_{3,4(p)} \equiv f_{3(p)} - f_{4(p)} \tag{2.7}$$

$$f_{5,6(p)} \equiv f_{5(p)} - f_{6(p)} \tag{2.8}$$

$$f_{7,8(p)} \equiv f_{7(p)} - f_{8(p)} \tag{2.9}$$

$$f_{9,10(p)} \equiv f_{9(p)} - f_{10(p)} \tag{2.10}$$

$$f_{11,12(p)} \equiv f_{11(p)} - f_{12(p)} \tag{2.11}$$

A positive value for $f_{i,i+1(p)}$ therefore corresponds to thruster i, and a negative value corresponds to thruster $i+1$. Based on the definitions of Equations 2.6

through 2.11, it is possible to construct a matrix $\mathrm{M}_{(p)}^{-1}$ that maps thruster pair forces into body-frame force and torque commands for the prototype geometry. Given force and torque commands $\boldsymbol{f} = [\ f_x \quad f_y \quad f_z\ ]^T$ and $\boldsymbol{m} = [\ m_x \quad m_y \quad m_z\ ]^T$, respectively, and assuming that all thrusters have equal moment arm r,

$$\begin{bmatrix} f_x \\ f_y \\ f_z \\ \frac{m_x}{r} \\ \frac{m_y}{r} \\ \frac{m_z}{r} \end{bmatrix} = \mathrm{M}_{(p)}^{-1} \begin{bmatrix} f_{1,2(p)} \\ f_{3,4(p)} \\ f_{5,6(p)} \\ f_{7,8(p)} \\ f_{9,10(p)} \\ f_{11,12(p)} \end{bmatrix} \tag{2.12}$$

where $\mathrm{M}_{(p)}^{-1}$ is defined using Table 2-6 as

$$\mathrm{M}_{(p)}^{-1} = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 1 & -1 & 0 \\ 1 & 0 & 1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 & 0 & 1 \end{bmatrix} \tag{2.13}$$

This matrix $\mathrm{M}_{(p)}^{-1}$ may be inverted to find $\mathrm{M}_{(p)}$, the mapping matrix from force and torque commands to thruster pair forces in the prototype sphere geometry.

$$\mathrm{M}_{(p)} = \begin{bmatrix} -\frac{1}{2} & 0 & 0 & -\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} \\ 0 & \frac{1}{2} & 0 & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ 0 & 0 & -\frac{1}{2} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ 0 & -\frac{1}{2} & 0 & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ 0 & 0 & -\frac{1}{2} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{2} & 0 & 0 & \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} \end{bmatrix} \tag{2.14}$$

Force and torque commands may be transformed to thruster pair forces as

$$\begin{bmatrix} f_{1,2(p)} \\ f_{3,4(p)} \\ f_{5,6(p)} \\ f_{7,8(p)} \\ f_{9,10(p)} \\ f_{11,12(p)} \end{bmatrix} = \mathrm{M}_{(p)} \begin{bmatrix} f_x \\ f_y \\ f_z \\ \frac{m_x}{r} \\ \frac{m_y}{r} \\ \frac{m_z}{r} \end{bmatrix} \tag{2.15}$$

With the thruster pair forces $f_{i,i+1(p)}$ known, the individual thruster commands $f_{i(p)}$ and $f_{i+1(p)}$ may be determined. Because the SPHERES thrusters are on-off actuators, a pulse modulation scheme such as that described in Section 2.2 must be applied to determine the relationship between commanded force and on-time for each thruster.

2.3.2 Thruster placement and mixing matrix - flight

The thruster placement geometry of the ISS flight hardware differs from that of the sphere prototype. The flight thruster geometry can produce pure body-axis force or torque using only two thrusters, while production of pure body-axis torque in the prototype geometry requires six thrusters. The flight sphere thruster configuration is shown in Figure 2-7, and the corresponding thruster force and torque direction properties are given in Table 2.3.

Using Table 2.3, it is possible to determine the combination of thrusters required to produce force along or torque about any body axis. Table 2.4 lists the thrusters required to produce pure body-axis actuation, assuming that all thrusters produce equal magnitude force and that all moment arms are equal.

As can be seen from Figure 2-7 and Table 2.3, the flight geometry opposing

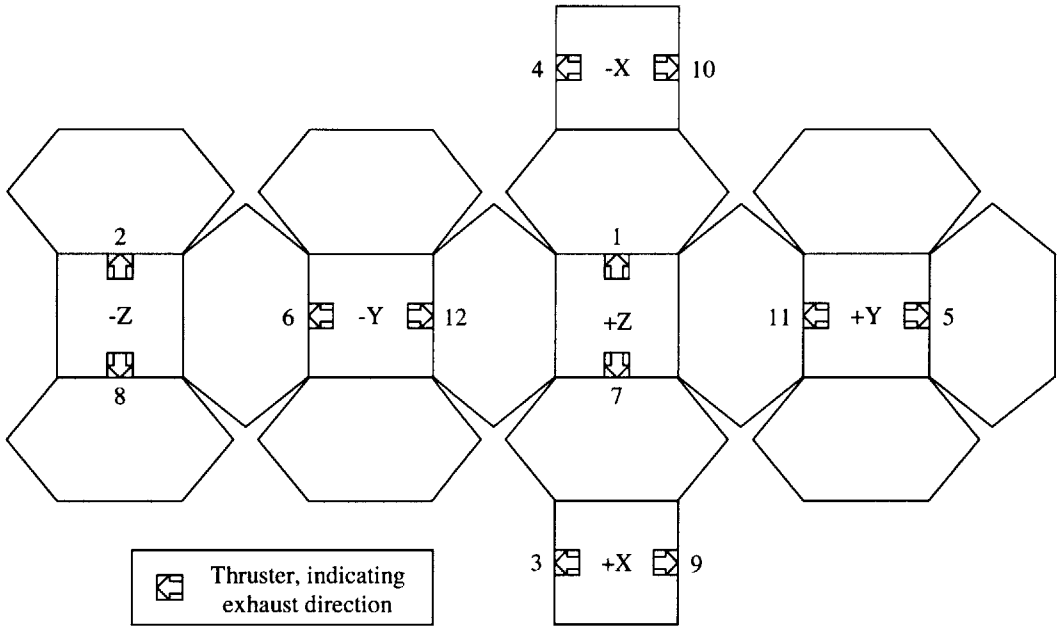


Figure 2-7: Flight sphere thruster geometry. This geometry allows for pure body-axis force or torque using only two thrusters.

Table 2.3: Flight sphere thruster geometry. Thruster positions are given in units of centimeters, and force and torque directions are unitless.

Thr #	Thruster position [cm]			Resultant force direction			Resultant torque direction		
	x	y	z	x	y	z	x	y	z
1	−5.2	0.0	9.7	1	0	0	0	1	0
2	−5.2	0.0	−9.7	1	0	0	0	−1	0
3	9.7	−5.2	0.0	0	1	0	0	0	1
4	−9.7	−5.2	0.0	0	1	0	0	0	−1
5	0.0	9.7	−5.2	0	0	1	1	0	0
6	0.0	−9.7	−5.2	0	0	1	−1	0	0
7	5.2	0.0	9.7	−1	0	0	0	−1	0
8	5.2	0.0	−9.7	−1	0	0	0	1	0
9	9.7	5.2	0.0	0	−1	0	0	0	−1
10	−9.7	5.2	0.0	0	−1	0	0	0	1
11	0.0	9.7	5.2	0	0	−1	−1	0	0
12	0.0	−9.7	5.2	0	0	−1	1	0	0

Table 2.4: Flight sphere thruster combinations to produce pure body-axis force and pure body-axis torque. To produce pure body-axis force or torque about a particular axis, fire the thrusters in the corresponding column.

Thr #	Body-axis force						Body-axis torque					
	$+x$	$-x$	$+y$	$-y$	$+z$	$-z$	$+x$	$-x$	$+y$	$-y$	$+z$	$-z$
1	×								×			
2	×									×		
3			×								×	
4			×									×
5					×		×					
6					×			×				
7		×								×		
8		×							×			
9				×								×
10				×							×	
11						×		×				
12						×	×					

thruster pair resultant forces are

$$f_{1,7(f)} \equiv f_{1(f)} - f_{7(f)} \tag{2.16}$$

$$f_{2,8(f)} \equiv f_{2(f)} - f_{8(f)} \tag{2.17}$$

$$f_{3,9(f)} \equiv f_{3(f)} - f_{9(f)} \tag{2.18}$$

$$f_{4,10(f)} \equiv f_{4(f)} - f_{10(f)} \tag{2.19}$$

$$f_{5,11(f)} \equiv f_{5(f)} - f_{11(f)} \tag{2.20}$$

$$f_{6,12(f)} \equiv f_{6(f)} - f_{12(f)} \tag{2.21}$$

Using the definitions of Equations 2.16 through 2.21, and assuming that all thrusters have moment arm r, the matrix $\mathrm{M}_{(f)}^{-1}$ that maps thruster pair forces into body-

frame force and torque commands for the flight geometry is

$$\begin{bmatrix} f_x \\ f_y \\ f_z \\ \frac{m_x}{r} \\ \frac{m_y}{r} \\ \frac{m_z}{r} \end{bmatrix} = \mathrm{M}_{(f)}^{-1} \begin{bmatrix} f_{1,7(f)} \\ f_{2,8(f)} \\ f_{3,9(f)} \\ f_{4,10(f)} \\ f_{5,11(f)} \\ f_{6,12(f)} \end{bmatrix} \tag{2.22}$$

where $\mathrm{M}_{(f)}^{-1}$ is defined using Table 2.3 as

$$\mathrm{M}_{(f)}^{-1} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \end{bmatrix} \tag{2.23}$$

The mapping matrix $\mathrm{M}_{(f)}$ from force and torque commands to thruster pair forces in the flight sphere geometry is therefore

$$\mathrm{M}_{(f)} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & 0 & 0 & -\frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & 0 & 0 & -\frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \end{bmatrix} \tag{2.24}$$

Force and torque commands may be transformed to thruster pair forces as

$$\begin{bmatrix} f_{1,7(f)} \\ f_{2,8(f)} \\ f_{3,9(f)} \\ f_{4,10(f)} \\ f_{5,11(f)} \\ f_{6,12(f)} \end{bmatrix} = \mathrm{M}_{(f)} \begin{bmatrix} f_x \\ f_y \\ f_z \\ \frac{m_x}{r} \\ \frac{m_y}{r} \\ \frac{m_z}{r} \end{bmatrix} \tag{2.25}$$

With the thruster pair forces $f_{i,i+6(f)}$ known, the individual thruster commands may be determined.

2.3.3 Body-axis actuation efficiency

It is expected that the majority of maneuvers will involve primarily body-axis rotations, and the flight geometry is significantly more propellant-efficient than the prototype geometry for this type of maneuver. For example, it is apparent from Table 2.2 that body z-axis torque is produced in the prototype geometry by simultaneously firing the six thrusters numbered 2, 4, 5, 8, 10, and 11. Given the prototype geometry thruster moment arm $r_p = 8.0\,\mathrm{cm}$, and assuming the thruster force magnitude $\varphi = 0.15\,\mathrm{N}$, the resultant torque is

$$\begin{aligned} \boldsymbol{m}_{z(p)} &= \varphi r_p \left(\begin{bmatrix} 0 \\ \frac{-\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix} + \begin{bmatrix} \frac{-\sqrt{2}}{2} \\ 0 \\ \frac{-\sqrt{2}}{2} \end{bmatrix} + \begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \\ 0 \end{bmatrix} + \begin{bmatrix} \frac{-\sqrt{2}}{2} \\ 0 \\ \frac{\sqrt{2}}{2} \end{bmatrix} + \begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{-\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} \end{bmatrix} \right) \\ &= \varphi r_p \begin{bmatrix} 0 \\ 0 \\ \sqrt{2} \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 0 \\ 1.7 \end{bmatrix} \mathrm{N\,cm} \end{aligned} \tag{2.26}$$

From Table 2.4 it can be seen that only two thrusters, numbers 3 and 10, must

be simultaneously fired to produce body z-axis torque in the flight geometry. Given the flight geometry thruster moment arm $r_f = 9.7\,\text{cm}$, the resultant torque is

$$\begin{aligned} \boldsymbol{m}_{z(f)} &= \varphi r_f \left(\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right) \\ &= \begin{bmatrix} 0 \\ 0 \\ 2.9 \end{bmatrix} \text{N cm} \end{aligned} \tag{2.27}$$

The torque capability about each body axis is therefore increased in the flight geometry with respect to the prototype geometry by a factor of $\frac{2.9}{1.7} = 1.7$, while the propellant expended to produce that torque is reduced by a factor of $\frac{6}{2} = 3$. The flight geometry is therefore $1.7 \times 3 = 5.1$ times more propellant efficient than the prototype geometry in the production of body-axis torque. In addition, the flight thruster geometry is easier to visualize than the prototype geometry, simplifying the analysis of thruster status lights during development and debug operations.

The flight and prototype configurations have identical body-axis force capability, and for a center of mass located at the body frame origin, the two configurations give identical body-axis translation performance. In reality, the sphere center of mass is offset slightly from the origin, and body-axis translation based on the body-axis actuation guidelines in Tables 2.2 or 2.4 will produce an undesired torque. The magnitude of this undesired torque can be reduced by scaling the thruster on-times based on the location of the sphere center of mass. When using this approach, the larger separation distance between thrusters in the flight geometry than in the prototype geometry enables more precise control over the elimination of unwanted torque.

Chapter 3

State Determination

3.1 Overview

State determination involves creating an estimate of the current position, velocity, attitude, and angular rate of each sphere. The hardware and software that perform this function on the SPHERES testbed are collectively termed the Position and Attitude Determination System (PADS). This chapter describes the hardware and algorithms used to determine the state estimate of each sphere. The attitude is calculated using a memoryless least-squares optimal quaternion algorithm, and collections of range measurements from which common-mode errors have been removed. Position and velocity are updated using a Kalman filter acting on modified range measurements, which are corrected to account for nonzero measurement bias errors.

3.2 PADS hardware

The SPHERES PADS provides real-time state (position, velocity, attitude, and angular rate) information to each vehicle through the fusion of information from two distinct, asynchronous local and global elements. Each sphere contains an independent local element, consisting of three accelerometers and three rate gyroscopes. These inertial sensors are used to propagate the state estimate of each sphere over time, based on inertial measurements. The global element provides measurements of

each sphere state with respect to the global (laboratory) reference frame, in the form of range measurements to points on each sphere from five external beacons mounted at known locations on the periphery of the test volume. The local element is used to propagate the state estimate at a high constant rate, and the state estimate is periodically updated using the global element measurements at a lower, possibly variable rate.

3.2.1 PADS local element

The PADS local element aboard each sphere consists of six instrument-grade inertial sensors. Three Honeywell QA-T160 single-axis accelerometers are used to measure linear acceleration. The accelerometer resolution of $< 5\,\mu$g is sufficient to provide high signal-to-noise ratio measurements for state propagation and system identification [13]. Three micro-machined, solid-state rate gyros are used to measure angular rate. The BEI Gyrochip II by Systron Donner measures angular rates in the range of $\pm 50°/s$ using a vibrating quartz tuning fork sensing element [3].

Ideally, the accelerometers would be mounted along the three axes of the sphere body frame, but this ideal mounting arrangement is not feasible given the spatial requirements of other subsystems. The accelerometers are therefore aligned parallel to the body axes, but mounted at various spatially distributed locations as space permits inside the sphere. The measured acceleration component $\dot{\boldsymbol{v}}_{\omega}(\boldsymbol{\omega})$ that occurs as a result of nonzero angular rate, is accounted for in the estimation algorithm. The gyroscopes are mounted in alignment with the body axes, and all inertial sensors are located on or near circuit boards to minimize electrical line noise [18].

3.2.2 PADS global element

The PADS global element performs a function similar to that of the Global Positioning System (GPS), by providing measurements of the sphere state with respect to a reference frame. These measurements take the form of ultrasound times of flight to the sphere from five external ultrasound beacons that are mounted at known locations

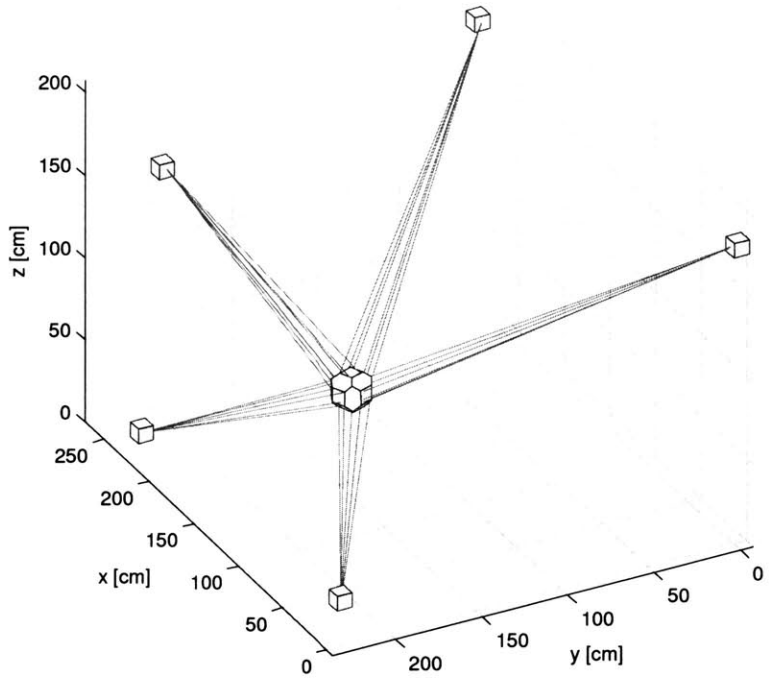


Figure 3-1: A diagram of the PADS global element. Range measurements are taken from the ultrasonic beacons mounted on the periphery of the test volume to 24 ultrasonic receivers mounted on the surface of the sphere. Direct range measurements made between the individual sphere vehicles are not shown in this diagram.

and orientations on the periphery of the test volume. These times of flight are then converted to range measurements using the speed of sound, and used to determine the position and attitude of each sphere with respect to the global reference frame. The range measurements are shown in Figure 3-1 as lines between the beacon transmitters and the ultrasound receivers mounted on the sphere faces. Each sphere also has a single ultrasonic transmitter on one of the faces that may be used for direct inter-sphere ranging.

The local element is used to propagate the state estimate, and the global element measurements are used to update the estimate at a variable rate of between zero and 8 Hz, with a default rate of 1 Hz. To request a "global update," a sphere designated as the PADS master flashes an omni-directional infrared synchronization signal. This infrared signal is received by all the spheres and by the transmitter bea-

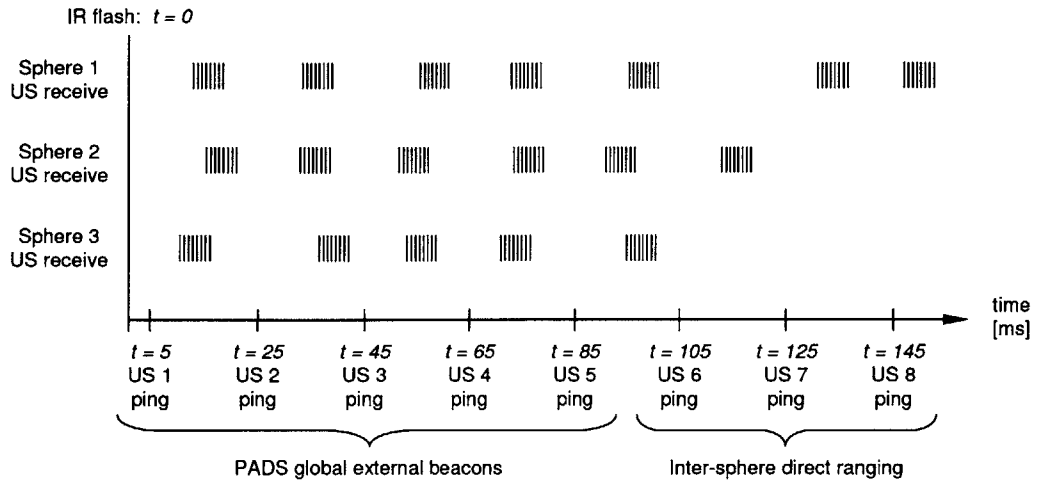


Figure 3-2: The PADS global element timing sequence. The ultrasonic receive times for each beacon/sphere pair are illustrated as series of lines rather than as isolated events to represent reception of each ultrasonic transmission by multiple sensors on each sphere. In this example, the direct-ranging transmitters on spheres 2 and 3 are pointed towards sphere 1, and the transmitter on sphere 1 is directed towards sphere 2.

cons. In response to the infrared signal, each beacon waits a specified time and then transmits a short ultrasonic pulse train. The ultrasonic pulse trains are detected by the ultrasonic receivers on each sphere using threshold detection, and times of flight are computed based on the difference in time between reception of the infrared and ultrasonic transmissions at each sphere receiver. The timeline of a global update is shown in Figure 3-2.

Each sphere has 24 ultrasonic receivers, distributed four per face on each of six faces. Due to signal attenuation from body blockage and atmospheric effects, a particular beacon signal will generally be seen by the receivers on a maximum of three of the six faces. The timing structure is designed to minimize or eliminate the occurrence of anomalous measurements produced by echoes off the walls of the test area.

The following sections detail the algorithms used by the SPHERES onboard software to determine the state estimate based on local and global measurements.

3.3 Attitude Determination

The orientation of a rigid body with respect to a reference coordinate frame may be parameterized in several ways, such as with a direction cosine (rotation) matrix, an Euler axis and angle, a quaternion, a Gibbs vector, or Euler angles. Of these parameterizations, only the direction cosine matrix and the quaternion are nonsingular for all rotations [26].

The direction cosine matrix Θ transforms any vector $\boldsymbol{\nu}$ in the reference frame to the equivalent vector $\boldsymbol{\mu}$ represented in the body frame. The primary disadvantages to the rotation matrix parameterization for attitude determination are the inclusion of six redundant parameters and the difficulty involved with normalizing the matrix after successive frame rotations [16].

The four-element attitude quaternion is non-singular, contains only one redundant parameter, is easily normalized, and has simple rules for successive rotations. In addition, there are several well-tested algorithms readily available for determination of the optimal attitude quaternion based on vector attitude measurements [16]. The quaternion is therefore used to parameterize attitude in the SPHERES testbed. Appendix A contains an overview of quaternion mathematics and the conventions used in the following discussion and in the SPHERES software.

3.3.1 Problem formulation

Most well-known algorithms for determining the optimal attitude given over-determined or noisy measurements solve Wahba's problem [25]. Wahba posed the question of how to determine the orthogonal matrix Θ with a determinant equal to one (i.e. a rotation matrix) that minimizes the cost function

$$J(\Theta) = \frac{1}{2}\sum_i \alpha_i \left|\boldsymbol{\mu}_i - \Theta\boldsymbol{\nu}_i\right|^2 \tag{3.1}$$

The measurement model is given by

$$\boldsymbol{\mu} = \Theta\boldsymbol{\nu} \tag{3.2}$$

for arbitrary pairs of noiseless physically equivalent vectors $\boldsymbol{\mu}$ and $\boldsymbol{\nu}$, measured in the body and reference frames, respectively. The scalar non-negative weights α_i are assumed to be unity in Wahba's original cost function, implying equally reliable measurements [16]. The weights may instead be chosen as inverse variances $\alpha_i = \sigma_i^{-2}$ to account for differences in measurement validity and to relate the problem to weighted least squares and maximum likelihood estimation [16, 22]. Inverse variance weighting is not currently implemented in the SPHERES attitude determination algorithm, but may be in the future.

The matrix Θ that rotates a vector from the reference frame into the body frame can be written in terms of the four-element attitude quaternion $\mathbf{q} = [\boldsymbol{q}\ q_4]^T = [q_1\ q_2\ q_3\ q_4]^T$ as

$$\Theta(\mathbf{q}) = \left(q_4^2 - \boldsymbol{q}^T\boldsymbol{q}\right) \mathrm{I}_{3\times 3} + 2\boldsymbol{q}\boldsymbol{q}^T - 2q_4\,[\boldsymbol{q}\times] \tag{3.3}$$

using shorthand notation based on the cross product operator [6]. The cross product $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$ is expressed as $\boldsymbol{c} = [\boldsymbol{a}\times]\boldsymbol{b}$ for the matrix $[\boldsymbol{a}\times]$ defined as

$$[\boldsymbol{a}\times] \equiv \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \tag{3.4}$$

The reference to body frame rotation matrix expanded in terms of the quaternion elements is

$$\Theta(\mathbf{q}) = \begin{bmatrix} q_1^2 - q_2^2 - q_3^2 + q_4^2 & 2(q_1q_2 + q_3q_4) & 2(q_1q_3 - q_2q_4) \\ 2(q_1q_2 - q_3q_4) & -q_1^2 + q_2^2 - q_3^2 + q_4^2 & 2(q_2q_3 + q_1q_4) \\ 2(q_1q_3 + q_2q_4) & 2(q_2q_3 - q_1q_4) & -q_1^2 - q_2^2 + q_3^2 + q_4^2 \end{bmatrix} \tag{3.5}$$

Table 3.1: Quantities used in attitude determination.

Name	Frame	Description
$\mathbf{q}$	Global	Orientation of body frame with respect to global frame
Θ	Global	Rotates global frame into body frame
$\boldsymbol{\nu}_{ij}$	Global	Negative wavefront unit normal for face i, beacon j
$\boldsymbol{v}_{ij}$	Global	Face i to beacon j vector
$\boldsymbol{\tau}_j$	Global	Beacon (transmitter) j unit normal
$\boldsymbol{t}_j$	Global	Beacon (transmitter) j position
$\boldsymbol{r}$	Global	Position of sphere body frame origin
$\boldsymbol{\mu}_{ij}$	Body	Negative wavefront unit normal for face i, beacon j
$\boldsymbol{s}_i$	Body	Vector from origin to center of side i
$\boldsymbol{n}_{ij}$	Sensor	Negative wavefront unit normal for face i, beacon j
$\boldsymbol{p}_i$	Sensor	Sensor plane i unit normal
Γ_i	Sensor	Rotates sensor frame i into body frame
ϕ_{ij}	-	Receiver angle for face i, beacon j
ψ_{ij}	-	Transmitter angle for face i, beacon j

If noise and uncertainty are added to the measured body vectors and estimated reference frame vectors, no unique solution to Equation 3.2 exists. The problem then becomes one of non-linear least squares, as in Equation 3.1. Several algorithms have been developed to solve Wahba's problem, such as Davenport's q-method, Singular Value Decomposition, the Quaternion Estimator (QUEST), the first and second Estimators of the Optimal Quaternion (ESOQ-1 and ESOQ-2), and the Fast Optimal Attitude Matrix (FOAM), along with first and second order variants on some of these methods [16].

3.3.2 Direction measurements

In order to make use of an established solution to Wahba's problem, measurements of vectors in the body frame and corresponding estimates of vectors in the reference frame are required. In the SPHERES testbed, the vectors $\boldsymbol{\mu}$ and $\boldsymbol{\nu}$ are unit vectors directed from the center of each receiver face to each ultrasonic beacon, along the vector $\boldsymbol{v}_{ij}$ shown in Figure 3-3. For reference purposes, the primary quantities used in the following discussion of attitude determination are summarized in Table 3.3.2.

