ABSTRACT

Title of Thesis: SYSTEM MODELING AND CONTROLLER DESIGN FOR A SINGLE DEGREE OF FREEDOM SPACECRAFT SIMULATOR

Melissa F. Vess, Master of Science, 2005

Thesis directed by: Professor Robert M. Sanner
Department of Aerospace Engineering

Control systems theory is an important field of study for many branches of engineering. Teaching control systems to engineering students, however, is often difficult due to the abstract nature of the subject. TableSat is a single degree of freedom spacecraft simulator that includes sensors, actuators, a power system, and a flight processor. Students can use TableSat to design and test controllers, allowing them to see how theoretically designed controllers function in a real system. TableSat, like all real systems, is highly nonlinear. To make TableSat an effective teaching tool, the system nonlinearities are identified and compensation methods undertaken to eliminate those nonlinearities. Linear and truth system models are created for use in controller design and testing. The system models are tested and verified and then used to design and test several controllers and estimators. Results are presented that compare results for the linear model, truth model, and real TableSat system.

SYSTEM MODELING AND CONTROLLER DESIGN FOR A SINGLE DEGREE OF FREEDOM SPACECRAFT SIMULATOR

by

Melissa F. Vess

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2005

ACKNOWLEDGEMENTS

There are many people who have helped me with this endeavor, and without whom I could not have accomplished this thesis. First, I would like to thank Tom Stengle, my supervisor, for allowing me the opportunity to pursue this project and get my Master's Degree. I would like to thank Dave Mangus and the other members of the TableSat team for letting me completely change TableSat as they knew it. I would also like to thank Rob Sanner, my advisor, for his knowledge, help and support. I would also like to thank my SDO colleagues for putting up with my ever varying and sometimes inconvenient school schedule. And finally, I would like to thank Dave Vess, my husband, for his ever present love and support. Without him, I probably would have given up a long time ago.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction and Background

Control systems theory and design is an important and ever changing field of study for many different types of engineers. From the simple feedback controller that maintains oven temperature, to the complex, multi-degree-of-freedom control system that steers a spacecraft as it flies through the Solar System, embedded controllers show up everywhere. And while based in mathematical principles, controller design is often as much art as it is science. A controller is not a physical object you can see. It is an algorithm, typically implemented in computer code, that alters system performance. It is intangible, it is abstract, it is vague, but it is also very powerful. It allows automatic control of a 3200 kg spacecraft without human intervention. It maintains an automobile's speed without having driver intervention.

The intangibility of the algorithms which enable these feats can make control systems engineering a difficult topic to teach students. The basic principles of linearity, stability, robustness, and control can be explained in a mathematical sense; Bode, Nyquist, and Root Locus plots can be generated; phase margin, gain margin, and bandwidth can be defined, but the physical meaning of these concepts and tools is often a leap that students cannot make based on theory alone. They cannot *see* how transforming errors into specific control inputs as specified by a given transfer function will produce the desired results. To help answer these questions, students need something that demonstrates a real control system, something they can see and touch. Something that allows them to see how a given controller will modify a dynamic system, and how modifying the controller parameters corresponds and changes system response.

TableSat is an interactive, single degree-of-freedom spacecraft simulator designed as a tool for demonstrating and teaching control system design. Specifically, it uses hardware and software similar to that found in a current spacecraft attitude control system (ACS), and therefore can illuminate how the ACS design interacts with the physics of a spacecraft. TableSat includes a mechanical structure, an on-board flight processor, attitude and rate sensors, propulsive actuators, a power supply, and a communications system. These features allow the user to simulate an ideal spacecraft with unlimited power, attitude control, communications, *etc.*, as well as a true-to-life spacecraft that includes realistic limitations on these subsystems. In this manner, students can see how neglecting things, such as friction, to allow for a linear analysis, can often have detrimental effects on the controllability and stability of the real system.

1.1 TableSat Background

TableSat was originally designed as a demonstration tool for a class held at NASA's Goddard Space Flight Center entitled *Attitude Control Systems for Non-ACS Engineers*. The purpose of this class was to provide a general overview of what it is that attitude control systems engineers do. Everything from quaternion math, to bode plots, to types of attitude sensors and actuators was discussed. This initial version of TableSat provided a way to demonstrate how the theoretical design of an attitude control system manifests on an actual spacecraft.

The original TableSat used a balanced, 15-inch diameter disc containing a three-axis magnetometer (TAM), four coarse sun sensors (CSSs), and a single-axis gyro for attitude sensors; two computer fans for actuators; and two battery packs for power. The onboard flight processor was a Microchip PIC16F874 8-bit micro controller, which was programmed in low level assembly language, and utilized an RF serial link for communications with a ground station. Due to the limited memory on the micro controller, the flight processor could not provide on-board, closed-loop feedback control. Instead, sensor readings were sent via the RF link to a laptop running Mathwork's Simulink. Data processing, control calculations, and data plotting were performed in Simulink, and the actuator commands were uplinked to TableSat using the RF serial link. This Simulink-in-the-loop set up resulted in a top control rate of approximately 2.5 Hz. The fans were actuated in a Pulse Width Modulated (PWM) format, with a resolution of four pulse cycles per control cycle.

At the completion of the ACS class, a group of ACS engineers from the Flight Dynamics Analysis Branch at Goddard started working with TableSat to extend and improve its uses and capabilities. One of the early such uses of TableSat was to demonstrate to middle and high school students the fundamentals of control systems engineering. A short presentation was put together that incorporated simple, hands-on activities to illustrate the concepts of control, stability, and feedback. At the end of the presentation, TableSat was used to show how those same ideas are incorporated into the control of a spacecraft.

After the demonstration of TableSat to middle and high school students, the TableSat team wanted to bring TableSat to colleges and universities as a teaching tool for undergraduate students. In particular, the TableSat team felt that an undergraduate course in linear control systems would be the ideal place to incorporate TableSat because it demonstrates a complete feedback control system. The goal was to present a demonstration of TableSat, and then the students would design a controller for TableSat using the tools they learned in linear control systems.

The TableSat team approached the Aerospace Engineering Department at the University of Maryland, College Park and met with favorable feedback from the faculty. A demonstration of TableSat was presented in the spring of 2003 and resulted in enthusiastic responses from the students. However, upon further investigation into the feasibility of using TableSat as a design project, it was determined that in its current state, TableSat would not be an effective project for a linear systems class. As with all "real" systems, TableSat was highly nonlinear, due most notably to the limitations in the control and actuation rates. In addition, the TableSat system dynamics were not carefully modelled and analysis by linear methods proved impossible. Without a system model, it would be difficult for students to design a linear controller that would effectively control TableSat. Furthermore, given the nonlinearities in the TableSat system, a controller designed with linear tools would likely not yield the desired response, and would thus be less effective as a teaching tool.

As a result of these discoveries, the TableSat team decided to redesign TableSat to make it more suitable for showing in a linear systems class, while retaining the ability to make TableSat a "real" nonlinear system for advanced analysis and applications. This functionality would allow students to see the response of an ideal system to a linear controller, then progressively add nonlinearities into the system to show the effect of those nonlinearities on the system response.

A team at the University of Maryland, headed by Dr. Robert Sanner and Dr. Ella Atkins, both of the Aerospace Engineering Department, was tasked with upgrading TableSat to provide a better teaching tool by reducing the nonlinearities in the system while retaining most of the current functionalities of the system. Most importantly, the TableSat team wanted to retain the ability to design and implement controllers using Simulink.

1.2 TableSat Goals

Based on discussions with the Goddard TableSat team, comprised of David Mangus, Kristin Makovec, Paul Mason, Oscar Hsu, John VanEeopel, and myself, and the University of Maryland, a series of specific project goals was established. The following is a list of these goals.

- Increase the TableSat control, estimation, and actuation loop rates so that the system can be considered a continuous, linear system.
- Implement analog control of the actuators to provide for a more continuous system.
- Establish the ability to switch between analog and discrete control of the actuators.
- Establish the ability to alter controller, estimator, and actuator loop rates.
- Develop TableSat Equations of Motion and characterize system parameters.
- Develop a TableSat truth model including fan and TableSat dynamics, friction, sensor noise parameters, etc., that can be used to test controllers and estimators.

- Identify major nonlinearities in the system, and develop algorithms and compensation methods to effectively remove those nonlinearities from the system.

- Retain the ability to add nonlinearities to the system to show the difference between ideal and "real" systems and for advanced controller and estimator design.

- Develop a TableSat linear model that can be used to design controllers and estimators.

- Verify truth and linear models by comparing predicted performance to real TableSat performance.

- Design controllers and estimators using the system model. Test and compare designed controllers and estimators to real system data.

- Create a simple user interface that allows the user to interact with the system (*e. g.* load new controllers and state estimators).

- Retain the ability to use Simulink to design and implement advanced controller and estimator designs.

1.3 Project Contributors

Over the course of the TableSat project, there have been many people who have contributed to the success of the project. The following is a list of contributors to the TableSat project and their main contributions.

- Dr. Ella Atkins, University of Maryland - Design and implementation of all hardware components; Design and implementation of all digital/analog circuits; Prometheus OS (QNX) configuration; Low level Prometheus initialization code

- Dan Proffen, University of Maryland - Low level flight software development: initial C code outline; global structure definitions; development of controller, estimator, and actuator thread timers; initial outline of controller, estimator, and actuator threads

- Dr. Robert Sanner, University of Maryland - Flight software upgrades; MATLAB-TableSat communications code development

In addition to the above contributors, my contributions to the project include:

- Low level flight software development: development of controller, estimator, and actuator thread timers; initial outline of controller, estimator, and actuator threads; conversion of raw sensor readings to counts; and conversion of sensor counts to engineering units

- Sensor calibration routine development
- TableSat GUI development
- Simulink truth model development and tuning
- Sensor noise characterization
- TableSat equations of motion parameter identification
- Linearization of physical system
- Linearized model analysis
- Classical controller development and testing
- Model Based Controller-Observer development and testing
- N-Sample state estimator development and testing
- Kalman Filter development and testing

1.4 Outline of Thesis

This thesis provides a description of the new TableSat system. Chapter 2 gives an overview and general description of the hardware currently onboard TableSat. Chapter 3 presents an outline of the new software. Details are provided on the layout and interaction between the different threads, and how the sensor readings are converted into meaningful engineering data. Chapter 4 provides a description on the different functionalities that TableSat possesses, and explains how the user interacts with TableSat via the Graphical User Interface (GUI), which allows the user to change TableSat's variable parameters. Chapter 5 describes the development of the TableSat system model and parameter identification. Chapter 6 explains the efforts undertaken to linearize the system and presents and validates the linear system model. Finally, Chapter 7 presents case studies of some of the different functionalities of TableSat, including using different controllers and state estimators.

Chapter 2

Description of TableSat Hardware

The complete TableSat system is pictured in Figure 2.1. The hardware is mounted on a 15-inch diameter disc, which is balanced on a spindle that acts as TableSat's rotation point. The spindle contact point on TableSat is an aluminium hub mounted in the center of the TableSat disc. The spindle and pivot hub are pictured in Figure 2.2. From a hardware prospective, TableSat's main components are the flight processor, a single-axis gyro, four coarse sun sensors, a three-axis magnetometer (TAM), two computer fans, a battery, a wireless ethernet access point, a router, and a voltage regulator card. Note that the router is the only hardware not mounted to the TableSat disc. Each of these components is discussed briefly in the sections below. All of the upgrades to the TableSat hardware were accomplished by Dr. Ella Atkins of the University of Maryland, and a more complete discussion of the hardware can be found in [1].

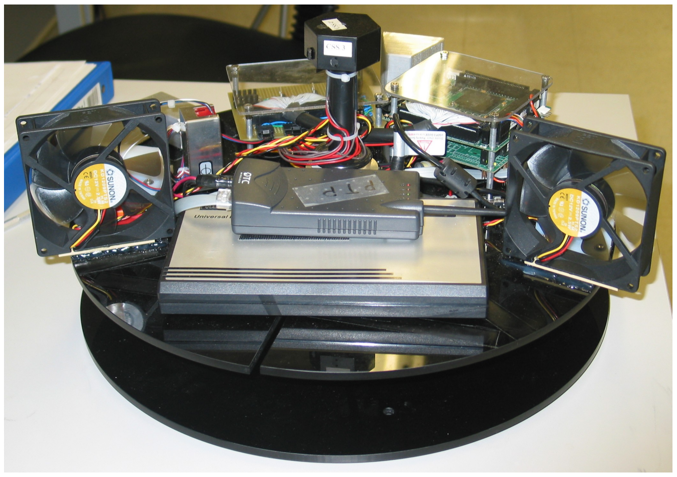


Figure 2.1: Complete TableSat system.

2.1 Flight Processor

The TableSat flight processor, pictured in Figure 2.3 is the Prometheus embedded PC/104 CPU manufactured by Diamond Systems Corporation. This card integrates three separate modules, CPU, Ethernet, and Analog Input/Output, on a single card.

2.1.1 CPU

The Prometheus CPU is a 486-DX2, 100 MHz processor with 32 MB SDRAM system memory and 2 MB flash memory. The CPU operating system is QNX Neutrino, a UNIX based Real-Time Operating System (RTOS) developed by QSSL.

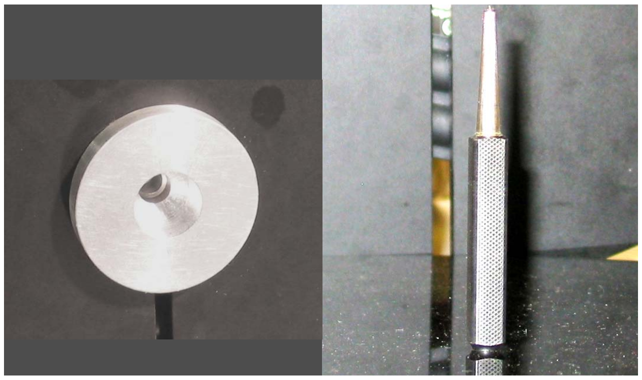


Figure 2.2: TableSat hub and spindle.

2.1.2 Ethernet

The Prometheus includes a 100 Mbps Ethernet connection using 100BaseT wiring. The Ethernet chip is the National Semiconductor DP83815 MacPhyter chip, and is connected to the CPU via the Prometheus's internal PCI bus.

2.1.3 Analog Inputs

The Prometheus has 16 single-ended or 8 differential analog inputs with 16-bit resolution. A *single-ended* input attaches to the Prometheus using two wires, input and ground. The measured input voltage is the difference between the input and the ground. A *differential* input attaches to the Prometheus using three wires, positive input, negative input, and ground. The measured input voltage is the difference between the positive and negative inputs. One advantage of using differential inputs is that they have a higher immunity to noise. Because noise affects the positive and negative input wires equally, the noise will be cancelled

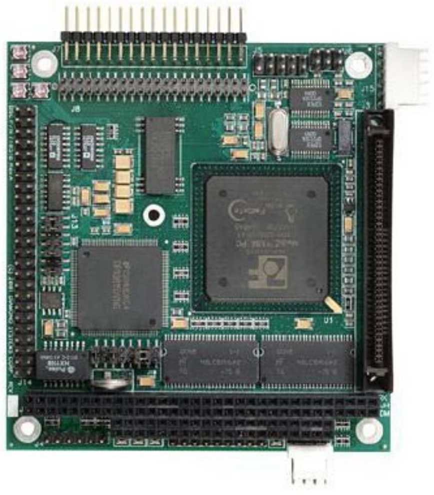


Figure 2.3: TableSat flight processor: Diamond Systems Corporation's Promethus PC/104 CPU

out when the difference is taken. The analog inputs can accept either bipolar or unipolar, variable input ranges with a maximum range of +/-10V or 0 - 10V and a minimum range of +/-1.25V or 0 - 2.5V.

Analog to digital (A/D) conversions can be performed on a single channel or a range of channels, and can be triggered internally or externally. Furthermore, the Prometheus can operate in scan mode and/or interrupt mode. When a scan of multiple channels is requested, the A/D conversion for the first channel of the scan is triggered externally, and the A/D conversions for each subsequent channel are triggered internally. Up to 48 A/D samples can be stored in the programmable FIFO (First-In, First-Out) for reliable high-speed sampling and scan operation. If the FIFO reaches its programmed threshold while in interrupt mode, an interrupt request will occur, which can trigger an interrupt routine to read the data from the FIFO. If the Prometheus is not being operated in interrupt mode, the A/D status bit has to be monitored. When the status bit is low, the A/D conversion or scan is complete and the data must be read from the FIFO. If the data is not read from the FIFO and the FIFO reaches its maximum 48 samples, the next attempt to sample an input will trigger the overflow bit and no more samples will be accepted until the FIFO is reset. A single A/D conversion can take up to 5 microseconds to complete. As most processors and software can run faster than that, it is important to wait until the status bit goes low before trying to read the A/D converter [3].

The A/D data is stored as a 16-bit value and is read as two 8-bit bytes. The least significant byte (LSB) must be read before the most significant byte (MSB) because the data is inserted into the board's FIFO in that order. The A/D data may only be read one time because each time the FIFO is read, its internal pointer advances and that byte is no longer available. The data from the A/D is a signed integer ranging from -32768 to +32767. That integer can be converted back to a voltage, and from there into a meaningful engineering unit. Conversion formulas vary depending on the range of the analog input [3].

Currently, the TableSat Flight Processor is running in differential mode with the three axes of the TAM, the gyro, and the four CSS's making up the eight differential analog inputs. All eight analog inputs have a unipolar range of 0 to 5V. A/D conversions of all eight inputs are performed sequentially using the scan mode of the board. The FIFO stores the eight readings until they are read or another scan is called for.

2.1.4 Analog Outputs

The Prometheus has four analog outputs, each with 12-bit resolution. Like the analog inputs, the analog outputs can be unipolar or bipolar; however the output ranges are not variable. The unipolar and bipolar output ranges are 0 to 10V or +/- 10V, respectively. Because the digital to analog (D/A) converter has a 12-bit resolution, the range of inputs into the D/A converter is 0 to 4095, where 0 corresponds to the lowest output voltage and 4095 corresponds to the largest output voltage. The resolution of the analog output, therefore, depends on the total desired voltage range. For a unipolar output, the resolution is 2.44 mV, and for a bipolar output, the resolution is 4.88 mV [3].

The TableSat Flight Processor currently uses only two analog outputs for the two 12-Volt computer fans that act as positive and negative thrusters. Each fan can accept a voltage input ranging from 0 to 12 V, therefore the flight processor is run in unipolar mode with an output range of 0 to 10 V. To maintain the full range of fan voltages, the output from the flight processor is sent to the voltage regulator card, which amplifies the output voltages such that the commands sent to the fans cover the full 0 to 12 V range.

2.2 Attitude and Rate Sensors

The TableSat rate sensor is a single-axis gyro. The attitude sensor suite is comprised of four coarse sun sensors and a three-axis magnetometer.

2.2.1 Gyro

The TableSat gyro is the GyroChip Horizon manufactured by BEI Technologies, Inc. It is small (2.20 x 0.994 x 0.994 inches), lightweight (60 grams), and features low drift and high reliability.

Figure 2.4: TableSat gyro: BEI Technologies's GyroChip Horizon

The Horizon utilizes a one piece, micromachined, vibrating quartz sensing element. When TableSat rotates, the Horizon produces a voltage proportional to the rate of rotation. The output voltage can vary between 0.5 and 4.5 Volts, which corresponds to a spin rate range of +/- 90 degrees per second [2].

2.2.2 Coarse Sun Sensors

The TableSat Coarse Sun Sensors (CSSs), or photovoltaic detectors, are miniature silicon solar cells, that convert light impulses directly into electrical charges. The output from the CSSs approximately resembles a cosine curve. When the light source is directly aligned with the CSS, the output is at its maximum. When the light source is 90 degrees away from the CSS, the output is at its minimum. In between, the output is roughly proportional to the cosine of the angle between the CSS and the light source. The practical field of view of a single TableSat CSS is about +/- 80 degrees around an axis normal to the CSS. In ambient lighting, the TableSat CSSs generate approximately 0.4 Volts, which is then amplified to produce a signal that ranges between zero and five volts.

TableSat has a total of four CSSs, which are mounted approximately 60 degrees apart on a hexagonal CSS head. With the practical, 80 degree CSS field of view, this CSS arrangement gives roughly 340 degrees of CSS coverage. The 20 degrees of no CSS coverage is centered about the +/-180 degree line. Figure 2.5 shows three of the CSSs mounted on their hexagonal head. The fourth CSS is mounted opposite to the one marked CSS1.

2.2.3 Three-Axis Magnetometer

The TableSat three-axis magnetometer (TAM) is the Honeywell HMC2003 Three-Axis Magnetic Sensor Hybrid. This small, highly sensitive TAM is used to mea-

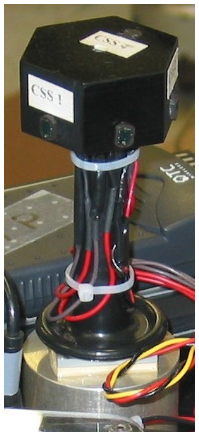


Figure 2.5: Coarse Sun Sensor (CSS) layout

sure the strength and direction of an incident magnetic field. It is a combination of Honeywell's most sensitive megneto-resistive sensors, the HMC1001 and HMC1002, and can detect fields less than 40 microgauss and up to +/- 2 gauss. The HMC2003 interface is completely analog, and outputs are available for each X, Y, and Z-axis.

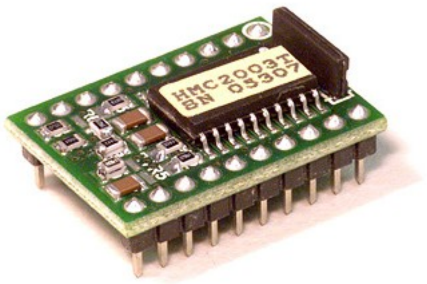


Figure 2.6: TableSat TAM: Honeywell's HMC2003 Three-Axis Magnetic Sensor Hybrid

The HMC2003 is packaged on a small printed circuit board, approximately

1 inch by 0.75 inches. The output voltage can vary between 0.5 and 4.5 Volts, which corresponds to a magnetic field strength of +/- 2 gauss. The TAM operates from a single 6 to 15 Volt power supply [4].

2.3 Actuators

The TableSat actuators are two Sunon KD1209PTS2 12 Volt computer fans mounted such that they spin TableSat in either a positive (counter clockwise) or negative (clockwise) direction. The fans are brushless DC fans, approximately 92 x 92 x 25 mm in size, with a mass of 95 g. The have a power consumption of 1.7 Watts and a current draw of 0.14 Amps. The fan speed is proportional to the applied voltage, with a maximum fan speed of approximately 2500 RPM, and a maximum volumetric flow rate of 44.5 CFM. Along with power and ground, these fans also include a third wire that feeds back the actual fan speed and can be used to close a control loop around the fan speed [8].

2.4 Power Supply

The TableSat power supply is the Polarmate PM111-DC external laptop battery. It is a lithium-ion battery and has two output voltages: 16 Volts and 19 Volts. TableSat uses the 16 Volt configuration. It has a capacity of 111 Watt-Hours, which will power TableSat for approximately ten hours. The battery has an

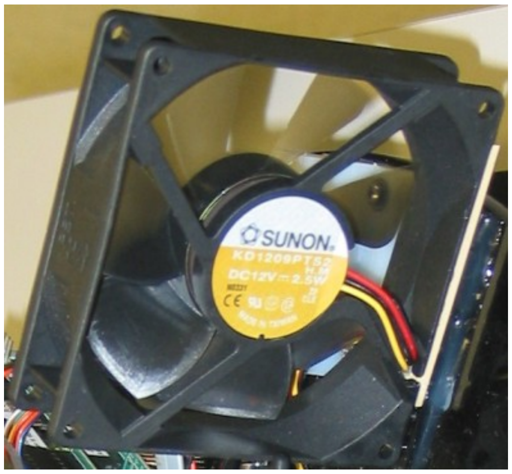


Figure 2.7: TableSat Actuator: Sunon's KD1209PTS2 12 Volt computer fan.

AC adapter that plugs into a standard wall socket for charging, which can take between three and five hours depending on the battery's state of charge. Five LEDs indicate the battery's charge state. It has overall dimensions of 7.7 x 4.5 x 1.0 inches, and weighs approximately 1.5 lbs.

Figure 2.8: TableSat Battery: Polarmate's PM111-DC external laptop battery.

2.5 Voltage Regulator

Recall that for TableSat, the analog outputs from the Prometheus output unipolar voltages from 0 to 10 Volts. Recall also that the TableSat actuator fans can accept inputs from 0 to 12 Volts. The voltage regulator circuit is used to scale the voltage output from the Prometheus such that the input into the fans range from 0 to 12 Volts. It also ensures that the correct current flows to the different components. With the 12-bit D/A converter, the resolution of the output is 2.44 mV. The voltage regulator circuit scales that resolution such that the minimum change in voltage that can be commanded to the actuators is 2.93 mV.

2.6 Communications Hardware

TableSat communicates to its ground station (laptop) via the OTC Wireless ASR102 wireless access point routed through a Linksys BEFW11S4 Wireless-B broadband router.

The ASR102, which can be seen in Figure 2.9, is a portable, high-speed 802.11b wireless access point with a maximum speed of 11 Mbps. It plugs directly into the ethernet, and the speed adjusts automatically depending on the RF link condition. It is protected by 64/128-bit WEP security and Dynamic Key Encryption. Its overall dimensions are 2.125 x 4.625 x 0.7 inches [7].

The BEFW11S4 is a plug-and-play wireless-B broadband router. It has one

Figure 2.9: TableSat Wireless Access Point: OTC Wireless's ASR102.

10/100 RJ-45 port for broadband modem and four RJ-45 switched ports and is capable of up to 128-bit encryption. Its overall dimensions are 7.31 x 6.16 x 1.88 inches [6].

Figure 2.10: TableSat Router: Linksys's BEFW11S4 Wireless-B broadband router.

Chapter 3

TableSat Onboard Software

The TableSat software is stored onboard the PC/104 card, which acts as TableSat's Flight Processor. The software is written in C code and compiled using the GNU compiler, SCC. It consists of two programs: the *Run Time Program* and the *Calibration Program*. Each routine will be discussed in this chapter.

3.1 TableSat Run Time Program

The TableSat Run Time Program is responsible for the overall control of TableSat. It is composed from four parallel threads: the *Communications Thread*, the *State Estimator Thread*, the *Actuator Thread*, and the *Controller Thread*. These four threads run simultaneously at frequencies established by their respective software timers. Each thread will be discussed in detail below. A fifth thread, the *TableSat Main Thread*, has overall control of the TableSat Run Time Program. The user interfaces with the Run Time Program via the Communications thread, which communicates via ethernet to an offboard control program running under Mathwork's MATLAB and Simulink software. Through the Run Time Program and MATLAB, the user can control TableSat using the TableSat Graphical User Interface (GUI), which allows the user to set and change certain system parameters such as control rate, estimator rate, and actuator rate, controller description, and estimator description. The TableSat GUI and Simulink interfaces will be discussed in detail in Chapter 4.

3.1.1 TableSat Main Thread

The TableSat Main Thread is responsible for the overall control of the TableSat Main Routine. It initializes the A/D and D/A hardware on the Prometheus; sets the software timers used by the other four threads; sets the default parameters for the controller, state estimator, and actuators; initializes and starts the other four threads; and waits for those threads to finish before shutting down the Main Routine.

The first thing the Main Thread does is initialize the input and output channels on the Prometheus. The A/D converter is initialized to read sensor inputs from 0.0 to 5.0 Volts, since all the sensors on TableSat output voltages in that range. In addition, the A/D converter is set up to perform a scan of the eight A/D channels to which the sensors are attached and allocates memory in which to store the sensor readings. Finally, the two channels of the D/A converter attached to the actuators are initialized to output voltages from 0.0 to 10.0 Volts.

The Main Thread then reads in the calibration constants that will be nominally used to convert the raw sensor readings into meaningful engineering data. These calibration constants are stored in the *TS_CalibrationConstants.dat* file and can be modified by running the TableSat Calibration Routine.

Next, the Main Thread initializes the software timers used to set the rates of the other four threads. The default values for these rates are 100 Hz for the Communications, Actuator, and Estimator Threads, and 20 Hz for the Controller Thread. The Actuator, Estimator, and Controller rates can be modified using the TableSat GUI discussed in Chapter 4.

Once the software timers have been initialized, the Main Thread then initializes the default parameters for the controller, state estimator, and actuators. The onboard state vector, $\hat{\mathbf{x}}$ contains five states: θ_{CSS}, θ_{TAM}, ω, ν_1 and ν_2. θ_{TAM} is the rotation angle about TableSat's spin axis as measured by TableSat's magnetometer. An θ_{TAM} angle of zero degrees aligns TableSat's magnetometer X-axis with the Earth's magnetic X-axis. θ_{CSS} is the rotation angle about TableSat's spin axis as measured by the four CSS's. A θ_{CSS} angle of zero degrees aligns the corner of the hexagonal CSS head between CSS2 and CSS3 with the sun. ω is the angular velocity of TableSat as measured by the gyro. ν_1 and ν_2 are the tachometer feedback from the two fans.

The default controller is a simple Proportional-Derivative (PD) controller whose target is zero rate and an θ_{TAM} angle of zero degrees. The default state estimator simply passes through the raw sensor readings converted into engineering units. The default actuator type is continuous, which sends voltage commands directly to the fans from the PD controller. The controller, state estimator, and actuator parameters can be varied by the user. These variable options will be presented in the descriptions of each thread and discussed in detail in Chapter 4.

After setting the default parameters for the controller, state estimator, and actuators, the Main Thread creates the four other threads. These threads will continue to run until the user decides to turn off TableSat. After receiving a shut down command, the Main Thread will wait for the other four threads to end before stopping the TableSat Main Routine.

3.1.2 State Estimator Thread

The State Estimator Thread has three main responsibilities: performing the A/D scan of the eight sensors, converting the raw sensor readings into engineering units, and estimating the five states of the TableSat system at each time step. The state estimate is then passed to the Controller Thread for use in calculating the actuator voltages.

Conversion to Engineering Units

The output from the A/D scan is a raw sensor reading in digital counts for each of the eight sensors. Before the TableSat state estimate can be calculated, those sensor counts must first be converted back into voltages and then into meaningful engineering units. Each of the sensors outputs a raw voltage between 0 and 5 Volts. Recall from Chapter 2 that the A/D converter is a 16-bit converter, which outputs a digital count ranging between -32768 and 32767. To convert back to voltage, the following equation must be applied to each of the A/D sensor readings:

$$Volts = \{(Counts + 32768)/65536\} * 5.0 \tag{3.1}$$

Once the sensor counts have been converted back to voltages, they can be turned into meaningful engineering data using the calibration constants loaded by the Main Thread.

Gyro To convert the gyro measurement in volts, $GyroVolts$, to degrees per second, ω_{raw}, the following equation can be applied:

$$\omega_{raw} = Volts2Dps * (GyroVolts - GyroOffset) \tag{3.2}$$

Nominally, $Volts2Dps$ is a manufacturer's specified value used to convert the gyro measurement in volts to degrees per second. The nominal value was then adjusted through experimental determination by commanding a constant rate and using the magnetometer measurements to determine how fast TableSat was actually spinning. Recall from Chapter 2 that the gyro is bidirectional, but its output voltage varies between 0.0 and 5.0 Volts. The $GyroOffset$, therefore, is used to recenter the voltage such that zero voltage corresponds to zero rate. In other words, $GyroOffset$ is the voltage output from the gyro when TableSat is stationary. For a gyro with an output ranging between 0.0 and 5.0 Volts, the bias should be approximately 2.5 Volts. The bias, however, can vary depending on the age of the gyro, its current operational temperature, and many other factors. Because of this variation, the $GyroOffset$ can be recalculated periodically by running the TableSat Calibration Routine.

CSS To convert the CSS measurements into a raw θ_{CSS} angle, the raw CSS voltages must first be converted into degrees. Recall from Chapter 2 that the CSSs output voltages proportional to the cosine of the angle between the sun and the CSS diode. When the sun is directly illuminating the CSS diode, the CSS will output its maximum voltage. When the sun is roughly 80 degrees away from the CSS diode, the CSS will output its minimum voltage. Therefore, to convert to degrees the following equations are applied to each CSS voltage:

$$\begin{aligned} &CssNorm = CssVolts/CssMax \\ &CssDeg = \arccos(CssNorm) * Rad2Deg \end{aligned} \tag{3.3}$$

The raw CSS voltage is normalized by $CssMax$ in order to limit $CssNorm$ to between 0 and 1, where 1 indicates full brightness on the CSS. $CssMax$ varies for each CSS and will change depending on how bright the light source is and the ambient light around TableSat. $CssMax$, therefore, should be recalculated periodically by running the TableSat Calibration Routine.

Once the CSS voltages have been converted to angles between a given CSS and the sun, an $\theta_{CSS,raw}$ angle for TableSat can be determined using the following algorithm based on the previous θ_{CSS}, the CSS layout seen in Figure 3.1, which CSSs are illuminated, and which CSS sees the most sun, *i. e.* the CSS for which $CssDeg$ is smallest.

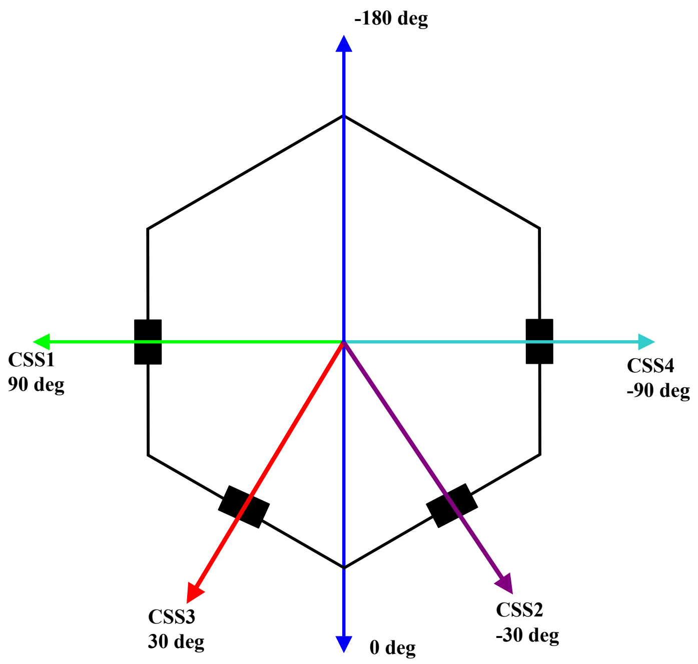


Figure 3.1: Coarse Sun Sensor (CSS) layout and $AzCss$ angles

The following algorithm describes how the four raw CSS angle readings are converted into a single estimate, $\theta_{CSS,raw}$, of TableSat's orientation with respect to the primary light source. The CSS layout and θ_{CSS} angles can be seen in Figure 3.1.

```
if($MinA >= 80.0$)
    if($lastAz > 0$)
        $\theta_{CSS,raw} = 180.0$
    else $\theta_{CSS,raw} = -180.0$
    endif

elseif($MinCss == 0$)
    if($CssDeg[2] > 60$)
        $\theta_{CSS,raw} = 90.0 + CssDeg[0]$
    else
        $\theta_{CSS,raw} = ((90.0 - CssDeg[0]) + (30.0 + CssDeg[2]))/2.0$
    endif

elseif($MinCss == 2$)
    if($CssDeg[0] > 60$)
        $\theta_{CSS,raw} = ((30.0 - CssDeg[2]) + (-30.0 + CssDeg[1]))/2.0$
    else
        $\theta_{CSS,raw} = ((30.0 + CssDeg[2]) + (90.0 - CssDeg[0]))/2.0$
    endif                                                                  (3.4)

elseif($MinCss == 1$)
    if($CssDeg[2] > 60$)
        $\theta_{CSS,raw} = ((-30.0 - CssDeg[1]) + (-90.0 + CssDeg[3]))/2.0$
    else
        $\theta_{CSS,raw} = ((30.0 - CssDeg[2]) + (-30.0 + CssDeg[1]))/2.0$
    endif

elseif($MinCss == 3$)
    if($CssDeg[1] > 60$)
        $\theta_{CSS,raw} = -90.0 - CssDeg[3]$
    else
        $\theta_{CSS,raw} = ((-90.0 + CssDeg[3]) + (-30.0 - CssDeg[1]))/2.0$
    endif

endif
$lastAz = \theta_{CSS,raw}$
```

where $MinA$ is the minimum $CssDeg$ among the four CSSs, $lastAz$ is the $\theta_{CSS,raw}$ angle from the previous time step, and $MinCss$ is the index number (0 through

3) of the CSS with the smallest $CssDeg$ angle.

TAM To convert the TAM measurements into a raw $\theta_{TAM,raw}$ angle, the raw TAM voltages must first be normalized by applying the following equation to each axis of TAM data:

$$TamNorm = TamGain * (TamVolts - TamBias) \tag{3.5}$$

Recall from Chapter 2 that the magnetometer outputs a voltage between 0.0 and 5.0 Volts for each of its three axes. $TamBias$ recenters $TamVolts$ such that the measurement will oscillate between a positive and negative amplitude. $TamGain$ then normalizes the TAM measurement such that each axis will range between +/-1.0. $TamBias$ and $TamGain$ should both be recalculated periodically by running the TableSat Calibration Routine. Nominally, $TamBias$ is about 2.5 Volts and $TamGain$ varies on an axis by axis basis and is usually about -2.0 Volts for the X-axis, -6.0 Volts for the Y-axis, and 1.0 Volt for the Z-axis.

Since TableSat is restricted to one-dimensional motion, the Z-axis magnetometer measurement can be assumed to be constant. A $\theta_{TAM,raw}$ angle for TableSat, therefore, can be determined as follows:

$$\theta_{TAM,raw} = -\text{arctan2}(TamNormY, TamNormX) \tag{3.6}$$

where arctan2 is a four-quadrant arc-tangent.

In addition to a $\theta_{TAM,raw}$ angle, the TAM measurements are also used to determine a magnetic unit vector for TableSat.

Fans Recall from Chapter 2 that the third wire in the fans can feedback a signal proportional to the fan speed. Code for converting the feedback signals into fan speed measurements would also be included in the state estimator thread. Currently, however, the circuitry has not been constructed to implement the fan speed feedback functionality.

State Estimation

Once the sensor voltage measurements have been converted into a raw state estimate, $\hat{\mathbf{x}}_{\mathbf{raw}}$, consisting of $\theta_{CSS,raw}$, $\theta_{TAM,raw}$, ω_{raw}, $\nu_{1,raw}$, and $\nu_{2,raw}$, the actual state estimate, $\hat{\mathbf{x}}$, can be calculated with the user-defined Estimator. The Estimator takes on the following form:

$$\begin{aligned} \mathbf{z}_{e,k} &= \mathbf{A_e}\mathbf{z}_{e,k-1} + \mathbf{B_e}\hat{\mathbf{x}}_{raw,k-1} \\ \hat{\mathbf{x}}_k &= \mathbf{C_e}\mathbf{z}_{e,k} + \mathbf{D_e}\hat{\mathbf{x}}_{raw,k} \end{aligned} \tag{3.7}$$

where $\mathbf{z_e}$ is the vector of Estimator states of length n_o,where n_o is the number of estimator states and can be greater than or equal to zero. $\hat{\mathbf{x}}_{\mathbf{raw}}$ is the 5x1 vector of raw state estimates, and $\hat{\mathbf{x}}$ is the 5x1 vector of state estimates that will be used by the Controller. The matrices $\mathbf{A_e}$, $\mathbf{B_e}$, $\mathbf{C_e}$, and $\mathbf{D_e}$ define the Estimator. They can be set by the user, but care must be taken to ensure that these matrices are of the correct dimension. $\mathbf{A_e}$ is of dimension n_oxn_o, $\mathbf{B_e}$ is of dimension n_ox5, $\mathbf{C_e}$ is of dimension 5xn_o, and $\mathbf{D_e}$ is of dimension 5x5. Ways of defining the Estimator matrices and some examples of different Estimator can be found in Chapter 4 and Chapter 7, respectively.

Autonomous Gyro Calibration

Also included in the state estimator thread are software “hooks” for advanced, autonomous gyro calibration. Currently this code is not implemented.

3.1.3 Controller Thread

The Controller Thread is responsible for using the current state estimate to calculate the voltage settings for the fans at each time step. The controller is defined using a set of six, user-defined matrices. The control law is of the form:

$$\begin{aligned} \mathbf{z}_{c,k+1} &= \mathbf{A_c}\mathbf{z}_{c,k} + \mathbf{B1_c}\hat{\mathbf{x}}_k + \mathbf{B2_c}\mathbf{x}_{d,k} \\ \mathbf{v}_{k+1} &= \mathbf{C_c}\mathbf{z}_{c,k} + \mathbf{D1_c}\hat{\mathbf{x}}_k + \mathbf{D2_c}\mathbf{x}_{d,k} \end{aligned} \tag{3.8}$$

where $\mathbf{z_c}$ is the vector of controller states of length n_c, where n_c is the number of controller states and can be any size. $\hat{\mathbf{x}}$ is the 5x1 vector of state estimates, $\mathbf{x_d}$ is the desired target, and $\mathbf{v}$ is the output to the actuators. $\mathbf{x_d}$ is always a 2x1 vector made up of the desired pointing angle (θ_d) and desired rate (ω_d). $\mathbf{v}$ is a vector of length one or two made up of the voltage commands to the fans. The default is to have $\mathbf{v}$ as length one. In this case, only one fan is commanded at any given time. If the controller output is positive, the positive fan is commanded. If the controller output is negative, the negative fan is commanded. The matrices $\mathbf{A_c}$, $\mathbf{B1_c}$, $\mathbf{B2_c}$, $\mathbf{C_c}$, $\mathbf{D1_c}$, and $\mathbf{D2_c}$ can be set by the user, but care must be taken to ensure these matrices are of the correct dimension. $\mathbf{A_c}$ is of dimension n_cxn_c, $\mathbf{B1_c}$ is of dimension n_cx5, $\mathbf{B2_c}$ is of dimension n_cx2, $\mathbf{C_c}$ is of dimension 1xn_c, $\mathbf{D1_c}$ is of dimension 1x5, and $\mathbf{D2_c}$ is of dimension 1x2, assuming v is of length one. Ways of defining these matrices and some examples of different Controllers are discussed in Chapter 4 and Chapter 7, respectively.

The desired target, $\mathbf{x_d}$ can be established in one of two ways. The user can define either a constant target, *i. e.* a desired angle or rate, or the user can define a tracking target, *i. e.* a sine wave with a specified frequency (ω) and amplitude (A). If the desired target is constant, $\mathbf{x_d}$ is also constant. If the desired target is a tracking target, $\mathbf{x_d}$ varies as a function of time and the Controller Thread calculates a new desired target at each time step using the following equations:

$$\begin{aligned} \mathbf{x_d}(0) &= A\sin\left(\omega * (t - t0)\right) \\ \mathbf{x_d}(1) &= A\omega\cos\left(\omega * (t - t0)\right) \end{aligned} \tag{3.9}$$

where t is the current time and $t0$ is the time at which TableSat control was started.

In addition to the above control law, the user also has the option of compensating for the friction in the TableSat system. When friction compensation is turned on, the user can define a friction compensation curve. The Controller

Thread will calculate the additional voltage required to compensate for the friction based on the defined friction compensation curve. How to define the friction compensation curve and examples of different curves are discussed in Chapters 4 and 7.

3.1.4 Actuator Thread

The Actuator Thread is responsible for taking the fan voltages calculated in the controller thread and applying those inputs to the fans. Recall from Chapter 2 that one fan is mounted to spin TableSat in a positive direction and one fan is mounted to spin TableSat in a negative direction, but that both fans only accept positive voltages from 0 to 12 Volts. The Controller will calculate both positive and negative voltages. The Actuator Thread must ensure that any negative commands from the Controller are converted into positive voltages applied to the negative fan. In addition, the Actuator Thread must ensure that the Controller commands do not exceed the maximum voltage of 12 Volts that the fans can accept. For commanded voltages within the static friction level of the fans, both fans are commanded simultaneously to compensate for the fan static friction. Otherwise, the Actuator Thread must also ensure that only one fan is commanded at a time, which ensures that the TableSat battery is not unnecessarily drained. The specifics of the fan static friction compensation are discussed in Chapter 6. The actuators can operate in one of three, user-selected actuation modes: Continuous Actuation, Bang-Bang Actuation, or PWM Actuation. How to change between different Actuation modes is discussed in Chapter 4.

Continuous Actuation

In Continuous Actuation mode, the actual analog voltages calculated in the control thread are applied directly to the fans. The minimum change in voltage is proportional to the resolution of the D/A converter. Recall from Chapter 2 that the resolution of a unipolar output is 2.44 mV. Recall also that the output from the D/A converter goes through a voltage regulator to scale the 10 Volts maximum D/A output to the 12 Volts that the fans can accept. The minimum output from the D/A converter will be likewise scaled such that the minimum change in voltage that can be applied to the fans is 2.93 mV.

Bang-Bang Actuation

In Bang-Bang Actuation mode, the fans are commanded to either full on (12 Volts) or off (0 Volts). If the voltage calculated by the Controller is positive and greater than the user-set deadzone, the positive fan is set to full on. If the voltage calculated by the Controller is negative and greater than the user-set deadzone, the negative fan is set to full on. The deadzone is a band of voltage centered about 0 Volts, the width of which can be set by the user. The fans will not be turned on if the output from the Controller is less than the deadzone.

PWM Actuation

In PWM, or Pulse Width Modulation, Actuation mode, the control authority calculated by the Controller is converted into a duty cycle according to the following calculation:

$$\begin{array}{l} numPos = (FanVoltage(0)/VMAXFAN) * FreqRatio \\ numNeg = (FanVoltage(1)/VMAXFAN) * FreqRatio \end{array} \tag{3.10}$$

where $numPos$ and $numNeg$ are the number of positive and negative intervals within one Controller cycle, respectively. $numPos$ and $numNeg$ are always integers. If Equation 3.10 does not yield an integer value, only the integer part of the result is assigned to $numPos$ and $numNeg$. $VMAXFAN$ is the maximum fan voltage that can be commanded. In the case of TableSat, that maximum voltage is 12 Volts. $FreqRatio$ is the integer part of the ratio of the Actuator rate to the Controller rate. This number tells how many complete Actuator cycles there are in one Controller cycle, and must be greater than or equal to one, which implies that the Controller rate must be greater than or equal to the Actuator rate. As an example, for the default sample rate settings, the controller thread runs at 100 Hz and the actuator thread runs at 20 Hz. $FreqRatio$, therefore, is equal to 5 and the PWM actuation produces levels of duty cycle resolution equal to 0%, 20%, 40%, 60%, 80%, or 100%. If $FreqRatio$ is equal to one, then PWM Actuation becomes the same as Bang-Bang Actuation without a deadzone.

Only one of $numPos$ or $numNeg$ can be greater than zero during any one Controller cycle. Unlike the Continuous Actuation mode, only one fan can ever be commanded at any given time. If $numPos$ is greater than zero, the positive fan will be commanded full on (12 Volts) for $numPos$ Actuator cycles. If $numPos$ is less than $FreqRatio$, the positive fan will be commanded off (0 Volts) for the remaining Actuator cycles within one Controller cycle. The same logic applies to $numNeg$ and the negative fan. When $FreqRatio$ Actuator cycles have passed, the Controller will have updated $FanVoltage$ and Equation 3.10 gets recalculated.

3.1.5 Communications Thread

The Communications Thread is responsible for communication between the user and TableSat. TableSat communications are implemented using the User Datagram Protocol (UDP), which is a connectionless transport layer protocol that belongs to the Internet Protocol (IP) family. UDP is basically a simple interface between IP and higher level processes. It uses datagrams (packets of raw bytes) to exchange information over the ethernet between computers. In the case of TableSat, UDP is used to send data back and forth between the MATLAB GUI and the TableSat Run Time Process. UDP packets are unstructured and have no delivery guarantees or error checks. To simplify the structure and add minimal error checks for UDP datagrams, the Space Systems Lab at the University of Maryland has implemented a package known as DSMP (Distributed Systems

Messaging Protocol), which is intended to extend the lab's previous PIVECS standard for serial communications to the ethernet. In DSMP, each UDP packet consists of a header byte and an arbitrary amount of data (up to the UDP packet limit of 64 K). Each header can be bound to a handler function using the command *dsmp_RegisterMesg*.

