# System Integration and Control of a Low-Cost Spacecraft Attitude Dynamics Simulator

A Thesis

Presented to the Faculty of
California Polytechnic State University, San Luis Obispo

In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Aerospace Engineering

by

Ryan L. Kinnett

March 9, 2010

## Thesis Committee Membership

| | |
|---|---|
| Title: | *System Integration and Attitude Control of a Low-Cost Spacecraft Attitude Dynamics Simulator* |
| Author: | Ryan Kinnett |
| Date Submitted: | March 9, 2010 |

# Abstract

System Identification and Attitude Control of a Low-Cost
Spacecraft Attitude Dynamics Simulator

Ryan Lloyd Kinnett

The CalPoly Spacecraft Attitude Dynamics Simulator mimics the rotational dynamics of a spacecraft in orbit and acts as a testbed for spacecraft attitude control system development and demonstration. Prior to this thesis, the simulator platform and several subsystems had been designed and manufactured, but the total simulator system was not yet capable of closed-loop attitude control. Previous attempts to make the system controllable were primarily mired by data transport performance. Rather than exporting data to an external command computer, the strategy implemented in this thesis relies on a compact computer onboard the simulator platform to handle both attitude control processing and data acquisition responsibilities. Software drivers were created to interface the computer's data acquisition boards with Matlab, and a Simulink library was developed to handle hardware interface functions and simplify the composition of attitude control schemes. To improve the usability of the system, a variety of actuator control, hardware testing, and data visualization utilities were also created. A closed-loop attitude control strategy was adapted to facilitate future sensor installations, and was tested in numerical simulation. The control model was then updated to interface with the simulator hardware, and for the first time in the project history, attitude control was performed onboard the CalPoly spacecraft attitude dynamics simulator. The demonstration served to validate the numerical model and to verify the functionality of the entire simulator system.

# Acknowledgements

I would like to thank:

The CalPoly Aerospace Faculty
*for bestowing upon me the skills and knowledge to succeed*

The Staff of the Naval Postgraduate School's
Unmanned Systems Lab
*for sharing the magic of robotics*

and

My Parents
*for their unwavering patience, guidance, and support,*
*and for teaching me everything else which truly matters.*

## Table of Contents

## List of Tables

## List of Figures

# 1. Introduction

Due to the extraordinary costs of spacecraft design, development, and launch, engineers must intimately understand how a spacecraft will behave on orbit. However, the differences between the environment in which a spacecraft is developed and that in which it is deployed pose challenges for spacecraft designers. For instance, ground testing of a spacecraft's control system can be problematic since the external torques acting on a satellite in its operational environment are minute relative to those imposed by ground support equipment during development. Furthermore, on-orbit testing of experimental control schemes is often not cost effective, and numerical control system modeling may not suffice for high risk and high performance space systems. Spacecraft attitude dynamics simulators provide the means for spacecraft control engineers to test controls systems prior to implementation onboard a spacecraft.

## 1.1. Spacecraft Dynamics Simulation Legacy

An early example of a three-axis spacecraft simulator (Figure 1) was built at NASA's Marshall Space Flight Center in 1959[1]. Since then, spacecraft dynamics simulators have been commissioned by most of the major spacecraft research centers and companies, and by many technical universities. The growing set of U.S. universities which own and operate spacecraft dynamics simulators includes: Stanford University, the University of Michigan, Utah State University, the Georgia Institute of Technology, the Massachusetts Institute of Technology, the Virginia Polytechnic Institute and State University, the University of California at Los Angeles in conjunction with the California Institute of Technology, and the Naval Postgraduate School, among others[1].

Numerous types of spacecraft simulators have been specifically designed to simulate translational motion, rotational motion, or a combination of both. Planar air bearings provide horizontal translation and rotation about the local vertical axis, much like an air hockey table or a hovercraft, to facilitate experimentation in both station-keeping and formation flying. In a similar manner, spherical air bearings make rotation possible around all three axes, to a limited extent. The Marshall Space Flight Center's Flight Robotics Laboratory (FRL) features a planar air bearing system which floats a self-contained pneumatic lift, which in turn supports a spherical air bearing, thus allowing 6-DOF motion[1]. Although other 6-DOF and some 5-DOF simulators exist, the majority of spacecraft dynamics simulator systems restrict motion to just 3-DOF by using either a planar or a spherical air bearing, exclusively[1]. Examples of various spacecraft dynamics simulator types are depicted in Figure 1 below.

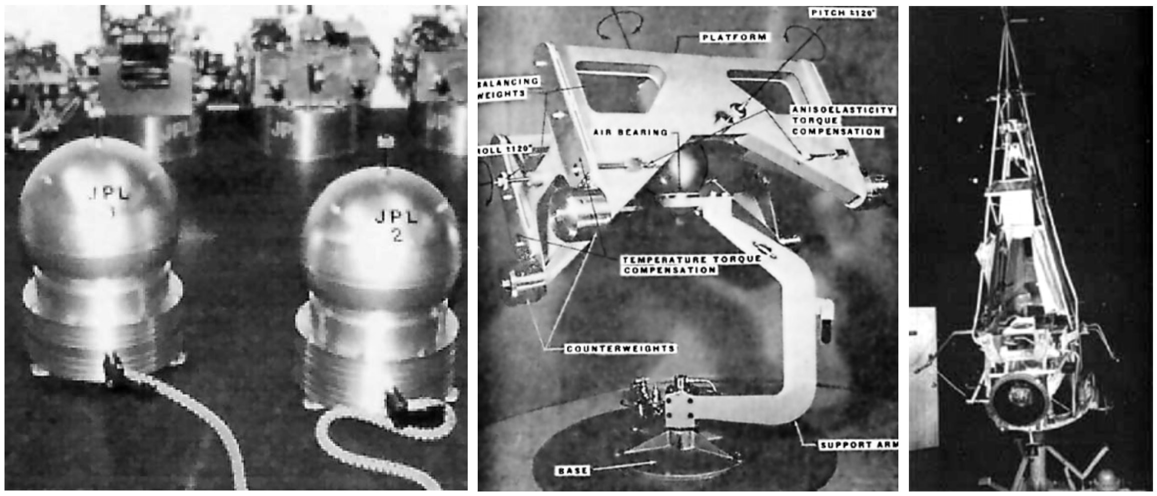


**Figure 1: Three Representative Spacecraft Dynamics Simulators[1].**
**Left:** The Jet Propulstion Laboratory's SPHERES simulators (2 translational DOF)
**Center:** Marshall Space Flight Center's Satellite Motion Simulator (3 rotational DOF)
**Right:** Langley Research Center's ALFA astronaut trainer (2 translational, 3 rotational DOF)

Attitude dynamics simulators are typically actuated by momentum exchange devices such as control moment gyros (CMGs), reaction wheels, or combinations of both. Some simulators additionally employ cold gas thrusters, although linear thrust devices are more commonly found on translational simulators. A variety of sensors has been used for attitude determination, including inertial rate sensors like microelectromechanical (MEMS) or ring-laser gyroscopes, magnetometers, accelerometers, horizon sensors, Global Positioning System (GPS) receivers, and optical alignment systems[1]. To the author's knowledge, no spacecraft simulator has yet incorporated an operational star tracker system for attitude dead-reckoning.

## 1.2. The CalPoly Spacecraft Attitude Dynamics Simulator Project

The project was first conceptualized in 2005, and since then has provided invaluable senior project and thesis opportunities to seventeen CalPoly graduate and undergraduate students. In accordance with CalPoly's motto, "learn by doing", the spacecraft simulator project is intended to provide hands-on learning opportunities to engineering students. The simulator may serve as a real-world technology demonstrator in controls classes or labs, and also as a testbed for innovative attitude determination and control research. CalPoly's PolySat Project has expressed interest in eventually employing the CalPoly spacecraft simulator for independent verification of future attitude determination systems for CubeSat nanosatellites.

The project is funded primarily by Northrop Grumman through CalPoly's Project-Based Learning Institute (PBLI). Also, Northrop Grumman loaned an LN-200 inertial measurement unit (IMU) to the project.

# 2. CalPoly Spacecraft Simulator Development History

The original simulator concept required that the platform, with all control equipment installed, would be small and light enough to be lifted by a single person[2]. The concept also called for a target angular acceleration of 0.1 rad/s$^2$ [2]. Due to mechanical simplicity, the preliminary design specified that a system of reaction wheels, rather than one or more control moment gyros, would be developed to provide control authority.

## 2.1. Simulator Design

In 2006, graduate student Carson Mittelsteadt refined the design requirements by specifying the platform's maximum dimensions and weight limits, rotation angle limits, and reaction wheel support geometry. Each aspect of the platform's design is discussed in detail in the following subsections.

### *2.1.1. Primary Structure*

The CalPoly spacecraft simulator platform consists of a rigid aluminum structure with two equipment decks, a spherical air bearing, supports for four reaction wheels in a pyramidal formation, and configurable counterweights in all three axes. Four channels for sliding counterweights comprise the 16 in. by 16 in. square base of the structure, while two vertical counterweight channels and the reaction wheel supports connect the base to the equipment decks. The structure is not expected to flex significantly in response to anticipated command torques.

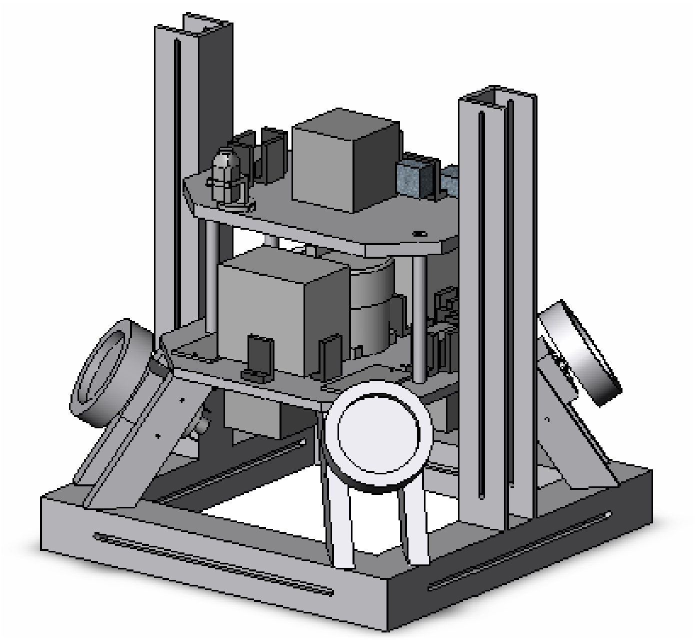


**Figure 2:** CalPoly Spacecraft Attitude Dynamics Simulator, early configuration

### 2.1.2. Air Bearing

A 4 in. diameter spherical air bearing restricts translational motion of the platform while imparting negligible torque. The socket of the bearing is mounted atop a rigid stand, and the ball is bolted to the bottom of the platform' lower equipment deck. The bearing allows free rotation about the local vertical axis, but physically restricts pitch and roll rotations to ±30°.

### 2.1.3. Mass Balance System

To allow the user freedom in arranging onboard components, the platform structure includes six channels in which sliding counterweights can be manually positioned. The counterweights are machined blocks of 1022 Steel, weighing 1.70 lbs each, and are secured in the counterweight channels by a nut and bolt through each[3]. Prior to each mission, the user adjusts the position of each counterweight, one at a time, until the platform's uncontrolled attitude is very close to neutrally stable.

To allow more precise control of the platform's center of gravity, a system of motor-driven translatable masses was designed and fabricated by former project member Christopher Saile. This fine balance system (FBS) is capable of shifting the platform center of mass by as much as 0.03 in., with precision of better than 0.0003 inches [3]. The FBS controllers are daisy chained in such a way that the entire 3-actuator system can be commanded over a single RS-232 serial link.

### *2.1.4. Reaction Wheel System*

Early project members selected a system of reaction wheels over other momentum exchange options for mechanical simplicity and because they do not exhibit the "gimbal lock" problems associated with control moment gyros. Furthermore, using fixed control torque orientations simplifies the equations of motion and control system design.

After estimating the platform's inertia tensor via solid modeling, Mittelsteadt approximated the net torque required to achieve the target platform angular acceleration of 0.1 rad/s$^2$. Mittelsteadt then searched for a commercially-available motor which, when coupled with an appropriately-sized reaction wheel, would be capable of generating the necessary torque. The selected Faulhaber 3863-24C DC Micromotors exhibit a stall torque of 1.250 N-m and a maximum turn rate of 6700 revolutions per minute[1]. Four aluminum reaction wheels, with inertias of 1.83 lb-in$^2$, were designed to match the selected motors[1]. Finally, Mittelsteadt modeled the reaction wheel system in Simulink to verify that the Faulhaber motors and aluminum wheels could generate the desired platform acceleration.

The Faulhaber Micromotors are controlled by high-current 24 V pulse-width modulated signals. Four Tecel D200 motor driver carrier boards generate this signal by mirroring a low-current TTL-level pulse-width modulated signal. The boards allow two acceleration modes, a coasting mode, and a braking mode, which are toggled by setting logic levels on two digital input pins. The braking mode uses back-EMF generated by a motor to further oppose the motor's spin, and must be used with caution to avoid overheating the driver carrier boards. Optical encoders on the reaction wheel drive motor shafts facilitate sensing of the reaction wheel speeds.

Jeff Logan attempted to develop tachometer circuits to measure the reaction wheel spin rates by reading the optical encoder signals. However, Logan's circuits were only partially functional[4], and his descriptions of the design failures included in his thesis draft were not cohesive enough for the system to be salvaged. Furthermore, his decentralized 4-board design was bulky and inconveniently placed on the volume-constrained platform.

### 2.1.5. Reaction Wheel Arrangement

The pyramid arrangement of reaction wheels provides control authority in all three axes. This arrangement consists of four reaction wheels located in the $x$-$z$ and $y$-$z$ planes, each tilted 29.3° above the $\pm x$ and $\pm y$ axes[2]. The tilt angle was designed by Mittelsteadt to balance control authority between the z-axis and the x and y-axes. By driving various combinations of wheel accelerations, torque can be applied in any direction.

This pyramid arrangement is inherently redundant. Since the spin axes of the four wheels are neither mutually orthogonal nor a minimal spanning set of three-space, any combination of just three of these axes still spans all of three-space.  Therefore, if any single reaction wheel fails, the remaining three wheels can still generate a net torque in any direction.  Although such redundancy is not necessary for a ground-based simulator, the design would be valuable for systems which require high levels of reliability, such as actual spacecraft.

#### *2.1.6. Navigation Sensors*

The primary strap-down inertial sensor will be the LN-200 inertial measurement unit. The IMU includes three laser-ring gyroscopes and three accelerometers, exhibits a very low drift rate of 0.1 °/hr, and transmits measurement data at a rate of 400 samples per second over an RS-485 serial link[5].  The IMU requires an unusual combination of three supply voltages, which must be applied in a specific order.

Phil Iversen, in 2005, demonstrated operation of the LN-200 IMU on the table top, but did not install the IMU on the simulator platform.  Iversen used a desktop power supply to power the IMU, and employed a Fastcomm PCI synchronous serial card in a desktop PC to read the IMU's data stream[6].  Later, in a first attempt to install the IMU on the platform, a team of undergraduate students explored two potential methods for transmitting the IMU data wirelessly.  The team concluded that the Gumstix computer could not handle the IMU's high data rate, and that a dedicated wireless serial transceiver could not be configured to read and forward the serial stream[7].  Although the team ultimately did not succeed in installing the IMU, they did provide valuable insight into what would be required.

Three single-axis micro-electromechanical (MEMS) gyros, model CRS03-01S by Silicon Sensing Systems, will also be used to track the platform's attitude. The MEMS gyros advertise a high drift rate of 0.55 °/s, and a maximum range of 100 °/s [2]. Due to the relatively low quality of the MEMS gyros, they either will be used to augment the IMU data or will be removed entirely once the IMU has been installed.

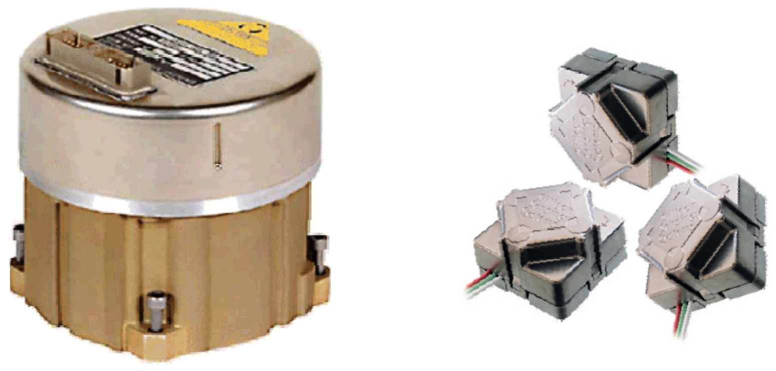


**Figure 3:** LN-200 IMU (left) and 3 Silicon Sensing MEMS gyros (right)

The platform's only dead-reckoning sensor will be a star tracker system. An artificial star field will be constructed and will be positioned over the platform with its center collocated with the platform's center of rotation. A webcam mounted on the platform will image the star dome, and the system will determine the platform's attitude by comparing observed star combinations to those in a catalog. An early version of the star tracker processing routine was developed by Taylore McClurg in 2008. McClurg's Matlab code was capable of determining the platform's azimuth during pure z-axis rotation, but could not reliably determine pitch or roll rotations[8]. The system was not developed to a state in which it could be installed and used effectively on the platform.

### 2.1.7. Data Acquisition & Processing

To avoid the torques an umbilical cable system would cause, it was determined that sensor data would need to be either processed onboard or transmitted wirelessly to an offboard computer. The first iteration of the simulator included a microcontroller to package sensor data, and a wireless transceiver integrated circuit to ship the data[2]. Mittelsteadt found that configuring the microcontroller and implementing the transmission protocol made this system too difficult to be an effective long-term solution for the simulator[2]. Upon Mittelsteadt's recommendation, later project contributors implemented a Gumstix embedded computer to sample, package, and ship sensor data via a Bluetooth link, but found this Gumstix-based acquisition and transmission system to be inadequate for effective platform attitude control[7].

#### *2.1.8. Onboard Power*

In order to avoid torques from the weight and tension of external power cables, the simulator requires a fully independent onboard power system. Also, the reaction wheel motors draw wildly varying amounts of current, which in turn causes the voltage of a battery bus to vary significantly. Even after regulation, a battery stack supporting the four reaction wheels would be too noisy for sensors like the MEMS gyros, which depend on a clean supply voltage to minimize measurement noise. Mittelsteadt's original power system design consisted of two separate battery buses: a 24 V bus to power the reaction wheel drive motors, and a separate 9 V bus to supply all other onboard equipment[2].

Both the amount and the fluctuation of current drawn by the reaction wheel drive motors necessitate lead-acid batteries. A pair of Power Sonic UB1250 12 V batteries, wired in series, is capable of supplying 50 A continuously at 24 V, with a total capacity of 120 W-hrs. This has proven to be sufficient current and capacity for more than two hours of operation with all four motors operating nominally[2].

The equipment power bus design consisted of two nickel metal-hydride rechargeable 9 V batteries, and was intended to power a wireless camera, four reaction wheel motor controllers, three MEMS gyroscopes, and an onboard microprocessor[2]. Mittelsteadt reported that the original equipment power bus was only capable of powering the equipment for about ten minutes, and recommended implementing a higher capacity battery stack for this bus[2]. Later project members expanded the equipment bus, but did not foresee the power requirements of the LN-200 IMU or an onboard computer[7].

### 2.1.9. System Identification

In 2006, Patrick Healey developed a method of system identification, which characterizes the platform's inertia by observing the platform's response to specific inputs. Healey demonstrated the method in simulation.

Later, in 2008, Seth Silva was the first to implement system identification with simulator hardware. Silva employed an onboard Gumstix computer to command the reaction wheels and transmit MEMS gyro data to an offboard computer. Unfortunately, the processing power of the Gumstix computer, the data rate of the Robostix-Gumstix interface, and the bandwidth of the Bluetooth link limited data transfer to roughly one measurement per second. Such poor temporal resolution was determined to be insufficient for effective system identification. Silva's method was further limited by the lack of true platform attitude control, since excitation of the platform's nutation modes requires driving the platform through complex trajectories. Without exciting these modes, the off-diagonal terms cannot be resolved accurately. Furthermore, Silva used the reaction wheel commanded rates as input to the system identification, rather than measuring and using the actual rates. At the time, there was no guarantee that the reaction wheels tracked anywhere near the commanded rates. Since the actual physical inputs to the system were not truly known, Silva's results were essentially invalid.

## 2.3. Summary of Status Prior to Current Development

Lyapunov stability of the pyramidal reaction wheel system has been proven analytically and verified via numerical simulation[2]. The platform has been fabricated and assembled, and the air bearing and course balance systems have been installed. A set of three fine balance mass movers has also been assembled and tested, mounted on the platform, and interfaced with the existing data acquisition system.

Four reaction wheels were constructed and installed onto four motors, which have been mounted on the platform in a pyramidal arrangement. Pololu driver carrier boards allow command of the motors' steady state turn rates, but no reliable method has been implemented for sensing the spin rates. Prior to current efforts, the transient responses of the motors could not be precisely controlled.

Two separate power buses have been installed on the platform. The reaction wheel power bus is matched well to the reaction wheel motors, while the electronics bus was deemed insufficient to power the platform for any useful period of time.

An embedded Gumstix data acquisition system was installed, but was only capable of passing data to and from the platform roughly once per second. This rate was found to be far too slow for effective closed-loop attitude control. The Gumstix computer does not have enough processing power to perform attitude control on its own.

Operation of the LN-200 IMU has been demonstrated on the desktop, but the platform did not have any facility for powering the IMU or reading its data stream. Three MEMS gyros were installed on the platform and read into the Gumstix computer.

Very basic system identification of the platform has been demonstrated, with limited success, using the Gumstix data acquisition system and MEMS gyros. However, the platform inertia tensor estimation generated by the system identification system was considered unrealistic because of data transfer rate limitations, the limited precision of the MEMS gyro measurements, the precision of the reaction wheel controllers, and the limited combinations of reaction wheel commands.

Prior to current efforts, closed-loop attitude control was not possible.

# 3. Current Objectives

The overall goal of current efforts is to develop the platform to a state in which future students or instructors can easily create and implement attitude control schemes. Fulfillment of this goal requires not only incorporating prior contributions and new improvements to finally implement full closed-loop attitude control, but also developing user interfaces and improving robustness of both platform hardware and software.

## 3.1. Concurrent Development

In addition to the author, two other graduate students were involved in the project while the work described in this thesis was conducted. In order to bring the platform to a state of controllability sooner, Matt Downs made several key contributions to the development of the simulator. Downs built and tested the reaction wheel tachometer and command circuits, developed closed-loop PID controllers for the wheels in Simulink, and converted Silva's system identification code into function form for post-processing of logged mission data. Downs completed his demonstrations of two adaptive attitude control techniques in August 2009.

Alan Wehrman worked to further develop the star tracker code started by McClurg. As of this writing, Wehrman's system was not ready to install on the simulator platform. Regardless, the navigation methods described in this thesis were designed to use star tracker input once the system becomes available.

## 3.2. System Development Goals

First, the platform needed a reliable power system with enough capacity to operate the platform for one hour. The reaction wheel battery bus has already been installed and shown to have sufficient capacity, but the equipment bus will need to be expanded to support the LN-200 IMU and other new hardware installations. To improve robustness and usability, several safety and user interface improvements will be made to both power buses. These improvements will include installing fuses, adding battery voltage sensing capabilities, consolidating external cabling, and developing a consolidated switch panel.