The ultrasonic transmitters and receivers used to obtain range measurements have

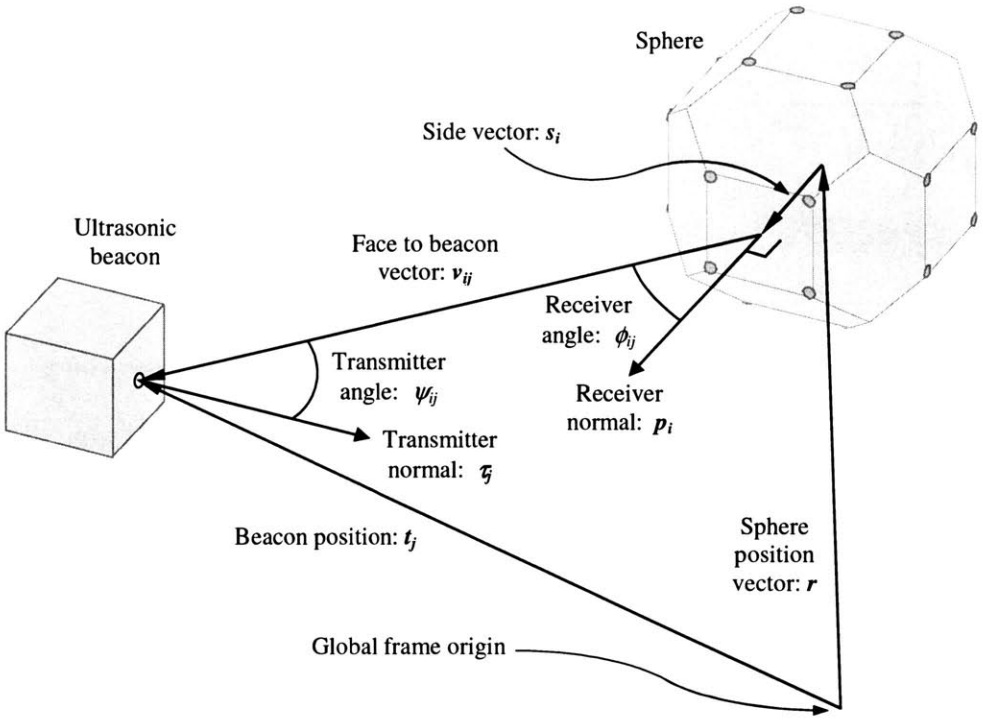


Figure 3-3: Quantities used in attitude determination.

angle and range-dependent sensitivity. The "transmitter angle" ψ is defined as the angle between the ultrasonic transmitter normal $\boldsymbol{\tau}$ and the vector from the transmitter to the center of the receiver face. The "receiver angle" ϕ is the angle between the ultrasonic receiver normal $\boldsymbol{p}$ and the vector from the center of the receiver face to the transmitter. The signal strength depends on these two angles and on the distance between the transmitter and receiver, and signal degradation causes nonzero-mean bias errors in the time-of-flight measurements.

The prototype spheres are equipped with three ultrasonic receivers per face on each of six faces. The receivers on each face are mounted flush with the flat surface of the face (the "sensor plane"), with their lines of sight normal to the plane. Because the receivers on a given face have parallel lines of sight and are mounted such that the separation distance between two adjacent receivers is much smaller than the distance from the receivers to the beacon, the time-of-flight bias errors due to transmitter angle, receiver angle, and distance may all be considered common mode.

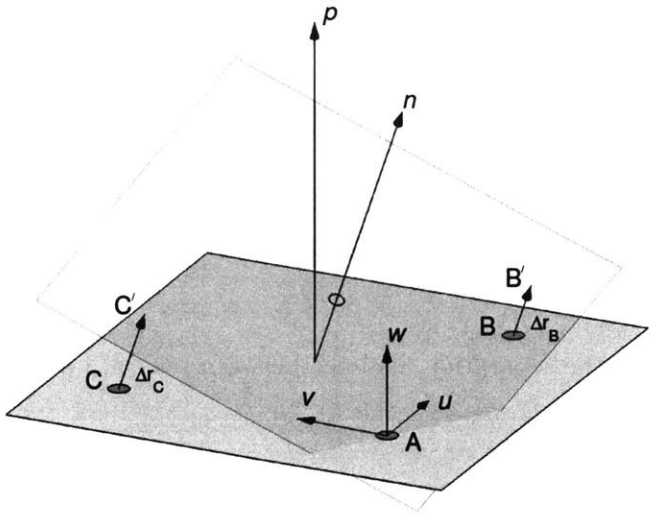


Figure 3-4: Sensor plane (solid square) with sensor coordinate system and unit normal $\boldsymbol{p}$, and incoming ultrasonic wavefront (translucent square) with negative unit normal $\boldsymbol{n}$. The three-sensor configuration corresponds to the prototype sphere geometry.

These common-mode errors are eliminated by considering only the differences between distance measurements at each combination of two receivers, and the resulting bias-free quantities are used to produce vector measurements in the body coordinate frame.

It is useful to define a u–v–w sensor plane coordinate system based on the sensor geometry shown in Figure 3-4, where the origin of the sensor plane coordinate frame coincides with the location of sensor A. The vector $\boldsymbol{n}$ is the negative of the unit normal to the incoming planar wavefront, and $\boldsymbol{p}$ is the unit normal to the sensor plane, pointing directly away from the geometric center of the sphere.

$$\boldsymbol{n} = [n_1 \; n_2 \; n_3]^T \tag{3.6}$$

$$\boldsymbol{p} = [0 \; 0 \; 1]^T \tag{3.7}$$

Let $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ be the positions of the ultrasonic receivers A, B, and C, respectively. These positions are expressed in terms of the sensor plane coordinates

as

$$\boldsymbol{a} = [0\ 0\ 0]^T \tag{3.8}$$

$$\boldsymbol{b} = [u\ 0\ 0]^T \tag{3.9}$$

$$\boldsymbol{c} = [0\ v\ 0]^T \tag{3.10}$$

where u and is the separation distance between sensor A and sensor B, and v is the separation distance between sensor A and sensor C.

Transmitter to receiver range measurements r are determined based on the times of flight of ultrasonic signals between the beacons and the receivers. All delta times are taken with respect to the receive time at sensor A, so by definition the sensor plane and the wavefront plane intersect at sensor A. The corresponding differences in measured distance are defined as $\Delta r_B \equiv r_B - r_A$ and $\Delta r_C \equiv r_C - r_A$. The wavefront plane may be described with the standard plane equation. At sensor A, this equation is

$$n_1\, a_u + n_2\, a_v + n_3\, a_w = \boldsymbol{n}^T \boldsymbol{a} = d = 0 \tag{3.11}$$

Since sensor A is located at the origin in the sensor plane coordinates, the plane equation constant $d = 0$. The positions of sensors B and C may be mapped onto the wavefront plane as

$$\boldsymbol{b'} = \boldsymbol{b} + \Delta r_B\, \boldsymbol{n} \tag{3.12}$$

$$\boldsymbol{c'} = \boldsymbol{c} + \Delta r_C\, \boldsymbol{n} \tag{3.13}$$

The points B′ and C′ have positions $\boldsymbol{b'}$ and $\boldsymbol{c'}$ on the wavefront plane, so they must obey the plane equation. Applying the plane equation with $\boldsymbol{b'}$ and $\boldsymbol{c'}$ and expanding terms gives:

$$\boldsymbol{n}^T \boldsymbol{b'} = \boldsymbol{n}^T \boldsymbol{b} + \boldsymbol{n}^T\, \Delta r_B\, \boldsymbol{n} = 0 \tag{3.14}$$

$$\boldsymbol{n}^T \boldsymbol{c'} = \boldsymbol{n}^T \boldsymbol{c} + \boldsymbol{n}^T\, \Delta r_C\, \boldsymbol{n} = 0 \tag{3.15}$$

Since Δr_B and Δr_C are scalars and $\boldsymbol{n}$ is a unit vector, Equations 3.14 and 3.15 simplify to

$$\boldsymbol{n}^T\boldsymbol{b} + \Delta r_B = 0 \tag{3.16}$$

$$\boldsymbol{n}^T\boldsymbol{c} + \Delta r_C = 0 \tag{3.17}$$

Equations 3.16 and 3.17 may be solved simultaneously with the constraint equation $\boldsymbol{n}^T\boldsymbol{n} = 1$ to determine the components n_1, n_2, and n_3 of the wavefront plane negative unit normal.

$$n_1 = -\left(\frac{\Delta r_B}{u}\right) \tag{3.18}$$

$$n_2 = -\left(\frac{\Delta r_C}{v}\right) \tag{3.19}$$

$$n_3 = \frac{\sqrt{u^2v^2 - u^2\,\Delta r_C^2 - v^2\,\Delta r_B^2}}{uv} \tag{3.20}$$

The strict solution is $\boldsymbol{n} = [n_1\ n_2\ \pm n_3]^T$, but the positive value of n_3 is chosen to create a vector pointing away from the sphere center. This solution for $\boldsymbol{n}$ is defined whenever $u^2v^2 \geq u^2\,\Delta r_C^2 + v^2\,\Delta r_B^2$, and this inequality must be verified in the attitude determination algorithm. The relationship of the sensor plane coordinate frame to the sphere body frame is determined by the geometry of the sphere, so the wavefront normal direction $\boldsymbol{n}$ may be transformed with a predetermined rotation matrix from the sensor plane coordinate system into the sphere body coordinate system. The result is a unit vector pointing from the sensor plane to the transmitter, expressed in the body coordinate system. This process is repeated for each beacon signal received at each sphere face, to produce a collection of body vectors $\boldsymbol{\mu}_{ij} = \Gamma_i\boldsymbol{n}_{ij}$ for face i and beacon j, where the pre-determined (fixed) rotation matrix Γ_i rotates a vector in the sensor plane frame of face i into the sphere body frame.

Estimates of these vectors in the reference frame ($\boldsymbol{\nu}_{ij}$) are obtained by vector subtraction using the state estimate and the known beacon locations, as illustrated in Figure 3-3. The last known attitude estimate $\hat{\mathbf{q}}$ is used to form an estimated

rotation matrix $\hat{\Theta}(\hat{\mathbf{q}})$. The body frame vectors $\boldsymbol{s}_i$ point from the origin of the body frame to the center of each sensor plane i. Given the known beacon locations $\boldsymbol{t}_j$ and the estimated sphere position $\hat{\boldsymbol{r}}$, the estimated vector $\boldsymbol{v}_{ij}$ from face i to beacon j may be expressed as

$$\boldsymbol{v}_{ij} = \boldsymbol{t}_j - \hat{\boldsymbol{r}} - \hat{\Theta}^T(\hat{\mathbf{q}})\boldsymbol{s}_i \tag{3.21}$$

and the reference frame representation $\boldsymbol{\nu}_{ij}$ of the measured body frame vectors $\boldsymbol{\mu}_{ij}$ from Equation 3.2 may be determined.

$$\boldsymbol{\nu}_{ij} = \frac{\boldsymbol{v}_{ij}}{|\boldsymbol{v}_{ij}|} \tag{3.22}$$

The receiver angle ϕ_{ij} for a given face/beacon pair may be determined from the dot product of the sensor plane normal $\boldsymbol{p}_i$ and the wavefront normal $\boldsymbol{n}_{ij}$, where both vectors are expressed in the sensor plane coordinate frame. The receiver angle ϕ_{ij} is therefore

$$\phi_{ij} = \arccos(\boldsymbol{p}_i^T \boldsymbol{n}_{ij}) \tag{3.23}$$

$$= \arccos(n_{ij,w}) \tag{3.24}$$

where $n_{ij,w}$ signifies the w-axis component of $\boldsymbol{n}_{i,j}$. The transmitter angle ψ_{ij} may likewise be found from the dot product of the transmitter normal vectors $\boldsymbol{\tau}_j$ and the reference frame attitude vectors $\boldsymbol{\nu}_{ij}$.

$$\psi_{ij} = \arccos(-\boldsymbol{\nu}_{ij}^T \boldsymbol{\tau}_j) \tag{3.25}$$

The quantities ϕ_{ij} and ψ_{ij} are used by the SPHERES onboard software in the determination of measurement reliability and in the calculation of range measurement bias error. The flight spheres will be equipped with four ultrasonic receivers on each of the six faces, resulting in an over-determined measurement problem. A least-squares approach will be taken with these excess data to find the optimal body-frame vectors.

3.3.3 Davenport's q-Method

One of the established solutions to Wahba's problem is used to compare the body and reference frame representations of each direction vector and so solve for the optimal attitude, as characterized by the optimal quaternion. The spheres onboard software currently uses Davenport's q-Method [16, 26, 15] because of its simplicity and robustness. Davenport's q-Method is outlined here for completeness following the derivation in [16]. Wahba's cost function of Equation 3.1 may be rewritten as

$$J(\Theta) = \lambda_0 - \mathrm{tr}(\Theta \mathrm{B}^T) \tag{3.26}$$

$$\lambda_0 \equiv \sum_i \alpha_i \tag{3.27}$$

$$\mathrm{B} \equiv \sum_i \alpha_i \boldsymbol{\mu}_i \boldsymbol{\nu}_i^T \tag{3.28}$$

The cost is therefore minimized by maximizing $\mathrm{tr}(\Theta \mathrm{B}^T)$. Since the rotation matrix $\Theta(\mathbf{q})$ is a homogenous quadratic function of $\mathbf{q}$, the trace may be rewritten as

$$\mathrm{tr}(\Theta \mathrm{B}^T) = \mathbf{q}^T \mathrm{K} \mathbf{q} \tag{3.29}$$

for the symmetric, zero trace matrix

$$\mathrm{K} \equiv \begin{bmatrix} 2\mathrm{B}_{11} - \mathrm{tr}(\mathrm{B}) & \mathrm{B}_{12} + \mathrm{B}_{21} & \mathrm{B}_{13} + \mathrm{B}_{31} & \mathrm{B}_{23} - \mathrm{B}_{32} \\ \mathrm{B}_{12} + \mathrm{B}_{21} & 2\mathrm{B}_{22} - \mathrm{tr}(\mathrm{B}) & \mathrm{B}_{23} + \mathrm{B}_{32} & \mathrm{B}_{31} - \mathrm{B}_{13} \\ \mathrm{B}_{13} + \mathrm{B}_{31} & \mathrm{B}_{23} + \mathrm{B}_{32} & 2\mathrm{B}_{33} - \mathrm{tr}(\mathrm{B}) & \mathrm{B}_{12} - \mathrm{B}_{21} \\ \mathrm{B}_{23} - \mathrm{B}_{32} & \mathrm{B}_{31} - \mathrm{B}_{13} & \mathrm{B}_{12} - \mathrm{B}_{21} & \mathrm{tr}(\mathrm{B}) \end{bmatrix} \tag{3.30}$$

The optimal quaternion is then the normalized eigenvector of K corresponding to the largest eigenvalue.

$$\mathrm{K}\mathbf{q} = \lambda_{max}\mathbf{q} \tag{3.31}$$

and the cost function may be written in terms of λ_{max} as

$$J(\Theta) = \lambda_0 - \lambda_{max} \tag{3.32}$$

If the two largest eigenvalues of K are equal, there are not enough data to determine a unique attitude solution [15].

3.4 Position and Velocity Determination

The position and velocity of a sphere are determined using a continuous-discrete extended Kalman filter. The SPHERES Kalman filter implementation currently updates only position and velocity, since a sufficiently accurate attitude estimate (the optimal quaternion) is obtained through the memoryless procedure described is Section 3.3, and the angular rate is measured directly.

3.4.1 Continuous-discrete extended Kalman filter equations

The continuous-discrete extended Kalman filter equations are used to propagate and update the state estimate of each sphere. A system with continuous dynamics and discrete measurements may be modelled by

$$\dot{\boldsymbol{x}} = \boldsymbol{f}[\boldsymbol{x}(t), \boldsymbol{u}(t)] + \boldsymbol{\epsilon}'(t) \qquad \boldsymbol{\epsilon}'(t) \sim N[\mathbf{0}, \mathrm{Q}(t)] \tag{3.33}$$

$$\boldsymbol{z}_k = \boldsymbol{h}_k[\boldsymbol{x}(t_k)] + \boldsymbol{\epsilon}_k \qquad \boldsymbol{\epsilon}_k \sim N[\mathbf{0}, \mathrm{R}_k] \tag{3.34}$$

for continuous time-varying system dynamics $\boldsymbol{f}[\boldsymbol{x}(t), \boldsymbol{u}(t)]$ and groups of measurements $\boldsymbol{z}_k$ received at discrete times t_k [12]. The vectors $\boldsymbol{\epsilon}'(t)$ and $\boldsymbol{\epsilon}_k = \boldsymbol{\epsilon}(t_k)$ are independent zero-mean white noise processes with covariance $\mathrm{Q}(t)$ and R_k, respectively. The state estimate $\hat{\boldsymbol{x}}(t)$ and error covariance matrix $\mathrm{P}(t)$ are propagated using

$$\dot{\hat{\boldsymbol{x}}}(t) = f[\hat{\boldsymbol{x}}(t), \boldsymbol{u}(t)] \tag{3.35}$$

$$\dot{\mathrm{P}}(t) = \mathrm{F}[\hat{\boldsymbol{x}}(t)]\mathrm{P}(t) + \mathrm{P}(t)\mathrm{F}^T[\hat{\boldsymbol{x}}(t)] + \mathrm{Q}(t) \tag{3.36}$$

where the over-hat, $\hat{}$, signifies that the quantity is an estimate. The state estimate and state covariance update equations are

$$\mathrm{K}_k = \mathrm{P}_k^{(-)}\,\mathrm{H}_k\big(\hat{\boldsymbol{x}}_k^{(-)}\big)\left[\mathrm{H}_k\big(\hat{\boldsymbol{x}}_k^{(-)}\big)\,\mathrm{P}_k^{(-)}\,\mathrm{H}_k^T\big(\hat{\boldsymbol{x}}_k^{(-)}\big) + \mathrm{R}_k\right]^{-1} \tag{3.37}$$

$$\hat{\boldsymbol{x}}_k^{(+)} = \hat{\boldsymbol{x}}_k^{(-)} + \mathrm{K}_k\left[\boldsymbol{z}_k - \boldsymbol{h}_k\left(\hat{\boldsymbol{x}}_k^{(-)}\right)\right] \tag{3.38}$$

$$\mathrm{P}_k^{(+)} = \left[\mathrm{I} - \mathrm{K}_k\,\mathrm{H}_k\big(\hat{\boldsymbol{x}}_k^{(-)}\big)\right]\mathrm{P}_k^{(-)} \tag{3.39}$$

where $^{(-)}$ signifies immediately prior to the update (a priori), and $^{(+)}$ signifies immediately after the update (a posteriori). The vector $\boldsymbol{h}_k(\hat{\boldsymbol{x}})$ contains the values of the measurements that are expected based on the current state estimate. The matrices $\mathrm{F}(\hat{\boldsymbol{x}})$ and $\mathrm{H}_k(\hat{\boldsymbol{x}}_k)$ are the state and measurement Jacobians, respectively, linearized about and evaluated at the current state estimate. They describe the linearized versions of the state dynamics and measurement influence, and are defined as

$$\mathrm{F}(\hat{\boldsymbol{x}}(t)) \equiv \left.\frac{\partial \boldsymbol{f}(\boldsymbol{x}(t))}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}(t)=\hat{\boldsymbol{x}}(t)} \tag{3.40}$$

$$\mathrm{H}_k(\hat{\boldsymbol{x}}(t_k)) \equiv \left.\frac{\partial \boldsymbol{h}(\boldsymbol{x}(t))}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}(t)=\hat{\boldsymbol{x}}(t_k)} \tag{3.41}$$

where differentiation of a vector by a vector is defined by

$$\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{b}} \equiv \begin{bmatrix} \frac{\partial a_1}{\partial b_1} & \frac{\partial a_1}{\partial b_2} & \cdots & \frac{\partial a_1}{\partial b_n} \\ \frac{\partial a_2}{\partial b_1} & \frac{\partial a_2}{\partial b_2} & \cdots & \frac{\partial a_2}{\partial b_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial a_n}{\partial b_1} & \frac{\partial a_n}{\partial b_2} & \cdots & \frac{\partial a_n}{\partial b_n} \end{bmatrix} \tag{3.42}$$

3.4.2 State propagation

The sphere state vector must be propagated between PADS global measurement updates, to provide current state information for control and to preserve the integrity of the a priori information used in the Kalman filter update. The sphere state vector contains position $\boldsymbol{r}$, velocity $\boldsymbol{v}$, quaternion $\mathbf{q}$, and angular rate $\boldsymbol{\omega}$ components. The position and velocity are expressed in the global reference frame, but the rate is

measured and expressed about the body frame axes. Only three quaternion elements are required to define the attitude, but the fourth is included in the state vector to simplify propagation and to help maintain normalization.

$$\begin{aligned} \boldsymbol{x} &\equiv [\boldsymbol{r}^T \ \boldsymbol{v}^T \ \mathbf{q}^T \ \boldsymbol{\omega}^T]^T \\ &= [r_x \ r_y \ r_z \ v_x \ v_y \ v_z \ q_1 \ q_2 \ q_3 \ q_4 \ \omega_x \ \omega_y \ \omega_z]^T \end{aligned} \tag{3.43}$$

The angular acceleration of a body-fixed axis with origin at the center of mass may be written with respect to the body frame as

$$\mathrm{J}\dot{\boldsymbol{\omega}} = -[\boldsymbol{\omega}\times]\mathrm{J}\boldsymbol{\omega} + \boldsymbol{m} \tag{3.44}$$

for angular rate ω, inertia tensor J, and applied body-frame moment $\boldsymbol{m}$ [14]. Expansion of the right hand side leads to

$$\mathrm{J}\dot{\boldsymbol{\omega}} = \mathrm{R}(\boldsymbol{\omega}) + \boldsymbol{m} \tag{3.45}$$

where the quantity $-[\boldsymbol{\omega}\times]\mathrm{J}\boldsymbol{\omega}$ has been denoted $\mathrm{R}(\boldsymbol{\omega})$. Expanding $\mathrm{R}(\boldsymbol{\omega})$ in terms of the components of $\boldsymbol{\omega}$ gives

$$\mathrm{R}(\boldsymbol{\omega}) \equiv \begin{bmatrix} (\mathrm{J}_{yy} - \mathrm{J}_{zz})\omega_y\omega_z + \mathrm{J}_{xy}\omega_x\omega_z + \mathrm{J}_{yz}(\omega_z^2 - \omega_y^2) - \mathrm{J}_{xz}\omega_x\omega_y \\ (\mathrm{J}_{zz} - \mathrm{J}_{xx})\omega_x\omega_z + \mathrm{J}_{yz}\omega_x\omega_y + \mathrm{J}_{xz}(\omega_x^2 - \omega_z^2) - \mathrm{J}_{xy}\omega_y\omega_z \\ (\mathrm{J}_{xx} - \mathrm{J}_{yy})\omega_x\omega_y + \mathrm{J}_{xz}\omega_y\omega_z + \mathrm{J}_{xy}(\omega_y^2 - \omega_x^2) - \mathrm{J}_{yz}\omega_x\omega_z \end{bmatrix} \tag{3.46}$$

The angular acceleration of the body frame may then be determined as a function of the body rates and the body-frame applied torque.

$$\dot{\boldsymbol{\omega}} = \mathrm{J}^{-1}\mathrm{R}(\boldsymbol{\omega}) + \mathrm{J}^{-1}\boldsymbol{m} \tag{3.47}$$

Given an applied force $\boldsymbol{f}$ expressed in the body frame, the reference to body frame rotation matrix $\Theta(\mathbf{q})$, and the sphere vehicle mass m, the acceleration of a sphere may be expressed in the reference frame as

$$\dot{\boldsymbol{v}} = \frac{1}{m}\Theta(\mathbf{q})^T \boldsymbol{f} \tag{3.48}$$

Using the mapping matrix M of Equation 2.14 or 2.24 (with parenthetical subscript denoting prototype or flight omitted), the influence of the thruster pair forces on the state dynamics is

$$\begin{bmatrix} \dot{\boldsymbol{v}} \\ \dot{\boldsymbol{\omega}} \end{bmatrix} = \begin{bmatrix} \frac{1}{m}\Theta(\mathbf{q})^T & 0 \\ 0 & r\mathrm{J}^{-1} \end{bmatrix} \mathrm{M}^{-1}\boldsymbol{f}_{i,j} + \begin{bmatrix} 0 \\ \mathrm{J}^{-1}\mathrm{R}(\boldsymbol{\omega}) \end{bmatrix} \tag{3.49}$$

where r is the thruster moment arm, assumed equal for all thrusters, and $\boldsymbol{f}_{i,j}$ denotes the six-place vector containing thruster pair forces from Equations 2.6 through 2.11 (prototype geometry) or Equations 2.16 through 2.21 (flight geometry), as appropriate. The attitude quaternion may be propagated in time with

$$\dot{\mathbf{q}} = \frac{1}{2}\Omega(\boldsymbol{\omega})\mathbf{q} \tag{3.50}$$

where the matrix $\frac{1}{2}\Omega(\boldsymbol{\omega})$ maps the quaternion into its derivative based on the body-frame rates $\boldsymbol{\omega}(t)$ [26]. This matrix is dependent on the time-varying body rates, and therefore must be computed at every propagation step using the current value for the angular rotation rate.

$$\Omega(\boldsymbol{\omega}) \equiv \begin{bmatrix} 0 & \omega_z & -\omega_y & \omega_x \\ -\omega_z & 0 & \omega_x & \omega_y \\ \omega_y & -\omega_x & 0 & \omega_z \\ -\omega_x & -\omega_y & -\omega_z & 0 \end{bmatrix} \tag{3.51}$$

Since the position and velocity components of the state vector are both expressed in the global frame, the time derivative of the position is simply the velocity component of the state.

$$\dot{\boldsymbol{r}} = \boldsymbol{v} \tag{3.52}$$

Given an initial state and the thruster on-times over a small time step Δt, Equa-

tions 3.49, 3.50, and 3.52 may be used to propagate the state estimate through time. During real-time SPHERES operations, it is unnecessary to use Equation 3.49 to calculate $\dot{\boldsymbol{v}}$ because accelerometers are used to measure linear acceleration. Knowledge of the angular acceleration vector $\dot{\boldsymbol{\omega}}$ is useful in some types of trajectory-following attitude control algorithms, but it is not required for computation of the angular rate vector, which is measured directly by rate gyroscopes. The primary applications of Equation 3.49 are attitude trajectory tracking, determination of the motion of a sphere in a simulation environment, and system identification.

It is therefore sufficient to propagate the state estimate $\hat{\boldsymbol{x}}$ using a relatively simple non-linear, time-varying differential equation, in which the linear acceleration $\dot{\boldsymbol{v}}(t)$ and angular rate $\boldsymbol{\omega}(t)$ are sampled directly from the PADS local sensors at every propagation step. The differential equation describing the state is then

$$\dot{\hat{\boldsymbol{x}}}(t) = \mathrm{A}(\hat{\boldsymbol{x}})\hat{\boldsymbol{x}}(t) + \mathrm{B}\dot{\boldsymbol{v}}(t) \tag{3.53}$$

in which the matrices $\mathrm{A}(\hat{\boldsymbol{x}})$ and B are defined as

$$\mathrm{A}(\hat{\boldsymbol{x}}) = \begin{bmatrix} 0_{3\times3} & \mathrm{I}_{3\times3} & 0_{3\times4} & 0_{3\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times4} & 0_{3\times3} \\ 0_{4\times3} & 0_{4\times3} & \frac{1}{2}\Omega(\hat{\boldsymbol{\omega}}) & 0_{4\times3} \\ 0_{3\times3} & 0_{3\times3} & 0_{3\times4} & 0_{3\times3} \end{bmatrix} \tag{3.54}$$

$$\mathrm{B} = \begin{bmatrix} 0_{3\times3} \\ \mathrm{I}_{3\times3} \\ 0_{4\times3} \\ 0_{3\times3} \end{bmatrix} \tag{3.55}$$

The sphere attitude is updated using the optimal quaternion method described in Section 3.3, so the Kalman filter updates only the position and velocity components of the sphere state. Therefore only the position and velocity components of the state error covariance matrix are propagated. The state process noise covariance $\mathrm{Q}(t)$ is

defined as

$$\mathrm{Q}(t) = E\left[\boldsymbol{\epsilon}'(t)\boldsymbol{\epsilon}'(t)^T\right] \tag{3.56}$$

where the noise $\boldsymbol{\epsilon}'$ acts only on the position and velocity, and $E[\cdot]$ is the expectation operator. The nature of this noise has not yet been well characterized on the SPHERES testbed. The following value of $\mathrm{Q}(t)$ has been successfully used in the laboratory:

$$\mathrm{Q}(t) = \mathrm{I}_{6\times 6} \tag{3.57}$$

The linearized state Jacobian for position and velocity is simply

$$\mathrm{F}(t) = \begin{bmatrix} 0_{3\times 3} & \mathrm{I}_{3\times 3} \\ 0_{3\times 3} & 0_{3\times 3} \end{bmatrix} \tag{3.58}$$

The error covariance matrix initial condition $\mathrm{P}(0) = \mathrm{I}_{6\times 6}$ has been used successfully in the laboratory.