The Communications Thread is responsible for handling requests to send the state or raw data to MATLAB or Simulink for display. When a request for data comes in from the user, the *dsmp_* functions are used to assemble the UDP header and data packet containing the requested data. The packet is then sent to the user via the wireless access point. When the user wishes to change any of the variable TableSat parameters, a UDP message is sent to TableSat with the desired change and any new data the change requires. The *dsmp_RecvMesg* function reads the message, extracts the header, and passes the the data to the correct thread. Messages can be sent by the user using either the TableSat GUI discussed in Chapter 4 or via MATLAB command line prompts, which are also discussed in Chapter 4. Each variable TableSat parameter has a unique message number, which identifies to TableSat the exact parameter to be changed. Furthermore, each unique message number may also have additional data identifying the new parameters. When the user is sending messages to TableSat, care must be taken to send the correct data with the correct message number. All of the variable TableSat parameters and their corresponding message numbers and data packet descriptions can be found in Table 3.1.

TableSat Messages			
Msg. Num	Meaning	Msg. Data	dsmp_ Handler
2	Send Message Acknowledged	byte	SendMesg
4	Set Run Mode	byte	RecvMesg
6	Set Control Mode	byte	RecvMesg
8	Set Fan Mode	byte	RecvMesg
10	Set Actuator Mode	byte	RecvMesg
12	Set Mode	byte	RecvMesg
15	Set Controller Data	double	RecvMesg
21	Set Fan Friction Data	double	RecvMesg
24	Set Controller Target	double	RecvMesg
27	Set Estimator Data	double	RecvMesg
30	Set Gyro Auto-Calibration Data	double	RecvMesg
33	Set Sample Rates	double	RecvMesg
60	Send Current State Data	double	SendMesg
63	Send Raw Sensor Data	double	SendMesg
66	Send Status Flags	byte	SendMesg
81	Request State Data	byte	RecvMesg
84	Request Raw Sensor Readings	byte	RecvMesg
87	Request Status Data	byte	RecvMesg
99	Request Sensor Recalibration	byte	RecvMesg

Table 3.1: TableSat Messages

3.2 TableSat Calibration Routine

At any time, the user can run the *Calibration Routine.* The calibration routine takes approximately 5 minutes to complete. During the first 20 seconds of the routine, TableSat is held stationary. At the end of the stationary period, the positive fan is powered on full for 10 seconds to spin up TableSat. Once TableSat is spinning, the fan power is decreased until TableSat is spinning at a roughly constant rate, where it remains for the remainder of the calibration routine.

The calibration routine is used to determine the calibration constants for each of the sensors. As was seen previously, these calibration constants are used to convert the voltages obtained from each sensor into meaningful engineering data. The calibration routine should be run each time TableSat is initially set up, or any time the system is moved. The following subsections discuss how each of the sensor calibration constants are determined.

3.2.1 Gyro

Recall from Section 3.1.2 that there are two calibration constants needed to convert the gyro voltage reading into the TableSat spin rate in degrees per second, $GyroBias$ and $Volts2Dps$. $GyroBias$ is calculated by averaging the gyro voltage readings during the 20 second stationary period of the Calibration Routine. $Volts2Dps$ is an experimentally determined value and is not recalculated during the Calibration Routine. It is determined by controlling TableSat at a constant rate and looking at the time it takes to move from one peak of the CSS voltages to the next peak.

3.2.2 Coarse Sun Sensors

Recall from Section 3.1.2 that $CssMax(i)$ is the one calibration constant needed to convert a given CSS voltage into an angle measurement. For each CSS, $CssMax(i)$ is determined by looking at the voltages of the CSS's as TableSat is spinning. The Calibration Routine compares the current CSS voltage reading to its current maximum. If the current reading is greater than the current maximum, that reading becomes the new maximum. At the end of the Calibration Routine, the maximum readings for each CSS become $CssMax(i)$.

3.2.3 Three Axis Magnetometer

Recall from Section 3.1.2 that two calibration constants are needed to convert the magnetometer voltages into normalized measurements that can be used to find θ_{TAM}: $TamBias$ and $TamGain$. For each axis of the magnetometer, $TamBias$ is found by averaging the voltage readings from that axis of the TAM while TableSat is spinning. For simplicity, $TamGain(2)$, the gain for the Z-axis of the TAM, is always set to 1.0, as this reading does not vary during the (nominally) planar rotations of TableSat. This gain is chosen because TableSat is located on the Earth, and the Z-axis magnetometer reading will always be constant.

Calculation of $TamGain(0)$ and $TamGain(1)$ is slightly more complicated. While the Calibration Routine is running, the voltage readings from the X- and Y-axes of the TAM are saved to a file. At the end of the routine, those voltages are read, one at a time, and the $TamBias$ is subtracted from the voltage readings. This subtraction recenters the voltages such that they now oscillate between a maximum and minimum amplitude. Recall from Section 3.1.2 that $TamGain$ is used to scale the X- and Y-axis recentered voltages such that they oscillate between +/- 1.0. If the absolute value of the X-axis recentered voltage is close to zero, then the Y-axis is near its amplitude. The absolute value of that Y-axis recenterd value gets included in a running average called $SemiMinor$. If the absolute value of the Y-axis recentered voltage is close to zero, then the X-axis is near its amplitude. The absolute value of that X-axis recentered voltage gets included in a running average called $SemiMajor$. After all the TAM data has been read in, $SemiMinor$ is the average of the Y-axis amplitudes, and $SemiMajor$ is the average of the X-axis amplitudes. $TamGain(0)$ then becomes the inverse of $SemiMajor$ and $TamGain(1)$ becomes the inverse of $SemiMinor$. So, when the recentered voltage of either axis is multiplied by its corresponding $TamGain$, the result will oscillate between +/- 1.0.

Chapter 4

TableSat Interface GUI

TableSat is an extremely versatile tool. It has many variable parameters that can be set and controlled by the user to establish an infinite number of ways to control TableSat. In addition, the user also has the option of controlling TableSat using the on-board flight processor, or by closing the control loop through Simulink to allow more complex control algorithms than those implemented by the onboard software. To fully utilize TableSat's versatility, the user must be able to effectively communicate with TableSat. The user needs to be able to change TableSat's user-definable parameters, start and stop TableSat, and get state information and data back from TableSat while it is running. In addition, the user also needs to be able to recalibrate TableSat when necessary.

The primary interface to TableSat is via a combination of MATLAB and Simulink GUIs. Both of these graphical user interfaces depend upon a set of low-level MATLAB functions which exchange data with the communications thread on the TableSat Run Time Process using the established UDP messaging structure. While it is possible to feed raw commands to TableSat from the MATLAB command line, this method of communication is not recommended.

4.1 Low Level Communication

A set of MATLAB functions was created to implement a DSMP-like environment in MATLAB using the publicly available PNET package, which allows MATLAB to directly communicate over an ethernet link. PNET is part of the TCP/UDP/IP Toolbox 2.0.5, and can be downloaded from Mathwork's MATLAB Central file exchange. Recall from Chapter 3 that the TableSat communications thread is a UDP-based protocol, meaning data is sent to and received from TableSat in packets via the wireless access point. The following sections briefly describes each of these low-level functions.

4.1.1 tsat_init

Before any messages can be sent to or received from TableSat, tsat_init must be used to initialize the communications port with TableSat. The function has two inputs, the port number and the IP address of TableSat. The default port on which TableSat listens for messages is 9877. The function returns the socket number, which is then used as an input to both *tsat_send_msg* and *tsat_recv_msg*.

4.1.2 tsat_send_msg

The MATLAB function, *tsat_send_msg* is used to send messages to TableSat. It is called only as needed to alter TableSat parameters or request additional data from TableSat. It takes three inputs: the message number, an integer; the data, a structure or cell array depending on the type of data sent; and the socket number returned by *tsat_init*. There are three main types of messages that can be sent to TableSat: variable parameters messages, state change messages, and data request messages.

The variable parameters messages are listed in Table 4.1. These messages are sent to TableSat informing it of any changes in its variable parameters. They are the same messages that are automatically sent to TableSat when the TableSat GUI is used to change the variable TableSat parameters. They correspond directly to the GUI subsections discussed above.

TableSat Variable Parameter Messages		
Msg. Num	Action	Required Data
8	Set fan/friction mode	Cell array with 8-bit integer: 0 = Friction or fan speed compensation off 1 = Friction/fan speed compensation on
10	Set actuator type	Cell array with 8-bit integer: 0 = Continuous actuation 1 = PWM actuation 2 = Bang-Bang actuation
12	Set gyro auto-calibration	Cell array with 8-bit integer: 0 = Off 1 = On
15	Set controller definition	Structure with doubles: A, B1, B2, D1, D2 matrices
18	Set fan speed compensation curve	Cell array with doubles: Number of compensation points Vector of fan speeds, Vector of fan force values
21	Set friction compensation curve	Cell array with doubles: Number of compensation points, Vector of angular velocities, Vector of compensation voltages
24	Set controller target	Cell array with doubles: Mode (0 = Stationary, 1 = Sine Wave), Vector with $\mathbf{x_d}$ ($\theta_\mathbf{d}$ & $\omega_\mathbf{d}$) or (A & ω)
27	Set estimator definition	Structure with doubles: A, B, C, D matrices
30	Set gyro auto-calibration gains	Cell array with doubles: P-Gain, D-Gain
33	Set sample rates	Cell array with doubles: Control rate, Estimator rate, Actuator rate

Table 4.1: TableSat Variable Parameter Messages

Table 4.2 lists the state change and data request messages that can be sent to TableSat using *tsat_send_msg*. State change messages are those that tell TableSat to change its operational state, for example changing from closed loop to open loop control, or shutting down TableSat. Data request messages are messages that request information such as the state estimate or raw sensor readings from TableSat. Using *tsat_send_msg* with data request messages must be followed with a call to *tsat_recv_msg* in order to receive the data that TableSat sends.

4.1.3 tsat_register_msg

The MATLAB function *tsat_register_msg* is used to bind the message headers to their respective handler functions.

4.1.4 tsat_recv_msg

The MATLAB function *tsat_recv_msg* is used to receive messages from TableSat. It reads the packet header, calls the handler bound by *tsat_register_msg*, and passes any included data. It takes only one input, the TableSat socket number returned by *tsat_init*. It returns the message number and data sent by TableSat. *tsat_recv_msg* is called continuously while TableSat is running in order to allow the passing of the state data from TableSat for real time plotting. Table 4.3 lists the message numbers and data that can be received from TableSat.

Additional Messages Sent to TableSat		
Msg. Num	Action	Required Data
4	Set run mode	Cell array with 8-bit integer: 0 = Shut down TableSat 1 = Run TableSat
6	Set control mode	Cell array with 8-bit integer: 0 = Off (no fan voltages commanded) 1 = On (close control loop on board TableSat)
81	Request state data	Cell array with 0
84	Request raw sensor readings	Cell array with 0
87	Request status data	Cell array with 0
99	Request sensor recalibration	Cell array with 0

Table 4.2: Additional messages sent to TableSat

Messages Received from TableSat		
Msg. Num	Action	Required Data
2	Receive acknowledgements	None
60	Receive current state data	$TimeStamp$, θ_{CSS}, θ_{TAM}, ω, ν_1, ν_2
63	Receive raw sensor data	$TimeStamp$, $Css1Volts$, $Css2Volts$, $Css3Volts$, $Css4Volts$, $GyroVolts$, $MagXVolts$, $MagYVolts$, $MagZVolts$, $Fan1Counts$, $Fan2Counts$
66	Receive status flags	$TSrunMode$, $TScontrolMode$, $TSfanMode$, $TSactuatorMode$, $TSgyroMode$

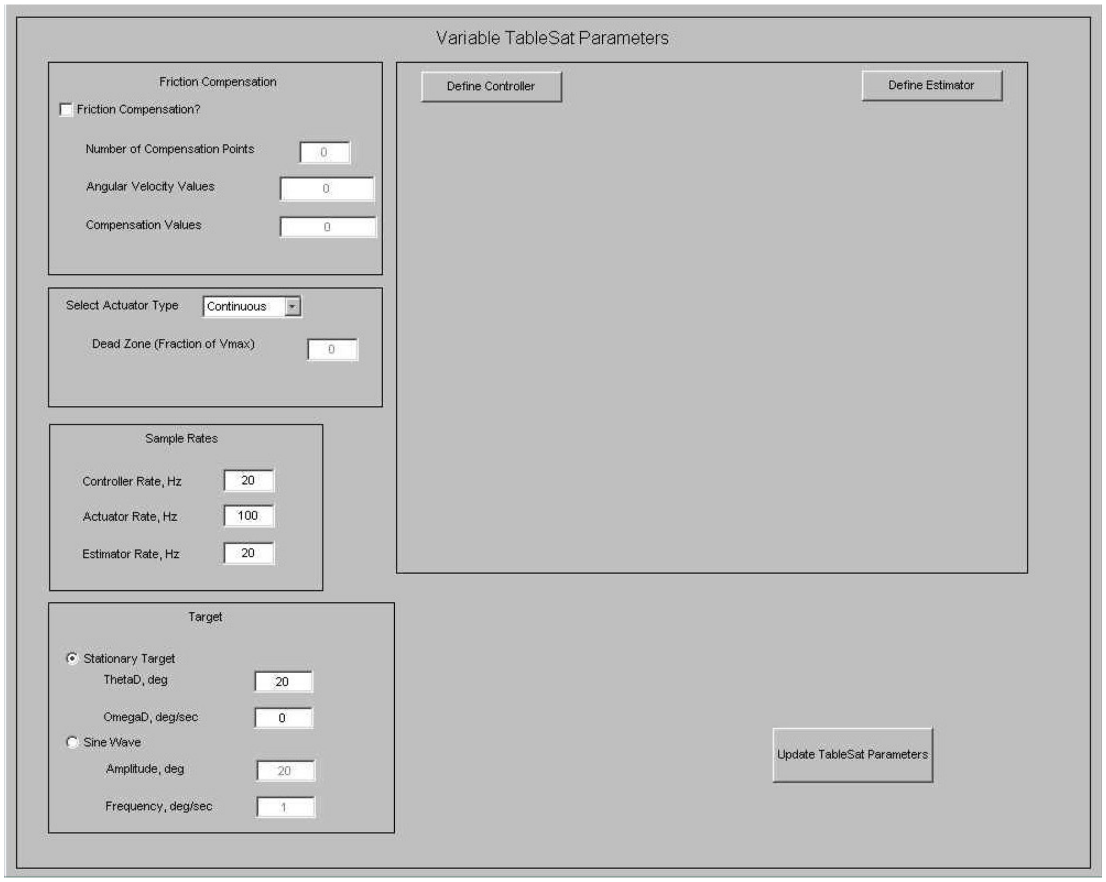
Table 4.3: Messages received from TableSat

4.2 TableSat GUI

The easiest method to communicate parameter changes to TableSat is through the TableSat Graphical User Interface (GUI), which runs using MATLAB. The GUI is divided into several sections, each of which will be explained in detail in the sections below. A snap-shot of the TableSat GUI can be seen in Figure 4.1. On the top left corner of the TableSat GUI is the *Friction Compensation* section. Below Friction Compensation is the *Actuator Type* section, followed by the *Sample Rates* section. Below Sample Rates is the *Target* section. On the right side of the TableSat GUI are the *Controller Definition* and *Estimator Definition* sections. Only one of these sections is visible at a time, and the user can toggle between them by selecting the correct button on the GUI. Finally, on the bottom right of the GUI is the *Update TableSat Parameters* button. When this button is pressed, the GUI sends messages to TableSat informing it of any parameter changes made by the user.

4.2.1 Friction Compensation

The Friction Compensation section of the TableSat GUI can be seen in Figure 4.2. The user has two main options in the Friction Compensation section; Friction Compensation can be turned on or off. The default is to have Friction Compensation off, or the box on the TableSat GUI unchecked. When the Friction Compensation box is checked, Friction Compensation is turned on and three addi-



Variable TableSat Parameters

Friction Compensation

☐ Friction Compensation?

Number of Compensation Points 0

Angular Velocity Values 0

Compensation Values 0

Select Actuator Type Continuous

Dead Zone (Fraction of Vmax) 0

Sample Rates

Controller Rate, Hz 20

Actuator Rate, Hz 100

Estimator Rate, Hz 20

Target

● Stationary Target

ThetaD, deg 20

OmegaD, deg/sec 0

○ Sine Wave

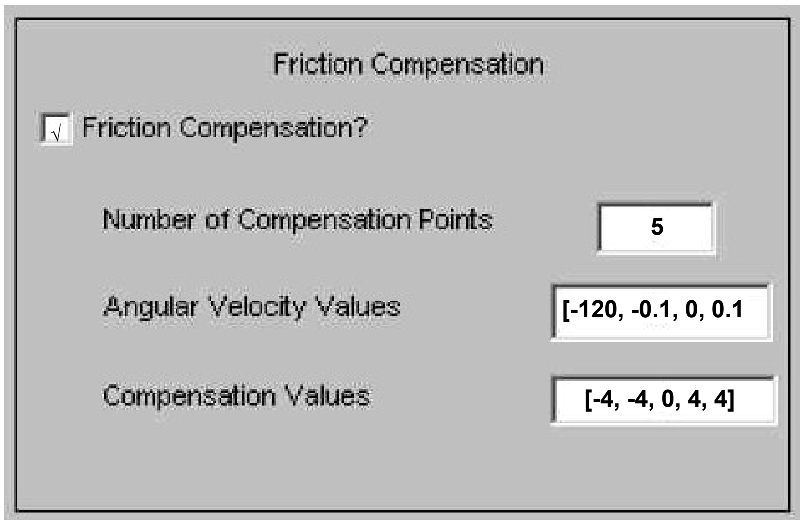
Amplitude, deg 20

Frequency, deg/sec 1

Figure 4.1: TableSat Graphical User Interface. Used to change variable TableSat Parameters.

tional parameters become highlighted: *Number of Compensation Points*, *Angular Velocity Values*, and *Compensation Values.* These three parameters define the friction compensation curve that will be used to calculate the additional voltage needed to compensate for the friction in the TableSat system.

Number of Compensation Points indicates the number of points on the Voltage *vs.* TableSat Angular Velocity plot that will be used to define the friction compensation curve. *Angular Velocity Values* (in degrees per second) and *Com-*


Friction Compensation

[✓] Friction Compensation?

Number of Compensation Points: 5

Angular Velocity Values: [-120, -0.1, 0, 0.1

Compensation Values: [-4, -4, 0, 4, 4]

Figure 4.2: TableSat GUI: Friction Compensation section. Default setting has Friction Compensation unchecked and Friction Compensation curve boxes greyed out.

pensation Values (in Volts) are the corresponding values that actually define the friction compensation curve. They are vectors whose lengths are equal to the *Number of Compensation Points*. The friction compensation curved is actually defined by connecting the points defined by the (Angular Velocity, Compensation Values) pairs.

As an example, suppose the user wants to use the friction compensation curve shown in Figure 4.3. In this case, $Number of Compensation Points$ equals five. $Angular Velocity Values$ is a vector of length five, whose values are (-120, -0.1, 0, 0.1, 120) deg/sec. $Compensation Values$ is also a vector of length five, and its values are (-4, -4, 0, 4, 4) Volts. TableSat will use the curve defined by this data, with linear interpolation between points, to compensate for the friction

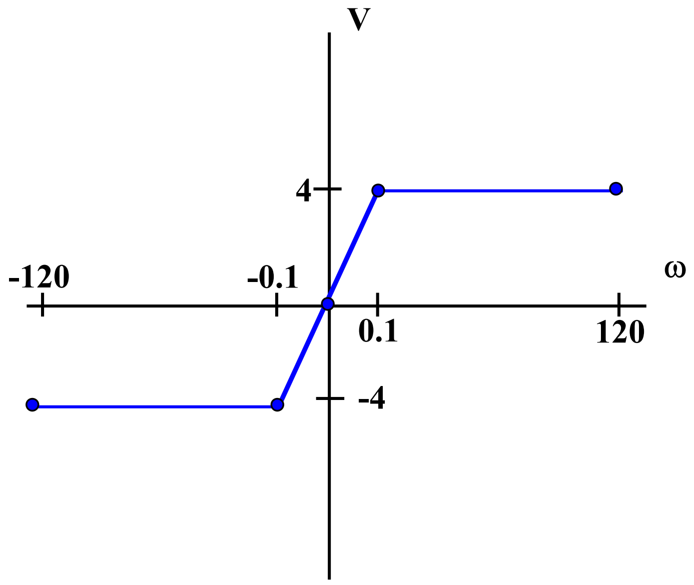
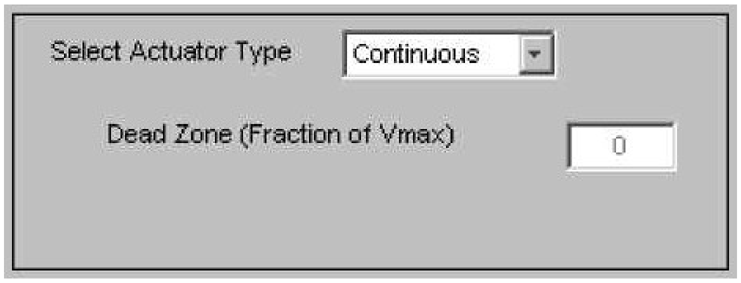


Figure 4.3: Example of a friction compensation curve. *Number of Compensation Points* equals five, *Angular Velocity Values* = (-120, -0.1, 0, 0.1, 120)deg/sec, *Compensation Values* = (-4, -4, 0, 4, 4)V

in the system.

4.2.2 Actuator Type

The Actuator Type section of the TableSat GUI is pictured in Figure 4.4. The user can select from the pull down menu one of the three actuator types discussed in Chapter 3. Recall from Chapter 3 that the three possible actuation types are Continuous, Bang-Bang, and PWM. If Bang-Bang actuation is chosen, the *Dead Zone* box becomes active. The user can then enter the desired size of the actuation dead zone as a fraction of the 12 Volt maximum voltage.


Select Actuator Type: Continuous

Dead Zone (Fraction of Vmax): 0

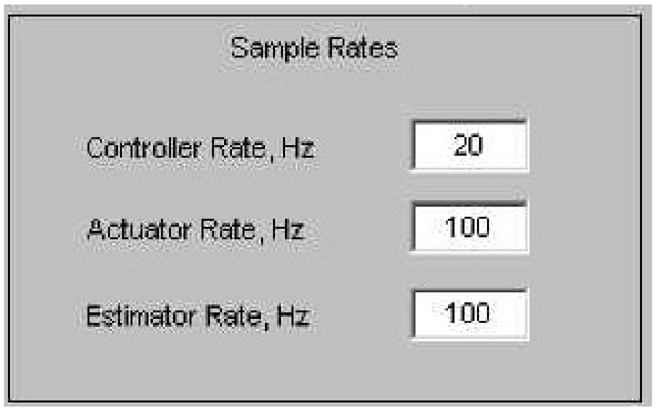
Figure 4.4: TableSat GUI: Actuator Type section. Default setting has Continuous actuation selected.

4.2.3 Sample Rates

The Sample Rates section of the TableSat GUI can be seen in Figure 4.5. In this section, the user can set the sample rates, in Hz, for the Controller, Actuator, and Estimator threads. Sample rates are limited only by the processor speed of the TableSat flight processor. The maximum sample rate is approximately 200 Hz. The Actuator rate must be greater than or equal to the Controller rate. The default rates for the Controller, Actuator, and Estimator threads are 20 Hz, 100 Hz, and 100 Hz, respectively.

4.2.4 Target

The Target section of the TableSat GUI is pictured in Figure 4.6. In this section of the GUI, the user can define the target at which TableSat will point. The target can either be a *Stationary Target*, or a *Sine Wave*. If the selected target is stationary, the user can define either an off-point angle (θ_d), in degrees, or a spin

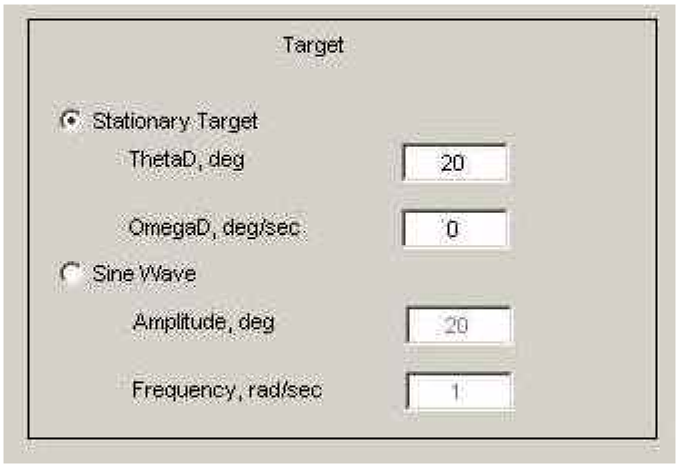


Sample Rates

Controller Rate, Hz	20
Actuator Rate, Hz	100
Estimator Rate, Hz	100

Figure 4.5: TableSat GUI: Sample Rates section. Default setting has sample rates of 20 Hz, 100 Hz, and 100 Hz, for the Controller, Actuator, and Estimator rates, respectively.

rate (ω_d), in degrees per second. Only one of these values should be non-zero at any given time. If the desired target is a sine wave, the user must input the Amplitude, A, in degrees, and Frequency, ω, in radians per second, of the desired sine wave. The default target is a Stationary Target with a θ_d of 20 degrees.



Target

[✓ Stationary Target]

ThetaD, deg	20
OmegaD, deg/sec	0

Sine Wave

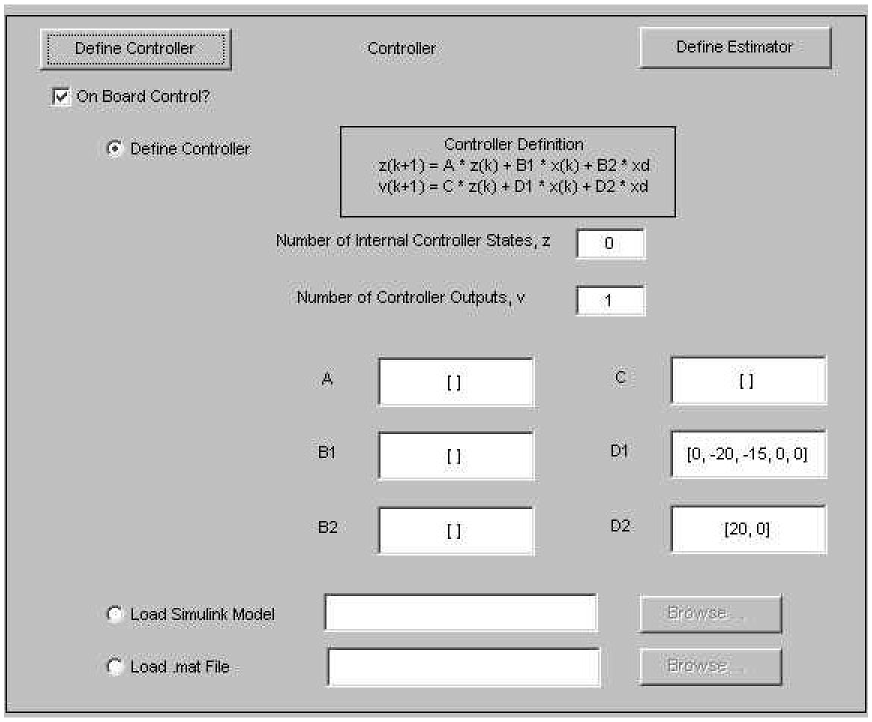
Amplitude, deg	20
Frequency, rad/sec	1

Figure 4.6: TableSat GUI: Target section. Default setting has a stationary target with a θ_d of 20 deg.

4.2.5 Controller Definition

The Controller Definition section of the TableSat GUI is pictured in Figure 4.7. The panel comes up when the "Define Controller" button on the GUI is pressed. In this section of the GUI, the user can define the Controller TableSat will use to control itself. The controller can be defined in one of three ways: with matrices, with a Simulink model, or with a MATLAB file.

The first method of controller definition uses explicit matrix definitions. When the *Define Controller* box is selected, the user can type the desired **A**, **B1**, **B2**, **C**, **D1**, and **D2** matrices directly into the boxes on the GUI. In addition,



[✓] On Board Control?

[● Define Controller]

Controller Definition
z(k+1) = A * z(k) + B1 * x(k) + B2 * xd
v(k+1) = C * z(k) + D1 * x(k) + D2 * xd

Number of Internal Controller States, z: 0

Number of Controller Outputs, v: 1

A	[]	C	[]
B1	[]	D1	[0, -20, -15, 0, 0]
B2	[]	D2	[20, 0]

○ Load Simulink Model

○ Load .mat File

Figure 4.7: TableSat GUI: Controller definition section. Default setting has a simple feedback PD controller with Kp = 20 and Kd = 15

the user must also define the number of controller states, n_c, and the number of controller outputs, p_c. Recall from Chapter 3 that the number of controller outputs can be either one or two and the number of controller states can be anything greater than or equal to zero. Furthermore, the length of the estimated state vector, $\hat{\mathbf{x}}$, is always five (θ_{CSS}, θ_{TAM}, ω, ν_1, ν_2). Care must be take by the user to ensure that the dimensions of the matrices match the length of the state vectors (**A** is of dimension n_cxn_c, **B1** is of dimension n_cx5, **B2** is of dimension n_cx2, **C** is of dimension p_cxn_c, **D1** is of dimension p_cx5, and **D2** is of dimension p_cx2). The GUI will not allow a mismatch of dimensions.

The second method of controller definition allows the user to use a stored Simulink model containing the desired controller. When the *Load Simulink Model* box is selected, the user can load a Simulink .mdl file containing the desired controller. The name of the file with the correct path can be typed directly into the box on the GUI, or the *Browse* button can be used to search for any .mdl files. An example of such a controller is pictured in Figure 4.8. This controller is a simple feedback PD controller. When using this type of controller definition, the user must make sure that the Simulink model has two inputs, $\mathbf{x_d}$ (the target) and $\hat{\mathbf{x}}$ (the state), and one output (the actuator commands). $\mathbf{x_d}$ must be a vector of length two and x must be a vector of length five, corresponding to the state (θ_{CSS}, θ_{TAM}, ω, ν_1, ν_2). Any corresponding gains must be of the correct dimension. As before, the output, which corresponds to the actuator commands, must be of length one or two. The GUI will then compute the **A**, **B1**, **B2**, **C**, **D1**, and **D2** matrices needed to implement the controller from the Simulink model.

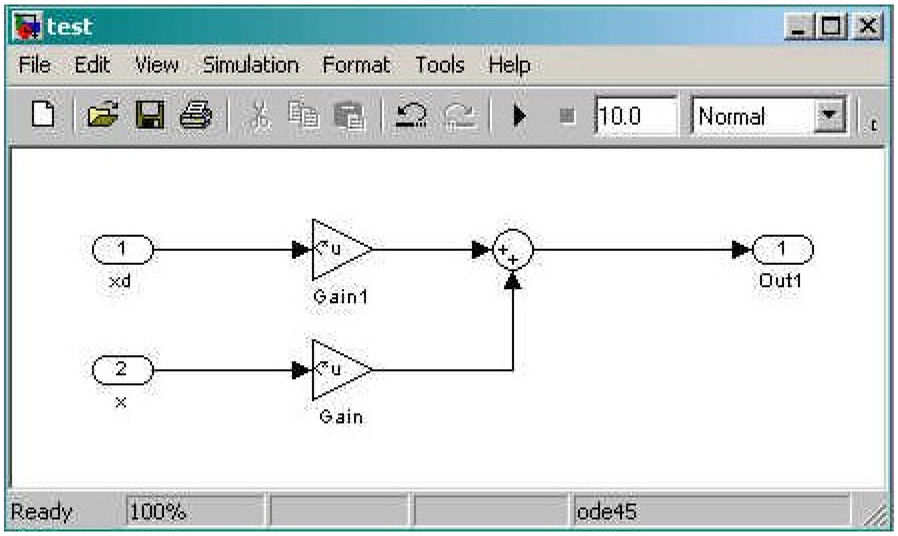


Figure 4.8: Example of a saved Simulink model of a simple feedback PD controller.

The final method of controller definition allows the user to use a controller stored in a MATLAB file. When the *Load .mat File* box is selected, the user can load a .mat file containing the desired controller. As with the Simulink model, the name of the file with the correct path can be typed directly into the box on the GUI, or the *Browse* button can be used to search for any .mat files. When using this type of controller definition, the user must make sure that the controller definition is stored in the correct format. The controller must be a structure called TS_Con. The format of the structure is as follows:

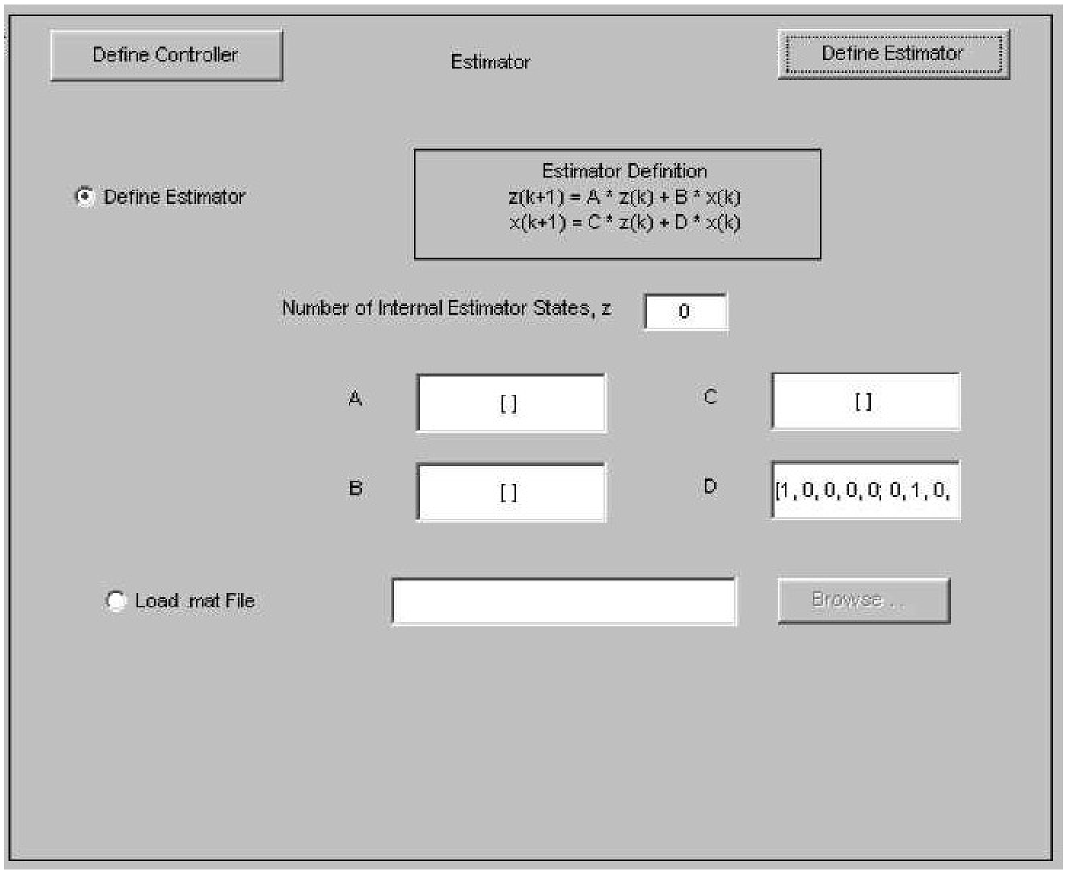
$$\begin{array}{l} TS\_Con.dims = [n_c, 5, 2, p_c, 5, 2] \\ TS\_Con.A = \mathbf{A} \\ TS\_Con.B1 = \mathbf{B1} \\ TS\_Con.B2 = \mathbf{B2} \\ TS\_Con.C = \mathbf{C} \\ TS\_Con.D1 = \mathbf{D1} \\ TS\_Con.D2 = \mathbf{D2} \end{array} \tag{4.1}$$

where n_c is the number of controller states, p_c is the number of controller outputs, and **A**, **B1**, **B2**, **C**, **D1**, and **D2** are the controller matrices. The structure must then be saved in a .mat file. The name of the file can be anything the user desires.

4.2.6 Estimator Definition

The Estimator definition of the TableSat GUI is shown in Figure 4.9. In this section of the GUI, the user can define the estimator that will be used by TableSat to estimate the state of the system. The Estimator can be defined in one of two ways: with matrices or with a MATLAB file.

The first method of estimator definition uses explicit matrix definitions. When the *Define Estimator* box is selected, the user can type the desired estimator matrices directly into the boxes on the GUI. In addition, the user must also enter the number of internal estimator states, n_o. Recall from Chapter 3 that the number of estimator states can be anything greater than or equal to zero, but that the state vector is always of length five. Care must be take by the user to ensure that the dimensions of the matrices match the lengths of the state vectors (**A** is of dimension n_oxn_o, **B** is of dimension n_ox5, **C** is of dimension 5xn_o, and



Define Estimator [✓]

Estimator Definition
z(k+1) = A * z(k) + B * x(k)
x(k+1) = C * z(k) + D * x(k)

Number of Internal Estimator States, z: 0

A: [ ]
C: [ ]
B: [ ]
D: [1, 0, 0, 0, 0; 0, 1, 0,

Load .mat File

Figure 4.9: TableSat GUI: Estimator definition section. Default setting has a simple pass-through estimator.

D is of dimension 5x5). The GUI will not allow a mismatch in dimensions.

The second method of estimator definition allow the user to use an estimator stored in a MATLAB file. Similar to the Controller Definition, if the *Load .mat File* box is selected, the user can load a .mat file containing the desire estimator. The file name with correct path can be typed directly into the box on the GUI, or the *Browse* button can be used to search for .mat files. When using this type of estimator definition, the user must ensure that the estimator is saved in the correct format. The estimator must be defined as a structure, TS_Est, whose format is as follows:

$$\begin{array}{l} TS\_Est.dims = [n_o, 5, 5, 5] \\ TS\_Est.A = \mathbf{A} \\ TS\_Est.B = \mathbf{B} \\ TS\_Est.C = \mathbf{C} \\ TS\_Est.D = \mathbf{D} \end{array} \tag{4.2}$$

where n_o is the number of internal observer states and **A**, **B**, **C**, and **D** are the estimator matrices. The estimator must then be saved as a .mat file. The name of the file can be anything the user desires.

4.2.7 Update TableSat Parameters

When the user has finished entering changes in the TableSat GUI, the *Update TableSat Parameters* button is pressed. When this button is pressed, a series of messages is distributed to TableSat, informing TableSat of any changes in its parameters. Recall from Chapter 3 that these messages are received and parsed within the Communications thread, and the data received is sent to the correct threads. Each variable TableSat parameter has one or more messages that are specific to that parameter. These messages are discussed in detail in Section 4.1. The GUI automatically sends the correct messages and data for any parameters changed in the GUI.

4.3 Simulink Block Diagrams

While the TableSat GUI can be used to change the variable TableSat parameters, it does not itself start and stop control of TableSat, nor can it send state data to the user. For that type of communication with TableSat, the user can use one of two different Simulink block diagrams: *tsat_CL.mdl* or *tsat_OL.mdl*.

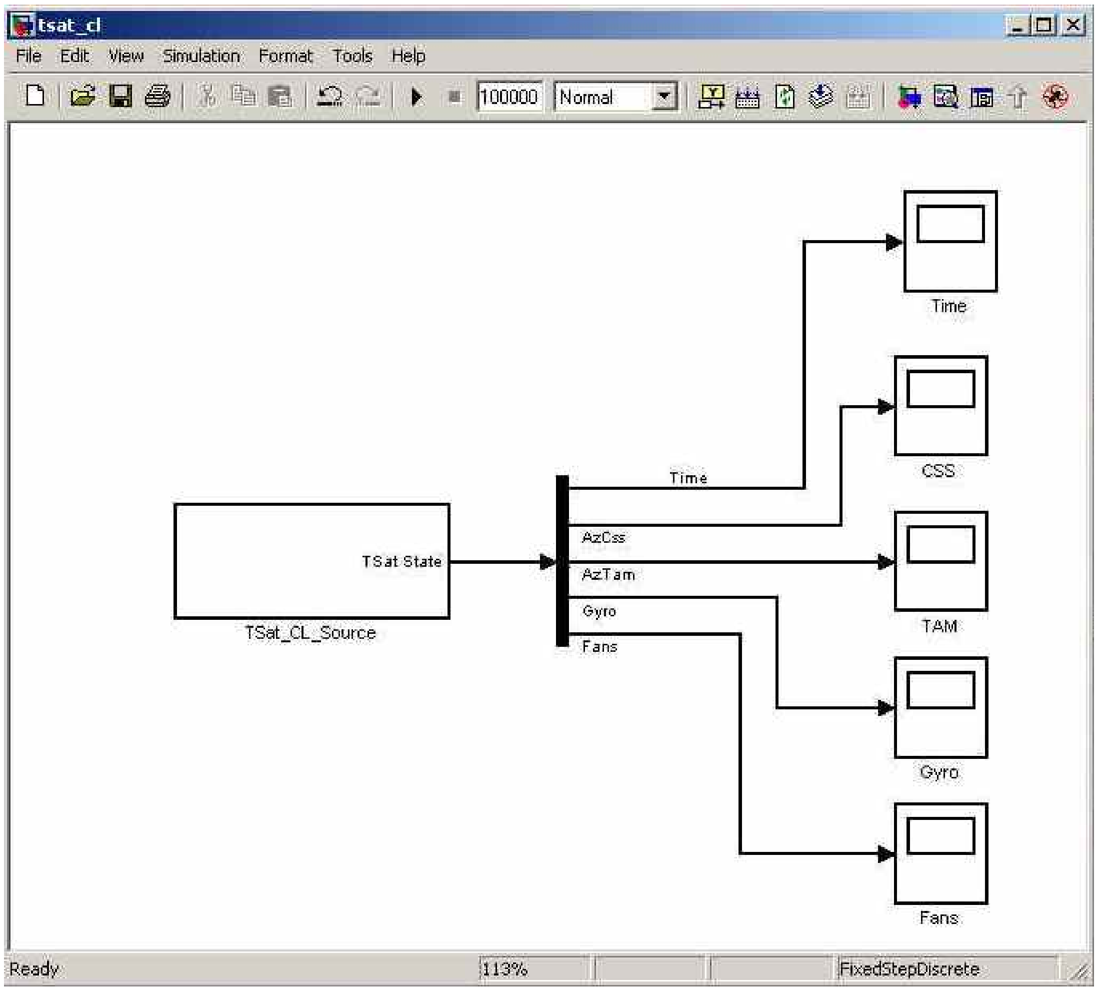


Figure 4.10: Simulink model controlling TableSat using the on-board flight processor.

When the user selects on-board control from the TableSat GUI, *tsat_CL.mdl*, shown in Figure 4.10, is automatically opened. This block diagram can also be opened manually if the user chooses not to use the GUI. The closed-loop block diagram uses the S-Function *tsat_cl_com*, which is located inside the *TSat_CL_Source*, block to initialize the communications port with TableSat, start TableSat running in closed-loop mode using the on-board controller, and feeds state data from TableSat to Simulink for visualization. State data from TableSat is then displayed using the scopes in the Simulink block diagram. All state estimation, control authority, and actuator commands are calculated using the on-board TableSat flight processor. All the variable parameters listed in the TableSat GUI and in Table 4.1 below can be changed when TableSat is operating in closed-loop mode.

If the user opts from the TableSat GUI not to use the on-board control option, *tsat_OL.mdl* is automatically opened. The open-loop block diagram can also be opened manually if the user is not using the TableSat GUI. This block diagram allows the user to design and create controllers and/or estimators within Simulink, and test the designs without loading them onto the TableSat flight processor. The block diagram uses the S-Function, *ts_ol_comm_in*, which is located inside the *TSat_OL_Source* block to initialize the communications port with TableSat and start running TableSat in closed-loop mode with the controller matrices defined as $\mathbf{A} = [\,]$, $\mathbf{B1} = [\,]$, $\mathbf{B2} = [\,]$, $\mathbf{C} = [\,]$, $\mathbf{D1} = 0_{1x5}$, and $\mathbf{D2} = [1, 0]$. With this particular controller definition, the calculated control voltage will be equal to the target. The S-Function then requests the raw state estimates from TableSat. In the block diagram, the user designed state estimator estimates the state, then the controller uses that state estimate to calculate the required control voltage. That control voltage is then sent back to TableSat as a change in pointing target using the S-Function, *tsat_ol_comm_out*. Onboard TableSat, the controller defined above and the change in pointing target sent from the block diagram are used to effectively pass the voltages calculated in the block diagram directly to the actuators.

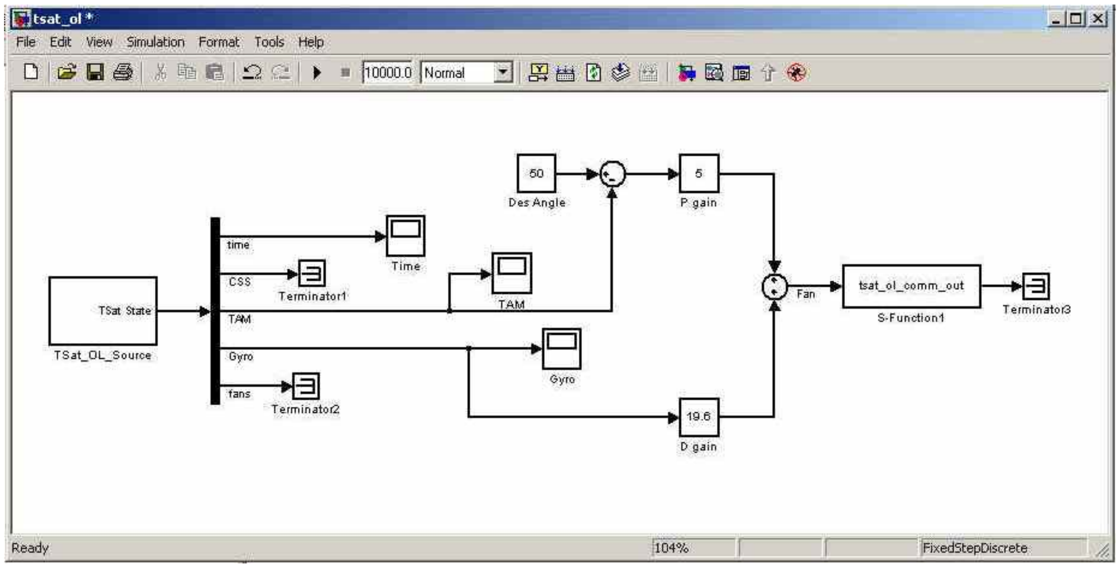


Figure 4.11: Example of an open-loop Simulink model. No state estimator, PD control based on the magnetometer and gyro measurements.

An example of an open-loop block diagram is shown in Figure 4.11. This example has no state estimator and uses a simple Proportional-Derivative (PD) controller based on the magnetometer and gyro measurements. While using the open-loop control option, only the sample rates and actuator type parameters can be changed.

Chapter 5

System Modeling

One of the main goals of this project is to develop a truth model of the TableSat system which can be used to design and test controllers and estimators. Ideally, the model would be linear so that linear controllers can be designed and will behave as predicted by theory. However, TableSat, like all real systems, is subject to nonlinearities such as friction and actuator saturation. Therefore, part of the truth model development includes finding ways to linearize the system by compensating for and reducing the effects of those nonlinearities in the system response. This chapter discusses the development and linearization of the TableSat model.