As discussed in previous sections, insufficient onboard computing has severely hampered the efforts of former project members to bring the platform to a state of controllability. A data acquisition system with more processing power than the Gumstix-based system must be designed and installed. The new system should be capable of either forwarding all sensor data to an external computer or performing all control functions onboard, or, ideally, both.

Interfacing the onboard data acquisition system with platform sensors and hardware controllers will require development of several hardware and software interfaces. These will include dedicated sensor interface circuits, data acquisition devices and software drivers for the computer, a robust wireless network link with some data transport technique, and methods for importing data into Simulink.

In partial fulfillment of the usability improvement goal, a strategy will be developed to simplify the process of developing and executing control techniques. Due to the external torque restriction, this strategy must be based on some type of wireless data transmission method. An offboard observer application will also be developed, and will utilize the wireless link to transfer and display telemetry data to the user.

A suite of Matlab tools will be developed to compartmentalize all hardware interface tasks and to facilitate easy creation of attitude control schemes. The suite will include data acquisition and hardware command functions, and a Simulink toolbox to handle all platform device interfaces. The Simulink toolbox will also include several control system components pre-configured for the simulator platform.

## 3.3. Attitude Control Implementation

The simulator system will feature a full-state feedback attitude controller for experimentation and demonstration purposes and to serve as a reference for future controller development. The composition of any Simulink-based control system will necessitate several key components, including data import modules, an attitude determination technique, and actuator command interfaces. Control schemes will also require a strategy for incorporating the lagged star tracker data with the real-time inertial sensors and tracking system.

The combined control system with the latest hardware will be modeled numerically and tested in simulation. Each of the simulator's subsystems will then be checked out individually to confirm the platform hardware is ready for control. Finally, the closed-loop attitude control scheme will be demonstrated in real time onboard the attitude dynamics simulator.

# 4. System Integration

Solutions to each of the hardware development goals outlined in Section 3.2 will be addressed in the following sections.

## 4.1. Power System Overhaul

As of 2008, the CalPoly spacecraft simulator platform power supply consisted of a pair of lead acid batteries which form the reaction wheel bus, and a lithium-ion battery to power the Gumstix computer and MEMS gyros. The reaction wheel bus was found to have adequate storage capacity and would only need minor changes. The equipment bus, however, needed to be expanded in order to support the LN-200 IMU and the more powerful onboard computer[7]. The original power system design also required the user to disconnect each battery bus from the platform in order to connect the battery chargers.

The new equipment bus design consists of two nickel-metal hydride batteries configured in series. The batteries were selected after considering the power consumption of all onboard components and the desired maximum mission duration. However, hardware additions have increased the total power consumption, and the capacity of the stack has deteriorated, resulting in support times of less than one hour.

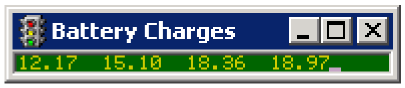
The battery voltages were selected, in part, based on the requirements of the IMU. Since the IMU requires three separate supply voltages at +15, +5, and -15 volts, an early version of the equipment bus design consisted of two 16 volt batteries wired in series. The IMU would have used the junction between the batteries as power ground, and drawn regulated +15 and +5 volts from the top of the stack, and regulated -15 volts from the bottom of the stack. Other devices would have used the bottom of the stack as power ground, and would have regulated their necessary supply voltages from the 32 volts at the top of the stack.

Unfortunately, after much time was invested in developing the IMU switching and regulation circuits, two considerable issues arose regarding the initial equipment bus design. First, this configuration forces a +16 volt difference between the IMU signal ground and the computer power ground. Since the documentation for the synchronous serial board (discussed in Section 4.2) does not address whether the board can handle a large difference between power and signal grounds, it is unknown if this original configuration could damage the computer. The manufacturer did not reply to requests for information regarding this issue. Furthermore, this design would load the batteries unevenly, since the +15 and +5 volt lines draw from just one of the two batteries, and these lines draw much more power than the -15 volt line. An improved design will provide the IMU with its required +15 and +5 volts from the top of the +32 volt equipment bus, while a separate battery pack will supply the -15 volts.

Several features were incorporated into the power system to improve system robustness and usability. First, the charger cables for all batteries were consolidated into a single umbilical with a Molex power connector, and all of the chargers were plugged in to a single AC power strip. With the new design, the user simply plugs in the charger umbilical and turns on the AC power strip to begin charging all of the system batteries. The new design eliminates the need to disconnect any equipment from the batteries, and even allows platform equipment to continue operating while the batteries charge. Also, the bus power switches were installed on a front switch panel for easy access, and several fast-acting fuses were integrated into the design to protect the platform equipment.

Finally, a battery level monitoring system was implemented. As will be discussed in Section 4.2, four analog input pins on the data acquisition board of the newly-installed onboard computer were dedicated to reading the voltage levels of the four batteries. Although the batteries at the bottom of each stack can be measured directly, the voltages of the top batteries must be determined by subtracting the lower battery voltages from the stack voltages. Rather than performing this subtraction in software after reading the necessary voltages, a subtraction circuit was designed. This circuit also scales the four voltages down, since the analog input pins allow a maximum of 10 V.

A program was written in C, using the Eagle software development kit, to read the analog input pins, re-scale the measurements using calibration values, and display the four battery voltages. When the voltages fall below hard-coded minimum values, the program sounds an audible alarm through the computer's main board speaker, and changes font color from yellow to red. A shortcut to this Battery Monitor program was placed in the platform computer's Startup folder to cause the program to load when Windows starts.



```
12.17   15.10   18.36   18.97
```

**Figure 4:** Battery Monitor Program, indicating healthy charges.

## 4.2. Onboard Computing

Several factors led to the decision to implement onboard processing. First, no solution had yet been found for transferring the LN-200 IMU data via wireless hardware, USB, or any other indirect method. In addition, the bandwidth between the Gumstix computer and an off-board controller proved to be inadequate for effective closed-loop control.

Although developing specialized hardware might have provided a possible solution, in the interest of time and system reliability, it was preferred to use off-the-shelf solutions wherever possible. Ultimately, a dedicated onboard x86-based computer was determined to be the best match for data acquisition and processing.

An onboard data acquisition computer makes several data processing strategies possible. In the first two options, the onboard data acquisition computer also performs attitude control processing. Since Simulink was chosen as the development environment for attitude control in this project, an onboard controller would need to either run Simulink models within a full onboard Simulink installation, or run executables generated off-board using Simulink's autocoding tools. An onboard Simulink installation would also require a fully-integrated graphical operating system, like Windows or Linux, whereas running auto-coded control software might only require a console-based operating system to execute control schemes. Of these two onboard controller strategies, running Simulink models directly onboard would be the most straightforward for the controls developer, but would cost far more CPU cycles, thus limiting the iteration rate of controller models. Alternatively, the onboard computer can simply gather, package, and export sensor data to an off-board controller, and execute commands returned from the controller. This method would require the least onboard processing power, although exporting data wirelessly would generate lag between the sensors and the controller. Table 1 summarizes the comparison of the three processing strategies for real-time attitude control of the CalPoly spacecraft attitude dynamics simulator.

**Table 1:** Comparison of Processing Strategies

| Scheme | Pros | Cons |
|---|---|---|
| Offboard controller | • Effectively unlimited processing resources | • Data transport introduces lag |
| Onboard controller, running Simulink | • Straightforward<br>• Minimal lag | • Matlab overhead limits iteration rate |
| Onboard controller, running autocode | • Minimal lag<br>• Fast iteration rate | • Convoluted deployment process |

The viability of each of the aforementioned processing options was determined by studying available hardware capabilities. A brief market survey revealed that the PC/104 architecture is the current standard for applications in which significant processing power is required but power and volume are constrained. Due to the popularity of the PC/104 standard, many PC/104 compatible data acquisition modules are commercially available. Furthermore, the processor speeds of available main boards are sufficient to run Windows XP with Matlab and Simulink, and therefore would support any of the three processing methods. The PC/104 architecture was decided to be ideal for data acquisition onboard the spacecraft simulator, and a complete PC/104 stack was designed and built.

Of the many PC/104 single board computers available on the market, the Kontron MOPS-PM PC/104+ board was selected as the system main board because it provided a good compromise between cost and processing power. The PC/104+ specification includes a PCI stack-through connector in addition to the standard PC/104 bus connector, and is therefore capable of supporting higher data transfer rates between stacked PC/104+ cards. Since PC/104+ boards are also compatible with standard PC/104 boards, the

Kontron PC/104+ main board will facilitate more options if the project requires additional PC/104 or PC/104+ expansion boards in the future.

The computer requirements list included a high-speed synchronous RS-485 serial interface for the LN-200 IMU, analog and digital inputs to read the MEMS gyros and the reaction wheel tachometer circuits, analog and digital outputs for commanding the reaction wheel driver circuits, and a single RS-232 serial port to command the automatic fine balance controllers. An additional four analog input pins could optionally allow the user to monitor the levels of each of the four platform batteries. In total, the system would need twelve analog input channels, four analog output channels, and twelve digital input/output pins, one high-speed synchronous serial port, and one RS-232 serial port.

The Eagle Technologies PC104PLUS-30C data acquisition board was selected to perform the analog and digital input/output functions, while a ConnectTech ComSync/104 board would handle the LN-200 IMU's serial data stream. A Cisco wireless USB adapter was chosen to provide wireless communication. Finally, the power requirements of these boards were totaled, and a PC/104 power supply board with sufficient power support capability was selected. The complete architecture is summarized below in Table 2, and the final build is shown in Figure 5.

**Table 2:** Computer Component Summary

| Component | Features | Function |
|---|---|---|
| Kontron MOPS-PM PC/104+ | 1 GHz Celeron CPU<br>1 GB RAM | Main processor |
| Eagle PC104PLUS-30C DAQ | 16x Analog input channels<br>4x Analog output channels<br>24x Digital I/O channels<br>3x Digital counters | Reading MEMS gyros (4x A/D)<br>battery level sensing (4x A/D)<br>RW tachs (4x A/D, 4x DI)<br>RW command (4x D/A, 8x DO) |
| ConnectTech ComSync/104 | 2x synchronous serial ports<br>Up to 4.9 Mbps transfer rate | Reading LN-200 IMU data stream |
| Tri-M HE104-DXL Power Supply | 60W, +5V@12A, +12V@2.5A | Power conditioning |
| Cisco Wireless-G USB adapter | 54 Mbps, USB 2.0 | Wireless communication |

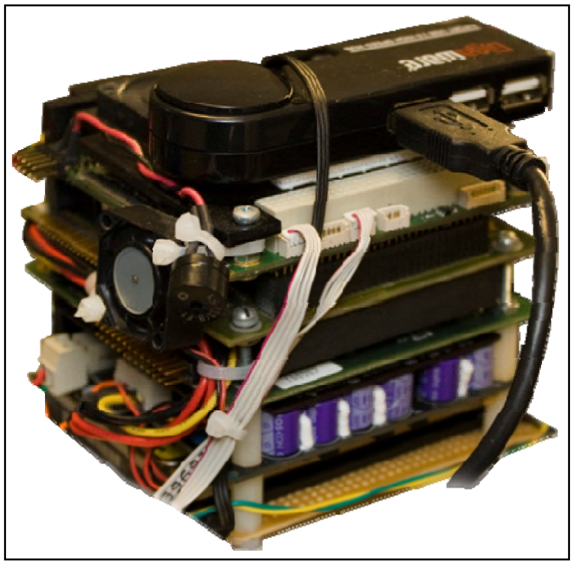


**Figure 5:** PC/104 Flight Computer

## 4.3. Sensor Interfaces

Two sensor interface boards were installed in the PC/104 flight computer: a Connecttech Comsync/104 synchronous serial board to read the LN-200 IMU's data line, and an Eagle Technologies data acquisition board to read all other sensors and to output control signals. Each of these boards shipped with Windows driver files, a software development kit (SDK), and various diagnostic and calibration utilities. However, since Matlab does not include built-in support for these boards, functions needed to be created to access the data from both the Eagle DAQ and the Comsync boards.

The SDKs provided for each board were designed for programming in C only. To interface with Matlab, compiled C programs need a form of inter-process communication. A set of Matlab executables, also called mex files, were created to perform functions from each board's SDK. Mex functions are Matlab's equivalent of Windows' dynamic link library file type; they are fully-independent executable programs which share data with Matlab. In Matlab, they are used in the same way as other functions; a function call is made with arguments listed within parentheses following the function name, and outputs from the function are assigned to variables on the left side of an equal sign just before the function name.

### *4.3.1. Eagle DAQ Matlab Interface*

Since the Eagle DAQ board initializes when the system starts up, no CPU effort is required for re-initialization during operation. Samples are requested from or written to the DAQ at any time using only a few Eagle SDK calls. Four DAQ interface programs were coded in C: eagleAin.c, eagleAout.c, eagleDin.c, and ealgeDout.c, and were compiled as mex functions of the same names but with the .mex file extension. These Eagle DAQ Matlab interface functions are described in detail in Table 3:

**Table 3:** Data Acqhisition Mex Functions

| Function | Purpose | Syntax |
|---|---|---|
| eagleAin | Read analog input pin | values = eagleAin(PinNumberArray) |
| eagleAout | Assign analog output pin | eagleAout(PinNumberArray,PinValueArray) |
| eagleDin | Read digital input channel | values = eagleDin(PinNumberArray) |
| eagleDout | Assign digital output channel | eagleDout(ChanNumberArray,ChanValueArray) |

While the DAQ board's analog input and output pins may be sampled or assigned independently, the 24 digital input/output pins are grouped into three 8-bit channels. The input/output modes of the pins are not independent of each other within a channel. For example, if an input call is made from any channel, all pins within the channel are switched to input mode and all are reset to zero except those which are externally maintained at higher values. The user must dedicate each channel to either input or output mode and avoid switching modes during missions. To determine the value sampled on any of the eight pins in a single channel, the decimal value must be read from the specified channel and converted into eight separate binary values. The opposite is true for assigning digital pin values; the user must keep track of the desired output values of all pins within a channel, convert the eight binary values to a decimal value, then assign that decimal value to the entire channel. Managing and decoding individual digital input/output pin values occurs in Matlab.

### *4.3.2. Synchronous Serial Simulink Interface*

A serial connection requires that a process initializes a port before capturing data from it, and the port can only be accessed from the process which initialized it. In contrast, the Eagle DAQ board allows any process to access data samples at any time without requiring each process to perform initialization tasks. This characteristic of the DAQ makes it possible to quickly instantiate a new process each time a data sample is requested. Since initializing a serial port on the synchronous serial board typically takes about one second, it would be grossly inefficient to use individual processes to capture each new message from a serial stream.

An S-function was created to handle initialization of the ComSync serial board and to allow an attitude controller to request the latest captured data on demand. However, since the IMU was not available at the time, the S-function has not yet been tested.

### *4.3.3. Reading the MEMS Gyroscopes*

The analog output signal of each of the Silicon Sensing MEMS gyroscopes varies between 0 and 5 V, and is centered near 2.5 V [4]. The deviation of the output signal from the center voltage, which is also known as gyro bias, is proportional to the spin rate about the gyro’s primary axis. Gyro bias tends to wander gradually over time.

The signal voltages of each of the MEMS gyros are measured through analog input pins in the Eagle DAQ, and delivered to Matlab through eagleAin mex function calls. A Matlab or Simulink observer then subtracts each gyro’s bias value from the measured voltage values, and scales the results by calibration constants to translate the measured voltages as spin rates about the platform body axes.

An initial control system design required the bias of each gyro to be determined prior to each mission by averaging each gyro’s signal voltage, and assumed the bias to be constant during the mission. Although bias drift significantly worsens the already poor accuracy of these gyros, neglecting bias drift may be acceptable for short missions. For longer duration missions, or when accuracy is critical, the bias of each gyro may be tracked as a separate state within a Kalman filter.

### *4.3.4. Measuring Reaction Wheel Spin Rates*

As noted previously, Downs developed the hardware portion of the wheel rate sensing system. An optical encoder on the shaft of each reaction wheel outputs a quadrature signal with frequency equal to 32 times the spin rate of the reaction wheel[10]. Downs designed a circuit to decode the quadrature signals from the encoders and output an analog voltage proportional to the quadrature frequency. The analog signals are then sensed via analog input pins on the DAQ board using the eagleAin acquisition function. Finally, the measured voltage is translated into a spin rate by applying offset and proportionality constants in Simulink. The calibration constants were determined by comparing analog tachometer circuit output signals to known wheel rates measured using an external optical frequency counter.

Although Downs' wheel spin direction circuits were able to determine the directions of the reaction wheels, the output of the circuit did not switch fast enough when the wheel speeds approached 0 RPM. To avoid instability in the wheel controllers, Downs and the author of this thesis decided to bias the wheels to 3000 counterclockwise RPM and avoid letting the wheel rates fall below 50 RPM.

### *4.3.5. Accessing the IMU Data Stream*

After lengthy discussions with a ConnectTech, Inc. technical representative, it was confirmed that the Comsync/104 synchronous serial board would be capable of reading the IMU data stream. The Comsync board will read the data clock signal generated by the IMU to trigger bit reads on the serial data line. Like the Eagle DAQ board, the ComSync shipped with an SDK for programming in C. An S-function serial client program has been written, but has not been tested with either the IMU or any similar synchronous RS-485 device. The S-function will provide angular rotation rate measurements within attitude control models.

### *4.3.6. Obtaining Star Tracker Updates*

Preparing an attitude control model in Simulink to eventually use the star tracker system posed two major programming challenges. As noted previously, Wehrman estimated that the star tracker system may require roughly eight seconds of processing time to return a quaternion measurement from a star image. Since the system cannot deliver measurements in real time, the near-real time control system requires some technique for adapting the controller to use the lagged star tracker data. This technique will be addressed in Section 5.2.

The second obstacle posed by the star tracker system was that it requires Matlab processing time. Simulink operates by executing blocks in a specific sequence, and is not designed for user-definable multitasking.

Four options were considered for processing star tracker images without interrupting an attitude control model. First, a star tracker processor model can run in parallel with an attitude control model, and can pass its data to the control model when necessary. The star tracker processor model would be comprised of triggering and data passing blocks and a Level-2 M-File S-function which contains the star tracker processing code. A second option would use named pipes to pass data from an external standalone C program into Matlab. Alternatively, the star tracker processing routine can be performed as a loop in the base workspace. The loop would use event listeners to sense when a trigger is raised in the control model, and the star tracker results would be collected from the base workspace by the control model. A fourth option wraps this loop into a Matlab graphical user interface (GUI), which runs in parallel with the control model and allows the base workspace to remain open for other tasks.

Of the four methods described above, the GUI option was preferred because it leaves the base workspace clean and could present the user with useful options. The star tracker processor GUI is shown below. The method by which an attitude controller model uses this data is described in Section 5.2.

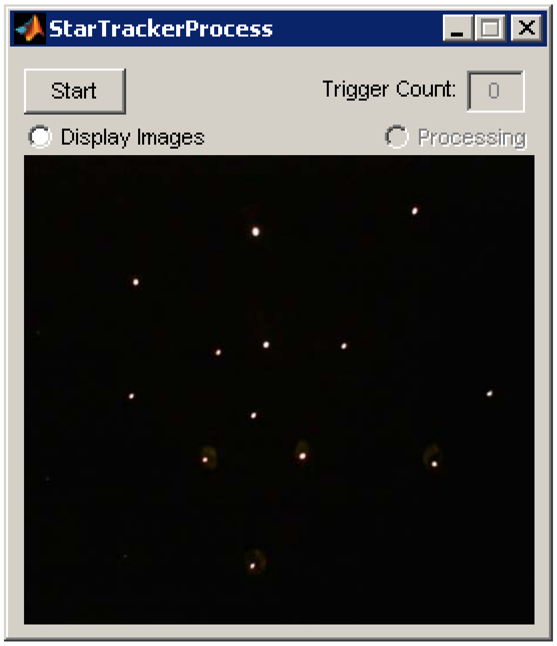


**Figure 6:** Star Tracker Processor GUI

## 4.4. Platform Actuation

The reaction wheel control system was developed, installed, and tuned by Matt Downs. The system consists of four off-the-shelf Pololu motor driver carrier circuits and four custom-designed PWM generation circuits. Each of the Pololu driver circuits requires two digital input signals to select between clockwise drive, counterclockwise drive, coasting, and braking modes.

In the current design of the reaction wheel command system, speed control is decoupled from direction control within Matlab. After a Matlab or Simulink control model determines desired reaction wheel velocities, the model translates the speeds into the necessary analog output voltage values by first taking the absolute values of the desired velocities, then applying calibration proportionality constants. These voltage values are then assigned to analog output pins using the eagleAout mex function. The model is also responsible for determining the necessary logic combination for the Pololu driver carriers and writing the values to the digital input/output pins using the eagleDout function. Figure 7 illustrates the reaction wheel command routing sequence for a single wheel. In total, the current system requires four analog output channels and eight digital pins to command all four reaction wheels.

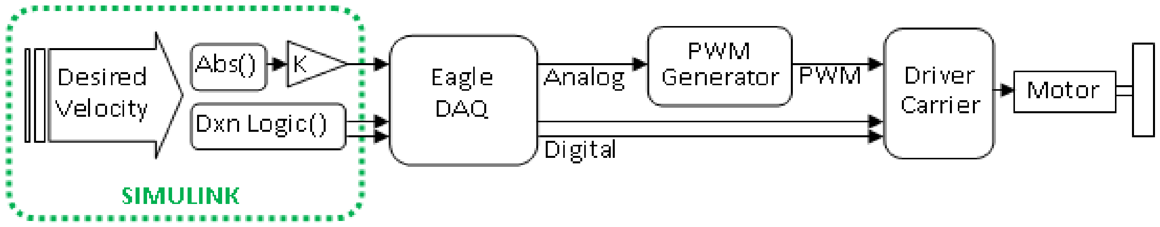


**Figure 7:** Reaction Wheel Command Signal Routing

A future version of the reaction wheel command system should externalize the decoupling of reaction wheel speed and direction commands. A control model should be allowed to output analog signals directly proportional to desired wheel velocities, and let the sign of the output signal dictate wheel direction rather than assigning digital output values. The circuit might also consider the time derivatives of the commanded reaction wheel rates in order to determine the necessary driver mode (accelerating/braking/coasting). Externalizing these functions would not only simplify control model composition, but would also significantly reduce CPU load. Lastly, revisions to the circuit should also incorporate external safety measures to better protect the driver carrier boards.

To allow precise control over reaction wheel transient responses and steady state speeds, Downs also designed proportional-integral-derivative (PID) controllers in Simulink. Downs' wheel controller design was modularized and incorporated into the SpaceSim Simulink toolbox.

## 4.5. SpaceSim Simulink Library

A custom Simulink library, titled "SpaceSim Toolbox" was developed to simplify the composition of attitude control schemes. The toolbox consists of a collection of common control system components which have been modularized and masked to look and act similar to built-in standard Simulink Toolbox blocks. To use the toolbox, the user opens the library in the same manner as opening a model.

Within the library, the modules are sorted into two categories: low-level modules and high-level modules. The set of low level modules includes analog and digital input and output blocks, and specialized reaction wheel modules. These blocks perform the hardware interface tasks for the high-level modules and in user-defined control models. Table 4 summarizes the functions of each of the high-level modules.