The state vector and error covariance matrix estimates are propagated by the SPHERES onboard software using Equations 3.36, 3.53, 3.54, 3.55, 3.57, and 3.58 whenever a new local element measurement is made. Local element measurements are received and the state and covariance are propagated in the PADS interrupt, at a fixed frequency. The PADS interrupt frequency may be set as desired, as high as 100 Hz.

3.4.3 State updates

In the specific case of the SPHERES PADS, a distinction may be made between the local element measurements, which may be considered continuous functions of time and are used for state propagation, and global element range measurements, which occur at discrete times and are used for state updates. The local measurements are considered continuous because the local element sensors are sampled at every propagation step, whereas the global measurements are made at a maximum frequency of 8 Hz. The global element measurement sets, $\boldsymbol{z}_k$, consist of range measurements made

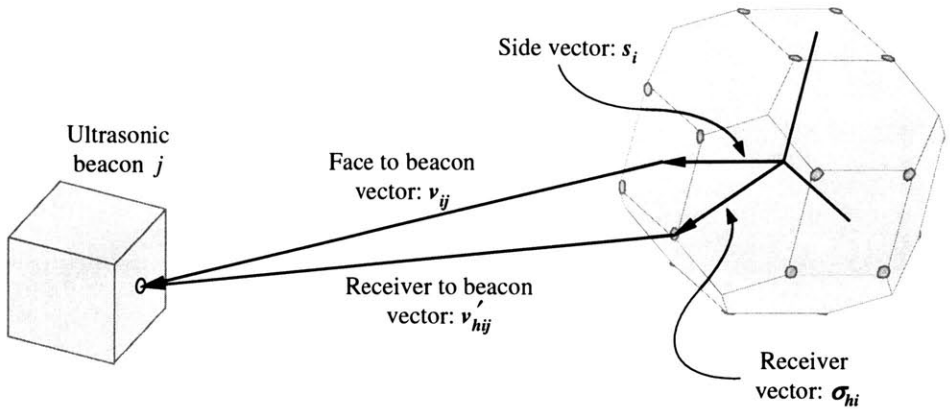


Figure 3-5: Quantities used in position determination and similar quantities used in attitude determination.

from external ultrasonic beacons to receivers mounted on the sphere faces.

At global update time t_k, a set of M noisy scalar range measurements $\lambda_{hij}(t_k) + \epsilon_{hij}(t_k)$ are made to receivers h on faces i from beacons j. For beacon location $\boldsymbol{t}_j$ and sphere position $\boldsymbol{r}$, the vector from receiver h on face i to beacon j may be expressed as

$$\boldsymbol{v}'_{hij} = \boldsymbol{t}_j - \boldsymbol{r} - \Theta^T(\mathbf{q})\boldsymbol{\sigma}_{hi} \tag{3.59}$$

where $\boldsymbol{\sigma}_{hi}$ signifies the location of the receiver expressed in the body frame. Note that this $\boldsymbol{v}'_{hij}$ is slightly different from the $\boldsymbol{v}_{ij}$ defined in Equation 3.21; $\boldsymbol{v}'_{hij}$ measures to the beacons from each receiver rather than from each face center. This distinction is depicted in Figure 3-5. An error-free range measurement may then be expressed as

$$\lambda_{hij} = \sqrt{(\boldsymbol{v}'_{hij})^T (\boldsymbol{v}'_{hij})} \tag{3.60}$$

$$= \sqrt{[\boldsymbol{t}_j - \boldsymbol{r} - \Theta^T(\mathbf{q})\,\boldsymbol{\sigma}_{hi}]^T\,[\boldsymbol{t}_j - \boldsymbol{r} - \Theta^T(\mathbf{q})\,\boldsymbol{\sigma}_{hi}]} \tag{3.61}$$

The Kalman filter update equations utilize a single measurement vector $\boldsymbol{z}_k$. The collections of M measured and estimated ranges must therefore be arranged in vector format, under a single index m. The function used to map the indices $m = q(i, j, k)$ does not matter, so long as it is one-to-one and consistently applied for all $\lambda_{ijk} \rightarrow \lambda_m$.

The mapping algorithm used in the SPHERES Kalman filter implementation arranges all valid measurements by beacon, then by face, and then by receiver.

It is necessary to form a vector of expected measurements $\boldsymbol{h}_k(\hat{\boldsymbol{x}}(t_k))$, which contains the corresponding expected range measurements $\lambda_{hij}(\hat{\boldsymbol{x}}) \rightarrow \lambda_m(\hat{\boldsymbol{x}})$, as computed in real-time based on the current state estimate. The linearized measurement Jacobian H_k of Equation 3.41 may then be determined. With the range measurements organized into vector form, the partial derivatives of λ_m with respect to the position components r_x, r_y, and r_z may be expanded as

$$\begin{aligned}
\frac{\partial \lambda_m}{\partial r_x} &= \frac{1}{\sqrt{\xi_m}}\Big[\left(q_1^2 - q_2^2 - q_3^2 + q_4^2\right)\sigma_{hi,x} + 2\left(q_1 q_2 - q_3 q_4\right)\sigma_{hi,y} \\
&\quad + 2\left(q_1 q_3 + q_2 q_4\right)\sigma_{hi,z} - t_{j,x} + r_x\Big] \qquad (3.62) \\
\frac{\partial \lambda_m}{\partial r_y} &= \frac{1}{\sqrt{\xi_m}}\Big[2\left(q_1 q_2 + q_3 q_4\right)\sigma_{hi,x} + \left(-q_1^2 + q_2^2 - q_3^2 + q_4^2\right)\sigma_{hi,y} \\
&\quad + 2\left(q_2 q_3 - q_1 q_4\right)\sigma_{hi,z} - t_{j,y} + r_y\Big] \qquad (3.63) \\
\frac{\partial \lambda_m}{\partial r_z} &= \frac{1}{\sqrt{\xi_m}}\Big[2\left(q_1 q_3 - q_2 q_4\right)\sigma_{hi,x} + 2\left(q_2 q_3 + q_1 q_4\right)\sigma_{hi,y} \\
&\quad + \left(-q_1^2 - q_2^2 + q_3^2 + q_4^2\right)\sigma_{hi,z} - t_{j,z} + r_z\Big] \qquad (3.64)
\end{aligned}$$

where for simplicity in notation the quantity $\xi_m = \xi_{hij}$ has been defined as

$$\begin{aligned}
\xi_{hij} \equiv\ & \left[\left(q_1^2 - q_2^2 - q_3^2 + q_4^2\right)\sigma_{hi,x} + 2\left(q_1 q_2 - q_3 q_4\right)\sigma_{hi,y}\right. \\
& \left. + 2\left(q_1 q_3 + q_2\, q_4\right)\sigma_{hi,z} - t_{j,x} + r_x\right]^2 \\
& + \left[2\left(q_1 q_2 + q_3 q_4\right)\sigma_{hi,x} + \left(-q_1^2 + q_2^2 - q_3^2 + q_4^2\right)\sigma_{hi,y}\right. \\
& \left. + 2\left(q_2 q_3 - q_1 q_4\right)\sigma_{hi,z} - t_{j,y} + r_y\right]^2 \\
& + \left[2\left(q_1 q_3 - q_2 q_4\right)\sigma_{hi,x} + 2\left(q_2 q_3 + q_1 q_4\right)\sigma_{hi,y}\right. \\
& \left. + \left(-q_1^2 - q_2^2 + q_3^2 + q_4^2\right)\sigma_{hi,z} - t_{j,z} + r_z\right]^2 \qquad (3.65)
\end{aligned}$$

The partial derivatives of λ_m with respect to the velocity components v_x, v_y, and

v_z may be expanded as

$$\frac{\partial \lambda_m}{\partial v_x} = 0 \tag{3.66}$$

$$\frac{\partial \lambda_m}{\partial v_y} = 0 \tag{3.67}$$

$$\frac{\partial \lambda_m}{\partial v_z} = 0 \tag{3.68}$$

and H_k may then be expressed in terms of the partial derivatives of λ_m:

$$\mathrm{H}(\hat{\boldsymbol{x}}) = \left[\begin{array}{cccccc} \frac{\partial \lambda_1}{\partial r_x} & \frac{\partial \lambda_1}{\partial r_y} & \frac{\partial \lambda_1}{\partial r_z} & 0 & 0 & 0 \\ \frac{\partial \lambda_2}{\partial r_x} & \frac{\partial \lambda_2}{\partial r_y} & \frac{\partial \lambda_2}{\partial r_z} & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial \lambda_M}{\partial r_x} & \frac{\partial \lambda_M}{\partial r_y} & \frac{\partial \lambda_M}{\partial r_z} & 0 & 0 & 0 \end{array} \right] \Bigg|_{\boldsymbol{x} = \hat{\boldsymbol{x}}} \tag{3.69}$$

The linearized measurement matrix H_k must be computed at each measurement step, based on the current state estimate. The measurement noise covariance R_k is defined by

$$\mathrm{R}_k = E\left[\boldsymbol{\epsilon}_k \boldsymbol{\epsilon}_k^T\right] \tag{3.70}$$

The SPHERES onboard Kalman filter currently uses a value of $\mathrm{R}_k = 25\mathrm{I}_{M \times M}$. The magnitude of the actual nonzero mean measurement noise is significantly smaller than the value of $\boldsymbol{\epsilon}$ used to determine this R_k, but the noise is artificially inflated to account for nonzero mean measurement error. A logic filter is used to remove the nonzero mean measurement error, but it is imperfect, and increasing the value of $\boldsymbol{\epsilon}_k$ in the measurement model has been empirically shown to result in filter stability in the presence of these errors.

The state update is performed using Equations 3.37, 3.38, 3.39, 3.69, and 3.70, where for the purposes of the update the quaternion and rate components of the state vector are ignored, such that $\hat{\boldsymbol{x}}$ is treated as consisting of only $\hat{\boldsymbol{r}}$ and $\hat{\boldsymbol{v}}$.

The state update as formulated here creates a significant computational load on the digital signal processor. Formation of the optimal Kalman gain requires inversion

of an $M \times M$ matrix, and M may realistically be as large as forty when using four sensors per face on the flight hardware. A possible alternative to this formulation that requires significantly less computational time is mentioned briefly in Section 3.5.1.

3.5 PADS algorithm path

Unbiased measurements corrupted only by white noise are required by the Kalman filter, but angle and range-dependent errors in the raw distance measurements have non-zero mean. Several approaches may be taken to solve this problem, such as augmentation of the state vector with quantities to represent the bias terms, or the inclusion of a logic filter to remove the current expected value of each bias term. The approach chosen for SPHERES is a logic filter, because the magnitudes of the bias terms change dramatically between global updates, making estimation of the bias quantities through augmentation of the state vector difficult. Several logic filters in series are used to discard range measurements of questionable quality, and a lookup table is used to correct for expected bias errors based on measured and estimated angles and ranges. The steps taken to perform state estimation are shown in flow chart form in Figure 3-6.

The steps taken to process range measurements are further detailed below, with initialization steps omitted. These steps are designed to maximize the use of quantities from which common-mode errors have been removed.

1. Read in ultrasound times of flight and multiply by the speed of sound to obtain raw range measurements.

2. Keep range measurements between 20 cm and 290 cm. This step is intended to reduce echo and multi-path effects.

3. Keep measurements on faces that received signals on at least three receivers. The prototype sphere has three receivers per face, and three ranges to a face are required for the attitude determination algorithm.

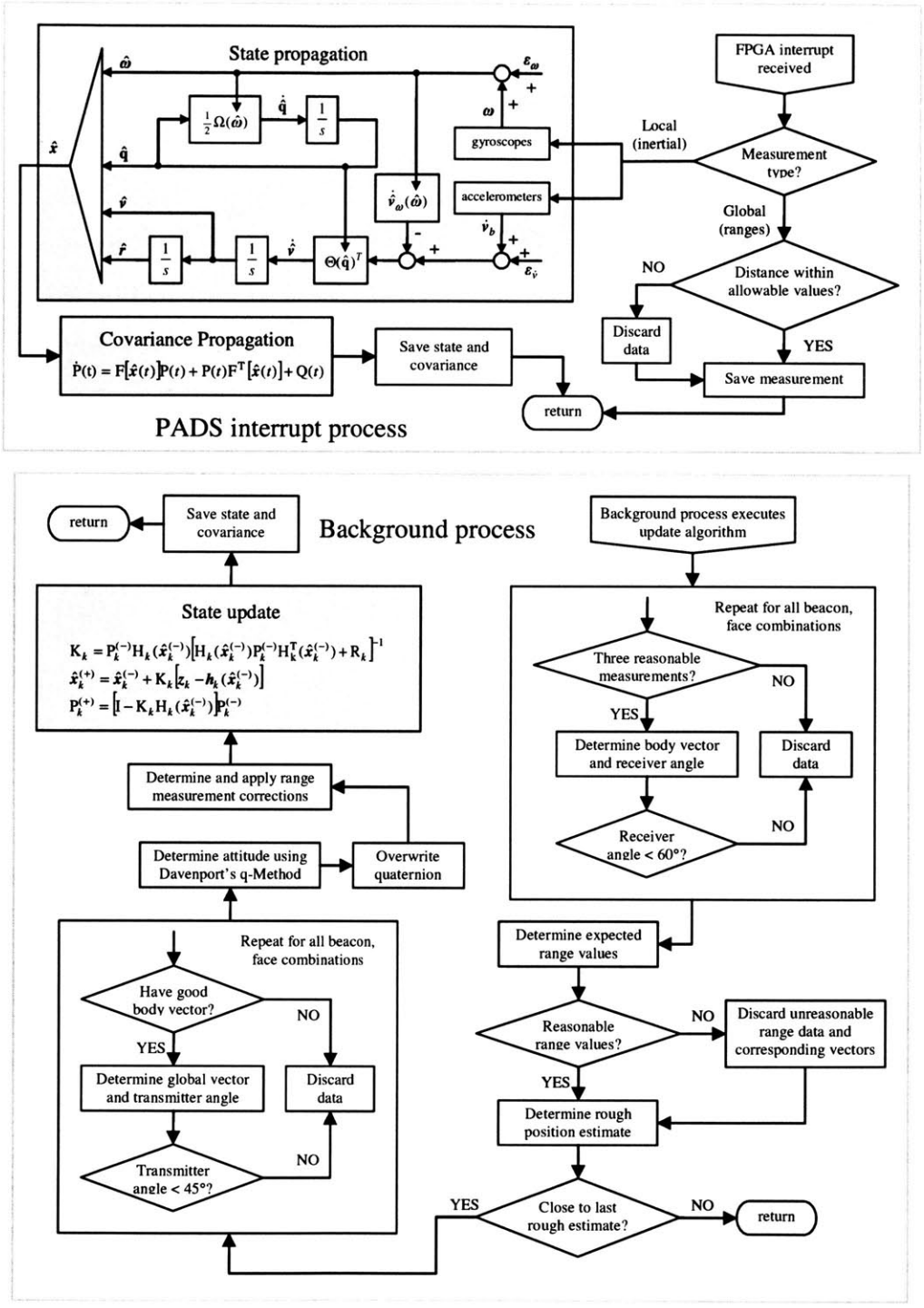


Figure 3-6: State estimation implementation in the PADS interrupt and background process.

4. Find body frame attitude vectors using Equations 3.18, 3.19, 3.20 and the predefined rotation matrices Γ_i.

5. Determine receiver angles using Equation 3.23. Keep ranges for each face i, beacon j pair that produces a receiver angle $\phi_{ij} \leq 60°$.

6. Keep ranges that are within some specified tolerance of their expected value, as determined from the current state estimate.

7. Perform a rough memoryless position estimate using only the body-frame attitude vectors and knowledge of the beacon positions. Compare this rough estimate with the previous rough estimate, and abort if the change in position exceeds tolerance.

8. Use Equation 3.22 to determine the reference frame attitude vectors from the rough position estimate and the current attitude estimate.

9. Keep ranges for each face i, beacon j pair that produces a transmitter angle $\psi_{ij} < 45°$. The transmitter angle is measured between the transmitter normal and the vector connecting the transmitter and receiver.

10. Perform memoryless attitude update using Davenport's q-Method.

11. Apply bias correction terms to the range measurements, based on receiver angles, transmitter angles, and estimated distances.

12. Perform Kalman filter update of position and velocity.

3.5.1 Future improvements to PADS

A memoryless algorithm is currently used to update the attitude of the sphere whenever global range measurements are received. For the purpose of testbed development, the attitude determination algorithm was intentionally created separately from the Kalman filter, which is currently used to update only the position and velocity estimates. Separate implementation of the two algorithms simplified the development

of the algorithms on the testbed hardware. Future plans include the integration of the attitude determination algorithm into the Kalman filter framework, in order to provide a more accurate attitude estimate.

Several additional approaches to the determination of position and velocity will also be tested. One possible approach is to use an incremental Kalman filter to process the range measurements received from each beacon at the time they are received, rather than waiting to process the complete collection of range measurements en masse. The primary obstacle to this method is the need for a rough state estimate, which is used to determine the bias correction terms to apply to the range measurements. This rough estimate is currently determined using the complete collection of range measurements. One solution to the biased measurement problem is to treat the face to beacon body-frame unit vectors $\boldsymbol{\mu}_{ij}$ discussed in Section 3.3.2 as measurements in the Kalman filter, rather than using the individual ranges. Common-mode errors are not present in the body vectors, and it is possible that this approach will yield superior results to the method currently in use, in terms of both estimate accuracy and computational load.

Chapter 4

Control Interface Design and Implementation

The SPHERES testbed is intended to support investigations by multiple guest scientists. To simplify the use of the testbed by guest scientists located at remote facilities, and to simplify the integration of guest scientist code by the MIT SPHERES team, a set of three distinct interface frameworks has been developed: the standard, direct, and custom interfaces. A package containing descriptions of the standard, direct, and custom interfaces, several source code examples, and a simulation environment for use in developing custom code is delivered to guest scientists. This package is termed the Guest Scientist Program (GSP) interface, and the following sections describe the SPHERES software framework, and relevant portions of the GSP interface. The standard, direct, and custom interfaces are described, and two examples are presented of using the standard interface to create a test.

4.1 Flight Software Overview

The onboard DSP hardware supports two timer interrupt processes running over a background process and other asynchronous support processes. A medium-priority control interrupt runs at a fixed frequency which can be changed at any time by the control algorithm. In the control interrupt, each thruster is assigned an on-time based

Table 4.1: Subsystem global variable data structures.

Structure	Subsystem
`comm`	Communications
`ctrl`	Control
`dbug`	Debug and error
`gsp`	Guest Scientist Program
`pads`	Position and attitude determination
`prop`	Propulsion and housekeeping
`sys`	Identification, time-keeping, etc.
`tele`	Telemetry

on maneuver, control, and pulse width modulation algorithms. The control interrupt may execute at any integer frequency as high as as 25 Hz. A high-priority propulsion interrupt runs at 1 kHz to handle low-level interaction with the thrusters, providing a pulse width resolution of one millisecond.

The low-priority background process contains an infinite loop that sequentially handles communications processing, position and attitude updates, and housekeeping tasks. An interrupt (the PADS interrupt) is triggered in the DSP whenever new sensor data are available, and state propagation occurs as local element (inertial) sensor data are received. The interactions between these processes are shown in Figure 4-1. Detailed descriptions of the communications and general software architectures are given by Saenz-Otero [21].

4.1.1 Global variable organization

To improve source code organization, all global variables are organized by subsystem into eight global structures, listed in Table 4.1. Contents of the `ctrl`, `pads`, `prop`, and `sys` structures that are relevant to guest scientists are described in Section B.2 of Appendix B. The `gsp` structure contains any custom global variables required by the guest scientist, and the structure definition may be changed as desired by the guest scientist. Custom global variables and custom preprocessor directives are discussed in Sections 4.3.9 and 4.3.10.

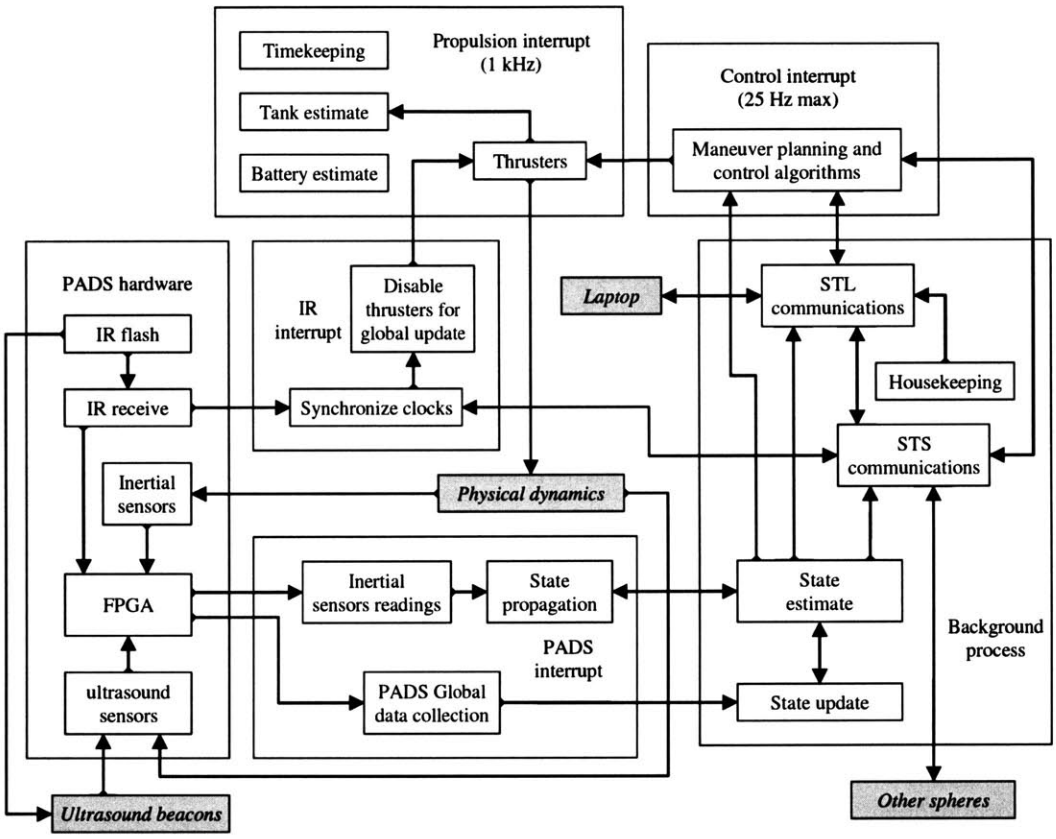


Figure 4-1: High-level organization of the flight code algorithms within the interrupt and background processes, and their interactions with the external environment. Process blocks with italicized text are external to the sphere onboard hardware and software [18, 21].

4.2 Interfaces Overview

4.2.1 Standard interface

The standard interface consists of two parts: the standard control interface, and the standard sensor interface. The standard control interface (SCI) framework is designed to facilitate rapid test development using modular algorithm blocks with pre-defined inputs and outputs. A collection of SCI algorithm modules is provided with the GSP interface, and several of these modules are described in Section 4.3. Two example tests in Section 4.4 make use of these modules to demonstrate the use of the SCI. Guest scientists are encouraged to create new modules using the input/output rules presented in Section 4.3. The modular, high-level design of the SCI framework allows for rapid maneuver development and simple tracking of code changes, as individual modules may be replaced without the need to write new code to perform the functions handled by the other modules. Pre-defined inputs and outputs for each module type ensure that old code can be reused, while allowing flexibility in the design of individual modules.

The standard sensor interface may be used to increase the traceability of the SPHERES testbed to actual space missions, by modifying or limiting the sensor information available to the onboard state estimator. The standard sensor interface provides a means to simulate the information produced by different types of sensors, through the use of algorithm modules that produce simulated sensor information based on the current state estimate. For example, given the position and attitude of the sphere, a sun sensor simulation module could produce a simulated sun sensor measurement, which may then be used in a custom state estimation module to produce a simulated state estimate based on the sun sensor data. The control code then operates on this simulated state estimate rather than on the actual best state estimate. Guest scientists are encouraged to write custom sensor simulation modules to improve the traceability of the testbed to their missions of interest.

4.2.2 Direct interface

The direct interface offers greater freedom in the design of Guest Scientist code blocks. In the direct interface, the contents of the control interrupt and background process are replaced by the guest scientist with custom code. Use of this interface requires familiarity with details of the SPHERES flight code and operations protocols, and results in a steeper learning curve for the guest scientist. Feedback from the MIT SPHERES team may take more time for tests created using the direct interface than for tests created using the standard interface, due to increased integration time.

The relationship between the sphere dynamics and the standard and direct interfaces is shown in the block diagram of Figure 4-2, using the notation introduced in Chapter 3. The top half of the figure represents the state dynamics and measurement models for a single sphere vehicle, and the bottom half of the figure shows the interaction of the flight software with the environment, through sensor readings and thruster firings. The standard interface allows modifications to or replacement of the individual algorithm blocks that are located within the shaded regions labelled Standard Control Interface and Standard Sensor Interface. The direct interface allows modifications to or replacement of all functions contained within the shaded region labelled Direct Interface.

4.2.3 Custom interface

It is possible to create algorithms that fall outside of the standard and direct frameworks. The custom interface allows for modification of the flight code at any level, and may be used if the guest scientist desires to completely reinvent the SPHERES flight software. Use of this interface requires knowledge of operations protocols and low-level hardware interfaces, and involves an extremely steep learning curve for the guest scientist. Development of tests using the custom interface requires significant interaction with the MIT SPHERES team.