5.1 TableSat Equations of Motion

To a first approximation, TableSat is assumed to have the following equations of motion:

$$I\dot{\omega} = lK_{wf}(\nu_1 - \nu_2) - f_{TS}(\omega) \tag{5.1}$$

$$\dot{\nu_1} = -\alpha\nu_1 + K_{v\dot{w}}(V_p - f_{fan1}(\nu_1) \tag{5.2}$$

$$\dot{\nu_2} = -\alpha\nu_2 + K_{v\dot{w}}(V_n - f_{fan2}(\nu_2)) \tag{5.3}$$

where I is the TableSat moment of inertia, ω is the TableSat angular velocity, ν_1 is the speed of the positive fan, ν_2 is the speed of the negative fan, l is the fan moment arm, f_{TS} is the TableSat friction and could be a function of ω, K_{wf} is the fan speed to force constant, V_p is the voltage applied to the positive fan, V_n is the voltage applied to the negative fan, α is the fan time constant, $K_{v\dot{w}}$ is the fan voltage to change in speed constant, and f_{fan1} and f_{fan2} are the friction in the fans and could be functions of ν_1 and ν_2, respectively. To effectively model the TableSat system, the constants I, K_{wf}, $K_{v\dot{w}}$, and α, as well as the functions $f_{TS}(\omega)$, $f_{fan1}(\nu_1)$, and $f_{fan2}(\nu_2)$ need to be determined.

5.2 TableSat Moment of Inertia Test

To effectively model the TableSat system, its moment of inertia must first be determined. As TableSat is a one degree-of-freedom system, only the moment of inertia about the spin axis is needed. There are several ways to experimentally determine this moment of inertia; the method chosen here is a torsional pendulum test. A derivation of the dynamic equations used to determine the moment of inertia and an explanation of the test procedure are given below.

5.2.1 Derivation of the Dynamic Equations

TableSat is assumed to be a disk of mass M and radius R. As seen in Figure 5.1, the disk is suspended at its radius by three lines of length L. In the undisturbed state, the lines will be parallel to the Earth's gravity vector and the only force acting on TableSat is gravity. When TableSat is rotated by an angle, θ about its spin axis, the three pendulum lines are displaced relative to the ceiling by an angle β and produce forces parallel to TableSat's disc as seen in Figure 5.2.

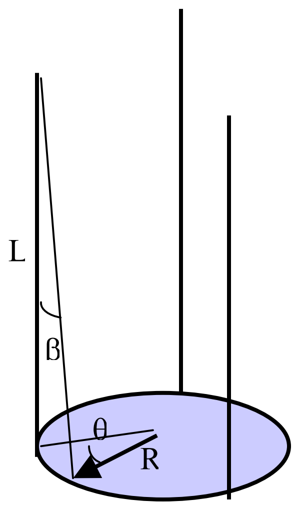


Figure 5.1: TableSat Torsional Pendulum Test Setup.

These forces, acting at radius, R, produce three torques that generate an angular acceleration. TableSat, therefore, is governed by the following single axis

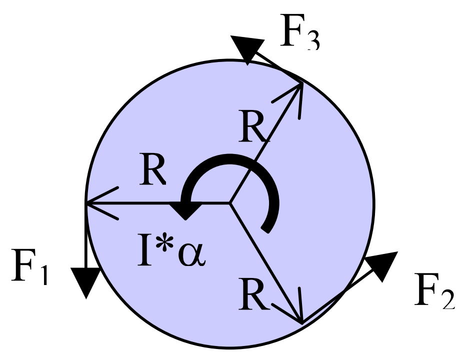


Figure 5.2: View of TableSat from above during Torsional Pendulum Test.

dynamic equation:

$$I\alpha = R(F_1 + F_2 + F_3) = 3RF_1 \tag{5.4}$$

To solve for I, we must find R, F_1, and α. As mentioned above, R is the radius of the TableSat disc, and can easily be measured. The force, F_1, as previously discussed, is a function of the displacement of the pendulum line relative to the ceiling. As the disk is displaced Δ, the pendulum line rotates β degrees relative to the Earth's gravity, g, producing the force, F_1, which can be seen in Figure 5.3.

F_1 and Δ, therefore, can be written as functions of β:

$$F_1 = -\frac{1}{3}Mg\tan(\beta) \tag{5.5}$$

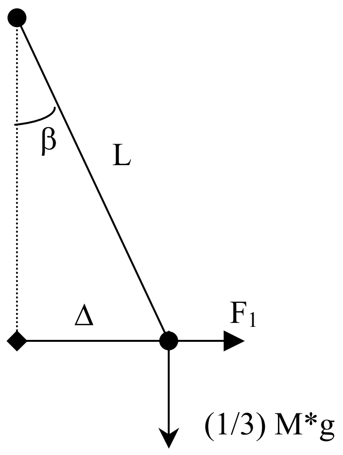


Figure 5.3: Definition of F_1 during the TableSat Torsional Pendulum Test.

$$\Delta = L\sin(\beta) \tag{5.6}$$

The displacement, Δ can be related to the radius vector, R. As TableSat is rotated θ degrees about its spin axis, its initial radius vector, R, is displaced a distance, Δ to R', as seen in Figure 5.4. Δ, can therefore be written as follows:

$$\Delta = 2R\sin\left(\frac{\theta}{2}\right) \tag{5.7}$$

Combining Equations 5.6 and 5.7 and using the small angle approximation, we can solve for β:

$$\beta = \frac{R\theta}{L} \tag{5.8}$$

Continuing the assumption of small angles, Equation 5.8 can be combined

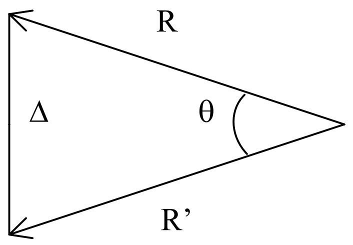


Figure 5.4: Relationship between Δ and R during the TableSat Torsional Pendulum Test.

with Equation 5.5 to yield:

$$F_1 = -\frac{MgR\theta}{3L} \tag{5.9}$$

Substituting the above equation into Equation 5.4, reduces the the dynamic equation of motion to:

$$I\alpha = -\frac{MgR^2\theta}{L} \tag{5.10}$$

Since α is the angular acceleration of TableSat, and hence the derivative of θ, the above equation can be written as a classic pendulum equation:

$$\alpha + \left(\frac{MgR^2}{LI}\right)\theta = 0 \tag{5.11}$$

The period, τ of a second order oscillator is:

$$\tau = 2\pi\sqrt{\frac{LI}{MgR^2}} \tag{5.12}$$

Solving Equation 5.12 for the inertia, I, yields:

$$I = \frac{MgR^2\tau^2}{4\pi^2 L} \tag{5.13}$$

5.2.2 Torsional Pendulum Test Procedure

The TableSat torsional pendulum test is conducted as follows. First, the mass of TableSat is measured. Then, TableSat is suspended from the ceiling using three lines attached about the perimeter of the table. The lines are adjusted until they are plumb with the ground and TableSat is balanced such that its center of gravity is located parallel to the table's spin axis. At this point, the length of the pendulum lines and the distance from TableSat's spin axis to each pendulum line are measured and recorded.

A point on TableSat is identified as the zero point, and TableSat is rotated about ten degrees from that point and released. TableSat will oscillate about the zero point, and, as seen in the derivation above, the period of this oscillation is proportional to TableSat's inertia. One person counts aloud the number of times TableSat passes the zero point from left to right. A second person acts as a time keeper, timing how long it takes TableSat to pass the zero point ten times. That time is recorded, and the rotation and counting are repeated two more times to obtain multiple sets of data. Finally, the pendulum lines and the distance from TableSat's spin axis to each pendulum line are remeasured and recorded.

The above test was conducted for TableSat on 14 September 2004. The resulting data can be seen in Table 5.1. Using this data and Equation 5.13, TableSat's inertia was determined to be 0.053 kg $*$ m^2 +/- 2%.

Torsional Pendulum Test Data	
Parameter	Value
Table Mass (kg)	3.328
Line 1 Length (in)	52.25
Line 2 Length (in)	52.75
Line 3 Length (in)	52.25
Radius 1 Length (in)	7.625
Radius 2 Length (in)	7.5
Radius 3 Length (in)	7.5
Finish Time 1 (sec)	15.14
Finish Time 2 (sec)	15.41
Finish Time 3 (sec)	15.24
Line 1 Length (in)	52.375
Line 2 Length (in)	52.75
Line 3 Length (in)	52.125
Radius 1 Length (in)	7.625
Radius 2 Length (in)	7.5
Radius 3 Length (in)	7.5

Table 5.1: TableSat Torsional Pendulum Test Data. The data was used to determine TableSat's inertia. Where multiple measurements exist, an average value for that parameter was used.

5.3 Friction Characterization

A series of spin tests were conducted which can be used to help characterize the TableSat parameters α and $f_{TS}(\omega)$. During these tests, a constant voltage is applied to TableSat, which is running in open-loop mode. The constant voltage causes TableSat to spin up. Once TableSat is accelerating and before the gyro saturates, the voltage is set to zero and the friction in the TableSat system causes it to spin down to a stop. The angular velocity of TableSat, as measured by the gyro, can be used to characterize α and $f_{TS}(\omega)$. To obtain a representative set of data, voltages ranging from -12 to 12 volts were applied in half volt increments. Figure 5.5 shows some plots of the angular velocity data taken during the spin tests. The sections below discuss how this data was used to parameterize α, and $f_{TS}(\omega)$.

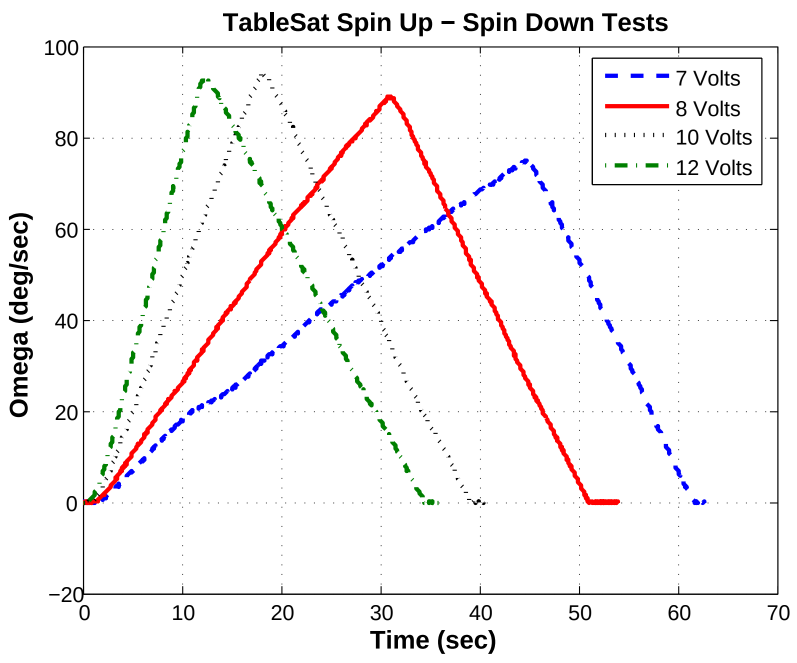


Figure 5.5: Representative angular velocity data taken during a series of TableSat spin up - spin down tests.

5.3.1 TableSat Friction

The TableSat Friction is composed of two parts: the static friction and the dynamic friction. The spin up-spin down tests can be used to characterize both types of TableSat friction.

Static Friction

The static friction in TableSat can be represented by the voltage magnitude, applied to either the positive or the negative fan, below which TableSat will not move. From the spin up-spin down tests, we can find the steady state angular acceleration of TableSat for each constant voltage applied to TableSat. Figure 5.6 shows the steady state angular acceleration plotted vs. the applied voltage. As can be seen in the figure, below voltages of ± 5.5 Volts, TableSat does not accelerate. The static friction, therefore, is approximately 5.5 Volts.

Dynamic Friction

The TableSat dynamic friction can be characterized by the shape of the spin down portion of the angular velocity curve. As can be seen in Figure, 5.5 the spin down portion of the angular velocity curves are straight lines with roughly the same slope, *i. e.* the lines are all parallel. The deceleration of TableSat due to friction is constant regardless of the spin rate of TableSat. In other words, the slope of the spin down curve, or the rate of change of TableSat's angular velocity,

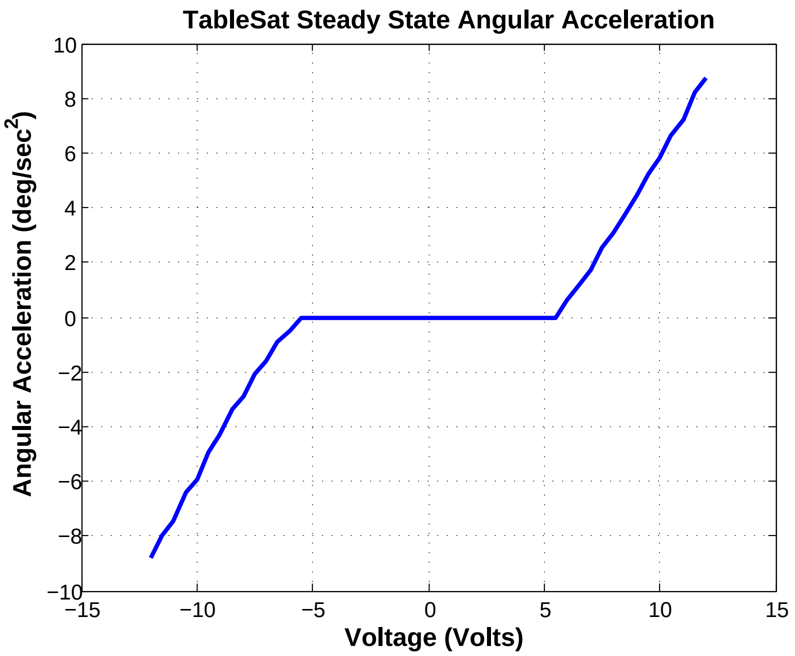


Figure 5.6: TableSat Angular Acceleration for a constant applied voltage. State friction is approximately 5.5 Volts.

is constant. Refer back to Equation, 5.1:

$$\begin{aligned} I\dot{\omega} &= \tau - f_{TS}(\omega) = S \\ \dot{\omega} &= \tfrac{\tau}{I} - \tfrac{f_{TS}(\omega)}{I} = S \end{aligned} \tag{5.14}$$

where S is the constant slope of the spin down portion of the angular velocity curve and $\tau = l * K_{wf}(\nu_1 - \nu_2)$ is the torque applied to TableSat from the fans. But, during the spin down portion of the test, no voltage is applied to the fans, so τ is zero and the TableSat dynamic friction is constant regardless of the speed that TableSat is spinning. This result implies that the magnitude of the dynamic friction is equal to the slope, S, of the spin down curve multiplied by the TableSat inertia, I. The dynamic friction, f_{TS}, is no longer a function of the magnitude of ω, but only of its sign. The magnitude of the friction is constant and will always act against TableSat's spin direction. Upon further analysis of the above results, the constant TableSat friction makes sense given that TableSat spins about a single, constant point of contact. No matter how fast TableSat spins, that point of contact does not change. The contact force is dependant only upon TableSat's mass, which does not change. This single contact point introduces the complication that the magnitude of the friction is then dependant on the condition of the contact point. As the spin point wears, friction can increase. Dirt and debris around the pivot point can also cause an increase in friction. Conversely, if the pivot point is well lubricated, the magnitude of both the static and dynamic friction can decrease. Table 5.2 shows the results of the above friction calculations using the data collected during the spin tests. An average TableSat dynamic friction magnitude was found to be $4.22\mathrm{e}-3\,\mathrm{Nm} \pm 3.2\%$.

5.3.2 Fan Friction

Like the TableSat friction, the fan friction is composed of two separate parts: static friction and dynamic friction. Theoretically, the TableSat fans can accept voltage inputs ranging from 0.0 to 12.0 Volts. With the fans mounted as they are, these inputs produce torques that can spin TableSat in either the positive or negative direction. In reality, however, the fan static friction reduces the effective

TableSat Spin Test Friction Data			
$Volts$	τ_{fric}/I	τ_{fric}/I	τ_{fric}
(V)	(deg/s^2)	(rad/s^2)	(Nm)
-12.0	4.44	0.0775	4.12e-3
-11.5	4.59	0.0801	4.26e-3
-11.0	4.55	0.0795	4.22e-3
-10.5	4.69	0.0818	4.35e-3
-10.0	4.70	0.0820	4.35e-3
-9.5	4.59	0.801	4.25e-3
-9.0	4.60	0.0804	4.27e-3
-8.5	4.68	0.0816	4.33e-3
-8.0	4.63	0.0808	4.27e-3
-7.5	4.65	0.0812	4.31e-3
-7.0	4.56	0.0795	4.22e-3
-6.5	4.51	0.0787	4.18e-3
-6.0	4.30	0.0751	3.99e-3
6.0	-4.33	-0.0756	-4.02e-3
6.5	-4.54	-0.0792	-4.21e-3
7.0	-4.57	-0.0798	-4.24e-03
7.5	-4.50	-0.0785	-4.17e-03
8.0	-4.55	-0.0794	-4.22e-03
8.5	-4.57	-0.0797	-4.23e-03
9.0	-4.55	-0.0794	-4.22e-03
9.5	-4.61	-0.0805	-4.27e-03
10.0	-4.61	-0.0805	-4.28e-03
10.5	-4.61	-0.0805	-4.27e-03
11.0	-4.62	-0.0807	-4.28e-03
11.5	-4.53	-0.0791	-4.20e-03
12.0	-4.31	-0.0753	-4.00e-03

Table 5.2: TableSat Spin Test Friction Data. The data was used to determine an average, nominal value for TableSat's friction.

voltage range of the fans. There exists a voltage below which the fan motor cannot turn the fan blades. This static friction voltage can be determined by commanding the fans with smaller and smaller voltages until they do not have enough power to overcome their static friction and thus do not spin. Through testing, the static friction in the fans was found to be approximately 2.75 Volts, resulting in an actuator deadband ranging from -2.75 Volts to 2.75 Volts.

To determine the dynamic friction of the fans, assuming both fans have the same friction characteristics, consider that the fan friction follows the general friction definition:

$$f_{fan}(\nu) = F_f \mathrm{e}^{-\nu/\beta} + b_f \nu \tag{5.15}$$

where the friction curve is made up of an exponential term and a linear term. $f_{fan}(\nu)$ is the fan friction in Volts as a function of fan speed, ν, F_f is the static friction of the fan, β is the time constant of the fan friction, or the fan speed it takes for the fan friction to decay to 63% of its nominal value, and b_f is the slope of the linear part of the friction curve.

The linear portion of the fan friction curve is modelled as a part of the fan time constant, α. Included in this term are the motor inductance and the back EMF from the motor circuit.

More difficult to determine is the time constant of the fan, which determines the overall shape of the fan friction curve. If the time constant is small, the friction in the fans essentially drops to zero as the fan speed increases. As β increases, the fan friction curve decreases at a slower rate until at very large values of β, the friction is essentially constant. Figure 5.7 shows plots of different fan friction curves as β changes.

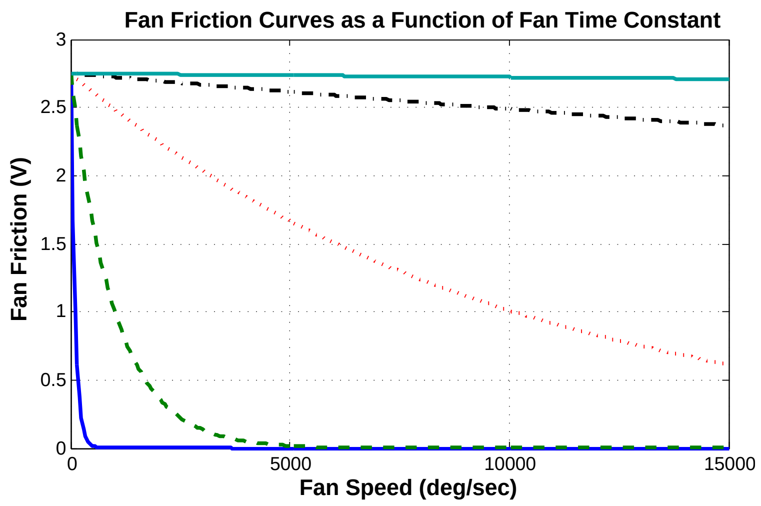


Figure 5.7: TableSat fan friction curves for changing values of β. As β increases, the curve flattens out until fan friction is constant.

Recall from the discussion of α and note again in the spin up portion of the angular velocity curves in Figure 5.5 that for any given voltage, the angular acceleration of TableSat reaches a constant steady state value, *i. e.* TableSat's angular velocity increases linearly. From this result, we can determine that the the exponential portion of the fan friction curve is not exponential at all, but rather is constant, implying that β is a very large value. If the fan friction were not constant, TableSat would not accelerate at a constant rate. As the friction in the fan decreased with increasing fan speed, the rate at which TableSat accelerates would also change.

5.4 TableSat Parameters

Now that the TableSat inertia, TableSat friction, and fan friction have been characterized, we wish to find nominal values for the other parameters in the equations of motion. A nominal value for α can be found using the data from the spin up-spin down tests. Nominal values for $K_{v\dot{w}}$ and K_{wf} can be determined using the hardware specifications from the fan and the TableSat equations of motion.

5.4.1 Fan Time Constant

The fan time constant, α is the time it takes for the fans to reach 63% of their steady state output force for a given commanded voltage. In other words, it is a measure of how quickly the fans can respond to a change in input. Without a tachometer to measure the fan's rotational speed, it is difficult to determine the fan time constant directly. However, α can be determined by using the angular velocity data from the TableSat spin up - spin down tests. During the spin up portion of the test, the fan's output is being used to apply a torque to accelerate TableSat. Under a constant voltage, the fans will spin up until they reach a constant rotational speed and thus a constant force output. As the fan's spin rate changes, TableSat's angular acceleration will increase until it reaches a constant angular acceleration. This constant angular acceleration can be noted in the constant slope of the angular velocity curves seen in Figure 5.5.

When looking for α, we are only interested in the first few seconds of the angular velocity data, the time during which the fan's spin rate and TableSat's angular acceleration are changing. To determine alpha, first find the steady state angular acceleration of TableSat for a given constant voltage by finding the slope, a of the spin up portion of the angular velocity data. Then, using the angular velocity data, ω, plot the curve fit $e = at - \omega(t)$ vs. time, which is, ideally, an exponential curve of the form $e = b(1 - \exp(-\alpha t))$. An example of such a curve fit using spin up-spin down test data can be seen in Figure 5.8

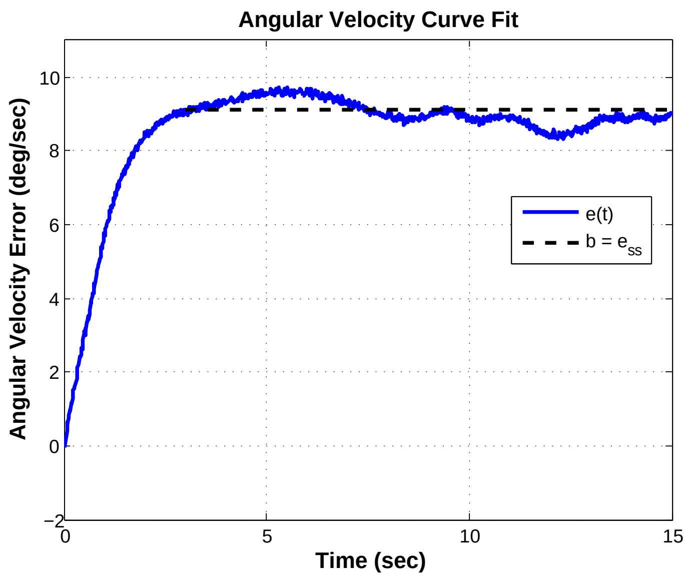


Figure 5.8: Angular velocity curve fit using spin up-spin down test data.

From the plot, find b, the steady state value of e. Then plot the natural log of $e - b$ vs. time. As can be seen in Figure, 5.9, the first few seconds of the plot is a straight line, whose slope is negative α. Regardless of the voltage applied to the fan, α should stay constant for a given fan. Therefore, the above calculations can be repeated for all of the spin up data collected, and an average value for α can be calculated. Using the spin test data from October 15, 2004 an average α value of 2.0 s^{-1} was determined.

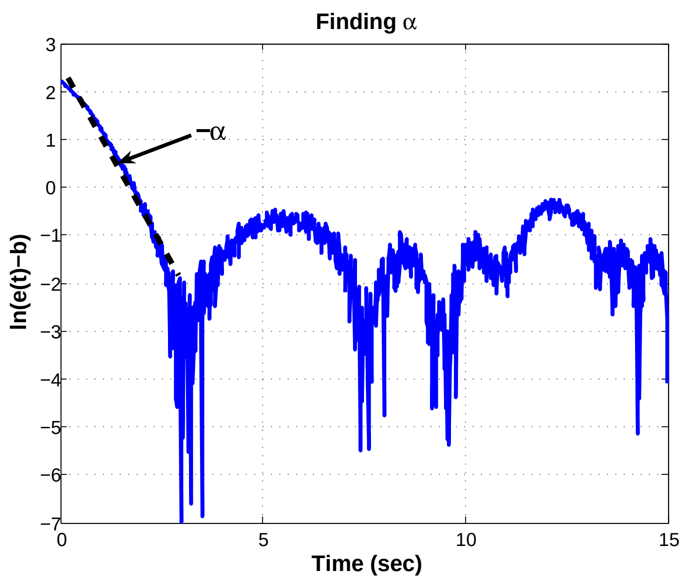


Figure 5.9: Finding α using spin up-spin down test data.

5.4.2 Voltage to Change in Fan Speed Constant

To determine a nominal estimate for $K_{v\dot{\omega}}$, recall from Chapter 2 that the TableSat actuators are 12 volt computer fans, whose speeds are proportional to their input voltage with a maximum speed of 2500 RPM (15000 deg/sec). Consider the fan equations in Equations 5.2 and 5.3. For a constant applied voltage, the change in fan speed will be equal to zero when the fans reach a steady state speed. The following equation can then be solved for $K_{v\dot{w}}$:

$$0 = -\alpha\nu + K_{v\dot{w}}(V_n - f_{fan}(\nu)) = -2 * 15000 + K_{v\dot{w}} * (12 - 2.75) \tag{5.16}$$

where both fans are assumed to have the same characteristics, and the assumed voltage is 12, giving a steady state fan speed of 15000 deg/sec. In addition, we have used the fact that the fan friction is constant and we have established nominal values for f_{fan} and α. Solving for $K_{v\dot{w}}$ yields 3242 deg/sec^2 V.

5.4.3 Fan Speed to Fan Force Constant

To determine a nominal estimate for K_{wf}, recall from Chapter 2 that the TableSat actuators have a maximum volumetric flow rate of 44.5 CFM (0.021 m^3/sec) at 2500 RPM (15000 deg/sec) and an approximate diameter of 92 mm. From fluid dynamics, we know that the thrust force of a fluid exiting a propulsion unit can be calculated as follows:

$$F_{thrust} = \dot{m}_e V_e - \dot{m}_i V_i \tag{5.17}$$

where $\dot{m}_e$ and $\dot{m}_i$ are the output and input mass flow rates of the fluid, respectively, and V_e and V_i are the output and input velocities of the fluid, respectively. For TableSat, the fluid being moved is air, and we assume that the input air velocity is zero, which implies that the air coming into the fan is stationary. Strictly speaking, if TableSat has a non-zero angular velocity, this assumption would be false. However, since the input air velocity would be much less than the output air velocity, the assumption is still reasonable. Based on the above assumption, we can neglect the second term in Equation 5.17. The output mass flow rate and fluid velocities can be calculated using the following equation:

$$\dot{m} = \rho V_r = \rho A V \tag{5.18}$$

where V is defined above, ρ is the density of the fluid being moved by the propulsive unit, V_r is the volumetric flow rate of the propulsion unit, and A is the area through which the fluid is being moved. Combining Equations 5.17 and 5.18 and neglecting the second term of Equation 5.17 yields:

$$F_{thrust} = \frac{\rho_e V_{re}^2}{A_e} \tag{5.19}$$

where the density of air is $1.29\,\frac{\mathrm{kg}}{\mathrm{m}^3}$, A for TableSat can be found using the fan diameter, and the volumetric flow rate of the TableSat fans is defined above. The above equation yields a maximum F_{thrust} of 0.0856 N for the TableSat fans. This F_{thrust} occurs when the TableSat fans are spinning at their maximum speed. Knowing this, K_{wf} can be approximated as:

$$K_{wf} = \frac{F_{thrust}\,\mathrm{N}}{\omega\,\mathrm{degsec}} = 5.71 * 10^{-6}\,\frac{\mathrm{N}}{\mathrm{deg/sec}} \tag{5.20}$$

5.5 Parameter Tuning

Now that we have found nominal values for the parameters found in the TableSat equations of motion, we can use real TableSat data, along with the data generated by the truth model discussed in Section 5.7, to tune those parameters. The goal is to get the truth model response and real TableSat response to match reasonably well.

5.5.1 TableSat Friction

To check the TableSat friction model, TableSat was commanded in open loop mode and given a constant command of 8 Volts for about 30 seconds, then 0 Volts were commanded. Figure 5.10 shows the truth model and real TableSat data plotted together using the nominal truth model system parameters established above.

As can be seen in the figure, the shape of the two curves are similar, but the truth model speed changes too quickly. The downward slope of the curve represents the friction in TableSat itself. It is difficult to tell from the plot, but the slope of the truth model is slightly less than half that of the real TableSat curve. This difference in slope implies that there is more friction in the system than what is being modelled in the truth model. Therefore, to obtain agreement between the truth model and real system, start by increasing the TableSat friction. Figure 5.11 shows the truth model and TableSat data with an approximately 6% increase

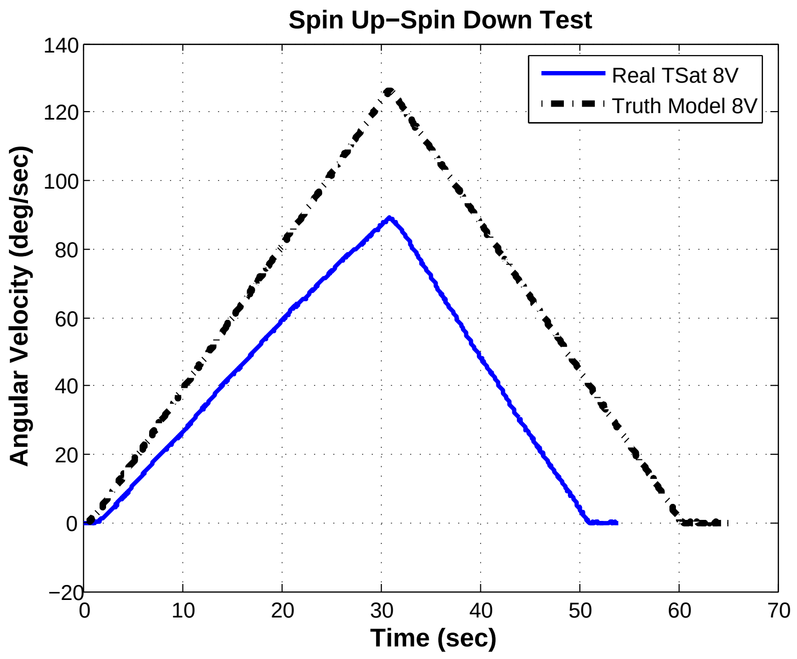


Figure 5.10: Comparison of the truth model and real TableSat system with open loop control and a constant command input of 8 Volts. Nominal system parameters used in the truth model.

in TableSat friction.

It may be difficult to see in this figure, but the downward slope of the truth model curve is now approximately the same as the upward slope, which implies that the TableSat friction is approximately correct.

5.5.2 Fan Friction and K_{wf}

It is easy to see in Figure 5.11 that the truth model still increases too quickly. The upward slope of the curve includes a combination of both the TableSat friction and the fan friction. Since we are now fairly confident that the TableSat

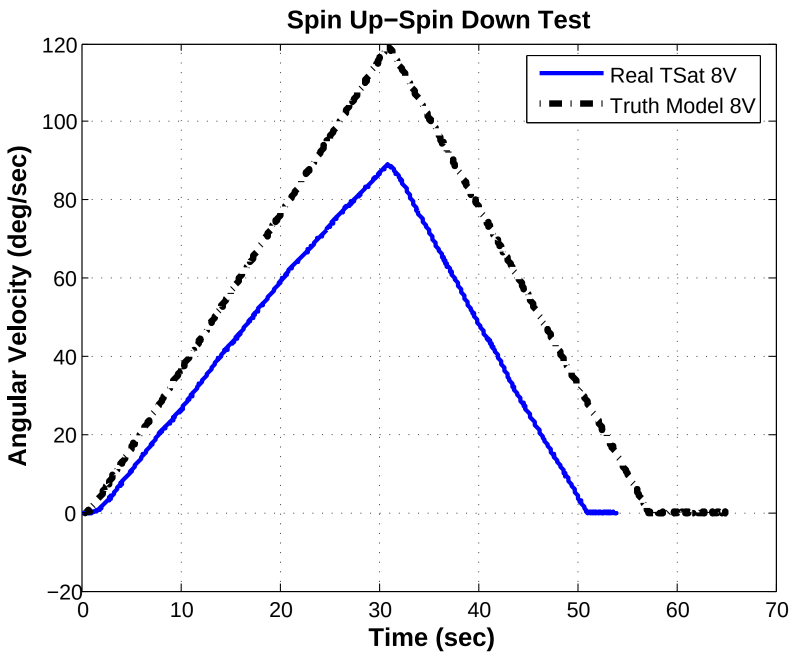


Figure 5.11: Comparison of the truth model and real TableSat system with open loop control and a constant command input of 8 Volts. Truth model TableSat friction increased 6%.

friction is correct, we must now look at adjusting the fan parameters. Start by just adjusting the fan friction. Recall that the fan friction was assumed to be constant. From the plots shown above, this assumption can be shown to be correct. If the time constant of the fan friction were lower, implying that fan friction decreased as fan speed increased, the truth model speed would increase at an even faster rate because there would be less friction opposing the fan force. In fact, the discrepancy in the angular velocity curves actually implies that the fan friction is greater. Figure 5.12 shows the truth model and TableSat angular velocity curves with a roughly 23% increase in fan friction.

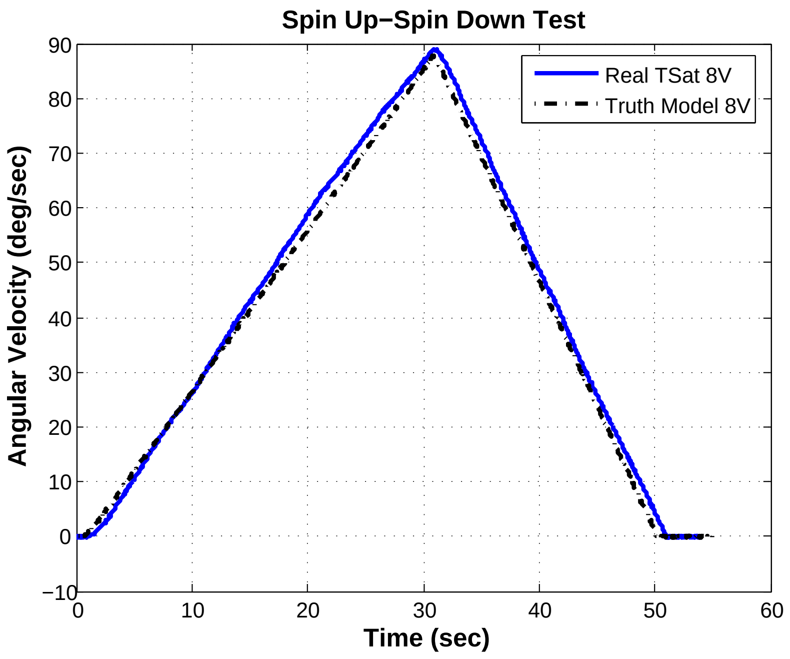


Figure 5.12: Comparison of the truth model and real TableSat system with open loop control and a constant command input of 8 Volts. Truth model TableSat friction increased 6%. Truth model fan friction increased 23%.

As the figure shows, there is now good agreement between the truth model and real TableSat data for this particular set of data. Now consider a second set of data. If the fan friction and K_{wf} are both accurately modelled, the truth model prediction and real TableSat data should match for another set of data. Figure 5.13 shows the truth model and real TableSat data for both the 8 Volt and 10 Volt spin up-spin down tests.

As can be seen in the figure, the 10 Volt truth model data does not match the real TableSat data. This result implies that adjusting just the fan friction was not correct. Try instead adjusting both the fan friction and the fan speed

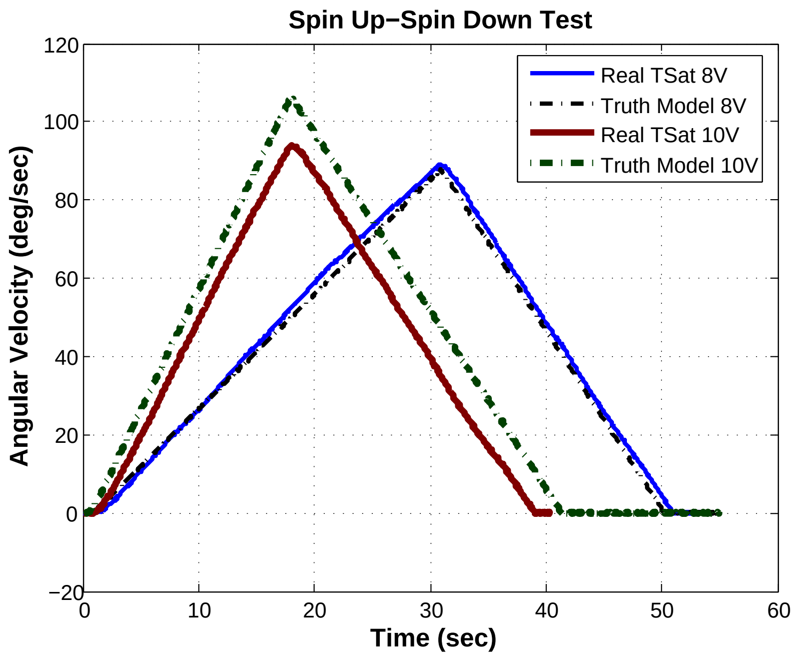


Figure 5.13: Comparison of the truth model and real TableSat system with open loop control and a constant command inputs of 8 and 10 Volts. Truth model fan friction increased 23%.

to force constant. Figure 5.14 shows the truth model and real TableSat data when the fan friction is set at 2.8 Volts and K_{wf} is decreased to 13% $5.05 * 10^{-6}$. As can be seen in the figure, both sets of curves show fairly good agreement, giving confidence that the fan friction and fan speed to force constant have been accurately modeled.

The above plots consider only a spin up of TableSat using the positive fan. To ensure that the fan friction model and K_{wf} are also accurate for the negative fan, consider a spin test with a command of -8 Volts. Figure 5.15 shows the truth model and TableSat data using the adjusted parameters determined above.

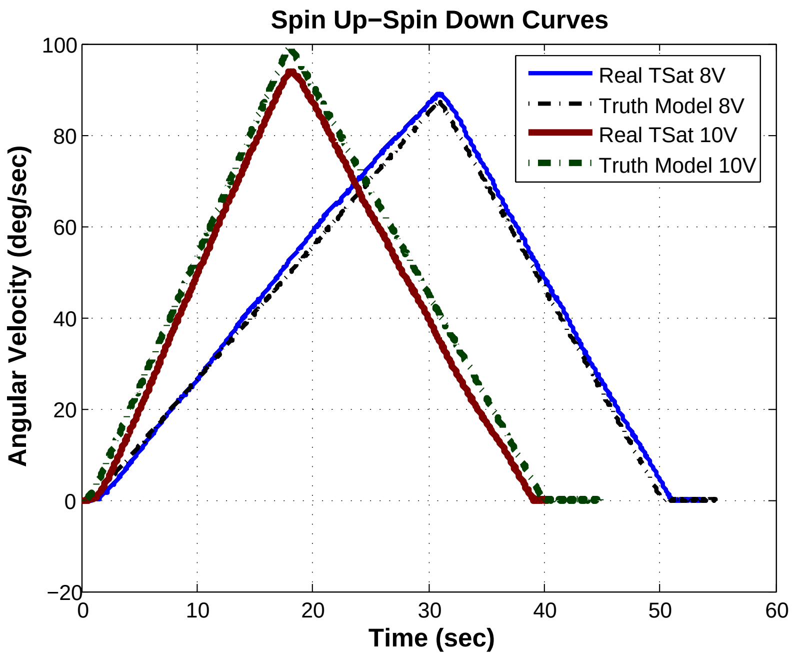


Figure 5.14: Comparison of the truth model and real TableSat system with open loop control and a constant command inputs of 8 and 10 Volts. Truth model fan friction at 2.8V, K_{wf} increased 13%.

Notice that the truth model peak is slightly higher than the real TableSat data. During the course of the TableSat testing, it has been noted that the negative fan appears to be slightly less powerful than the positive fan, but the truth model assumes that both fans are identical. Figure 5.16 shows the truth model and real TableSat data if the fan speed to force constant, K_{wf}, for the negative fan is decreased by 1%. As can be seen in this figure, there is even better agreement between the two curves, which indicates that the TableSat and fan friction are modelled accurately.

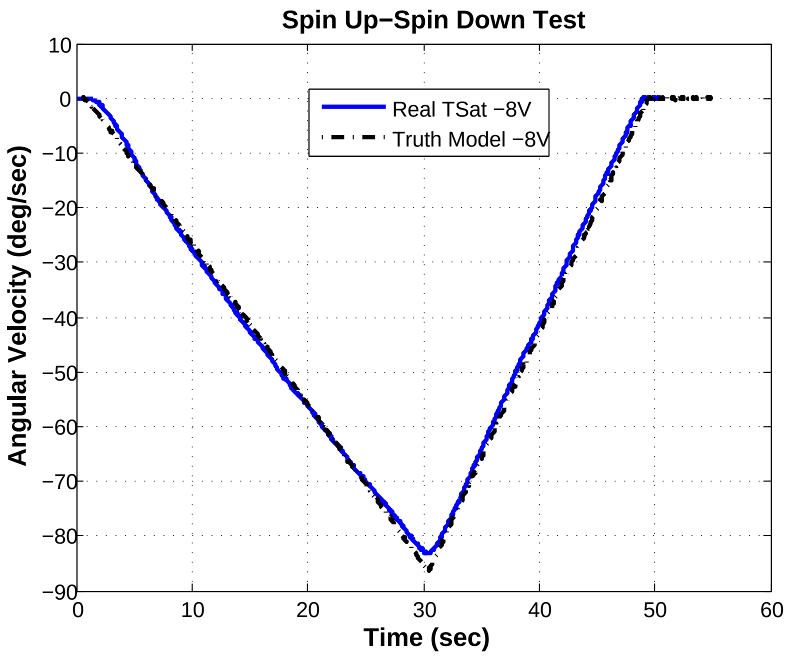


Figure 5.15: Comparison of the truth model and real TableSat system with open loop control and a constant command input of -8 Volts. Adjusted friction parameters used in truth model.

5.5.3 Fan Time Constant

In Section 5.4 we used the spin up-spin down test data to determine a nominal value for the fan time constant α. To verify this value, look at the first few seconds of the spin up-spin down truth model and real TableSat data from Figure 5.14. Figure 5.17 show a zoom in of the first 4 seconds of data. As can be seen in the figure, the acceleration of the truth model and real TableSat data match well, implying that α is correct.

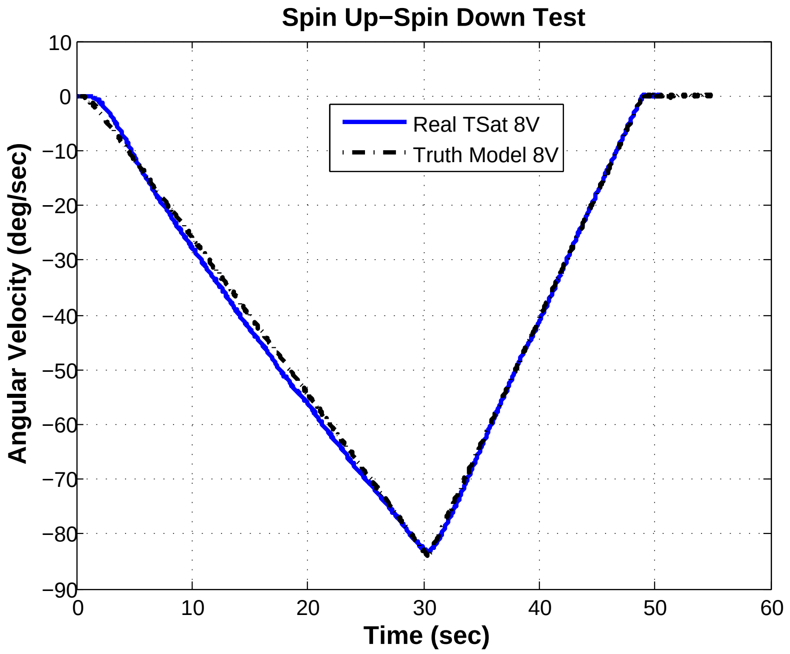


Figure 5.16: Comparison of the truth model and real TableSat system with open loop control and a constant command input of -8 Volts. Adjusted friction parameters and 1% decrease in K_{wf} for the negative fan used in truth model.

5.5.4 Fan Voltage to Change in Speed Constant

In Section 5.4 we established a nominal value for the fan voltage to change in speed constant, $K_{v\dot{w}}$, using the hardware specifications for the fan and the fan equations of motion. This nominal value assumes that the maximum fan speed matches the 2500 RPM specification. In reality, the maximum fan is probably not exactly 2500 RPM. Unfortunately, without tachometer feedback from the fan, we have no way of measuring the real fan speed. If we had fan tachometer data, we could use that to tune $K_{v\dot{w}}$.

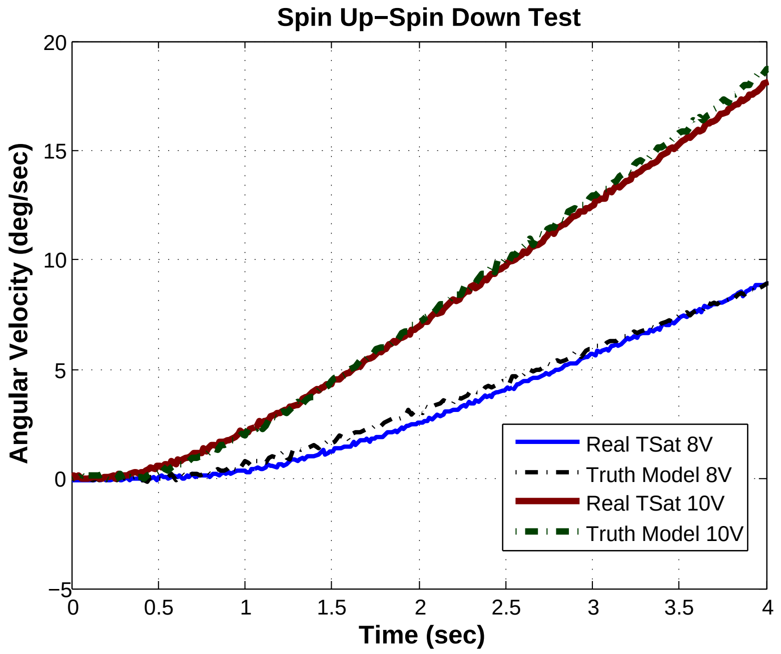


Figure 5.17: Verification of α by comparing the TableSat acceleration in the truth model and real TableSat data.

5.6 Sensor Noise Characterization

To create an accurate truth model the TableSat system, we need to accurately characterize the noise of each of the sensors. To characterize the gyro noise, TableSat is given a target rate of 0 deg/sec. A plot of the gyro signal can be seen in Figure 5.18. MATLAB was then used to find the standard deviation of the gyro signal, σ_g. σ_g was found to be approximately 0.09 deg/sec.