**Table 4:** High-Level SpaceSim Toolbox Modules Summary

| **Module Name** | **Description** |
|---|---|
| Read Gyro | Returns MEMS gyro spin rate measurement by interpreting voltage on specified analog input pin and adjusting according to specified calibration values. |
| Meas Wheel Rates | Returns spin rates of four reaction wheels. User specifies the analog pins connected to the tachometer circuits and the proportional calibration constant for each wheel. |
| Gyro Filter | A Kalman filter from Simulink Signal Processing Blockset, pre-configured to filter spin rates from a MEMS gyro. |
| Wheel Rate Filter | A Kalman filter from Simulink Data Acquisition Toolbox, pre-configured to filter a reaction wheel spin rate. |
| Track Platform | A compartmentalized attitude tracking scheme consisting of a MEMS gyro Kalman filter block and a basic quaternion integrator. |
| Command Wheels | Performs internal closed-loop control of all four reaction wheels, including measuring and filtering wheel rates. |

A portion of the SpaceSim Library is shown in Figure 8.

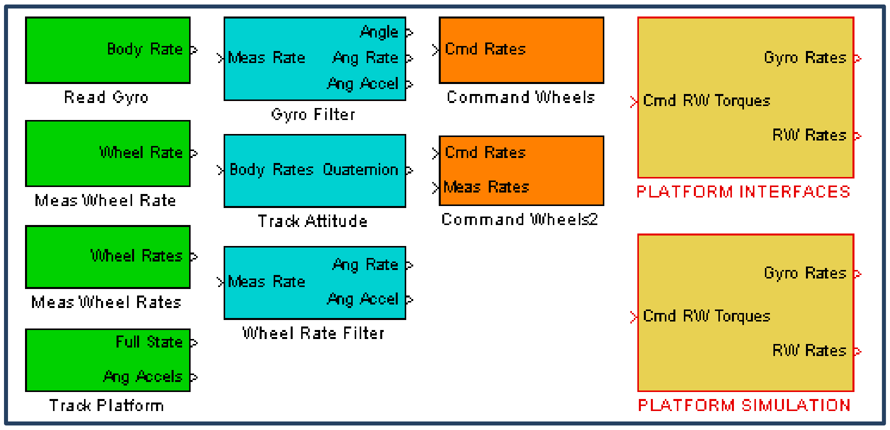


**Figure 8:** High-Level SpaceSim Library Modules

The PLATFORM INTERFACES and PLATFORM SIMULATION blocks in the above figure are swappable plant blocks, and are described in detail in Sections 6.2 and 6.3.

## 4.6. Wireless Command & Data Link

A Cisco USB wireless Ethernet adaptor has been installed on the platform computer. An ad-hoc wireless network allows the desktop computer and any other nearby computer to connect to the platform computer and access network services and resources. The platform computer is set up as an FTP server, so users may transfer files either by using an offboard FTP client, or by using shared Windows directories.

### *4.6.1. Command via Remote Desktop*

Although the Kontron main board of the flight computer features keyboard, mouse, and VGA support, frequently connecting and disconnecting these devices would be a nuisance to the user. Instead, the Windows user interface of the flight computer is accessed over the wireless network using Microsoft’s Remote Desktop Protocol. The flight computer is configured as a Remote Desktop server. When a Remote Desktop client connects, all VGA screen outputs from the flight computer are routed over the network to be displayed on the client computer’s monitor, while the client computer’s keyboard and mouse inputs are routed to the flight computer to give the feel that the user is using the flight computer directly.

Using Remote Desktop, the user can develop Matlab scripts or Simulink models directly on the flight computer without attaching a monitor, mouse, or keyboard. However, due to slight network lag, the user may find Simulink programming via Remote Desktop cumbersome, and may prefer to perform all development work on the desktop PC and transfer the files using shared directories. The Matlab and Simulink programs must still be initialized manually via Remote Desktop.

### *4.6.2. Telemetry Display*

The 802.11G network also allows data transfer using User Datagram Protocol (UDP). Unlike TCP, a UDP connection does not require any handshaking between server and client computers. Although datagrams sent over UDP contain routing information, the server computer does not check that the datagram was received correctly or even that it was received at all. This method is ideal for displaying telemetry since nothing can be damaged by erroneous data messages. The lack of overhead makes the UDP protocol ideal for situations in which a server cannot afford to wait for a client to connect or when data integrity is less critical than processing expense.

Simulink supports UDP transmissions with built-in UDP Send and UDP Receive blocks. The UDP Send blocks have been inserted in various control models and configured to send platform attitude and reaction wheel status information to the desktop computer. A Matlab script on the desktop computer captures the datagrams and parses the received data to update a figure window with several useful displays. First, the reaction wheel spin rates are plotted individually, thus allowing the user to know when the wheels are nearing saturation. Next, the quaternion is used to rotate a scaled 3-D model of the platform comprised of patches. The quaternion is also logged and displayed in a quaternion history plot. The tilt of the platform's body z-axis is determined from the latest quaternion and displayed in a polar plot in which the radial dimension represents tilt angle from vertical. A red circle in the tilt plot indicates the region in which the platform is in danger of contacting the air bearing support. When the platform hits this maximum allowable 30° tilt angle, an audible beep is sounded through the desktop's motherboard speaker. The telemetry display figure window is shown here:

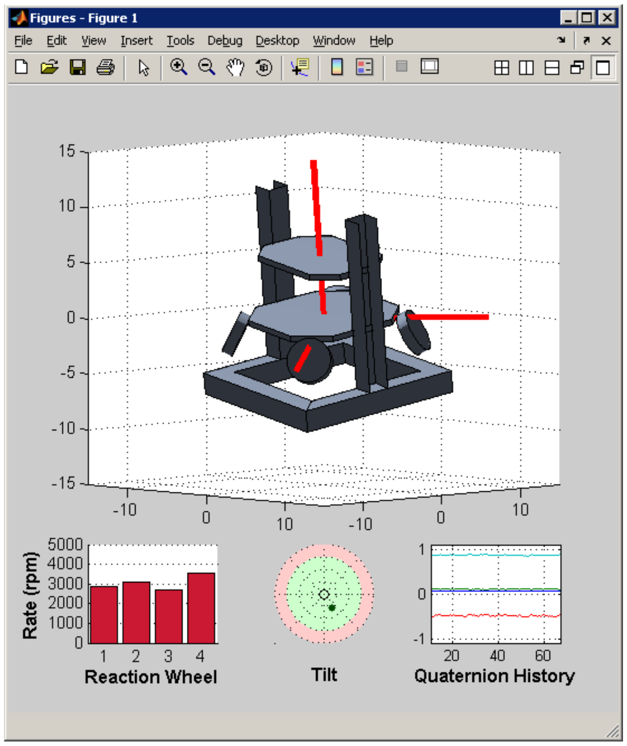


**Figure 9:** Telemetry Display Figure Window

## 4.7. Testing & Troubleshooting Utilities

A collection of subsystem testing utilities has been developed. The script titled tests_setup.m contains most of the initialization procedures and variables required by the test models, and should be executed prior to running any of the tests. Table 5 lists each of the main test utilities and the purpose of each. It is recommended that the user execute some of the tests to ensure the sensors and data acquisition, and the command and control systems are in full working order prior to attempting to execute an attitude control mission. When testing reaction wheel control, the user should be ready to shut down the reaction wheel power circuit immediately if any of the wheels seems to spin too fast or does not spin at all.

**Table 5:** Platform Test Utilities

| **Test Model** | **Function** |
|---|---|
| GyrosReadTest.mdl | Displays angular velocity measured by gyros. |
| FullStateTrackingTest.mdl | Uses a rudimentary gyro integration scheme to track the platform attitude while a user turns the platform by hand. |
| WheelRateReadTest.mdl | Displays the wheel rates of each reaction wheel to ensure the optical encoders and data acquisition system are operating properly. |
| WheelControlTest.mdl | Spins each wheel to test command system functionality. |
| MissionTest.mdl | Visualizes a mission plan by forwarding command quaternions to the Telemetry Display program on the off-board computer. Useful for confirming a plan will result in the trajectory intended by the mission designer, and for ensuring the plan does not try to drive the platform beyond its physical limits. |

## 4.8. System Identification & Fine Balancing

The system identification algorithm developed by Healey was reprogrammed by Downs to operate on logged mission data. This system estimates the inertia tensor of the platform and the offset of the platform's center of gravity from its center of rotation.

Downs also derived a set of equations to compute the position adjustments of the fine balance system masses required to correct the center of gravity offset.

The fine balance mass mover controllers were daisy-chained after assigning unique serial addresses, so that all three controllers can be commanded over a single serial connection. A function titled moveFineBalanceMasses was written in Matlab to handle the serial port initialization and message transmission. An automatic balancing script uses Downs' system identification routine to prescribe mass adjustments, and then executes the adjustments using the moveFineBalanceMasses function. Also, a graphical user interface, shown below, was created to help the user adjust fine balance system masses individually.

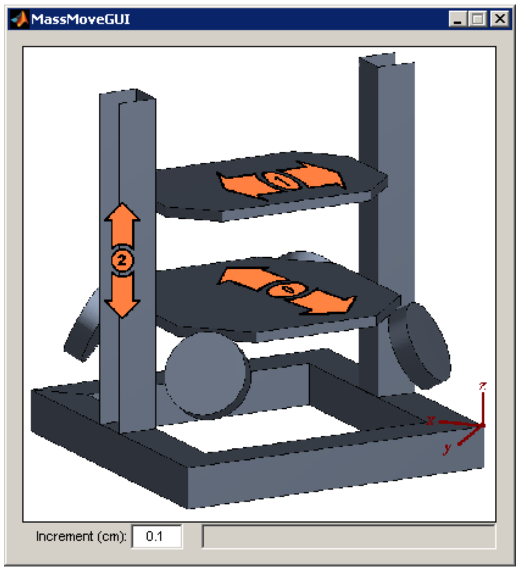


**Figure 10:** Fine Balance Manual Adjustment GUI

# 5. Attitude Determination

The attitude of the platform at each time step will be described as a rotation relative to some arbitrary initial attitude. To avoid the mathematical singularities inherent to Euler angle-based attitude tracking, modern spacecraft control systems typically use quaternions. In the case of the CalPoly spacecraft attitude dynamics simulator, the pitch and roll limitations restrict the platform from approaching orientations in which such Euler angle singularities occur. Regardless, the simulator control system will employ quaternions, in light of industry and project members' preference.

Until the IMU and star tracker systems have been fully installed, mission times are limited to less than two minutes due to accumulated gyro drift error and reaction wheel saturation. Since the local topocentric frame rotates very little relative to inertial space in this short period of time, the local topocentric frame is treated as inertial and no coordinate transformations are used.

## 5.1. Quaternion Integration

The attitude of the simulator platform is tracked by integrating the measured body axis rotational velocity vector. According to Bong Wie, the body spin rates are translated to a quaternion time derivative according to the following equation[38]:

$$\dot{q}_k = \frac{1}{2}\begin{bmatrix} 0 & \omega_z & -\omega_y & \omega_x \\ -\omega_z & 0 & \omega_x & \omega_y \\ \omega_y & -\omega_x & 0 & \omega_z \\ -\omega_x & -\omega_y & -\omega_z & 0 \end{bmatrix}_k \cdot q_{k-1} \tag{1}$$

where $\omega_x$, $\omega_y$, and $\omega_z$ represent the spin rates about each body axis. A 4 x 4 matrix Ω will be defined as the skew-symmetric body rates matrix in equation 1 above. If the body rates matrix was known to be constant, then the quaternion at any moment in time could be determined as the general solution to first-order ordinary differential equations, in matrix form:

$$q(t) = e^{\frac{1}{2}\Omega(t-t_0)} q_0 \quad , \tag{2}$$

where $q_0$ represents some arbitrary initial reference quaternion, and the exponential is the matrix exponential function.

In the case of the spacecraft simulator, the body spin rate matrix will never be constant during an entire mission. However, a discrete integrator with small time steps can still use this solution as a very close approximation by linearizing about the current body rates. In other words, the body spin rate matrix will be updated by new gyro measurements during each iteration, and will be assumed constant over each iteration period. Due to 2-nd order smoothing effects of inertia and the limited torque available, the platform's angular acceleration will always be small, and therefore the constant body spin rate assumption is considered valid over small time steps. The discrete approximate solution is then described by

$$q_k = e^{\frac{1}{2}\Omega_\mathrm{k}(t_k-t_{k-1})} q_{k-1} \, . \tag{3}$$

While this form can be used in Matlab as the quaternion discrete integrator, a less computationally expensive form was derived using a Taylor Series expansion of the above matrix exponential. Within the expansion, two patterns were recognized as sine and cosine expansions, and the series was condensed and inserted into the discrete solution to form:

$$q_k = \left[\frac{\Omega}{|\vec{\omega}|} sin\left(\frac{1}{2}|\vec{\omega}|\Delta t\right) + I_{4x4} cos\left(\frac{1}{2}|\vec{\omega}|\Delta t\right)\right] q_{k-1} . \qquad (4)$$

This version of the discrete body rate integrator will be implemented in the inertial sensors Kalman filter, which will be described in Section 5.3.1.

## 5.2. Incorporating the Star Tracker System

As of this writing, the star tracker system was still in development. One of the goals of this project, however, was to incorporate the star tracker system with the inertial sensors as part of a fully-functional inertial navigation system. The attitude determination routine described here was designed to facilitate the star tracker system once it has been installed.

The image processing time required by the star tracker system presented a significant challenge to the real-time navigation system. After acquiring each new image, the star tracker processing routine begins analyzing the image to determine the orientation of the platform at the instant the image was taken. Wehrman estimated conservatively that each image may take as long as eight seconds to process. Therefore, by the time the image processing completes, the star tracker reports a quaternion which represents the orientation of the platform eight seconds ago. The navigation system cannot treat the star tracker data as a measurement of the current platform attitude.

An original technique was developed to adapt the real-time attitude determination system to accept the severely lagged star tracker data. At any time during a mission, the current attitude is calculated as the platform's attitude when the previous image was captured, rotated by the relative rotation after the previous image and before the latest image, and then by the relative rotation since the latest image was captured. In quaternion form, this three-part rotation sequence is implemented as follows:

$$q_{current} = q_{prev\ img} \otimes q_{prev\ img \to latest\ img} \otimes q_{latest\ img \to current\ time} \qquad (5)$$

A loop within the attitude determination routine will trigger the star tracker to acquire a new image once every ten seconds, thus allowing margin in image processing time. The same loop also updates a dedicated extended Kalman filter, titled the Star Tracker Filter (STF), which tracks the orientation of the platform at the time of the previous image capture. In Equation 5 above, $q_{prev}$ represents the latest STF estimate, which is updated every ten seconds.

A separate filter, titled the Inertial Sensors Filter, tracks the deviation of the platform after the latest image capture by integrating gyroscope measurements. This filter operates at or near 100 Hz. When the star tracker loop captures a new image, the loop will also store the latest total deviation value from the ISF and reset the ISF back to a zero-rotation quaternion. This stored value represents the deviation of the platform, according to the inertial sensors, between the previous and the latest images, and is updated every ten seconds. Equation 5 above can be rewritten to emphasize the origin of each value as:

$$q_{current} = q_{Star\ Tracker\ Filter} \otimes q_{prev\ img \to latest\ img} \otimes q_{Inertial\ Sensors\ Filter} \quad . \qquad (6)$$

## 5.3. Kalman Filter Design

The purpose of Kalman filtering is to maintain a better estimate of the actual state of a system than a state observation alone can provide. This is accomplished by comparing each observed state to predicted values and selecting a combination of the two which minimizes accumulated estimation error. During the prediction phase of a Kalman filter, values of a state vector and a state noise covariance matrix are projected from previous values using a state transition model. Although these values are predictions of the state and state error covariance at the current time step, they are termed “a-priori” estimates because the estimates are produced prior to considering the latest sensor measurements. The update phase of a Kalman filter combines the a-priori estimates with a state vector generated from actual sensor data to produce an “a-posteriori” state estimate. The Kalman gain calculated during each filter iteration designates the amount by which the a-posteriori will be influenced by the observed state versus the a-priori predicted state[12].

For several reasons, it has been found to be most convenient to develop the star tracker and inertial sensors filters separately. First, as was described in the previous section, the star tracker filter maintains a lagged estimate of the platform attitude quaternion, while the inertial sensors filter provides a real-time estimate of the angular velocity vector. The inertial sensors sample as fast as or faster than the control model iteration rate, while the star tracker system is expected to require around eight seconds to generate attitude estimates. Furthermore, due to the nonlinearity of the process of incrementing a quaternion, the application of the Kalman gain in the star tracker filter will be very different from that of the inertial sensors filter.

### 5.3.1. Inertial Sensors Kalman Filter

The inertial sensors Kalman filter is the simpler of the two filters and will be described first. Prior to incorporating each new measurement, the filter predicts the platform spin rates by integrating the angular accelerations which the filter estimates based on the current commanded net torque vector. This *a-priori* estimate of the body angular velocity vector is designated by the "-" superscript and is found at the k'th time step as

$$\vec{\omega}^{-}{}_{\mathrm{k}} = \vec{\omega}_{k-1} + J^{-1}\vec{T}_{cmd_{\mathrm{k}}} \cdot (\mathrm{t}_{\mathrm{k}} - \mathrm{t}_{\mathrm{k-1}}) \quad . \tag{7}$$

In the general form of a linear discrete a-priori update equation, a state transition matrix exists between the previous state estimate and the current a-priori estimate. In this case, the state transition matrix, $A$, is equal to the 3 x 3 identity matrix, and while it is not shown in Equation 7, it will be necessary for the update of the a-priori estimate error covariance matrix, $P^{-}$. The following equation updates this matrix during the prediction phase of the filter, before considering the latest IMU measurements[12]:

$$P^{-}{}_{k} = A \cdot P_{k-1} \cdot A^{T} + Q \ , \tag{8}$$

where Q represents the process noise covariance matrix. The process noise covariance matrix will be assumed constant and small since no significant external noise sources are expected to effect the process, and also diagonal with equal diagonal values since the gyroscopes would presumably be affected equally by external noise and are not expected to affect each other in any way. For the spacecraft simulator, the constant process noise covariance matrix will typically be hard-coded as

$$Q = 0.000001 \cdot I_{3x3} \quad , \tag{9}$$

and an initial value for the estimate error covariance matrix is set to

$$P_0 = 0.25 \cdot I_{3x3} \quad . \tag{10}$$

Next, the update phase will combine the a-priori spin vector estimate with IMU spin rate measurements to generate an improved state estimate. This process will require values for the device measurement matrix, H, and the measurement covariance, R. The measurement matrices for both the IMU gyros and the MEMS gyros will be 3 x 3 identity matrices since the sensor's axes will be aligned with the platform's body axes and since no scaling is required.

The measurement covariance matrix will vary depending on which sensor is used. In both cases, the measurement matrix will be formed as a 3 x 3 diagonal matrix with its diagonal values equal to the standard deviation of the sensor's measurements. Since the standard deviation value of the LN-200's laser-ring gyros could not be found in the device's manual, and since the IMU was not available for experimentation at the time of this writing, the standard deviation is assumed to be 0.001 rad/s. Based on the 0.1°/hr drift rate reported in the IMU manual, this standard deviation value is expected to be too high by multiple orders of magnitude and should be reduced after experimentation. The standard deviation of the MEMS gyro measurements was estimated to be 0.5 rad/s.

Next, the Kalman gain will be calculated as[12]

$$K_k = P^-{}_k \cdot H^T \cdot [H \cdot P^-{}_k \cdot H^T + R]^{-1} \quad . \tag{11}$$

With all of the necessary elements defined, the a-posteriori spin rate state estimate is found as a linear combination of the measured and a-priori state estimates:

$$\vec{\omega}_k = \vec{\omega}^-{}_k + K_k \left( \vec{\omega}_{Gyro_k} - H \cdot \vec{\omega}^-{}_k \right) . \tag{12}$$

From Equation 12, one can see that the Kalman gain determines the amount by which the filter will "trust" the measured state over the predicted state. If, for example, the diagonal terms of the Kalman gain are nearly equal to one, then the measured state will be favored over the estimated state, while near-zero values will force the a-posteriori state to track the prediction.

Finally, the a-posteriori estimation error covariance is updated by the following equation[12]:

$$P_k = (I_{3x3} - K_k \cdot H) \cdot P^{-}{}_k \quad . \tag{13}$$

The inertial sensors Kalman filter was tested and tuned by observing the output of the filter as it operated on a real stream from the MEMS gyros. Effectiveness of the filter is clearly seen in Figure 10, which compares filter outputs to raw sensor streams:

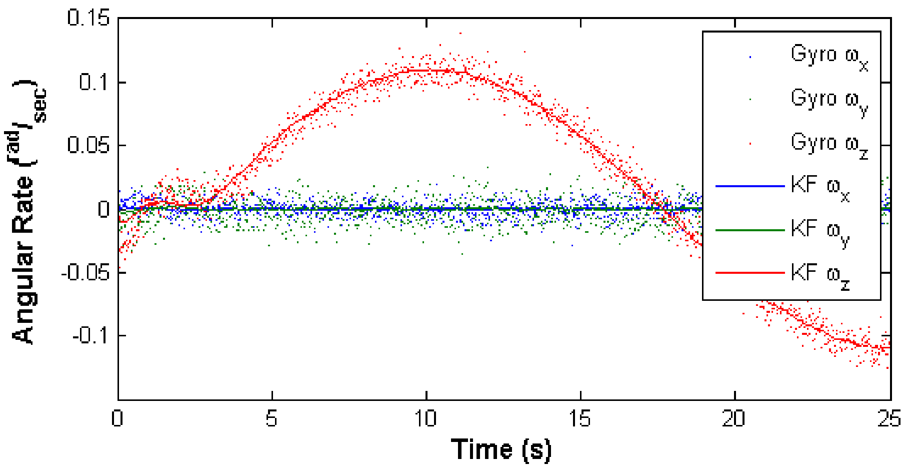


**Figure 11:** Comparison of Inertial Sensors Filter Output to Raw MEMS Gyro Data

### *5.3.2. Star Tracker Kalman Filter*

Unlike the real-time inertial sensors Kalman filter, the star tracker filter estimates a former platform state. This fact simplifies the prediction step since an estimate of the platform's attitude already exists for the time at which the previous image was captured. This stored attitude quaternion will be used as the a-priori estimate.

Although the a-priori quaternion estimate has already been calculated, the star tracker Kalman filter requires a state transition matrix in order to maintain the estimate noise covariance. It is important to note that the a-priori estimate was originally generated using the previous star tracker Kalman filter attitude quaternion, which was propagated by integrating gyroscope measurements, but that the state transition matrix is still required for the error covariance prediction. This matrix will be found after recalling the rotation sequence which generated the stored (a-priori) estimate:

$$q_{latest\ img} = q_{prev\ img \rightarrow latest\ img} \otimes q_{prev\ img} \quad , \tag{14}$$

where $q_{prev\ img}$ represents the previous star tracker filter quaternion estimate, $q_{prev\ img \rightarrow latest\ img}$ translates as the rotation of the platform in the time between the previous and latest images, and $q_{latest\ img}$ is the attitude estimate stored at the time the latest image was captured. The quaternion multiplication in Equation 14 can be transcribed into matrix form, resulting in[11]

$$q^{-}_{m} = \begin{bmatrix} q_4 & -q_3 & q_2 & q_1 \\ q_3 & q_4 & -q_1 & q_2 \\ -q_2 & q_1 & q_4 & q_3 \\ -q_1 & -q_2 & -q_3 & q_4 \end{bmatrix}_{m-1 \rightarrow m} \cdot q_{m-1} \ . \tag{15}$$

Although the a-priori sate estimate already exists, the matrix in the above equation is used in the calculation of the a-priori estimate noise covariance matrix.