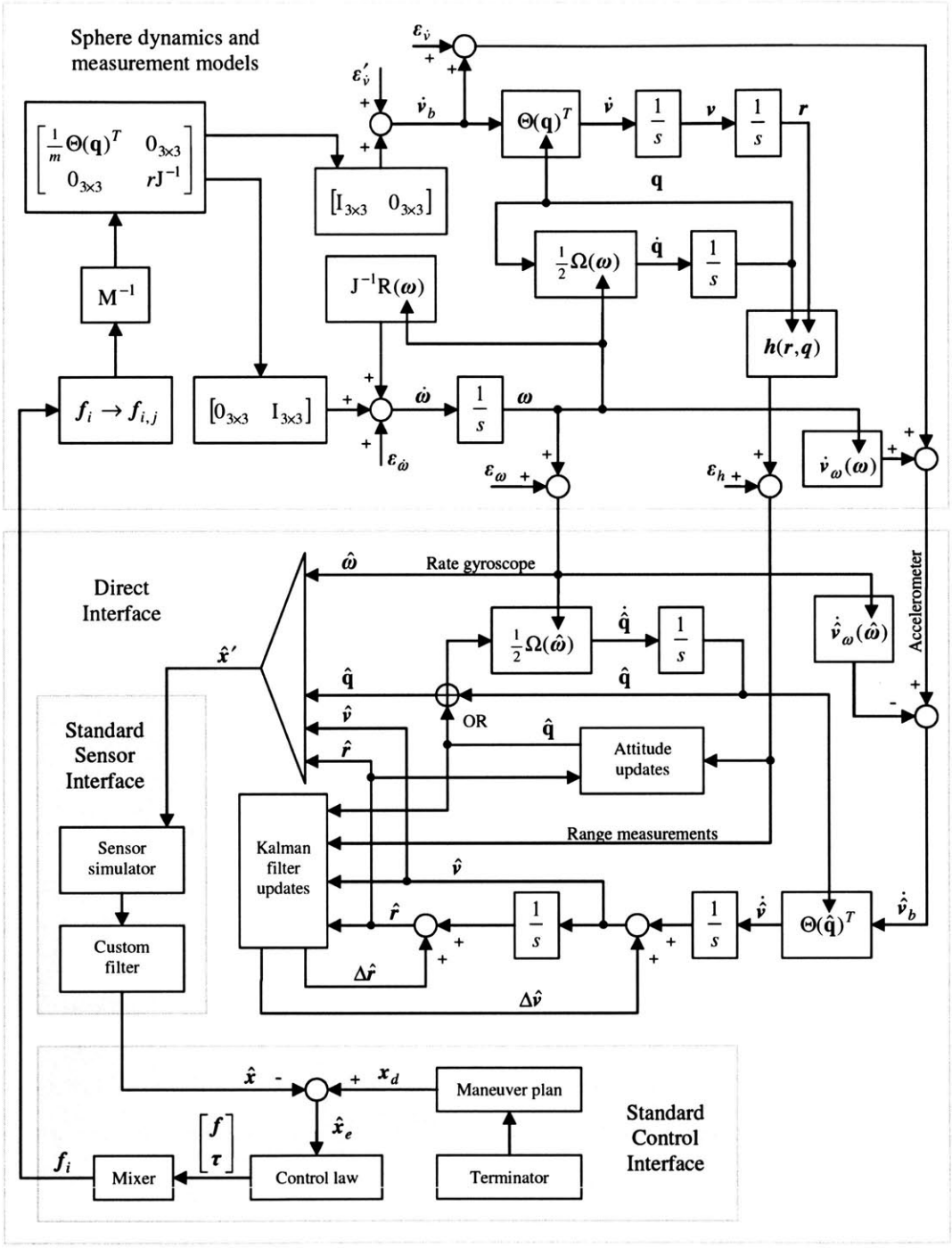


Figure 4-2: The relationship between the sphere dynamics and the standard and direct interfaces.

4.2.4 ISS operational requirements

International Space Station operational and safety considerations require that several conditions must be met during pre-test, test, and post-test maneuvers. The following conditions are handled automatically by the standard control interface. Guest scientists choosing to use the direct or custom interfaces must guarantee that these conditions are satisfied.

1. An Enable button is located on the sphere switch panel. When the Enable button is pushed or the sphere receives an Enable command through the STL communications channel, the sphere must begin state regulation to maintain a stationary position.

2. The control code must disable the thrusters whenever the sphere leaves the test volume.

3. When a test has completed, the sphere must notify the laptop of completion, null any residual velocity, and then disable the thrusters. If the residual velocity has not reached an acceptable threshold within a specified time period, the thrusters must be disabled.

4.3 Standard Control Interface Modules

Under the standard control interface, a maneuver is defined as the repeated execution of a sequence of algorithm modules. A maneuver ends when one or more specified termination conditions are met. A test is defined as a group of related maneuvers that may be executed in either a linear or non-linear sequence. During laboratory and ISS operations, housekeeping tasks such as battery and propellent tank replacement are performed as necessary between tests.

A basic SCI maneuver is represented in source code by a sequence of function calls to the four basic types of SCI modules: command, controller, mixer, and terminator. This sequence of function calls is termed a module sequence, and repeated executions

of a module sequence result in a maneuver. Figure 4-3 shows a schematic representative module sequence that could define a simple maneuver. A fifth module type, flow control, may be used to modify the sequential flow of maneuvers during a test.

A maneuver may be defined using one or more of the modules included with the GSP interface, or the guest scientist may create custom modules following the input/output rules set forth in Sections 4.3.2 through 4.3.6. The maneuvers to be performed during a specific test are specified by function calls to modules in the function `do_maneuver(...)`, in the file `maneuverlist.c`. During run-time, the module sequence specified by the current maneuver number is executed repeatedly at the control frequency. When the set of termination conditions specified in the termination module is met, the terminator module sets the termination flag. Each time the function `do_maneuver(...)` returns, a layer of underlying controller housekeeping code checks the status of the termination flag. If the flag is set, the maneuver number is incremented and maneuver-specific variables such as the elapsed maneuver time are reset. The next instance of the control interrupt will execute the module sequence corresponding to the new maneuver number. Additional tasks performed by the controller housekeeping code are discussed in Section 4.5.

In order to clearly define the boundary between constant and test-specific code, the source code files are organized in a specific directory structure, shown in Figure 4-4. Files that may be modified or customized for individual tests are contained in directories specific to each test. Files that are required by most or all tests are included in the higher-level directories. For a test created using the standard control interface, the three files `maneuverlist.c`, `gsp.h`, and `gsp.c` contain the unique information needed to define the behavior of a particular sphere during that test. Appendix B contains source code for some of the pre-defined modules included with the GSP interface.

The following sections describe the required inputs and outputs of each type of SCI module, and the rules that must be followed in writing custom modules, in order that the modules function correctly with the rest of the onboard code. Examples of each module type are presented, and these example modules are then used in two

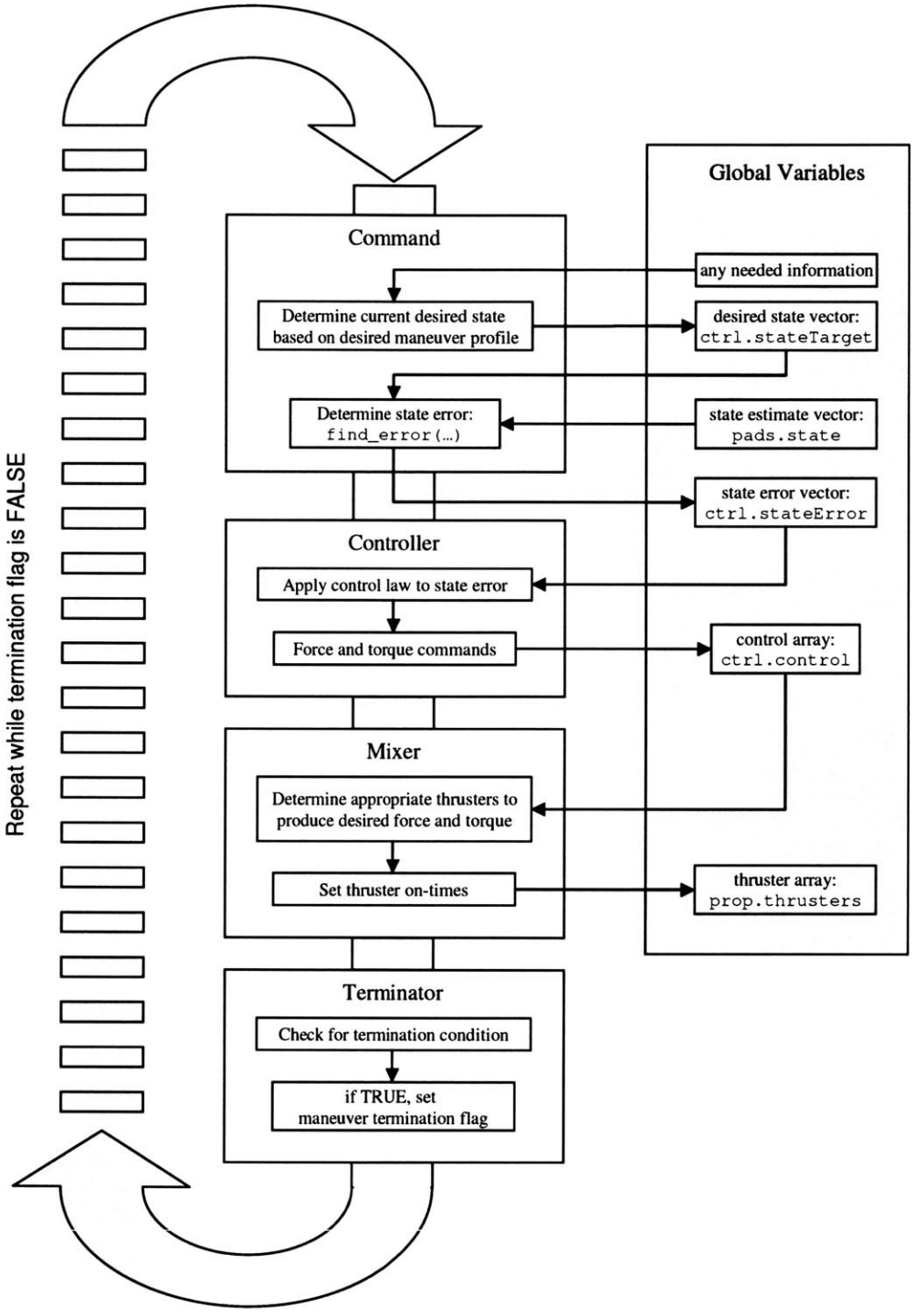


Figure 4-3: A schematic representative module sequence defining a basic standard control interface maneuver.

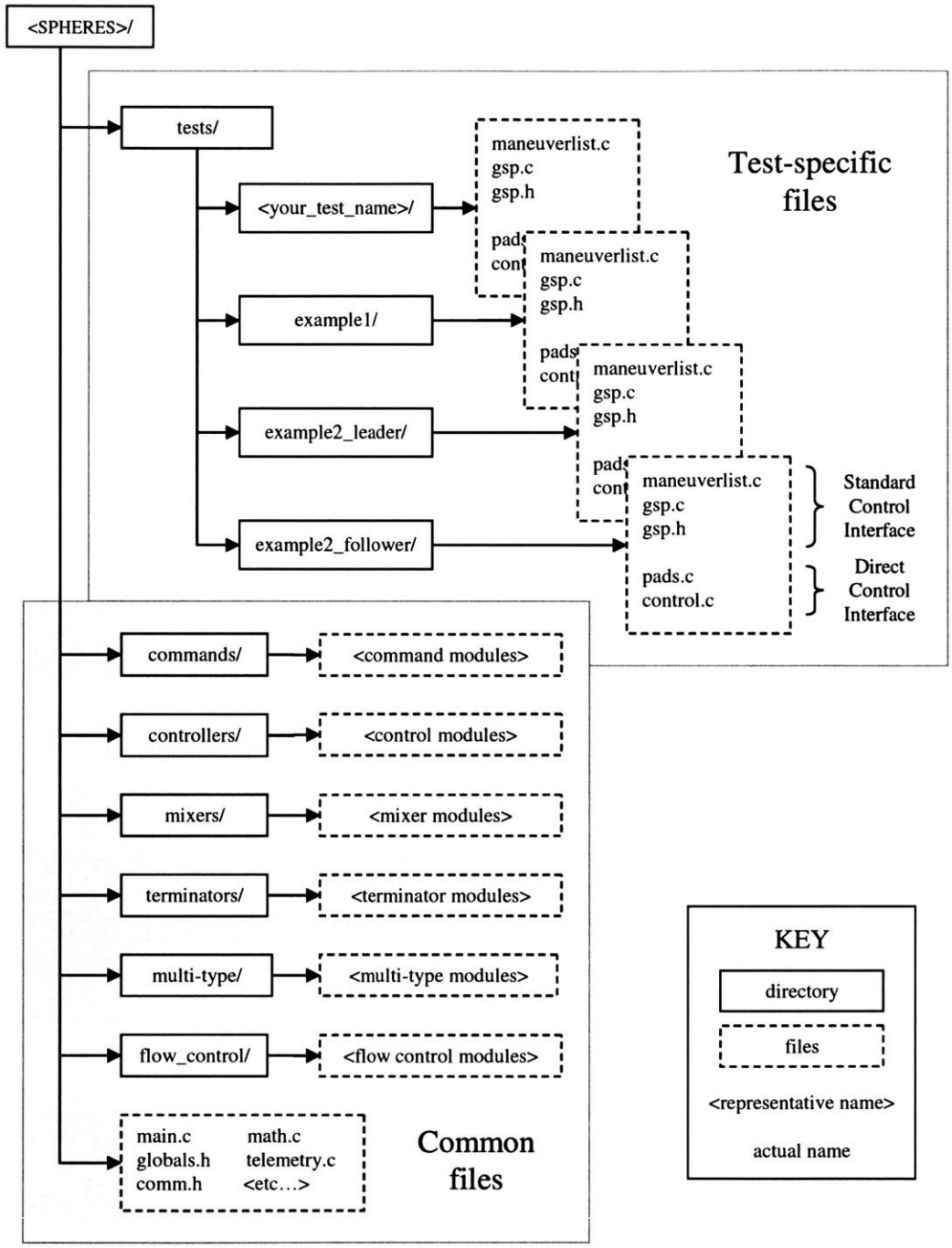


Figure 4-4: The SPHERES source code directory organization is designed to clearly demarcate the boundary between constant files and test-specific files that may be modified by the user, and to simplify the creation of new projects.

example tests in Section 4.4.

4.3.1 Module robustness and the universal module rule

There is one rule that must be followed when creating a new module of any type: local variables of type `static` (e.g. `static int var1`, `static float var2`, etc.) must be explicitly re-initialized whenever the maneuver elapsed time `ctrl.maneuverTime == 0.0`. This condition is true only during the first execution of a module sequence when the sphere begins a new maneuver. Explicit re-initialization is necessary because `static` variables retain their values between successive function calls, and the ending value of a `static` local variable from a previous maneuver should not affect the value of that variable at the beginning of the current maneuver. Explicit re-initialization of local variables that are not `static` is not necessary, as these variables do not retain their values between successive function calls.

A module containing `static` variables is termed fragile, because if the module is not called when the maneuver elapsed time `ctrl.maneuverTime == 0.0`, any local `static` variables will not be initialized and the module will "break," resulting in unexpected behavior. Such a situation could occur only if a fragile module was embedded either in a conditional statement in the module sequence, or inside another module. It may then be possible for the conditional to evaluate `FALSE` or the higher-level module to not call the embedded module during the first execution of the module sequence, resulting in uninitialized variables. It is possible to embed fragile modules within other modules, but care must be taken to ensure that the fragile modules are always called during the first execution of the module sequence. Modules not containing `static` variables are termed robust, and need not be called when `ctrl.maneuverTime == 0.0` to guarantee proper operation.

4.3.2 SCI module type 1: command

Each standard control interface maneuver begins with a command module. The command module updates the values in the current target (i.e. desired) state vector

`ctrl.stateTarget` based on some desired trajectory algorithm, and calls the function `find_error(...)`, which results in the creation of the current state error vector `ctrl.stateError`. Valid indices into `ctrl.stateTarget` are listed in Table B.1 in Appendix B. The error vector is used by the control module, the next function called in the maneuver sequence. The following rules must be followed by all command modules:

1. The command module must write some or all of the contents of the desired state vector `ctrl.stateTarget`.

2. The command module must call `find_error(...)` with appropriate arguments to create the state error vector `ctrl.stateError`.

The following command modules are among those included with the GSP interface, and serve to illustrate the variety of maneuvers that can be performed using the SCI. Several of these modules are used in the example tests in Section 4.4.

```
void regulate(void)

void regulate_specified(
        float desiredPosX,      // desired x position
        float desiredPosY,      // desired y position
        float desiredPosZ,      // desired z position
        float desiredQuat1,     // desired q1
        float desiredQuat2,     // desired q2
        float desiredQuat3,     // desired q3
        float desiredQuat4)     // desired q4

void regulate_polar(
        float radius,           // position radius
        float offsetAngle,      // position offset angle
        float circleX,          // circle center, x-axis
        float circleY,          // circle center, y-axis
        float circleZ,          // circle center, z-axis
        float q1,               // desired q1
        float q2,               // desired q2
        float q3,               // desired q3
        float q4)               // desired q4
```

```
void circle_z(
        float radius,           // position radius
        float circleX,          // circle center, x-axis
        float circleY,          // circle center, y-axis
        float circleZ,          // circle center, z-axis
        float startAngle,       // initial angular position
        float elapsedTime,      // maneuver elapsed time
        float period)           // trajectory period

void circle_z_lag(
        float radius,           // position radius
        float circleX,          // circle center, x-axis
        float circleY,          // circle center, y-axis
        float circleZ,          // circle center, z-axis
        float lagAngle,         // angular offset from leader
        float leaderID)         // ID number of leader
```

The function `regulate()` samples the position and quaternion when the elapsed maneuver time `ctrl.maneuverTime == 0.0`. The state error is then determined on all subsequent executions with respect to this initial sampled state. This module is fragile, since the target state is determined only when `ctrl.maneuverTime == 0.0`.

The function `regulate_specified(...)` takes as arguments a desired position and quaternion. The state error is produced based on these arguments and the current state estimate. This module is robust, because it does not use any static variables.

The function `regulate_polar(...)` is similar to `regulate_specified(...)`, but the desired state is specified in polar coordinates. The origin of the polar system in the global frame is defined by the point (`circleX`, `circleY`, `circleZ`), and the desired position of the sphere is specified in the polar frame by (`radius`, `offsetAngle`, 0), where `offsetAngle` is measured in radians from the global frame x-axis. This module is robust.

The function `circle_z(...)` creates a circular target trajectory in the x-y plane, at a radial distance of `radius` from the point (`circleX`, `circleY`, `circleZ`). The argument `startAngle` specifies the initial angular offset of the trajectory from the global frame x-axis, in radians. The argument `period` specifies the trajectory period,

and `elapsedTime` is used to determine the current position in the trajectory. The desired position at a particular `elapsedTime` is determined by the equations

$$\begin{aligned}
\texttt{ctrl.stateTarget[POS_X]} &= \texttt{radius}\ \sin\left(2\pi\frac{\texttt{elapsedTime}}{\texttt{period}} + \texttt{startAngle}\right) \\
\texttt{ctrl.stateTarget[POS_Y]} &= \texttt{radius}\ \cos\left(2\pi\frac{\texttt{elapsedTime}}{\texttt{period}} + \texttt{startAngle}\right) \\
\texttt{ctrl.stateTarget[POS_Z]} &= \texttt{circleZ}
\end{aligned}$$

The desired quaternion is calculated such that the body z-axis is aligned with the polar (and global) frame z-axis, and the body x-axis points directly towards the circle center. The state error is then determined based on this desired state and the current state estimate. This module is fragile.

The function `circle_z_lag(···)` determines the desired position in the specified polar frame based on the specified radius and angular offset from another sphere, rather than on a pre-determined trajectory. The argument `leaderID` specifies the sphere identification number (`SPHERE1`, `SPHERE2`, or `SPHERE3`) of the sphere that is being tracked. The argument `lagAngle` is the desired angular offset in the polar frame from the current angular position of the leader sphere. The desired quaternion is computed such that the body z-axis is aligned with the polar (and global) frame z-axis, and the body x-axis points directly towards the circle center. An example showing the use of `circle_z(···)` and `circle_z_lag(···)` is given in Section 4.4.2. This module is robust.

4.3.3 SCI module type 2: control

The control module applies a control law to the contents of the state error vector, and assigns force and torque commands to the six-place array `ctrl.control`. Force commands are represented in the global coordinate frame, and torque commands are represented in the body coordinate frame. It is often convenient to use two separate control modules to write the control vector: one for position and velocity and another for attitude quaternion and angular rate. Valid indices into `ctrl.stateError` are

listed in Table B.1, and valid indices into `ctrl.control` are listed in Table B.2 in Appendix B. The control array is used by the next function call, the mixer module. The following rules must be followed by all control modules:

1. The control module must write the contents of the control array `ctrl.control` with global-frame forces and/or body-frame torques.

The following control modules are among those included with the GSP interface, and are used to determine the force and torque commands for both of the example tests in Section 4.4.

```
void control_position_PD(
        float gainPos,      // position gain
        float gainVel)      // velocity gain

void control_attitude_NLPD(
        float gainAng,      // angle gain
        float gainRate)     // rate gain
```

The function `control_positionPD(...)` acts on the position and velocity components of the state error with a PD control law using the specified position and velocity gains. This module writes only the force components of the control array, and a separate attitude control law must be used to write the torque components. This module is robust.

The function `control_attitude_NLPD(...)` acts on the attitude quaternion and angular rate components of the state error with a non-linear PD-like control law, using the specified angle and rate gains. This module writes only the torque components of the control array, and a separate position control law must be used to write the force components. This module is robust.

4.3.4 SCI module type 3: mixer

The mixer module assigns thruster on-times to `prop.thrusters` based on pulse modulation and deadband rules, the thruster geometry, and the control array `ctrl.control`

of force and torque commands. The following rules must be followed by all mixer modules.

1. The mixer module must write the contents of the twelve-place thruster array `prop.thrusters` with integer millisecond on-times.

The following mixer function is included with the GSP interface, and is used in both example tests in Section 4.4. A more complex mixer could be created that accounts for thruster failure, or that implements a phase space deadband, for instance.

```
void mix_simple(
        int minPulseWidth)  // thruster on-time deadband
```

The function `mix_simple(···)` assigns thruster on-times based on force and torque commands, a simple model of the thruster geometry, pulse width modulation rules, and the specified thruster deadband in milliseconds. This module is robust.

4.3.5 SCI module type 4: terminator

The terminator function compares current conditions with one or more criteria for maneuver termination. When the termination criteria are met, the terminator function sets the variable `*fTerminate = TRUE`. This variable is called the termination flag, and must be included as an argument in all terminator modules. Each time the function `do_maneuver(···)` returns, the controller housekeeping code checks the status of the termination flag. If the flag is set, the maneuver number is incremented and maneuver-specific variables such as the elapsed maneuver time are reset. The next instance of the control interrupt will execute the module sequence corresponding to the new maneuver number. The following rules must be followed by all terminator functions:

1. Terminator modules must include as an argument `int *fTerminate`.

2. The termination condition is signaled by setting the flag `*fTerminate = TRUE`.

The following terminator functions are among those included with the GSP interface, and serve to illustrate the variety of criteria that can be used to determine maneuver termination.

```
void terminate_elapsed(
        int   *fTerminate,  // termination flag
        float endTime)      // maneuver time at which to terminate

void terminate_clock(
        int   *fTerminate,  // termination flag
        float endTime)      // test time at which to terminate

void terminate_commanded(
        int *fTerminate)    // termination flag

void terminate_hold_vel(
        int   *fTerminate,  // termination flag
        float threshold,    // velocity threshold
        float holdTime)     // minimum hold time

void terminate_hold_pos(
        int   *fTerminate,  // termination flag
        float threshold,    // position threshold
        float holdTime)     // minimum hold time

void terminate_nreps(
        int   *fTerminate,  // termination flag
        int   nreps)        // number of iterations

void terminate_ready(
        int *fTerminate)    // termination flag
```

The function `terminate_elapsed(...)` ends the current maneuver when the maneuver elapsed time clock `ctrl.maneuverTime` indicates a time greater than or equal to the argument `endTime`. This module is robust.

The function `terminate_clock(...)` ends the current maneuver when the test clock `ctrl.testTime` indicates a time greater than or equal to the argument `endTime`. This module is robust.

The function `terminate_commanded(···)` ends the current maneuver according to parameters specified by another sphere over the STS communication channel. The termination command is sent by the remote sphere using `send_terminate(···)`, with parameters specifying the desired termination time and the commanded maneuver number. Upon receipt of the `send_terminate` command, the recipient sphere saves the commanded termination time and the commanded maneuver number in the global variables `ctrl.commandedTime` and `ctrl.commandedManeuver`, respectively. The function `terminate_commanded(···)` ends the current maneuver when the test clock `ctrl.testTime` indicates a time greater than or equal to `ctrl.commandedTime`. Upon termination the new maneuver number is set equal to `ctrl.commandedManeuver`, unless `ctrl.commandedManuever == NEXT_MANEUVER`, in which case the maneuver number will increment as usual. This module is fragile.

The function `terminate_hold_vel(···)` terminates the current maneuver when the magnitude of the velocity error has been below a specified threshold continuously for some specified time. The velocity error magnitude is determined from the state error vector, and compared to the specified velocity error threshold. If the velocity error is less than the threshold value, a timer is incremented. If the threshold is violated, the timer is reset to zero. When the value of the timer surpasses the specified cumulative hold time, the function is terminated. This terminator is used by the controller housekeeping algorithm to terminate a regulation maneuver that executes automatically at the completion of each test. This module is fragile.

The function `terminate_hold_pos(···)` acts on position error rather than velocity error, but is otherwise identical to `terminate_hold_vel(···)`. This module is fragile.

The function `terminate_nreps(···)` ends the current maneuver after `nreps` instances of the control interrupt have occurred during the current maneuver. This module is fragile.

The function `terminate_ready(···)` ends the current maneuver when all spheres have signalled a state of readiness. Readiness is signalled using the `signal_ready(···)` function.

4.3.6 SCI module type 5: maneuver flow control

Maneuver flow control modules are used to modify the sequential flow of the maneuvers within a test. The following flow control functions are included with the GSP interface, and are located in the `flow_control/` directory.

```
void goto_maneuver(
        int *fTerminate,    // termination flag
        int maneuverNum)    // maneuver number to go to

void delay_termination(
        int   *fTerminate,  // termination flag
        float delayTime)    // delay from current time

void signal_ready(
        int   fSendMsg)     // sends message when nonzero

void send_terminate(
        int   fSendMsg,     // sends message when nonzero
        int   sphereID,     // commanded sphere ID number
        int   maneuverNum,  // maneuver number command
        float delayTime)    // delay for comm latency

void force_terminate(
        int   fSendMsg,     // sends command when nonzero
        int   sphereID,     // commanded sphere ID number
        int   maneuverNum,  // maneuver number command
        float delayTime)    // delay for comm latency

void force_next(
        int   fSendMsg,     // sends message when nonzero
        int   sphereID,     // commanded sphere ID number
        int   maneuverNum)  // maneuver number command

void wait_for_all(
        int   *fTerminate,  // termination flag
        float delayTime)    // delay for comm latency
```

When the termination flag is set to `TRUE`, the function `goto_maneuver(...)` causes the next instance of the control interrupt to switch to maneuver number `maneuverNum`

rather than the next maneuver in the sequence. This function must appear in the module sequence after the terminator module. This module is robust.

The function `delay_termination(...)` is used to temporarily delay termination after the termination flag is first set TRUE. This function must appear in the module sequence after the terminator module(s) that it is intended to delay. This module is fragile.

The function `signal_ready(...)` is used to signal to all operating spheres that this sphere has reached a state of readiness. The ready signal is sent the first time that the input argument `fMsgSend` is nonzero, and will not be sent again in the current maneuver. Each recipient sphere saves readiness information in the global variable `ctrl.fReady[sphereID]`, where `sphereID` is the identification number of the sending sphere. This module is fragile.