To characterize the magnetometer noise, TableSat was issued a constant desired pointing angle. Upon reaching its steady state pointing, the TAM data was used to find the standard deviation of the TAM noise, σ_T. Figure 5.19 shows

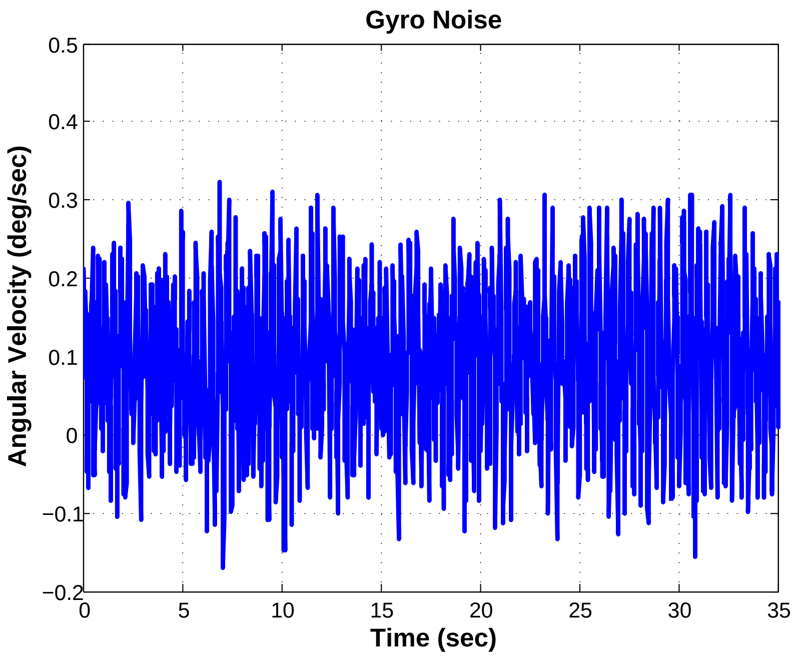


Figure 5.18: Gyro Noise, $\sigma_g = 0.09 deg/sec$.

the steady state magnetometer angle. The standard deviation of the angle is approximately $\sigma_T = 2.3$ deg.

To characterize the CSS noise, recall first from Chapter 3 that the θ_{CSS} angle is calculated based on readings from one or two CSS's. Each CSS will have slightly different noise parameters, and in addition, the physical relationship between adjacent CSS's can also contribute to the noise in the θ_{CSS} angle. So, to characterize the CSS noise, TableSat was rotated manually to different positions to see how the noise varies from one orientation to another. The CSS data collected was then adjusted so that the mean CSS value was zero. Figure 5.20 shows the adjusted CSS data collected. As can be seen in the figure, there are

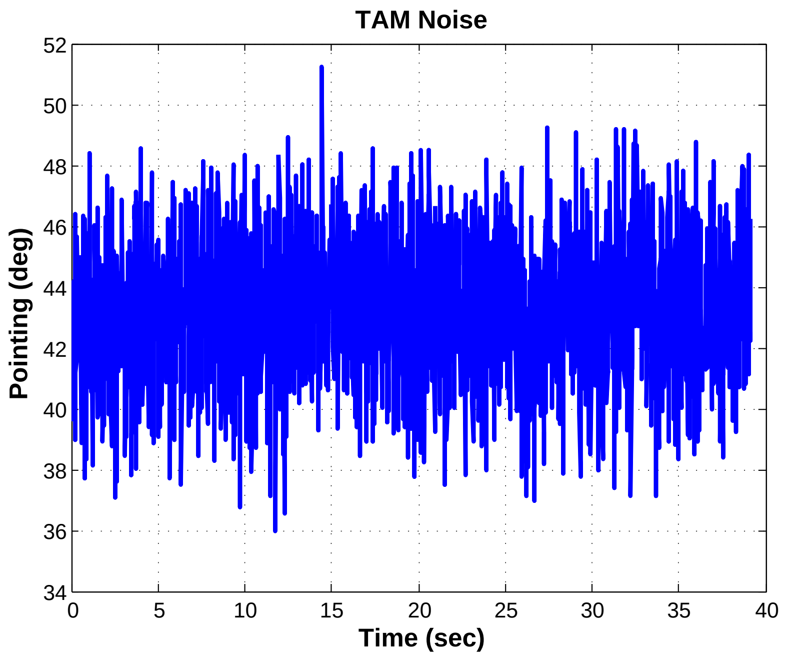


Figure 5.19: TAM Noise, $\sigma_T = 2.2 deg/sec$.

approximately seven different noise levels, depending on TableSat's orientation. Some noise levels are very good, while others are quite large. The variation is due to the fact that the CSS sensors are not permanently mounted flush to the CSS head. They are slightly canted, therefore the CSS boresight is not necessarily normal to the CSS head. Calculating the CSS angles as the arccosine of the normalized CSS measurement is prone to a lot of noise. When the CSS's have been permanently mounted, the standard deviation of θ_{CSS}, σ_C, can be determined. For now, σ_C was estimated to be about 1.2 deg.

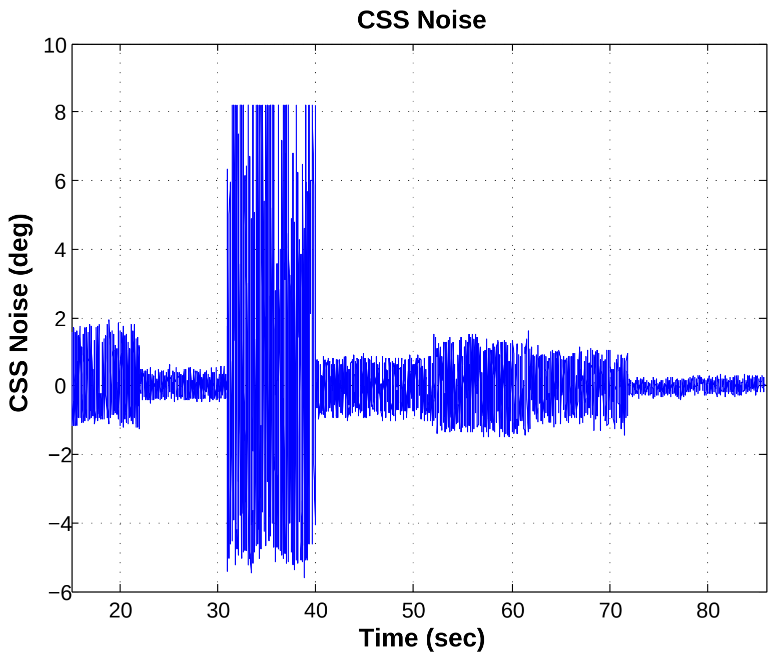


Figure 5.20: CSS Noise, $\sigma_C = 1.2 deg/sec$.

5.7 TableSat Truth Model

Of use in designing and testing controllers is a truth model that accurately simulates the dynamics of the system in question. A TableSat Truth Model was created, using MATLAB and Simulink, and can be seen in Figure 5.21. The truth model integrates the equations of motion shown in Equations 5.1, 5.2, and 5.3 and mimics the actual compensation methods used on board the real TableSat system. These nonlinear compensation methods are presented in Chapter 6. The truth model also includes sensor noise, which is modelled as Gaussian distributed random noise with a different variance for each type of sensor. The noise variances were determined through testing and are presented in Section 5.6

above.

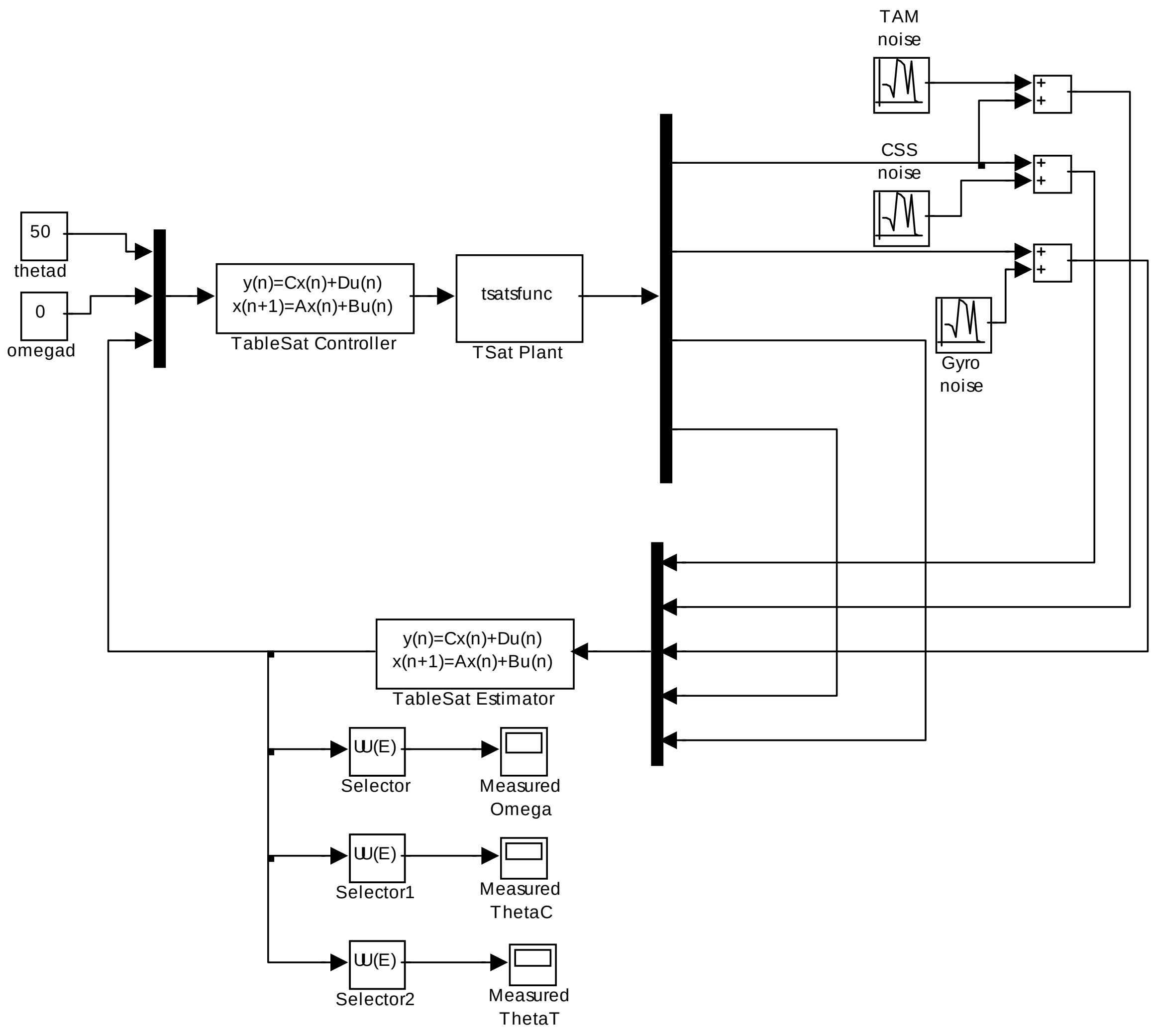


Figure 5.21: TableSat time domain truth model integrating full system equations of motion.

The truth model has a total of 10 parameters whose values can be varied in order to change the response of the system. These parameters are: I, l, f_{TS}, α, K_{wf}, $K_{v\dot{w}}$, F_f, β, $fcomp_{TS}$, and $Flag_{fcomp}$. The first six parameters are the same parameters defined in Equations 5.1, 5.2, and 5.3. F_f and β were also defined above and represent the magnitude and time constant of the fan friction as defined in Equation 5.15, where the positive and negative fans are assumed to have the same friction. $fcomp_{TS}$ is the magnitude of the voltage applied to TableSat to compensate for the constant TableSat friction. Finally, $Flag_{fcomp}$ is a flag that turns on and off the TableSat friction compensation. Nominal values for all of these parameters were found earlier in this chapter.

Fine tuning these parameters was discussed in Section 5.5 above and was accomplished by comparing truth model data to real TableSat data collected for the same controller. If designed correctly, the time domain simulation should mimic what is actually happening with TableSat. The parameter values can be tuned until the truth model data matches the real data. In addition, the truth model can also be used to test controllers and estimators before actually running them on the real TableSat system. Chapter 7 shows data generated by the TableSat truth model for several different controllers.

Chapter 6

Linear Model Development

In Chapter 5 the TableSat equations of motion were developed and parameterized and a TableSat truth model was created using these equations of motion. Recall that the equations of motion include friction terms for the fans and for TableSat itself, making the model nonlinear in nature. The nature of these friction components were identified in Chapter 5. The goal in this chapter is to determine ways to compensate for these friction terms, effectively eliminating their contributions to the equations of motion. Without friction in the system, the equations of motion can be reduced to a linear model of TableSat.

6.1 TableSat Friction Compensation

In Chapter 5 the TableSat friction was determined to be constant in magnitude, depending only on the direction of TableSat's angular velocity. This particular friction model implies that a simple friction compensation curve can be created to compensate for the constant TableSat friction, thus eliminating that particular nonlinearity from the TableSat system. Depending on the direction of spin, a constant voltage is added to the commanded voltage to overcome the friction in the system. Recall from Chapter 4 that the TableSat friction compensation function interpolates between a set of points, (ω, $Volts$), to determine how much additional voltage to add to the command voltage to compensate for friction. This interpolation implies that there must be a deadband around $\omega = 0$, so that the friction compensation curve is continuous and does not have any discrete jumps.

Recall from Chapter 5 that a nominal friction value for TableSat was determined to be approximately 6 Volts. A nominal friction compensation curve would then be: $\omega =$ [-120, -0.1, 0, 0.1, 120] deg/sec and $Volts =$ [-6, -6, 0, 6, 6] V. This particular friction compensation curve would add 6 Volts when TableSat's angular velocity is between 0.1 and 120 deg/sec, -6 Volts when TableSat's angular velocity is between -0.1 and -120 deg/sec, and would interpolate linearly between -0.1 and 0.1 deg/sec such that at 0 deg/sec, 0 additional voltage is added. The width of the deadband may need to be adjusted depending on the noise level of the angular velocity measurement.

The above suggested friction compensation curve is a nominal estimate of the friction in TableSat. Recall also from Chapter 5 that the level of friction in the TableSat system can vary depending on the condition of the contact point. Any time TableSat is set up, the friction compensation curve should be checked and adjusted if necessary. One method for adjusting the friction compensation curve is to command TableSat in open loop mode. Start with a friction compensation curve using the nominal voltage compensation values of -6 and 6 Volts. Start TableSat running with a commanded rate of zero degrees per second. It should remain at rest. From a rest, manually spin TableSat in the positive direction. If friction is perfectly compensated for, TableSat should maintain a constant rate. If the rate drifts downward, increase the voltage in the compensation curve. If the rate drifts upwards, decrease the voltage in the compensation curve. Iterate until TableSat maintains a constant rate. Repeat this procedure after spinning TableSat in a negative direction. Figure 6.1 shows what TableSat's angular velocity should look like if friction is correctly compensated for. After the initial transients from each of the positive and negative impulses, a relatively constant angular velocity is maintained.

6.2 Fan Friction Compensation

Recall from Chapter 5 that the TableSat fan friction was identified to have a linear component and a constant, static component. The linear friction component is contained as a part of α, the fan time constant. After the fan's initial acceleration, that friction component should not be a factor. The static friction, however, still needs to be compensated for. One method for removing this deadband

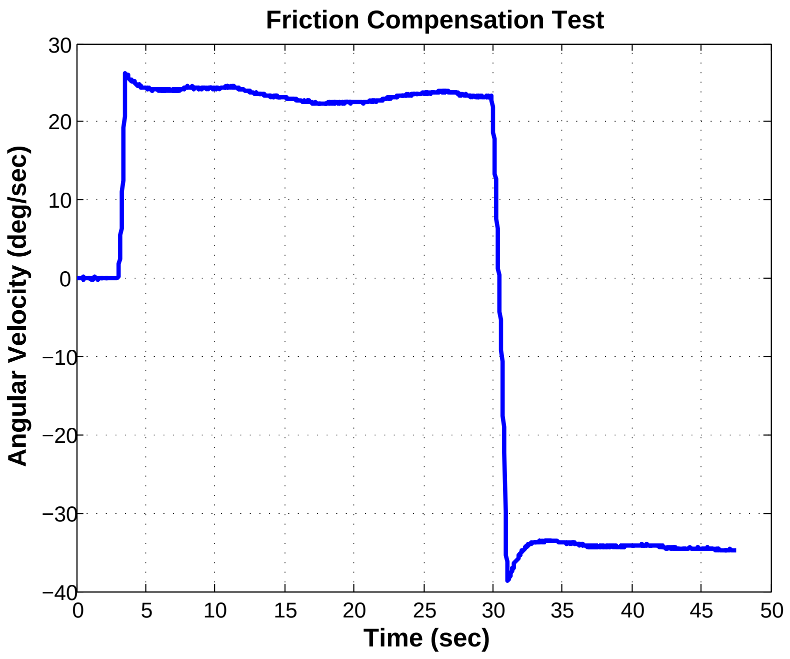


Figure 6.1: TableSat friction compensation test; TableSat commanded in open-loop mode and manually given impulses in positive and negative direction.

nonlinearity, was to implement a new TableSat actuator commanding that utilizes the fact that both TableSat fans can be commanded at the same time. Recall from Chapter 5 that the static fan friction was determined to be approximately 2.8 Volts. If the command issued by the controller, v, is between 0.1 and 3 Volts, the positive fan is commanded at v plus 3 V. At the same time, the negative fan is commanded at 3 V. The net result is that TableSat will spin in the positive direction with an effective command of v. If the command issued by the controller, v, is between -0.1 and -3 Volts, the negative fan is commanded at $-v$ plus 3 V. At the same time, the positive fan is commanded at 3 V. The net result is that TableSat will spin in the negative direction with an effective command of v. Figure 6.2 shows this fan static friction compensation method in a graphical form. The above actuator commanding effectively reduces the actuator static friction such that it now ranges only from -0.1 to 0.1 Volts.

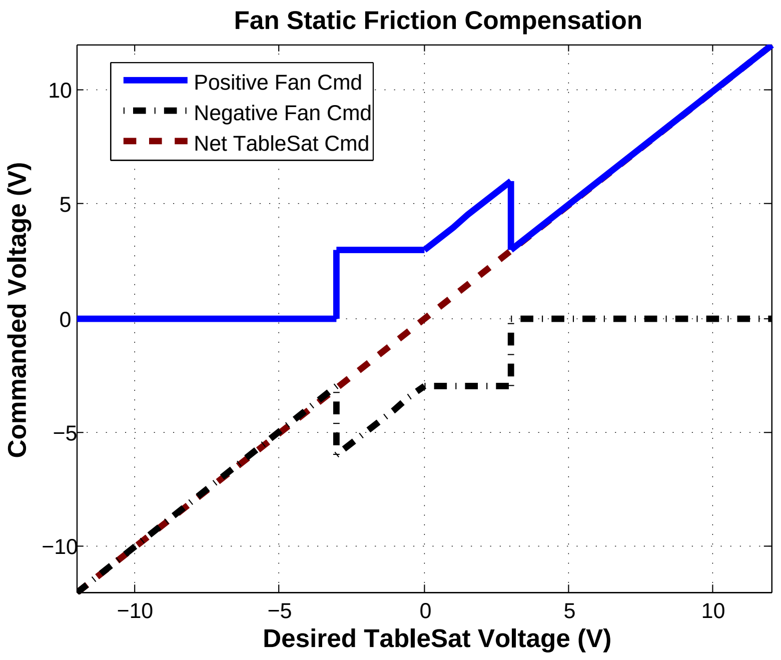


Figure 6.2: Graphical representation of the TableSat fan static friction compensation method.

6.2.1 Linear Model

One of the goals of the TableSat project was to create a linear system model of TableSat that can be used to design controllers and state estimators for the system. Of course, a linear model is only valid if the system itself can be considered linear. When the friction compensation methods described above are implemented, the friction nonlinearities in the system can be virtually eliminated, which implies that f_{TS}, f_{fan1}, and f_{fan2} can be neglected in the TableSat equations of motions, Equations 5.1, 5.2, and 5.3. In addition, because the static fan friction has been virtually eliminated, the two separate fan equations can be reduced to one equation, representing a single bi-directional fan. The simplified equations of motion are then:

$$\begin{aligned}\dot{\theta} &= \omega \\ \dot{\omega} &= \tfrac{lK_{wf}}{I}\nu \\ \dot{\nu} &= -\alpha\nu + K_{v\dot{w}}V\end{aligned} \tag{6.1}$$

which can be written in state space form as:

$$\begin{aligned}\begin{bmatrix}\dot{\theta}\\ \dot{\omega}\\ \dot{\nu}\end{bmatrix} &= \begin{bmatrix}0 & 1 & 0\\ 0 & 0 & \frac{lK_{wf}}{I}\\ 0 & 0 & -\alpha\end{bmatrix}\begin{bmatrix}\theta\\ \omega\\ \nu\end{bmatrix} + \begin{bmatrix}0\\ 0\\ K_{v\dot{w}}\end{bmatrix}V \\ \begin{bmatrix}\theta\\ \omega\end{bmatrix} &= \begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\end{bmatrix}\begin{bmatrix}\theta\\ \omega\\ \nu\end{bmatrix}\end{aligned} \tag{6.2}$$

or

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{b}V \\ \mathbf{y} &= \mathbf{C}\mathbf{x}\end{aligned} \tag{6.3}$$

where

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 8.784\text{e}^{-4} \\ 0 & 0 & -2.0 \end{bmatrix}$$
$$\mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ 3242 \end{bmatrix} \tag{6.4}$$

For this linear state space model, TableSat is assumed to be a continuous system (*i. e.* TableSat is commanded with continuous actuation and the controller, state estimator, and actuator cycles are reasonably fast. The given C matrix corresponds to the only measurable outputs being θ and ω. This state space model can be used to design Model Based Controllers, which will be discussed in Chapter 7, and other state space controllers. For controller and estimator frequency domain analysis and testing, as well as traditional linear controller design, it is easier to reduce the equations of motion to a single input - single output linear transfer function that converts from Voltage input to angle output. Again, assuming TableSat is a continuous system, taking the LaPlace transform of Equation 6.1 yields:

$$\begin{array}{l} s\theta(s) = \omega(s) \\ s\omega(s) = \frac{lK_{wf}}{I}V(s) \\ s\nu(s) = -\alpha\nu(s) + K_{v\dot{w}}V(s) \end{array} \tag{6.5}$$

The above equations can then be solved simultaneously to yield the following TableSat transfer function:

$$G_{TS}(s) = \frac{\theta(s)}{V(s)} = \frac{lK_{wf}K_{v\dot{w}}}{Is^2(s+\alpha)} = \frac{0.1595}{0.056s^2 + 0.112s} \tag{6.6}$$

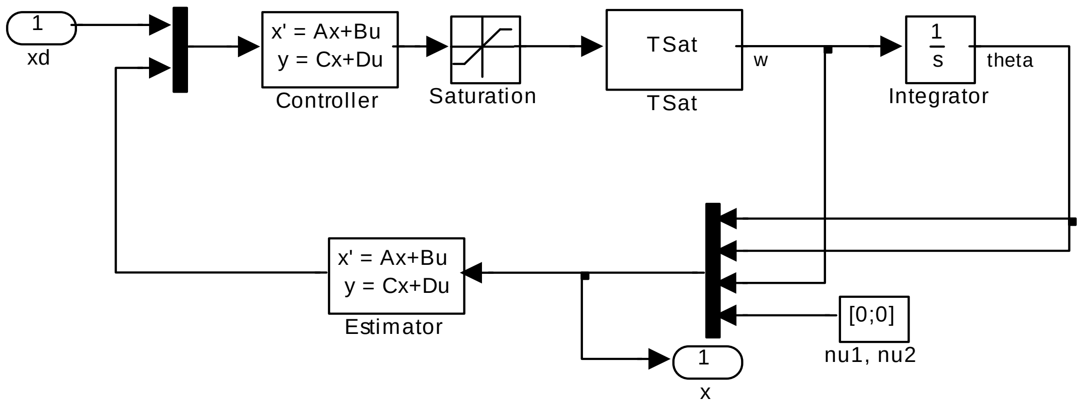


Figure 6.3: TableSat frequency domain simulation using the linear model, $T_{TS}(s)$.

$G_{TS}(s)$ is referred to as the TableSat linear model. Using $G_{TS}(s)$ and Simlink, a closed loop model of the linearized TableSat system was created. This model, which is shown in Figure 6.3, can be used to do frequency domain analysis on any controllers or estimators designed for TableSat using the MATLAB Control Systems Toolbox. The controllers and estimators must be represented in state space form, where the **A** and **C** matrices of the controller and the **A**, **B**, **C**, and **D** matrices of the estimator correspond to the respective matrices of the controller and estimator definitions in the TableSat GUI. The **B** and **D** matrices of the controller relate to the TableSat **B1**, **B2**, **D1**, and **D2** matrices as follows:

$$\begin{aligned} \mathbf{B} &= [\mathbf{B2}, \mathbf{B1}] \\ \mathbf{D} &= [\mathbf{D2}, \mathbf{D1}] \end{aligned} \tag{6.7}$$

6.3 Linear Model Verification

Now that a linear TableSat model has been developed, the next step is to verify that the linear model is a valid approximation of TableSat that can be used in controller design. To do so, a simple proportional controller was used to control the TableSat rate. Figure 6.4 shows the the linear and truth models plotted on the same plot when a step command of 20 degrees/second is issued at time zero.

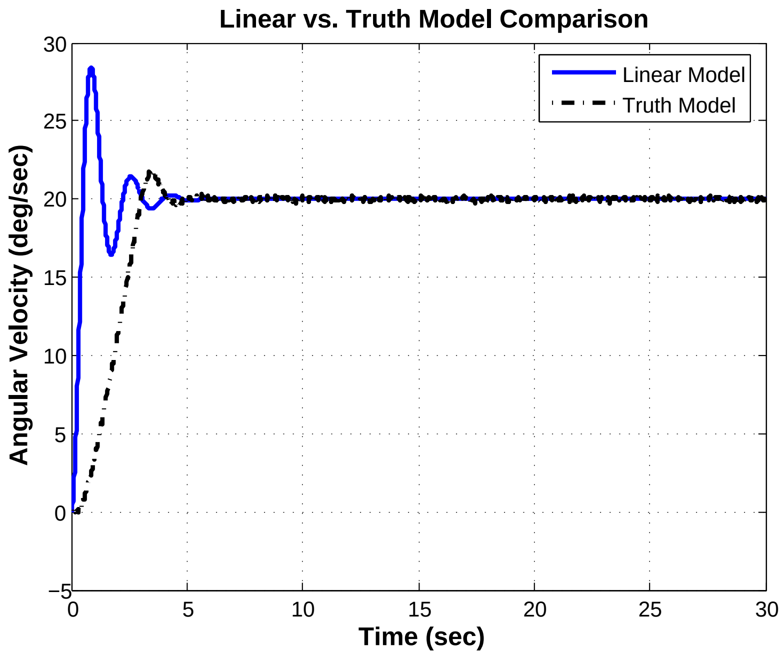


Figure 6.4: Comparison of the TableSat Linear and Truth Models with a simple P controller, $K_P = 5$ and a step input of 20 deg/sec.

As can be seen in the figure, the linear model shows a much quicker response and a much higher overshoot; however once the transient has died down, the plots match quite well. The discrepancy in the transients of the two models can be explained in that the linear model assumes infinite control, whereas the truth model (and the real TableSat) actuators will saturate at about 6 volts, depending on the level of friction compensation. Instead of commanding a step input of 20 deg/sec, which quickly saturates the actuators, try commanding a step input of 2 deg/sec. This command will still saturate the actuators when $K_p = 5$, but not nearly as much. Figure 6.5 shows the plots of the two models for the smaller step input.

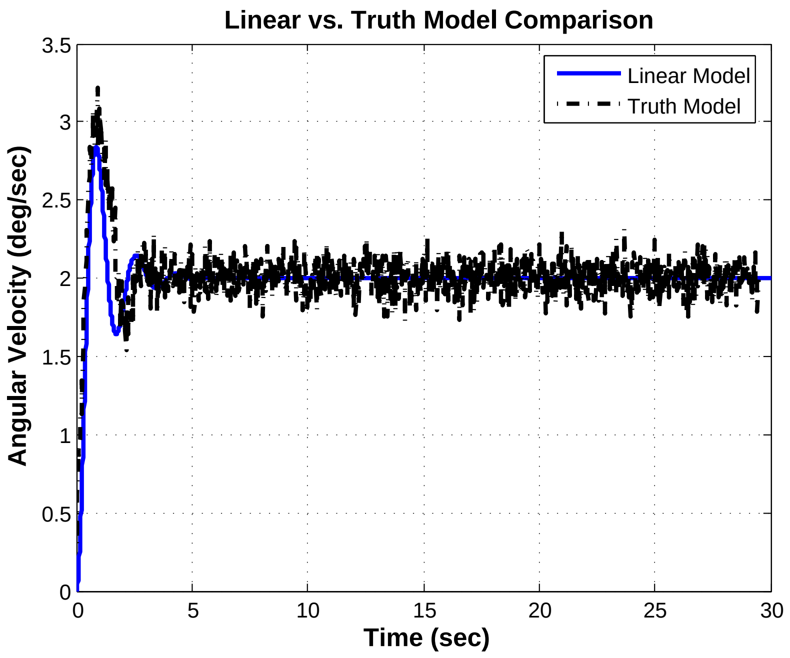


Figure 6.5: Comparison of the TableSat Linear and Truth Models with a simple P controller, $K_P = 5$ and a step input of 2 deg/sec.

As can be seen in the figure, the transient responses of the two models are much closer now. It looks like the truth model has a higher overshoot, but that could be mostly the noise in the truth model, which is much more prevalent in this case. Despite the large noise, it can still be seen that the steady state values of the linear and truth models are both the same. From these plots, it appears that the linear model is a valid approximation of the TableSat system as long as you remain within the saturation level of the actuators.

Another way to check the validity of the linear model is to create frequency response plots using the linear model and comparing that to the truth model and real TableSat results. To do so, TableSat is commanded in open-loop mode and issued a sinusoidal rate command with a given frequency and amplitude. Theoretically, if TableSat is a completely linear system the output rate should also be a sinusoid, perhaps with a different amplitude. A series of different frequency sine waves can be commanded and the ratio of the output amplitude over the input amplitude can be plotted versus the input frequency. If the ratio is converted to dB and the frequency is plotted on a log scale, the resulting plot is an experimentally obtained Bode magnitude diagram. To start, Figure 6.6 shows the linear model prediction for a commanded sine wave with a frequency of 1.0 rad/sec and an amplitude of 5 deg/sec.

As can be seen in the figure, the angular velocity is indeed a sine wave with an amplitude of about 6 deg/sec, giving an output over input ratio of about

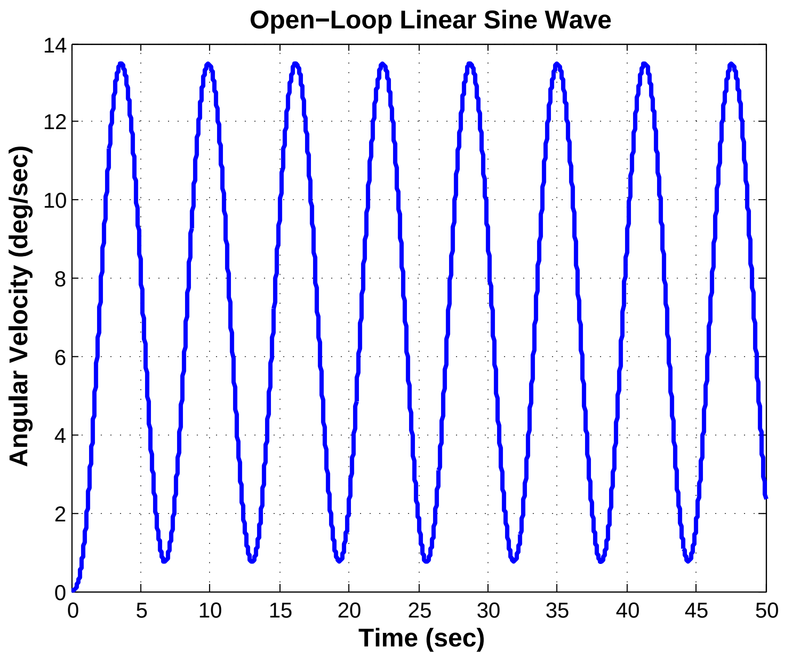


Figure 6.6: Linear model prediction for open-loop commanded sine wave.

1.2, or 1.6 dB. Figure 6.7 shows the truth model and real TableSat data plotted together for the same commanded sine wave.

As can be seen in the figure, both the truth model and real TableSat data match well. The overall trend in the data is a sinusoid, but the sinusoidal rate walks away instead of staying constant. Because TableSat is a marginally stable system (it has two poles at zero), any small nonlinearity will cause the system to walk away like that, so this behavior implies there is some sort of nonlinearity that has not been completely compensated for in the system. One possibility is the fact that the linear part of the fan friction, which is modelled as part of α, is not being compensated for. Another possibility is the fact that the positive fan

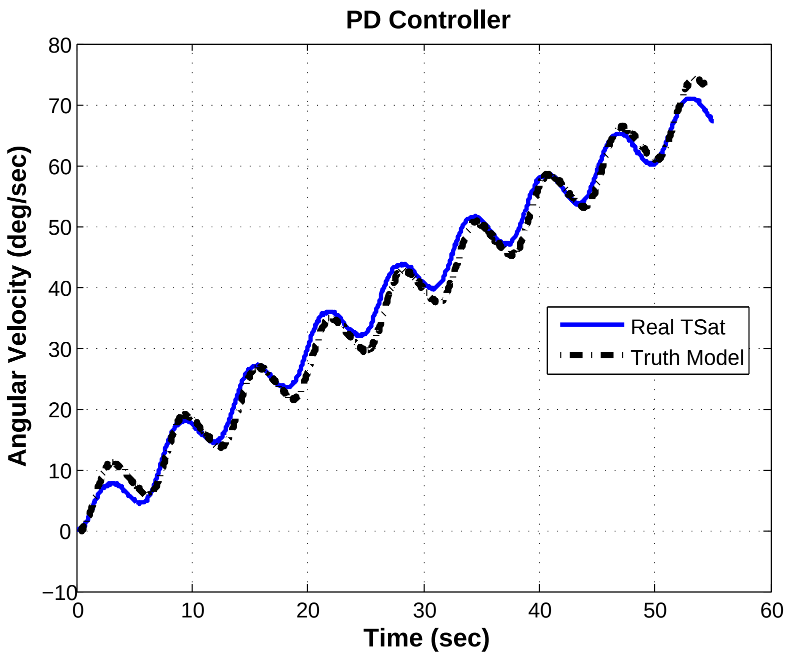


Figure 6.7: Comparison of the truth model and real TableSat system with open loop control and a sinusoidal rate input.

is approximately 1% more powerful than the negative fan. A third possibility is that TableSat and/or fan friction effects at low speeds have not been correctly modelled and thus compensated for. More system analysis is needed to determine how to compensate for these nonlinearities. Until the walk away behavior is compensated for, it is not possible to get open-loop experimental Bode data as was originally desired.

One possible alternative to creating an experimental open-loop Bode diagram, and another way to validate the linearity of the system, is to close a proportional controller around the TableSat rate and command sine waves at varying frequencies to create an experimental closed-loop Bode diagram of TableSat. The sine wave data can then be compared to linear and truth model predictions. A total of 12 different sine wave frequencies was tested. Figure 6.8 shows the input sine waves and output sine waves for the real TableSat system, linear model predictions, and truth model predictions. The input sine wave in this case has a frequency of 0.25 rad/sec and an amplitude of 20 deg/sec. The controller used has a $\mathbf{D1_c}$ matrix of [0.4, 0] and a $\mathbf{D2_c}$ matrix of [0, 0, -0.4, 0, 0]. All other controller matrices are empty matrices.

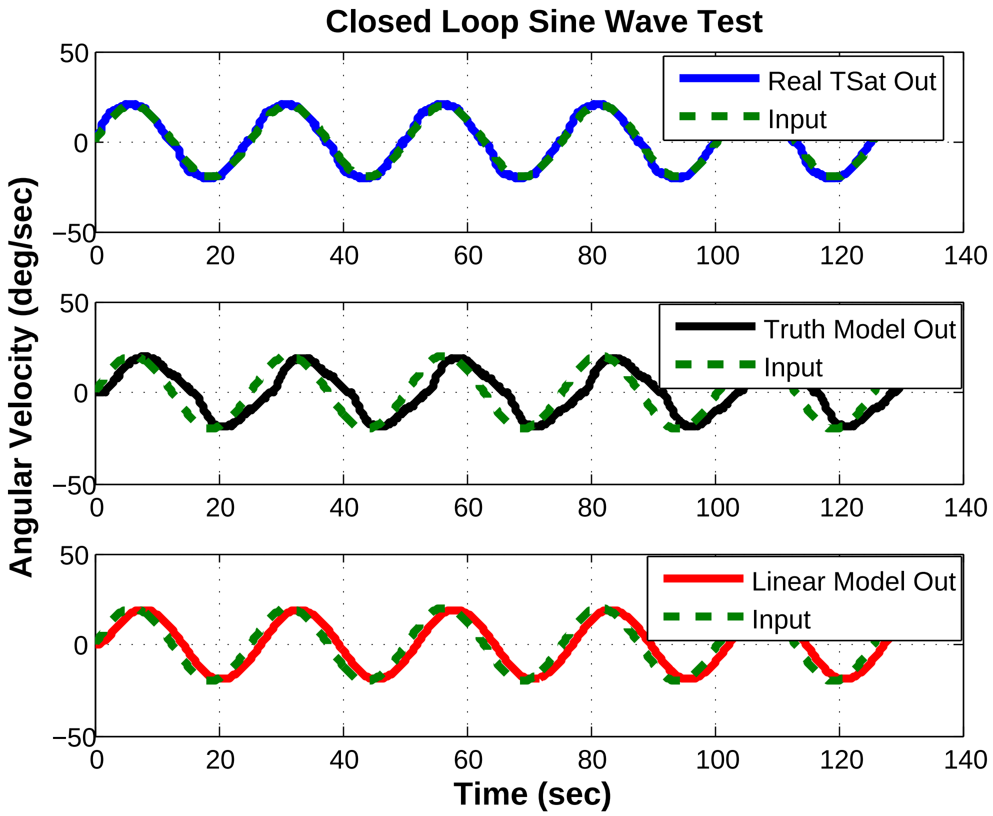


Figure 6.8: Closed-loop control with sinusoidal input with amplitude 20 deg/sec and frequency 0.25 rad/sec. Input is compared to outputs from real TableSat, linear model, and truth model.

As can be seen in the figure, the input sine wave and real TableSat output match quite well in both amplitude and phase. The linear and truth model outputs match well in amplitude, but have a slight phase shift. The reason for this phase shift is unclear. So, at this frequency the system is able to match the commanded sine wave. As the frequency increases, the system's ability to follow the commanded sine wave drops off and there is a decrease in the output amplitude. Figure 6.9 shows the input and output sine waves for the real TableSat system, linear model predictions, and truth model predictions for an input sine wave of amplitude 20 deg/sec and frequency 1.0 rad/sec.

As can be seen in the figure, the output amplitude for all three curves has dropped, implying that the system cannot keep up with a sine wave of this frequency. It is important to note that while the real TableSat and linear model output curves match reasonably well in amplitude, the amplitude of the truth model output is actually about half the amplitude of the other curves. In is unclear why there is this discrepancy between the truth model and the linear model and real TableSat system. More investigation is needed to determine the cause. All three output curves also show a phase shift, but the phase shift for the real TableSat output is smaller than for the truth and linear models. Again, the reason for this discrepancy is unknown. If the output over input magnitudes, in dB, from each of the sine wave tests are calculated and plotted against frequency, we can create an experimental Bode magnitude diagram of

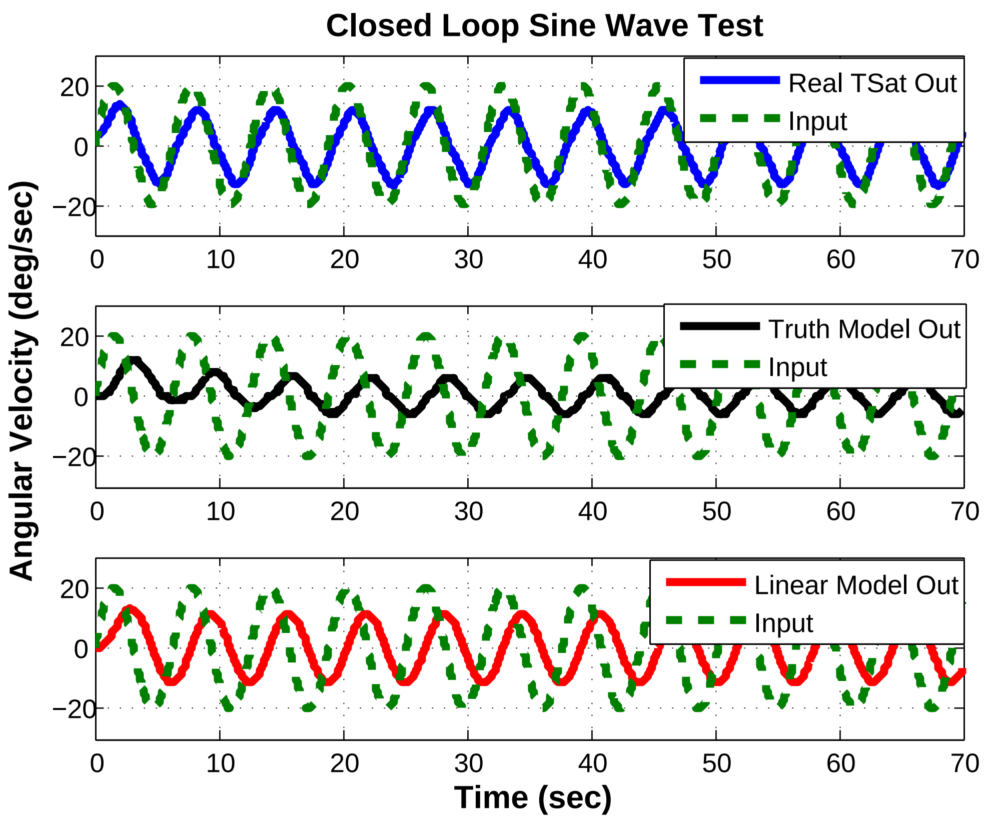


Figure 6.9: Closed-loop control with sinusoidal input with amplitude 20 deg/sec and frequency 1.0 rad/sec. Input is compared to outputs from real TableSat, linear model, and truth model.

this closed-loop system. Figure 6.10 shows the theoretical Bode magnitude plot from the linear model along with the experimental Bode magnitude plot from the real TableSat response.

As can be seen in the figure, the experimental Bode magnitude diagram does not match exactly with the theoretical Bode magnitude plot; however, the curves are reasonably close. Both curves show the same bandwidth, but the real TableSat curve has a resonance that the linear model does not predict. The cause

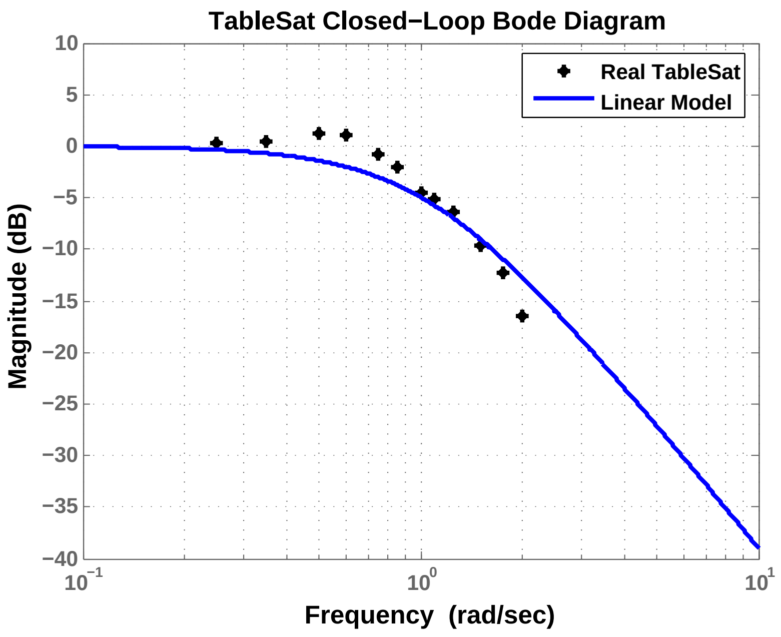


Figure 6.10: Closed-loop Bode magnitude diagram for theoretical prediction and real TableSat response.

of this discrepancy is unknown, but likely stems from the fact that TableSat, as we have already seen, is not completely linear. Another possibility is that TableSat, because of its rather large moment of inertia, has a slight wobble as it spins, *i. e.* its motion is not entirely planer. This wobble could result in a discrepancy in the output magnitude. A second discrepancy between the two curves occurs at the higher frequencies tested and is that the experimental Bode plot drops off in magnitude faster than the theoretical Bode plot. A possible explanation for this discrepancy is that the real TableSat actuators saturate at the higher frequencies, which reduces the output magnitude.

From the comparisons presented above, it is clear that more work is needed to refine the linearization of the TableSat system. However, despite the small discrepancies between the linear model, truth model, and real TableSat response, we feel that with the current nonlinear compensation methods, TableSat captures the major features of a linear response. We therefore assume that the linear model can be used to design controllers and state estimators for TableSat. Chapter 7 presents several case studies using the linear model to design controllers and state estimators.

Chapter 7

Case Studies

In Chapter 6 a simplified, linearized TableSat system model was developed. This chapter presents several case studies that use the linear model to develop controllers for TableSat. These case studies show how the TableSat model is used to develop a Proportional-Derivative (PD), a Model Based Controller-Observer (MBCO), and a MBCO with Integral Augmentation. In addition, the creation of a simple, N-sample averaging estimator and a simplified Kalman Filter are discussed. Results are presented that show how using different estimators can improve steady state pointing. For all case studies, three sets of results are presented: linear model, truth model, and real TableSat results.

7.1 Classical Controller Design

In basic linear control systems theory, students learn how to design linear controllers, given a plant transfer function, using various design methods. The problem with such design exercises is that they are often purely theoretical. Students cannot see the effect of their controller on a real system. They cannot visualize how different design trade-offs result in different controller performances. With TableSat, such design exercises can be much more effective. This section shows how linear design methods, specifically designing via Root Locus can be used to design a simple PD controller for TableSat. The designed controller is then tested on the real TableSat system. Results are presented that compare the predicted results, based on the linear and truth models, to results from the actual TableSat.

7.1.1 Design Methodology

Recall from Chapter 5 that the TableSat transfer function is:

$$G_{TS}(s) = \frac{lK_{wf}K_{v\dot{w}}}{Is^2(s+\alpha)} \tag{7.1}$$

Start the design process by looking at the TableSat open loop plant. From the denominator, it can be seen that TableSat is a marginally stable third order system. The plant has three open loop poles: two at zero and one at $-\alpha$. Most basic linear systems design approaches assume the plant is a second order system. When the system is of higher order, the higher order terms are neglected during controller design. This assumption, however, is only valid if the higher order poles are far into the left hand plan, *i. e.* the higher order poles are much greater than the second order poles. Recall from Chapter 5 that α is approximately equal to 2, which unfortunately means that the third pole is rather close to zero for the system to be treated as a second order system. To compensate for that, a possible design method is to do a dual loop compensator design like that seen in Figure 7.1.

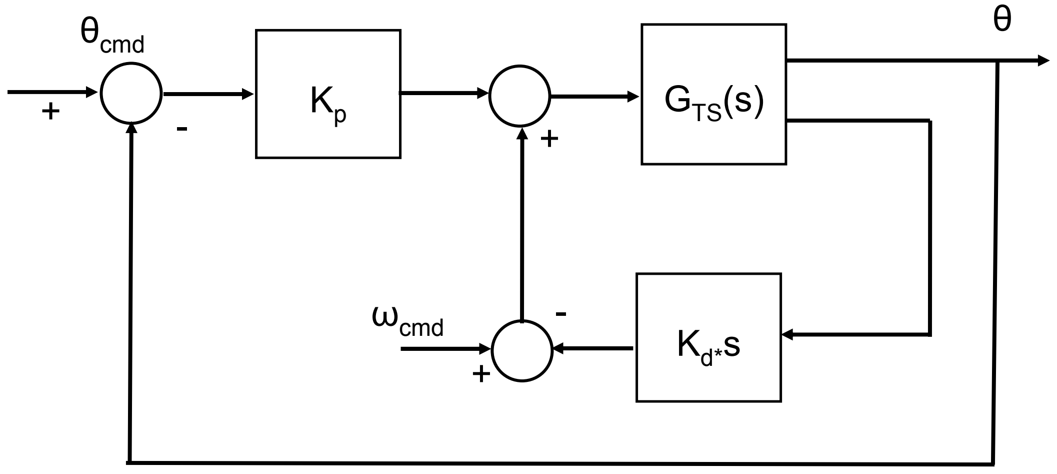


Figure 7.1: Dual loop feedback compensator.