Designing the star tracker Kalman filter requires an understanding of the noise sources in the star tracker system. The anticipated noise sources include star field positioning and mapping error, camera alignment error, limited camera resolution, and calculation error. These noise sources may be formally characterized after the star tracker has been installed, but for now, a few assumptions will be made in lieu of empirical data. First, the star dome and star tracker camera may be moved inadvertently between missions, but will remain reasonably stationary during a mission. Since these offsets cause persistent error in the attitude calculations, they are categorized as process noise. The process noise covariance matrix Q will contain a significantly large value of 0.0001 along its diagonal.

The camera resolution and calculation error were categorized as measurement noise sources. To form the measurement noise covariance matrix, an approximation was made for the standard deviation of the star tracker attitude measurements. Based on a conservative guess of 3° typical Euler angle error, the standard deviation of each output quaternion term was assumed to be 0.05. The star tracker measurement covariance matrix was formed with the square of this value as its diagonal terms.

The star tracker Kalman filter prediction phase will use Equation 8 to calculate the a-priori estimate covariance. The state transition matrix in this case will be the deviation quaternion rotation matrix described in Equation 15. The diagonal terms of the quaternion estimate covariance matrix will be initialized as 0.25 to accommodate sloppiness in the initial orientation of the star dome relative to the platform.

Despite the nonlinear nature of quaternion interpolation, the star tracker filter Kalman gain will be applied in the same linear manner as was performed by the inertial sensors filter. A spherical linear interpolation scheme was considered but not adopted because the method does not allow the Kalman gain matrix to operate on each quaternion component individually. The a-posteriori estimate during the m'th iteration of the star tracker filter is calculated by linear combination according to:

$$q_m = q^-{}_m + K_k\left(q_{ST_m} - H \cdot q^-{}_m\right) \ . \tag{16}$$

Finally, in order to maintain unitary magnitude of the quaternion, the a-posteriori estimate must be normalized at each iteration step. To preserve both the axis direction and rotation angle represented by the quaternion, a non-standard normalization procedure has been devised. Scaling the entire quaternion by its own 2-norm, as is commonly done with vector normalization, changes the magnitude of the quaternion scalar term, thereby altering the represented rotation angle. Instead, only the vector part of the quaternion will be scaled by an amount designed to normalize the entire quaternion. To avoid a singularity in this process, the normalization function must first ensure the norm of the vector part is non-zero. The complete normalization function is defined as follows:

$$QuaternionNormalize(q) \equiv \left\{ \begin{array}{c} \begin{bmatrix} \frac{\vec{q}\sqrt{1-q_4{}^2}}{|\vec{q}|} \\ q_4 \end{bmatrix}, |\vec{q}| \neq 0 \\ \begin{bmatrix} \vec{0} \\ \frac{q_4}{|q_4|} \end{bmatrix}, |\vec{q}| = 0 \end{array} \right\} , \tag{17}$$

Lastly, the a-posteriori error covariance matrix is updated using Equation 13 with the star tracker filter's a-priori error covariance, Kalman gain, and measurement matrices.

# 6. Attitude Control

A closed-loop attitude controller design was first simulated numerically and later implemented and demonstrated onboard the spacecraft simulator platform. A general design for a closed-loop attitude control system using reaction wheels was adapted by Mittelsteadt for use with the reaction wheel geometry specific to the CalPoly spacecraft simulator. What follows is a high-level description of the method, with focus on the unique aspects of the current implementation.

## 6.1. Optimal Attitude Controller Design

Any attitude control scheme for the simulator platform will feature several necessary components, including a state observer, a feedback strategy, and a method for resolving reaction wheel commands. The general closed-loop system is illustrated in block form below.

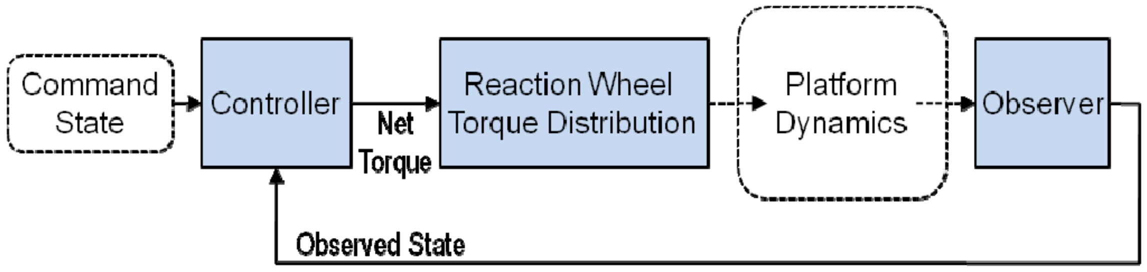


**Figure 12:** General closed-loop control scheme

This general form will be used in the designs of both the numerical simulation control system and the physical simulator control system.

### *6.1.1. Optimal Full-State Feedback Controller*

As its name implies, a full-state feedback (FSFB) controller uses knowledge of the full state of a system to resolve an input which will drive the system to a desired state. In the case of the CalPoly spacecraft simulator FSFB controller, the state vector consists of a quaternion which represents the attitude of the platform, and the platform's angular velocity, which is a measure of the first time derivative of attitude.  The FSFB controller uses this state to determine a net torque which minimizes the difference between the commanded and observed states.

A quaternion error vector is calculated by comparing the commanded and observed states.  This quaternion error vector and the angular vector are scaled by the matrices C and K, respectively, and then summed to calculate a net torque vector.  The Simulink implementation of this calculation is presented as Figure 13.

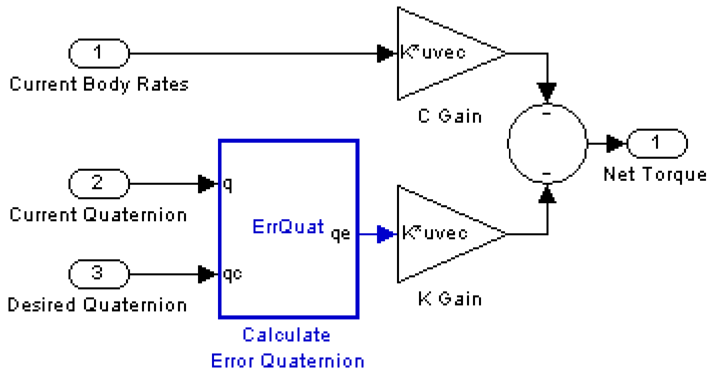


**Figure 13:**  Full State Feedback Controller

Since the system input is formed as a combination of representations of the attitude and its derivative, the FSFB controller essentially acts as a proportional-derivative attitude controller.  It should be noted that, without an error integration contribution in the torque calculation, the system may settle with some amount of steady-state error.  This could potentially manifest as a small oscillation near a commanded quaternion.

It can be shown that this feedback scheme offers arbitrary pole placement[11].  In other words, the FSFB controller gives the designer freedom to shape the transient response of the closed-loop system. Mittelsteadt worked out formulae for calculating gain matrices which drive the system to exhibit desired natural frequency $\omega_n$ and damping ratio $\xi$ values. The constant C and K gain matrices are calculated as[2]

$$C = 2\xi\omega_n \begin{bmatrix} J_{xx} & 0 & 0 \\ 0 & J_{yy} & 0 \\ 0 & 0 & J_{zz} \end{bmatrix}, \text{ and} \tag{18}$$

$$K = 2{\omega_n}^2 \begin{bmatrix} J_{xx} & 0 & 0 \\ 0 & J_{yy} & 0 \\ 0 & 0 & J_{zz} \end{bmatrix}. \tag{19}$$

These gains are calculated prior to a numerical simulation or simulator mission and are typically held constant. However, actively controlling the closed-loop system's transient response by adjusting these gains during a mission could prove tremendously useful.

Next, some combination of reaction wheel accelerations must be found to generate the prescribed net torque. Before doing so, the net torque must be adjusted to account for gyroscopic torques generated when the spin vectors of any spinning components are forced to change direction. For example, when the platform is at rest but the wheels are turning at steady bias rates, then applying a net torque which is intended to accelerate the platform by a certain amount will inadvertently generate a counter-torque due to the reluctance of the reaction wheel spin vectors to allowing their orientations to be changed. A portion of the commanded torque will be converted to platform momentum, as intended, but some will be expended to match the gyroscopic torque.

Distributing the reaction wheel torques is not a straightforward process since an infinite set of reaction wheel torque combinations may satisfy the net torque constraint.

In other words, the torque vectors of the four reaction wheels do not form a minimal spanning set of 3-space. Therefore, summing the torque contributions of the four reaction wheels yields three net torque component equations with four unknown reaction wheel torques. The system is under-constrained and a unique solution cannot be found without introducing an additional constraint. Wie suggested a minimum effort approach which generates a fourth constraint by considering the partial derivatives of the sum of the wheel torque magnitudes[11]. This process of minimizing the sum of the reaction wheel acceleration magnitudes results in the least amount of power drawn from the onboard battery system, and may also potentially extend the lives of the motors. Mittelsteadt adapted this method for the CalPoly reaction wheel system.

The following equation combines the geometric relationships between the four reaction wheels and the components of the prescribed net torque with the equation derived from the summed wheel torques minimization[11]:

$$\begin{bmatrix} T_{x_{cmd}} \\ T_{y_{cmd}} \\ T_{z_{cmd}} \\ 0 \end{bmatrix} = \begin{bmatrix} sin(\beta) & 0 & -sin(\beta) & 0 \\ 0 & sin(\beta) & 0 & -sin(\beta) \\ cos(\beta) & cos(\beta) & cos(\beta) & cos(\beta) \\ 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} T_{RW1} \\ T_{RW2} \\ T_{RW3} \\ T_{RW4} \end{bmatrix} . \qquad (20)$$

Inverting the above matrix allows the optimal combination of reaction wheel torques to be determined as a function of the required net torque. With knowledge of the inertias of the reaction wheels, the wheel accelerations are calculated and integrated over the iteration period to determine a new set of wheel rates to be commanded. This relationship was encapsulated as a Simulink block and used by the numerical simulation and physical simulator control schemes described in the following subsections.

## 6.2. Numerical Simulation Model

The purpose of numerical simulation in this and future CalPoly spacecraft simulator endeavors is to validate control schemes prior to deployment on the platform. The simulation model shown the figure below incorporates several features to ensure accurate simulation of the platform's true behavior, including realistic sensor interfaces, and was designed to maximize compatibility with future students' models and control ideas.

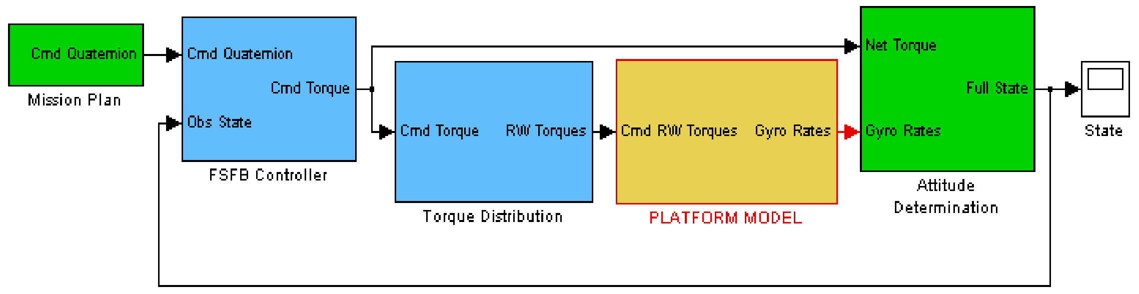


**Figure 14:** Numerical simulation model, top level

In the above model, the Mission Plan, FSFB Controller, Torque Distribution, and Attitude Determination blocks exactly match those intended for real-time physical simulator control models, and will be described in detail in Section 6.3. The Platform Model block is the only unique block in the numerical simulation model, and essentially represents the hardware interfaces as well as all physical interactions within the system. The internal composition of the Platform Model block is illustrated in Figure 15:

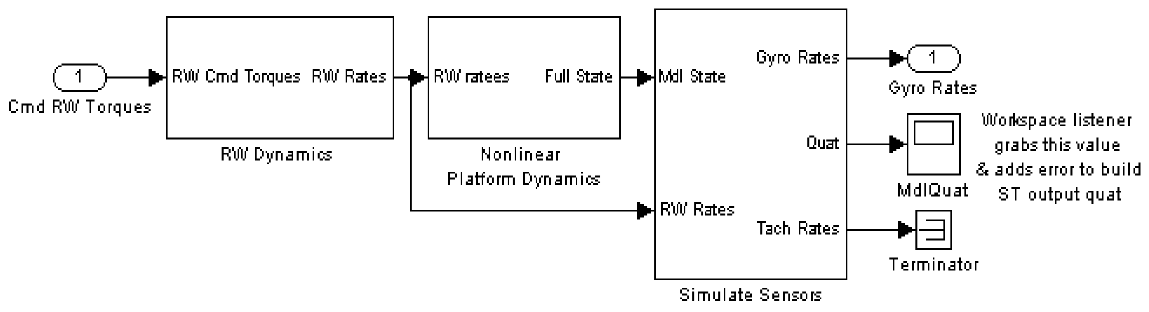


**Figure 15:** Plant simulation model

The RW Dynamics block handles both the dynamics of each motor and their associated PID controllers. The Nonlinear Platform Dynamics block predicts the platform's response to reaction wheel accelerations according to a set of nonlinear equations of motion. The general spacecraft equations of motion derived by Wie were adapted by Mittelsteadt to represent the CalPoly spacecraft simulator platform. Mittelsteadt also derived a set of equations which describe the responses of the reaction wheel motors. Finally, the plant synthesizes sensor measurements by adding realistic noise and lag to the components of the modeled system state.

This numerical simulation model was used to validate the controller, torque distribution, and attitude determination blocks prior to implementing the control system design in real-time onboard the platform. Section 6.3.4 compares both the numerical simulation trajectory and the platform's trajectory to commanded trajectories.

## 6.3. Physical Attitude Control Implementation

The optimal full-state feedback control system was adapted for real-time control of the simulator platform by incorporating the hardware interface modules in the SpaceSim Library. As shown in Figure 16 below, the general form of the system and each of the core components have been retained from the numerical simulation model.

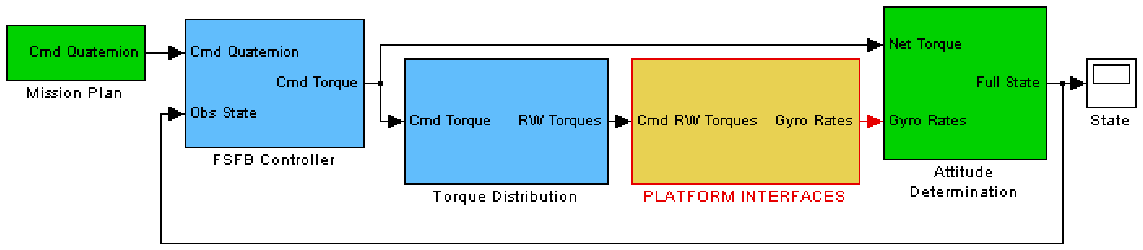


**Figure 16:** Simulator optimal FSFB control scheme

### 6.3.1. Mission Definition

The Mission Plan block executes a user-defined schedule of attitude command quaternions. This schedule is composed as a Matlab function in a .m file, and the function handle is specified in the base workspace and referred to by the Mission Plan block in Simulink. This scheme allows the user to make changes in a mission plan or select a different plan without having to wait for Simulink to recompile the model.

The simulation time is passed as input to the schedule function, and the output of the function must be a 5-element array containing a command quaternion and a flag value which is used to indicate that the mission has completed. The mission definition function may define quaternion command sequences either as piecewise step functions (i.e. waypoints) or as continuous functions of time, or a combination of the two, so long as the function defines a command quaternion for every time value. The following code defines a very basic mission plan consisting of two z-axis turns:

```
function out = missionplan(t)
if t<60, quitflag = 0;
else quitflag = 1;
end;
if t<10 || t>40, qcmd = [0 0 0 1];
else qcmd = [0 0 sin(pi/4) cos(pi/4)];
end;
out=[qcmd,quitflag];
```

### 6.3.2. Real-Time Attitude Determination

The previous chapter described the strategy designed to maintain a real-time platform attitude estimate by combining the lagged star tracker data with gyroscope measurements. This strategy is contained in the Attitude Determination module, displayed here:

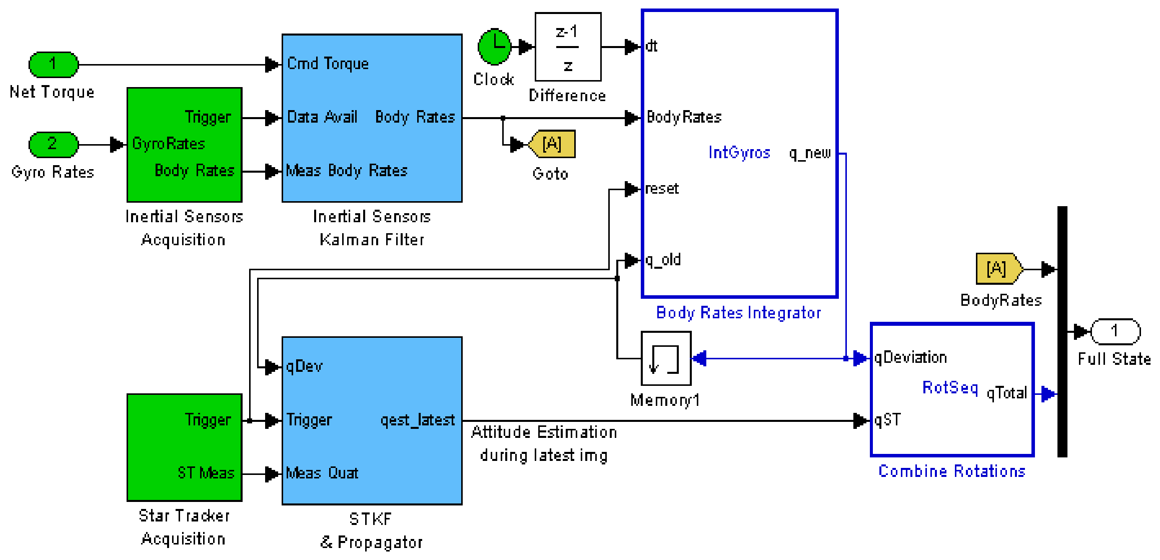


**Figure 17:** Attitude Determination module

The Inertial Sensors Acquisition generates a trigger signal for the Inertial Sensors Kalman Filter block, and latches onto the latest gyroscope measurements. The measured angular velocity vector is then filtered by the inertial sensors Kalman filter. From there, the filtered angular velocity is integrated and combined with filtered star tracker rotations to generate an estimate of the current platform attitude quaternion.

Because of the inherent processing delay in the star tracker system, the acquisition and filtering process for the star tracker system is more complex than that for the inertial sensors, and warrants in-depth discussion. To facilitate an explanation of the complex star tracker data sharing process, the Star Tracker Scheduling & Acquisition subsystem is included here.

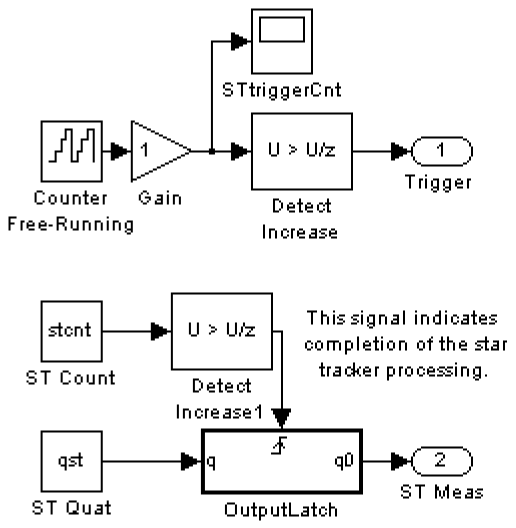


**Figure 18:** Star Tracker Scheduling & Acquisition subsystem

In order to allow Matlab to process star tracker images without interrupting the model, the image processing routine is performed in the Matlab base workspace. The counter in the Star Tracker Scheduling & Acquisition block increases once every ten seconds. This value is monitored within the star tracker processing loop, which captures a new image on the rising edge of the counter. The star tracker loop immediately begins processing the new image to resolve the attitude quaternion of the platform at the time when the image was captured. After completing the image processing routine, the star tracker loop pushes the resolved quaternion into the ST Quat storage block and raises the value of the ST Count counter to signal to the model that the quaternion has been updated. At the time of the image capture, the model also uses the star tracker trigger signal to trigger a storage block in the Attitude Determination system to latch onto the values of the attitude estimation and deviation quaternion during that iteration. These stored values will be used by the Star Tracker Kalman Filter after the image has been processed. Although this process seems convoluted, it has been found to be the simplest and most effective method for multiprocessing within Matlab.

### *6.3.3. Closing the Loop*

Referring back to Figures 14 and 16, the reader may notice that the observed state is utilized by the real-time attitude control system in the exact same way as does the numerical simulation system, at least until the execution of reaction wheel commands. An FSFB controller determines state error by comparing the commanded and observed states, and applies gains to the error vector to compute a new net torque command. The Torque Distribution block then solves for an optimal set of reaction wheel rates. Finally, the wheel rate commands are issued to the real reaction wheels using a Command Wheels block from the SpaceSim Library.

### *6.3.4. Attitude Control Demonstration*

The optimal full-state feedback control system was demonstrated first numerically, and later on the platform. Two mission plans, one smooth and one step function, were designed to demonstrate the system's responses to each of these command trajectory types. The smooth mission consists of a 30 second smooth yaw turn to 60° and back to home. The step function mission plan jumps instantaneously from the home quaternion to a 30° yaw turn. Figure 19 illustrates the responses of the numerical simulations to the aforementioned mission plans.