The function `send_terminate(...)` is used to command another sphere to terminate the current maneuver and begin a particular maneuver number. For each value of `sphereID`, the command is sent the first time that the argument `fMsgSend` is nonzero, and will not be sent again for the remainder of the maneuver. The recipient sphere will be commanded to start maneuver number `maneuverNum`, unless the commanded maneuver number is `NEXT_MANEUVER`, in which case the maneuver number will increment as usual. The argument `delayTime` specifies how long the recipient of the command should wait before beginning the new maneuver, to account for communication processing delay. The commanded termination time is the sum of the current test time `ctrl.testTime` and `delayTime`. The module sequence executing on the recipient sphere must call `terminate_commanded(...)` to act on the command. This module is fragile, and initialization occurs separately for each value of the argument `sphereID`.

The function `force_terminate(...)` is similar to `send_maneuver(...)`. The difference is that this command is handled by the control housekeeping algorithm in the recipient sphere, so the recipient sphere need not call `terminate_commanded(...)` to act on the command. Receipt of a `force_terminate` command immediately terminates the current maneuver, and forces transition to the commanded maneuver number.

This function may be used to pre-empt the maneuver being performed by another sphere, bypassing the need for maneuver termination on the remote sphere. This module is fragile, and initialization occurs separately for each value of the argument `sphereID`.

The function `force_next(...)` is used to change the maneuver sequence of another sphere. For each value of `sphereID`, the command is sent the first time that the argument `fMsgSend` is nonzero, and will not be sent again for the remainder of the maneuver. The argument `maneuverNum` is saved by the recipient sphere in the global variable `ctrl.forcedManeuver`, and upon maneuver termination, the recipient sphere maneuver number will be set to `ctrl.forcedManeuver`. Note that this module does not force immediate maneuver termination by the recipient sphere. The current maneuver must terminate normally before the commanded maneuver will begin. This module is fragile, and initialization occurs separately for each value of the argument `sphereID`.

The function `wait_for_all(...)` is used to force the maneuvers in each sphere to terminate simultaneously, after the termination conditions in all spheres have been satisfied. A function call to `wait_for_all(...)` must appear after the termination module in the currently executing module sequence of all spheres. Internal to `wait_for_all(...)`, the state of the termination flag is checked, and when it is first set, the `signal_ready(...)` module is called to signal readiness. The termination flag is then set `FALSE` to delay termination. When all spheres have signaled a state of readiness, the instance of `wait_for_all(...)` running on the PADS master calls `send_terminate(...)` and then `delay_termination(...)` with the specified `delayTime`. The remaining spheres wait for a termination command, effectively executing `terminate_commanded(...)`. Note that during the time period in which termination is delayed, the current maneuver module sequence continues to execute at the control frequency; the setting of the termination flag is simply delayed until all spheres are ready. This module is fragile.

4.3.7 SCI multi-type modules

It is occasionally necessary or convenient to combine one or more of the four types of modules into a single function call, or to bypass one or more module types. An example multi-type module is `thrusters_timed(···)`, which turns on one or more specific thrusters for a specified time.

```
void thrusters_timed(
        long  thrusters,    // thruster numbers to fire
        float onTime)       // commanded on-time
```

The function `thrusters_timed(···)` performs low-level operations on the thrusters, rather than implementing a control algorithm, and therefore bypasses the command and control rules listed in Sections 4.3.2 and 4.3.3. Note that `thrusters_timed(···)` replaces only the command, control, and mixer modules, and the maneuver must still contain a termination module. Multiple calls to `thrusters_timed(···)` in a single maneuver may be used to simultaneously fire two or more specific thrusters with different on-time commands. The following sequence simultaneously turns on thrusters 1, 4, and 12 for 50 ms, thruster 7 for 100 ms, and thrusters 8 and 9 for 35 ms. The maneuver in this example is terminated after 100 ms. This module is robust.

```
thrusters_timed(THR1 | THR4 | THR12, 0.050);
thrusters_timed(THR7, 0.100);
thrusters_timed(THR8 | THR9, 0.035);
terminate_elapsed(fTerminate, 0.100);
```

4.3.8 Maneuver list file

The maneuver list file contains function calls to the modules that comprise the maneuvers to be performed by a particular sphere during one or more tests. The function `do_maneuver(···)` contains two conditional statements, one that conditions on the test number, and one that conditions on the maneuver number. When `do_maneuver(···)`

is called with a particular test and maneuver number, the module sequence in the corresponding block of the nested conditional is executed. The maneuver list file must be included with the preprocessor directive `#include` in the file `gsp.c`. In the following discussions, the file containing the definition of `do_maneuver(...)` will be referred to as `maneuverlist.c`, though the actual name of the file may be chosen by the Guest Scientist to be representative of the test contents. The following rules must be followed when writing a maneuver list file.

1. The maneuver list file must include with `#include` the files containing all modules called by `do_maneuver(...)`.

2. The maneuver list file must contain the function `do_maneuver(...)`, with the function prototype `void do_maneuver (int testNum, int maneuverNum, long elapsedTime, unsigned int *fTerminate, unsigned int *fTestDone)`.

3. The function `do_maneuver(...)` must contain an outermost `switch` statement on `testNum`, beginning at a value of 1, and with a maximum allowable value of 200.

4. The function `do_maneuver(...)` must contain one or more inner `if` or `switch` conditional statements to distinguish between maneuvers based on `maneuverNum`, beginning at a value of 1, and with a maximum value of 200.

5. The flag `*fTestDone = TRUE` must be set when the last maneuver in a test has completed.

A template maneuver list file and several example maneuver lists are included with the GSP interface. Three of these example maneuver lists are used to create the two example tests presented in Section 4.4.

4.3.9 Custom header: `gsp.h`

Custom global variables may be added by Guest Scientists through modification of the definition of the global structure `gsp` in the header file `gsp.h`. Preprocessor directives such as `#define` and `#include` may be used in `gsp.h` to define quantities or include

additional header files for use by Guest Scientist code. The variables in the gsp global structure are initialized with the function init_gsp(), which must be defined by the Guest Scientist in the file gsp.c. The following rules apply if custom global variables are required for the test:

1. A structure to hold global variables must be defined with typedef struct.

2. An instance of that structure called gsp must be created.

4.3.10 Custom source: gsp.c

The file gsp.c contains versions of the functions init_gsp() and free_gsp() specific to a particular sphere, for a particular test. The initialization function init_gsp() is used to set parameter and variable initial conditions, and to allocate memory for vectors and matrices. This function is called only upon sphere power-on and reset. The function free_gsp() is used to deallocate any memory space that was dynamically allocated in init_gsp(), and is required for the GSP simulation (discussed in Chapter 5) but not by the flight code. The following rules must be followed in writing gsp.c:

1. A call must be made in init_gsp() to init_sphereID(···), where the single argument specifies the identification number of this sphere. Each sphere must have a unique identification number, where valid values of the identification number are SPHERE1, SPHERE2, and SPHERE3

2. A call must be made in init_gsp() to init_comm(···), with three arguments specifying which spheres are involved in this test. Valid values of the arguments are TRUE and FALSE, where TRUE indicates that a particular sphere is involved in the test. All spheres must specify the same arguments in order to properly initialize the token ring communications protocol.

3. A call must be made in init_gsp() to set_pads_master(···), where the single argument specifies the identification number of the sphere that will function as

the PADS master. Valid values of the argument are `SPHERE1`, `SPHERE2`, and `SPHERE3`. All spheres must specify the same PADS master.

4. A file containing the maneuver list procedure `do_maneuver(...)` must be included with `#include` in `gsp.c`.

5. Each memory space that is dynamically allocated in `init_gsp()` must have a corresponding deallocation call in `free_gsp()`.

4.4 Standard Control Interface Examples

Two example SCI tests are used to demonstrate the use of the standard control interface. The first test involves a single sphere moving through a sequence of state variable waypoints. The second test demonstrates the use of repeated maneuvers to test multiple feedback gains during a propellant-intensive two-sphere formation rotation.

4.4.1 SCI example 1: single sphere waypoint sequence

The following example illustrates the use of the standard control interface to instruct the sphere to follow a simple sequence of maneuvers, in which each maneuver commands a different target state. The contents of the three files `gsp.h`, `gsp.c`, and `maneuverlist.c` (containing the function `do_maneuver(...)`) are discussed. This test does not require any custom variables, so it is unnecessary to define the custom structure `gsp` in `gsp.h`.

`tests/example1/gsp.h`

```
// this test requires no custom constants or variables
```

This instance of the function `init_gsp()` (in `gsp.c`) includes the required initialization function calls, and initializes several other variables as well. In this case, the three required function calls specify that this sphere is designated Sphere 1, that the token-ring communications should be configured to support only Sphere 1, and that

Sphere 1 is the PADS master. Optional function calls appearing in this example initialize the state estimate and set the control frequency to 10 Hz. In this example the function `free_gsp()` is empty, as no memory is dynamically allocated in `init_gsp()`.

tests/example1/gsp.c

```
// include the maneuver list file
#include "maneuverlist.c"

void init_gsp(void)
{
    // set the sphere ID number
    //  valid values:  SPHERE1, SPHERE2, SPHERE3
    init_sphereID( SPHERE1 );

    // initialize token ring based on spheres being used
    init_comm( TRUE, FALSE, FALSE );     // sphere 1 only

    // choose sphere to be in charge of requesting global updates
    //  valid values:  SPHERE1, SPHERE2, SPHERE3
    set_pads_master( SPHERE1 );

    // initialize state estimate
    init_pos (100.0, 100.0, 100.0);     // [cm]     (global frame)
    init_vel (0.0, 0.0, 0.0);           // [cm/s]   (global frame)
    init_quat(0.0, 0.0, 0.0, 1.0);      // [-]      (normalized)
    init_rate(0.0, 0.0, 0.0);           // [rad/s]  (body frame)

    // initialize other variables and parameters as desired
    ctrl.controlFrequency = 10;
}

void free_gsp(void)
{
    return;
}
```

This instance of `do_maneuver(...)` specifies a sequence of three maneuvers for the sphere to perform. The first maneuver involves travelling to and regulating about the desired initial state. This maneuver is terminated when the position error has been less than or equal to 2.0 cm for at least 3.0 seconds. The second maneuver

specifies a different desired state in the `regulate_specified(···)` command. This second state is offset in angle about the z-axis by 90° from the first state. This maneuver is terminated when 10.0 seconds have elapsed. The third maneuver makes another change to the arguments of `regulate_specified(···)`, telling the sphere to move to the position coordinates (110.0, 80.0, 90.0). This maneuver is terminated when the position error is less than or equal to 1.0 cm for at least 2.0 s, at which point the test is ended by setting `*fTestDone = TRUE`.

`tests/example1/maneuverlist.c`

```
// commands
#include "../../commands/regulate_specified.c"

// controllers
#include "../../controllers/control_attitude_NLPD.c"
#include "../../controllers/control_position_PD.c"

// mixers
#include "../../mixers/mix_simple.c"

// terminators
#include "../../terminators/terminate_elapsed.c"
#include "../../terminators/terminate_hold_pos.c"

void do_maneuver(int  testNum,              // test number
                 int  maneuverNum,          // maneuver number
                 long elapsedTime,          // elapsed maneuver time
                 unsigned int *fTerminate,  // termination flag
                 unsigned int *fTestDone)   // test finished flag
{
    switch (testNum)
    {
    case 1:

        switch (maneuverNum)
        {
        // go to initial state and hold
        case 1:
            regulate_specified(100.0, 100.0, 100.0,  // position
                               0.0, 0.0, 0.0, 1.0);  // quaternion

            // apply control laws to state error
```

```
    control_attitude_NLPD(0.10, 0.04);  // attitude control
    control_position_PD(0.5, 0.2);      // position control

    // set thruster on-times using pulse modulation rules
    mix_simple(15);                     // deadband is 15 ms

    // terminate when position error is <=2.0 cm for >=3.0 s
    terminate_hold_pos(fTerminate, 2.0, 3.0);
    break;                          // don't forget to break!

// rotate 90 degrees about the global (and body) z-axis
case 2:
    regulate_specified(100.0, 100.0, 100.0,  // position
                       0.0, 0.0, 1.0, 0.0);  // quaternion

    // apply control and mixer laws
    control_attitude_NLPD(0.10, 0.04);  // attitude control
    control_position_PD(0.5, 0.2);      // position control
    mix_simple(15);                     // deadband is 15 ms

    // terminate after 10.0 seconds
    terminate_elapsed(fTerminate, 10.0);
    break;                          // don't forget to break!

// translate to new position
case 3:
    regulate_specified(110.0, 80.0, 90.0,    // position
                       0.0, 0.0, 1.0, 0.0);  // quaternion

    // apply control and mixer laws
    control_attitude_NLPD(0.10, 0.04);  // attitude control
    control_position_PD(0.5, 0.2);      // position control
    mix_simple(15);                     // deadband is 15 ms

    // terminate when position error is <=1.0 cm for >=2.0 s
    terminate_hold_pos(fTerminate, 1.0, 2.0);
    break;                          // don't forget to break!

default:
    *fTestDone = TRUE;
    break;
}
```

```
        // break switch on test number
        break;

    } // end switch(testNum)
} // end do_maneuver()
```

4.4.2 SCI example 2: comparing control gain performance during two-sphere circular formation rotation

This example demonstrates maneuver list synchronization between two spheres and the use of an `if` conditional statement on `maneuverNum` to repeatedly execute a sequence of maneuvers. The goal of this test is to characterize the performance of a PD control law with several values of the proportional gain during a propellent-intensive maneuver. Two maneuvers are required to test each value of the gain. The first maneuver is used to achieve desired initial conditions for the second. The second maneuver is a leader-follower formation rotation, where the leader uses a constant set of known effective gains, and the follower uses a set of test gains. This sequence of two maneuvers is executed repeatedly until all specified test gains have been applied.

The formation rotation follows a circular trajectory in the global frame x-y plane. Maintaining a circular trajectory requires constant actuation, and it is desirable to see the effect on performance of changing the position gain in the proportional-derivative position/velocity control law of the follower sphere. During the formation rotation maneuvers, Sphere 1 acts as the leader, tracking a predefined circular trajectory of specified radius, center, and period. Sphere 2 attempts to follow the same circular trajectory, but always offset by π radians from the current angular position of the leader. As in the single sphere example, the control functions `control_attitude_NLPD(...)` and `control_position_PD(...)` are used to determine the force and torque commands, respectively, and the mixer function `mix_simple(...)` determines the actual thruster on-times based on the force and torque commands, the specified deadband, and the thruster geometry. The circular trajectory (odd-numbered) maneuvers are terminated after a specified elapsed time, but the regulation (even-numbered) maneuvers termi-

nate only after a specified state error deadband is met. Termination is then delayed with `wait_for_all(...)` until both spheres have achieved the desired error deadband. When both spheres have achieved the error deadband, the regulation maneuvers of the two spheres are terminated simultaneously. Note that during the delay time when only one sphere has entered the error deadband, the current maneuver module sequence continues to execute at the control frequency; the setting of the termination flag is simply delayed until both spheres are ready. Synchronized termination of the regulation maneuvers ensures that the ensuing trajectory-following maneuvers are synchronized to within the length of the control period.

The leader sphere does not require any custom variables, so it is unnecessary to define the custom structure `gsp` in the leader version of `gsp.h`. The defined quantity `CIRC_PERIOD` is used to choose the period of the circular trajectory.

`tests/example2_leader/gsp.h`

```
#define CIRC_PERIOD   72.0  // formation rotation period, in seconds
```

The follower instance of `gsp.h` includes the custom `float` variable `posGain` in the `gsp` structure, and defines three additional custom constants that are used to choose the test gain values. A total of `NUM_GAINS` values of the position feedback gain will be tested, beginning with `FIRST_GAIN` and incrementing each time by `DELTA_GAIN`.

`tests/example2_follower/gsp.h`

```
// define custom constants here
#define CIRC_PERIOD  72.0   // formation rotation period, in seconds
#define FIRST_GAIN   0.0    // first gain to be tested
#define DELTA_GAIN   0.2    // gain increment
#define NUM_GAINS    6      // number of gains to test

// define custom global variables in this structure
typedef struct
{
    float posGain;   // test value for proportional (position) gain
} gsp_g;

// create gsp structure
gsp_g gsp;
```

The leader and follower versions of `init_gsp()` are identical, with the exception of the argument passed to `init_sphereID(···)`.

tests/example2_leader/gsp.c

```
// include the maneuver list file
#include "maneuverlist.c"

void init_gsp(void)
{
    init_sphereID ( SPHERE1 );        // this is Sphere 1
    init_comm ( TRUE, TRUE, FALSE );  // spheres 1 and 2 are in use
    set_pads_master ( SPHERE1 );
}

void free_gsp(void)
{
    return;
}
```

tests/example2_follower/gsp.c

```
#include "maneuverlist.c"   // include the maneuver list file

void init_gsp(void)
{
    init_sphereID ( SPHERE2 );        // this is Sphere 2
    init_comm ( TRUE, TRUE, FALSE );  // spheres 1 and 2 are in use
    set_pads_master ( SPHERE1 );
}

void free_gsp(void)
{
    return;
}
```

In this test, the leader sphere follows the circular trajectory with identical gains each time, so only two distinct module sequences are required in the leader version of `maneuverlist.c`. The function `telemetry(···)` is used in this example to send the current state of the leader sphere to the follower sphere. The second argument to `telemetry(···)` is `SAT2`, the communications-specific identifier for the sphere having

identification number SPHERE2. Each sphere has a single-bit communications identifier, simplifying the processing of communications data. The communications-specific identifiers SAT1, SAT2, and SAT3 have values equal to 1<<SPHERE1, 1<<SPHERE2, and 1<<SPHERE3, respectively.

tests/example2_leader/maneuverlist.c

```
// commands
#include "../../commands/regulate_polar.c"
#include "../../commands/circle_z.c"

// controllers
#include "../../controllers/control_attitude_NLPD.c"
#include "../../controllers/control_position_PD.c"

// mixers
#include "../../mixers/mix_simple.c"

// terminators
#include "../../terminators/terminate_elapsed.c"
#include "../../terminators/terminate_hold_vel.c"

// maneuver flow control
#include "../../flow_control/wait_for_all.c"

void do_maneuver(int  testNum,      // test number
                 int  maneuverNum,  // maneuver number
                 long elapsedTime,  // elapsed maneuver time
                 int  *fTerminate,  // termination flag
                 int  *fTestDone)   // test finished flag
{
    switch (testNum)
    {
    case 1:

        // place position data in the telemetry queue to SPHERE2
        telemetry(DATA_POS, SAT2);  // comm-specific identifier

        // end test when all gains have been evaluated
        if (maneuverNum > 2*NUM_GAINS)
        {
           *fTestDone = TRUE;
        }
```

```
        // reach desired initial state using standard gains
        else if (maneuverNum % 2)   // odd numbered maneuvers
        {
            regulate_polar(50.0, 0.0,           // r, theta
                           100.0, 100.0, 100.0, // circle center
                           0.0, 0.0, 1.0, 0.0); // quaternion
            control_attitude_NLPD(0.10, 0.04);
            control_position_PD(0.5, 0.2);
            mix_simple(15);

            // null residual velocity
            terminate_hold_vel(fTerminate, 0.5, 2.0);

            // go to next maneuver when all spheres are ready
            wait_for_all(fTerminate,     // termination flag
                         1.0)            // delay for comm latency
        }

        // circle with standard gains
        else                        // even numbered maneuvers
        {
            circle_z(50.0,                 // radius
                     100.0, 100.0, 100.0,  // center of circle
                     0.0,                  // initial angular position
                     elapsedTime,          // elapsed maneuver time
                     CIRC_PERIOD);         // trajectory period

            control_attitude_NLPD(0.10, 0.04);
            control_position_PD(0.5, 0.2);
            mix_simple(15);
            terminate_elapsed(fTerminate, CIRC_PERIOD);
        }

        // break switch on test number
        break;

    } // end switch(testNum)
} // end do_maneuver()
```

The follower version of the maneuver list determines the value of the test gain based on the current maneuver number and the constants defined in `gsp.h`.

tests/example2_follower/maneuverlist.c

```
// commands
#include "../../commands/regulate_polar.c"
#include "../../commands/circle_z_lag.c"

// controllers
#include "../../controllers/control_attitude_NLPD.c"
#include "../../controllers/control_position_PD.c"

// mixers
#include "../../mixers/mix_simple.c"

// terminators
#include "../../terminators/terminate_elapsed.c"
#include "../../terminators/terminate_hold_vel.c"

// maneuver flow control
#include "../../flow_control/wait_for_all.c"

void do_maneuver(int  testNum,      // test number
                 int  maneuverNum,  // maneuver number
                 long elapsedTime,  // elapsed maneuver time
                 int  *fTerminate,  // termination flag
                 int  *fTestDone)   // test finished flag
{
    switch (testNum)
    {
    case 1:

        // end test when all gains have been evaluated
        if (maneuverNum > 2*NUM_GAINS)
        {
           *fTestDone = TRUE;
        }

        // reach desired initial state using known effective gains
        else if (maneuverNum % 2)   // odd numbered maneuvers
        {
            regulate_polar(50.0, PI,             // r, theta
                           100.0, 100.0, 100.0,  // circle center
                           0.0, 0.0, 0.0, 1.0);  // quaternion
            control_attitude_NLPD(0.10, 0.04);
            control_position_PD(0.5, 0.2);
```

```
            mix_simple(15);

            // verify that velocity is small
            terminate_hold_vel(fTerminate, 0.4, 3.0);

            // go to next maneuver when all spheres are ready
            wait_for_all(fTerminate,     // termination flag
                         1.0)            // dummy value
        }

        // follow leader's circle using test gains
        else
        {
            // choose gains for this time through
            gsp.posGain = (maneuverNum/2-1)*DELTA_GAIN + FIRST_GAIN;

            circle_z_lag(50.0,                // radius
                         100.0, 100.0, 100.0, // circle center
                         PI,                  // angular pos. offset
                         SPHERE1);            // leader sphere ID
            control_attitude_NLPD(0.10, 0.04);
            control_position_PD(gsp.posGain, 0.2);
            mix_simple(15);
            terminate_elapsed(fTerminate, CIRC_PERIOD);
        }

        // break switch on test number
        break;

    } // end switch(testNum)
} // end do_maneuver()
```

4.5 Controller Housekeeping

An underlying controller housekeeping algorithm in `control.c` is the foundation on which the standard control interface is built. The housekeeping algorithm handles lower-level and repetitive tasks such resetting maneuver and test variables in response to maneuver termination and test done flags, and managing default control sequences before and after each test. A block diagram outlining the controller housekeeping

algorithm is shown in Figure 4-5. The source code for the housekeeping algorithm is in Section B.3.6 of Appendix B.

4.6 Control Interfaces Summary

A set of three control interfaces has been created, in order to facilitate the use of the SPHERES testbed by remotely located guest scientists. The standard control interface provides a framework into which algorithm modules with pre-defined inputs and outputs are placed. This enables the guest scientist significant freedom in the design of new modules, while maximizing the useability of old code and ensuring that operational requirements are satisfied. The direct interface allows greater freedom in algorithm design, but involves a steeper learning curve. The custom interface is available for guest scientists who wish to completely redesign the SPHERES flight software.

To assist guest scientists in the creation of new SCI modules and tests, a simulation is provided with the GSP interface. This simulation is the subject of Chapter 5.

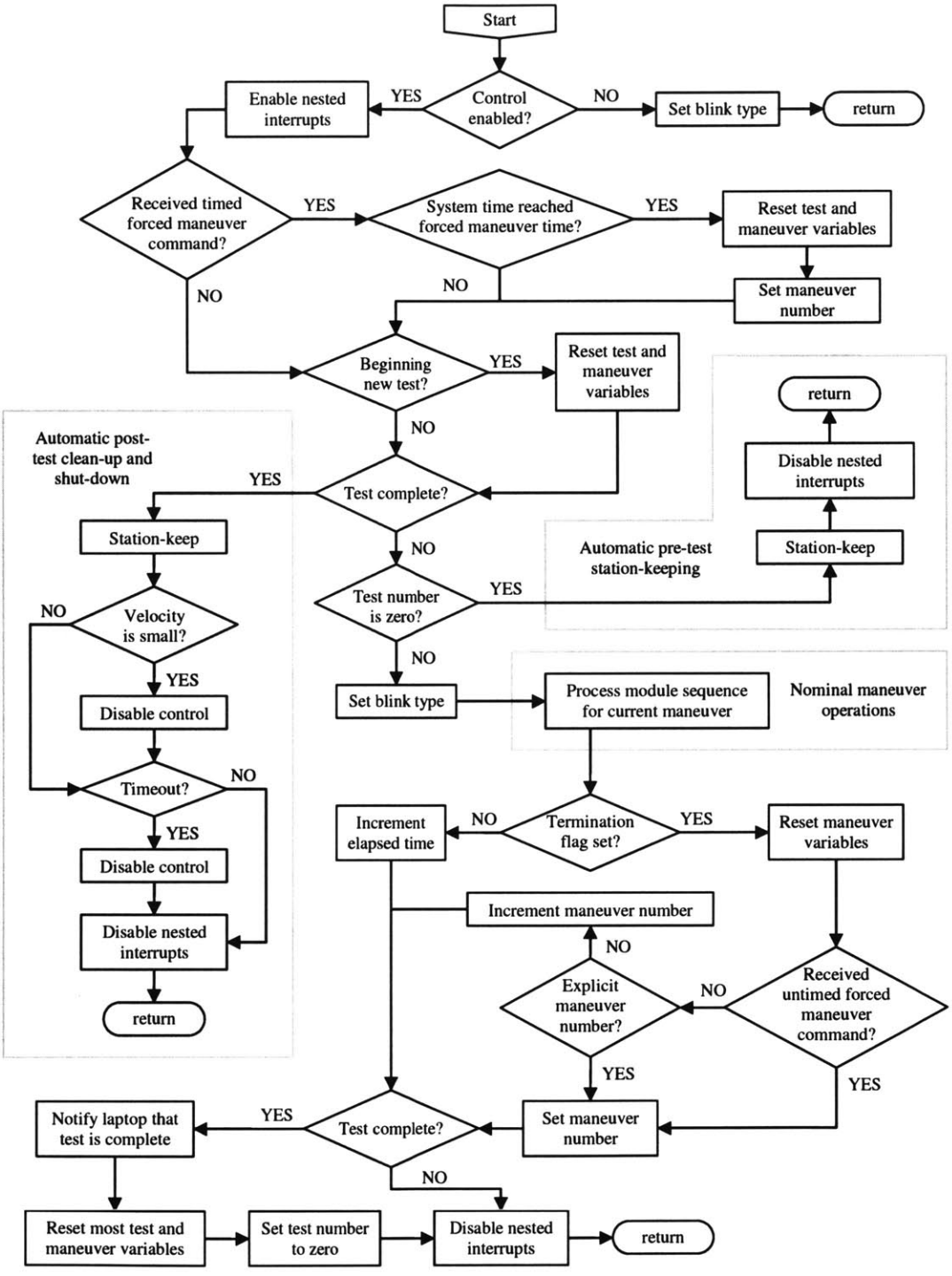


Figure 4-5: The controller housekeeping algorithm handles low-level and repetitive tasks. This flow chart represents a single instance of the control interrupt. Shaded regions indicate the algorithm blocks involving pre-test, test, and post-test maneuvers.

Chapter 5

Guest Scientist Program

5.1 Guest Scientist Program Overview

The goal of the SPHERES Guest Scientist Program (GSP) is to provide sufficient information to allow remote investigators to independently design, code, and debug control and autonomy algorithms for use on the SPHERES testbed. The GSP development environment consists of four parts: the SPHERES GSP simulation, the Generalized FLight Operations Processing Simulator (GFLOPS) SPHERES simulation, the 1-g ground laboratory, and the 0-g International Space Station laboratory. These four parts vary in their degrees of accessibility and fidelity, as shown in Figure 5-1.