Start by closing a feedback loop on the plant with $C_1(s) = K_d * s$ in the feedback path. Multiplying the plant by this compensator results in the following open-loop transfer function.

$$G_{TS}(s) * C_1(s) = \frac{K_d l K_{wf} K_{v\dot{w}}}{Is(s+\alpha)} \tag{7.2}$$

which is still a marginally stable system with one pole at zero and one pole at $-\alpha$. At this point, because the system is now a second order system, root locus techniques can be used to design K_d so that the new "plant", $G_{TS}(s) * s$, has a certain transient response. In this case, the desired transient response is a percent overshoot of 10% (damping ratio, ζ, of 0.6). This percent overshoot was chosen because TableSat does not have infinite control authority, and the goal is to design a controller that can be realistically applied to TableSat. Figure 7.2 shows the root locus of the plant, $G_{TS}(s) * s$, with the line representing the desired percent overshoot.

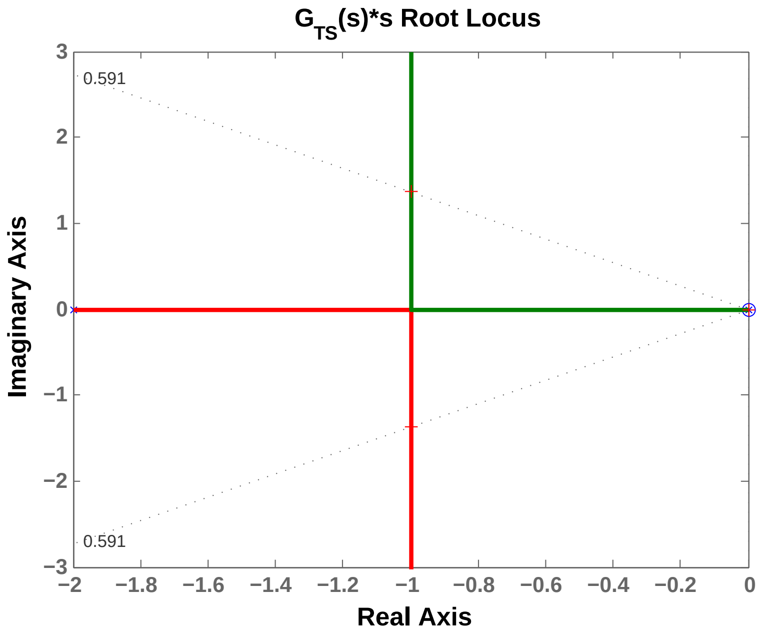


Figure 7.2: Root locus plot of $G_{TS}(s) * s$ with the desired percent overshoot line.

As can be seen in the figure, the current root locus does cross the desired percent overshoot line, so the desired K_d will be the gain that results at the point where the root locus crosses the desired percent overshoot line, and can be found using the MATLAB command *rlocfind*. The resulting gain is $K_d = 1.0$. Now that we have found the inner loop gain, we can find the closed-loop transfer function, $L(s)$, for the inner loop by reducing the inner loop to a single transfer function:

$$L(s) = \frac{K_d l K_{wf} K_{v\dot{w}}}{Is^2 + Is\alpha + K_d l K_{wf} K_{v\dot{w}}} \tag{7.3}$$

where $L(s)$ can now be used to design the outer loop compensator, $C_2(s) = K_p$. The root locus of the resulting transfer function can be seen in Figure 7.3. We wish to maintain the current design point, so use *rlocfind* to select a gain that is close to the open loop poles. The resulting gain is $K_p = 0.25$, and the resulting closed-loop poles are p_1, $p_2 = -0.85 \pm 1.3$ and $p_3 = -0.303$.

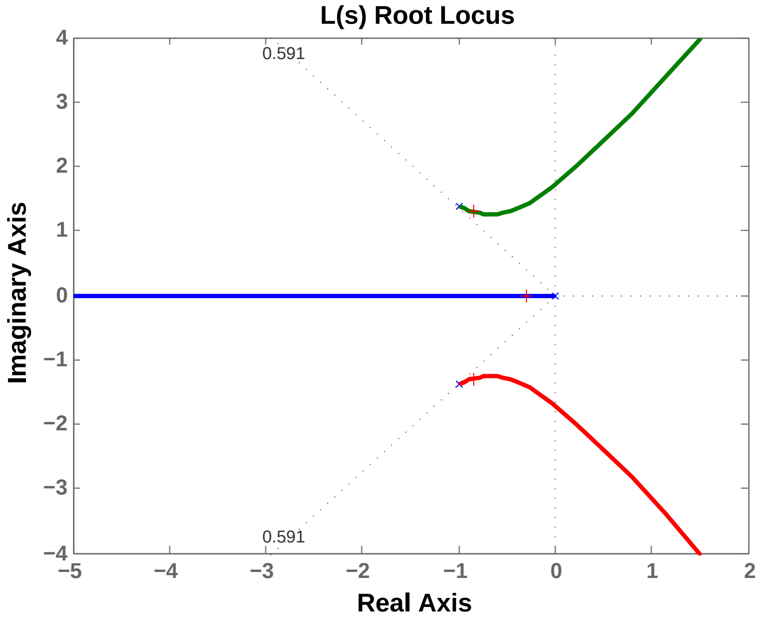


Figure 7.3: Root locus plot of $L(s)$ with closed-loop poles for chosen K_p.

Combining C_1 and C_2, the total controller transfer function is then:

$$C_{tot} = K_d s + K_p \tag{7.4}$$

which can be put into the state space form used by the TableSat flight code such that:

$$\begin{aligned} &n_c = 0 \\ &\mathbf{A} = [\,] \\ &\mathbf{B1} = [\,] \\ &\mathbf{B2} = [\,] \\ &\mathbf{C} = [\\ &\mathbf{D1} = \begin{bmatrix} 0 & -K_p & -K_d & 0 & 0 \end{bmatrix} \\ &\mathbf{D2} = \begin{bmatrix} K_p & 0 \end{bmatrix} \end{aligned} \tag{7.5}$$

7.1.2 Controller Results

The PD controller was tested on the linear model, truth model, and the real TableSat system with a simple pass through estimator. Figure 7.4 shows the predicted results for a step input of 20 degrees using the linear model.

As can be seen in the figure, the linear model predicts an overdamped response, which is not what was anticipated. To explain this result, look again at the closed-loop poles. The closed-loop system is a third order system, and all three poles are close to zero. The fact that these poles are so close together implies we cannot expect a second order, underdamped response. Despite the overdamped response, the steady state value is still reached fairly quickly, with the steady state being reached in about 20 seconds. The linear model predicts no steady state error. Figure 7.5 shows the truth model prediction plotted with

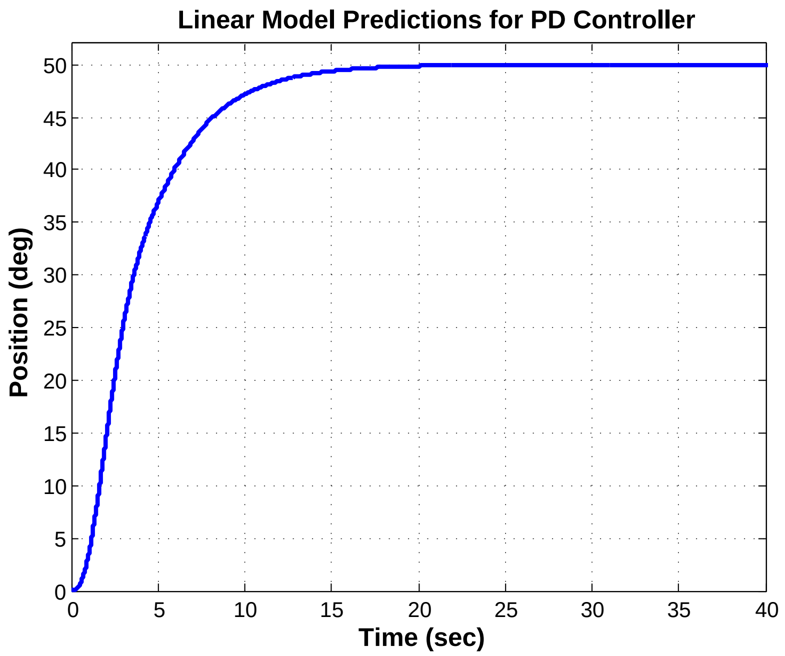


Figure 7.4: Predicted results for a step input of 50 degrees using the PD controller.

the real TableSat response for a step input of 50 degrees.

As can be seen in the figure, the truth model responds quicker than the linear model prediction, reaching its steady state value in under 10 seconds, but like the prediction, shows no steady state error. The real TableSat results, also respond faster than the linear model prediction, reaching steady state in about 5 seconds. However, unlike the linear model or truth model, the real TableSat results show a rather large steady state error of about six degrees. One possible reason for this discrepancy is that the friction in the real TableSat system is not completely compensated for. With the gains chosen, a six degree steady state error would result in a commanded voltage of about 1.5 Volts. In a linear system,

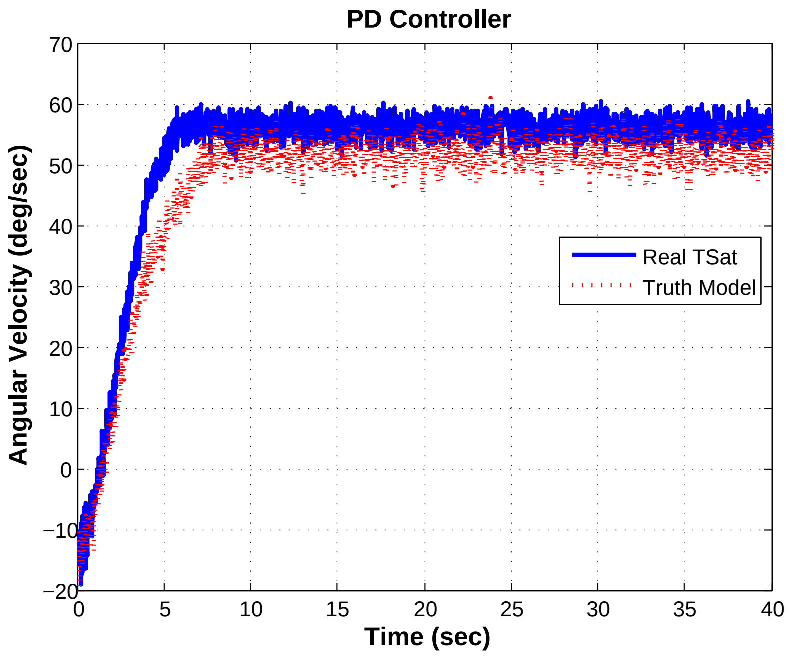


Figure 7.5: Truth model prediction and real TableSat results for a step input of 50 degrees using the PD controller.

this voltage should be enough to move TableSat to reduce the steady state error. Figure 7.6 shows the truth model response for a step command of six degrees with the given PD gains.

As can be seen in the figure, the truth model is also able to respond to such a step command. In the real TableSat system, however, it appears that at low speeds and low voltage commands, the system does not act in a linear manner. If the low voltage commands are the reason for the discrepancy between the truth model and real TableSat response, then using larger values for K_p and K_d should result in better agreement between the truth model and real TableSat results. To that end, new gain values of $K_p = 5$ and $K_d = 19.6$ were chosen so that even small

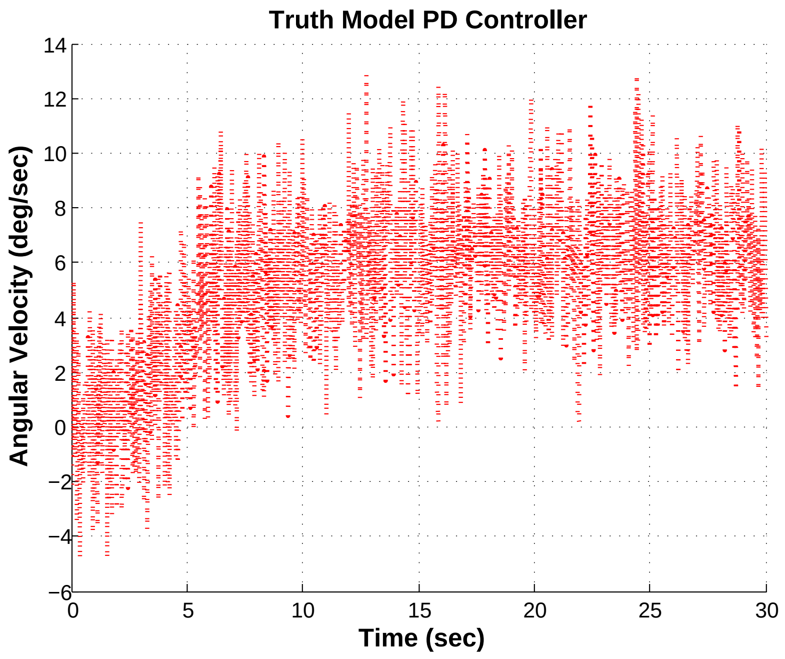


Figure 7.6: Truth model prediction for a step input of 6 degrees using the PD controller.

pointing errors would result in relatively large voltage commands. The selection of these new gains results in a system with closed-loop poles of -0.87 ± 1.3 and -0.25. Figure 7.7 shows the linear model prediction for this new PD controller with the same 50 degree step input.

As can be seen in the figure, this new PD controller also has an over-damped response that looks similar to the original PD controller. This similarity in response makes sense when comparing the closed loop poles between the two designs. The new closed-loop poles are not much different than the original closed-loop poles. The new controller does have a slightly slower response than the original PD controller, reaching its steady state in just over 20 seconds. This

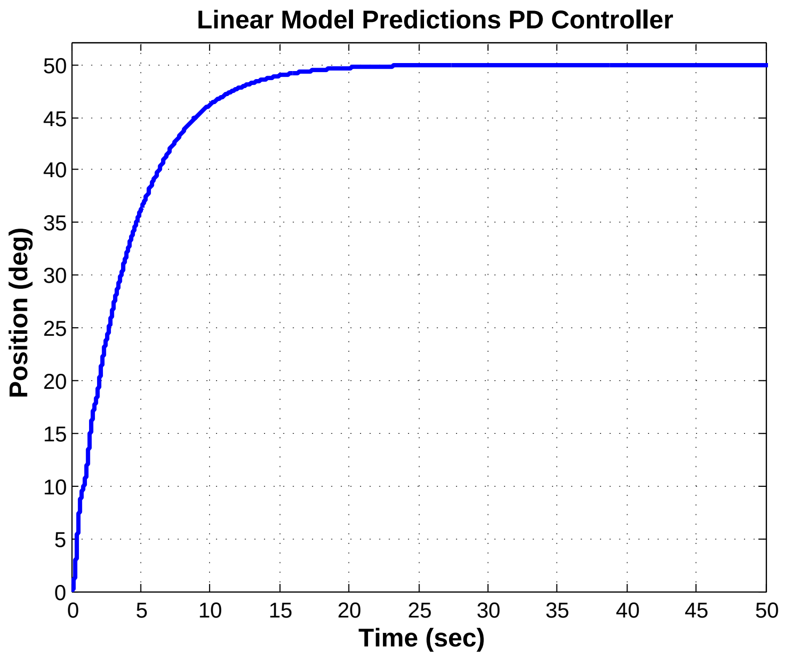


Figure 7.7: Linear model prediction for a step input of 50 degrees using new PD controller with $K_p = 5$ and $K_d = 19.6$.

slower response is to be expected considering the relatively high value of K_d. It is difficult to see in this plot because of the thickness of the line, but there also appears to be an oscillation in the transient response. As before, the linear model shows no steady state error. Figure 7.8 shows the truth model prediction and real TableSat results for the new PD controller.

As can be seen in the figure, there is much better agreement between the truth model and real TableSat results for this controller, most likely due to the higher gains giving larger voltage commands for small errors. There are some slight discrepancies between the truth model and TableSat results and the linear model predictions. Both the truth model and TableSat show a slightly faster

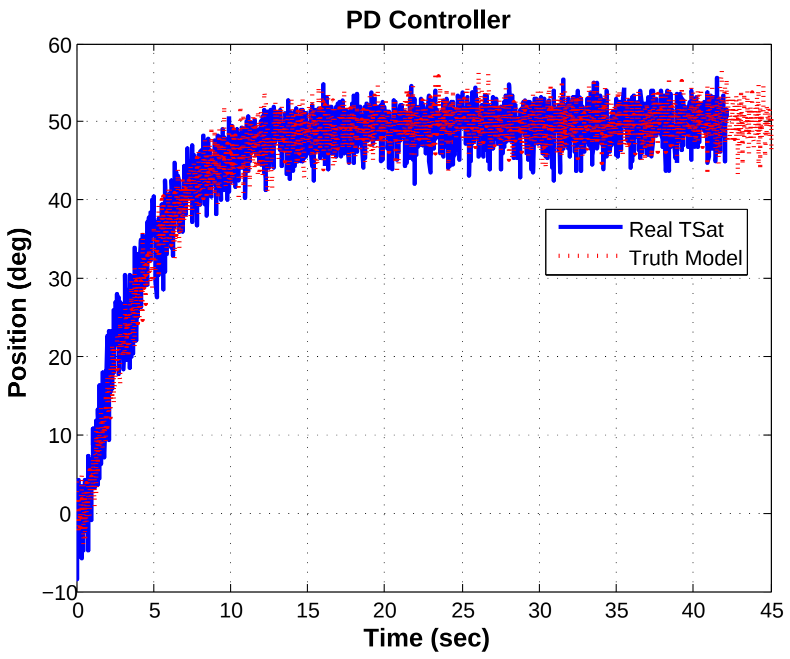


Figure 7.8: Truth model prediction and real TableSat results for a step input of 50 degrees using the new PD controller with $K_p = 5$ and $K_d = 19.6$.

response than the linear model prediction, reaching steady state in about 12 seconds. Recall that this discrepancy in the transient response was also seen with the original PD controller. It is unclear at this time why the truth model and real TableSat respond faster than the linear model prediction. More analysis is needed to determine the cause.

Both the truth model and real TableSat results also show a slight steady state error in pointing, on the order of about 3 degrees. The linear model prediction showed no steady state error. One possible explanation for this steady state error is due to the fan resolution. As mentioned before, the linear model assumes infinite control. Not only does infinite control mean no upper bound on the amount of control the system can receive, it also implies the system can receive infinite resolution. The system gets exactly the amount of control it asks for. As the system approaches its target, it will ask for less and less control, which could potentially be a problem given that there is a finite amount of control resolution in the real actuators. Recall from Chapter 2 that the real TableSat system has a 12 bit D/A converter. For the voltage range of the fans and the voltage regulator circuit used, this implies a resolution of 2.93 mV. Control commands to the fans can only change in 2.93 mV increments. It is possible that the fans cannot give the level of control requested. However, the actuator resolution is not modelled in the truth model, and, since the truth model and real TableSat system responses are almost identical, the actuator resolution cannot be the reason for the steady state error. Furthermore, as was seen previously, due to the imperfect friction compensation at low speeds and low voltages, TableSat cannot even respond to voltages on the level of the fan resolution. A more likely explanation for the steady state error is that there is a nonlinear interaction between the noise and the friction compensation, resulting in friction not being completely compensated for. For example, because the TableSat friction compensation is calculated based on the angular velocity estimate, a noisy estimate can result in improper friction compensation. If noisy state estimates are the cause for the steady state errors, then improving the state estimate should improve the steady state pointing.

The above controller used the magnetometer as the estimate of the TableSat attitude. The same PD gains can be used to control TableSat using the CSS signals. Figure 7.9 shows the truth model and real TableSat results using the new PD controller with the CSS angle as the attitude estimate. As can be seen in the figure, the truth model and real TableSat results agree quite well. The truth model shows more noise, but, as was discussed in Chapter 5 the truth model CSS noise is only an estimate of the system noise. Once the CSS's have been permanently attached to the CSS head, a more realistic noise value can be determined. As was the case when using the magnetometer, the results show a slight steady state error and a faster transient response than the linear model prediction.

7.2 Sensor Noise Reduction: State Estimators

It was hypothesized in the above section that the likely contributor to the steady state pointing error seen by the new PD controller is the noise in the state estimate. Because the controller uses the default TableSat state estimator to obtain its state, the sensor measurements get passed directly through to the controller without any sort of smoothing of the signals. Some of the steady state errors in the TableSat pointing could conceivably be reduced by employing a state estimator to feed better estimates of the state to the controller. One estimator that is simple to employ is an N-sample averaging estimator. A slightly more

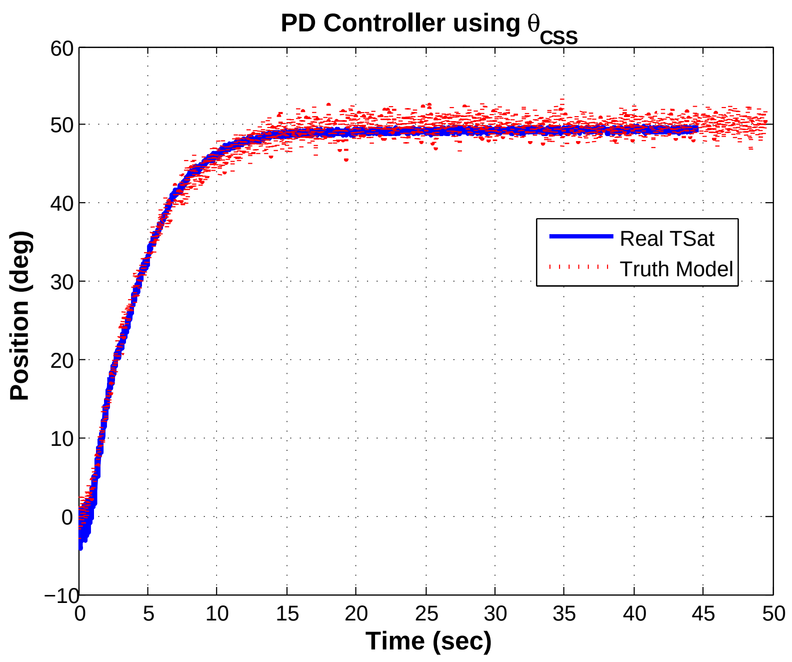


Figure 7.9: Truth model prediction and real TableSat results for a step input of 50 degrees using the new PD controller with θ_{CSS} as the attitude estimate.

complicated estimator presented in this section is a simplified, kinematic Kalman Filter.

7.2.1 N-Sample Estimator Design Methodology

The concept of the N-sample averaging estimator is simple: average the last N state measurements to reduce the noise on the estimate. Theory states that for an N-sample averaging estimator, noise on the estimator can be reduced by a factor of $\sqrt{N}$. The difficulty in the design lies in how to incorporate that simple concept using the TableSat state estimator definition, which has the form:

$$\begin{aligned}\mathbf{ze}_{k+1} &= \mathbf{A_e}\mathbf{ze}_k + \mathbf{B_e}\hat{\mathbf{x}}\_\mathbf{raw}_k \\ \hat{\mathbf{x}}_{k+1} &= \mathbf{C_e}\mathbf{ze}_k + \mathbf{D_e}\hat{\mathbf{x}}\_\mathbf{raw}_k\end{aligned} \tag{7.6}$$

where $\mathbf{ze}$ is the estimator states, $\hat{\mathbf{x}}$ is the state estimate, $\hat{\mathbf{x}}\_\mathbf{raw_k}$ is the vector of raw sensor measurements, and $\mathbf{A_e}$, $\mathbf{B_e}$, $\mathbf{C_e}$, and $\mathbf{D_e}$ are the matrices we are trying to find.

For simplicity, start by assuming a two sample estimator, *i. e.* $N = 2$. The state estimate would be:

$$\mathbf{x}_{k+1} = \frac{\mathbf{x}_k + \mathbf{x}_{k-1}}{2} = \frac{1}{2}\mathbf{x}_k + \frac{1}{2}\mathbf{x}_{k-1} \tag{7.7}$$

By inspection, it is easy to see that $\mathbf{D_e} = \frac{1}{2}I_{5\text{x}5}$, where I_{5x5} is the 5x5 identity matrix because there are 5 states. Furthermore, $\mathbf{z}_k = \mathbf{x}_{k-1}$, so that n_o, or the number of estimator states is 5 and $\mathbf{C_e} = \frac{1}{2}I_{5\text{x}5}$. Finally, if $\mathbf{z}_k = \mathbf{x}_{k-1}$, then $\mathbf{z}_{k+1} = \mathbf{x}_k$; therefore $\mathbf{A_e} = 0_{5\text{x}5}$ and $\mathbf{B_e} = I_{5\text{x}5}$.

Next, extend the definition to a three sample filter, *i. e.* $N = 3$. The state estimate would be:

$$\mathbf{x}_{k+1} = \frac{\mathbf{x}_k + \mathbf{x}_{k-1} + \mathbf{x}_{k-2}}{3} = \frac{1}{3}\mathbf{x}_k + \frac{1}{3}\mathbf{x}_{k-1} + \frac{1}{3}\mathbf{x}_{k-2} \tag{7.8}$$

Again, by inspection it is easy to see that $\mathbf{D_e} = \frac{1}{3}I_{5\text{x}5}$. Furthermore, $\mathbf{C_e}z_k = \frac{1}{3}(\mathbf{x}_{k-1} + \mathbf{x}_{k-2})$, which implies that:

$$\mathbf{z}_k = \begin{bmatrix} \mathbf{x}_{k-1} \\ \mathbf{x}_{k-2} \end{bmatrix} \tag{7.9}$$

and $\mathbf{C_e} = \begin{bmatrix} \frac{1}{3}I_{5\text{x}5} & \frac{1}{3}I_{5\text{x}5} \end{bmatrix}$. If $\mathbf{z}_k$ is defined as above, then $\mathbf{z}_{k+1} = \begin{bmatrix} \mathbf{x}_k & \mathbf{x}_{k-1} \end{bmatrix}^T$, which implies that $n_o = 10$ and $\mathbf{A_e}$ and $\mathbf{B_e}$ are defined as follows:

$$\begin{aligned} \mathbf{A_e} &= \begin{bmatrix} \frac{1}{3}I_{5\text{x}5} & 0_{5\text{x}5} \\ 0_{5\text{x}5} & 0_{5\text{x}5} \end{bmatrix} \\ \mathbf{B_e} &= \begin{bmatrix} I_{5\text{x}5} \\ 0_{5\text{x}5} \end{bmatrix} \end{aligned} \tag{7.10}$$

So, we have found the matrix definitions for two and three sample estimator. If we continue in this manner, a generic, N-sample estimator can be defined as follows:

$$\begin{aligned} & n_o = 5 * (N-1) \\ & \mathbf{A_e} = \begin{bmatrix} 0_{5\text{x}5} & 0_{5\text{x}5} & 0_{5\text{x}5} & \cdots & 0_{5\text{x}5} \\ I_{5\text{x}5} & 0_{5\text{x}5} & 0_{5\text{x}5} & \cdots & 0_{5\text{x}5} \\ 0_{5\text{x}5} & I_{5\text{x}5} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0_{5\text{x}5} & 0_{5\text{x}5} & 0_{5\text{x}5} & I_{5\text{x}5} & 0_{5\text{x}5} \end{bmatrix}_{5(N-1)\text{x}5(N-1)} \\ & \mathbf{B_e} = \begin{bmatrix} I_{5\text{x}5} \\ 0_{5(N-2)\text{x}5} \end{bmatrix}_{5(N-1)\text{x}5} \\ & \mathbf{C_e} = \begin{bmatrix} \frac{1}{N}I_{5\text{x}5} & \cdots & \frac{1}{N}I_{5\text{x}5} \end{bmatrix}_{5\text{x}5(N-1)} \\ & \mathbf{D_e} = \frac{1}{N}I_{5\text{x}5} \end{aligned} \tag{7.11}$$

Now that the estimator matrices have been established, different sample estimators can be implemented on TableSat to determine their effect on TableSat pointing. The following results discuss the differences between using a pass through estimator, a 2-sample averaging estimator, and a 5-sample averaging estimator along with the new PD controller. Results from both the truth model and the real TableSat system are compared.

7.2.2 N-Sample Estimator Results

According to the Central Limit Theorem of statistics, as you sample a population with mean, μ, and finite standard deviation, σ, the mean of n independent random observations tends towards a normal distribution with mean, μ and standard deviation, $\sigma_m = \sigma/\sqrt{n}$ [5]. This theorem implies that because the noise in the sensor measurements can be assumed random with zero mean and standard deviation σ, then, for a sampling estimator, σ will be reduced by the square root of N, where N is the number of samples in the estimator. Of course, there are limits on how much an estimate can improve, given the sampling rate of the estimator. At some point, because the system is being actively controlled, the samples are too old to actual improve the estimate, and can, in fact, make the estimate worse. For TableSat, we decided to try 2-sample and 5-sample estimators.

Recall from Section 7.1 that the new PD controller, coupled with a simple pass through estimator, resulted in a steady state error of about 3 degrees. It was hypothesized that one of the causes of this steady state error is the noise in the sensor measurements. If that is true, then reducing the noise should also reduce the steady state error. For the angle measurement, θ_{TAM}, the standard deviation of the noise is 2.3 degrees for both the truth model and the real TableSat system.

Figure 7.10 shows the results when the pass through estimator is replaced by a 2-sample averaging estimator.

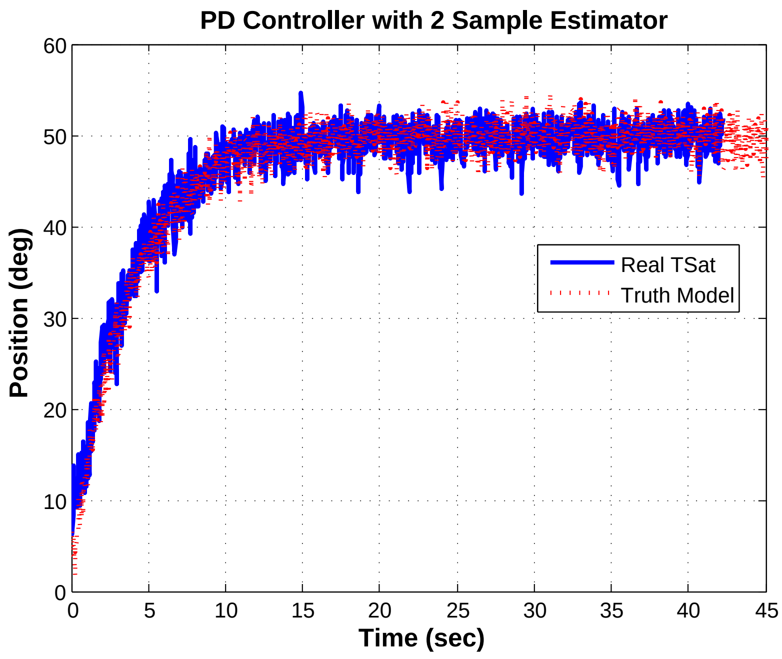


Figure 7.10: Predicted results and real TableSat results using the new PD controller and a 2-sample averaging estimator.

As can be seen in the figure, the system slowly approaches its desired target of 50 degrees. The steady state value is reached after about 12 seconds, which is about the same time as when the pass through estimator was used. However, in this case the steady state error has decreased to about 1 degree in both the truth model and real TableSat system. This reduction in steady state error supports the hypothesis that the sensor noise is causing most of the steady state error when using this controller. The standard deviation in both the truth model and real TableSat measurements is 1.6 degrees, which is almost exactly a reduction by $\sqrt{2}$ from the pass through estimator.

So, since the 2-sample averaging estimator cut the steady state error by more than half, the 5-sample averaging estimator should reduce the steady state error even more. Figure 7.11 shows the system response when the 2-sample estimator is replaced with a 5-sample estimator.

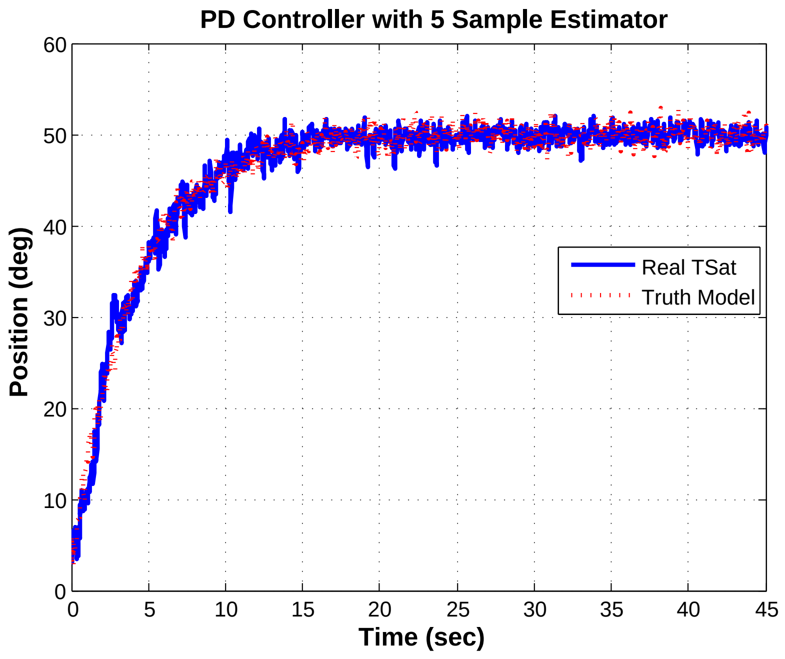


Figure 7.11: Predicted results and real TableSat results using the new PD controller and a 5-sample averaging estimator.

As can be seen in the figure, the system reaches its steady state value in about 15 seconds, which is slightly longer than the time it took in the previous example. However, the steady state error is reduced to nothing. The standard deviation of the angle measurement is about 1.0 degrees for the truth model and 0.8 degrees for the real TableSat measurement. For the truth model, this standard deviation is a reduction by almost exactly $\sqrt{5}$. For the real TableSat, the reduction in noise is actually greater than the predicted $\sqrt{5}$. It is possible that implementing an estimator with even more samples could reduce the noise levels even more. However, as was mentioned before, at some point the larger number of samples will actually degrade the performance because the measurements become too old to be of value in improving the estimate. Regardless, these results shows that a simple averaging estimator can eliminate steady state error for the new PD controller.

7.2.3 Kalman Filter Design Methodology

In Section 7.2.1 we saw that implementing a simple averaging estimator can reduce steady state error and reduce the standard deviation of the sensor measurements. We now wish to implement a more advanced filter to try and smooth our state estimates even more and perhaps improve performance. The Kalman Filter, first proposed by R. E. Kalman in 1960, is a set of mathematical equations that recursively estimates the state of a process by minimizing the mean of the squared error. It is very powerful in that it can predict past, present, and even future states even if the exact nature of the modelled system are unknown [9]. It is based on a probabilistic treatment of process and measurement noises. The traditional Kalman Filter design uses the system state equations:

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} + \mathbf{\Gamma}\mathbf{w_1}\\ \mathbf{y} &= \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u} + \mathbf{w_2}\end{aligned} \tag{7.12}$$

where $\mathbf{w_1}$ and $\mathbf{w_2}$ represent the uncertainties in plant dynamics and outputs, respectively, and are Gaussian distributed white noise, which are uncorrelated with respect to time. $\mathbf{\Gamma}$ is the matrix that reflects how the model uncertainties are reflected in the plant dynamics. Furthermore,

$$\begin{aligned}\mathrm{E}[\mathbf{w_1}(t)\mathbf{w_1}(t)^{\mathrm{T}}] &= \mathbf{W_1}\\ \mathrm{E}[\mathbf{w_2}(t)\mathbf{w_2}(t)^{\mathrm{T}}] &= \mathbf{W_2}\\ \mathrm{E}[\mathbf{w_1}(t)\mathbf{w_2}(t)^{\mathrm{T}}] &= 0\end{aligned} \tag{7.13}$$

where $\mathbf{W_1}$ and $\mathbf{W_2}$ are symmetric, positive definite matrices representing the variance of the model uncertainty and output uncertainty, respectively. The goal is then to design an estimator that minimizes the cost function:

$$\mathrm{J}[\hat{\mathbf{x}}] = \lim_{t\to\infty} \mathrm{trace}\mathrm{E}[\tilde{\mathbf{x}}(t)\tilde{\mathbf{x}}(\mathbf{t})^{\mathrm{T}}] \tag{7.14}$$

where $\tilde{\mathbf{x}}(t) = \mathbf{x}(t) - \hat{\mathbf{x}}(t)$. In other words, the estimator uses the knowledge of the system dynamic equations, along with the control inputs to predict the future values of the state. The resulting state dynamics with the estimator are then:

$$\begin{aligned}\dot{\hat{\mathbf{x}}} &= (\mathbf{A} - \mathbf{L}\mathbf{C})\hat{\mathbf{x}} + \mathbf{B}\mathbf{u} + \mathbf{L}\mathbf{y}\\ \hat{\mathbf{y}} &= \mathbf{C}\hat{\mathbf{x}} + \mathbf{D}\mathbf{u}\end{aligned} \tag{7.15}$$

where $\mathbf{L} = \mathbf{P_o}\mathbf{C}^{\mathrm{T}}\mathbf{W_2}^{-1}$ is the observer gain matrix and P_o is the symmetric, positive definite solution to the Algebraic Riccati Equation:

$$\mathbf{AP_o} + \mathbf{P_o A}^T = -\mathbf{\Gamma W_1 \Gamma}^T - \mathbf{P_o C}^T \mathbf{W_2}^{-1} \mathbf{CP_o} \tag{7.16}$$

With the current arrangement of the generic TableSat state estimator equation, the state estimator does not have access to the calculated control voltages. Therefore, we cannot implement a full Kalman Filter estimator. Instead, consider a simplified, "kinematic" Kalman filter with the state equations:

$$\begin{gathered} \dot{\mathbf{x}} = \mathbf{Ax} + \mathbf{b}v + \mathbf{\Gamma} w_1 \\ \begin{bmatrix} \dot{\theta} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \theta \\ \omega \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 1 \end{bmatrix} w_1 \end{gathered} \tag{7.17}$$

$$\begin{gathered} \mathbf{y} = \mathbf{Cx} + \mathbf{w_2} \\ \begin{bmatrix} \tilde{\theta} \\ \tilde{\omega} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \theta \\ \omega \end{bmatrix} + \mathbf{w_2} \end{gathered} \tag{7.18}$$

where we are using the pure kinematic relationship between θ and ω as the plant dynamics, and we neglect the control input into the system. As mentioned before, w_1 represents the uncertainty in plant dynamics and $\mathbf{w_2}$ represents the uncertainty in the system outputs. For TableSat, the corresponding uncertainty matrices were assumed to be:

$$\begin{gathered} W_1 = 1 \\ \mathbf{W_2} = \begin{bmatrix} (1.5)^2\,\text{deg}^2 & 0 \\ (0.1)^2\,\text{deg/sec}^2 & \end{bmatrix} \end{gathered} \tag{7.19}$$

The MATLAB command *kalman* was then used to find the Kalman Filter gain, **L**. *kalman* accepts as inputs the state matrices, **A**, $[\;\mathbf{B}\;\;\mathbf{\Gamma}\;]$, **C**, and **D**;

and returns, among other outputs, the Kalman gain, **L**. Once the Kalman gain has been determined, the whole Kalman Filter must be written in the form of the TableSat state estimator matrices. By inspection, these matrices can be seen to be:

$$\begin{aligned}
&n_o = 2 \\
&\mathbf{A_e} = \mathbf{A} - \mathbf{LC} \\
&\mathbf{B_e} = \begin{bmatrix} 0_{2\text{x}1} & \mathbf{L} & 0_{2\text{x}2} \end{bmatrix} \\
&\mathbf{C_e} = \begin{bmatrix} 0_{1\text{x}2} \\ \mathbf{C} \\ 0_{2\text{x}2} \end{bmatrix} \\
&\mathbf{D_e} = 0_{5\text{x}5}
\end{aligned} \tag{7.20}$$

where the extra zeros in the $\mathbf{B_e}$ and $\mathbf{C_e}$ matrices correspond with the fact that the TableSat state estimate, $\hat{\mathbf{x}}$ is of length 5. Up until this point, the Kalman Filter design process has assumed a continuous system. TableSat, however, is a discrete system. The resulting Kalman Filter matrices, in TableSat state estimator form, can be converted into discrete time using the MATLAB function, *c2d*. We have assumed the estimator thread will be running at 50 Hz; however any time step may be chosen as long as it is within the capability of TableSat's processor speed. The resulting estimator definition for the kinematic Kalman Filter is then:

$$
\begin{aligned}
&n_o = 2 \\
&\mathbf{A_e} = \begin{bmatrix} 0.9987 & -0.0017 \\ -0.0001 & 0.8190 \end{bmatrix} \\
&\mathbf{B_e} = \begin{bmatrix} 0 & 0.0013 & 0.0217 & 0 & 0 \\ 0 & 0.0001 & 0.1810 & 0 & 0 \end{bmatrix} \\
&\mathbf{C_e} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \\
&\mathbf{D_e} = 0_{5\text{x}5}
\end{aligned}
\qquad (7.21)
$$

7.2.4 Kalman Filter Estimator Results

Now that the TableSat Kalman Filter has been designed, it can be tested using the same PD controller used in Section 7.2.2. Figure 7.12 shows the truth model and real TableSat results for the new PD controller with Kalman Filter estimator.

As can be seen in the figure, the θ_{TAM} estimates are much smoother using the Kalman Filter than using the N-sample estimators. The standard deviation of the truth model noise is 0.13 degrees and the standard deviation of the real TableSat results is 0.33 degrees. With the noise in the estimates virtually eliminated, an oscillation in the transient response of both the truth model and real TableSat results is clearly visible. Recall from Figure 7.7, that the linear model predictions using this controller also showed an oscillation in the transient response. That oscillation could not be seen in the unfiltered estimates or even after using the N-sample estimators, but with the Kalman Filter reducing noise so drastically,

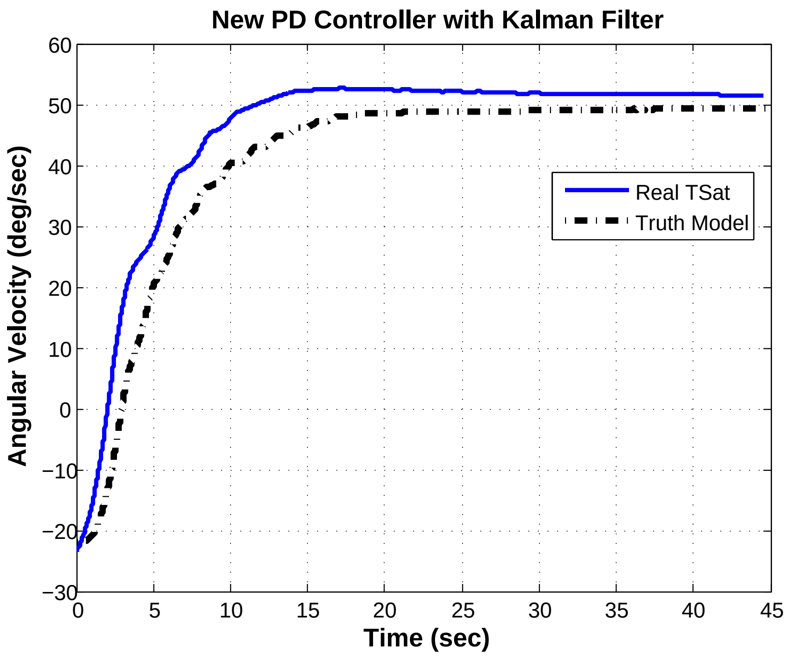


Figure 7.12: Predicted results and real TableSat results using the new PD controller and the Kalman Filter estimator.

the oscillation is now apparent in the actual TableSat results. This oscillation in the transient response is caused by the lightly damped, non-dominant poles in the closed-loop system.

While the noise reduction when using the Kalman Filter is much better than the noise reduction using N-sample estimators, the agreement between the truth model and real TableSat results is not as good. Furthermore, the real TableSat performance appears to degrade slightly. The real TableSat results show a steady state error of about 1.5 degrees, while the truth model still shows no steady state error. This discrepancy is, again, probably a result of imperfect friction modelling, and thus friction compensation, at low speeds and low voltages.

7.3 Model Based Controller-Observer

In Section 7.2.3 we designed a simplified, kinematic Kalman Filter that assumed a purely kinematic relationship between θ and ω and neglected the system control inputs and their affect on ω. We designed the Kalman Filter in this manner, because the TableSat estimator does not have access to the calculated control inputs. It would be nice, to be able to design and implement a full Kalman Filter. One possibility for doing so would be to rewrite the flight code to allow the state estimator thread access to the control voltages calculated by the controller thread. A better solution that uses the existing TableSat functionality is to use the TableSat state space model to design a Model Based Controller-Observer (MBCO). This solution has the further advantage that it also shows how advanced, multi-input, multi-output (MIMO) linear theory can be used to design and implement a controller for TableSat. This section describes the methodology for using the TableSat plant model to design a MBCO, which can be implemented on the real TableSat. Results are then shown that compare predicted results to actual TableSat data.

7.3.1 Design Methodology

Recall from Chapter 6 that the TableSat simplified state space model is:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{b}v \\ \mathbf{y} &= \mathbf{C}\mathbf{x} \end{aligned} \tag{7.22}$$

with:

$$\begin{aligned} \mathbf{A} &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & \frac{lK_{\dot{w}f}}{I} \\ 0 & 0 & -\alpha \end{bmatrix} \\ \mathbf{b} &= \begin{bmatrix} 0 \\ 0 \\ K_{vf} \end{bmatrix} \\ \mathbf{C} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \\ \mathbf{D} &= 0 \end{aligned} \tag{7.23}$$

and the system state is $x = [\theta, \omega, \nu]^T$. Before designing a model based controller, first check to see if the system is controllable. For $[\mathbf{A}, \mathbf{B}]$ to be controllable, the matrix,

$$\mathbf{C_c} = [\ \mathbf{B} \quad \mathbf{AB} \quad \mathbf{A}^2\mathbf{B} \quad \cdots \quad \mathbf{A}^{n-1}\mathbf{B}\ ] \tag{7.24}$$

must have full rank, n. Since TableSat has three states, the controllability matrix, $\mathbf{C_c} = [\ \mathbf{b} \quad \mathbf{Ab}\ ]$ must have rank three. Using the $\mathbf{A}$ and $\mathbf{b}$ matrices from Equation 7.23, we find that $\mathbf{C_c}$ does have rank three, and the system is controllable.

Since the system is controllable, we can design a controller to put the closed loop system poles anywhere in the left hand plane. The proposed control law is

$v = -\mathbf{Kx}$, where v is the commanded voltage and K is a gain matrix to be designed. This control law assumes the full state is available for feedback. Note, however, from the $\mathbf{C}$ matrix that the fan speed, ν is not directly measurable. The system, therefore, does not have full state feedback, and an observer is needed to estimate the missing state.