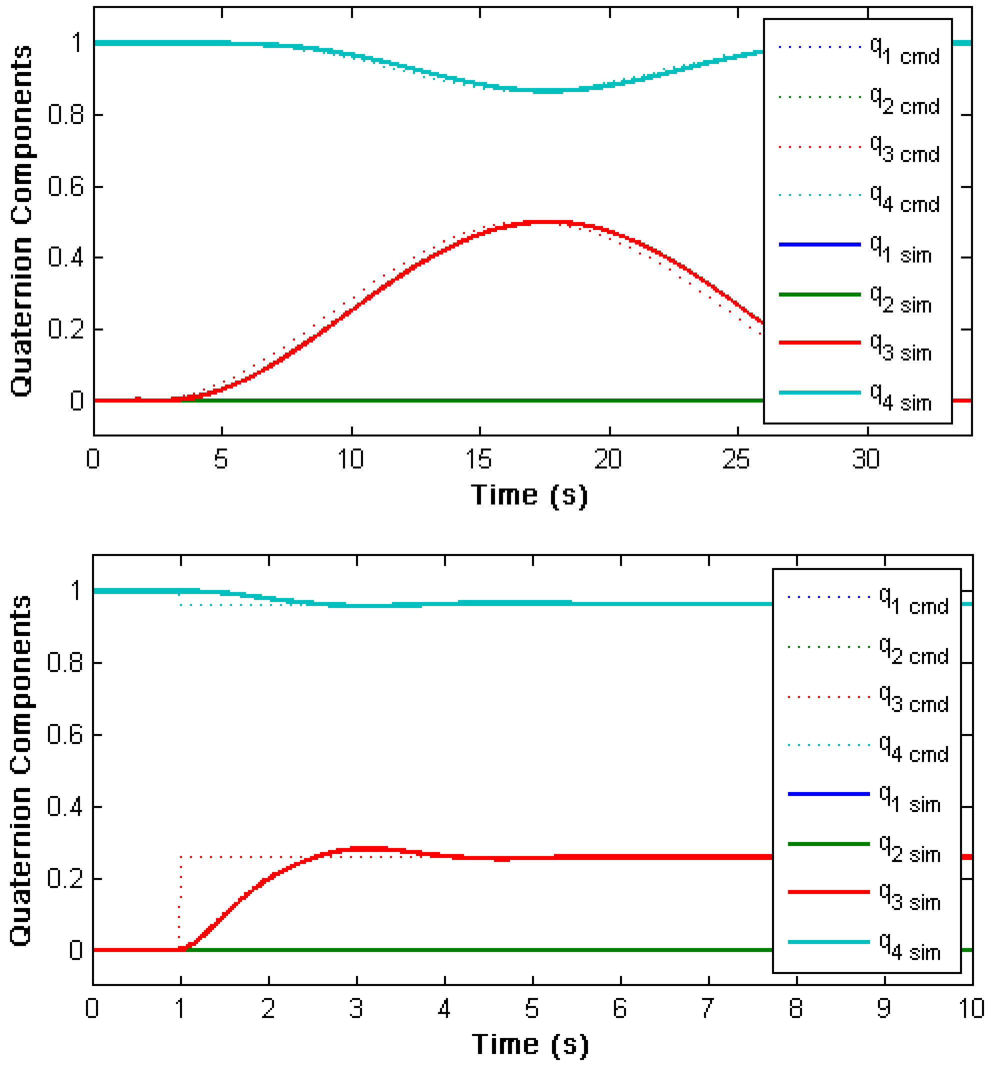
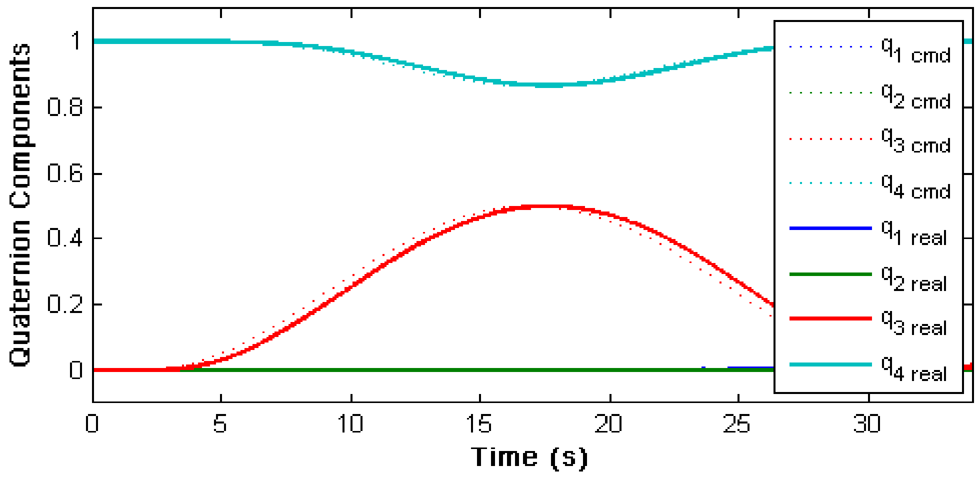


**Figure 19:** Simulation responses to smooth (above) and step (below) commands

As expected, the under-damped numerical model tracked the smooth trajectory closely, with slight phase lag, and slightly overshot and converged with the step command trajectory.

Next, the numerical simulation model was adapted for real-time control by simply swapping the Platform Model block with the Platform Interface block, and transferred to the platform over the wireless network. The platform was then balanced by manually adjusting the course balance system counterweights, and each of the diagnostic tests was run to ensure the system was ready for closed-loop attitude control. The following figures illustrate the system's physical responses according to the control model's attitude determination system.



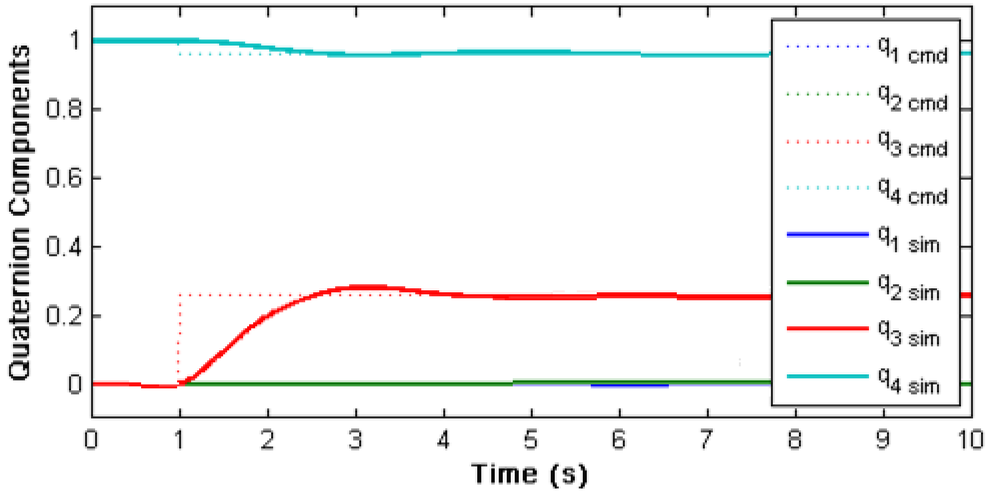


**Figure 20:** Physical responses to smooth (above) and step (below) commands

According to the attitude data logged by the attitude determination system, the platform's physical responses appear to have precisely mirrored those of the numerical simulation. In actuality, however, the platform was observed to have deviated from its steady state target by 5-10° around each body axis by the end of each mission. This discrepancy between the reported and actual accuracies is due to the high drift rates of the MEMS gyros and the fact that the system at the time had no dead-reckoning (star tracker) capability. A formal evaluation of the system's attitude tracking performance would require some external and independent tracking system. Despite the steady state error caused by gyro drift, the similarities in the logged simulation and physical trajectories attest to the accuracy of the numerical plant dynamics model.

# 7. Conclusions

Most, although not all, of the original goals of this project have been met. Numerous improvements were made to bring the system to a state of controllability and to make the simulator more accessible, useable, and demonstrable to future students. The onboard power system was redesigned for longer performance and higher reliability. A computer was built and installed on the platform to perform data acquisition, control, and user interface functions. Software tools were developed to handle hardware interfaces in Matlab and to simplify the creation of future control models. A wireless user interface and data transfer system was established and used to display key telemetry data to the user in real time. Closed-loop controllers were created in Simulink to precisely command the reaction wheel spin rates, although the wheels currently may only operate in one direction because of faulty spin direction determination circuits. Three MEMS gyroscopes were mounted on the platform and interfaced with the flight computer, and were the only sensors available on the platform during the attitude control demonstrations. Although neither the LN-200 inertial measurement nor the star tracker system was installed, arrangements were made in an attitude determination module to accommodate the sensors once they become available.

A baseline demonstration of closed-loop attitude control was conducted for the first time in the history of the project. The control model used in the demonstration relied only on MEMS gyros, although the model is capable of incorporating measurements from the IMU or the star tracker. Although no external or independent tracking system was used to quantify the true attitude error, the platform was visually observed to drift by roughly 5-10° relative to inertial space after typical thirty second attitude control missions. This relatively high drift rate was very close to the specification drift rate of the MEMS gyros, and is expected to vastly improve after installation of the IMU. The star tracker is also expected to improve attitude tracking by limiting the accumulation of drift from the inertial sensors. Despite the considerable steady-state attitude error, the MEMS gyro-based demonstration mission confirmed the accuracy of the numerical model and validated the SpaceSim Toolbox's PLATFORM SIMULATION block.

With current capabilities, mission durations are limited to 1-2 minutes. The two limiting factors are drift in the attitude tracking system and reaction wheel saturation. Although installation of the IMU and the star tracker will eliminate tracking drift as a limiting factor, reaction wheel saturation may continue to be problematic. Reaction wheel saturation in this case is caused primarily by gravitational torque, which results when the platform's center of mass is offset from the center of rotation. Using system identification with IMU data will allow this offset to be determined, and the fine balance system will be used to correct it. However, informal experimentation seemed to indicate that the system is highly sensitive to shifting masses, despite efforts to securely fasten all platform components. Mission durations may continue to be limited to only a few minutes, even after improving center of mass determination with the IMU.

## 8. Recommendations

Although the minimum components for attitude control are in place, there exists plenty of potential for further improvement. First and foremost, the star tracker system and IMU will need to be installed on the platform and incorporated into the attitude control model. The system identification procedure should be used with the IMU to generate a more accurate estimation of the platform's inertia tensor.

The attitude control scheme demonstrated in this project did not perform coordinate transformations between the local vertical local horizontal (LVLH) and a truly inertial reference frame. Mission durations were expected to be less than two minutes, and the rotation of the LVLH frame during this time was deemed small enough to neglect. However, as system improvements make longer duration missions possible, coordinate transformations may be required somewhere in the system, either within the mission definition and telemetry display processes, or within the controller model. Since attitude relative to the LVLH frame is easier to imagine, future users may prefer to continue to define mission plans relative to the LVLH frame, and incorporate a pre-processing routine to transform the plan into an inertial frame prior to running a mission. The control model in this case would not require internal coordinate transformations; the quaternion attitude estimation maintained by the controller will be relative to the inertial frame. The telemetry display program would then need to transform the inertially-defined quaternion estimations into the LVLH frame to make the display more understandable to human observers.

To formally evaluate future control models, the project will require some method for independently and externally tracking the attitude of the platform relative to the LVLH frame. One possibility for such a system might use laser or infrared ranging devices underneath the platform to track the platform's rotation and tilt relative to the vertical axis. Alternatively, infrared LEDs could be mounted in various locations on the platform and tracked by a camera installed inside the star dome. This second method is commonly used by modern computer games to track the rotational and translational motion of a user's head. Such systems are capable of much higher return rates than is expected by the star tracker system, and may prove to be ideal for this project.

The 50 Hz iteration rate of the current attitude control system is sufficient for attitude control, but faster rates may improve attitude tracking capability of future controllers. This iteration rate is constrained by the processing speed of the PC/104 computer, but may be improved by streamlining a control model's sub-processes. Simulink's built-in Profiler tool may help to identify potential areas of improvement.

Processor load can be reduced further by externalizing the reaction wheel control system. Each reaction wheel's driver carrier board requires an analog speed command signal and a pair of digital logic signals to toggle spin direction and acceleration modes. The values of these signals are currently determined within the attitude control model. Future versions of the system should allow the control model to assign digital output values directly proportional to the desired wheel speeds. Dedicated circuits should be designed to handle the absolute value function and determine the logic values for the mode pins according to the sign and time derivative of the speed signal. This circuit should also externalize the reaction wheel PID controllers, and should incorporate safety measures to protect the driver carrier boards. The reaction wheel interface circuits designed and built by Matt Downs should be redesigned by electrical engineering students to include these suggestions and other reliability improvements, and to reduce the footprint of the circuitry.

The CalPoly spacecraft attitude dynamics simulator might be unique in its eventual use of a star tracker system. Future project members might also consider installing other sensors, such as magnetometers, horizon sensors, accelerometers, and Global Positioning Receivers, as additional inputs to the attitude determination system. Each of these sensor types is in use both on satellites and on other spacecraft simulators.

Lastly, an emergency abort system should be created. A routine was written by Downs to return the reaction wheel speed command signals to zero and to set the driver carrier boards in coast mode, and a shortcut to this routine was placed in the Matlab command window on the flight computer. However, if the reaction wheels ever become uncontrollable and this cutoff shortcut fails, then the only way for the user to stop the reaction wheels will be to flip the reaction wheel bus switch while the platform is spinning uncontrollably. This situation might occur, for example, if the flight computer's operating system freezes during a mission. An emergency abort system might consist of a power relay in the reaction wheel power bus, which can be toggled by a radio, a laser, or an infrared signaling system. Miniature 2.4 GHz transceivers and infrared detection systems can be purchased from www.sparkfun.com, and relays and momentary switches can be acquired from RadioShack or from www.digikey.com.

## List of References

[1] Hall, C., Peck, M., and Schwartz, J., "Historical Review of Spacecraft Simulators," AIAA Journal of Guidance, Control, and Dynamics, vol. 26, n. 4, July-August 2003, pp 513-522.

[2] Mittlesteadt, C., "Results on the Development of a Four-Wheel Pyramidal Reaction Wheel Platform," Master's Thesis, Aerospace Engineering Dept., California Polytechnic State Univ., San Luis Obispo, CA, 2006.

[3] Saile, C., "Center of Mass Fine Tuning System for the Cal Poly Spacecraft Simulator," Senior Project, Aerospace Engineering Dept., California Polytechnic State Univ., San Luis Obispo, CA, 2007.

[4] Logan, J., "Control and Sensor Development on a Four-Wheel Pyramidal Reaction Wheel Platform," Master's Thesis, Aerospace Engineering Dept., California Polytechnic State Univ., San Luis Obispo, CA, 2008.

[5] *LN-200 Inertial Measurement Unit* manual, Rev. AJ, Litton Systems Inc., Guidance & Control Systems Dept., Woodland Hills, CA, 2005.

[6] Iversen, P., "LN-200," Senior Project, Aerospace Engineering Dept., California Polytechnic State Univ., San Luis Obispo, CA, (2005).

[7] Rollins, C., and Gallegos, E., "Integration of an LN-200 Inertial Measuremet Unit on the CalPoly Spacecraft Attitude Dynamic Simulator," Senior Project, Aerospace Engineering Dept., California Polytechnic State Univ., San Luis Obispo, CA, 2008.

[8] McClurg, T., "Development and Characterization of a Star Tracker System for Satellite Attitude Determination in Simulation," Master's Thesis, Aerospace Engineering Dept., California Polytechnic State Univ., San Luis Obispo, CA, 2007.

[9] Healy, P., "Mass Property System Identification of a Spacecraft Simulator," Master's Thesis, Aerospace Engineering Dept., California Polytechnic State Univ., San Luis Obispo, CA, 2006.

[10] Downs, M., "Adaptive Control Applied to the Cal Poly Spacecraft Attitude Dynamics Simulator," Master's Thesis, Aerospace Engineering Dept., California Polytechnic State Univ., San Luis Obispo, CA, 2009.

[11] Wie, B., *Space Vehicle Dynamics and Control*, AIAA, Reston, VA, 1998.

[12] Bishop, G., and Welch, G., "An Introduction to the Kalman Filter," Dept. of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC, 2006. <http://www.cs.unc.edu/~welch/kalman/kalmanIntro.html>

## Appendix A. Attitude Control Demonstration Procedure

1) **Charge the batteries.** Switch off the AC power strip to which the three chargers are plugged in. Connect the power umbilical cable to the matching Molex connector on the platform. Switch the power strip on. Observe the LEDs on the green Smart Chargers; green on both chargers indicates the equipment bus batteries are fully charged.

2) **Boot the flight computer.** Toggle the equipment bus power switch (labeled "Comp") on the front panel to power the computer. The computer has its own push-button switch; if the computer does not boot when the equipment bus is turned on, then toggle the red button by the computer. When the computer has finished booting, it will make whistling sounds, like R2-D2 from Star Wars.

3) **Logon to the flight computer.** On the desktop PC or on a laptop, join the SpaceSim wireless network, if it wasn't done automatically. Both the PC and the laptop have shortcuts on their desktops to start a Remote Desktop session. If the shortcut is not present, open Remote Desktop from the Start menu and connect to the following IP address: 192.168.0.100. Upon successful connection, the user is prompted for credentials. Logon as user "Aero", and leave the password field blank.

4) **Prepare telemetry display**. On the Desktop PC, start Matlab. Press the PlatformDisplay shortcut button near the top of the Matlab window. The program is now waiting to receive data via a UDP link over the wireless network. After starting a control model or test utility which has been set up for transmission, the data will stream automatically and the figure will be updated at roughly 4 frames per second.

5) **Run test utilities (optional).** These tests should be run if significant changes have been made to the system or if the system has not been operated in a long time. Start Matlab on

the flight computer. Navigate to C:\Spacesim_software\Tests. Run the tests_setup.m script. Turn on the reaction wheel power bus by toggling the switch labeld “RWs” on the platform front panel. Open the desired test model and run. Recommended tests: WheelControlTest.mdl, GyrosReadTest.mdl, MissionTest.mdl. During the reaction wheel test, be ready to remove power to the reaction wheel circuit.

6) **Standby the platform.** Check the voltages on the batteries by reading the Battery Charges window on the flight computer. The first two values should read at least 12.5 V, and the third and fourth values should at least 17 V; if so, then turn off the power strip and disconnect the power umbilical cable. Turn on the air compressor by pulling the red knob. When the platform starts to float, manually balance the platform by adjusting the course counterweights. Fine adjustments can be applied by tapping the lead acid batteries and other equipment, or by running moveFineBalanceMasses.m from the System ID folder. Once balanced, use a chair or some other rigid object to hold the platform steady.

7) **Run the control model.** Navigate to C:\SpaceSim_Software\FullStateFeedback, open the fsfb.mdl model, and run the fsfb_setup.m script. After the gyros calibrate, you will be asked to select a mission plan. Enter the number corresponding to “SmoothYawTurn.m”, or another mission file. Remove the chair and hold the platform steady by hand. Start the control model and continue to hold as the reaction wheels accelerate to their bias speeds. When you hear two beeps, immediately let go but be careful not to impart any rotational velocity as you do so. As always, be ready to stop the simulation and press the StopMotors shortcut in the Matlab window if the platform or the wheels do not seem to behave as expected. During the simulation, you may minimize the Remote Desktop window to observe the telemetry display on the desktop PC.

## Appendix B.   C Code

### *B.1.  Analog Input Mex Function eagleAin.c*

```
/**********************************************************************
**
 * Function eagleAin()
 *      Ryan Kinnett, Apr 31, 2009.
 *
 * Syntax:
 *  data = eagleAin( SerialNumber, ChannelList)

**********************************************************************
*
 */


 /*
To Compile:
mex -setup
  ...select any compiler other than the default (lcc)
mex 'C:\SpaceSim_Software\dev\daq\mex\eagleDout4.c' -outdir
'C:\SpaceSim_Software\dev\daq\mex\' -
IC:\Progra~1\EagleTech\EDRE\include -LC:\Progra~1\EagleTech\EDRE\lib -
lEDRAPI
    (make sure source file name and all directories are accurate)
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "C:\Program Files\MATLAB\R2008a\extern\include\mex.h"
#include "C:\Program Files\EagleTech\EDRE\Include\EDRAPI.H"

void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray
*prhs[])
{
    unsigned long Sn = 1000019524;
    unsigned long nchan;
    //unsigned long ADNUMCHAN;
    char *StrError[128];
    unsigned long err;
    double *errout;
    long portvals[3];
    mwIndex n;
    double *value, *outvals;

    //SYNTAX CHECK:
    //check input size:
    if (nrhs!=1)
    {
        mexPrintf("ERROR:  Usage:  err = eagleDout4(Value_24bit)
%i\n",nrhs);
        return;
    }
    //check input value:
    value   = mxGetPr(prhs[0]);
    if (*value>(double)16777216 || *value<(double)0)
//(2^24=16777216)
    {
        mexPrintf("ERROR:  Invalid input value.  (0 < value <
2^24)\n");
        return;
    }

    plhs[0] = mxCreateDoubleScalar(mxREAL);
    errout  = mxGetPr(plhs[0]);
    *errout = 0;


    for(n=2;n>=0;n--)
    {
        portvals[n]=floor((double)*value/(double)pow(2,n*8));
        *value=*value-portvals[n]*pow(2,n*8);

        err=EDRE_DioWrite (Sn, (unsigned long)n, (unsigned
long)portvals[n]);


        if(err<0 && nlhs==1)
        {
            EDRE_StrError(err, StrError);
            mexPrintf("ERROR:  %s\n",StrError);
            *errout=err;
            return;
        }

    }

////////////////////////////////////////////////////////////////////////
}
```

### B.2. Analog Output Mex Function eagleAout.c

```
/**********************************************************************
**
 * Function eagleAout()
 *      Ryan Kinnett, Apr 31, 2009.
 *
 * Syntax:
 *  data = eagleAout( SerialNumber, ChannelList, ValuesList)

***********************************************************************
*
 */
 /*
To Compile:
mex -setup
  ...select any compiler other than the default (lcc)
mex 'C:\SpaceSim_Software\dev\daq\mex\eagleAout.c' -outdir
C:\SpaceSim_Software\DAQ\MEX\ -IC:\Progra~1\EagleTech\EDRE\include -
LC:\Progra~1\EagleTech\EDRE\lib -lEDRAPI
    (make sure source file name and all directories are accurate)
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "C:\Program Files\MATLAB\R2008a\extern\include\mex.h"
#include "C:\Program Files\EagleTech\EDRE\Include\EDRAPI.H"

void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray
*prhs[])
{
    long nchan;
    double *data;
    char *StrError[128];
    unsigned long err;
    double *errout;
    long val;
    unsigned long Sn;
    double *output;
    mwIndex n;
    mwSize nchans;
    mwSize nvals;
    double *chans;
    double *vals;

    //SYNTAX CHECK:
    if (nrhs!=3)
    {
        mexPrintf("ERROR:  Usage:  err = eagleAout(SerialNumber,
Channels, Values)\n");
        return;
    }
    Sn = (long) mxGetScalar(prhs[0]);
    nchan = EDRE_Query(Sn,ADNUMCHAN,0);
    if (nchan<1)
    {
        mexPrintf("ERROR:  Board not recognized\n");
        return;
    }


    //CHECK CHANNELS LIST HERE...
    chans   = mxGetPr(prhs[1]);
    vals    = mxGetPr(prhs[2]);
    nchans  = mxGetNumberOfElements(prhs[1]);
    nvals   = mxGetNumberOfElements(prhs[2]);
    if(nchans!=nvals)
```

```
            return;
        for(n=0;n<nchans;n++)
        {
            if(chans[n]>3 || chans[n]<0)
            {
                mexPrintf("Error:  Channel %i not valid\n");
                if (nlhs==1)
                    *errout=-1;
                return;
            }
            if(vals[n]>10)
                vals[n]=10.0;
            if(vals[n]<-10)
                vals[n]=-10.0;
        }

        //FOR EACH CHANNEL, get data and store to output data array:
        plhs[0] = mxCreateDoubleMatrix(1,nchans,mxREAL);
        if(nlhs==1)
        {
            errout  = mxGetPr(plhs[0]);
            *errout = 0;
        }
        for(n=0;n<nchans;n++)
        {
            val=0;
            err=EDRE_DAWrite (Sn, (int)chans[n], (long)(vals[n]*1000000));
            if(err<0 && nlhs==1)
                *errout=err;
        }


////////////////////////////////////////////////////////////////////////
}
```