The process by which guest scientists develop and implement algorithms is outlined in Figure 5-2. Guest scientists are provided with the information necessary to create new algorithms, and the algorithms are verified using each of the four parts of the development process. The process is recursive, and feedback regarding algo-

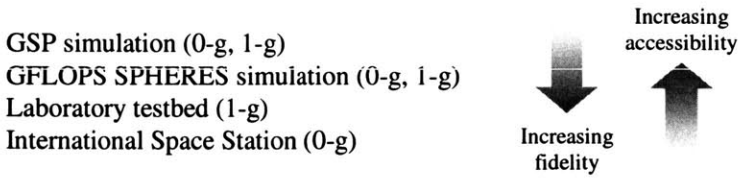


Figure 5-1: Variation in accessibility and fidelity among elements of the SPHERES Guest Scientist Program.

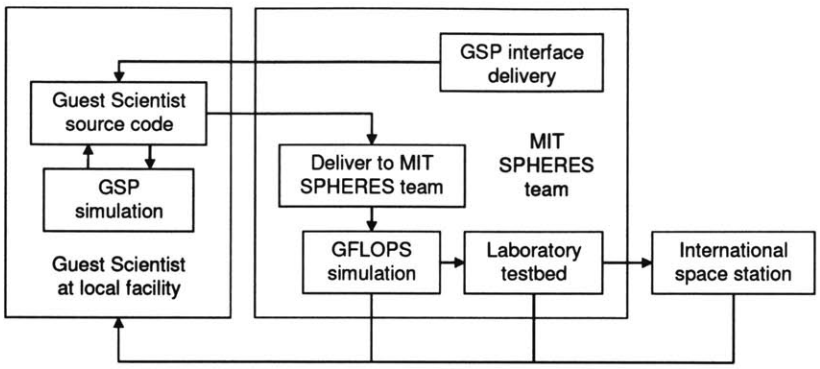


Figure 5-2: High-level overview of the SPHERES Guest Scientist Program development and implementation process.

rithm performance is provided to guest scientists at each test step. Performance data include flight and/or simulation telemetry, and, when possible, video footage.

5.2 SPHERES GSP Simulation

The SPHERES GSP simulation is a tool for the independent development and coding of SPHERES control and autonomy algorithms at MIT and in remote locations. The simulation allows guest scientists to create tests and algorithm modules for the standard and direct control interfaces described in Chapter 4.

The GSP simulation consists of the SPHERES onboard flight source code, and additional code that simulates dynamics, communications, and other environmental interaction. The simulation is designed to significantly reduce or eliminate the need for direct interaction between Guest Scientists and the MIT SPHERES team during early stages of algorithm development and implementation. Successful compilation of guest scientist code and basic algorithm performance in a low-fidelity simulation of the testbed 0-g or 1-g environment can be verified. The simulation supports three-sphere operations, and provides both sphere-to-sphere (STS) and sphere-to-laptop (STL) communication channels.

5.2.1 Simulation files

The GSP simulation consists of four parts: the simulation control panel, the simulation server, source code for sphere applications, and a data reduction script. The control panel and server are pre-compiled executables, and are located in the `<SPHERES>/simulation/` directory, where `<SPHERES>` refers to the root directory containing the sphere flight code files, as depicted in Figure 4-4 on page 76. The sphere application source code consists of the sphere flight software source code, a few simulation-specific replacements for low-level functions, and a wrapper that simulates the test environment.

A new test is created by copying the contents of the `<SPHERES>/tests/template/` directory to a new subdirectory of `<SPHERES>/tests/`. This new test-specific directory should have a descriptive name that reflects the purpose of the test. The project workspace is opened by double-clicking the file `sphere.dsw` in the test-specific directory. The workspace contents are shown in Figure 5-3. The GSP standard interface files `gsp.c`, `gsp.h`, and `maneuverlist.c` are discussed at length in Chapter 4, and the GSP direct interface is accessed through modification of the files `control.c` and `pads.c`. The file `simulation_parameters.txt` is used to specify simulation parameters and initial conditions, and is read at run-time by the sphere application.

5.2.2 System requirements and design trade-offs

The SPHERES GSP simulation requires an x86-compatible personal computer running the Microsoft® Windows 2000 operating system, and Microsoft Visual C++. The control panel, server, and sphere applications communicate using Windows interprocess communication (IPC). This protocol was chosen in order to maximize the amount of actual flight code that is used in the simulation. By using IPC, each sphere can be a separate application, and it is not necessary to modify the variables, function calls, or file inclusions in the flight code to support multiple simultaneous sphere instances within a single executable. Another benefit of IPC is that additional elements may be easily added to the simulation, with very few changes to the existing

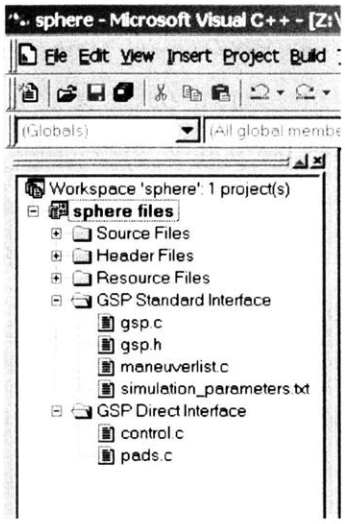


Figure 5-3: GSP simulation project files. User-modifiable files are grouped according to the standard and direct interface guidelines.

applications. Examples of possible additional elements include a run-time telemetry viewer and a run-time 3-D display of the spheres in the test volume. These additional applications simply need to open a handle to the appropriate process pipe, and listen for data. The existing server application need only be modified to include the new application in the data broadcast list. The simulation IPC framework is based on sample code from Microsoft Source Code Samples [17].

5.2.3 Work in progress

The SPHERES GSP simulation is currently incomplete. The IPC framework, user interface, measurement simulator, and telemetry communications are fully functional, but command communications and the dynamics simulation are only partially implemented at this time. As soon as the simulation is complete, the GSP interface package containing descriptions of the control interfaces, sample code, and the GSP simulation will be delivered to guest scientists.

An early incarnation of the GSP simulation was successfully used to test the state update algorithms given in Chapter 3, before implementation was attempted on the testbed hardware. Problems were identified and solutions were applied using the

simulation, resulting in considerable savings in time during the hardware phase of the implementation. This smooth transition from theory to implementation verified the usefulness of the simulation as a development tool.

5.2.4 Simulation server

The GSP simulation server acts as a hub for STS, STL, and simulation-specific communications. The server creates a GSP (simulation-specific) pipe, and then waits for a client to connect, as shown in the process flow chart of Figure 5-4. If a sphere connects, STS and STL pipes are created. When clients are connected, the server reads each pipe and redirects messages as appropriate to the other clients. For example, an STS message sent to the server by a sphere is then broadcast by the server to all spheres except for the message originator, simulating the behavior of radio-frequency communications and enabling inter-sphere communication. Commands sent from the control panel are rebroadcast by the server to all connected spheres.

The server is shown in the lower-left corner of Figure 5-5. The control panel is shown in the top half of the figure, and three sphere processes can be seen to the right of the server application. Shortcuts to the simulation executables are present on the Windows desktop. When the control panel or a sphere connects to the server, the corresponding icon changes from grey-scale to color. This visual feedback makes it easy for the user to always know which spheres are connected to the server. During simulation run-time, the server controls the simulation time, ensuring that all spheres have completed a given time step before commanding the next one.

5.2.5 Control panel

In order to accurately model the behavior of the spheres during a test, the simulation requires user input to separately power on the spheres and enable the onboard controllers (see Section 4.2.4 for additional details regarding enabling the controller). The control panel graphical user interface (GUI) is used to perform these tasks, and to interface with the spheres during simulation run-time. The control panel GUI is

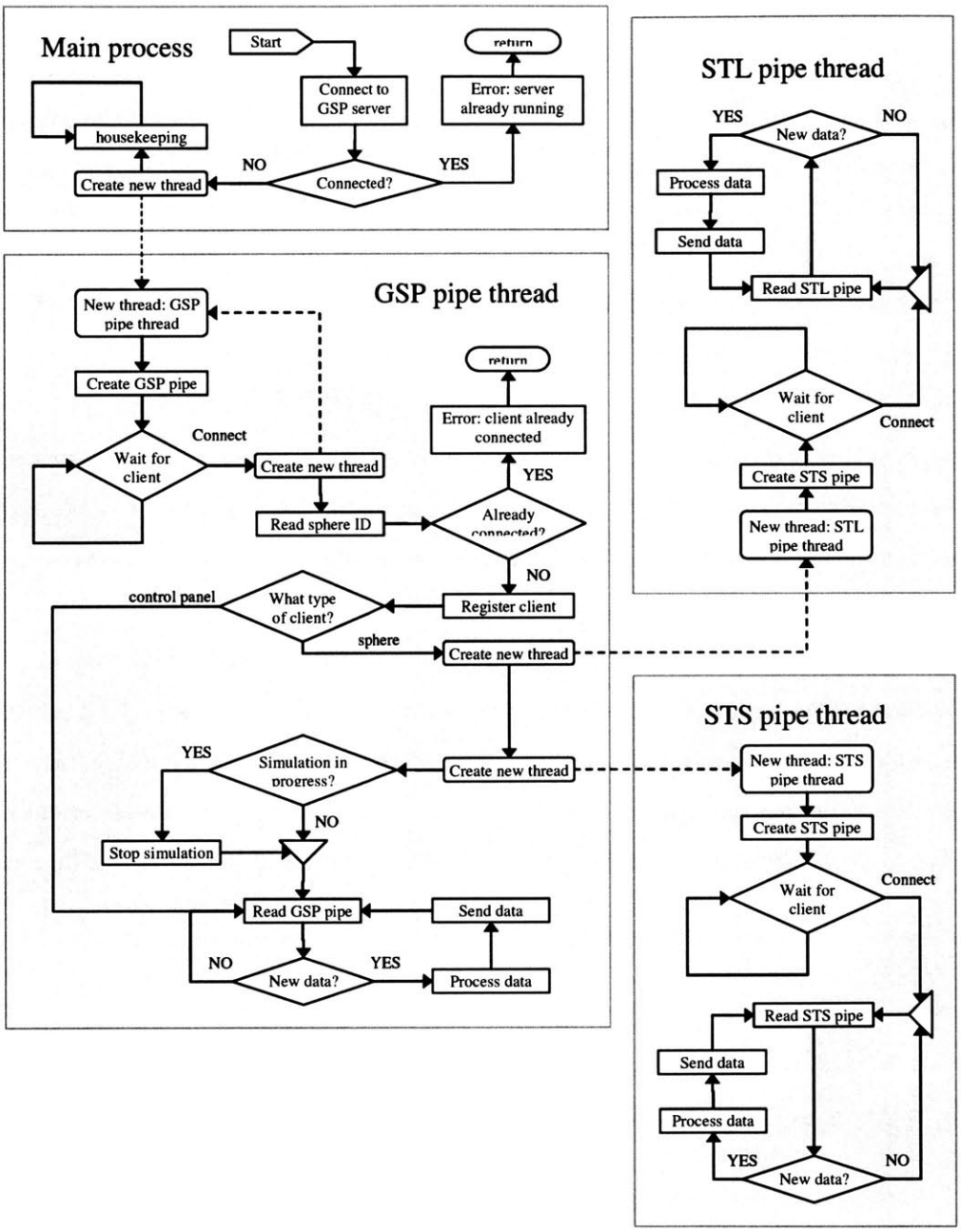


Figure 5-4: High-level block diagram of GSP Simulation server application.

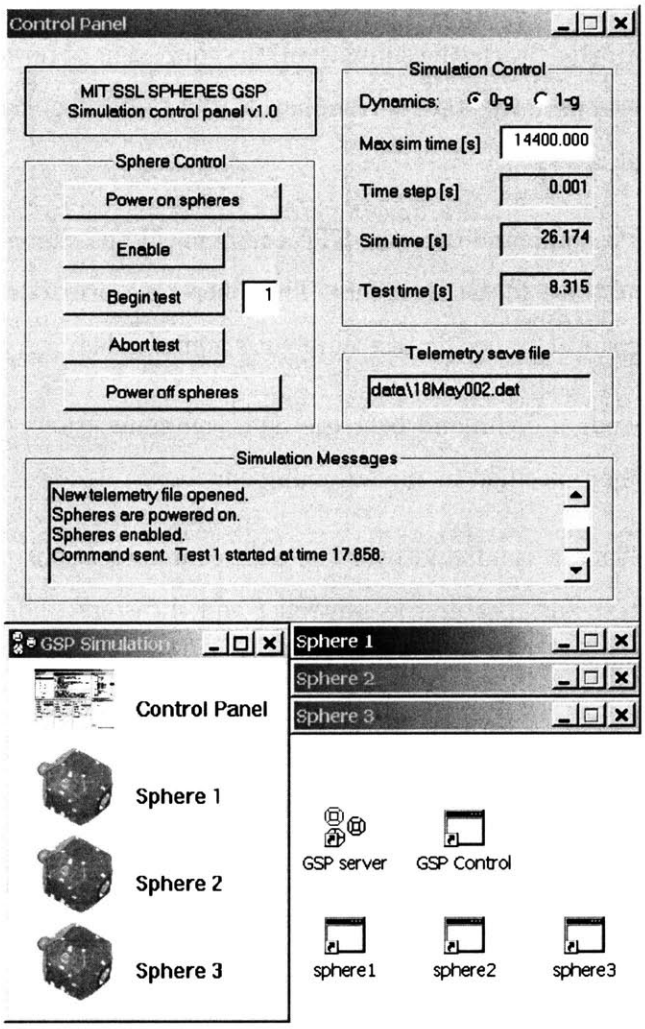


Figure 5-5: The GSP simulation control panel graphical user interface, simulation server, and three sphere applications. Shortcuts to the executables can be seen on the Windows desktop.

shown in the top half of Figure 5-5. Buttons on the "Sphere Control" section of the control panel are used to power on and enable the spheres, and then to send commands over the simulated STL communications channel. The following five buttons on the control panel are used to control the simulation.

Power on spheres Starts the simulation, the equivalent of powering on the spheres. The spheres perform state determination, but perform no control until an enable command is sent.

Enable Sends a command over the STL communications channel to enable the controller and the cold-gas thrusters. The spheres perform a default station-keeping maneuver until a specific test number is commanded.

Begin test Sends a command over the STL communications channel to begin the test number specified in the adjoining edit box.

Abort test Sends a command over the STL communications channel to abort the current test and disable the controller and thrusters. The spheres drift freely, performing state determination but no control.

Power off spheres Ends the simulation, equivalent to powering off the spheres.

The "Simulation Control" section of the control panel GUI allows the user to view and modify some simulation parameters. Two different time counters are used to track the simulation progress. The **Sim time** counter tracks the number of simulated seconds that have elapsed since the spheres were powered on. The **Test time** counter tracks the number of simulated seconds that have elapsed since the most recent **Begin test** command was sent. The **Telemetry save file** specifies the relative path (from `<SPHERES>/simulation/`) to the file where telemetry from the current test is located. The file number increments whenever a new **Begin test** command is sent.

A high-level view of the control panel algorithm is shown in the process block diagram of Figure 5-6. The control panel attempts to connect to the GSP pipe created by the server. Upon connection, it enters an infinite loop where it reads

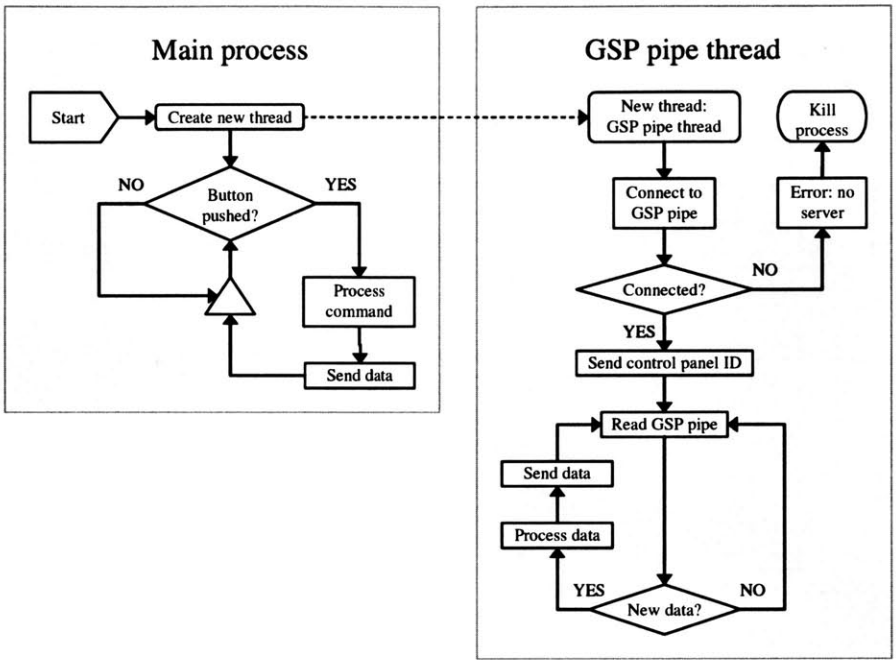


Figure 5-6: High-level block diagram of GSP Simulation control panel application.

and processes information received through the GSP pipe. When the user presses a button, the control panel sends the corresponding command through the GSP pipe to the server application.

5.2.6 Sphere executables

The GSP simulation supports up to three simultaneous sphere processes, the number of spheres that will be used on the International Space Station. A sphere executable is created by opening the workspace file `sphere.dsw` and pressing the Build button in Microsoft Visual C++. Modifications can be made to the standard control interface or direct control interface files following the guidelines in Chapter 4 before building the executable. The sphere flight code files are included with `#include` in the file `simulation.c`, which takes the place of `main.c` in the simulation code. The function `simTimeStep(...)` in `simulation.c` is called once during each simulation time step. This function calls the various interrupt and background process functions at appropriate times, in order to simulate the timed-interrupt nature of the digital signal

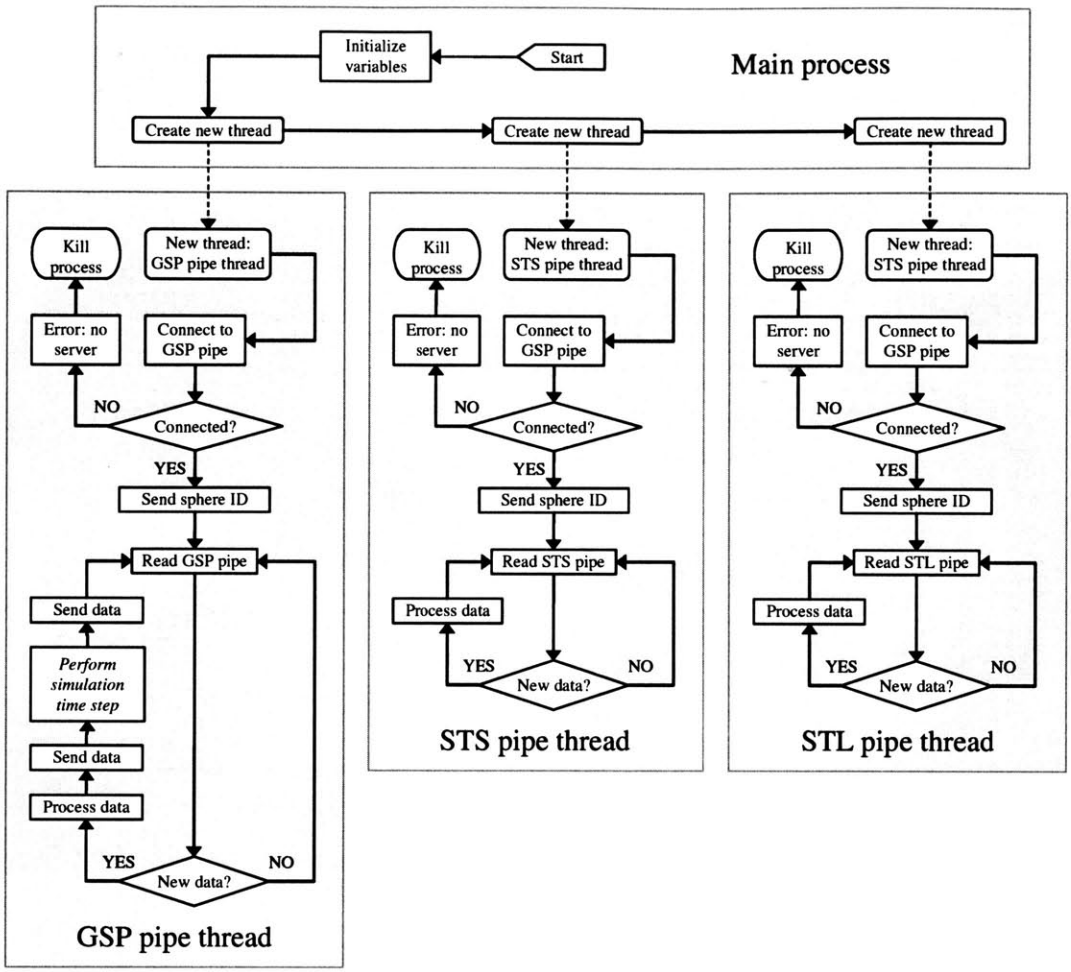


Figure 5-7: High-level block diagram of GSP Simulation sphere application.

processor.

The file `sphere.c` contains the wrapper code that communicates with the server and calls `simTimeStep(...)`. A high-level block diagram of the tasks handled by the functions in `sphere.c` is given in Figure 5-7. Simulation-specific functions replace low-level flight software functions that access hardware directly, and provide simulated communications queues and sensor readings. The dynamics of each sphere are simulated by the wrapper code, rather than by the server, to minimize the message traffic on the GSP pipe. Three instances of sphere processes (spheres 1, 2, and 3) are shown below the control panel and to the right of the server in Figure 5-5.

5.2.7 Data reduction

The simulation server records all received telemetry to a text file. Telemetry data include estimated and actual state information, and may include any other data of interest to the guest scientist. A MATLAB® script m-file is included with the GSP interface, to be used for reduction of the telemetry data. The script produces time-history plots of the state and measurement information.

5.3 GFLOPS SPHERES Simulation

After guest scientist code compilation and basic algorithm performance are demonstrated with the GSP simulation, the code is verified using a SPHERES simulation running on the high-fidelity Generalized FLight Operations Processing Simulator (GFLOPS) [8]. The GFLOPS testbed consists of eight networked PowerPC single-board computers, running a real-time operating system. The real-time, multi-processor nature of this testbed provides advantages over the GSP simulation. The GFLOPS testbed better represents the asynchronous interrupt and background process timing of the individual spheres' digital signal processors, and provides a higher-fidelity model of the expected laboratory and on-orbit disturbance environments. Use of the GFLOPS testbed requires that guest scientists deliver to the MIT SPHERES team code that has been tested using the GSP simulation. The GFLOPS simulation is a verification step in the development process, and test results are returned to guest scientists so that algorithm performance may be improved through iterations of the GSP simulation to GFLOPS simulation cycle. The GFLOPS SPHERES simulation is described in detail by Radcliffe [19].

5.4 Laboratory Testbed

The laboratory testbed provides accessible hardware for implementation of developed algorithms. This hardware is identical to the ISS flight hardware, and realistic imperfections, uncertainties, and unmodeled disturbances will be present.

In the laboratory, the SPHERES testbed may be used in two configurations. In the laboratory standard configuration, the spheres are mounted on air carriages, and suspended by compressed air on a 1.25 m $\times$ 1.25 m glass surface. The PADS beacons are mounted to optimize coverage over the 2-D test surface. The laboratory standard configuration is limited to one rotational and two translational degrees of freedom.

In the laboratory station configuration, the PADS beacons are mounted approximately as they will be in orbit aboard the ISS. One or more spheres may be suspended in the test volume to verify correct state determination in 3-D. Rotational maneuvers about a single axis may be performed, but the primary purpose of this configuration is to verify the performance of state determination algorithms.

5.5 International Space Station

The SPHERES ISS testbed provides a risk-tolerant, long-duration, representative dynamic environment for the validation of control, metrology, and autonomy algorithms for spacecraft formation flight, rendezvous, and docking. Accessibility to the testbed by the MIT SPHERES team and guest scientists is limited to software changes and the addition of accessory hardware via an expansion port on each sphere. The microgravity environment of the ISS allows for 6-DOF maneuvers. The useable test volume in the ISS is as yet undetermined, and may range from a 2m $\times$ 2m $\times$ 2m cube to a 1.5m $\times$ 1.5m $\times$ 3m box.

Chapter 6

Conclusions and Recommendations

6.1 Thesis Summary

This thesis has examined the components of the SPHERES testbed that are relevant to control algorithm development. The topics addressed in each chapter are summarized below.

Chapter 2 presented an overview of the SPHERES testbed and the sphere vehicle subsystems relevant to control system design. A pulse modulation scheme was developed to overcome some of the physical limitations of the non-linear propulsion system actuators. Mapping algorithms for the transformation of control commands to and from thruster on-times were developed for the prototype and flight sphere geometries.

Chapter 3 described the Position and Attitude Determination Subsystem hardware, and the methodology used to measure the sphere state with respect to the laboratory reference frame. A memoryless optimal quaternion algorithm was used to determine the sphere attitude from direction vectors, and a method was given for the determination of those vectors from raw distance measurements. A Kalman filter was used to update the position and velocity components of the state estimate. Suggestions were made for improvements to the algorithms.

Chapter 4 presented the SPHERES Standard Control Interface, designed to facilitate rapid test development through the use of modular algorithm blocks with pre-defined inputs and outputs. The SCI enforces rules to ensure code compatibility,

while allowing freedom in the design of individual algorithm modules. The different types of algorithm modules are explained, and two examples are used to clarify the presentation.

Chapter 5 briefly described the SPHERES Guest Scientist Program simulation, a tool for the development of SPHERES algorithms. The GSP simulation provides a means for guest scientists to implement algorithms in the flight code environment, without requiring access to the flight hardware.

6.2 Conclusions

The work presented in this thesis addresses all and satisfies most of the objectives specified in Section 1.3.

Models were formulated to describe the sphere components. Sections 2.2 and 2.3 addressed the cold-gas thrusters that serve as the sphere actuators. A model of the sphere dynamics was developed in Chapter 3, and the control system interface was presented in Chapter 4.

A state estimator was developed in Chapter 3. The estimator provides real-time estimates of the sphere position, velocity, attitude, and angular rate. The performance of this estimator has been verified in the laboratory and in micro-gravity aboard NASA's KC-135 aircraft, but several possible improvements remain to be investigated.

A simple, flexible user interface to the sphere flight hardware and software was developed. The Standard Control Interface utilizes a modular approach to maneuver and test design. The SCI specifies rules for module inputs and outputs, but allows complete freedom of design within individual modules. Modules are arranged in source code in a list format, facilitating the rapid recognition of test contents. An underlying controller housekeeping algorithm performs low-level background tasks without the need for user interaction, and ensures that ISS requirements are satisfied. Support for fundamental tasks such as maneuver synchronization is provided by pre-defined modules.

A simulation for use in the development of control and autonomy algorithms was

partially developed. The simulation uses the SPHERES onboard flight code in its entirety, with the exception of a few functions that directly access flight hardware, and are replaced by simulation-specific versions. The flight software runs in a wrapper that handles communications, simulation timing, and the sphere dynamics. The simulation is not yet complete.