In order to design an estimator, the system must be observable. To check observability, the observability matrix, O, must have full rank, where:

$$\mathbf{O} = \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ \mathbf{CA}^2 \\ \vdots \\ \mathbf{CA}^{n-1} \end{bmatrix} \tag{7.25}$$

Using the $\mathbf{A}$ and $\mathbf{C}$ matrices from Equation 7.23, the rank of $\mathbf{O}$ is found to be three, and the system is observable. The proposed state estimator is of the form:

$$\begin{aligned} \dot{\hat{\mathbf{x}}} &= \mathbf{A}\hat{\mathbf{x}} + \mathbf{b}v + \mathbf{L}(\mathbf{y} - \hat{\mathbf{y}}) - \mathbf{L}\mathbf{y_d} \\ \hat{\mathbf{y}} &= \mathbf{C}\hat{\mathbf{x}} \end{aligned} \tag{7.26}$$

where $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ are the estimates of the state and output, respectively, $\mathbf{y_d}$ is the desired output, and $\mathbf{L}$ is the constant gain matrix used to adjust the state estimates. Combining Equations 7.22 and 7.26 with the new control law, $v = -\mathbf{K}\hat{\mathbf{x}}$ yields the following closed loop dynamics:

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{A} - \mathbf{bK}\hat{\mathbf{x}} \\ \dot{\hat{\mathbf{x}}} &= (\mathbf{A} - \mathbf{bK} - \mathbf{LC})\hat{\mathbf{x}} + \mathbf{LCx} - \mathbf{Ly_d} \\ \mathbf{y} &= \mathbf{Cx}\end{aligned} \tag{7.27}$$

where $\mathbf{y_d}$ is the desired output. If we assume $\tilde{\mathbf{x}} = \mathbf{x} - \hat{\mathbf{x}}$, the above equations can be rewritten in matrix form as:

$$\begin{bmatrix} \dot{\mathbf{x}} \\ \dot{\tilde{\mathbf{x}}} \end{bmatrix} = \begin{bmatrix} \mathbf{A} - \mathbf{bK} & \mathbf{bK} \\ 0 & \mathbf{A} - \mathbf{LC} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \tilde{\mathbf{x}} \end{bmatrix} + \begin{bmatrix} 0 \\ \mathbf{L} \end{bmatrix} \mathbf{y_d} \tag{7.28}$$

$$= \mathbf{A_{CL}} \begin{bmatrix} \mathbf{x} \\ \tilde{\mathbf{x}} \end{bmatrix} + \mathbf{b_{CL}} \mathbf{y_d} \tag{7.29}$$

In the above form, it is easy to see that the closed loop system is stable as long as the poles of $\mathbf{A_{CL}}$ are less than zero, or if the determinant of $(s\mathbf{I} - \mathbf{A_{CL}})$ equals zero. But,

$$\mid s\mathbf{I} - \mathbf{A_{CL}} \mid = \mid s\mathbf{I} - (\mathbf{A} - \mathbf{bK}) \mid\mid s\mathbf{I} - (\mathbf{A} - \mathbf{LC}) \mid \tag{7.30}$$

Therefore the closed loop poles of the system are the poles of $(\mathbf{A} - \mathbf{bK})$ and $(\mathbf{A} - \mathbf{LC})$. The system will be stable as long as these poles are in the left hand plane. This result implies that the control gain matrix, $\mathbf{K}$ and the state estimator gain matrix, $\mathbf{L}$ can be designed separately, and will not effect the stability of the overall closed loop system.

Theoretically, because the system is both controllable and observable, $\mathbf{K}$ and $\mathbf{L}$ can be designed to place the controller and observer poles anywhere in the left hand plane. In a real system like TableSat, however, unlimited **K** and **L** matrices are not practical. The TableSat actuators, for example, will saturate above a certain voltage. To take that into consideration, consider the following cost function that weighs the speed of TableSat's response to the amount of control required:

$$J(v) = \frac{1}{2}\int_0^{\text{inf}} (\mathbf{x}^T(t)\mathbf{Q}\mathbf{x}(t) + rv(t)^2)\,\mathrm{d}t \tag{7.31}$$

where **Q** is a symmetric, positive definite matrix, and r is a positive scalar. Solving for the v which minimizes the above cost function yields $v^*(t) = -\mathbf{K}\mathbf{x}(\mathbf{t})$ with $\mathbf{K} = r^{-1}\mathbf{b}^T\mathbf{P_c}$, where P_c is the symmetric, positive semi-definite solution to the Algebraic Riccati Equation:

$$\mathbf{P_c}\mathbf{A} + \mathbf{A}^T\mathbf{P_c} = -\mathbf{Q} + \mathbf{P_c}\mathbf{b}r^{-1}\mathbf{b}^T\mathbf{P_c} \tag{7.32}$$

Selection of r and **Q** controls the trade off between speed of response and required control authority. One method of selecting r and **Q** takes into consideration the maximum allowable deviation in the output and the saturation level of the actuator:

$$\begin{aligned} \mathbf{Q} &= \mathbf{C}^T \begin{bmatrix} \frac{1}{\theta_{max}} & 0 \\ 0 & \frac{1}{\omega_{max}} \end{bmatrix} \mathbf{C} \\ r &= \rho\frac{1}{v_{max}} \end{aligned} \tag{7.33}$$

where θ_{max} and ω_{max} are the maximum allowable deviations in TableSat's angular position and velocity, respectively, v_{max} is the actuator saturation level, and ρ is a single scalar parameter that controls the tradeoff between speed and control. For TableSat, θ_{max} is set at 5 degrees, ω_{max} is set at 2 deg/sec, v_{max} is 12 volts, and ρ is 0.01. MATLAB can then be used to solve for $\mathbf{K}$ using the MATLAB command *care*. The above controller definition is called the Linear Quadratic Regulator.

The observer gain matrix, $\mathbf{L}$ can be found in a manner similar to the controller gain matrix. As in the Kalman Filter design, let $\mathbf{L} = \mathbf{P_o C}^T \mathbf{W_2}^{-1}$, where $\mathbf{P_o}$ is the steady state covariance of the observer estimates and is the symmetric, positive definite solution to the Algebraic Riccati Equation:

$$\mathbf{AP_o} + \mathbf{P_o A}^T = -\mathbf{\Gamma W_1 \Gamma}^T - \mathbf{P_o C}^T \mathbf{W_2}^{-1} \mathbf{CP_o} \tag{7.34}$$

where, as before, $\mathbf{W_1}$ represents uncertainties in the plant model that cannot be measured, $\mathbf{\Gamma}$ reflects how those disturbances enter the plat dynamics, and $\mathbf{W_2}$ represents the uncertainties in the system outputs, which is assumed to be the expected noise on the measurements. For TableSat, assume $\mathbf{\Gamma}$ is the 3x3 identity matrix and $\mathbf{W_1}$ and $\mathbf{W_2}$ are defined as follows:

$$\begin{aligned} \mathbf{W_1} &= \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 0.5 \end{bmatrix} \\ \mathbf{W_2} &= \begin{bmatrix} 2.2^2\,\mathrm{deg}^2 & 0 \\ 0 & 0.09^2\,\mathrm{deg/sec}^2 \end{bmatrix} \end{aligned} \tag{7.35}$$

With the above matrix definitions and the MATLAB command, *care*, $\mathbf{P_o}$ and thus $\mathbf{L}$ can be found. With $\mathbf{W_1}$ and $\mathbf{W_2}$ chosen as they are, the above optimal state estimator is the full Kalman Filter.

Once $\mathbf{K}$ and $\mathbf{L}$ have been found, the combined controller/state estimator needs to be defined in terms of the TableSat controller matrices. Using Equation 7.27 and the fact that $v = -\mathbf{K}\hat{\mathbf{x}}$, the following TableSat controller matrices can be found:

$$\begin{array}{ll} n_c = 3 & \\ \mathbf{A_c} = \mathbf{A} - \mathbf{bK} - \mathbf{LC} & \mathbf{C_c} = -\mathbf{K} \\ \mathbf{B1_c} = [\ \mathbf{0}_{3x1} \quad \mathbf{LC} \quad \mathbf{0}_{3x1}\ ] & \mathbf{D1_c} = [0_{1x5}] \\ \mathbf{B2_c} = -\mathbf{L} & \mathbf{D2_c} = [0_{1x2}] \end{array} \tag{7.36}$$

where the extra zeros are needed in the $\mathbf{B1_c}$ matrix to give it correct dimension. For the TableSat system matrices and chosen $\mathbf{Q}$, r, $\mathbf{W_1}$, W_2, and $\mathbf{\Gamma}$ matrices, the predicted closed loop poles of the system are -0.32, -17.3±16.3.

As with the kinematic Kalman Filter designed in Section 7.2.3, the above MBCO has been designed in the continuous time. Before being implemented on TableSat, it will have to be converted to discrete time using *c2d*. It should be noted that *c2d* expects only four state space matrices, while the TableSat controller has six. To account for this, the $\mathbf{B1_c}$ and $\mathbf{B2_c}$ can be combined into one matrix, $\mathbf{B_c} = \left[\begin{array}{cc} \mathbf{B2_c} & \mathbf{B1_c} \end{array} \right]$. Likewise the $\mathbf{D1_c}$ and $\mathbf{D2_c}$ can be combined into one matrix, $\mathbf{D_c} = \left[\begin{array}{cc} \mathbf{D2_c} & \mathbf{D1_c} \end{array} \right]$ After the system has been discretized using *c2d*, the the resulting matrices can be separated into their individual matrices. After discretization, the resulting controller definition is:

$$\begin{aligned}
&n_c = 3 \\
&\mathbf{A_c} = \begin{bmatrix} 0.9935 & -4.98\text{e}^{-4} & -3.96\text{e}^{-9} \\ -0.0241 & 0.746 & 1.22\text{e}^{-5} \\ -2736 & -4670 & 0.5952 \end{bmatrix} \\
&\mathbf{B1_c} = \begin{bmatrix} 0 & 0.0065 & 0.0204 & 0 & 0 \\ 0 & 0 & 0.2135 & 0 & 0 \\ 0 & -9.707 & 48.48 & 0 & 0 \end{bmatrix} \\
&\mathbf{B2_c} = \begin{bmatrix} -0.0065 & -0.0204 \\ 0 & -0.2135 \\ 9.71 & -48.48 \end{bmatrix} \\
&\mathbf{C_c} = \begin{bmatrix} -53.67 & -89.75 & -0.0076 \end{bmatrix} \\
&\mathbf{D1_c} = [0_{1\text{x}5}] \\
&\mathbf{D2_c} = [0_{1\text{x}2}]
\end{aligned} \tag{7.37}$$

7.3.2 Controller Results

The MBCO was tested on the linear model, truth model, and real TableSat system with a simple pass through used as the state estimator. Figure 7.13 shows the predicted output from the linear model for a step input of 50 degrees. As can be seen, the linear model predicts an asymptotic approach to the target with a relatively quick response. Steady state is reached within about 25 seconds. There is no steady state error in this prediction.

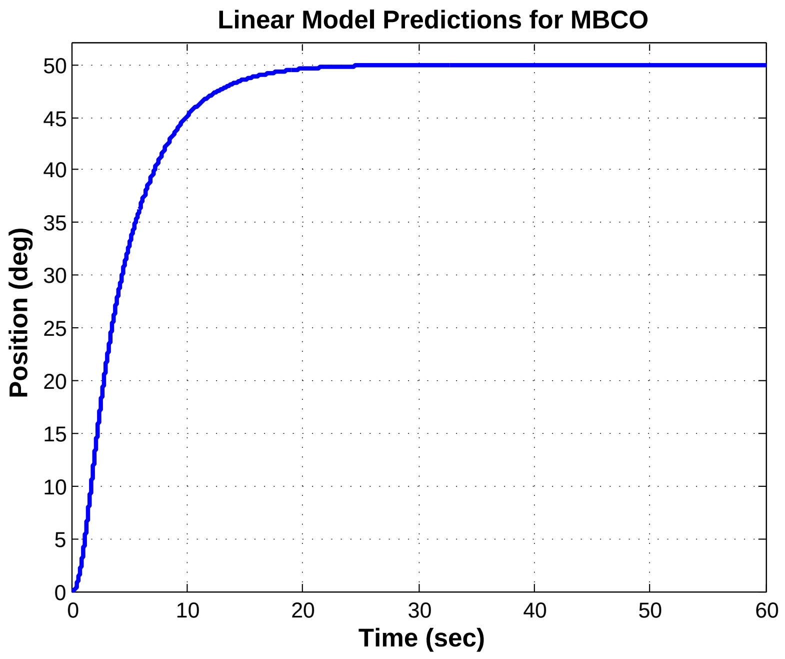


Figure 7.13: Predicted results from the linear model for a step input of 50 degrees using the MBCO.

This linear prediction can then be compared to the truth model and real TableSat results. Figure 7.14 shows the predicted and actual response of TableSat for a desired target of 50 degrees. As can be seen in this figure, both the truth model and real TableSat responses have the same shape as the linear model, but, as with the PD controllers, their response times are faster. The responses are overdamped and approach steady state asymptotically. TableSat reaches its desired target within about 10 seconds with about a 2 degree steady state error.

One thing that should be emphasized when looking at Figure 7.14 is that although the state estimate plot looks noisy, the controller is not actually calculating its control voltage based on these noisy measurements. Remember that the

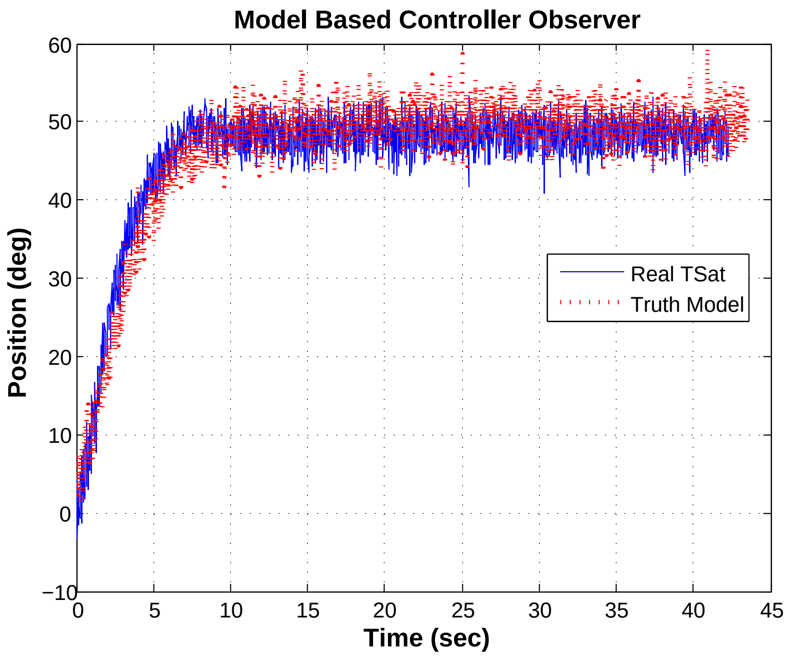


Figure 7.14: Predicted results from the truth model plotted with actual TableSat results using the MBCO with a desired target of 50 degrees.

Kalman Filter is incorporated as a part of the controller itself, and the controller states themselves are what the controller part of the MBCO uses to calculate the control voltage. The controller states for the MBCO should look very similar to the kinematic Kalman Filter estimates, which explains the steady state error seen in the plot. Recall that we also saw a steady state error with the kinematic Kalman Filter, which was likely caused by imperfect friction compensation. The smoothed controller states are not outputted from the controller thread; the data in the plot comes from the state estimator thread, which is still just a pass through and thus has no noise smoothing.

7.4 MBCO with Integral Augmentation

The results of the MBCO show a steady state error in the desired pointing angle. One possible cause of this steady state error is the imperfect friction compensation in the TableSat system. According to the truth model, if the friction compensation is adequate, the MBCO should be able to point TableSat in the desired direction with little or no steady state error. However, as was observed previously, we have not exactly modelled or compensated for all of the friction in the system, especially friction at low speeds and low voltage commands. In addition, the MBCO is essentially a Proportional-Derivative controller, meaning that there is no guarantee that it can remove all steady state error for a step input. To ensure zero steady state error for a step input, integral augmentation can be added to the MBCO.

7.4.1 Design Methodology

From single-input, single-output linear controller design theory, we know that adding an integrator to the forward path of a system will allow the closed loop system to perfectly track any step input. When extended to multi-input, multi-output systems, this implies that there must be an integrator in each input (or output) channel. Consider the augmented TableSat plant dynamics:

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{b}u \\ \dot{u} &= v \\ \mathbf{y} &= \mathbf{C}\mathbf{x}\end{aligned} \tag{7.38}$$

which can be written in state space form as follows:

$$\begin{aligned}\begin{bmatrix} \dot{\mathbf{x}} \\ \dot{u} \end{bmatrix} &= \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ u \end{bmatrix} + \begin{bmatrix} 0 \\ I \end{bmatrix} = \mathbf{A_a}\mathbf{x_a} + \mathbf{b_a}v \\ \mathbf{y} &= \begin{bmatrix} \mathbf{C} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ u \end{bmatrix} = \mathbf{C_a}\mathbf{x_a}\end{aligned} \tag{7.39}$$

As with the MBCO, the control law $v = -\mathbf{K_a}\mathbf{x_a}$ is assumed, but because TableSat does not have full state feedback, a state estimator is needed to estimate the missing state. The same state estimator used in the MBCO, the Kalman Filter, can be used with the augmented plant, yielding the following closed loop dynamics:

$$\begin{aligned}\dot{\hat{\mathbf{x}}}_{\mathbf{a}} &= (\mathbf{A_a} - \mathbf{b_a}\mathbf{K_a} - \mathbf{L_a}\mathbf{C_a})\hat{\mathbf{x}} + \mathbf{L_a}\mathbf{C_a}\mathbf{x_a} - \mathbf{L_a}\mathbf{y_d} \\ \mathbf{y} &= \mathbf{C_a}\mathbf{x_a} \\ v &= -\mathbf{K_a}\hat{\mathbf{x}}_{\mathbf{a}}\end{aligned} \tag{7.40}$$

where $\mathbf{K_a}$ and $\mathbf{L_a}$ are the controller and estimator gain matrices, respectively, for the augmented plant. $\mathbf{K_a}$ and $\mathbf{L_a}$ can be found in the same manner as $\mathbf{K}$ and $\mathbf{L}$ in Section 7.3. For finding $\mathbf{K_a}$, r and $\mathbf{Q}$ are defined as in Equation 7.33, with $\mathbf{C_a}$ replacing $\mathbf{C}$. For finding $\mathbf{L_a}$, $\mathbf{W_2}$ is defined as in Equation 7.35 and $\mathbf{W_1}$ is modified to include the additional plant state, u:

$$\mathbf{W_1} = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 1.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0.5 \end{bmatrix} \tag{7.41}$$

After finding $\mathbf{K_a}$ and $\mathbf{L_a}$, the MBCO with integral augmentation can be translated into TableSat form:

$$\begin{array}{ll} n_c = 4 & \\ \mathbf{A_c} = \mathbf{A_a} - \mathbf{b_a K_a} - \mathbf{L_a C_a} & \mathbf{C_c} = -\mathbf{K_a} \\ \mathbf{B1_c} = [\ \mathbf{0}_{4x1} \quad \mathbf{L_a C_a} \quad \mathbf{0}_{4x1}\ ] & \mathbf{D1_c} = [0_{1\text{x}5}] \\ \mathbf{B2_c} = -\mathbf{L_a} & \mathbf{D2_c} = [0_{1\text{x}2}] \end{array} \tag{7.42}$$

where, as before, the extra zeros in $\mathbf{B1_c}$ are needed to give the matrix the correct dimension. As with the MBCO, the above controller is defined in continuous time and will need to be converted to discrete time before implementing it on TableSat. The predicted closed-loop poles for the MBCO with integral augmentation (MBCOi) are -0.32, -5.2±10.6, and -15.0. After discretization, the MBCOi controller definition is:

$$
\begin{aligned}
&n_c = 4 \\
&\mathbf{A_c} = \begin{bmatrix} 0.9935 & -1.06\text{e}^{-4} & -9.85\text{e}^{-10} & -2.02\text{e}^{-8} \\ -2.16\text{e}^{-4} & 0.7752 & 1.57\text{e}^{-5} & -4.79\text{e}^{-4} \\ -31.69 & -1060 & 1.0096 & 56.42 \\ -0.9143 & -1.303 & -6.43\text{e}^{-4} & 0.7171 \end{bmatrix} \\
&\mathbf{B1_c} = \begin{bmatrix} 0 & 0.0065 & 0.020 & 0 & 0 \\ 0 & 0 & 0.2244 & 0 & 0 \\ 0 & -0.0739 & 999.5 & 0 & 0 \\ 0 & -0.0032 & -0.454 & 0 & 0 \end{bmatrix} \\
&\mathbf{B2_c} = \begin{bmatrix} -0.0065 & -0.020 \\ 0 & -0.2244 \\ 0.0739 & -999.5 \\ 0.0032 & 0.454 \end{bmatrix} \\
&\mathbf{C_c} = \begin{bmatrix} -53.67 & -102.2 & -0.036 & -15.28 \end{bmatrix} \\
&\mathbf{D1_c} = [0_{1\text{x}5}] \\
&\mathbf{D2_c} = [0_{1\text{x}2}]
\end{aligned}
\tag{7.43}
$$

7.4.2 Controller Results

Like the MBCO, the MBCO with integral augmentation (MBCOi) was tested on the linear model, truth model, and real TableSat system with a simple pass through state estimator. Figure 7.15 shows the predicted output from the linear model with a step input of 50 degrees. As can be seen, the linear model predicts an overdamped response, with a slightly longer settling time that the MBCO. The MBCOi reaches its steady state in about 25 seconds. But, as with the MBCO, the MBCOi predicts no steady state error.

The linear model prediction can now be compared to the predicted results

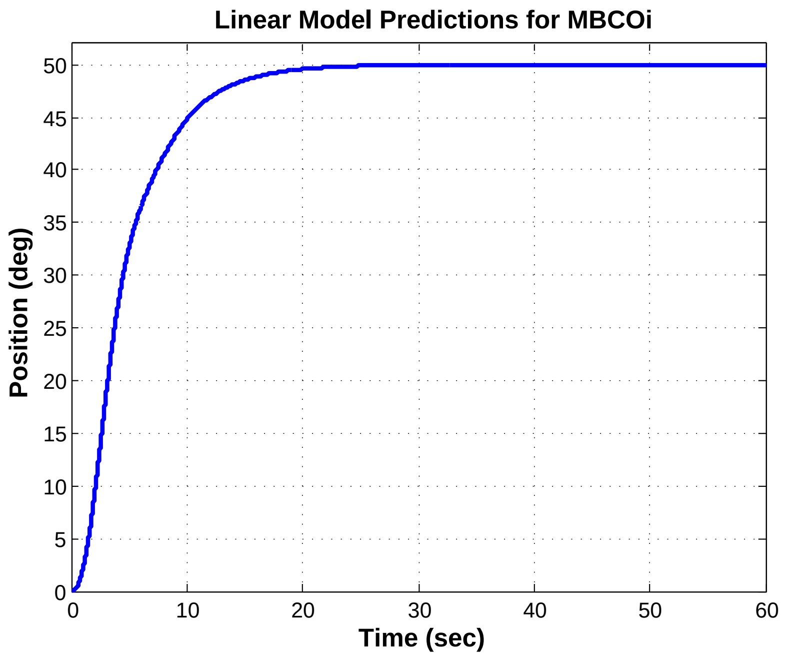


Figure 7.15: Predicted results from the linear model for a step input of 50 degrees using the MBCO with integral augmentation.

from the truth model and real TableSat system. Figure 7.16 shows the predicted and actual TableSat response for a desired target of 50 degrees. As can be seen in the figure, the truth model prediction and real TableSat results are similar to the linear prediction, but like the MBCO, the truth model and real TableSat system respond quicker, reaching steady state in about 10 seconds. From the figure, it can be seen that the truth model predicts no steady state error, which is an improvement over the MBCO, which showed a steady state error of about 2.0 degrees. This improvement in steady state pointing implies that the integral augmentation did remove the steady state error in the truth model as desired. On the other hand, the real TableSat results still show a steady state error of about 2.0 degrees, which implies the integral augmentation did not act exactly as hoped.

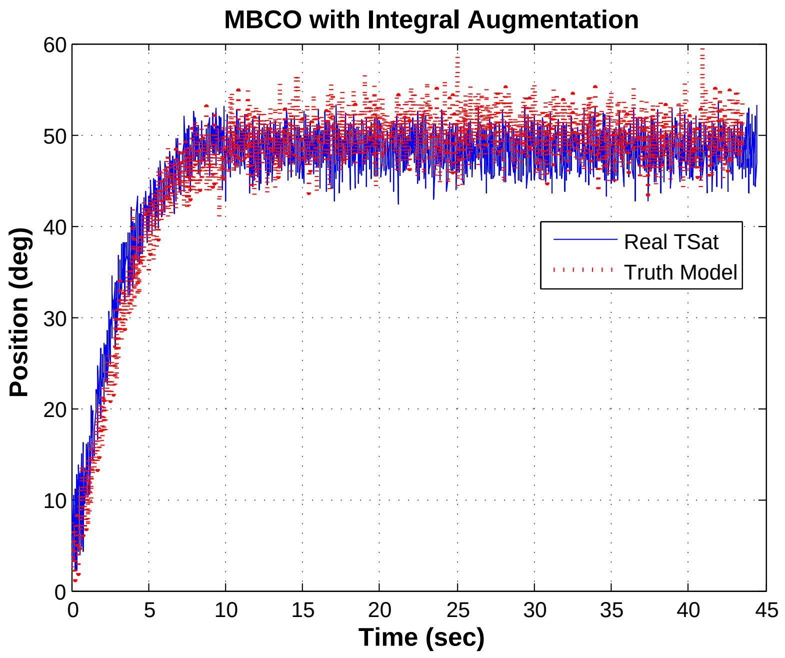


Figure 7.16: Predicted results from the truth model plotted with actual TableSat results using the MBCOi with a desired target of 50 degrees.

Chapter 8

Conclusions

The main goal throughout the TableSat project has been to develop and model a system that can be used to demonstrate the often abstract concepts of control systems engineering and show how those concepts can be applied to a real system. More specifically, TableSat represents a complete, single degree of freedom spacecraft that uses sensors and actuators to estimate and control the attitude and rate of the system. While the focus has been on using TableSat to demonstrate SISO linear controls techniques, those taught to undergraduate engineering students, it was also desirable to show how TableSat could also be used to demonstrate more complicated multivariable and state space controls techniques. Originally, TableSat started as a highly nonlinear system. To effectively function as a teaching tool for linear controls theory, however, TableSat needs to act as a linear system; and it must be possible to accurately model the system. Through hardware and software upgrades it was possible to reduce some of the nonlinearities in the TableSat system. Once the hardware and initial software upgrades were complete, system modelling was used to identify and sometimes quantify additional nonlinearities in the system. Once identified, in many cases it was possible to eliminate or reduce these nonlinearities, allowing for the development of a linear system model and a truth model. Both models were then verified and used to develop example controllers and state estimators for the new TableSat system. Finally, these controllers and estimators were tested on the real TableSat system and the results compared to predictions based on the TableSat models.

8.1 TableSat Hardware Upgrades

Recall from Chapter 1 that the original TableSat flight processor was the PIC16F874 8-bit micro controller programmed in low-level assembly language. The most important upgrade to the TableSat hardware was replacing the PIC with the Prometheus embedded PC/104 CPU. The Prometheus, which is discussed in detail in Chapter 2, has a faster processor and more memory than the PIC. These features allow the flight code to be stored and run onboard TableSat instead of calculating the state estimate and control from the laptop with sensor measurements and actuator commands sent back and forth via the RF serial link. In addition, the Prometheus allows the actuator fans to be issued direct voltage commands instead of using PWM actuation. The faster processor speed and removing the laptop from the loop greatly increased the speed of the TableSat control cycle, making it act more like an ideal, continuous system. In addition, with the new processor, the flight code could be written in C code. Because C is a higher level language than assembly language, the code now has more functionality, and can be used to compensate for many of the nonlinearities found in the TableSat system.

The new flight processor did come with some drawbacks, however. Because it is a more powerful computer, it also requires more power. Due to the current draw of the PC/104, the original TableSat battery pack could only supply power for approximately 30 minutes. It was therefore replaced with the Polarmate external laptop battery. The new battery, while much heavier than the original battery pack, is able to run TableSat for at least ten hours before needing to be recharged. However, because of the increased battery mass, TableSat's moment of inertia also increased. To compensate for this increased inertia, the original TableSat actuator fans were replaced with more powerful computer fans. Both the new battery and new fans are discussed in Chapter 2.

Originally, the communications between TableSat and the user were accomplished via an RF serial link. One of the other upgrades was to replace the RF serial link with an OTC wireless access point routed through a Linksys router. The wireless access point has a top speed of 11 Mbs, much faster than the RF serial link, which increases the speed at which the user can communicate with TableSat. Because all of the TableSat control can now be performed onboard, this upgrade does not increase the speed of the control cycle in general, but it does allow the user to see how the TableSat state changes in real time. If the user chooses to command TableSat in “open-loop mode,” *i. e.* using Simulink in the loop, then this new communications system does increase the TableSat control cycle. The new TableSat communications hardware is also discussed in Chapter 2.

8.2 TableSat Software Upgrades

As was mentioned before, the new flight processor allowed for the TableSat flight code to be written in C code. While this required all of the flight code to be rewritten from scratch, it allowed for much more functionality in the new code. The flight code, which is discussed in detail in Chapter 3 is split into five main threads which run simultaneously on the processor. The main TableSat thread initializes the TableSat hardware and the other four threads. It then runs in the background, essentially doing nothing, until stopped. The state estimator thread reads the TableSat sensors, converts them to meaningful engineering units, and uses the measurements, as well as a user established state estimator, to estimate the state of the system. It then makes that state estimate available to the controller thread. The controller thread uses the state estimate, along with a user developed controller, to calculate the control authority required to move TableSat such that its state meets a desired value. It then makes the required control authority available to the actuator thread. The actuator thread converts the desired control authority into the form required by the actuators and sends those commands to the actuators. In the background runs the communications thread. The communications thread conveys information to the user and receives commands from the user and parses them to the correct thread. It allows the user to start and stop TableSat, change the controller, and change the state estimator, for example.

As was mentioned earlier, the new flight processor can send analog commands directly to the actuators. The new software takes advantage of this fact by commanding the fans in a continuous fashion, which means the actuators get the exact voltage they request within about ± 3 mV, or the resolution of the D/A conversion with voltage regulation. Recall from Chapter 1 that the original TableSat software used pulse width modulation (PWM), a highly nonlinear process. When using PWM, the actuators are commanded at their full voltage for a fraction of a control cycle. The fraction depends on the amount of control desired and the speed of the actuator cycle. For the original TableSat software, there were four actuator cycles per control cycle, which resulted in an actuator resolution of about 3 V. By allowing continuous actuation, the nonlinearity due to using PWM can be eliminated. One of the other goals of the TableSat project was to retain the ability to add nonlinearities back into the system if desired. To that end, the new TableSat software also allows for PWM or Bang-Bang actuation, if desired.

The state estimator, controller, and actuator threads can be executed at any speed up to about 100 Hz, which is the limit of the processor speed. The user can set the cycle speeds, and, the faster the cycles are run, the more TableSat acts as an ideal, continuous system. With its original hardware/software compliment, TableSat's maximum execution speed was on the order of 2.5 Hz, too slow to be considered a continuous system.

In addition to the continuous actuation and cycle speed variability, there exists within some of the software threads additional functions that can be used to compensate for some of the other nonlinearities in the system. These functions are discussed in detail in Chapter 4. Most notable is the friction compensation function, which accepts as input from the user a desired friction compensation curve. Expressed as voltage per speed, the curve describes the amount of additional voltage to be applied to the calculated controller voltage to compensate for friction in the TableSat system. If the friction compensation curve is correctly identified, this functionality effectively eliminates the nonlinearity due to TableSat's internal friction. If the user wishes to put friction back into the system, the friction compensation can be disabled.

Along with the function to compensate for TableSat friction, the flight code also includes a function that eliminates the dead zone in the actuators. Each actuator has internal friction, which reduces the effective voltage and thus the effective control authority of the actuator. Because of this internal friction, there is a voltage below which the actuators will not spin. The actuator dead zone compensation reduces this dead zone from ± 2.75 V to ± 0.1 V. It does not, however compensate for the internal actuator friction in any other way. So, the nonlinearity is reduced, but not completely eliminated.

Accompanying the new software is a new user interface and TableSat Simulink models with which the user can interact with TableSat. The TableSat GUI and Simulink models are discussed in detail in Chapter 4. The GUI can be used to load different controllers and state estimators, enable/disable friction compensation, load new friction compensation curves, and change the cycle speeds, along with other things. In this way, the user can have TableSat act as a linear system, or introduce the nonlinearities back into the system to see how the nonlinearities effect the system response. The TableSat Simulink models allow the user to start and stop control of TableSat and plot the TableSat state real time. In addition, the open-loop Simulink model allows the user to design and implement controllers directly from Simulink, which can allow for more advanced controller and/or estimator design than that which can be implemented using the onboard controller and estimator definitions. Being able to design and implement advanced controllers and state estimators was one of the goals of this project.

8.3 TableSat System Identification

As was mentioned previously, one of the main goals of the TableSat project was to develop a model of the TableSat system that can be used to develop and test controllers and state estimators for the system. Once the hardware and software upgrades were complete, a series of tests and experiments were performed to develop the TableSat model. Chapter 5 discusses this model development process in detail. As a starting point, a system of differential equations was assumed as the TableSat equations of motion. These equations model the fan and TableSat dynamics and include nonlinear friction terms for both the fans and TableSat itself.

Including the friction values, there were a total of nine system parameters that needed to be determined to establish the TableSat system model. A combination of experimental tests and theoretical equations were used to determine initial values for each of the parameters and to identify and quantify the nonlinearities in the system. For example, a torsional pendulum test was used to determine the TableSat moment of inertia, spin up-spin down tests were done to characterize the TableSat friction, and fluid dynamics equations along with the fan specifications were used to estimate the fan speed to voltage constant.

There were two main types of nonlinearities found during these tests. The first was the friction in TableSat itself. From the spin up-spin down tests it was determined the the friction in TableSat, because it spins about a single point, is constant. It can therefore be compensated for using a simple friction compensation curve that adds additional voltage depending on the direction TableSat is spinning. This method of friction compensation can easily be employed using the friction compensation function discussed above.

The second main nonlinearity in the TableSat system is the friction of the TableSat fans. Through experimentation, it was determined that the fan friction consists of a constant term and a linear term. Compensation for the fan static friction, or dead zone, was mentioned above and is discussed in detail in Chapter 5. There is currently no means to compensate for the linear fan friction. That friction is modelled as a part of the fan time constant. Once the fans have finished accelerating, that term is effectively gone; however, at low voltages there may be additional effects that have not be exactly determined.

Once initial values for the system parameters were determined, a TableSat truth model was developed. The truth model integrates the assumed equations of motion using the parameter values determined through analysis and testing and includes representative sensor noise. After creation of the truth model, it was then used to tune the TableSat parameters by comparing real TableSat data to truth model data and "tweaking" the parameters until the two sets of data matched reasonably well. In this manner, the TableSat truth model was determined to be a reasonable approximation of the real TableSat system.

Once the truth model was developed and the friction nonlinearities identified, methods for compensating for those nonlinearities were developed and implemented. With friction out of the system, the equations of motion reduced to linear equations of motion that could be solved to obtain a linear model of TableSat, which was also one of the goals of the project. The linear model was verified by comparing both closed-loop and open-loop data from the linear model, truth model, and real TableSat system. The data showed that, when controlling TableSat in closed-loop mode, the linear model does a good job of modelling the system as long as you stay within the saturation level of the actuators. The open-loop data showed that there are still some nonlinearities that are not compensated for in the system. The exact nature of these nonlinearities and how to compensate for them still needs to be investigated. Despite these nonlinearities, it is felt that the linear TableSat model is reasonable and can be used to design controllers and state estimators for TableSat.

8.4 Case Studies

With verification of the TableSat system model accomplished, the model was then used to develop and test controllers and state estimators for TableSat. Chapter 7 discusses the development of three different controllers, a simple N-sample averaging state estimator, and a simplified kinematic Kalman Filter. The three controllers are a PD controller, a Model Based Controller/Observer, and a Model Based Controller/Observer with Integral Augmentation. In all three cases, linear controls methods were employed to develop the controllers. The PD controller was developed using SISO, Root Locus techniques. The other two controllers were developed using MIMO, state space techniques. All three controllers were tested on the linear and truth models as well as the real TableSat system. The results were discussed and explanations for any discrepancies were attempted. The most notable difference between the linear model predictions and truth model and real TableSat responses for all three controllers is that the truth model and real TableSat consistently respond faster than the linear model predicts. Further investigation is needed to determine the cause of this discrepancy.

By combining a simple N-sample averaging state estimator to the PD controller, steady state pointing performance could be improved. In addition, noise in the pointing angle was decreased. The degree of improvement in both steady state error and noise depends on the number of samples in the estimator. A 2-sample estimator reduced the steady state error by more than a factor of two and reduced the noise by $\sqrt{2}$. A 5-sample estimator eliminated the steady state error and reduced the noise by $\sqrt{5}$.

By combining a kinematic Kalman Filter to the PD controller, noise in the state estimates was essentially eliminated. In addition, the smooth estimates revealed interesting oscillations in the transient response of TableSat. Further analysis is needed to determine the cause of these oscillations, but they were also seen in the linear model prediction. The Kalman Filter also showed that there are still some nonlinearities in the system that have not be fully determined and compensated for. It appears these nonlinearities occur at low speeds and low voltages. Further investigation is needed to determine these nonlinearities and ways to compensated for them.

8.5 Final Conclusions and Future Work

All in all, the TableSat project is a success. With the new hardware and software upgrades that eliminate or reduce most of the nonlinearities in the TableSat system, TableSat can be assumed to be a fairly linear system. The linear model developed during the system identification process has been successfully used to develop and test controllers for the real TableSat system using linear design methods. The developed controllers have then been tested on the real TableSat system and performed more or less as expected. Where there were discrepancies in the predicted and actual steady state performance can be traced to the fact that TableSat was at a low speed, or the voltage commands were low. It is apparent that work still needs to be done to determine the exact cause of these discrepancies and find ways to compensate for them. As was mentioned previously, there were also discrepancies in the transient responses, the cause of which still needs to be determined. Of potential use in diagnosing these nonlinearities would be the tachometer feedback from the fans. We would like to incorporate the circuitry necessary to use this feature of the fans, which would also be useful in fine tuning the fan voltage to speed constant, which was the only parameter unable to be verified through testing.

Once these nonlinearities have been compensated for, it should be possible to create experimental Bode diagrams for the real TableSat system. One major test of whether we have an accurate linear model of TableSat, which we have been unable to accomplish thus far, would be to compare an experimental Bode diagram to the theoretical prediction from the linear model. If they match reasonably well, we can be sure we have successfully modelled and linearized TableSat.

It is felt at this point that TableSat can be a very effective teaching tool for linear controls systems. Students can use the TableSat linear model to develop controllers, test the controllers on the truth model, and see the controllers in action on the real TableSat. Then, after they see the ideal, linear TableSat response, the nonlinearities can be reintroduced into the TableSat system to show how a real system would react. Not only can TableSat help bridge the gap between theoretical and applied controls, it can also show the differences between an ideal system and a real system. Hopefully, these hands-on experiences will show students how much fun control systems engineering can be, and will convince them to pursue controls, especially Attitude Control Systems, as a field of study.

Appendix A

TableSat Source Code

A.1 Include Files

```
TS_ACTUATOR.H

#ifndef _TSACTUATORH_ #define _TSACTUATORH_

#include "../include/TS_Includes.h"

/* Max volts output for D/A on fans */

#define VMAXFAN 12.0

/* Different actuator operating modes */

#define CONTINUOUS 0 #define PWM        1 #define BANGBANG   2

void setFanVoltage(int pc, double *v);

#endif

TS_CONTROLLER.H

#ifndef _TSCONTROLLERH_ #define _TSCONTROLLERH_

#define HOLD 0 #define SINE 1

#include "../include/TS_Includes.h"

void desiredTrajectory(Controller_t *Controller, double *xd,
double t);
```

```
#endif

TS_ESTIMATOR.H

#ifndef _TSESTIMATORH_ #define _TSESTIMATORH_

#include "../include/TS_Includes.h"

#define Deg2Rad 0.017453293 #define Rad2Deg 57.29577951 #define
Cos80   0.173648178

void ConvertToEngineeringUnits(SensorReadings_t*,
SensorCalibration_t*,EngineeringUnits_t*);

#endif

TS_GLOBALS.H

#ifndef TS_GLOBALS_H #define TS_GLOBALS_H

#ifndef TS_TIMERVALS_H
    #define EXTERN1 extern
#else
    #define EXTERN1
#endif EXTERN1 int numCycles;

#ifndef TS_MAINTABLESAT_H
    #define EXTERN2 extern
#else
    #define EXTERN2
#endif EXTERN2 EngineeringUnits_t EngineeringUnits; EXTERN2 float
FanVoltage[2]; EXTERN2 void *controller_thread(); EXTERN2 void
*state_est_thread(); EXTERN2 void *actuator_thread(); EXTERN2 void
*communication_thread();

EXTERN2 pthread_rwlock_t SensorDataLock; EXTERN2 pthread_rwlock_t
StateEstimateLock; EXTERN2 pthread_rwlock_t EngineeringLock;

EXTERN2 pthread_rwlock_t ControlDataLock; EXTERN2 pthread_rwlock_t
FanDataLock; EXTERN2 pthread_rwlock_t ObserverDataLock; EXTERN2
pthread_rwlock_t ActuatorDataLock;
```

```
EXTERN2 pthread_rwlock_t StatusDataLock;

EXTERN2 struct timeval start_time;

EXTERN2 timer_t timeridC; EXTERN2 timer_t timeridSE; EXTERN2
timer_t timeridCOM; EXTERN2 timer_t timeridACT;

EXTERN2 int chan_idC; EXTERN2 int chan_idSE; EXTERN2 int
chan_idCOM; EXTERN2 int chan_idACT;

#ifndef TS_PROMINIT_H
    #define EXTERN3 extern
#else
    #define EXTERN3
#endif

#ifndef TS_TIMERSINIT_H
    #define EXTERN4 extern
#else
    #define EXTERN4
#endif EXTERN4 struct _pulse pulseSE; EXTERN4 struct _pulse
pulseC; EXTERN4 struct _pulse pulseCOM; EXTERN4 struct _pulse
pulseACT;

#ifndef TS_STATEESTIMATOR_H
    #define EXTERN5 extern
#else
    #define EXTERN5
#endif EXTERN5 SensorReadings_t SensorReadings; EXTERN5
StateEstimate_t StateEstimate; EXTERN5 double *zo_old, *zo_new;
EXTERN5 double *Ao, *Bo, *Co, *Do;

#ifndef TS_CONTROLLER_H
    #define EXTERN6 extern
#else
    #define EXTERN6
#endif EXTERN6

#define Deg2Rad 0.017453293 #define Rad2Deg 57.29577951

#endif
```

TS_INCLUDES.H

```
#ifndef _TSINCLUDESH_ #define _TSINCLUDESH_

#include <stdio.h> #include <time.h> #include <pthread.h> #include
<sys/time.h> #include <sys/timeb.h> #include <time.h> #include
<stdlib.h> #include <sys/siginfo.h> #include <sys/neutrino.h>
#include <math.h> #include <string.h> #include
"../include/TS_structs.h"

#endif
```

TS_INIT.H

```
#ifndef _TSINITH_ #define _TSINITH_

#include "../include/TS_Includes.h"

void InitCalibration(SensorCalibration_t *theCal); int
MakeTimer(timer_t *theTimer); void InitTimers(struct itimerspec
*timerCptr, struct itimerspec *timerSEptr,
        struct itimerspec *timerCOMptr, struct itimerspec *timerACTptr);
void setTimerVal(struct itimerspec *theTimer, double theRate);


#endif
```

TS_MAIN.H

```
#ifndef _TSMAINH_ #define _TSMAINH_

#include "TS_Includes.h" #include "TS_Actuator.h"

void *controller_thread(void *); void *state_est_thread(void *);
void *actuator_thread(void *); void *communication_thread(void *);

pthread_rwlock_t SensorLock, StateLock,EstimatorLock;
pthread_rwlock_t ControllerLock, FanLock, ActuatorLock;
pthread_rwlock_t StatusLock;

timer_t timeridC, timeridSE, timeridCOM, timeridACT; int chan_idC,
chan_idSE, chan_idCOM, chan_idACT; struct _pulse pulseSE, pulseC,
pulseCOM, pulseACT;

pthread_t  threadC, threadSE, threadCOM, threadACT;

struct timeval start_time;

unsigned char TSrunMode      = TRUE;  // Start with tsat running
unsigned char TScontrolMode  = OFF;   // Start with controller off
unsigned char TSactuatorMode = CONTINUOUS; unsigned char
TSgyroMode     = OFF;   // Auto gyro calibration off unsigned char
TSfanMode      = OFF;   // No compensation for fan dynamics

#endif

TS_MATH.H

#ifndef _TSMATHH_ #define _TSMATHH_

#include "../include/TS_Includes.h"

#define TINY 1e-6 #define ABS(X) ( (X) < 0 ? -(X) : (X) )

double interpolateData(double *xvals, double *yvals, int nvals,
double x); sparseMat *makeSparse(double *M, int nrows, int ncols);
void freeSparse(sparseMat *M); void sparseMatVec(sparseMat *Mat,
double *vecIn, double *vecOut, int lenOut); void matVec(double
*mat, double *vec, double *vecres,int nrow,int ncol); void
printSparse(sparseMat *Mat, const char *Mesg);

#endif

TS_MESSAGES.H

#ifndef _TSMESSAGESH_ #define _TSMESSAGESH

#define myID 0

//  Message identifiers

// Mode changes from console
```

```
#define MSG_ackMsg               2 #define MSG_setRunMode 4
#define MSG_setControlMode       6 #define MSG_setFanMode 8
#define MSG_setActuatorMode     10 #define MSG_setGyroMode 12

// Controller/observer updates from console

#define MSG_setControlData      15 #define MSG_setFanSpeedData 18
#define MSG_setFanFricData      21 #define MSG_setControlTarget 24
#define MSG_setEstimatorData    27 #define MSG_setGyroCalibData 30
#define MSG_setSampleRate       33

// Requests for vehicle data from console

#define MSG_uplinkStateData     60 #define MSG_uplinkRawData 63
#define MSG_uplinkStatusData    66

// Data uplink packets to console

#define MSG_sendStateData       81 #define MSG_sendRawData 84
#define MSG_sendStatusData      87

// Request to calibrate sensors

#define MSG_calibrateSensors    99


// Prototypes for associated message handlers
typedef int HandlerFunc(byte,byte *);

HandlerFunc setRunMode,setControlMode,setFanMode, setGyroMode,
setActuatorMode; HandlerFunc
setControlData,setFanSpeedData,setFanFricData; HandlerFunc
setControlTarget,setSampleRate; HandlerFunc
setEstimatorData,setGyroCalibData; HandlerFunc sendStateData,
sendRawData, sendStatusData; HandlerFunc calibrateSensors;

// Default TableSat controller console ID

#define SAT_CON 0

#endif
```

TS_PACKET.H

```
#ifndef _TSPACKETH_ #define _TSPACKETH_

#include <sys/types.h> #include <sys/socket.h> #include
<netinet/in.h> #include <arpa/inet.h> #include <stdio.h> #include
<stdlib.h> #include <string.h> #include <errno.h> #include
<sys/time.h> #include <signal.h> #include <unistd.h> #include
<fcntl.h>

typedef unsigned char byte; typedef int
(*HandlerFuncPtr)(byte,byte *);

#define MAXCLI 16 #define MAXLINE 65535 #define dsmpSERV_PORT 9877
#define SA struct sockaddr

int dsmp_Init(unsigned char ident, int portNum); void
dsmp_RegisterClient(byte cliNum, char *cliIP); void
dsmp_RegisterMesg(unsigned char msgNum, unsigned char msgFlags,
               HandlerFuncPtr msgHandler, short msgSize);
int dsmp_RecvMesg(void); int dsmp_RecvALL(void); int
dsmp_SendMesg(byte mesgNum, void *mesgData, byte mesgClient); int
dsmp_SendRaw(byte mesgNum, void *mesgData, unsigned short
mesgSize, byte mesgFlags, byte mesgClient); void
dsmp_AcceptMcast(char *addrStr);

#define TRUE 1 #define FALSE 0 #define TS_VARSIZE 32767

#endif
```