### B.3. Digital Input Mex Function eagleDin.c

```
/**********************************************************************
**
 * Function eagleAin()
 *      Ryan Kinnett, Apr 31, 2009.
 *
 * Syntax:
 *  data = eagleAin( SerialNumber, ChannelList)

**********************************************************************
*
 */

 /*
To Compile:
mex -setup
```

```
  ...select any compiler other than the default (lcc)
mex 'C:\SpaceSim_Software\dev\daq\mex\eagleDin.c' -
IC:\Progra~1\EagleTech\EDRE\include -LC:\Progra~1\EagleTech\EDRE\lib -
lEDRAPI
    (make sure source file name and all directories are accurate)
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "C:\Program Files\MATLAB\R2008a\extern\include\mex.h"
#include "C:\Program Files\EagleTech\EDRE\Include\EDRAPI.H"

void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray
*prhs[])
{
    long nchan;
    double *data;
    char *StrError[128];
    unsigned long err;
    long val;
    unsigned long Sn;
    double *output;
    mwIndex n;
    mwSize nchans;
    double *chans;
    unsigned int port, pin, binval;

    //SYNTAX CHECK:

    if (nrhs!=2)
    {
        mexPrintf("ERROR:  Check syntax.  Need device serial number and
array of desired channels\n");
        return;
    }
    Sn = (long) mxGetScalar(prhs[0]);
    nchan = EDRE_Query(Sn,ADNUMCHAN,0);
    if (nchan<1)
    {
        mexPrintf("ERROR:  Board not recognized\n");
        return;
    }


    //CHECK CHANNELS LIST...
    chans   = mxGetPr(prhs[1]);
    nchans  = mxGetNumberOfElements(prhs[1]);
    for(n=0;n<nchans;n++)8
    {
        if(chans[n]>23 || chans[n]<0)
        {
            mexPrintf("ERROR: Invalid channel array\n");
            return;
        }
    }

    //FOR EACH CHANNEL, get single sample and store to output data
array:
    plhs[0] = mxCreateDoubleMatrix(1,nchans,mxREAL);
    data    = mxGetPr(plhs[0]);
    for(n=0;n<nchans;n++)
    {
        port    = (unsigned int) floor(chans[n]/8);
        pin     = chans[n]-port*8;
        val     = 0;
        err     = EDRE_DioRead (Sn, port, &val);
        if(err<0)
        {
            EDRE_StrError(err, StrError);
            mexPrintf("ERROR:  %s\n",StrError);
            data[n]=err;
        }
        else
        {
            binval  = (unsigned
int)(floorf((float)val/(float)pow(2.0,pin))) % 2;
            data[n] = (double) binval;
            //mexPrintf("Channel: %u, Port: %u, Pin: %u, PinValue: %u,
PortValue: %u\n",(unsigned int)chans[n],port,pin,binval,val);
        }
        dana[n]=(double)val;
    }

///////////////////////////////////////////////////////////////////////
}
```

### B.4. Digital Output Mex Function eagleDout.c

```
/*********************************************************************
**
 * Function eagleAin()
 *      Ryan Kinnett, Apr 31, 2009.
 *
 * Syntax:
 *  data = eagleAin( SerialNumber, ChannelList)

**********************************************************************
*
 */

 /*
To Compile:
mex -setup
  ...select any compiler other than the default (lcc)
mex 'C:\SpaceSim_Software\dev\daq\mex\eagleDout4.c' -outdir
'C:\SpaceSim_Software\dev\daq\mex\' -
```

```
IC:\Progra~1\EagleTech\EDRE\include -LC:\Progra~1\EagleTech\EDRE\lib -
lEDRAPI
    (make sure source file name and all directories are accurate)
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "C:\Program Files\MATLAB\R2008a\extern\include\mex.h"
#include "C:\Program Files\EagleTech\EDRE\Include\EDRAPI.H"

void mexFunction(int nlhs,mxArray *plhs[],int nrhs,const mxArray
*prhs[])
{
    unsigned long Sn = 1000019524;
    unsigned long nchan;
    //unsigned long ADNUMCHAN;
    char *StrError[128];
    unsigned long err;
    double *errout;
    long portvals[3];
    mwIndex n;
    double *value, *outvals;

    //SYNTAX CHECK:
    //check input size:
    if (nrhs!=1)
    {
        mexPrintf("ERROR:  Usage:  err = eagleDout4(Value_24bit)
%i\n",nrhs);
        return;
    }
    //check input value:
    value   = mxGetPr(prhs[0]);
    if (*value>(double)16777216 || *value<(double)0)
//(2^24=16777216)
    {
        mexPrintf("ERROR:  Invalid input value.  (0 < value <
2^24)\n");
        return;
    }

    plhs[0] = mxCreateDoubleScalar(mxREAL);
    errout  = mxGetPr(plhs[0]);
    *errout = 0;



    for(n=2;n>=0;n--)
    {
        portvals[n]=floor((double)*value/(double)pow(2,n*8));
        *value=*value-portvals[n]*pow(2,n*8);

        err=EDRE_DioWrite (Sn, (unsigned long)n, (unsigned
long)portvals[n]);
```

```
        if(err<0 && nlhs==1)
        {
            EDRE_StrError(err, StrError);
            mexPrintf("ERROR:  %s\n",StrError);
            *errout=err;
            return;
        }

    }

//////////////////////////////////////////////////////////////////////
}
```

### B.5. Battery Status Monitor Utility BatteryStatus.c

```
//BATTERY STATUS DISPLAY
//RYAN KINNETT, 2009

#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include "C:\Program Files\EagleTech\EDRE\Include\EDRAPI.H"

// gcc -I C:\Progra~1\EagleTech\EDRE\include -L
C:\Progra~1\EagleTech\EDRE\lib -c BatteryStatus.c -o BatteryStatus.exe

int main(void)
{
    unsigned long Sn        = 1000019524;
    int pins[]              = {11, 12, 13, 14};
    float scalefactors[]    = {5.006, 4.968, 4.972, 4.920};
    double Vmin[]           = {11.0, 11.0, 16.0, 16.0};
    double V[4];
    int okay                = 1;
    unsigned long err;
    long val;
    int n, i, j;

    printf("%i \n",15.1>16.1);

    system("color 2E");
    while(1)
    {
        okay=1;
        printf("\n");
        for(n=0;n<4;n++)
        {
            val=0;
            err=EDRE_ADSingle (Sn, pins[n], 2, .25, &val);
            V[n]=(double) val / (double)1000000*scalefactors[n];
        }
        V[1]=V[1]-V[0];
        V[3]=V[3]-V[2];
        for(n=0;n<4;n++)
            okay=okay*(V[n]>(Vmin[n]+okay*0.1) || V[n]<1);

        printf("\n%2.2f  %2.2f  %2.2f  %2.2f",V[0],V[1],V[2],V[3]);
        if(!okay)
        {
            system("color 4C");
            for(i=0;i<60;i++)
                Beep(20*i+600,10);

        }
        else
        {
            system("color 2E");
        }
        Sleep(10000);
    }
    return 0;
}
```

## B.5. R2-D2 Sound r2d2.c

```
//R2-D2 SOUND THROUGH PC SPEAKER
//RYAN KINNETT, 2009

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "windows.h"

/* TO COMPILE:   (@command prompt)
gcc r2d2_1.c -o r2d2.exe
*/

// PROTOTYPES:
int wait(int milliseconds);

// MAIN:
int main(int argc, char* argv[])
{
    //INITIAL DELAY (give time for other startup procs to finish
loading)
    //Sleep(10000);

    int n, i;

    // INITIAL SWEEPING WHISTLE:
    int basefreqs1[5]   = {2200,2200,3200,  3800, 3800};
    int slopes1[5]      = {-100, 100,  50,  0,    -80};
    int durs1[5]        = {50,  100,  100,  150,  250};
    int pauses1[5]      = {150,  0,      0,    0,    0};
    for(n=0; n<5; n++)
    {
        for(i=0; i<durs1[n]/10; i++)
        {
            BOOL RC =
Beep((DWORD)basefreqs1[n]+slopes1[n]*i,(DWORD)10);
        }
        if(n==0) wait(pauses1[n]);
        //Sleep(durs1[n]);
    }

    wait(400);

    // RANDOM BEEPS:
    int basefreqs[11]   =
{1900,2700,1950,1560,1800,2100,1740,1740,2300,2400,2720};
    int slopes[11]      = {100,0,-150,0,40,-80,0,50,20,80,0};
    int durs[11]        = {30,40,30,30,40,40,20,20,30,40,30};
    int pauses[11]      = {50,40,50,50,40,40,0,50,40,0};
    for(n=0; n<11; n++)
    {
        for(i=0; i<durs[n]/10; i++)
        {
            BOOL RC = Beep((DWORD)basefreqs[n]+slopes[n]*i,(DWORD)10);
        }
        Sleep(durs[n]);
    }
}

// CUSTOM WAIT FUNCTION: (more accurate than Sleep function)
int wait(int milliseconds)
{
    clock_t t0;                             //initialize Begin and End
for the timer
    int elapTicks;
    double elapMilli = 0;
    t0 = clock() * CLK_TCK;                 //start the timer

    while( elapMilli < milliseconds )
        elapMilli = (clock() * CLK_TCK- t0)/1000;  //milliseconds from
Begin to End
    return 0;
}
```

# Appendix C:  Matlab Code

## C.1.  Test Setup test_setup.m

```
%tests_setup.m
%RYAN KINNET, 2009
%generates all values needed by various test models

clear; clc;

%% SENSOR SETUP

dt      = 0.02;     %time (s) per iteration

load 'MotorSetup.mat';
load 'rwcal.mat';

%WHEEL CONTROLLER GAINS:
Kp      = 0.05;
Ki      = 0.0001;
Kd      = 0.002;

biasspeed = 2500*2*pi/60;

%CALIBRATE GYROS AND REACTION WHEELS:
for n=1:3, mexBeep( 2000, 50); pause(0.05); end;
disp('Calibrating Gyros.  Keep er steady');
sim('GyroCal');
disp('...done.');

disp('Calibrating reaction wheels.');
sim('RwOffsetCal');
disp('...done.');
mexBeep( 2000, 50);


%% FSFB

%FSFB GAINS:
I = [1.5,.0045,-.055;.0045,1.65,.015;-.055,.015,1];
Idiag = [I(1,1),0,0;0,I(2,2),0;0,0,I(3,3)];
ts      = 20;
damping = .75;
pcntsettle  = 0.01;
wn      = -log(pcntsettle)/ts/damping;
Kfsfb   = 2*wn^2*Idiag;
Cfsfb   = 2*damping*wn*Idiag;
Iw      = .0005355;


Beta    = 28.3*pi/180;  % Wheel Inclination Angle, rad
betas   = [cos(Beta),sin(Beta)];    % Sine and cosine of beta
```

```
Rws      = [cos(Beta),0,-cos(Beta),0;0,cos(Beta),0,-
cos(Beta);sin(Beta),sin(Beta),sin(Beta),sin(Beta)];  % Wheel Frame to
Body Frame
BTorque2WTorque = .5*[1,0,.5;0,1,.5;-1,0,.5;0,-1,.5];   % Torque
Distributi
```

### C.2. Mission Plan Template missionplan_template.m

```
function out = missionplan3(t0)
initdelay   = 5;    % time (s) to hold or acquire the pre-mission
orientation
                    % note: this delay might not be necessary, but it's
a
                    % good idea to let the sensors warm up and filters
                    % initialize.
duration    = 60;   % time (s) in the primary mission sequence


stopflag    = 0;
qcmd        = [0 0 0 1];
if t0<initdelay
    %command an initial orientation here
    %leave blank for stay at home

elseif t0<initdelay+duration
    %t is time inot the primary mission sequence
    t=t0-initdelay;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
    %PRIMARY MISSION SEQUENCE HERE:

    %EXAMPLES:
    %ex1: flying a specific trajectory:
    %{
    yaw     = pi/2      *sin(2*pi/60 *t);
    pitch   = 3/180*pi  *sin(2*pi/30 *t);
    roll    = 0;
    Q       = R3(yaw)*R2(pitch)*R1(roll);
    qcmd    = dcm2quat(Q);
    qcmd    = [qcmd(2:end),q(1)]; %since matlab orders its quaternions
differently
    %}

    %ex2: flying waypoints:
    %{
    if      t<15,   qcmd=[0     0       sin((pi/2)/2)
cos((pi/2)/2)];
    elseif  t<30,   qcmd=[0     0       sin((pi)/2)
cos((pi)/2)];
    elseif  t<45,   qcmd=[0     0       sin((3*pi/2)/2)
cos((3*pi/2)/2)];
```

```
    else            qcmd=[0     0       sin((2*pi)/2)
cos((2*pi)/2)];
    end;
    %}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
else
    stopflag=1;
end;

out = [qcmd,stopflag];

end


%ROTATION MATRIX GENERATION:  (angle in rads)
function [R1_out] = R1(angle)
    R1_out = [1,  0,  0;
              0,  cos(angle), sin(angle);
              0, -sin(angle), cos(angle)];
end
function [R2_out] = R2(angle)
    R2_out = [cos(angle),  0,  -sin(angle);
              0,           1,             0;
              sin(angle),  0,   cos(angle)];
end
function [R3_out] = R3(angle)
    R3_out = [cos(angle), sin(angle), 0;
             -sin(angle), cos(angle), 0;
              0           0            1];
end
```

## C.3. *Fine Balance Mass Move Function moveFineBalanceMasses.m*

```
function moveFineBalanceMasses(ds)
% moveFineBalanceMasses(ds) moves the Nth fine balance mass
% ds(N) cm from its current position.  ds is a 3-element array.
%
% Fine balance masses are numbered 0-2, with positive displacement
% directions as follows:
% mass0: +ds = -x,+y    (middle deck)
% mass1: +ds = +x,+y    (top deck)
% mass2: +ds = -z       (vertical)
%
serialport = 'COM7';

%CHECK IF COM7 ALREADY OPEN ELSEWHERE:
porthandles = instrfind('Type','serial');
if length(porthandles)
    for n=1:length(porthandles)
```

```
        if strfind(porthandles(n).Name,serialport)
            if strcmp(porthandles(n).Status,'open')
                error(sprintf('ERROR:  %s already in use elsewhere in
Matlab',serialport));
                return;
            end;
        end;
    end;
end;
s1  = serial(serialport,'Baudrate',9600);
fopen(s1);
set(s1,'Terminator','CR');

cmd = ds*2460000;
for n=0:2,
    fprintf(s1,sprintf('%iEN\n',n));
    pause(0.02);
    fprintf(s1,sprintf('%iLR%i\n',n,cmd(n+1)));
    pause(0.02);
    fprintf(s1,sprintf('%iM\n',n,cmd(n+1)));
end;

fclose(s1);
delete(s1);
```

### *C.4. Fine Balance Mass Adjustment GUI MassMoveGUI.m*

```
function varargout = MassMoveGUI(varargin)
% MASSMOVEGUI M-file for MassMoveGUI.fig
%      MASSMOVEGUI, by itself, creates a new MASSMOVEGUI or raises the
existing
%      singleton*.
%
%      H = MASSMOVEGUI returns the handle to a new MASSMOVEGUI or the
handle to
%      the existing singleton*.
%
%
% See also: GUIDE, GUIDATA, GUIHANDLES


% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @MassMoveGUI_OpeningFcn, ...
                   'gui_OutputFcn',  @MassMoveGUI_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end
```

```
if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT



% --- Executes just before MassMoveGUI is made visible.
function MassMoveGUI_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.

% Choose default command line output for MassMoveGUI
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);
imgdata=imread('platform2.bmp');
axes(handles.axes1);
img = image(imgdata);
set(gca,'XTick',[],'YTick',[]);
%axexcallback = get(handles.axes1,'ButtonDownFcn')
set(img,'ButtonDownFcn',@imgfcn);



% --- Outputs from this function are returned to the command line.
function varargout = MassMoveGUI_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
varargout{1} = handles.output;




function edit1_Callback(hObject, eventdata, handles)



% --- Executes during object creation, after setting all properties.
function edit1_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function edit2_Callback(hObject, eventdata, handles)

function edit2_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end

function edit3_Callback(hObject, eventdata, handles)
```

```
% --- Executes during object creation, after setting all properties.
function edit3_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on mouse press over axes background.
function axes1_ButtonDownFcn(hObject, eventData, handles)
disp(1);
mouseloc = get(0,'PointerLocation');
set(handles.edit1,'String',num2str(mouseloc(1)));
figloc  = get(handles.figure1,'Position');
set(handles.edit2,'String',num2str(figloc(1)));


function pushbutton1_Callback(hObject, eventdata, handles)

function edit4_Callback(hObject, eventdata, handles)

function edit4_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

### *C.4. Full State Feedback Setup FSFB_setup.m*

```
clear; clc;

%% SENSOR SETUP

dt      = 0.01;     %time (s) per iteration
n       = 4;        %select motor # (1-4)
m       = 2;

load 'MotorSetup.mat';
load 'rwcal.mat';
v2f = volts2freq(n);

cmdpin=Motor(n).cmdpin;
spdpin=Motor(n).spdpin;
dxnpin=Motor(n).dxnpin;
dxnpins=Motor(n).dxnpins;

%WHEEL CONTROLLER GAINS:
```

```
Kp      = 0.03;
Ki      = 0.0001;
Kd      = 0.001;

biasspeed = 2500*2*pi/60;
eagleDout4(1000019524,[Motor(:).dxnpins],[1 0 1 0 1 0 1 0]);    %set
all to ccw accel mode


CALIBRATE GYROS AND REACTION WHEELS:
for n=1:3, mexBeep( 2000, 50); pause(0.05); end;
disp('Calibrating Gyros.  Keep er steady');
sim('GyroCal');
disp('...done.');

disp('Calibrating reaction wheels.');
sim('RwOffsetCal');
disp('...done.');
mexBeep( 2000, 50);


%% FSFB

%FSFB GAINS:
I = [1.5,.0045,-.055;.0045,1.65,.015;-.055,.015,1];
Idiag = [I(1,1),0,0;0,I(2,2),0;0,0,I(3,3)];
ts      = 4;
damping = .6;
pcntsettle  = 0.01;
wn      = -log(pcntsettle)/ts/damping;
Kfsfb   = 2*wn^2*Idiag;
Cfsfb   = 2*damping*wn*Idiag;

Iw      = .0005355; %kg-m^2;
Beta    = 28.3*pi/180;  % Wheel Inclination Angle, rad
betas   = [cos(Beta),sin(Beta)];    % Sine and cosine of beta
Rws     = [ cos(Beta),  0,          -cos(Beta),     0;
            0,          cos(Beta),  0,              -cos(Beta);
            sin(Beta),  sin(Beta),  sin(Beta),      sin(Beta)]; % Wheel
Frame to Body Frame
%       Top-down view of RW ordering:
%             y\      /x
%               (2)(1)
%               (3)(4)
BTorque2WTorque = .5*[1,0,.5;0,1,.5;-1,0,.5;0,-1,.5];   % Torque
Distribution

% availmissions=dir('c:\spacesim_software\mission plans\*.m');
% availmissions = dir('c:\Users\user\Desktop\Thesis\New\*.m');
availmissions = dir([pwd,'\*.m']);
availmissions = {availmissions(:).name};
for n=1:length(availmissions)
    disp(sprintf('%i)\t%s',n,availmissions{n}));
end;
```

```
n=input(['Select mission (1-',num2str(n),'): ']);
missionfunction = str2func(availmissions{n}(1:end-2));
feval(missionfunction,0);

q0 = [0;0;0;1];


%REACTION WHEEL SIM:
% For FSFB Gains
wnrw = 8; % Natural Frequency
zetarw = 1.1; % Damping Ratio

GyroStdDev = 0.01;
GyroMeasCovar = GyroStdDev^2*eye(3);
GyroMeasMatrix = eye(3);


Jval    =  [0.65        0.00045     -0.00055;
            0.00045     0.6         0.0015;
            -0.00055    0.0015      0.56];
Jinv    = inv(Jval);

STper   = 100;
stcnt   = 0;
qst     = [0;0;0;1];
x0      = [0;0;0; 0;0;0;1];

wnrw    = 10; % Natural Frequency
zetarw  = 1.1; % Damping Ratio

Acont   = [0 1; -wnrw^2 -2*zetarw*wnrw];
Bcont   = [0; wnrw^2];
Adisc   = [eye(size(Acont))+Acont*dt];
Bdisc   = (eye(size(Acont))*dt+.5*Acont*dt^2)*Bcont;
C       = [1 0];
D       = [0];

ArwDist = [Rws; 1 -1 1 -1];
invArwDist = inv(ArwDist);

gyroperiod = 0.02;
RwsIw   = Rws*Iw;
Tstall = 1.25; %(N-m) Faulhaber motor stall torque
```

### *C.5. Telemetry Display GUI TelemetryDisplay_V2.m*

```
%% SETUP
%GENERATE PLATFORM GEOMETRY:
%{
%HORIZONTAL BALANCE STRUCTURE:
Face(1).bcf = [ 8 -8 -8 8 8 6 6 -6 -6 6 8;
```

```
                -8 -8 8 8 -8 -6 6 6 -6 -6 -8;
                -5*ones(1,11)];
Face(2).bcf = [ 8 8 8 8 8;
                8 8 -8 -8 8;
                -5 -7 -7 -5 -5];
Face(3).bcf = [ 8 8 -8 -8 8;
                -8 -8 -8 -8 -8;
                -5 -7 -7 -5 -5];
Face(4).bcf = [ -8 -8 -8 -8 -8;
                8 8 -8 -8 8;
                -5 -7 -7 -5 -5];
Face(5).bcf = [ 8 8 -8 -8 8;
                8 8 8 8 8;
                -5 -7 -7 -5 -5];
Face(6).bcf = [ 6 6 6 6 6;
                6 6 -6 -6 6;
                -5 -7 -7 -5 -5];
Face(7).bcf = [ 6 6 -6 -6 6;
                -6 -6 -6 -6 -6;
                -5 -7 -7 -5 -5];
Face(8).bcf = [ -6 -6 -6 -6 -6;
                6 6 -6 -6 6;
                -5 -7 -7 -5 -5];
Face(9).bcf = [ 6 6 -6 -6 6;
                6 6 6 6 6;
                -5 -7 -7 -5 -5];

%MIDDLE DECK:
Face(10).bcf =[ 6 6 3.25 -3.25 -6 -6 -3.25 3.25 6;
                3.25 -3.25 -6 -6 -3.25 3.25 6 6 3.25;
                zeros(1,9)];
Face(11).bcf =[ 6 6 3.25 -3.25 -6 -6 -3.25 3.25 6;
                3.25 -3.25 -6 -6 -3.25 3.25 6 6 3.25;
                0.5*ones(1,9)];
Face(12).bcf =[ 6*ones(1,5);
                3.25*[1 1 -1 -1 1];
                .5*[1,0 0 1 1]];
Face(13).bcf =[ -6*ones(1,5);
                3.25*[1 1 -1 -1 1];
                .5*[1,0 0 1 1]];
Face(14).bcf =[ 3.25*[1 1 -1 -1 1];
                6*ones(1,5);
                .5*[1 0 0 1 1]];
Face(15).bcf =[ 3.25*[1 1 -1 -1 1];
                -6*ones(1,5);
                .5*[1 0 0 1 1]];
Face(16).bcf =[ 6 3.25 3.25 6 6;
                3.25 6 6 3.25 3.25;
                .5*[1 1 0 0 1]];
Face(17).bcf =[ -6 -3.25 -3.25 -6 -6;
                3.25 6 6 3.25 3.25;
                .5*[1 1 0 0 1]];
Face(18).bcf =[ 6 3.25 3.25 6 6;
                -3.25 -6 -6 -3.25 -3.25;
                .5*[1 1 0 0 1]];
Face(19).bcf =[ -6 -3.25 -3.25 -6 -6;
```