6.3 Future Work

Recommendations for future work are as follows:

- Complete the GSP simulation and associated documentation.
- Improve the state update implementation, making use of the redundant ultrasonic receivers, higher communications bandwidth, and upgraded processing power of the flight sphere. Incorporate attitude into the Kalman filter, and design and test an incremental Kalman filter to process direction vectors as they are calculated, rather than processing ranges after all measurements have been received.
- Better characterize the process and measurement noise to improve the performance of the existing and future Kalman filters.
- The flight sphere has a transmitter for use in direct inter-sphere ranging. These direct range measurements should be incorporated into the Kalman filter.

Appendix A

Quaternions

A.1 Properties of the attitude quaternion

Euler's Theorem states that "the most general displacement of a rigid body with one point fixed is a rotation about some axis [26]." This rotation may be quantified by describing the axis of rotation $\boldsymbol{n}$ and the angle of rotation θ. The following discussion will outline quaternion mathematics and the specific conventions followed in the SPHERES onboard code for the use of the attitude quaternion in the representation of three-space orientation.

The four-element quaternion $\tilde{\mathbf{q}}$ may be used to represent an arbitrary rotation or orientation in three-space. The quaternion consists of a three-element hyper-imaginary "vector" part and a single-element scalar part, viz. [9],

$$\tilde{\mathbf{q}} \equiv q_1\hat{\mathrm{i}} + q_2\hat{\mathrm{j}} + q_3\hat{\mathrm{k}} + q_4 \tag{A.1}$$

where the quantities $\hat{\mathrm{i}}$, $\hat{\mathrm{j}}$, $\hat{\mathrm{k}}$ follow a set of rules analogous to the single-dimension imaginary number $\mathrm{i} \equiv \sqrt{-1}$, and similar in form to the rules for forming cross prod-

ucts.

$$\hat{\imath}^2 = \hat{\jmath}^2 = \hat{k}^2 = -1 \tag{A.2}$$

$$\hat{\imath}\hat{\jmath} = \hat{k} = -\hat{\jmath}\hat{\imath} \tag{A.3}$$

$$\hat{\jmath}\hat{k} = \hat{\imath} = -\hat{k}\hat{\jmath} \tag{A.4}$$

$$\hat{k}\hat{\imath} = \hat{\jmath} = -\hat{\imath}\hat{k} \tag{A.5}$$

It is often convenient to utilize a real-vector expression of the quaternion, when the hyper-imaginary nature has been accounted for elsewhere. The real coefficients of the quaternion components may be expressed in vector notation as

$$\mathbf{q} \equiv \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix}^T \tag{A.6}$$

A relatively simple physical interpretation of the real instantiation of the attitude quaternion may be made through the following definition. Given a rigid-body rotation of angle θ about the axis $\boldsymbol{n}$ expressed in some reference frame, the resulting orientation of the body may be characterized by

$$\mathbf{q} = \begin{bmatrix} \boldsymbol{q} \\ q_4 \end{bmatrix} = \begin{bmatrix} \boldsymbol{n}\sin(\frac{\theta}{2}) \\ \cos(\frac{\theta}{2}) \end{bmatrix} = \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix}^T \tag{A.7}$$

From the physical interpretation of the attitude quaternion, it can be seen that a rotation of angle θ about the unit vector $\boldsymbol{n}$ followed by a rotation of angle $-\theta$ about $\boldsymbol{n}$ results in zero net change in attitude, so the inverse of a quaternion may be found simply by changing the sign on the vector part.

$$\begin{bmatrix} \boldsymbol{q} \\ q_4 \end{bmatrix}^{-1} = \begin{bmatrix} -\boldsymbol{q} \\ q_4 \end{bmatrix} \tag{A.8}$$

It is also apparent from Equation A.7 that the length of a quaternion must always be equal to identity, since

$$\begin{aligned}\sqrt{\mathbf{q}^T\mathbf{q}} &= \sqrt{q_1^2 + q_2^2 + q_3^2 + q_4^2} \\ &= \sqrt{\boldsymbol{n}^T\boldsymbol{n}\sin^2\left(\frac{\theta}{2}\right) + \cos^2\left(\frac{\theta}{2}\right)} \\ &= \sqrt{\sin^2\left(\frac{\theta}{2}\right) + \cos^2\left(\frac{\theta}{2}\right)} \\ &= 1 \end{aligned} \tag{A.9}$$

The quaternion must be periodically re-normalized by dividing by its length, in order to maintain this property in the presence of round-off errors incurred through digital computation.

According to Euler's theorem, the attitude of the body frame with respect to the reference frame can be specified by the rotation (represented by $\mathbf{q}$) that transforms the reference frame into the body frame. The 3×3 matrix Θ that rotates a vector from the reference frame into the body frame can be written in terms of the attitude quaternion as

$$\Theta(\mathbf{q}) = \left(q_4^2 - \boldsymbol{q}^T\boldsymbol{q}\right) \mathrm{I}_{3\times 3} + 2\boldsymbol{q}\boldsymbol{q}^T - 2q_4\left[\boldsymbol{q}\times\right] \tag{A.10}$$

using shorthand notation based on the cross product operator [6]. The cross product $\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$ is expressed as $\boldsymbol{c} = [\boldsymbol{a}\times]\boldsymbol{b}$ for the matrix $[\boldsymbol{a}\times]$ defined as

$$[\boldsymbol{a}\times] \equiv \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \tag{A.11}$$

The reference to body frame rotation matrix expanded in terms of the quaternion elements is

$$\Theta(\mathbf{q}) = \begin{bmatrix} q_1^2 - q_2^2 - q_3^2 + q_4^2 & 2(q_1q_2 + q_3q_4) & 2(q_1q_3 - q_2q_4) \\ 2(q_1q_2 - q_3q_4) & -q_1^2 + q_2^2 - q_3^2 + q_4^2 & 2(q_2q_3 + q_1q_4) \\ 2(q_1q_3 + q_2q_4) & 2(q_2q_3 - q_1q_4) & -q_1^2 - q_2^2 + q_3^2 + q_4^2 \end{bmatrix} \tag{A.12}$$

The product of two quaternions may be found using complex algebra with the definition of Equation A.1 and the rules of Equations A.2 through A.5, viz. [9],

$$\begin{aligned} \tilde{\mathbf{q}}'' &= \tilde{\mathbf{q}}\tilde{\mathbf{q}}' && \text{(A.13)} \\ &= \hat{\imath}(q_1q_4' + q_2q_3' - q_3q_2' + q_4q_1') \\ &\quad +\hat{\jmath}(-q_1q_3' + q_2q_4' + q_3q_1' + q_4q_2') \\ &\quad +\hat{\mathrm{k}}(q_1q_2' - q_2q_1' + q_3q_4' + q_4q_3') \\ &\quad -q_1q_1' - q_2q_2' - q_3q_3' + q_4q_4' && \text{(A.14)} \end{aligned}$$

The resulting quaternion $\tilde{\mathbf{q}}''$ represents the rotation of a rigid body through the rotation defined by the quaternion $\tilde{\mathbf{q}}$, and then through the rotation defined by the quaternion $\tilde{\mathbf{q}}'$ [9]. It can be verified using Equations A.12 and A.14 that quaternion multiplication and rotation matrix multiplication have opposite order of operations; quaternions operate from left to right, and rotation matrices operate from right to left.

$$\Theta(\mathbf{qq}') = \Theta(\mathbf{q}')\Theta(\mathbf{q}) \tag{A.15}$$

The text "Rotations, Quaternions, and Double Groups" by Altmann deals extensively with quaternions, and is recommended reading for advanced information [1].

A.2 Quaternion composition

To minimize confusion over order of operations, it is desirable to find a way to express quaternion multiplication such that the multiplication of two quaternions corresponds

in order of operations to the multiplication of the corresponding rotation matrices. The relationship of Equation A.14 may be expressed using matrix notation as

$$\mathbf{q}'' = \begin{bmatrix} q'_4 & q'_3 & -q'_2 & q'_1 \\ -q'_3 & q'_4 & q'_1 & q'_2 \\ q'_2 & -q'_1 & q'_4 & q'_3 \\ -q'_1 & -q'_2 & -q'_3 & q'_4 \end{bmatrix} \mathbf{q} \tag{A.16}$$

The appearance of the first (in order of operations) rotation $\mathbf{q}$ on the far right hand side of this equation suggests the definition of a quaternion composition operator $*$, where composition is defined in terms of a matrix-vector multiplication.

$$\mathbf{q}'' \equiv \mathbf{q}' * \mathbf{q} \tag{A.17}$$

$$= [\mathbf{q}'*]\,\mathbf{q} \tag{A.18}$$

where for simplicity in notation and to enable the use of matrix mathematics, the quantity $[\mathbf{q}'*]$ is defined as

$$[\mathbf{q}'*] \equiv \begin{bmatrix} q'_4 & q'_3 & -q'_2 & q'_1 \\ -q'_3 & q'_4 & q'_1 & q'_2 \\ q'_2 & -q'_1 & q'_4 & q'_3 \\ -q'_1 & -q'_2 & -q'_3 & q'_4 \end{bmatrix} \tag{A.19}$$

$$= q'_4 \mathrm{I}_{4\times 4} + \begin{bmatrix} -[\boldsymbol{q}'\times] & \boldsymbol{q}' \\ -(\boldsymbol{q}')^T & 0 \end{bmatrix} \tag{A.20}$$

Given this new composition operator, successive quaternion rotations may be written in the same order as successive rotation matrix rotations.

$$\Theta(\mathbf{q}' * \mathbf{q}) \equiv \Theta(\mathbf{q}')\Theta(\mathbf{q}) \tag{A.21}$$

The approach of redefinition of quaternion multiplication in terms of a composition operator, in order to follow the order of operations of rotation matrix multiplication,

has been taken by Markley and others.

Note from Equations A.17 and A.19 that each step in successive quaternion composition involves $4 \times 4 = 16$ multiplicative and $4 \times 4 = 16$ additive operations. Each step in multiplication of successive rotation matrices requires $3 \times 3 \times 3 = 27$ multiplicative and $3 \times 3 \times 3 = 27$ additive operations, so it is significantly more efficient to perform successive rotations using quaternion composition than using rotation matrices.

A.3 The error quaternion

The convention used in the SPHERES code is that error in a state quantity is defined as the change that must be made to the current state in order to reach the desired state. Using the example of position $\boldsymbol{r}$, desired position $\boldsymbol{r}_d$, and position error $\boldsymbol{r}_e$, the error may be defined such that $\boldsymbol{r}_d = \boldsymbol{r}_e + \boldsymbol{r}$. Following this convention, the attitude error may be expressed in terms of rotation matrices as

$$\Theta(\mathbf{q}_d) = \Theta(\mathbf{q}_e)\Theta(\mathbf{q}) \tag{A.22}$$

in which the error rotation $\Theta(\mathbf{q}_e)$ is applied to the current attitude in order to achieve the desired attitude. Solving for the error results in

$$\begin{aligned} \Theta(\mathbf{q}_e) &= \Theta(\mathbf{q}_d)\Theta^{-1}(\mathbf{q}) && \text{(A.23)} \\ &= \Theta(\mathbf{q}_d)\Theta(\mathbf{q}^{-1}) && \text{(A.24)} \end{aligned}$$

where the quaternion inverse was defined by Equation A.8. The error quaternion may therefore be written using quaternion multiplication or quaternion composition, respectively, as

$$\begin{aligned} \tilde{\mathbf{q}}_e &= \tilde{\mathbf{q}}^{-1}\tilde{\mathbf{q}}_d && \text{(A.25)} \\ \mathbf{q}_e &= \mathbf{q}_d * \mathbf{q}^{-1} && \text{(A.26)} \end{aligned}$$

A.4 Quaternion propagation using body rates

Given measurements or estimates of body-frame rates, such as may be sampled directly with rate gyroscopes, the attitude quaternion may be propagated in time with

$$\dot{\mathbf{q}} = \frac{1}{2}\Omega(\boldsymbol{\omega})\mathbf{q} \tag{A.27}$$

where the matrix $\frac{1}{2}\Omega(\boldsymbol{\omega})$ maps the quaternion into its derivative based on the body rates $\boldsymbol{\omega}(t) = \begin{bmatrix} \omega_x(t) & \omega_y(t) & \omega_z(t) \end{bmatrix}^T$ [26]:

$$\Omega(\boldsymbol{\omega}) \equiv \begin{bmatrix} 0 & \omega_z & -\omega_y & \omega_x \\ -\omega_z & 0 & \omega_x & \omega_y \\ \omega_y & -\omega_x & 0 & \omega_z \\ -\omega_x & -\omega_y & -\omega_z & 0 \end{bmatrix} \tag{A.28}$$

Note that the matrix $\Omega(\boldsymbol{\omega})$ is dependent on the time-varying body rates, and is therefore non-constant in time.

Appendix B

Guest Scientist Program Reference

B.1 Defined Quantities

Several commonly-used quantities are preprocessor defined to facilitate source code organization and increase readability. Table B.1 lists the defined indices that may be used to access elements of the state vectors `pads.state`, `ctrl.stateTarget`, `ctrl.stateError`, `pads.state1`, `pads.state2`, and `pads.state3`. Table B.2 lists the indices that may be used to access elements of the control array `ctrl.control`, which contains force and torque commands. Defined quantities are capitalized to enhance recognition.

B.2 Global Variables

All global variables in the onboard code are contained in the global structures listed in Table B.2. The contents of several of these structures are listed in the following discussion, and selected variables are described in detail. Standard ANSI C zero-offset arrays are of type `int`, `unsigned int` (`uint`), `long`, or `float`, while dynamically allocated single offset vectors and matrices are of type `int*`, `float*`, `int**`, `float**`, etc. Distance is measured in centimeters, speed in centimeters per second, angle in radians, and temperature in degrees Celsius. The contents of the `ctrl`, `pads`, `sys`, and `prop` structures are listed in full in the following discussion, and selected contents

Table B.1: Indices into state vectors.

Quantity	Description
POS_X	x-component of position
POS_Y	y-component of position
POS_Z	z-component of position
VEL_X	x-component of velocity
VEL_Y	y-component of velocity
VEL_Z	z-component of velocity
QUAT_1	first vector component of quaternion
QUAT_2	second vector component of quaternion
QUAT_3	third vector component of quaternion
QUAT_4	scalar component of quaternion
RATE_X	x-component of angular rate
RATE_Y	y-component of angular rate
RATE_Z	z-component of angular rate

Table B.2: Indices into the control array `ctrl.control`.

Quantity	Description
FORCE_X	x-component of force command
FORCE_Y	y-component of force command
FORCE_Z	z-component of force command
TORQUE_X	x-component of torque command
TORQUE_Y	y-component of torque command
TORQUE_Z	z-component of torque command

Table B.3: Contents of the global structure `ctrl`.

Type	Name	Description
`uint`	`cDelayControl`	delay enabling of control
`uint`	`fDoControl`	enable control flag
`uint`	`fManeuverDone`	maneuver complete flag
`uint`	`fManeuverReset`	reset maneuver variables
`uint`	`fTestReset`	reset test variables
`uint`	`fEnableAngX`	enable x-axis angle control
`uint`	`fEnableAngY`	enable y-axis angle control
`uint`	`fEnableAngZ`	enable z-axis angle control
`uint`	`fEnableRateX`	enable x-axis rate control
`uint`	`fEnableRateY`	enable y-axis rate control
`uint`	`fEnableRateZ`	enable z-axis rate control
`uint`	`fEnablePosX`	enable x-axis position control
`uint`	`fEnablePosY`	enable y-axis position control
`uint`	`fEnablePosZ`	enable z-axis position control
`uint`	`fEnableVelX`	enable x-axis velocity control
`uint`	`fEnableVelY`	enable y-axis velocity control
`uint`	`fEnableVelZ`	enable z-axis velocity control
`uint`	`fReady`	readiness flags for all spheres
`int`	`testNum`	current test number
`int`	`maneuverNum`	current maneuver number
`int`	`nextManeuver`	next maneuver number
`int`	`controlFrequency`	control frequency
`float`	`testTime`	test elapsed time
`float`	`maneuverTime`	maneuver elapsed time
`float`	`control`	force & torque control array
`float*`	`stateTarget`	current desired state vector
`float*`	`stateError`	current state error vector

of these structures are described in detail.

B.2.1 Control data structure: `ctrl`

The global variables contained in the global control structure `ctrl` are listed in Table B.3. A subset of those variables are described in detail in the following discussion.

`(float*) stateTarget[state index]` is the current desired state vector. The contents of this vector are written by the command module, and are compared to the current state estimate to generate the state error vector. Valid indices into

`ctrl.stateTarget` are the defined state indices of Table B.1.

`(float*) stateError[state index]` is the current state error, the difference between the desired state and the actual state. This vector is generated in the command function from the state target vector `ctrl.stateTarget` and the current state estimate `pads.state` through a call to the function `find_error(...)`. Valid indices into `ctrl.stateError` are the defined state indices of Table B.1.

`(uint) fReady[sphereID]` contains `TRUE/FALSE` flags specifying which spheres have signalled a state of readiness.

`(int) controlFrequency` is the control interrupt frequency, in units of Hz. The default value is 10 Hz, but the value may be changed at any time, up to a maximum of 25 Hz.

`(float) control[control index]` contains three force commands, represented in the global frame, and three torque commands, represented in the body frame. Valid indices into `ctrl.control` are the defined control indices of Table B.2.

`(uint) maneuverTime` is the elapsed time since the beginning of the current maneuver.

B.2.2 PADS data structure: pads

The global variables contained in the global PADS structure `pads` are listed in Table B.4, and are used for position and attitude determination. A subset of those variables are described in detail in the following discussion. PADS arrays have indices organized by a subset of the order [transmitter][face][receiver][dimension].

`(float*) state[state index]` is the current estimated state vector. Valid indices into `pads.state` are the defined state indices of Table B.1.

`(float) matrix[transmitter][face][receiver]` begins as the array of raw distance measurements. The values in `pads.matrix` are overwritten during the distance matrix correction routine called from within `pads_global(...)`.

Table B.4: Contents of the global structure pads.

Type	Name	Description
uint	fGotGlobal	have good state estimate
uint	fSendGlobal	download distance matrix
uint	fUpdateAttitude	perform attitude updates
uint	fUpdatePosition	perform pos/vel updates
uint	fBiasReady	bias calculation is complete
uint	fGlobalTime	listening for ultrasound
uint	fNewMatrix	have a new distance matrix
uint	cGlobal	cumulative number of IR flashes
long	tstampMatrix	time stamp of last matrix
long	tstampIMU	time stamp of last IMU reading
long	tstampIMU_onboard	local clock version of tstampIMU
int	IMU_dt	PADS local delta time
int	localPeriod	PADS local period
int	globalPeriod	PADS global period
int**	txVecUse	measurement validity matrix
float	IMU_raw	raw IMU data from FPGA
float	mass	mass of sphere
float	bias	gyro. and acc. bias terms
float	temperature	ambient temperature
float	speedOfSound	speed of sound
float	convGlobal	distance conversion factor
float	matrix	PADS global distance matrix
float*	state	current state estimate vector
float*	state1	sphere 1 state estimate vector
float*	state2	sphere 2 state estimate vector
float*	state3	sphere 3 state estimate vector
float*	stateRoughPrev	previous rough position estimate
float**	stateDynamics	state dynamics matrix
float**	dynamicsAtt	attitude dynamics submatrix
float**	dynamicsPos	pos/vel dynamics submatrix
float**	Q_att	attitude noise matrix
float**	Q_pos	position/velocity noise matrix
float**	body2Glo	body to global frame rotation matrix
float**	covPos	position/velocity covariance matrix
float**	covAtt	attitude covariance matrix
float**	inertia	inertial matrix
float**	inertiaInv	inverse of inertia matrix
float**	rxAng	measured body vector angles
float**	txAng	estimated global vector angles
float***	txVecBody	measured body vectors
float***	txVecGlo	estimated global vectors

(int**) `txVecUse[transmitter][face]` contains TRUE/FALSE flags specifying the validity of sphere to beacon unit vector information.

(float***) `txVecBody[transmitter][face][dimension]` contains the measured unit vectors μ_{ij} from sphere faces to beacons, expressed in the body coordinate frame. These vectors are produced from the raw distance measurements of `pads.matrix`, and are used in attitude determination. Known bad vector measurements are identified by zeros at the corresponding locations in `pads.txVecUse`.

(float***) `txVecGlo[transmitter][face][dimension]` contains estimated unit vectors ν_{ij} from sphere faces to transmitters, expressed in the global coordinate frame. These vectors are based on a rough position estimate and the last known attitude estimate, and are used in attitude determination. Known bad vector estimates are identified by zeros at the corresponding locations in `pads.txVecUse`.

(float**) `rxAng[transmitter][face]` contains measured receiver angles in radians. Known bad angle measurements are identified by zeros at the corresponding locations in `pads.txVecUse`.

(float**) `txAng[transmitter][face]` contains the estimated transmitter angles, in units of radians. Known bad angle estimates are identified by zeros at the corresponding locations in `pads.txVecUse`.

(float**) `body2Glo[i][j]` is the 3×3 rotation matrix that transforms a vector from the body frame into the global frame.

(float**) `inertia[i][j]` is the 3×3 inertia matrix about the center of mass, not about the body frame origin.

(float**) `inertiaInv[i][j]` is the inverse of the inertia matrix `pads.inertia`.

(int) `globalPeriod` is the time delay between PADS global update requests, expressed in milliseconds. The value of `pads.globalPeriod` may be changed at

Table B.5: Contents of the global structure **sys**.

Type	Name	Description
uint	fTimeStarted	testbed clock is running
uint	fTimeRequested	an IR time update was requested
uint	fPadsMaster	array of TRUE/FALSE specifying PADS master
uint	fAlive	array of TRUE/FALSE specifying if alive
uint	ID	sphere ID number
int	cGlobalRequest	number of global requests
int	Wdog	watchdog
int	Wdog_on_off	watchdog bit
long	spheresTime	testbed clock
long	onboardTime	onboard clock
long	tstampNextIR	testbed time stamp of next IR
long	tankTime	cumulative time on current tank
long	battTime	cumulative time on current batteries

any time by the control code.

(int) localPeriod is the expected time delay between PADS local sensor readings. The local sensor readings are received in the PADS interrupt process, which is triggered externally when new measurements are made.

B.2.3 System data structure: sys

The global variables contained in the global system structure **sys** are listed in Table B.5. A subset of those variables are described in detail in the following discussion.

(uint) fPadsMaster[sphere ID] contains a set of TRUE/FALSE flags, one for each sphere. The flag is TRUE for the sphere ID number of the current PADS master and FALSE for other indices. Valid indices are SPHERE1, SPHERE2, and SPHERE3.

(uint) ID is the sphere ID number of the current sphere. Valid values are SPHERE1, SPHERE2, and SPHERE3.

(long) timeSpheres is the testbed clock, expressed in milliseconds. Periodic clock synchronization occurs with IR flashes.

Table B.6: Contents of the global structure `prop`.

Type	Name	Description
`int`	`thrusters`	array of thruster remaining on-times
`int`	`thrustersUsed`	array of on-times between propagation steps
`float`	`thrForce`	array of thruster forces
`float`	`avgForce`	average force of a single thruster
`float**`	`forces2thr`	force and torque to thruster force mapping matrix

`(long) onboardTime` is the local clock, expressed in milliseconds. `sys.onboardTime` counts from the previous DSP reset.

`(long) tstampNextIR` is the time at which the next IR flash will occur. The PADS master sends the value of `sys.tstampNextIR` to the other spheres prior to an IR flash. Upon reception of the IR flash, each sphere then sets the value of `spheresTime` equal to `tstampNextIR`.

`(long) tankTime` counts the number of milliseconds for which the thrusters have been open. This number is reset to zero upon tank replacement.

`(long) battTime` counts the number of milliseconds for which the batteries have been in use. This number is reset to zero upon battery replacement.

B.2.4 Propulsion data structure: `prop`

The global variables contained in the propulsion subsystem structure `prop` are listed in Table B.6. A subset of those variables are described in detail in the following discussion.

`(int) thrusters` is a 12-place array containing the commanded thruster on-times in milliseconds. The elements of `prop.thrusters` are set by the controller at the control frequency, and are decremented at 1 kHz in the propulsion interrupt.

`(int) thrustersUsed` is a 12-place array containing the cumulative on-time of each thruster since the last state propagation step. The elements increment at the propulsion interrupt frequency whenever the corresponding elements of

`prop.thrusters` are greater than zero. The elements of `prop.thrustersUsed` are set to zero during state propagation.

`(float) thrForce` is a 12-place array containing the best known estimate for the force of each thruster.

`(float**) forces2thr` is a mapping matrix which transforms the `ctrl.control` array of force and torque commands into thruster-pair force commands.