TS_PROMIO.H

```
#ifndef _TSPROMIOH_ #define _TSPROMIOH_

#include "../include/TS_Includes.h" #undef TRUE #undef FALSE
#undef OFF #undef ON #include <dscud.h>

#define BASEADD 0x280 #define ERROR_PREFIX "Prom Driver ERROR:"

void InitPrometheus(void); int ReadSensors(SensorReadings_t
*sensors); void commandMotor(double *v);
```

```
#endif

TS_STRUCTS.H

#ifndef _TSSTRUCTH_ #define _TSSTRUCTH_

#define NY    5 #define NX    5 #define NU    2

#define TRUE  1 #define FALSE 0

#define OFF   0 #define ON    1

typedef struct {
  int col;
  double val;
} cvals;

typedef struct {
  int nrows;
  int *nvals;
  cvals **colVals;
} sparseMat;

typedef struct {
  double TimeStamp;
  double CSS1Volts;
  double CSS2Volts;
  double CSS3Volts;
  double CSS4Volts;
  double GyroVolts;
  double MagXVolts;
  double MagYVolts;
  double MagZVolts;
  double Fan1Counts;
  double Fan2Counts;
} SensorReadings_t;

typedef struct {
  double TimeStamp;
  double NormToMaxCSS1Volts;
  double NormToMaxCSS2Volts;
  double NormToMaxCSS3Volts;
  double NormToMaxCSS4Volts;
  double AzCss;
  double CSSUnitVecX;
  double CSSUnitVecY;
  double CSSUnitVecZ;
  double GyroValue;
  double BhatX;
  double BhatY;
  double BhatZ;
  double AzTam;
  double Fan1Speed;
  double Fan2Speed;
} EngineeringUnits_t;

typedef struct {
  double TimeStamp;
  double AzCss;
  double AzTam;
  double omega;
  double nu1;
  double nu2;
} StateEstimate_t;

typedef struct {
  double TamBias[3];
  double TamGain[3];
  double CssMax[4];
  double BEarthRef[3];
  double Volts2Dps;
  double GyroOffset;
  double CssMaxMean;
  double Azm;
} SensorCalibration_t;

typedef struct {
  int nc, pc;
  double *z, *zold;
  sparseMat *A, *B1, *B2, *C, *D1, *D2;
  double xd[2];
  double t0;
  double comtorq[2];
  unsigned char TrackMode;
```

```
} Controller_t;

typedef struct {
  int no;
  double *z, *zold;
  sparseMat *A, *B, *C, *D;
  double xhat[NY];
  double adaptGyroGains[2];
  double th, thold;
  double ahat, bhat;
} Estimator_t;

typedef struct {
  double ref_voltage;
  double Freq, FreqRatio;
  double FanVoltage[2];
  double Deadzone;
  int    counts;
} Actuator_t;

typedef struct {
  double *wfric, *wfan;
  double *fanFric, *fanForce;
  int    nFricPts, nFanPts;
} FanData_t;

#endif

TS_TIMERSINIT.H

#ifndef TS_TIMERSINIT_H #define TS_TIMERSINIT_H

#include "TS_Includes.h"

#endif

TS_UNITS.H

#ifndef TS_CONVERTTOENGUNITS_H #define TS_CONVERTTOENGUNITS_H

#include </home/vess/project/TS_Includes.h>
```

```
void SaveCssData(float, double, double, double, double); void
SaveGyroData(float, double); void SaveMagData(float, double,
double, double, double); void SaveCssVolts(float, double, double,
double, double); void SaveGyroVolts(float, double); void
SaveMagVolts(float, double, double, double);

#endif
```

A.2 Source Code

```
#include "../include/TS_Actuator.h" #include <math.h>

/* Main Actuator structures */

extern Actuator_t Actuator;

/* Interprocess signalling and synchronization */

extern struct _pulse pulseACT; extern int chan_idACT;

extern pthread_rwlock_t ActuatorLock, StatusLock; extern unsigned
char TSrunMode, TSactuatorMode;

/* Begin actuator thread */

void *actuator_thread(void *TSfoo) {

  unsigned char TSrunning=TRUE, actMode=CONTINUOUS;
  static double fullPos[2]={VMAXFAN,0.0};
  static double fullNeg[2]={0.0,VMAXFAN};
  static double fanOff[2] = {0.0,0.0};
  static int numPos=0, numNeg=0;

  printf("Starting actuator thread\n"); fflush(stdout);

  while (TSrunning) {

    /* Block until Controller thread pulse */
    MsgReceive(chan_idACT, &pulseACT, sizeof(pulseACT), NULL);
```

```
/* Grab actuator read lock */
pthread_rwlock_rdlock(&ActuatorLock);

/* Continuous Mode -- use voltage computed in control law */
if (actMode == CONTINUOUS) {
  // No need to do anything
  // commandMotor(Actuator.FanVoltage);
}
/* Bang-Bang mode -- full on pos or full on neg */
else if (actMode == BANGBANG) {
  if (Actuator.FanVoltage[0]>Actuator.Deadzone) {
Actuator.FanVoltage[0] = fullPos[0];
Actuator.FanVoltage[1] = fullPos[1];
  }
  else if (Actuator.FanVoltage[1]>Actuator.Deadzone) {
Actuator.FanVoltage[0] = fullNeg[0];
Actuator.FanVoltage[1] = fullNeg[1];
  }
}
/* PWM mode -- full pos/neg on for a percentage of time */
else if (actMode == PWM) {
  /* Compute % on for next cycle */
  if (Actuator.counts++>=Actuator.FreqRatio) {
numPos = (int) (Actuator.FanVoltage[0]/VMAXFAN)*
Actuator.FreqRatio;
numNeg = (int) (Actuator.FanVoltage[1]/VMAXFAN)*
Actuator.FreqRatio;
Actuator.counts = 1;
  }
  if (Actuator.counts<=numPos) {
Actuator.FanVoltage[0] = fullPos[0];
Actuator.FanVoltage[1] = fullPos[1];
  }
  else if (Actuator.counts<=numNeg) {
Actuator.FanVoltage[0] = fullNeg[0];
Actuator.FanVoltage[1] = fullNeg[1];
  }
}

/* Actually fire off the motors */
commandMotor(Actuator.FanVoltage);
```

```
    /* Release actuator read lock */
    pthread_rwlock_unlock(&ActuatorLock);

    /* Check for status changes */
    pthread_rwlock_rdlock(&StatusLock);

    TSrunning = TSrunMode;
    actMode = TSactuatorMode;

    pthread_rwlock_unlock(&StatusLock);
  }

  commandMotor(fanOff);
}

void setFanVoltage(int pc, double *v) {

  double absv=0.0, v1 = 0.0, v2 = 0.0, eps = .1;

  absv = (v[0]>=0) ? v[0] : -v[0];
  if (absv>=3.0) {
    v1 = absv;
    v2 = 0.0;
  }
  else {
    v2 = 3.0;
    v1 = absv+3.0;
  }

  if (absv<eps) {
    v1 = 0.0;
    v2 = 0.0;
  }

  // printf("volts=%lf\n",volts); fflush(stdout);

  /* Grab actuator write lock to update settings settings */
  pthread_rwlock_wrlock(&ActuatorLock);

  if (pc<2) /* Only using v[0] here! */
    {
      if (v[0] >= 0.0)
    {
      Actuator.FanVoltage[0] = v1;
      Actuator.FanVoltage[1] = v2;
    }
      else
    {
      Actuator.FanVoltage[0] = v2;
      Actuator.FanVoltage[1] = v1;
    }
    }
  else /* Control law will command both motors simultaneously */
       /* Don't allow negative voltages, regardless of
       /* controller commands */
       /* Motors wired to spin in only one direction! */
    {
      if (v[0]>0)
    Actuator.FanVoltage[0] = v[0];
      else
    Actuator.FanVoltage[0] = 0.0;
      if (v[1]>0)
    Actuator.FanVoltage[1] = v[1];
      else
    Actuator.FanVoltage[1] = 0.0;
    }

  /* Clip voltages to max allowable */
  if (Actuator.FanVoltage[0]>VMAXFAN) {
    Actuator.FanVoltage[0]=VMAXFAN;}
  if (Actuator.FanVoltage[1]>VMAXFAN) {
    Actuator.FanVoltage[1]=VMAXFAN;}

  /* Release actuator lock */
  pthread_rwlock_unlock(&ActuatorLock);
}


#include "../include/TS_Includes.h" #include
"../include/TS_Packet.h" #include "../include/TS_Messages.h"

/* Interprocess signalling and synchronization */

extern struct _pulse pulseCOM; extern int chan_idCOM;
extern pthread_rwlock_t StatusLock; extern unsigned char
TSrunMode;


/* Begin communications thread */

void *communication_thread(void *TSfoo) {
  unsigned char TSrunning = TRUE;
  int dsmpPort;

  printf("Starting comm thread\n"); fflush(stdout);

  /* Block until Comm thread pulse */
  MsgReceive(chan_idCOM, &pulseCOM, sizeof(pulseCOM), NULL);

  /* Initialize communications interface on first invocation */

  dsmpPort = dsmp_Init(myID,dsmpSERV_PORT);

  /* Register messages which we will send and respond to */

  printf("Registering messages\n"); fflush(stdout);

  /* Outgoing messages (no associated handlers) */

  dsmp_RegisterMesg(MSG_ackMsg,0,NULL,sizeof(byte));
  dsmp_RegisterMesg(MSG_uplinkStateData,0,NULL,
  sizeof(StateEstimate_t));
  dsmp_RegisterMesg(MSG_uplinkRawData,0,NULL,
  sizeof(SensorReadings_t));
  dsmp_RegisterMesg(MSG_uplinkStatusData,0,NULL,5*sizeof(byte));

  /* Incoming messages */

            /*  Mode switches  */

  dsmp_RegisterMesg(MSG_setRunMode,0,setRunMode,sizeof(byte));
  dsmp_RegisterMesg(MSG_setControlMode,0,setControlMode,
  sizeof(byte));
  dsmp_RegisterMesg(MSG_setFanMode,0,setFanMode,sizeof(byte));
  dsmp_RegisterMesg(MSG_setActuatorMode,0,setActuatorMode,
  sizeof(double));
  dsmp_RegisterMesg(MSG_setGyroMode,0,setGyroMode,sizeof(byte));

           /* Controller/observer data */

  dsmp_RegisterMesg(MSG_setControlData,0,setControlData,
  TS_VARSIZE);
  dsmp_RegisterMesg(MSG_setFanSpeedData,0,setFanSpeedData,
  TS_VARSIZE);
  dsmp_RegisterMesg(MSG_setFanFricData,0,setFanFricData,
  TS_VARSIZE);
  dsmp_RegisterMesg(MSG_setControlTarget,0,setControlTarget,
  3*sizeof(double));
  dsmp_RegisterMesg(MSG_setSampleRate,0,setSampleRate,
  2*sizeof(double));

  dsmp_RegisterMesg(MSG_setEstimatorData,0,setEstimatorData,
  TS_VARSIZE);
  dsmp_RegisterMesg(MSG_setGyroCalibData,0,setGyroCalibData,
  2*sizeof(double));

           /* Data uplink requests */

  dsmp_RegisterMesg(MSG_sendStateData,0,sendStateData,
  sizeof(byte));
  dsmp_RegisterMesg(MSG_sendRawData,0,sendRawData,sizeof(byte));
  dsmp_RegisterMesg(MSG_sendStatusData,0,sendStatusData,
  sizeof(byte));

           /* Vehicle Calibration Request */

  dsmp_RegisterMesg(MSG_calibrateSensors,0,calibrateSensors,
  sizeof(byte));


  printf("Entering main comm loop\n"); fflush(stdout);

  /* Main loop  */

  while (TSrunning) {

    /* Block until Controller thread pulse */
```

```
    MsgReceive(chan_idCOM, &pulseCOM, sizeof(pulseCOM), NULL);

    /* Check for messages, and handle as needed */
    dsmp_RecvMesg();

    /* Check for any status changes */
    pthread_rwlock_rdlock(&StatusLock);
    TSrunning = TSrunMode;
    pthread_rwlock_unlock(&StatusLock);
  }
  return(0);
}

#include "../include/TS_Controller.h" #include
"../include/TS_Actuator.h" #include "../include/TS_Math.h"

/* Main Controller structures */

extern Controller_t Controller; extern FanData_t  FanData;

/* Helper structures for controller computations */

extern StateEstimate_t StateEstimate;

/* Interprocess signalling and synchronization */

extern struct _pulse pulseC; extern int chan_idC;

extern pthread_rwlock_t StateLock, ControllerLock, FanLock,
StatusLock; extern unsigned char TSrunMode, TScontrolMode,
TSfanMode;

/* Begin Controller thread */

void *controller_thread(void *TSfoo) {

        static double xd[2], x[NX], v[NU], theTime;
    static double fanOFF[2]={0.0,0.0};
    double fricTorq = 0.0;
    unsigned char TSrunning=TRUE, controllerOn=TRUE,
    unsigned char fanCompOn=FALSE;
    int i;
```

```
printf("Starting controller thread\n"); fflush(stdout);
while (TSrunning) {

  /* Block until Controller thread pulse */
  MsgReceive(chan_idC, &pulseC, sizeof(pulseC), NULL);

  if (controllerOn) {

    /* Grab read lock for state data */
    pthread_rwlock_rdlock(&StateLock);

    /* Load state estimate into x */
    theTime = StateEstimate.TimeStamp;
    x[0] = StateEstimate.AzCss;
    x[1] = StateEstimate.AzTam;
    x[2] = StateEstimate.omega;
    x[3] = StateEstimate.nu1;
    x[4] = StateEstimate.nu2;

    /* We can release the state data lock now */
    pthread_rwlock_unlock(&StateLock);

    /* Grab read lock for the controller data and flags */
    pthread_rwlock_rdlock(&ControllerLock);

    /* Compute desired trajectory */
    desiredTrajectory(&Controller,xd,theTime);

    /* Compute control law:
       zc_{k+1} = Ac*zc_k + Bc1*x_k + Bc2*xd_k
            v_k = Cc*zc_k + Dc1*x_k + Dc2*xd_k  */

    for (i=0;i<NU;i++) Controller.comtorq[i] = 0.0;
    if (Controller.nc > 0) {
      for (i=0;i<Controller.nc;i++) Controller.z[i] = 0.0;
      sparseMatVec(Controller.A,Controller.zold,
      Controller.z,Controller.nc);
      sparseMatVec(Controller.B1,x,Controller.z,NX);
      sparseMatVec(Controller.B2,xd,Controller.z,2);
      sparseMatVec(Controller.C,Controller.zold,
      Controller.comtorq,Controller.nc);
    }
    sparseMatVec(Controller.D1,x,Controller.comtorq,NX);
    sparseMatVec(Controller.D2,xd,Controller.comtorq,2);

    /* Store the new zc for next iteration */
    for (i=0;i<Controller.nc;i++) {
        Controller.zold[i] = Controller.z[i];}

    Controller.comtorq[0] *= .767;
    fricTorq = 0.0;
    if (fanCompOn) {

      pthread_rwlock_rdlock(&FanLock);

    /* Put in nonlinear corrections for */
    /* fan T-w curve; include PI terms */

    /* Put in nonlinear corrections for spindle/fan friction */

      fricTorq += interpolateData(FanData.wfric,
      FanData.fanFric,FanData.nFricPts,x[2]);

      // printf("Friction=%lf\n",fricTorq);fflush(stdout);
      pthread_rwlock_unlock(&FanLock);

    }    /* End if_fanComp */

    Controller.comtorq[0] += fricTorq;
    for (i=0;i<NU;i++) v[i] = Controller.comtorq[i];

    /* Update the actuator settings */
    setFanVoltage(Controller.pc,v);

    /* Release the controller data lock */
    pthread_rwlock_unlock(&ControllerLock);

    // printf("%lf\tTorque=%lf\n",theTime,v[0]);

  }  /* End if_controllerOn */

  /* Check for status changes */
  pthread_rwlock_rdlock(&StatusLock);
```

```
      TSrunning = TSrunMode;
      if (TScontrolMode != controllerOn) {
        for (i=0;i<Controller.nc;i++)
           Controller.z[i] = Controller.zold[i] = 0.0;
            setFanVoltage(Controller.pc,fanOFF);
      }
      controllerOn = TScontrolMode;
      fanCompOn = TSfanMode;

      pthread_rwlock_unlock(&StatusLock);

    }  /* Do-while loop until vehicle shutdown */
}

void desiredTrajectory(Controller_t *Controller, double *xd,
double t) {

  static double A, w, t0;

  if (Controller->TrackMode == HOLD) {
    xd[0] = Controller->xd[0];
    xd[1] = Controller->xd[1];
  }
  else if (Controller->TrackMode == SINE) {
    A = Controller->xd[0];
    w = Controller->xd[1];
    t0 = Controller->t0;
    xd[0] = A*sin(w*(t-t0));
    xd[1] = A*w*cos(w*(t-t0));
  }

}

/* State Estimator Thread is responsible for the following:   */
/* 1) Perform A/D scan of the sensors, and save readings      */
/*      in SensorReadings array.                              */

/* 2) Perform state estimation (currently take a running      */
/*      average of sensor readings).                          */

/* 3) Convert sensor counts into engineering values for       */
/* use by the Controller thread.                              */

#include "../include/TS_Estimator.h" #include
"../include/TS_PromIO.h" #include "../include/TS_Math.h"

/* Main estimator structures */

extern Estimator_t Estimator; extern StateEstimate_t
StateEstimate;

/* Helper structures for data processing */

EngineeringUnits_t EngineeringUnits; SensorReadings_t
SensorReadings; extern SensorCalibration_t sensorCals;

/* Interprocess signalling and synchronization */

extern pthread_rwlock_t StateLock, EstimatorLock, SensorLock,
StatusLock; extern struct _pulse pulseSE; extern int chan_idSE;
extern unsigned char TSrunMode, TSgyroMode;


/* Begin estimator thread */

void *state_est_thread(void *TSfoo) {
    unsigned char TSrunning = TRUE, TSgyroCal = FALSE;
    static double xhat_raw[NY];
    int i;

    printf("Starting estimator thread\n"); fflush(stdout);

    while (TSrunning)
    {
      /* Wait for State Estimator pulse */
      MsgReceive(chan_idSE, &pulseSE, sizeof(pulseSE), NULL);

      /* Take a write lock for the sensor data */
      pthread_rwlock_wrlock(&SensorLock);

      /* Read the raw sensor values */
      if (!ReadSensors(&SensorReadings)) {
        printf("A/D error in ReadSensors -- aborting\n");
        TSrunMode = FALSE;
        return;
      }

          // printf("Starting unit conversion\n");fflush(stdout);
      /* Convert to sensor data to engineering units */
      ConvertToEngineeringUnits(&SensorReadings,&sensorCals,
      &EngineeringUnits);

      /* Release sensor data lock */
      pthread_rwlock_unlock(&SensorLock);

      /* Load the raw state estimate into xhat_raw */

      xhat_raw[0] = EngineeringUnits.AzCss;
      xhat_raw[1] = EngineeringUnits.AzTam;
      xhat_raw[2] = EngineeringUnits.GyroValue;
      xhat_raw[3] = EngineeringUnits.Fan1Speed;
      xhat_raw[4] = EngineeringUnits.Fan2Speed;

//    for (i=0;i<5;i++) printf("xraw[%1d]=%lf\n",i,xhat_raw[i]);
//    fflush(stdout);

      /* Take a read lock for the estimator data */
      pthread_rwlock_rdlock(&EstimatorLock);

      /* Filter xhat_raw according to specified estimation strategy
         zo_{k+1} = Ao*zo_k + Bo*xhat_raw_k
            xhat_k = Co*zo_k + Do*xhat_raw_k */
         //  printf("Beginning to filter data\n");fflush(stdout);

      for (i=0;i<NY;i++) Estimator.xhat[i] = 0.0;
      if (Estimator.no > 0) {
        for (i=0;i<Estimator.no;i++) Estimator.z[i] = 0.0;
        sparseMatVec(Estimator.A,Estimator.zold,Estimator.z,
        Estimator.no);
        sparseMatVec(Estimator.B,xhat_raw,Estimator.z,NY);
        sparseMatVec(Estimator.C,Estimator.zold,Estimator.xhat,
        Estimator.no);
      }
      sparseMatVec(Estimator.D,xhat_raw,Estimator.xhat,NY);
      /* Save the estimator state for the next step */
      for (i=0;i<Estimator.no;i++) {
        Estimator.zold[i] = Estimator.z[i]; }

      /* Implement Julie's adaptive gyro calib here, eventually */
      if (TSgyroCal) {}
        /*    adaptGyroCalib(&ahat,&bhat,&th_hat,xhat_new[2]); */

      /* Release the read lock for observer data*/
      pthread_rwlock_unlock(&EstimatorLock);

      /* Take a write lock for the State Estimate */
      pthread_rwlock_wrlock(&StateLock);

      /* Load the filtered estimates into StateEstimate */
      StateEstimate.TimeStamp = SensorReadings.TimeStamp;
      StateEstimate.AzCss = Estimator.xhat[0];
      StateEstimate.AzTam = Estimator.xhat[1];
      StateEstimate.omega = Estimator.ahat*Estimator.xhat[2]-
        Estimator.bhat;
      StateEstimate.nu1 = Estimator.xhat[3];
      StateEstimate.nu2 = Estimator.xhat[4];

//    for (i=0;i<5;i++) {
//      printf("xhat[%1d]=%lf\n",i,Estimator.xhat[i]);}
//    fflush(stdout);

      /* Release the estimate write lock */
      pthread_rwlock_unlock(&StateLock);

      /* Check for status changes */

      pthread_rwlock_rdlock(&StatusLock);

      TSrunning = TSrunMode;
      TSgyroCal = TSgyroMode;

      pthread_rwlock_unlock(&StatusLock);

    } /* Loop until vehicle shutdown */

    return;
```

```
}

void ConvertToEngineeringUnits(SensorReadings_t *sensors,
                  SensorCalibration_t *cals,
                  EngineeringUnits_t *Eunits)
{
    /* Misc. Variables  */

    double CSS1Volts, CSS2Volts, CSS3Volts, CSS4Volts;
    double GyroVolts, MagXVolts, MagYVolts, MagZVolts;
    double TamMag, TamTemp[3], BhatX, BhatY, BhatZ, AzTam=0.0;
    double CssNorm[4], CssA[4], Az=0.0;
    double CssMag, GyroValue, MinA;
    static double lastAz=5.0;
    short  i, MinCss;
    float TimeStamp;

    /* Get raw sensor readings in Volts */

    CSS1Volts = sensors->CSS1Volts;
    CSS2Volts = sensors->CSS2Volts;
    CSS3Volts = sensors->CSS3Volts;
    CSS4Volts = sensors->CSS4Volts;

    GyroVolts = sensors->GyroVolts;

    MagXVolts = sensors->MagXVolts;
    MagYVolts = sensors->MagYVolts;
    MagZVolts = sensors->MagZVolts;

    TimeStamp = sensors->TimeStamp;

    /*************************************************************
    /* Convert Volts to Engineering Units*/

    /* Gyro */
    GyroValue = (cals->Volts2Dps * (GyroVolts - cals->GyroOffset));

    /* Tam */
    /* Calculate Unit Magnetic Vector (Bhat) */
    TamTemp[0] = cals->TamGain[0] * (MagXVolts - cals->TamBias[0]);
    TamTemp[1] = cals->TamGain[1] * (MagYVolts - cals->TamBias[1]);
```

```
TamTemp[2] = cals->TamGain[2] * (MagZVolts - cals->TamBias[2]);
TamMag = sqrt( pow(TamTemp[0],2) + pow(TamTemp[1],2) +
    pow(TamTemp[2],2) );

BhatX = TamTemp[1]/TamMag;
BhatY = -TamTemp[0]/TamMag;
BhatZ = TamTemp[2]/TamMag;

/* Compute TAM azimuth */
AzTam = -atan2(TamTemp[1],TamTemp[0])*Rad2Deg;

/* CSS */
/* Normalize to CssMax */
CssNorm[0] = CSS1Volts/cals->CssMax[0];
CssNorm[1] = CSS2Volts/cals->CssMax[1];
CssNorm[2] = CSS3Volts/cals->CssMax[2];
CssNorm[3] = CSS4Volts/cals->CssMax[3];

/* Limit to +/- 1.0 */
if (CssNorm[0] > 1.0) CssNorm[0] = 1.0;
if (CssNorm[1] > 1.0) CssNorm[1] = 1.0;
if (CssNorm[2] > 1.0) CssNorm[2] = 1.0;
if (CssNorm[3] > 1.0) CssNorm[3] = 1.0;

/* Find the angle between Cssi and the sun. i = 1, 2, 3 */
CssA[0] = acos(CssNorm[0]) * Rad2Deg;
CssA[1] = acos(CssNorm[1]) * Rad2Deg;
CssA[2] = acos(CssNorm[2]) * Rad2Deg;
CssA[3] = acos(CssNorm[3]) * Rad2Deg;

/* Find the CSS angle which is min */

MinA = 80.0;
for (i=0;i<4;i++) {
  if (CssA[i]<MinA) {
    MinA = CssA[i];
    MinCss = i;
  }
}

if (MinA >= 80.0) {
  if (lastAz>0) Az = 180.0;
  else Az=-180.0;
}
else if (MinCss==0) {
  if (CssA[2]>60)
    Az = 90.0 + CssA[0];
  else
    Az = ( (90.0-CssA[0]) + (30.0+CssA[2]) )/2.0;
}
else if (MinCss==2) {
  if (CssA[0]>60)
    Az = ( (30.0-CssA[2]) + (-30.0+CssA[1]))/2.0;
  else
    Az = ( (30.0+CssA[2]) + (90.0-CssA[0]))/2.0;
}
else if (MinCss==1) {
  if (CssA[2]>60)
    Az = ( (-30.0-CssA[1]) + (-90.0+CssA[3]))/2.0;
  else
    Az = ( (30.0-CssA[2]) + (-30.0+CssA[1]))/2.0;
}
else if (MinCss==3) {
  if (CssA[1]>60)
    Az = -90.0-CssA[3];
  else
    Az = ( (-90.0+CssA[3]) + (-30.0-CssA[1]))/2.0;
}

lastAz = Az;

/***************************************************************/
/* Store Engineering Data */

/* Gyro */
Eunits->GyroValue = GyroValue;

/* Tam */
Eunits->BhatX = BhatX;
Eunits->BhatY = BhatY;
Eunits->BhatZ = BhatZ;
Eunits->AzTam = AzTam;

/* CSS */
```

```
    Eunits->NormToMaxCSS1Volts = CssNorm[0];
    Eunits->NormToMaxCSS2Volts = CssNorm[1];
    Eunits->NormToMaxCSS3Volts = CssNorm[2];
    Eunits->NormToMaxCSS4Volts = CssNorm[3];
    Eunits->CSSUnitVecX = 0.0;
    Eunits->CSSUnitVecY = 0.0;
    Eunits->CSSUnitVecZ = 1.0;
    Eunits->AzCss = Az;

    /* Fan speeds -- must write the code for these */
    Eunits->Fan1Speed = 0.0;
    Eunits->Fan2Speed = 0.0;

    /* Time */
    Eunits->TimeStamp = TimeStamp;

    return;
}


/* void adaptGyroCalib(double *ahat, double *bhat, double *th_hat,
double wm) */

#include "../include/TS_Init.h" #include
"../include/TS_Actuator.h" #include "../include/TS_Math.h"

double ActRate=100.0;       /* Can be modified by InitTimers below
*/ double ControlRate = 20.0;  /* Can be modified by InitTimers
below */

void InitController(Controller_t *Controller) {
  FILE *fp;
  char dummy[50];
  double Pgain, Dgain, xd;
  double D1[5], D2[2];

  if ((fp = fopen("../init/TS_InitController.dat","r")) == NULL)
    {
      printf("Error opening Controller Vars input file.\n");
      exit(-1);
    }
  fscanf(fp," %s %lf %s %lf %s %lf", dummy, &Pgain, dummy,
   &Dgain, dummy, &xd);

  D1[0] = 0.0;
  D1[1] = -Pgain;
  D1[2] = -Dgain;
  D1[3] = 0.0;
  D1[4] = 0.0;

  D2[0] = Pgain;
  D2[1] = 0.0;

  /* Initialize controller to be PD tracking at "sun" offset xd */

  Controller->nc=0;
  Controller->pc=1;

  Controller->z = NULL;
  Controller->zold = NULL;

  Controller->A = NULL;
  Controller->B1 = NULL;
  Controller->B2 = NULL;
  Controller->C = NULL;

  Controller->D1 = makeSparse(D1,1,5);
  Controller->D2 = makeSparse(D2,1,2);

  Controller->xd[0] = xd;
  Controller->xd[1] = 0.0;

  Controller->TrackMode = 0;  /* 0 = HOLD */

  Controller->t0 = 0.0;

  Controller->comtorq[0] = 0.0;
  Controller->comtorq[1] = 0.0;

  printSparse(Controller->D1,"Controller D1 is \n");
  printSparse(Controller->D2,"Controller D2 is \n");

}
```

```
void InitEstimator(Estimator_t *Estimator) {
  double Do[25];
  short i,j;

  for (i=0;i<5;i++) {
    for (j=0;j<5;j++) {
      if (i==j) Do[5*i+j]=1.0;
      else Do[5*i+j] = 0.0;
    }
  }

  /* Initialize estimator to simply pass through raw, */
  /* scaled readings */

  Estimator->no=0;

  Estimator->z = NULL;
  Estimator->zold = NULL;

  Estimator->A = NULL;
  Estimator->B = NULL;
  Estimator->C = NULL;

  Estimator->D = makeSparse(Do,5,5);

  for (i=0;i<NY;i++) Estimator->xhat[i] = 0.0;

  Estimator->adaptGyroGains[0] = 0.0;
  Estimator->adaptGyroGains[1] = 0.0;

  Estimator->th = 0.0;
  Estimator->thold = 0.0;
  Estimator->ahat = 1.0;
  Estimator->bhat = 0.0;
}

void InitActuator(Actuator_t *Actuator) {

  Actuator->ref_voltage = VMAXFAN;
  Actuator->Freq = ActRate;
  Actuator->FreqRatio = (int) (ActRate/ControlRate);
  Actuator->FanVoltage[0] = 0.0;
  Actuator->FanVoltage[1] = 0.0;

  Actuator->Deadzone = 0.0;
  Actuator->counts = 1;
}

void InitFanData(FanData_t *FanData) {

  int nFricPts = 5;
  int nFanPts = 2;

  FanData->nFricPts = nFricPts;
  FanData->nFanPts  = nFanPts;

  FanData->wfric = (double *) malloc(nFricPts*sizeof(double));
  FanData->wfan = (double *) malloc(nFanPts*sizeof(double));

  FanData->fanFric = (double *) malloc(nFricPts*sizeof(double));
  FanData->fanForce = (double *) malloc(nFanPts*sizeof(double));

  FanData->wfric[0] = -120.0;    /* This is deg/sec! */
  FanData->wfric[1] = -.1;
  FanData->wfric[2] =    0;
  FanData->wfric[3] =  .1;
  FanData->wfric[4] = 120.0;

  FanData->wfan[0]  = 0.0;      /* This is RPM! */
  FanData->wfan[0]  = 3500.0;

  /* Simple stiction + constant rolling friction model */
  /* Units are fan volts */

  FanData->fanFric[0] = -4;
  FanData->fanFric[1] = -4;
  FanData->fanFric[2] = 0.0;
  FanData->fanFric[3] = 4;
  FanData->fanFric[4] = 4;

  FanData->fanForce[0] = 0.0;
  FanData->fanForce[1] = 1.0;
```

```
 /* Nominal linear fan output model */
}

void InitCalibration(SensorCalibration_t *theCals) {

  FILE* fp;
  /* Calibration Constants from CalibrationConstants.dat:   */
  /*        GyroOffset (Volts), TamBias (Volts), TamGain, */
  /*        CssMax (Volts), BEarthRef (nT) */
  double TamBias[3];
  double TamGain[3];
  double CssMax[4];
  double BEarthRef[3];
  double GyroOffset;
  double Azm;
  char dummy[50];
  int i;

  /************************************************************/
  /*Read in Calibration constants */
  if ((fp = fopen("../init/TS_CalibrationConstants.dat","r"))
     == NULL)
    {
      printf("Error opening Calibration Constant input file.\n");
      exit(-1);
    }

  fscanf(fp," %s %lf %s %lf %lf %lf %s %lf %lf %lf %s %lf
    %lf %lf %lf %s %lf %lf %lf %lf %s %lf %lf %lf",
     dummy, &GyroOffset, dummy, &TamBias[0], &TamBias[1],
     &TamBias[2], dummy, &TamGain[0],&TamGain[1],&TamGain[2],
     dummy, &CssMax[0], &CssMax[1], &CssMax[2], &CssMax[3],
     dummy, &BEarthRef[0],&BEarthRef[1], &BEarthRef[2]);
  fclose(fp);

  /************************************************************/
  /* Define relevent conversions */
  theCals->Volts2Dps = -47.45; /* dps/V (unknown at this point) */
  theCals->CssMaxMean = (CssMax[0] + CssMax[1] + CssMax[2])/3.0;
   /* Volts */
  theCals->Azm = 0.0;
  theCals->GyroOffset = GyroOffset;
  for (i=0;i<3;i++) {
    theCals->TamBias[i] = TamBias[i];
    theCals->TamGain[i] = TamGain[i];
    theCals->CssMax[i] = CssMax[i];
    theCals->BEarthRef[i] = BEarthRef[i];
  }
  theCals->CssMax[3] = CssMax[3];

  return;
}

int MakeTimer(timer_t *theTimer) {
  struct sigevent event;
  int connect_id, chan_id;

  /************** Open and test the new timer  ***************/

  if ( (chan_id = ChannelCreate (0) ) == -1) {
    printf("Error creating timer channel. \n");
    exit(-1);
  }
  if ( (connect_id = ConnectAttach(0, 0, chan_id, 0, 0) ) == -1) {
    printf("Error connecting timer channel.\n");
    exit(-1);
  }
  SIGEV_PULSE_INIT (&event, connect_id,
    SIGEV_PULSE_PRIO_INHERIT, 1, 0);
  if (timer_create(CLOCK_REALTIME, &event, theTimer) == -1) {
    printf ("Error creating timer. \n");
    exit(-1);
  }

  /* Note: timer is now inactive.  It will activate with call to
     timer_settime in main() after R/W locks are inited.  */

  return(chan_id);

}

void InitTimers(struct itimerspec *timerCptr, struct itimerspec
*timerSEptr,struct itimerspec *timerCOMptr,
             struct itimerspec *timerACTptr)
{
  double EstRate, CommRate;
  char  dummy[100];

  FILE* fp;

  if ((fp = fopen("../init/TS_InitTimerVals.dat","r")) == NULL)
    {
      printf("Error opening Timer Init input file.\n");
      exit(-1);
    }
  fscanf(fp," %s %lf %s %lf", dummy, &EstRate, dummy, &ControlRate);
  fscanf(fp," %s %lf %s %lf", dummy, &ActRate, dummy, &CommRate);
  fclose(fp);

  printf("Est rate = %lf\n",EstRate);

  /************* Set timer values ***********************/

  setTimerVal(timerCptr,ControlRate);
  setTimerVal(timerSEptr,EstRate);
  setTimerVal(timerCOMptr,CommRate);
  setTimerVal(timerACTptr,ActRate);

  return;
}

void setTimerVal(struct itimerspec *theTimer, double theRate) {
  int theSecs;
  long theNanoSecs;

  theSecs = (int)(1/theRate);
  theNanoSecs = (long)(((1.0/theRate) - theSecs)*1E9);

  theTimer->it_value.tv_sec = 1;
  theTimer->it_value.tv_nsec = 0;
  theTimer->it_interval.tv_sec = theSecs;
  theTimer->it_interval.tv_nsec = theNanoSecs;

}
```

```
#include "../include/TS_Main.h" #include "../include/TS_Init.h"
#include "../include/TS_PromIO.h"

/* Declare the fundamental TableSat data structures */

Controller_t    Controller; Estimator_t     Estimator; Actuator_t
Actuator; FanData_t       FanData; StateEstimate_t StateEstimate;

/* Structure to store sensor calibrations */ SensorCalibration_t
sensorCals;

int main() {

  /* Declare timers and threads */
  struct itimerspec timerC, timerSE, timerCOM, timerACT;
  pthread_t  threadC, threadSE, threadCOM, threadACT;

  /* Store the start time */
  gettimeofday(&start_time, NULL);

  /* Initialize Prometheus Board */
  InitPrometheus();

  /* Initialize Sensor Calibration Constants  */
  InitCalibration(&sensorCals);

  /* Initialize Timer settings */
  InitTimers(&timerC,&timerSE,&timerCOM,&timerACT);

  /* Initialize Estimator parameters */
  InitEstimator(&Estimator);

  /* Initialize State Estimate */

  StateEstimate.TimeStamp = 0.0;
  StateEstimate.AzTam     = 0.0;
  StateEstimate.AzCss     = 0.0;
  StateEstimate.omega     = 0.0;
  StateEstimate.nu1       = 0.0;
  StateEstimate.nu2       = 0.0;

  /* Initialize Controller parameters */
  InitController(&Controller);

  /* Initialize Actuator parameters */
  InitActuator(&Actuator);
  InitFanData(&FanData);

  /* Create Timers */
  chan_idC   = MakeTimer(&timeridC);
  chan_idSE  = MakeTimer(&timeridSE);
  chan_idCOM = MakeTimer(&timeridCOM);
  chan_idACT = MakeTimer(&timeridACT);

  /* Initialize R/W Locks */
  pthread_rwlock_init(&SensorLock, NULL);
  pthread_rwlock_init(&StateLock, NULL);
  pthread_rwlock_init(&EstimatorLock, NULL);
  pthread_rwlock_init(&ControllerLock, NULL);
  pthread_rwlock_init(&FanLock, NULL);
  pthread_rwlock_init(&ActuatorLock, NULL);
  pthread_rwlock_init(&StatusLock, NULL);

  /* Start Timers at default rates */
  timer_settime(timeridC, 0, &timerC, NULL);
  timer_settime(timeridSE, 0, &timerSE, NULL);
  timer_settime(timeridCOM, 0, &timerCOM, NULL);
  timer_settime(timeridACT, 0, &timerACT, NULL);

  /* Spawn threads */
  pthread_create(&threadC,NULL,&controller_thread,NULL);
  pthread_create(&threadSE,NULL,&state_est_thread,NULL);
  pthread_create(&threadCOM,NULL,&communication_thread,NULL);
  pthread_create(&threadACT,NULL,&actuator_thread,NULL);

  /* Wait for threads to finish */
  pthread_join(threadC, NULL);
  pthread_join(threadSE, NULL);
  pthread_join(threadCOM, NULL);
  pthread_join(threadACT, NULL);

  return(0);
}
```

```
#include "../include/TS_Math.h"

/* Note: no sanity checking here. Can bomb if 2 successive points
in x array are the same.  Clips out-of-range values -- does not
extrapolate */

double interpolateData(double *xvals, double *yvals, int nvals,
double x) {
  double *xptr = xvals;
  double *yptr = yvals;

  if (x<=xvals[0])
    return(yvals[0]);
  else if (x>=xvals[nvals-1])
    return(yvals[nvals-1]);
  else {
    while (x>*xptr) {xptr++;yptr++;}
    xptr--;yptr--;
    return( yptr[0] + (yptr[1]-yptr[0])*(x-xptr[0])/
    (xptr[1]-xptr[0]) );
  }
}

/* Note that the result vector must be initialized to zero by the
caller
   if needed/wanted!  */

sparseMat *makeSparse(double *M, int nrows, int ncols) {
  int i,j;
  int *nvals, **cols;
  double **vals;
  sparseMat *Ms;

  nvals = (int *) malloc(nrows*sizeof(int));
  cols = (int **) malloc(nrows*sizeof(int));
  vals = (double **) malloc(nrows*sizeof(double));

  Ms = (sparseMat *) malloc(sizeof(sparseMat));
  Ms->nrows = nrows;
  Ms->nvals = (int *) malloc(nrows*sizeof(int));
  Ms->colVals = (cvals **) malloc(nrows*sizeof(cvals *));
  for (i=0;i<nrows;i++) {
    nvals[i]=0;
    cols[i] = (int *) malloc(ncols*sizeof(int));
    vals[i] = (double *) malloc(ncols*sizeof(double));
    for (j=0;j<ncols;j++) {
      if (ABS(M[i*ncols+j])>TINY) {
    cols[i][nvals[i]] = j;
    vals[i][nvals[i]] = M[i*ncols+j];
    nvals[i]++;
      }
    }
  }

  for (i=0;i<nrows;i++) {
    Ms->nvals[i] = nvals[i];
    Ms->colVals[i] = (cvals *) malloc(nvals[i]*sizeof(cvals));
    for (j=0;j<nvals[i];j++) {
      Ms->colVals[i][j].col = cols[i][j];
      Ms->colVals[i][j].val = vals[i][j];
    }
  }

  for (i=0;i<nrows;i++) {
    free(cols[i]);
    free(vals[i]);
  }
  free(nvals);
  free(cols);
  free(vals);

  return(Ms);
}

void freeSparse(sparseMat *M) {

  int i;

  if (M==NULL) {
    printf("Possible error: freeing empty sparseMat\n");
    fflush(stdout);
    return;
  }
  for (i=0;i<M->nrows;i++) free(M->colVals[i]);
  free(M->colVals);
  free(M->nvals);
  free(M);
}


void sparseMatVec(sparseMat *Mat, double *vecIn, double *vecOut,
int lenIn) {

  int i,j,col;
  double val;

  if (Mat==NULL) {
    printf("Error: multiplying by empty sparseMat.\n");
    fflush(stdout);
    return;
  }

  for (i=0;i<Mat->nrows;i++) {
    for (j=0; j<Mat->nvals[i]; j++) {
      col = Mat->colVals[i][j].col;
      val = Mat->colVals[i][j].val;
      if (col > lenIn-1) {
    printf("Size mismatch in sparseMatVec! \n");
    return;
      }
      vecOut[i] += val*vecIn[col];
    }
  }
}

void printSparse(sparseMat *Mat, const char *Mesg) {

  int i,j,col;
  double val;

  if (Mat==NULL) {
    printf("Possible error: attempt to print empty sparseMat\n");
    fflush(stdout);
    return;
  }

  printf("%s",Mesg);
  for (i=0;i<Mat->nrows;i++) {
    for (j=0; j<Mat->nvals[i]; j++) {
      col = Mat->colVals[i][j].col;
      val = Mat->colVals[i][j].val;
      printf("M[%d][%d]=%lf\n",i,col,val);
    }
  }
  fflush(stdout);
}

void matVec(double *mat, double *vec, double *vecres,int nrow,int
ncol) {
  double sum;
  short i,j;

  for (i=0;i<nrow;i++) {
    sum = 0.0;
    for (j=0;j<ncol;j++) sum += mat[i*ncol+j]*vec[j];
    vecres[i] = sum;
  }
}

#include "../include/TS_Includes.h" #include
"../include/TS_Packet.h" #include "../include/TS_Messages.h"
#include "../include/TS_Init.h" #include
"../include/TS_Actuator.h" #include "../include/TS_Math.h"

/* Need read/write access to all of the basic TableSat structures
*/

extern Controller_t Controller; extern Estimator_t Estimator;
extern Actuator_t Actuator; extern FanData_t    FanData;

extern StateEstimate_t StateEstimate; extern SensorReadings_t
SensorReadings;

/* Interprocess signalling and synchronization */

extern pthread_rwlock_t ActuatorLock, ControllerLock,
EstimatorLock; extern pthread_rwlock_t StateLock, SensorLock,
StatusLock, FanLock;

extern timer_t timeridC, timeridSE;

extern byte TSrunMode; extern byte TScontrolMode, TSactuatorMode;
extern byte TSfanMode, TSgyroMode;

extern struct timeval start_time;


/************* Messages to change vehicle status/mode  *********/

int setRunMode(byte whoFrom, byte *data) {
  static byte thisMesg=MSG_setRunMode;

  pthread_rwlock_wrlock(&StatusLock);

  memcpy(&TSrunMode,data,sizeof(byte));

  printf("Received new run mode: %u\n",TSrunMode);
  fflush(stdout);

  pthread_rwlock_unlock(&StatusLock);

  dsmp_SendMesg(MSG_ackMsg, (void *) &thisMesg, SAT_CON);

  return(TRUE);
}

int setControlMode(byte whoFrom, byte *data) {
  struct timeval cur_time;
  double nowtime;

  static byte thisMesg=MSG_setControlMode;

  pthread_rwlock_wrlock(&StatusLock);

  memcpy(&TScontrolMode,data,sizeof(byte));

  printf("Received new control mode: %u\n",TScontrolMode);
  fflush(stdout);
  pthread_rwlock_unlock(&StatusLock);

  gettimeofday(&cur_time, NULL);
  nowtime = ((double) (cur_time.tv_sec - start_time.tv_sec))
    +  (((double) (cur_time.tv_usec - start_time.tv_usec))/1.0E6);

  pthread_rwlock_wrlock(&ControllerLock);
  Controller.t0 = nowtime;
  pthread_rwlock_unlock(&ControllerLock);

  dsmp_SendMesg(MSG_ackMsg, (void *) &thisMesg, SAT_CON);

  return(TRUE);
}

int setFanMode(byte whoFrom, byte *data) {

  static byte thisMesg=MSG_setFanMode;

  pthread_rwlock_wrlock(&StatusLock);

  memcpy(&TSfanMode,data,sizeof(byte));

  printf("Received new fan mode: %u\n",TSfanMode);
  fflush(stdout);

  pthread_rwlock_unlock(&StatusLock);

  dsmp_SendMesg(MSG_ackMsg, (void *) &thisMesg, SAT_CON);

  return(TRUE);
}

int setActuatorMode(byte whoFrom, byte *data) {

  double actModed;
  static byte thisMesg=MSG_setActuatorMode;

  memcpy(&actModed,data,sizeof(double));

  pthread_rwlock_wrlock(&StatusLock);
  TSactuatorMode = (int) actModed;

  printf("Received new actuator mode: %u\n",TSactuatorMode);
  fflush(stdout);

  pthread_rwlock_unlock(&StatusLock);

  /* Compute new deadzone, if appropriate for mode */

  if (actModed>=BANGBANG) {
    pthread_rwlock_wrlock(&ActuatorLock);
    Actuator.Deadzone = VMAXFAN*(actModed-BANGBANG);
    printf("Actuator deadzone is: %lf\n",Actuator.Deadzone);
     fflush(stdout);
    pthread_rwlock_unlock(&ActuatorLock);
  }

  dsmp_SendMesg(MSG_ackMsg, (void *) &thisMesg, SAT_CON);

  return(TRUE);
}

int setGyroMode(byte whoFrom, byte *data) {

  static byte thisMesg=MSG_setGyroMode;

  pthread_rwlock_wrlock(&StatusLock);

  memcpy(&TSgyroMode,data,sizeof(byte));

  printf("Received new gyro mode: %u\n",TSgyroMode);
  fflush(stdout);

  pthread_rwlock_unlock(&StatusLock);

  dsmp_SendMesg(MSG_ackMsg, (void *) &thisMesg, SAT_CON);

  return(TRUE);
}

/**********  Messages to uplink vehicle state, measurements, and
```

```
/* status  **********/

int sendStateData(byte whoFrom, byte *data) {

  pthread_rwlock_wrlock(&StateLock);

  pthread_rwlock_rdlock(&ActuatorLock);
  StateEstimate.nu1 = Actuator.FanVoltage[0];
  StateEstimate.nu2 = Actuator.FanVoltage[1];
  pthread_rwlock_unlock(&ActuatorLock);

  // printf("Time is %lf\n",StateEstimate.TimeStamp);
  fflush(stdout);
  dsmp_SendMesg(MSG_uplinkStateData, (void *) &StateEstimate,
   SAT_CON);

  pthread_rwlock_unlock(&StateLock);

  return(TRUE);
}

int sendRawData(byte whoFrom, byte *data) {

  // printf("Received request for Raw data:\n"); fflush(stdout);

  pthread_rwlock_rdlock(&SensorLock);

  // printf("Current CSS volts are %lf\n",
  // SensorReadings.CSS2Volts);
  // printf("Current Gyro volts are %lf\n",
  // SensorReadings.GyroVolts);
  // printf("Current TAM volts are %lf\n",
  // SensorReadings.MagYVolts);
  // fflush(stdout);

  dsmp_SendMesg(MSG_uplinkRawData, (void *)
  &SensorReadings, SAT_CON);

  pthread_rwlock_unlock(&SensorLock);

  return(TRUE);
}
```

```
int sendStatusData(byte whoFrom, byte *data) {

  byte statusData[5];

  printf("Received request for Status data:\n"); fflush(stdout);

  pthread_rwlock_rdlock(&StatusLock);

  statusData[0] = TSrunMode;
  statusData[1] = TScontrolMode;
  statusData[2] = TSfanMode;
  statusData[3] = TSactuatorMode;
  statusData[4] = TSgyroMode;

  pthread_rwlock_unlock(&StatusLock);

  dsmp_SendMesg(MSG_uplinkStatusData, (void *)
  &statusData, SAT_CON);

  return(TRUE);
}

/**** Message to adjust the onboard sample rate *******/

int setSampleRate(byte whoFrom, byte *data) {

  double newRate[2];
  int newSecs;
  long newNanoSecs;
  struct itimerspec timerC, timerSE;
  byte thisMesg = MSG_setSampleRate;

  memcpy(&newRate,data,2*sizeof(double));
  printf("Received request for new sample rates: %lf\t%lf\n",
    newRate[0],newRate[1]);
  fflush(stdout);

  setTimerVal(&timerC, newRate[0]);
  setTimerVal(&timerSE, newRate[1]);

  /* Set the control and estimation threads to run at new rates */
  timer_settime(timeridC, 0, &timerC, NULL);
  timer_settime(timeridSE, 0, &timerSE, NULL);

  /* Update (Actuator rate)/(controller rate) */
  /* ratio for PWM (if used) */

  pthread_rwlock_wrlock(&ActuatorLock);
  Actuator.FreqRatio = (int) (Actuator.Freq/newRate[0]);
  pthread_rwlock_unlock(&ActuatorLock);

  dsmp_SendMesg(MSG_ackMsg, (void *) &thisMesg, SAT_CON);

  return(TRUE);
}

/* Messages to load friction and fan models for controller*/

int setFanFricData(byte whoFrom, byte *data) {
  double nFricd;
  int dataLen;
  byte *dataPtr = data;
  byte thisMesg = MSG_setFanFricData;

  pthread_rwlock_wrlock(&FanLock);

  printf("Old fan friction model with %d points\n",
  FanData.nFricPts);

  free(FanData.wfric);
  free(FanData.fanFric);
  memcpy(&nFricd,dataPtr,sizeof(double));
  dataPtr += sizeof(double);

  FanData.nFricPts = (int) nFricd;
  dataLen = FanData.nFricPts*sizeof(double);
  printf("New fan friction model with %d points\n",
  FanData.nFricPts);

  FanData.wfric   = (double *) malloc(dataLen);
  FanData.fanFric = (double *) malloc(dataLen);

  memcpy(FanData.wfric,dataPtr,dataLen);
  dataPtr += dataLen;

  memcpy(FanData.fanFric,dataPtr,dataLen);

  printf("New fan friction model with %d points\n",
  FanData.nFricPts);
  printf("First omega is %lf\n",FanData.wfric[0]);
  printf("Last omega is %lf\n",FanData.wfric[FanData.nFricPts-1]);
  printf("First fric is %lf\n",FanData.fanFric[0]);
  printf("Last fric is %lf\n",FanData.fanFric[FanData.nFricPts-1]);
  fflush(stdout);

  pthread_rwlock_unlock(&FanLock);

  dsmp_SendMesg(MSG_ackMsg, (void *) &thisMesg, SAT_CON);

  return(TRUE);
}

int setFanSpeedData(byte whoFrom, byte *data) {
  double nFricd;
  int dataLen;
  byte *dataPtr = data;
  byte thisMesg = MSG_setFanSpeedData;

  pthread_rwlock_wrlock(&FanLock);

  printf("Old fan friction model with %d points\n",
  FanData.nFanPts);
  fflush(stdout);

  free(FanData.wfan);
  free(FanData.fanForce);

  memcpy(&nFricd,dataPtr,sizeof(double));
  dataPtr += sizeof(double);

  FanData.nFanPts = (int) nFricd;
  dataLen = FanData.nFanPts*sizeof(double);

  FanData.wfan     = (double *) malloc(dataLen);
  FanData.fanForce = (double *) malloc(dataLen);
  memcpy(FanData.wfan,dataPtr,dataLen);
  dataPtr += dataLen;

  memcpy(FanData.fanForce,dataPtr,dataLen);

  printf("New fan speed model with %d points\n",
  FanData.nFanPts);
  printf("First omega is %lf\n",FanData.wfan[0]);
  printf("Last omega is %lf\n",FanData.wfan[FanData.nFanPts-1]);
  fflush(stdout);

  pthread_rwlock_unlock(&FanLock);

  dsmp_SendMesg(MSG_ackMsg, (void *) &thisMesg, SAT_CON);

  return(TRUE);
}

/** Messages to set other Controller data ******/

int setControlData(byte whoFrom, byte *data) {

  int nc, pc;
  int Asz, B1sz, B2sz, Csz, D1sz, D2sz;
  double *Ac, *B1c, *B2c, *Cc, *D1c, *D2c;
  sparseMat *Acs, *B1cs, *B2cs, *Ccs, *D1cs, *D2cs;
  static double mSizes[6];

  byte *dataPtr=data;
  static byte thisMesg=MSG_setControlData;

  /* Read in and verify the matrix sizes */

  memcpy(mSizes,dataPtr,6*sizeof(double));

  nc = (int) mSizes[0];
  pc = (int) mSizes[3];
  if ( ( (int) mSizes[1] != NX ) ||  // Cols of B1
       ( (int) mSizes[2] != 2  ) ||  // Cols of B2
       ( (int) mSizes[3]  > 2  ) ||  // Rows of C
       ( (int) mSizes[4] != NX ) ||  // Cols of D1
```

```
     ( (int) mSizes[5] != 2  ) )   // Cols of D2
  {
    printf("Invalid controller specification!\n"); fflush(stdout);
    return(0);
  }

/* Read in the fully populated controller matrices */

dataPtr+=6*sizeof(double);
printf("New controller: nc=%d\n",nc);
fflush(stdout);

if (nc>0)
  {
    Asz =  nc*nc*sizeof(double);
    B1sz = nc*NX*sizeof(double);
    B2sz = nc*2*sizeof(double);
    Csz =  pc*nc*sizeof(double);
  }
D1sz = pc*NX*sizeof(double);
D2sz = pc*2*sizeof(double);

if(nc>0)
  {
    Ac  = (double *) malloc(Asz);
    B1c = (double *) malloc(B1sz);
    B2c = (double *) malloc(B2sz);
    Cc  = (double *) malloc(Csz);
  }
D1c = (double *) malloc(D1sz);
D2c = (double *) malloc(D2sz);

if (nc>0)
  {
    memcpy(Ac,dataPtr,Asz);
    dataPtr += Asz;

    memcpy(B1c,dataPtr,B1sz);
    dataPtr += B1sz;

    memcpy(B2c,dataPtr,B2sz);
    dataPtr += B2sz;
    memcpy(Cc,dataPtr,Csz);
    dataPtr += Csz;
  }

memcpy(D1c,dataPtr,D1sz);
dataPtr += D1sz;

memcpy(D2c,dataPtr,D2sz);

/* Sparsify the downlinked matrices */

if (nc>0)
  {
    Acs = makeSparse(Ac,nc,nc);
    B1cs = makeSparse(B1c,nc,NX);
    B2cs = makeSparse(B2c,nc,2);
    Ccs = makeSparse(Cc,pc,nc);
  }
D1cs = makeSparse(D1c,pc,NX);
D2cs = makeSparse(D2c,pc,2);

/* Take a write lock on the controller data */
pthread_rwlock_wrlock(&ControllerLock);

/* Free the old controller data */

if (Controller.nc>0)
  {
    freeSparse(Controller.A);
    freeSparse(Controller.B1);
    freeSparse(Controller.B2);
    freeSparse(Controller.C);
  }
freeSparse(Controller.D1);
freeSparse(Controller.D2);
free(Controller.z);
free(Controller.zold);

/* Fill in the new controller data */

Controller.nc = nc;
Controller.pc = pc;

if (nc>0)
  {
    Controller.A  = Acs;
    Controller.B1 = B1cs;
    Controller.B2 = B2cs;
    Controller.C  = Ccs;
  }
else
  {
    Controller.A  = NULL;
    Controller.B1 = NULL;
    Controller.B2 = NULL;
    Controller.C  = NULL;
  }
Controller.D1 = D1cs;
Controller.D2 = D2cs;

if (nc>0)
  {
    Controller.z    = (double *) calloc(nc,sizeof(double));
    Controller.zold = (double *) calloc(nc,sizeof(double));
  }
else
  {
    Controller.z    = NULL;
    Controller.zold = NULL;
  }

printSparse(Controller.A,"New controller A matrix:\n");
printSparse(Controller.B1,"New controller B1 matrix:\n");
printSparse(Controller.B2,"New controller B2 matrix:\n");
printSparse(Controller.C,"New controller C matrix:\n");
printSparse(Controller.D1,"New controller D1 matrix:\n");
printSparse(Controller.D2,"New controller D2 matrix:\n");

/* Release controller data lock */
pthread_rwlock_unlock(&ControllerLock);

/* Free the temporary arrays that we used */
```

```
  if (nc>0)
    {
      free(Ac);
      free(B1c);
      free(B2c);
      free(Cc);
    }
  free(D1c);
  free(D2c);

  dsmp_SendMesg(MSG_ackMsg, (void *) &thisMesg, SAT_CON);

  return(TRUE);
}

int setControlTarget(byte whoFrom, byte *data) {
  static byte thisMesg=MSG_setControlTarget;
  double theData[3];

  pthread_rwlock_wrlock(&ControllerLock);

  memcpy(&theData,data,3*sizeof(double));

  Controller.TrackMode = (int) theData[0];
  Controller.xd[0] = theData[1];
  Controller.xd[1] = theData[2];

  // printf("Received new controller target: %lf\t%lf\n",
  //    theData[1],theData[2]);
  // fflush(stdout);

  pthread_rwlock_unlock(&ControllerLock);

  // dsmp_SendMesg(MSG_ackMsg, (void *) &thisMesg, SAT_CON);

  return(TRUE);

}

/*  Messages to set the Estimator data *****/

int setEstimatorData(byte whoFrom, byte *data) {
```

```
int no;
int Asz, Bsz, Csz, Dsz;
double *Ac, *Bc, *Cc, *Dc;
sparseMat *Acs, *Bcs, *Ccs, *Dcs;
static double mSizes[4];

byte *dataPtr=data;
static byte thisMesg=MSG_setEstimatorData;

/* Read in and verify the matrix sizes */

memcpy(mSizes,dataPtr,4*sizeof(double));

no = (int) mSizes[0];
if ( ( (int) mSizes[1] != NX ) ||  // Cols of B
     ( (int) mSizes[2] != NX  ) || // Rows of C
     ( (int) mSizes[3] != NX ) )   // Cols of D
  {
    printf("Invalid observer specification!\n"); fflush(stdout);
    return(0);
  }

/* Read in the fully populated estimator matrices */

dataPtr+=4*sizeof(double);
printf("New observer: no=%d\n",no); fflush(stdout);

if (no>0)
  {
    Asz =  no*no*sizeof(double);
    Bsz =  no*NX*sizeof(double);
    Csz =  NX*no*sizeof(double);
  }
Dsz = NX*NX*sizeof(double);

if(no>0)
  {
    Ac  = (double *) malloc(Asz);
    Bc = (double *) malloc(Bsz);
    Cc  = (double *) malloc(Csz);
  }
```

```
Dc = (double *) malloc(Dsz);

if (no>0)
  {
    memcpy(Ac,dataPtr,Asz);
    dataPtr += Asz;

    memcpy(Bc,dataPtr,Bsz);
    dataPtr += Bsz;

    memcpy(Cc,dataPtr,Csz);
    dataPtr += Csz;
  }

memcpy(Dc,dataPtr,Dsz);

/* Sparsify the downlinked matrices */

if (no>0)
  {
    Acs = makeSparse(Ac,no,no);
    Bcs = makeSparse(Bc,no,NY);
    Ccs = makeSparse(Cc,NY,no);
  }
Dcs = makeSparse(Dc,NY,NY);

/* Take a write lock on the observer data */
pthread_rwlock_wrlock(&EstimatorLock);

/* Free the old estimator data */

if (Estimator.no>0)
  {
    freeSparse(Estimator.A);
    freeSparse(Estimator.B);
    freeSparse(Estimator.C);
  }
freeSparse(Estimator.D);
free(Estimator.z);
free(Estimator.zold);

/* Fill in the new controller data */
```

```
Estimator.no = no;

if (no>0)
  {
    Estimator.A  = Acs;
    Estimator.B = Bcs;
    Estimator.C  = Ccs;
  }
else
  {
    Estimator.A  = NULL;
    Estimator.B = NULL;
    Estimator.C  = NULL;
  }
Estimator.D = Dcs;

if (no>0)
  {
    Estimator.z    = (double *) calloc(no,sizeof(double));
    Estimator.zold = (double *) calloc(no,sizeof(double));
  }
else
  {
    Estimator.z    = NULL;
    Estimator.zold = NULL;
  }

printSparse(Estimator.A,"New estimator A matrix:\n");
printSparse(Estimator.B,"New estimator B matrix:\n");
printSparse(Estimator.C,"New estimator C matrix:\n");
printSparse(Estimator.D,"New estimator D matrix:\n");

/* Release observer data lock */
pthread_rwlock_unlock(&EstimatorLock);

/* Free the temporary arrays that we used */

if (no>0)
  {
    free(Ac);
    free(Bc);
```

```
      free(Cc);
    }
  free(Dc);

  dsmp_SendMesg(MSG_ackMsg, (void *) &thisMesg, SAT_CON);

  return(TRUE);
}

int setGyroCalibData(byte whoFrom, byte *data) {
  double gyroGains[2];
  static byte thisMesg=MSG_setGyroCalibData;

  memcpy(&gyroGains,data,2*sizeof(double));

  printf("Received new adaptive gyro calib gains: %lf\t%lf\n",
     gyroGains[0],gyroGains[1]);
  fflush(stdout);

  pthread_rwlock_wrlock(&EstimatorLock);

  Estimator.adaptGyroGains[0] = gyroGains[0];
  Estimator.adaptGyroGains[1] = gyroGains[1];

  pthread_rwlock_unlock(&EstimatorLock);

  dsmp_SendMesg(MSG_ackMsg, (void *) &thisMesg, SAT_CON);

  return(TRUE);
}

int calibrateSensors(byte whoFrom, byte *data) {
  static byte thisMesg=MSG_calibrateSensors;

  printf("Received calibrate sensor request: %u\n",TSrunMode);
  fflush(stdout);

  dsmp_SendMesg(MSG_ackMsg, (void *) &thisMesg, SAT_CON);

  return(TRUE);
}
```

```
#include "../include/TS_Packet.h"

void dsmp_PackMesg(byte mesgNum, void *mesgData, byte *mesgBuf);
void dsmp_RecvTimeout(int signo); int dsmp_PacketAvail(int fd, int
usec);

byte mesg[MAXLINE]; HandlerFuncPtr dsmpMesgHandlers[256]={};
unsigned short dsmpMesgSizes[256]={}; unsigned char
dsmpMesgFlags[256]={}; struct sockaddr_in dsmpClients[MAXCLI];
byte dsmpCliReg[MAXCLI]={FALSE};

int dsmpSocket; struct sockaddr_in dsmpAddr;

unsigned char dsmpIdentity;

int numSent=0, numRecv=0;

int dsmpTimeout = 0;  // Timeout in microsecs

void dsmp_RecvTimeout(int signo) {

  printf("error num is %d\n",errno);
  return;
}

int dsmp_Init(unsigned char ident, int portNum) {
  int ret;
  u_char loop=FALSE;

  printf("Initializing dsmp interface\n"); fflush(stdout);

  dsmpIdentity = ident;
  dsmpSocket = socket(AF_INET, SOCK_DGRAM, 0);

  if (dsmpSocket < 0) {
    printf("socket error\n");
    exit(1);
  }

  bzero(&dsmpAddr, sizeof(dsmpAddr));
  dsmpAddr.sin_family      = AF_INET;
  dsmpAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  dsmpAddr.sin_port        = htons(portNum);

  ret = bind(dsmpSocket, (SA *) &dsmpAddr, sizeof(dsmpAddr));
  if (ret < 0) {
    printf("bind error\n");
    exit(1);
  }

  // Disable multicast loopback by default

   ret = setsockopt(dsmpSocket, IPPROTO_IP, IP_MULTICAST_LOOP,
    &loop, sizeof(loop));

   return(dsmpSocket);
}

void dsmp_RegisterClient(byte cliNum, char *cliIP) {
  struct sockaddr_in *cliaddr;
  int ret;

  printf("Registering client #%u\n",cliNum); fflush(stdout);

  cliaddr = dsmpClients+cliNum;
  bzero(cliaddr, sizeof(struct sockaddr_in));
  dsmpClients[cliNum].sin_family = AF_INET;
  dsmpClients[cliNum].sin_port = htons(dsmpSERV_PORT);
  ret = inet_pton(AF_INET, cliIP, &dsmpClients[cliNum].sin_addr);

  if (ret < 0) {
    printf("inet_pton error\n");
    exit(1);
  }

  dsmpCliReg[cliNum]=TRUE;

}

void dsmp_AcceptMcast(char *addrStr) // Not working. . .??? {

  int ret;
  struct ip_mreq mreq;
  struct sockaddr_in mcastAddr;
  byte nhops = 10;

  printf("Joining multicast group\n");fflush(stdout);

  ret = inet_pton(AF_INET, addrStr, &mreq.imr_multiaddr);

  if (ret < 0) {
    printf("inet_pton error\n");
    exit(1);
  }

  memcpy(&mreq.imr_interface,&dsmpAddr.sin_addr,sizeof(struct in_addr));
  mreq.imr_interface.s_addr = htonl(INADDR_ANY);

  ret = setsockopt(dsmpSocket, IPPROTO_IP,IP_ADD_MEMBERSHIP,
  &mreq,sizeof(mreq));

  if (ret < 0) {
    printf("setsockopt error:");
    printf("errno is %d\n",errno);
    perror(NULL);
    exit(1);
  }
}

void dsmp_RegisterMesg(unsigned char msgNum, unsigned char
msgFlags, HandlerFuncPtr msgHandler, short msgSize) {

  printf("Registering message #%u \n",msgNum); fflush(stdout);

  dsmpMesgHandlers[msgNum]=msgHandler;
  dsmpMesgSizes[msgNum] = msgSize;
  dsmpMesgFlags[msgNum] = msgFlags;
}

int dsmp_PacketAvail(int fd, int usec) {
  fd_set rset;
  struct timeval tv;

  FD_ZERO(&rset);
  FD_SET(fd,&rset);
  tv.tv_sec = 0;
  tv.tv_usec = usec;

  return( select(fd + 1, &rset, NULL, NULL, &tv) );
}

#define DSMP_RECVTOUT 0

int dsmp_RecvAll(void) {

  while (dsmp_RecvMesg() != DSMP_RECVTOUT);
}


int dsmp_RecvMesg(void) {
  int i,n,ret;
  byte mesgNum,mesgFlags,mesgFrom;
  unsigned short mesgSize, expectedSize;
  HandlerFuncPtr handler;
  struct sockaddr_in  pcliaddr;
  char cliName[80];
  socklen_t len = sizeof(pcliaddr);

  if (!dsmp_PacketAvail(dsmpSocket, dsmpTimeout)) {
    return(0);
  } else {
    n = recvfrom(dsmpSocket, mesg, MAXLINE, 0, (SA *) &pcliaddr,
     &len);
  }

  if (n<0) {
    printf("Receive error:%d\n",errno);
    return(-1);
  }

//  for (i = 0; i < n; i++) printf("%d\t%x\n",i,mesg[i]);
//  printf("\n");

  mesgNum = mesg[0];
  mesgFlags = mesg[1];
  mesgSize = 256*mesg[2]+mesg[3];
  mesgFrom = mesg[4];
  // printf("Message number %d is %d\n",numRecv,mesgNum);
  //  fflush(stdout);

  if (dsmpCliReg[mesgFrom]==FALSE) {
    inet_ntop(AF_INET,(void *) &pcliaddr.sin_addr, cliName, 80);
        printf("Incoming client connection from %s \n",cliName);
        fflush(stdout);
        dsmp_RegisterClient(mesgFrom,cliName);
  }

//  Note: if we get an incoming message from an already
// registered client at a new address, we will ignore
// the new address!

  handler = dsmpMesgHandlers[mesgNum];
  if (handler == NULL) {
    printf("Error -- no handler for message %d!\n",mesgNum);
        fflush(stdout);
//    exit(1);
    return(-1);
  }

  expectedSize = dsmpMesgSizes[mesgNum];

  if ((mesgSize != expectedSize) && (expectedSize != TS_VARSIZE)) {
    printf("Error -- wrong message size:  expected %d received %d",
        dsmpMesgSizes[mesgNum],mesgSize);
    return(-1);
  }

  // If we get here everything must check out, so invoke the handler and return

  return(handler(mesgFrom,mesg+5));
}

void dsmp_PackMesg(byte mesgNum, void *mesgData, byte *mesgBuf) {
  int i;
  byte *buf, *dat;
  int mesgSize = dsmpMesgSizes[mesgNum];

  mesgBuf[0] = mesgNum;
  mesgBuf[1] = dsmpMesgFlags[mesgNum];
  mesgBuf[2] = mesgSize / 256;
  mesgBuf[3] = mesgSize % 256;
  mesgBuf[4] = dsmpIdentity;

  dat = (byte *) mesgData;
  buf = mesgBuf+5;
  for (i = 0; i < mesgSize; i++,dat++,buf++) *buf = *dat;
}

int dsmp_SendMesg(byte mesgNum, void *mesgData, byte mesgClient) {
 struct sockaddr_in *cliaddr;
 int i,ret;

 cliaddr = dsmpClients+mesgClient;

 dsmp_PackMesg(mesgNum, mesgData, mesg);

 ret =sendto(dsmpSocket, mesg, dsmpMesgSizes[mesgNum]+5,0,
         (SA *) &dsmpClients[mesgClient],
         sizeof(dsmpClients[mesgClient]));

//  for (i = 0; i < dsmpMesgSizes[mesgNum]+5; i++)
//  printf("%d\t%x\n",i,mesg[i]);
//  printf("\n");

  if (ret < 0) {
    printf("SendMesg error\n");
    printf("errno=%d\n",errno);
    return(1);
    //    exit(1);
  }

  return(0);

  //  numSent++;
  //  printf("Sent message #%d\n",numSent); fflush(stdout);
}

#include "../include/TS_PromIO.h" #include
"../include/TS_Actuator.h"
```

```
#define FLIGHTVERSION

DSCB       dscb; ERRPARAMS  errorParams; DSCADSCAN  dscadscan;
DSCSAMPLE  *samples;

extern struct timeval start_time; extern pthread_rwlock_t
SensorLock;

void InitPrometheus(void) {
  DSCCB          dsccb; // structure containing board settings
  DSCADSETTINGS  dscadsettings;
    // structure containing A/D conversion settings

  BYTE result;

#ifdef FLIGHTVERSION

  /***** Initialize DSCUD library *****/

  if( dscInit( DSC_VERSION ) != DE_NONE )
    {
      dscGetLastError(&errorParams);
      fprintf( stderr, "%s %s\n", ERROR_PREFIX,
        errorParams.errstring );
      return;
    }
#endif

  /***** Initialize Prometheus board *****/

  dsccb.boardtype = DSC_PROM;
  dsccb.base_address = BASEADD;
  dsccb.int_level = 5;
  dsccb.dma_level = 3;
  dsccb.clock_freq = 10000000L;

#ifdef FLIGHTVERSION

  if(dscInitBoard(DSC_PROM, &dsccb, &dscb)!= DE_NONE)
    {
      dscGetLastError(&errorParams);
      fprintf(stderr, "%s %s\n", ERROR_PREFIX,
        errorParams.errstring);
      return;
    }
#endif

  /***** Initialize AD settings *****/

  dscadsettings.range = RANGE_5;
  dscadsettings.polarity = UNIPOLAR;
  dscadsettings.gain = GAIN_4;
  dscadsettings.load_cal = (BYTE)TRUE;
  dscadsettings.current_channel = 0;

  dscadsettings.load_cal = 0;

#ifdef FLIGHTVERSION

  if( ( result = dscADSetSettings( dscb, &dscadsettings ) )
     != DE_NONE )
    {
      dscGetLastError(&errorParams);
      fprintf(stderr, "%s %s\n", ERROR_PREFIX,
        errorParams.errstring);
      return;
    }

  /***** Set D/A polarity *****/
  dscDASetPolarity(dscb, UNIPOLAR);

#endif

  /***** Initialize scan settings *****/
  dscadscan.low_channel  = 0;
  dscadscan.high_channel = 7;
  dscadscan.gain      = GAIN_1;
  dscadscan.gain      = dscadsettings.gain;

  samples = (DSCSAMPLE*) malloc(sizeof(DSCSAMPLE)*
    (dscadscan.high_channel - dscadscan.low_channel+1));

  dscadscan.sample_values = samples;
  return;
}

int ReadSensors(SensorReadings_t *sensors) {
  struct timeval cur_time;
  double nowtime;
  BYTE result;

  /* Call for a new A/D Scan */

#ifdef FLIGHTVERSION

  if( (result = dscADScan( dscb, & dscadscan, samples) )!= DE_NONE )
    {
      fprintf(stderr, "%s%s\n", ERROR_PREFIX,
        dscGetErrorString(result) );
      free(samples);
      return(0);
    }
#endif

  gettimeofday(&cur_time, NULL);
  nowtime = ((double) (cur_time.tv_sec - start_time.tv_sec))
    +  (((double) (cur_time.tv_usec - start_time.tv_usec))/1.0E6);

  /* Take write lock to update sensor readings */
  pthread_rwlock_wrlock(&SensorLock);

  // printf("Loading sensor data\n");fflush(stdout);
  /* Load the sensor structure with current data */
  sensors->TimeStamp = nowtime;

  sensors->CSS1Volts = (double) (((short) dscadscan.sample_values[7]
     + 32768)/65536.0 * 5.0);
  sensors->CSS2Volts = (double) (((short) dscadscan.sample_values[6]
     + 32768)/65536.0 * 5.0);
  sensors->CSS3Volts = (double) (((short) dscadscan.sample_values[5]
     + 32768)/65536.0 * 5.0);
  sensors->CSS4Volts = (double) (((short) dscadscan.sample_values[4]
     + 32768)/65536.0 * 5.0);

  sensors->GyroVolts =  (double) (((short) dscadscan.sample_values[0]
      + 32768)/65536.0 * 5.0);

    sensors->MagXVolts = (double) (((short) dscadscan.sample_values[1]
      + 32768)/65536.0 * 5.0);
    sensors->MagYVolts = (double) (((short) dscadscan.sample_values[2]
      + 32768)/65536.0 * 5.0);
    sensors->MagZVolts = (double) (((short) dscadscan.sample_values[3]
      + 32768)/65536.0 * 5.0);

    /* Release write lock */
    pthread_rwlock_unlock(&SensorLock);

    // printf("Done!\n");fflush(stdout);

    return(1);
  }

/*  CommandMotor takes the commanded voltages for each fan
    in the FanVoltage structure, converts them to counts,
    and calls the D/A converter to power the fans.  */

void commandMotor(double *v) {
    int pos_counts, neg_counts;
    //  float ref_voltage = 12.0;
    int pos_channel, neg_channel;
    BYTE result;

    /* Must modify these once board configuration is known */
    pos_channel = 3;
    neg_channel = 1;
    /*******************************************************/

    pos_counts = (int)(v[0]/VMAXFAN*4095.0);
    neg_counts = (int)(v[1]/VMAXFAN*4095.0);

#ifdef FLIGHTVERSION
    if( ( result = dscDAConvert(dscb, pos_channel, pos_counts ) )
      != DE_NONE )
    {
        dscGetLastError(&errorParams);
        fprintf(stderr, "%s %s\n", ERROR_PREFIX,
            errorParams.errstring);
        return;
    }

    if( ( result = dscDAConvert(dscb, neg_channel, neg_counts ) )
         != DE_NONE )
    {
        dscGetLastError(&errorParams);
        fprintf(stderr, "%s %s\n", ERROR_PREFIX,
            errorParams.errstring);
        return;
    }
#endif

    return;
}
```