```
                -3.25 -6 -6 -3.25 -3.25;
                .5*[1 1 0 0 1]];

%VERTICAL BALANCE STRUCTURES:
Face(20).bcf =[ 6*ones(1,5);
                1 -1 -1 1 1;
                -5 -5 11 11 -5];
Face(21).bcf =[ 6 8 8 6 6;
                ones(1,5);
                -5 -5 11 11 -5];
Face(22).bcf =[ 6 8 8 6 6;
                -ones(1,5);
                -5 -5 11 11 -5];
Face(23).bcf =[ -6*ones(1,5);
                1 -1 -1 1 1;
                -5 -5 11 11 -5];
Face(24).bcf =[ -6 -8 -8 -6 -6;
                ones(1,5);
                -5 -5 11 11 -5];
Face(25).bcf =[ -6 -8 -8 -6 -6;
                -ones(1,5);
                -5 -5 11 11 -5];

%TOP DECK:
Face(26).bcf =[ 6 6 4 -4 -6 -6 -4 4 6;
                2 -2 -4 -4 -2 2 4 4 2;
                5.5*ones(1,9)];
Face(27).bcf =[ 6 6 4 -4 -6 -6 -4 4 6;
                2 -2 -4 -4 -2 2 4 4 2;
                6*ones(1,9)];
Face(28).bcf =[ 6*ones(1,5);
                2*[1 1 -1 -1 1];
                6 5.5 5.5 6 6];
Face(29).bcf =[ -6*ones(1,5);
                2*[1 1 -1 -1 1];
                6 5.5 5.5 6 6];
Face(30).bcf =[ 4*[1 1 -1 -1 1];
                4*ones(1,5);
                6 5.5 5.5 6 6];
Face(31).bcf =[ 4*[1 1 -1 -1 1];
                -4*ones(1,5);
                6 5.5 5.5 6 6];
Face(32).bcf =[ 6 4 4 6 6;
                2 4 4 2 2;
                6 6 5.5 5.5 6];
Face(33).bcf =[ -6 -4 -4 -6 -6;
                2 4 4 2 2;
                6 6 5.5 5.5 6];
Face(34).bcf =[ 6 4 4 6 6;
                -2 -4 -4 -2 -2;
                6 6 5.5 5.5 6];
Face(35).bcf =[ -6 -4 -4 -6 -6;
                -2 -4 -4 -2 -2;
                6 6 5.5 5.5 6];

for n=1:numel(Face),
    Face(n).bcf = R3(-pi/4)*Face(n).bcf;
    hold on;
end;


%GENERATE WHEELS:
theta=linspace(0,2*pi);
Face(36).bcf =[ 7*ones(1,100);
                2*cos(theta);
                2*sin(theta)-5];
Face(40).bcf =[ 6*ones(1,100);
                2*cos(theta);
                2*sin(theta)-5];
for n=1:length(theta),
    Face(36).bcf(:,n) = R2(28*pi/180)   *Face(36).bcf(:,n);
    Face(37).bcf(:,n) = R3(pi/2)        *Face(36).bcf(:,n);
    Face(38).bcf(:,n) = R3(pi)          *Face(36).bcf(:,n);
    Face(39).bcf(:,n) = R3(3*pi/2)      *Face(36).bcf(:,n);
    Face(40).bcf(:,n) = R2(28*pi/180)   *Face(40).bcf(:,n);
    Face(41).bcf(:,n) = R3(pi/2)        *Face(40).bcf(:,n);
    Face(42).bcf(:,n) = R3(pi)          *Face(40).bcf(:,n);
    Face(43).bcf(:,n) = R3(3*pi/2)      *Face(40).bcf(:,n);
end;

for f=1:length(theta)-1,
    Wheel(1).Face(f).bcf = [7 7 6 6 7;
                        2*cos([theta(f) theta(f+1) theta(f+1) theta(f)
theta(f)]);
                        2*sin([theta(f) theta(f+1) theta(f+1) theta(f)
theta(f)])-5];
    for v=1:5
        Wheel(1).Face(f).bcf(:,v)=R2(28*pi/180)
*Wheel(1).Face(f).bcf(:,v);
        Wheel(2).Face(f).bcf(:,v)=R3(pi/2)
*Wheel(1).Face(f).bcf(:,v);
        Wheel(3).Face(f).bcf(:,v)=R3(pi)
*Wheel(1).Face(f).bcf(:,v);
        Wheel(4).Face(f).bcf(:,v)=R3(3*pi/2)
*Wheel(1).Face(f).bcf(:,v);
    end;
end;
wfaces=length(Wheel(1).Face);

%}


%% INITIALIZE DISPLAYS:
if ~exist('Face')
    load('PlatformGeometry');
end;

%PLOT INITIAL POSITION:
clf;
subplot(4,3,[1:9]);
colormap bone;
```

```
plot3(0,0,0); hold on;
for n=1:numel(Face),
    p(n)=patch(Face(n).bcf(1,:),Face(n).bcf(2,:),Face(n).bcf(3,:),.65);
end;
for w=1:4,
    for f=1:wfaces,
        wp((w-
1)*wfaces+f)=patch(Wheel(w).Face(f).bcf(1,:),Wheel(w).Face(f).bcf(2,:),
Wheel(w).Face(f).bcf(3,:),.65,'LineStyle','none');
    end;
end;

%PLOT UNIT VECTORS:
unitvecs = [15 0 0;
            0  15 0;
            0  0 15];
for n=1:3,
    bcfunitvecs(n)=plot3([0 unitvecs(1,n)],[0 unitvecs(2,n)],[0
unitvecs(3,n)],'r-','LineWidth',4);
end;

%FORMAT AXES:
view(45,5);
camlight(-135,60);
% set(gcf,'Renderer','openGL'); lighting gouraud;
set(gcf,'Renderer','zbuffer'); lighting phong;
material([.4 .1 .8 .8 .1])
grid on;
axis equal;
axis([-1 1 -1 1 -1 1]*15);
%remove background:
%{

  grid off;
  set(gca,'Color',[.8 .8 .8]);
  set(gca,'XColor',[.8 .8 .8],'XGrid','off');
  set(gca,'YColor',[.8 .8 .8],'YGrid','off');
  set(gca,'ZColor',[.8 .8 .8],'ZGrid','off');
%}


%SET UP WHEEL RATE GAGES:
for n=1:4,
    subplot(4,3,10);
    rw(n)=patch([-1 -1 1 1]*.4+n,[0 50 50 0],[.8 .1 .2]);
end;
set(gca,'XTick',[1:4]);
set(gca,'YTick',[-5000:1000:5000]);
set(gca,'YGrid','on');
xlabel('Reaction Wheel');
ylabel('Rate (rpm)');
axis([.5 4.5 0 5000]);

%SET UP TILT LIMIT PLOT:
%In this plot, radius is proportional to the angle between the K body
```

```
%vector and the K eci vector (tilt), and direction (theta) is simply
the
%x-y direction of the tilt.  Think of this plot as a top-down view of
the
%travel of the body "up" direction from inertial "up".
subplot(4,3,11);
tiltplotmax = 40;
tiltmax = 30;
theta=linspace(0,2*pi)';
patch(tiltplotmax*cos(theta),tiltplotmax*sin(theta),[1 .8
.8],'EdgeColor',[1 .8 .8]); hold on;
patch(tiltmax*cos(theta),tiltmax*sin(theta),[.8 1 .8],'EdgeColor',[.8 1
.8]);
plot(0,0,'ko');
for tilt=0:10:tiltplotmax,
    plot(tilt*cos(theta),tilt*sin(theta),'k:');
end;
for theta=0:45:360,
    plot([0 cosd(theta)*tiltplotmax],[0 sind(theta)*tiltplotmax],'k:');
end;
axis([-1 1 -1 1]*(tiltplotmax));
tiltmarker = plot(0,0,'.','Color',[0 .3 0],'MarkerSize',15);
set(gca,'XColor',[.8 .8 .8],'YColor',[.8 .8
.8],'XTick',[],'YTick',[],'XTickLabel',[],'YTickLabel',[],'Color',[.8
.8 .8])
axis equal;


%SETUP QUATERNION HISTORY PLOT:
qhist = [zeros(3,100);ones(1,100)];
thist = zeros(1,100);
subplot(4,3,12);
qhistplot = plot(thist,qhist);


%FINISHED SETUP:
drawnow;
%pause;
subplot(4,3,[1:9]);


%% DATA LINK
%SETUP UDP PORT IF NOT ALREADY INITIALIZED:
delete(instrfind('Type','udp'));
u1 = udp('192.168.0.101', 'RemotePort', 8844, 'LocalPort', 8866);
set(u1,'Timeout',0.2);
set(u1,'BytesAvailableFcnCount',2048);
set(u1,'InputBufferSize',2048);
set(u1,'DatagramTerminateMode','off');
fopen(u1);

%WAIT FOR DATA TO BE DETECTED:
warning('off','instrument:fread:unsuccessfulRead');
disp('waiting for data...');
while 1,
    pause(0.5);
    if numel(fread(u1,4,'double'))>0,
        break;
    end;
end;

%% UPDATE GEOMETRY:
disp('updating...');
tic;
while 1
    pause(0.1); %otherwise cpu can't keep up with data stream
    %dump accumulated data:
    if u1.BytesAvailable>64,
        fread(u1,u1.BytesAvailable);
    end;
    a = fread(u1,8,'double')';
    if length(a)==8,

        q       = a(5:8);
        qhist   = [qhist(2:end),q];
        thist   = [thist(2:end),toc];
        disp(['q:   ',num2str(q,4)]);
        C       = quat2dcm([q(4),-q(1:3)]);
        for n=1:numel(Face),
            for m=1:size(Face(n).bcf,2),
                Face(n).icf(:,m) = C*Face(n).bcf(:,m);
            end;
            set(p(n),'Vertices',Face(n).icf');
        end;

        %update wheel sides:
        for w=1:4,
            for f=1:wfaces,
                for m=1:5,

Wheel(w).Face(f).icf(:,m)=C*Wheel(w).Face(f).bcf(:,m);
                end;
                set(wp((w-
1)*wfaces+f),'Vertices',Wheel(w).Face(f).icf')
            end;
        end;
        %   UPDATE UNIT VECTORS:
        for n=1:3,
            eciunitvecs(:,n)=C*unitvecs(:,n);
            set(bcfunitvecs(n),'XData',[0 eciunitvecs(1,n)],'YData',[0
eciunitvecs(2,n)],'ZData',[0 eciunitvecs(3,n)]);
        end;
        %}


        %   UPDATE WHEEL GUAGES:
        w = a(1:4);
        for n=1:4,
            wrpm = w(n)*60/2/pi;
            set(rw(n),'YData',[0 1 1 0]*wrpm);
        end;
        %}

        %   UPDATE TILT POLAR & CHECK VALUE:
        subplot(4,3,11)
        K=eciunitvecs(:,3);
        theta=atan2(K(2),K(1));
        tilt=atan2(sqrt(K(1)^2+K(2)^2),K(3))*180/pi;

set(tiltmarker,'XData',tilt*cos(theta),'YData',tilt*sin(theta));
        if tilt>tiltmax,
            %audible alarm when platform reaches tilt limit
            mexBeep(1000,100);
            set(tiltmarker,'Color',[.8 0 0]);
        else
            set(tiltmarker,'Color',[0 .3 0]);
        end;


        %   UPDATE QUATERNION HISTORY PLOT:
        for n=1:4,
            set(qhistplot(n),'XData',thist,'YData',qhist(n,:));
        end;


        drawnow;
    end;
end;
```

## C.6. *Star Tracker Processor GUI StarTrackerProcessor.m*

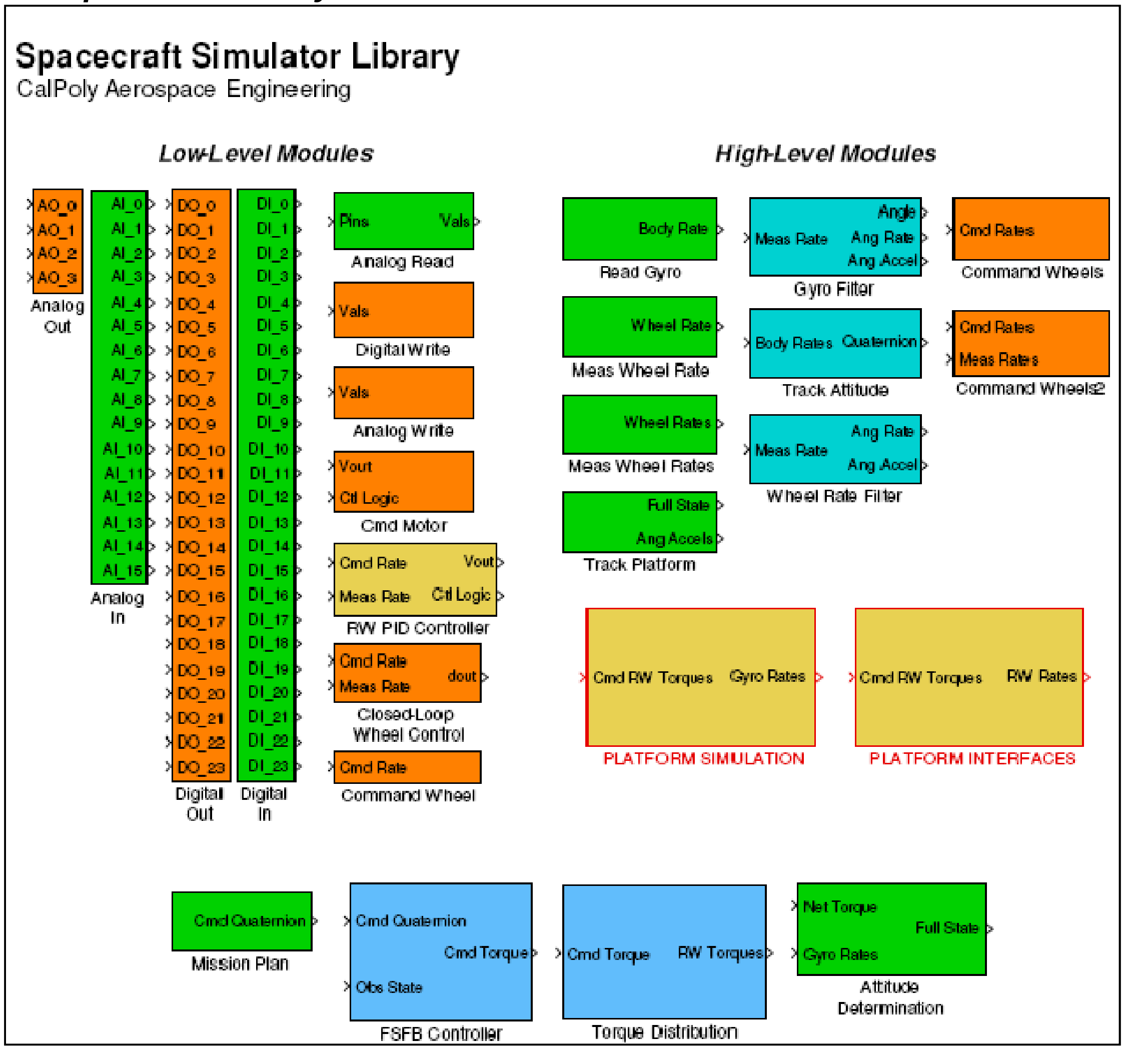
```
function varargout = StarTrackerProcess(varargin)
% STARTRACKERPROCESS M-file for StarTrackerProcess.fig
%      STARTRACKERPROCESS, by itself, creates a new STARTRACKERPROCESS
or raises the existing
%      singleton*.
%
%      H = STARTRACKERPROCESS returns the handle to a new
STARTRACKERPROCESS or the handle to
%      the existing singleton*.
%
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @StarTrackerProcess_OpeningFcn,
...
                   'gui_OutputFcn',  @StarTrackerProcess_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before StarTrackerProcess is made visible.
function StarTrackerProcess_OpeningFcn(hObject, eventdata, handles,
varargin)
% This function has no output args, see OutputFcn.
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

%OPEN MODEL, IF NOT ALREADY OPEN:
if ~evalin('base','exist(''mdlname'')')
    while 1
        prompt={'Enter the name of the navigation or star tracker
acquisition model:'};
        name='Identify Model';
        numlines=1;
        defaultanswer={''};
        answer=inputdlg(prompt,name,numlines,defaultanswer);
        mdlname = answer{1};
        assignin('base','mdlname',mdlname);
        if mdlname(end-3)=='.'
            mdlname = mdlname(1:end-4); %truncate the file extension
        end;
        if exist(mdlname)
            break;
        else
            disp('Model not found in path');
            disp(' Models in current directory:')
            dir '*.mdl';
        end;
    end;
end;
mdlname = evalin('base','mdlname');
if isempty(find_system('Type','block_diagram','Name',mdlname));
    open_system(mdlname);
end
if ~evalin('base','exist(''qst'')')
    assignin('base','qST',[0 0 0 1]');
    assignin('base','STcnt',0);
end;
set(handles.pushbutton2,'Enable','on');
```

```
% UIWAIT makes StarTrackerProcess wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command line.
function varargout = StarTrackerProcess_OutputFcn(hObject, eventdata,
handles)
varargout{1} = handles.output;


% --- Executes on button press in enablebutton.
function enablebutton_Callback(hObject, eventdata, handles)

% --- Executes on button press in plotbutton.
function plotbutton_Callback(hObject, eventdata, handles)

% --- Executes on button press in triggerbutton.
function triggerbutton_Callback(hObject, eventdata, handles)

% --- Executes on button press in pushbutton2.
function pushbutton2_Callback(hObject, eventdata, handles)
get(hObject,'UserData')
if get(hObject,'UserData')==0;
    set(hObject,'UserData',1);
    set(hObject,'String','Stop');
    disp('Starting ST Loop');
    startrack(eventdata,handles);
    %(CALL ST LOOP FUNCTION HERE)
else
    set(hObject,'String','Start');
    set(hObject,'UserData',0);
    disp('Stopping ST Loop');
end;

% --- Executes on button press in procbutton.
function procbutton_Callback(hObject, eventdata, handles)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
%%%% STAR TRACKER LOOP
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function startrack(eventdata,handles)
mdlname = evalin('base','mdlname');

%Setup event listener to monitor ST trigger in model:
global stTrigCnt; %#ok
stTrigCnt       = 0;
stCnt           = 0;
qst             = [0 0 0 1]';
set_param(mdlname,'SimulationCommand','start');
```

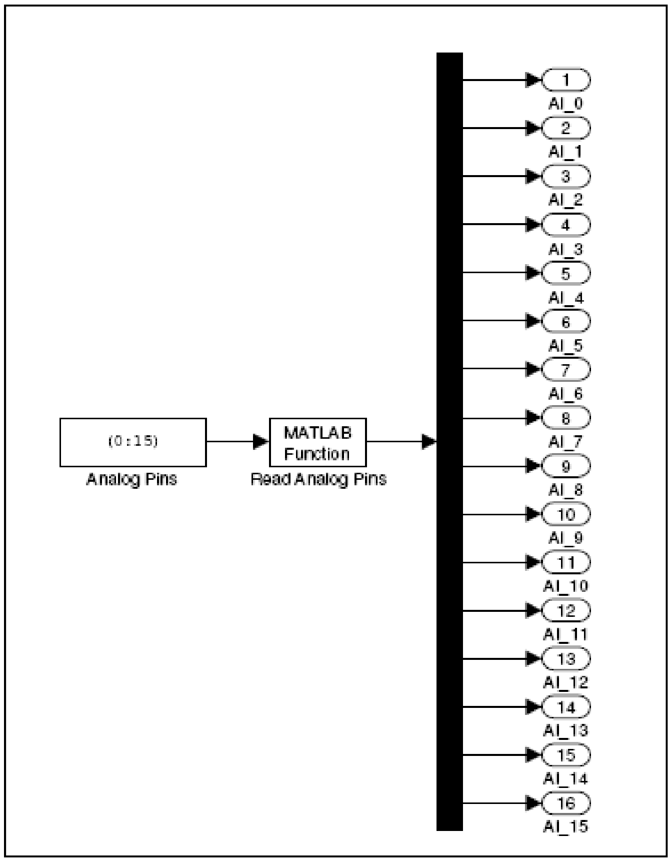
```
listener = add_exec_event_listener([mdlname,'/TrigCnt'],'PostOutputs',
@StarTrackerTriggerMonitor);
prevtrigcnt = stTrigCnt;
set(handles.trigcnt,'String',num2str(stTrigCnt));

%Init webcam:
caminfo = imaqhwinfo('winvideo',1);
cam     = videoinput('winvideo', 1,caminfo.SupportedFormats{5});
set(cam,'FramesPerTrigger',Inf);
start(cam);
axes(handles.axes1);

procsimtime = 3; %(for fake ST only)

%THE STAR TRACKER LOOP:
while get(handles.pushbutton2,'UserData')==1

    %check if simulation stopped:
    if strcmp(get_param(mdlname,'SimulationStatus'),'stopped');
        set(handles.pushbutton2,'String','Start');
        set(handles.pushbutton2,'UserData',0);
        break;
    end;

    %TRIGGER ST PROCESS EACH TIME TRIGGER COUNT IS RAISED:
    if stTrigCnt>prevtrigcnt
        tic;
        set(handles.trigcnt,'String',num2str(stTrigCnt));
        set(handles.procbutton,'Value',1);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %CAPTURE NEW IMAGE:

        img = peekdata(cam,1);
        toc;
        if get(handles.plotbutton,'Value')
            imagesc(ycbcr2rgb(img));
            set(handles.axes1,'XTick',[],'YTick',[]);
        end;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %PROCESS IMAGE:
        %(REPLACE THE FOLLOWING LINES WITH ACTUAL PROCESSING CODE)
        t=toc; %(for fake ST only)
        pause((procsimtime-t) *(t<procsimtime)); %(for fake ST only)
        qst = rand(4,1);
        assignin('base','qST',rand(4,1));   %send outputs to base
workspace
        evalin('base','STcnt=STcnt+1;');        %signals that process
has completed
        set_param(mdlname,'SimulationCommand','update');
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        stCnt = stCnt+1;
        set(handles.procbutton,'Value',0);
        prevtrigcnt = stTrigCnt;
    end;
    pause(0.01);    %(to free up some cpu time between checks)
end;
stop(cam);
delete(cam);


function trigcnt_Callback(hObject, eventdata, handles)

% --- Executes during object creation, after setting all properties.
function trigcnt_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