B.3 SCI module source code

B.3.1 SCI command modules

The command module sets the values in the current target (i.e. desired) state vector `ctrl.stateTarget` based on some desired trajectory algorithm, and calls the function `find_error(···)`, which results in the creation of the current state error vector `ctrl.stateError`. The source code for the following SCI command modules is included for reference and clarification purposes. A description of each module is given in Section 4.3.2.

commands/regulate.c

```
#ifndef INCLUDE_REGULATE
#define INCLUDE_REGULATE

void regulate(void)
{
    int i;

    // capture the initial position and attitude
    if (ctrl.maneuverTime == 0.0)
    {
        for (i=POS_X; i<=POS_Z; i++)
            ctrl.stateTarget[i]  = pads.state[i];

        for (i=VEL_X; i<=VEL_Z; i++)
            ctrl.stateTarget[i]  = 0.0;
```

```
        for (i=QUAT_1; i<=QUAT_4; i++)
            ctrl.stateTarget[i] = pads.state[i];

        for (i=RATE_X; i<=RATE_Z; i++)
            ctrl.stateTarget[i] = 0.0;
    }

    find_error(ctrl.stateError, pads.state, ctrl.stateTarget);
}
#endif
```

commands/regulate_specified.c

```
#ifndef INCLUDE_REGULATE_SPECIFIED
#define INCLUDE_REGULATE_SPECIFIED

void regulate_specified(
        float desiredPosX,      // desired x position
        float desiredPosY,      // desired y position
        float desiredPosZ,      // desired z position
        float desiredQuat1,     // desired q1
        float desiredQuat2,     // desired q2
        float desiredQuat3,     // desired q3
        float desiredQuat4)     // desired q4
{
    // set the target state
    ctrl.stateTarget[POS_X]  = desiredPosX;
    ctrl.stateTarget[POS_Y]  = desiredPosY;
    ctrl.stateTarget[POS_Z]  = desiredPosZ;

    ctrl.stateTarget[VEL_X]  = 0.0;
    ctrl.stateTarget[VEL_Y]  = 0.0;
    ctrl.stateTarget[VEL_Z]  = 0.0;

    ctrl.stateTarget[QUAT_1] = desiredQuat1;
    ctrl.stateTarget[QUAT_2] = desiredQuat2;
    ctrl.stateTarget[QUAT_3] = desiredQuat3;
    ctrl.stateTarget[QUAT_4] = desiredQuat4;

    ctrl.stateTarget[RATE_X] = 0.0;
    ctrl.stateTarget[RATE_Y] = 0.0;
    ctrl.stateTarget[RATE_Z] = 0.0;
```

```
    find_error(ctrl.stateError, pads.state, ctrl.stateTarget);
}
#endif
```

B.3.2 SCI controller modules

The control module applies a control law to the contents of the state error vector, and assigns force and torque commands to the six-place array `ctrl.control`. The source code for the following SCI command module is included for reference and clarification purposes. A description of this module is given in Section 4.3.3.

controllers/control_position_PD.c

```
#ifndef INCLUDE_CONTROL_POSITION_PD
#define INCLUDE_CONTROL_POSITION_PD

void control_position_PD(float gainPos, float gainVel)
{
    // controller force outputs are in global coordinate frame
    ctrl.control[FORCE_X] = -1.0 *
        (((float)ctrl.fEnablePosX)*gainPos*ctrl.stateError[POS_X] +
         ((float)ctrl.fEnableVelX)*gainVel*ctrl.stateError[VEL_X]);

    ctrl.control[FORCE_Y] = -1.0 *
        (((float)ctrl.fEnablePosY)*gainPos*ctrl.stateError[POS_Y] +
         ((float)ctrl.fEnableVelY)*gainVel*ctrl.stateError[VEL_Y]);

    ctrl.control[FORCE_Z] = -1.0 *
        (((float)ctrl.fEnablePosZ)*gainPos*ctrl.stateError[POS_Z] +
         ((float)ctrl.fEnableVelZ)*gainVel*ctrl.stateError[VEL_Z]);
}
#endif
```

B.3.3 SCI terminator modules

The terminator module compares current conditions with one or more criteria for maneuver termination, and ends the maneuver when those criteria are met. The

source code for the following SCI terminator modules is included for reference and clarification purposes. A description of each module is given in Section 4.3.5.

terminators/terminate_elapsed.c

```
#ifndef INCLUDE_TERMINATE_ELAPSED
#define INCLUDE_TERMINATE_ELAPSED

void terminate_elapsed(
        int   *fTerminate,  // termination flag
        float endTime)      // maneuver time at which to terminate
{
  if (ctrl.maneuverTime >= endTime)
    *fTerminate = TRUE;
}
#endif
```

terminators/terminate_clock.c

```
#ifndef INCLUDE_TERMINATE_CLOCK
#define INCLUDE_TERMINATE_CLOCK

void terminate_clock(
        int   *fTerminate,  // termination flag
        float endTime)      // test time at which to terminate
{
    if (ctrl.testTime >= endTime)
        *fTerminate = TRUE;
}
#endif
```

terminators/terminate_commanded.c

```
#ifndef INCLUDE_TERMINATE_COMMANDED
#define INCLUDE_TERMINATE_COMMANDED

void terminate_commanded(
        int *fTerminate)    // termination flag
{
    static int currentManeuver;

    // initialize static variables
    if (ctrl.maneuverTime == 0.0;)
```

```
        currentManeuver = FALSE;

    // respond to maneuver commands from another sphere
    if (ctrl.commandedManeuver)
    {
        if (ctrl.testTime >= ctrl.commandedTime)
        {
            *fTerminate = TRUE;

            // set commanded maneuver number
            if (ctrl.commandedManeuver != NEXT_MANEUVER)
                ctrl.nextManeuver = ctrl.commandedManeuver;

            // don't force until commanded again
            ctrl.commandedManeuver = FALSE;
            currentManeuver = TRUE;
        }
    }

    // occurs only if maneuver was unterminated
    else if (currentManeuver)
        *fTerminate = TRUE;
}
#endif
```

terminators/terminate_hold_vel.c

```
#ifndef INCLUDE_TERMINATE_HOLD_VEL
#define INCLUDE_TERMINATE_HOLD_VEL

void terminate_hold_vel(
        int   *fTerminate,  // termination flag
        float threshold,    // velocity threshold
        float holdTime)     // minimum hold time
{
    static float heldTime;
    float vel = 0.0;
    int   i;

    // initialize static variables
    if (ctrl.maneuverTime == 0.0)
        heldTime    = -1.0;
```

```
    // find magnitude of velocity error
    for (i=VEL_X; i<=VEL_Z; i++)
        vel += square(ctrl.stateError[i]);
    vel = sqrt(vel);

    // check to see if velocity error is under threshold
    if (vel <= threshold)
    {
        // get time stamp when entering threshold
        if (heldTime == -1.0)
            heldTime = ctrl.maneuverTime;

        // see if elapsed time in threshold is long enough
        if (ctrl.maneuverTime-heldTime >= holdTime)
            *fTerminate = TRUE;
    }
    else
    {
        heldTime = -1.0;
    }
}
#endif
```

terminators/terminate_hold_pos.c

```
#ifndef INCLUDE_TERMINATE_HOLD_POS
#define INCLUDE_TERMINATE_HOLD_POS

void terminate_hold_pos(
        int   *fTerminate,  // termination flag
        float threshold,    // position threshold
        float holdTime)     // minimum hold time
{
    static float heldTime;
    float pos = 0.0;
    int   i;

    // initialize static variables
    if (ctrl.maneuverTime == 0.0)
        heldTime    = -1.0;

    // find magnitude of velocity error
    for (i=POS_X; i<=POS_Z; i++)
```

```
        pos += square(ctrl.stateError[i]);
    pos = sqrt(pos);

    // check to see if position error is under threshold
    if (pos <= threshold)
    {
        // get time stamp when entering threshold
        if (heldTime == -1.0)
            heldTime = ctrl.maneuverTime;

        // see if elapsed time in threshold is long enough
        if (ctrl.maneuverTime-heldTime >= holdTime)
            *fTerminate = TRUE;
    }
    else
    {
        heldTime = -1.0;
    }
}
#endidf
```

terminators/terminate_nreps.c

```
#ifndef INCLUDE_TERMINATE_NREPS
#define INCLUDE_TERMINATE_NREPS

void terminate_nreps(
        int   *fTerminate,  // termination flag
        int   nreps)        // number of iterations
{
    static int reps;

    // initialize static variables
    if (ctrl.maneuverTime == 0.0)
        reps = 0;

    reps++;     // increment repetition counter
    if (reps >= nreps)
        *fTerminate = TRUE;
}
#endif
```

terminators/terminate_ready.c

```
#ifndef INCLUDE_TERMINATE_READY
#define INCLUDE_TERMINATE_READY

void terminate_ready(
        int *fTerminate)    // termination flag

    int i, alive=0, ready=0;

    for (i=SPHERE1; i<=SPHERE3; i++)

        alive += sys.fAlive[i];
        ready += ctrl.fReady[i];


    // terminate if all spheres are ready
    if (alive == ready)
        *fTerminate = TRUE;

#endif
```

B.3.4 SCI flow control modules

Maneuver flow control modules are used to modify the sequential flow of the maneuvers within a test. The source code for the following SCI flow control modules is included for reference and clarification purposes. A description of each module is given in Section 4.3.6.

flow_control/goto_maneuver.c

```
#ifndef INCLUDE_GOTO_MANEUVER
#define INCLUDE_GOTO_MANEUVER

void goto_maneuver(
        int *fTerminate,    // termination flag
        int maneuverNum)    // maneuver number to go to
{
    if (*fTerminate)
        ctrl.nextManeuver = maneuverNum;
}
#endif
```

flow_control/delay_termination.c

```
#ifndef INCLUDE_DELAY_TERMINATION
#define INCLUDE_DELAY_TERMINATION

void delay_termination(
        int   *fTerminate,  // termination flag
        float delayTime)    // delay from current time
{
    static initialTime;

    // initialize static variables
    if (ctrl.maneuverTime == 0.0)
        initialTime = -1.0;

    // when first terminated, save timestamp
    if (*fTerminate && (initialTime == -1.0))
        initialTime = ctrl.maneuverTime;

    // terminate only after delayTime has elapsed
    if (initialTime != -1.0)
    {
        if (ctrl.maneuverTime-initialTime >= delayTime)
            *fTerminate = TRUE;
        else
            *fTerminate = FALSE;
    }
}
#endif
```

flow_control/signal_ready.c

```
#ifndef INCLUDE_SIGNAL_READY
#define INCLUDE_SIGNAL_READY

void signal_ready(
        int   fSendMsg)     // sends message when nonzero
{
    static int fFirstTime;
    unsigned char data = DATA_READY;

    // initialize static variables
    if (ctrl.maneuverTime == 0.0)
        fFirstTime = TRUE;
```

```
    // send the ready message
    if (fSendMsg && fFirstTime)
    {
        send_command(SAT1+SAT2+SAT3-comm.ME,
                     sys.spheresTime, 1, &data);
        fFirstTime = FALSE;
    }
}
#endif
```

---

flow_control/send_terminate.c

```
#ifndef INCLUDE_SEND_TERMINATE
#define INCLUDE_SEND_TERMINATE

void send_terminate(
        int   fSendMsg,     // sends message when nonzero
        int   sphereID,     // commanded sphere ID number
        int   maneuverNum,  // maneuver number command
        float delayTime)    // delay for comm latency
{
    static int fSent[SPHERE3+1];
    int i;
    long endTime;
    unsigned char data[5];

    // initialize static variables
    if (ctrl.maneuverTime == 0.0)
        fSent[sphereID] = FALSE;

    if (fSendMsg && !fSent[sphereID])
    {
        // figure out the termination time
        endTime = sys.spheresTime+(long)(1000.0*delayTime);

        // fill in the data
        data[0] = DATA_TERMINATE_CMD;
        data[1] = manueverNum;
        data[2] = endTime & 0xFF;
        data[3] = (endTime>>8) & 0xFF;
        data[4] = (endTime>>16) & 0xFF;
```

```
        // send the data
        send_command(1<<sphereID, sys.spheresTime, 5, data);
        fSent[sphereID] = TRUE;
    }
}
#endif
```

flow_control/force_terminate.c

```
#ifndef INCLUDE_FORCE_TERMINATE
#define INCLUDE_FORCE_TERMINATE

void force_terminate(
        int   fSendMsg,     // sends command when nonzero
        int   sphereID,     // commanded sphere ID number
        int   maneuverNum,  // maneuver number command
        float delayTime)    // delay for comm latency
{
    static int fSent[SPHERE3+1];
    int i;
    long endTime;
    unsigned char data[5];

    // initialize static variables
    if (ctrl.maneuverTime == 0.0)
        fSent[sphereID] = FALSE;

    if (fSendMsg && !fSent[sphereID])
    {
        // figure out the termination time
        endTime = sys.spheresTime+(long)(1000.0*delayTime);

        // fill in the data
        data[0] = DATA_TERMINATE_FORCE;
        data[1] = manueverNum;
        data[2] = endTime & 0xFF;
        data[3] = (endTime>>8) & 0xFF;
        data[4] = (endTime>>16) & 0xFF;

        // send the data
        send_command(1<<sphereID, sys.spheresTime, 5, data);
        fSent[sphereID] = TRUE;
    }
```

```
}
#endif
```

flow_control/force_next.c

```
#ifndef INCLUDE_FORCE_NEXT
#define INCLUDE_FORCE_NEXT

void force_next(
        int   fSendMsg,     // sends message when nonzero
        int   sphereID,     // commanded sphere ID number
        int   maneuverNum)  // maneuver number command
{
    static int fSent[SPHERE3+1];
    int i;
    long endTime;
    unsigned char data[2];

    // initialize static variables
    if (ctrl.maneuverTime == 0.0)
        fSent[sphereID] = FALSE;

    if (fSendMsg && !fSent[sphereID])
    {
        // fill in the data
        data[0] = DATA_NEXT_MANEUVER;
        data[1] = manueverNum;

        // send the data
        send_command(1<<sphereID, sys.spheresTime, 2, data);
        fSent[sphereID] = TRUE;
    }
}
#endif
```

flow_control/wait_for_all.c

```
#ifndef INCLUDE_WAIT_FOR_ALL
#define INCLUDE_WAIT_FOR_ALL

void wait_for_all(
        int   *fTerminate,  // termination flag
```

```
    float delayTime)    // delay for comm latency
{
    static int fSent, fReady, currentManeuver;
    int i;

    // initialize static variables
    if (ctrl.maneuverTime == 0.0)
    {
        fSent           = FALSE;
        fReady          = FALSE;
        currentManeuver = FALSE;
        delay_termination(&fReady,0.0);
        for (i=SPHERE1; i<=SPHERE3; i++)
            send_terminate(FALSE, i, 0, 0.0);
    }

    // signal readiness and unterminate
    signal_ready(*fTerminate);
    *fTerminate = FALSE;

    // wait until all ready signals have been received
    terminate_ready(fTerminate);

    if (*fTerminate)
        fReady = TRUE;

    if (fReady)
    {
        if (sys.fPadsMaster[sys.ID])
        {
            for (i=SPHERE1; i<=SPHERE3; i++)
                send_terminate(TRUE, i, NEXT_MANEUVER, delayTime);
            delay_termination(fTerminate, delayTime);
        }
        else
        {
            // unterminate until a maneuver command is received
            *fTerminate = FALSE;
            if (ctrl.commandedManeuver)
            {
                if (ctrl.testTime >= ctrl.commandedTime)
                {
                    *fTerminate = TRUE;
```

```
                    // set commanded maneuver number
                    if (ctrl.commandedManeuver != NEXT_MANEUVER)
                        ctrl.nextManeuver = ctrl.commandedManeuver;

                    // don't force until commanded again
                    ctrl.commandedManeuver = FALSE;
                    currentManeuver = TRUE;
                }
            }

            // occurs only if maneuver was unterminated
            else if (currentManeuver)
                *fTerminate = TRUE;

        } // not PADS master
    } // if (fReady)
}
#endif
```

B.3.5 SCI multi-type modules

It is possible to combine multiple module types into a single function call, or to bypass one or more module types. The source code for the following SCI multi-type module is included for reference and clarification purposes. A description of this module is given in Section 4.3.7.

multi-type/thrusters_timed.c

```
#ifndef INCLUDE_THRUSTERS_TIMED
#define INCLUDE_THRUSTERS_TIMED

void thrusters_timed(
        long  thrusters,    // thruster numbers to fire
        float onTime)       // commanded on-time
{
    int i;

    for (i=0; i<12; i++)    // cycle through thrusters
    {
        if (thrusters & (1 << i))
```

```
        {
            if (ctrl.maneuverTime < (thrTime+THR_DELAY))
                prop.thrusters[i] = THR_DELAY
                    + (int) (1000.0*(thrTime - ctrl.maneuverTime))
        }
    }
}
#endif
```

B.3.6 Control housekeeping algorithm

The standard control interface is built on a foundation provided by a controller housekeeping algorithm. The housekeeping algorithm is described in Section 4.5, and is implemented in the function `c_int02()` in `control.c`.

<SPHERES>/tests/<testname>/control.c

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*  DIRECT CONTROL INTERFACE USERS:                              */
/*    Replace the contents of c_int02() with your own code.  */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

// these files are required for default station-keeping
#include "../../commands/regulate.c"
#include "../../controllers/control_attitude_NLPD.c"
#include "../../controllers/control_position_PD.c"
#include "../../mixers/mix_simple.c"
#include "../../terminators/terminate_hold_vel.c"
#include "../../terminators/terminate_elapsed.c"

/* * * * * * * * * * * * * * * * * */
/*  Control interrupt routine  */
/* * * * * * * * * * * * * * * * * */

void c_int02()
{
    unsigned char data[1];

    // exit if control is not enabled
    if (!ctrl.fDoControl)
    {
```

```
    dbug.blinkType = LED_BLINK_FAST;
    return;
}

NEST_INT();  // enable nested interrupts

// respond to forced maneuver commands from other spheres
if (ctrl.forcedManeuver &&
   (ctrl.forcedTime != WAIT_FOR_TERMINATION))
{
    if (sys.spheresTime >= ctrl.forcedTime)
    {
        // prepare for a new maneuver
        ctrl.fTestDone      = FALSE;
        ctrl.fTerminate     = FALSE;
        ctrl.fManeuverReset = TRUE;
        ctrl.maneuverTime   = 0.0;

        // set commanded maneuver number
        if (ctrl.forcedManeuver == NEXT_MANEUVER)
            ctrl.maneuverNum++;
        else
            ctrl.maneuverNum    = ctrl.forcedManeuver;

        // don't force until commanded again
        ctrl.forcedManeuver = FALSE;
    }
}

if (ctrl.fTestReset)            // if new test
{
    ctrl.fTestReset   = FALSE;  // clear new test flag
    ctrl.fTestDone    = FALSE;  // clear test done flag
    ctrl.fTerminate   = FALSE;  // clear termination flag
    ctrl.testTime     = 0.0;    // reset elapsed test time
    ctrl.maneuverTime = 0.0;    // reset elapsed maneuver time
    ctrl.maneuverNum  = 1;      // start with first maneuver
}

// null residual velocity and disable control
if (ctrl.fTestDone)
{
    // regulate about initial state
```

```
    regulate();
    control_attitude_NLPD(0.10, 0.04);
    control_position_PD(0.5, 0.2);
    mix_simple(20);
    terminate_hold_vel(&ctrl.fTerminate,  // termination flag
                       1.0,               // velocity threshold
                       2.0);              // threshold hold time
    terminate_elapsed(&ctrl.fTerminate,   // termination flag
                       10.0);             // timeout

    // track elapsed time
    ctrl.maneuverTime += 1.0/(float)controlFrequency;
    ctrl.testTime     += 1.0/(float)controlFrequency;

    if (ctrl.fTerminate)
    {
        ctrl.fDoControl   = FALSE;  // disable control
        ctrl.fTerminate   = FALSE;  // clear termination flag
        ctrl.fTestDone    = FALSE;  // clear test done flag
        ctrl.maneuverTime = 0.0;    // reset elapsed maneuver time
        ctrl.testTime     = 0.0;    // reset elapsed test time
        UN_NEST();
        return;
    }
}

// stationkeep if no start command has been received
else if (ctrl.testNum == 0)
{
    // regulate about initial state indefinitely
    regulate();
    control_attitude_NLPD(0.10, 0.04);
    control_position_PD(0.5, 0.2);
    mix_simple(20);
    UN_NEST();                      // disable nested interrupts
    return;
}

// process module sequences in the current maneuver
else
{
    dbug.blinkType = LED_BLINK_CONTROL;
```

```
    do_maneuver(ctrl.testNum,       // test number
                ctrl.maneuverNum,   // maneuver number
                ctrl.maneuverTime,  // elapsed maneuver time
                &ctrl.fTerminate,   // maneuver termination flag
                &ctrl.fTestDone);   // test finished flag
}

// if maneuver was terminated
if (ctrl.fTerminate)
{
    ctrl.fTerminate = FALSE;
    ctrl.maneuverTime   = 0.0;

    // check to see if specific maneuver was commanded
    if ((ctrl.forcedManeuver) &&
       (ctrl.forcedTime == WAIT_FOR_TERMINATION))
    {
        ctrl.maneuverNum    = ctrl.forcedManeuver;
        ctrl.forcedManeuver = FALSE;
        ctrl.nextManeuver   = FALSE;  // forced takes preference
    }
    else if (ctrl.nextManeuver)
    {
        ctrl.maneuverNum  = ctrl.nextManeuver;
        ctrl.nextManeuver = FALSE;
    }

    // otherwise increment maneuver number
    else
        ctrl.maneuverNum ++;
}
else                            // increment elapsed time
    ctrl.maneuverTime += 1.0/(float)ctrl.controlFrequency;

// if test has completed
if (ctrl.fTestDone)
{
    data[0] = DATA_TEST_DONE;   // notify laptop of status
    send_command(GROUND, sys.spheresTime, 1, data);

    ctrl.fTerminate   = FALSE;  // clear termination flag
    ctrl.testNum      = 0;      // set default test number
    ctrl.maneuverTime = 0.0;    // reset elapsed maneuver time
```

```
        ctrl.testTime     = 0.0;    // reset elapsed test time

        // change to known good control frequency
        ctrl.controlFrequency = TEST_DONE_CTRL_FREQ;
    }
    else
        ctrl.testTime +=1.0/(float)ctrl.controlFrequency;

    UN_NEST();                      // disable nested interrupts
    return;
}


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*  find_error(...) is called by the command module  */
/*  to calculate the state error vector              */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * */

void find_error(float *error, float *state, float *target)
{
    int i;

    // elements of the various quaternions
    float q1,q2,q3,q4, q1d,q2d,q3d,q4d;

    for (i=POS_X; i<=VEL_Z; i++)
        error[i] = target[i] - state[i];

    for (i=RATE_X; i<=WDOT_Z; i++)
        error[i] = target[i] - state[i];

    // desired quaternion
    q1d = target[QUAT_1];
    q2d = target[QUAT_2];
    q3d = target[QUAT_3];
    q4d = target[QUAT_4];

    // current quaternion
    q1  = state[QUAT_1];
    q2  = state[QUAT_2];
    q3  = state[QUAT_3];
    q4  = state[QUAT_4];
```

```
    // error quaternion
    error[QUAT_1] = -q1*q4d - q2*q3d + q3*q2d + q4*q1d;
    error[QUAT_2] =  q1*q3d - q2*q4d - q3*q1d + q4*q2d;
    error[QUAT_3] = -q1*q2d + q2*q1d - q3*q4d + q4*q3d;
    error[QUAT_4] =  q1*q1d + q2*q2d + q3*q3d + q4*q4d;

    quat_normalize(&error[QUAT_1]);
}
```

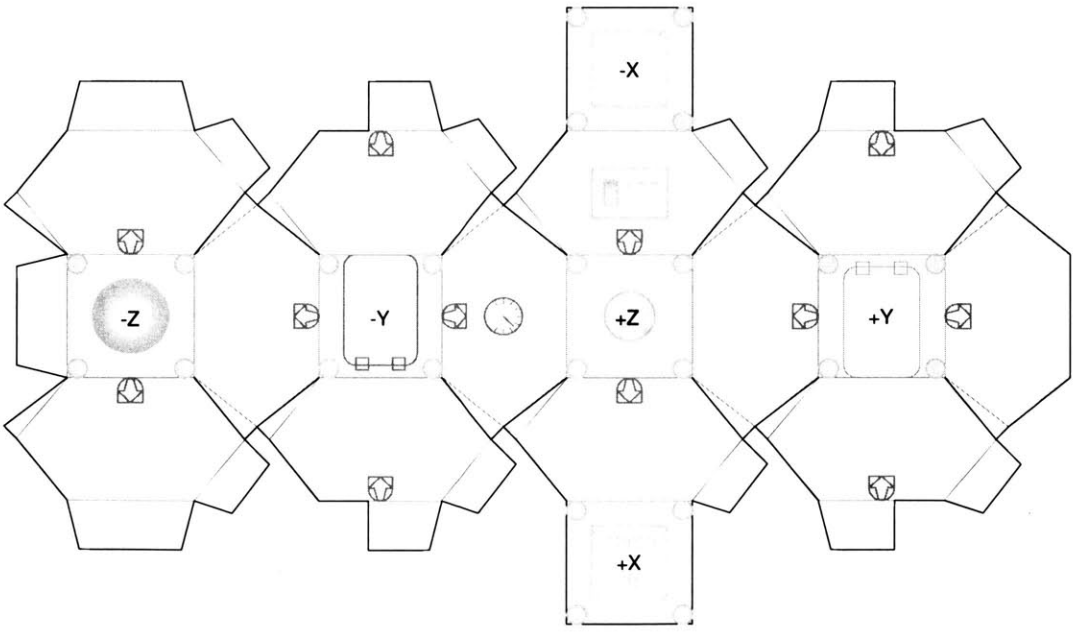
Appendix π

Sphere Paper Cutout Model

This appendix contains a flight sphere paper cutout model. To make your own paper sphere model, follow these directions. Allow yourself at least an hour, as it's harder than it looks to make it fit together cleanly.

1. Photocopy this page
2. Cut along all dark lines, and fold along straight light lines (the lines connecting sides and tabs).
3. Attach tabs to the insides of the sides with glue (this is the hard part).

Congratulations! You now have yourself a sphere. Now make two more and you can perform your own formation flying experiments.



Bibliography

[1] Simon L. Altmann. *Rotations, Quaternions, and Double Groups.* Oxford Science Publications. Cambridge University Press, 1986.

[2] C. A. Beichman, N. J. Woolf, and C. A. Lindensmith, editors. *The Terrestrial Planet Finder (TPF): A NASA Origins Program to Search for Habitable Planets.* JPL Publication 99-3, 1999.

[3] BEI Systron Donner Inertial Division. BEI GYROCHIP II micromachined angular rate sensor. http://www.systron.com/pdfs/GyroIIDS.pdf, 21 March 2002.

[4] Allen Chen, Alvar Saenz-Otero, Mark Hilstad, and David W. Miller. Development of formation flight and docking algorithms using the SPHERES testbed. *AIAA/Utah State University Conference on Small Satellites*, August 2001. SSC01VIIIA-2.

[5] Allen Chen. Propulsion system characterization for the SPHERES formation flight and docking testbed. Master's thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, June 2002.

[6] John L. Crassidis, Stephen F. Andrews, F. Landis Markley, and Kong Ha. Contingency designs for attitude determination of TRMM. NASA Goddard Space Flight Center.

[7] D. B. DeBra. Pulse modulators. Unpublished lecture notes for AA 277, Stanford University.

[8] John Enright. *A Flight Software Development and Simulation Framework for Advanced Space Systems.* PhD dissertation, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, June 2002.

[9] Lawrence Fallon, III. *Spacecraft Attitude Determination and Control*, volume 73 of *Astrophysics and Space Science Library*, appendix D, pages 758–759. Kluwer Academic Publishers, Dordrecht, Holland, 1978.

[10] Philip Ferguson and Jonathan P. How. Formation flying experiments on the ORION-Emerald mission. http://www.mit.edu/people/jhow/orion/space2001_latest2.pdf, 2001.

[11] D. Folta, L. Newman, and T. Gardner. Foundations of formation flying for mission to planet Earth and new millenium. *AIAA-96-3645-CP*, 1996. NASA Goddard Space Flight Center.

[12] Arthur Gelb, editor. *Applied Optimal Estimation.* MIT Press, 1974. The Technical Staff, The Analytic Sciences Corporation.

[13] Honeywell International Inc., Inertial Sensor Products Redmond. Q-Flex QA-160/185 Accelerometer. http://www.inertialsensor.com/docs/qa-t160.pdf, 21 March 2002.

[14] Peter C. Hughes. *Spacecraft Attitude Dynamics.* John Wiley & Sons, Inc., New York, 1986.

[15] F. Landis Markley and Daniele Mortari. How to estimate attitude from vector observations. In *Astrodynamics 1999*, volume 103, pages 1979–1996, 1999. AAS 99-427.

[16] F. Landis Markley and Daniele Mortari. New developments in quaternion estimation from vector observations. In *The Richard H. Battin Astrodynamics Symposium*, volume 106, pages 373–393, 2000. AAS 00-266.

[17] Microsoft Corporation. MSDN Library Visual Studio 6.0. Sample projects: namepipe/npserver/server32.dsp, namepipe/npclient/client32.dsp, 2000.

[18] David W. Miller et. al. SPHERES critical design review, February 2002.

[19] Andrew D. B. Radcliffe. A real-time simulator for the SPHERES formation flying satellites testbed. Master's thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, June 2002.

[20] Alvar Saenz-Otero, Allen Chen, David W. Miller, and Mark Hilstad. SPHERES: Development of an ISS laboratory for formation flight and docking research. *IEEE Aerospace Conference*, 2002. 0-7803-7231-X/01.

[21] Alvar Saenz-Otero. The SPHERES satellite formation flight testbed: Design and initial control. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, August 2000.

[22] Malcolm D. Shuster. Maximum likelihood estimation of spacecraft attitude. *Journal of the Astronautical Sciences*, 37(1):79–88, January-March 1965.

[23] Stanford University. Emerald mission requirements document, revision 5.0. http://ssdl.stanford.edu/Emerald/mission/pdf/Mission_Reqs.PDF, Dec 1999.

[24] University of Washington. ION-F/UW Dawgstar critical design review: Session I. http://www.aa.washington.edu/research/dawgstar/docs/ppt/session1.ppt, July 2000.

[25] Grace Wahba. A least squares estimate of satellite attitude. *SIAM Review*, 7(3):409, 1965. Problems and Solutions, Problem 65-1.

[26] James R. Wertz. *Spacecraft Attitude Determination and Control*, volume 73 of *Astrophysics and Space Science Library*. Kluwer Academic Publishers, Dordrecht, Holland, 1978.