## Appendix D. Simulink Models

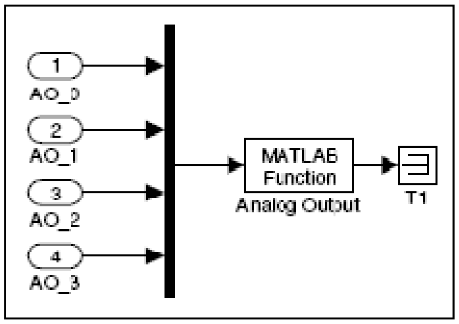
### D.1. SpaceSim Library



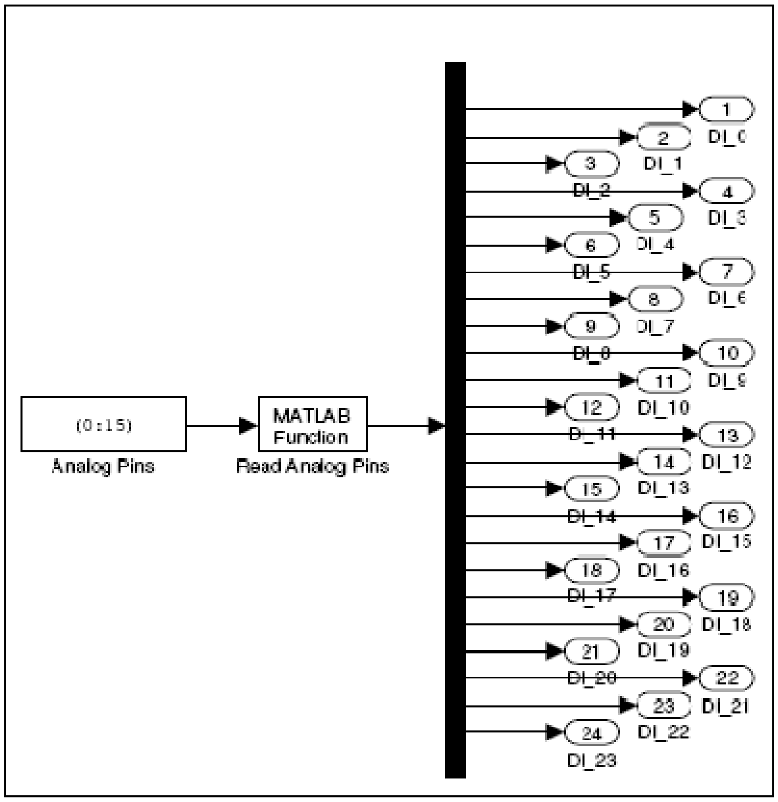
**SpaceSimLib.mdl**



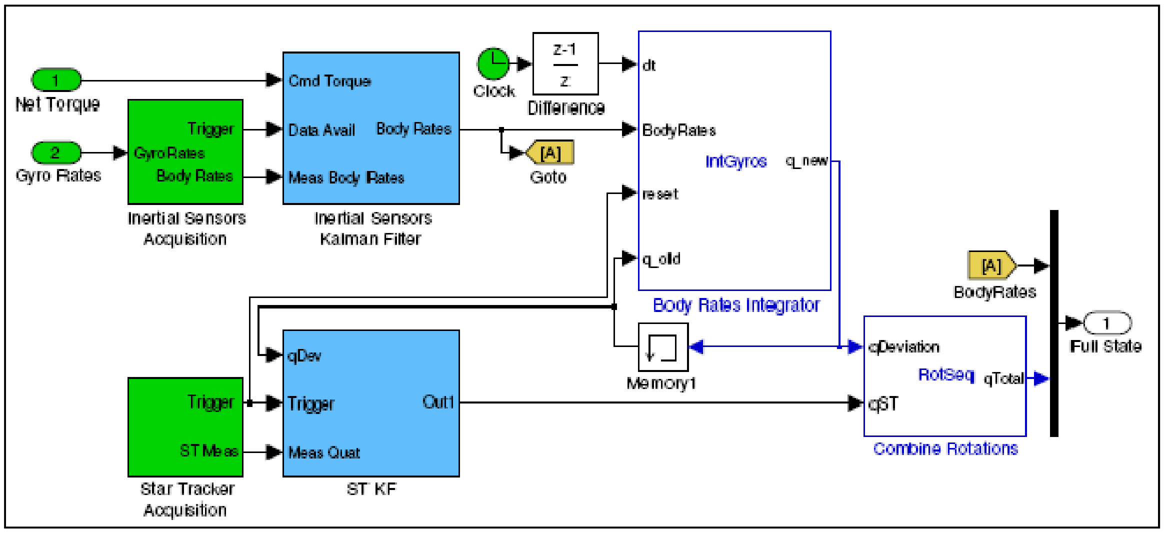
**Analog In**



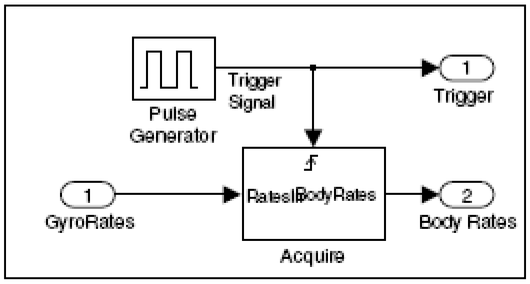
**Analog Out**



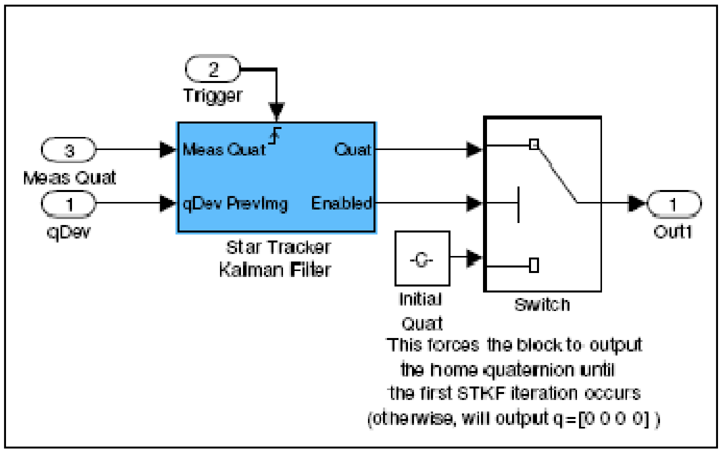
**Digital In**



**Attitude Determination**

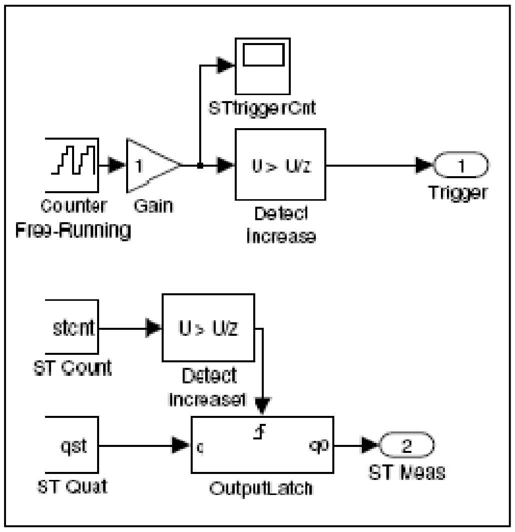


**Inertial Sensors Acquisition**

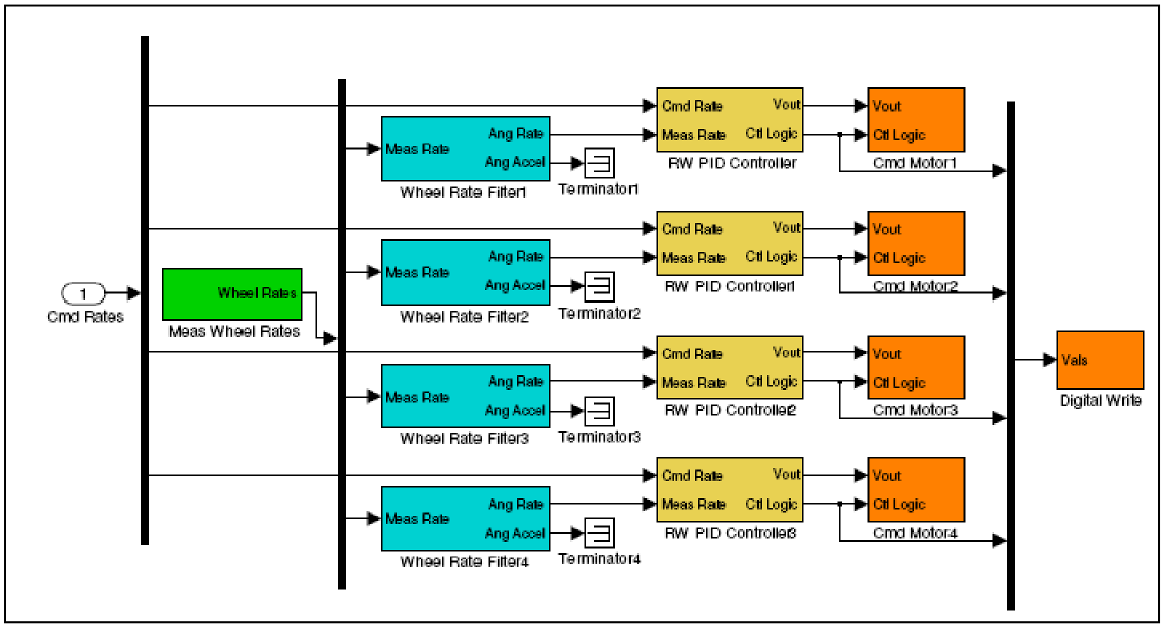


2 Trigger
3 Meas Quat
1 qDev
Meas Quat
qDev PrevImg
Quat
Enabled
Star Tracker Kalman Filter
-C-
Initial Quat
Switch
1 Out1
This forces the block to output the home quaternion until the first STKF iteration occurs (otherwise, will output q=[0 0 0 0] )

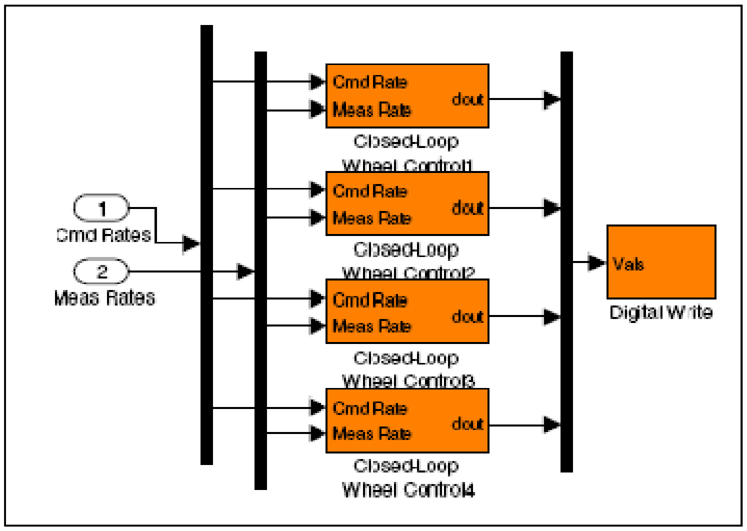
**ST KF**



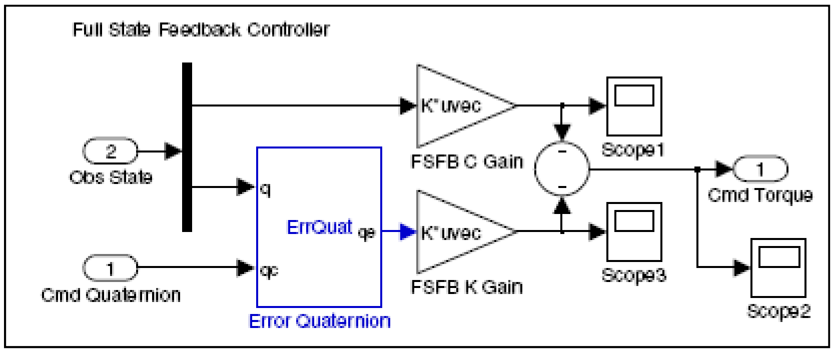
**Star Tracker Acquisition**



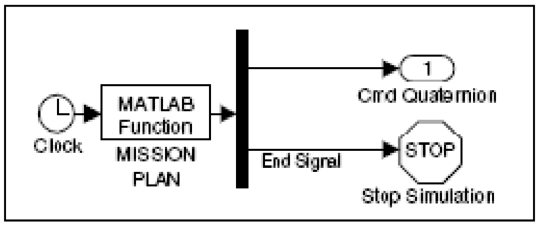
**Command Wheels**


1
Cmd Rates
2
Meas Rates
Cmd Rate
Meas Rate
dout
Closed-Loop Wheel Control1
Closed-Loop Wheel Control2
Closed-Loop Wheel Control3
Closed-Loop Wheel Control4
Vals
Digital Write

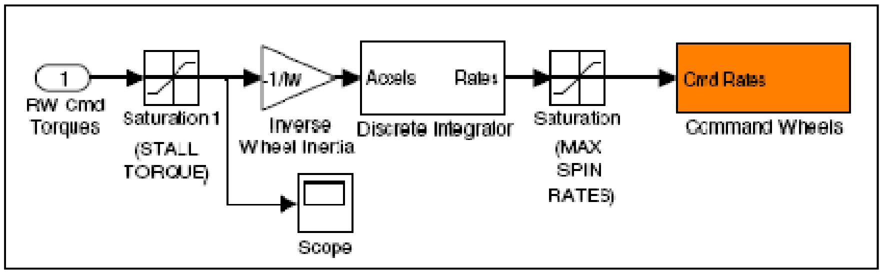
**Command Wheels 2**



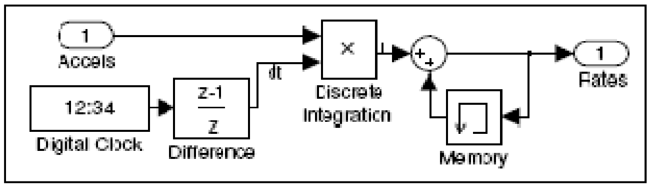
**FSFB Controller**



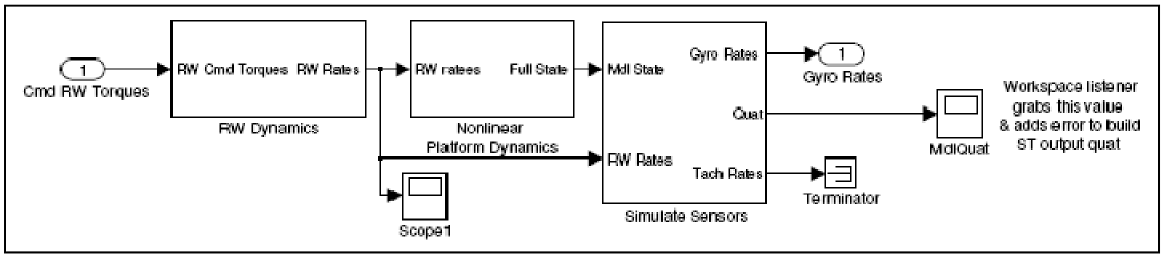
**Mission Plan**



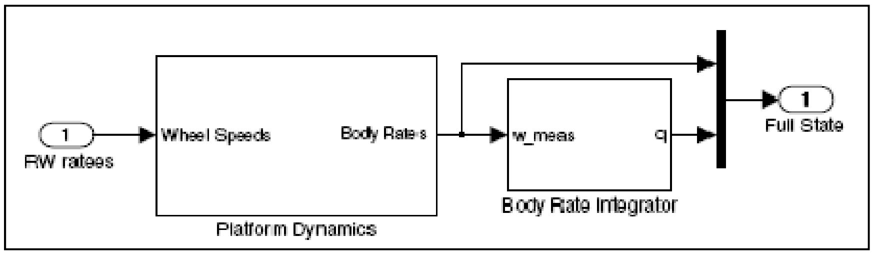
**RW Interfaces**



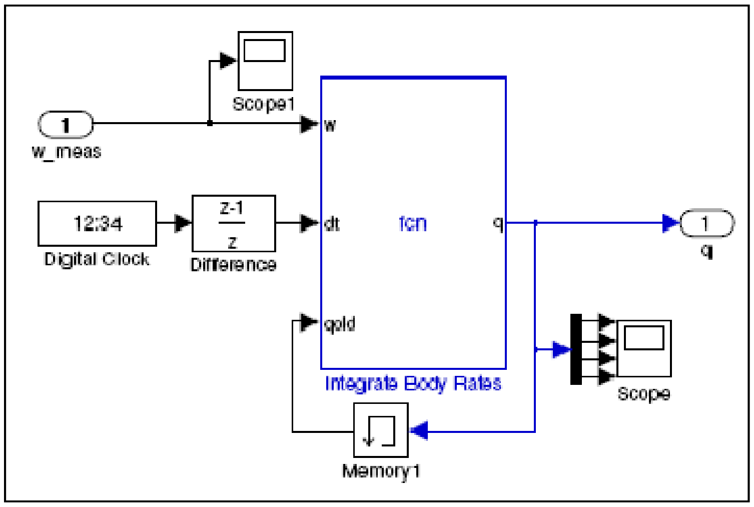
**Discrete Integrator**



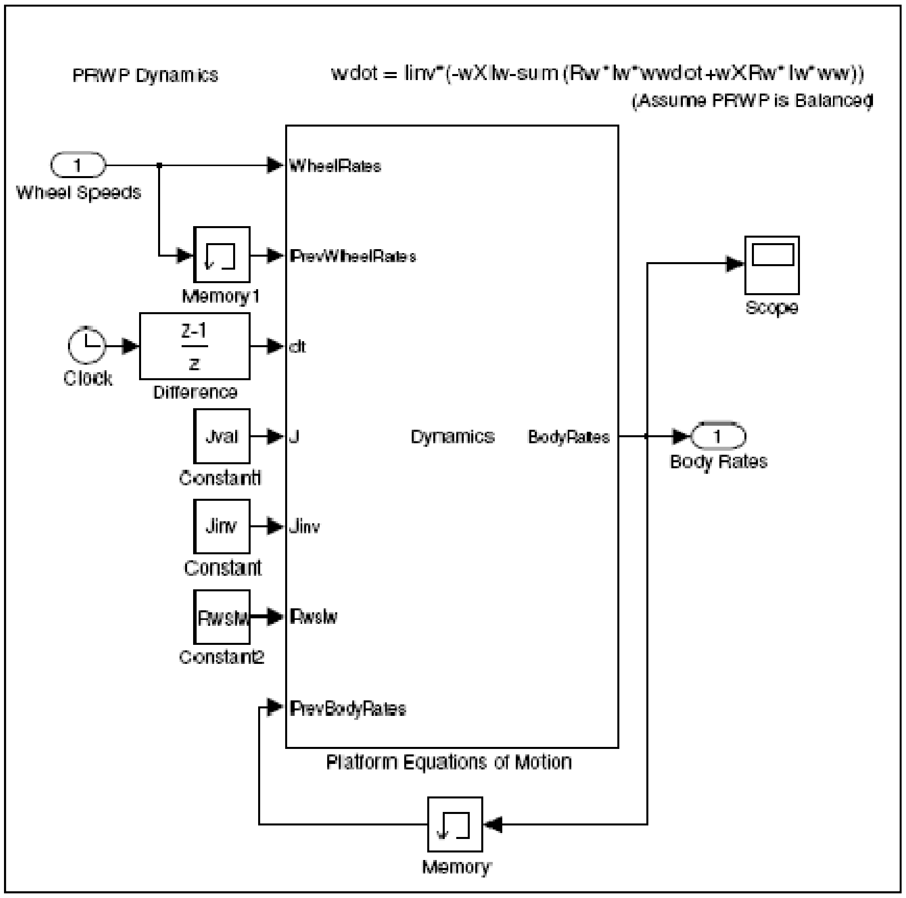
**PLATFORM SIMULATION**



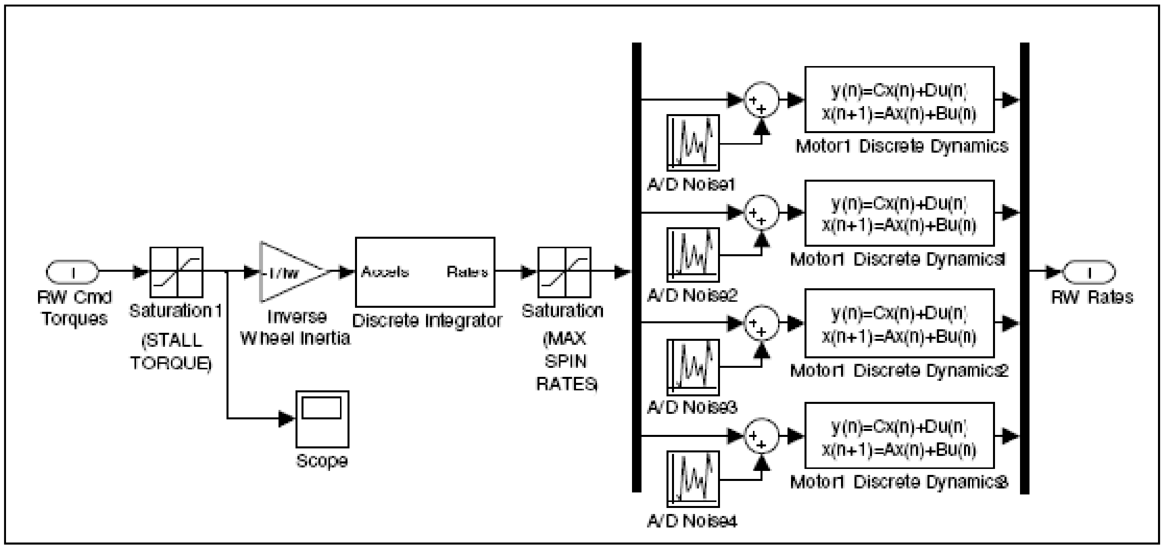
**Nonlinear Platform Dynamics**



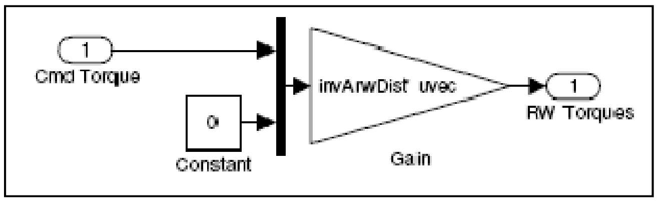
**Body Rate Integrator**



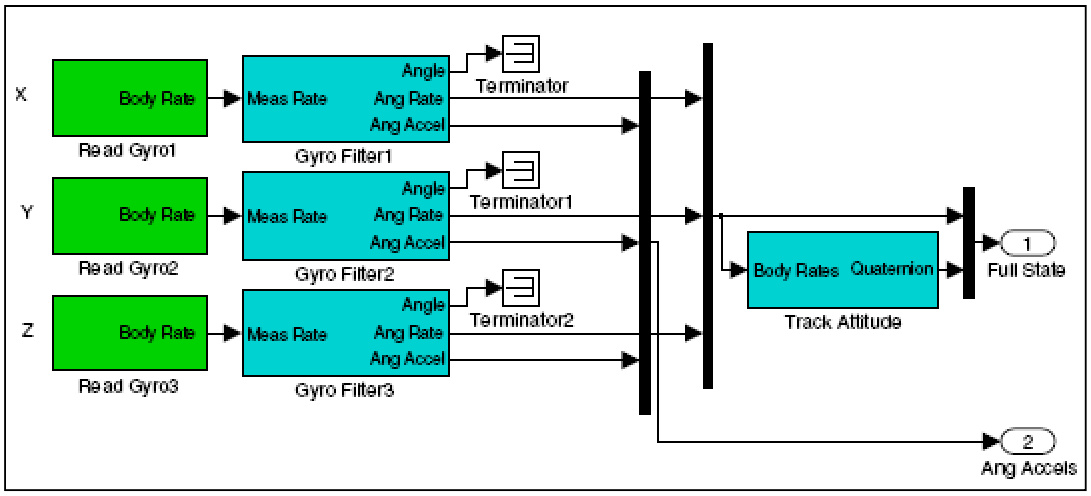
**Platform Dynamics**



RW Dynamics



Torque Distribution



Track Platform