Appendix B

TableSat GUI Code

```
function varargout = TSat2(varargin)
% TSAT2 M-file for TSat2.fig
%      TSAT2, by itself, creates a new TSAT2 or raises the
%      existing singleton*.
%
%      H = TSAT2 returns the handle to a new TSAT2 or
%      the handle to the existing singleton*.
%
%      TSAT2('CALLBACK',hObject,eventData,handles,...)
%      calls the local function named CALLBACK in TSAT2.M with
%      the given input arguments.
%
%      TSAT2('Property','Value',...) creates a new TSAT2 or
%      raises the existing singleton*.  Starting from the left,
%      property value pairs are applied to the GUI before
%      TSat2_OpeningFunction gets called.  An unrecognized
%      property name or invalid value makes property application
%      stop.  All inputs are passed to TSat2_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.
%      Choose "GUI allows only one instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help TSat2

% Last Modified by GUIDE v2.5 22-Jun-2004 13:04:53

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1; gui_State = struct('gui_Name',mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @TSat2_OpeningFcn, ...
                   'gui_OutputFcn',  @TSat2_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin & isstr(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 TSat2 is made visible.
function TSat2_OpeningFcn(hObject, eventdata, handles,
varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to betext defined in a future version
% of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to TSat2 (see VARARGIN)

% Choose default command line output for TSat2
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

 %UIWAIT makes TSat2 wait for user response (see UIRESUME)
 %uiwait(handles.figure1);

% --- Outputs from this function are returned to the command line.
function varargout = TSat2_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to betext defined in a future version
% of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in FrictionComp.
function FrictionComp_Callback(hObject, eventdata, handles)

if get(hObject,'Value') == 1
    set(handles.NumFricPts,'Enable','On');
    set(handles.wValues,'Enable','On');
    set(handles.FricCompValues,'Enable','On');
else
    set(handles.NumFricPts,'Enable','Off');
    set(handles.wValues,'Enable','Off');
    set(handles.FricCompValues,'Enable','Off');
end

% --- Executes during object creation, after setting all properties.
function NumFricPts_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function NumFricPts_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1])
user_entry=str2num(get(hObject,'String')); if isempty(user_entry)
| user_entry<0 | round(user_entry) ~= user_entry
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('Please Enter a Positive Integer',
    'Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function wValues_CreateFcn(hObject, eventdata, handles)

if ispc
```

```
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function wValues_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1])
user_entry=str2num(get(hObject,'String')); if isempty(user_entry)
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('Please Enter a Value or Range','Invalid Entry','modal')
end

if length(user_entry)~= str2num(get(handles.NumFricPts,'String'))
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('Number of Angular Velocity Values must equal
    Number of Compensation Points','Invalid Entry','modal')

end

% --- Executes during object creation, after setting all properties.
function FricCompValues_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function FricCompValues_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1])
user_entry=str2num(get(hObject,'String')); if isempty(user_entry)
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('Please Enter a Value or Range',
    'Invalid Entry','modal')
end

if length(user_entry)~= str2num(get(handles.NumFricPts,'String'))
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('Number of Friction Compensation Values must equal
    Number of Compensation Points','Invalid Entry','modal')

end

% --- Executes during object creation, after setting all properties.
function ActType_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end


% --- Executes on selection change in ActType.
function ActType_Callback(hObject, eventdata, handles)

if get(hObject,'Value') == 3
    set(handles.DeadZone,'Enable','On');
else
    set(handles.DeadZone,'Enable','Off');
end

% --- Executes during object creation, after setting all properties.
function DeadZone_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end


function DeadZone_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1])
user_entry=str2num(get(hObject,'String')); if isempty(user_entry)
| user_entry<0 | user_entry>1
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('Value must be positive and less than 1',
    'Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function ContRate_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function ContRate_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1])
user_entry=str2num(get(hObject,'String')); if isempty(user_entry)
| user_entry<0
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('Value must be positive','Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function ActRate_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function ActRate_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1])
user_entry=str2num(get(hObject,'String')); if isempty(user_entry)
| user_entry<0
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('Value must be positive','Invalid Entry','modal')
end
```

```
if user_entry/str2num(get(handles.ContRate,'String'))<1
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('Actuator Rate must be greater than or equal
    to Controller Rate', 'Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function EstRate_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function EstRate_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1])
user_entry=str2num(get(hObject,'String')); if isempty(user_entry)
| user_entry<0
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('Value must be positive','Invalid Entry','modal')
end

% --- Executes on button press in ControllerOn.
function ControllerOn_Callback(hObject, eventdata, handles)

if get(hObject,'Value') == 0
    set(handles.nc,'Enable','Off');
    set(handles.pc,'Enable','Off');
    set(handles.A,'Enable','Off');
    set(handles.B1,'Enable','Off');
    set(handles.B2,'Enable','Off');
    set(handles.C,'Enable','Off');
    set(handles.D1,'Enable','Off');
    set(handles.D2,'Enable','Off');
    set(handles.FrictionComp,'Enable','On');
    set(handles.ActType,'Enable','On');
    set(handles.ContRate,'Enable','On');
    set(handles.ActRate,'Enable','On');
    set(handles.EstRate,'Enable','On');
    set(handles.ContDef,'Enable','Off');
    set(handles.ContSim,'Enable','Off');
    set(handles.ContMat,'Enable','Off');
    set(handles.SimFile,'Enable','Off');
    set(handles.MatFile,'Enable','Off');
    set(handles.SimBrowse,'Enable','Off');
    set(handles.MatBrowse,'Enable','Off');
    set(handles.OmegaD,'Enable','Off');
    set(handles.ThetaD,'Enable','Off');
    set(handles.TargetS,'Enable','Off');
    set(handles.SineWave,'Enable','Off');
    set(handles.Amp,'Enable','Off');
    set(handles.Freq,'Enable','Off');
else
    set(handles.nc,'Enable','On');
    set(handles.pc,'Enable','On');
    set(handles.A,'Enable','On');
    set(handles.B1,'Enable','On');
    set(handles.B2,'Enable','On');
    set(handles.C,'Enable','On');
    set(handles.D1,'Enable','On');
    set(handles.D2,'Enable','On');
    set(handles.FrictionComp,'Enable','On');
    set(handles.ActType,'Enable','On');
    set(handles.ContRate,'Enable','On');
    set(handles.ActRate,'Enable','On');
    set(handles.EstRate,'Enable','On');
    set(handles.ContDef,'Enable','On');
    set(handles.ContSim,'Enable','On');
    set(handles.ContMat,'Enable','On');
    set(handles.SimFile,'Enable','On');
    set(handles.MatFile,'Enable','On');
    set(handles.SimBrowse,'Enable','On');
    set(handles.MatBrowse,'Enable','On');
    set(handles.OmegaD,'Enable','On');
    set(handles.ThetaD,'Enable','On');
    set(handles.TargetS,'Enable','On');
    set(handles.SineWave,'Enable','On');
end

% --- Executes during object creation, after setting all properties.
function nc_CreateFcn(hObject, eventdata, handles)
```

```
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function nc_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1])
user_entry=str2num(get(hObject,'String')); if isempty(user_entry)
| user_entry<0
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('Value must be positive or 0',
    'Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function A_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function A_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1]);

user_entry=str2num(get(hObject,'String'));
[r,c]=size(user_entry);
nc = str2num(get(handles.nc,'String'));

if nc>0
    if (~isempty(user_entry) & (nc>0 & r~=c)) |
    (~isempty(user_entry) & (nc>0 & r~=nc))
        set(hObject,'BackgroundColor',[1 1 0]);
        errordlg('A-matrix must be ncXnc',
        'Invalid Entry','modal')
    end

elseif nc == 0
    if ~isempty(user_entry)
        set(hObject,'BackgroundColor',[1 1 0]);
        errordlg('If nc = 0, A-matrix must be [ ]',
        'Invalid Entry','modal')
    end
end

% --- Executes during object creation, after setting all properties.
function B1_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function B1_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1]);

user_entry=str2num(get(hObject,'String'));
[r,c]=size(user_entry);
nc = str2num(get(handles.nc,'String'));

if ~isempty(user_entry) & c~=5
    set(hObject,'BackgroundColor',[1 1 0]);
    errordlg('B1-matrix must have 5 columns or be [ ])',
    'Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function B2_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
```

```
end

function B2_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1]);

user_entry=str2num(get(hObject,'String'));
[r,c]=size(user_entry);
nc = str2num(get(handles.nc,'String'));

if ~isempty(user_entry) & c~=2
    set(hObject,'BackgroundColor',[1 1 0]);
    errordlg('B2-matrix have 2 columns or be [ ]',
    'Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function C_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function C_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1]);

user_entry=str2num(get(hObject,'String'));
[r,c]=size(user_entry);
nc = str2num(get(handles.nc,'String')); pc =
str2num(get(handles.pc,'String'));

if nc>0
    if (~isempty(user_entry) & c~=nc) |
    (~isempty(user_entry) & r~=pc)
        set(hObject,'BackgroundColor',[1 1 0]);
        errordlg('C-matrix must be pc X nc or [ ]',
        'Invalid Entry','modal')
    end
```

```
else
    if ~isempty(user_entry)
        set(hObject,'BackgroundColor',[1 1 0]);
        errordlg('If nc = 0, C-matrix must be [ ]',
        'Invalid Entry','modal')
    end

    if sum(user_entry) ~= 0
        set(hObject,'BackgroundColor',[1 1 0]);
        errordlg('C-matrix must be zeros
        (there are no internal controller states)',
        'Invalid Entry','modal')
    end
end

% --- Executes during object creation, after setting all properties.
function D1_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function D1_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1]);

user_entry=str2num(get(hObject,'String'));
[r,c]=size(user_entry);
nc = str2num(get(handles.nc,'String')); pc =
str2num(get(handles.pc,'String'));

if isempty(user_entry) | r~=pc | c~=5
    set(hObject,'BackgroundColor',[1 1 0]);
    errordlg('D1-matrix must a pc x 5 matrix',
    'Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function D2_CreateFcn(hObject, eventdata, handles)
```

```
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function D2_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1]);

user_entry=str2num(get(hObject,'String'));
[r,c]=size(user_entry);
nc = str2num(get(handles.nc,'String')); pc =
str2num(get(handles.pc,'String'));

if isempty(user_entry) | r~=pc | c~=2
    set(hObject,'BackgroundColor',[1 1 0]);
    errordlg('D2-matrix must a pc x 2 matrix',
    'Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function pc_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function pc_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1])
user_entry=str2num(get(hObject,'String')); if isempty(user_entry)
| user_entry<=0 | user_entry>2
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('Value must be 1 or 2','Invalid Entry','modal')
end
```

```
% --- Executes on button press in TsatParams.
function TsatParams_Callback(hObject, eventdata, handles)

% Initialize TableSat interface
TSsock=tsat_init(9877,'192.168.1.110');

%************* Friction Compensation **********************
fr=get(handles.FrictionComp,'Value');
n=str2num(get(handles.NumFricPts,'String'));
w=str2num(get(handles.wValues,'String'));
f=str2num(get(handles.FricCompValues,'String'));

data = [n, w, f];

if fr == 1
    %Friction Compensation is on
    tsat_send_msg(8,{uint8(1)},TSsock);
    tsat_send_msg(21,{data},TSsock);
    fprintf('sending friction comp on message\n')
    fprintf('sending friction comp message\n')
else
    %Friction Compensation is off
    tsat_send_msg(8,{uint8(0)},TSsock);
    fprintf('sending friction comp off message\n')
end

%********* Actuator Select ********************************
act=get(handles.ActType,'Value'); act = act - 1;
dz=str2num(get(handles.DeadZone,'String'));

if act == 2
    data = act+dz;
else
    data = act;
end

tsat_send_msg(10,{data},TSsock); fprintf('sending actuator type
message\n')

%*********** Sample Rates *********************************
cr = str2num(get(handles.ContRate,'String')); ar =
str2num(get(handles.ActRate,'String')); er =
str2num(get(handles.EstRate,'String'));

data = [cr, er];

tsat_send_msg(33,{data},TSsock); fprintf('sending sample rates
message\n')

%******** Controller Selection ***********************
c = get(handles.ControllerOn,'Value');

cd = get(handles.ContDef,'Value'); cs =
get(handles.ContSim,'Value'); cm = get(handles.ContMat,'Value');

if c
    %tsat_send_msg(6,{uint8(1)},TSsock);
    close_system('tsat_OL.mdl')
    open_system('tsat_CL.mdl')
    fprintf('sending controller on message\n')
else
    tsat_send_msg(6,{uint8(0)},TSsock);
    close_system('tsat_CL.mdl')
    open_system('tsat_OL.mdl')
    fprintf('sending controller off message\n')
end

if c & cd
    nc = str2num(get(handles.nc,'String'));
    pc = str2num(get(handles.pc,'String'));
    A = str2num(get(handles.A,'String'));
    B1 = str2num(get(handles.B1,'String'));
    B2 = str2num(get(handles.B2,'String'));
    C = str2num(get(handles.C,'String'));
    D1 = str2num(get(handles.D1,'String'));
    D2 = str2num(get(handles.D2,'String'));

    TS_Con.dims = [nc, 5, 2, pc, 5, 2];
    TS_Con.A = A;
    TS_Con.B1 = B1;
    TS_Con.B2 = B2;
    TS_Con.C = C;
    TS_Con.D1 = D1;
    TS_Con.D2 = D2;
    tsat_send_msg(15,TS_Con,TSsock);
    fprintf('sending controller message\n')

elseif c & cs

    file = get(handles.SimFile,'String');
    %Note: the model is assumed to have 7 inputs,
    %the first 2 are xd,
    %the last 5 are the state variables, x.
    [pathstr,name,ext,versn] = fileparts(file);
    open_system(file);
    save_system(name, 'temp')
    close_system('temp')
    [A,B,C,D] = linmod('temp');
    [r,c] = size(A);
    nc = r;
    [r,c] = size(C);
    pc = r;
    TS_Con.dims = [nc, 5, 2, pc, 5, 2];

    if ~isempty(A)
        TS_Con.A = A;
    else
        TS_Con.A = [];
    end

    if ~isempty(B)
        TS_Con.B1 = B(:,3:7);
        TS_Con.B2 = B(:,1:2);
    else
        TS_Con.B1 = [];
        TS_Con.B2 = [];
    end

    if ~isempty(C)
        TS_Con.C = C;
    else
        TS_Con.C = [];
    end

    if ~isempty(D)
```

```
        TS_Con.D1 = D(:,3:7);
        TS_Con.D2 = D(:,1:2);
    else
        TS_Con.D1 = [];
        TS_Con.D2 = [];
    end

    tsat_send_msg(15,TS_Con,TSsock);
    fprintf('sending controller message\n')

elseif c & cm
    file = get(handles.MatFile,'String');
    load(file);

    tsat_send_msg(15,TS_Con,TSsock);
    fprintf('sending controller message\n')
end

%******** Estimator Selection ***********************
ed = get(handles.EstDef,'Value'); em =
get(handles.MatLoadE,'Value');

if ed
    no = str2num(get(handles.no,'String'));
    Ae = str2num(get(handles.Ae,'String'));
    Be = str2num(get(handles.Be,'String'));
    Ce = str2num(get(handles.Ce,'String'));
    De = str2num(get(handles.De,'String'));

    TS_Est.dims = [no, 5, 5, 5];
    TS_Est.Ae = Ae;
    TS_Est.Be = Be;
    TS_Est.Ce = Ce;
    TS_Est.De = De;

    tsat_send_msg(27,TS_Est,TSsock);
    fprintf('sending estimator message\n')

elseif em
    file = get(handles.MatFileE,'String');
    load(file);
    tsat_send_msg(27,TS_Est,TSsock);
    fprintf('sending estimator message\n')
end

%************ Target Selection *********************
ts = get(handles.TargetS,'Value'); sw =
get(handles.SineWave,'Value');

if ts
    th = str2num(get(handles.ThetaD,'String'));
    w = str2num(get(handles.OmegaD,'String'));

    data=[0 th w];

    tsat_send_msg(24,{data},TSsock);
    fprintf('sending target message\n')

elseif sw
    amp = str2num(get(handles.Amp,'String'));
    f = str2num(get(handles.Freq,'String'));

    data = [1 amp f];

    tsat_send_msg(24,{data},TSsock);
    fprintf('sending target message\n')
end

clear pnet

% --- Executes on button press in ContDef.
function ContDef_Callback(hObject, eventdata, handles)

if get(hObject,'Value') == 0
    set(handles.nc,'Enable','Off');
    set(handles.pc,'Enable','Off');
    set(handles.A,'Enable','Off');
    set(handles.B1,'Enable','Off');
    set(handles.B2,'Enable','Off');
    set(handles.C,'Enable','Off');
    set(handles.D1,'Enable','Off');
    set(handles.D2,'Enable','Off');
else
```

```
    set(handles.nc,'Enable','On');
    set(handles.pc,'Enable','On');
    set(handles.A,'Enable','On');
    set(handles.B1,'Enable','On');
    set(handles.B2,'Enable','On');
    set(handles.C,'Enable','On');
    set(handles.D1,'Enable','On');
    set(handles.D2,'Enable','On');
    set(handles.ContSim,'Value',0)
    set(handles.ContMat,'Value',0)
    set(handles.MatFile,'Enable','Off')
    set(handles.MatBrowse,'Enable','Off')
    set(handles.SimFile,'Enable','Off')
    set(handles.SimBrowse,'Enable','Off')
end

% --- Executes on button press in ContSim.
function ContSim_Callback(hObject, eventdata, handles)

if get(hObject,'Value') == 0
    set(handles.SimFile,'Enable','Off')
    set(handles.SimBrowse,'Enable','Off')
else
    set(handles.SimFile,'Enable','On')
    set(handles.SimBrowse,'Enable','On')
    set(handles.ContDef,'Value',0)
    set(handles.ContMat,'Value',0)
    set(handles.MatFile,'Enable','Off')
    set(handles.MatBrowse,'Enable','Off')
    set(handles.nc,'Enable','Off');
    set(handles.pc,'Enable','Off');
    set(handles.A,'Enable','Off');
    set(handles.B1,'Enable','Off');
    set(handles.B2,'Enable','Off');
    set(handles.C,'Enable','Off');
    set(handles.D1,'Enable','Off');
    set(handles.D2,'Enable','Off');
end

% --- Executes on button press in ContMat.
function ContMat_Callback(hObject, eventdata, handles)
```

```
if get(hObject,'Value') == 0
    set(handles.MatFile,'Enable','Off')
    set(handles.MatBrowse,'Enable','Off')
else
    set(handles.MatFile,'Enable','On')
    set(handles.MatBrowse,'Enable','On')
    set(handles.ContDef,'Value',0)
    set(handles.ContSim,'Value',0)
    set(handles.SimFile,'Enable','Off')
    set(handles.SimBrowse,'Enable','Off')
    set(handles.nc,'Enable','Off');
    set(handles.pc,'Enable','Off');
    set(handles.A,'Enable','Off');
    set(handles.B1,'Enable','Off');
    set(handles.B2,'Enable','Off');
    set(handles.C,'Enable','Off');
    set(handles.D1,'Enable','Off');
    set(handles.D2,'Enable','Off');
end

% --- Executes during object creation, after setting all properties.
function SimFile_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function SimFile_Callback(hObject, eventdata, handles)

set(h,'BackgroundColor',[1 1 1]) if isempty(get(hObject,'String'))
    % do nothing
elseif exist(get(hObject,'String'))==0
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('File Does Not Exist As Entered',
    'Invalid Entry','modal')
else
    fokay=fileokay(get(hObject,'String'));
    if fokay==0
        errordlg('Please load a Simulink Model',
        'Invalid Entry','modal')
        set(h,'BackgroundColor',[1 1 0])
    end
end

function MatFile_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function MatFile_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1]) if
isempty(get(hObject,'String'))
    % do nothing
elseif exist(get(hObject,'String'))==0
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('File Does Not Exist As Entered',
    'Invalid Entry','modal')
else
    fokay=fileokay(get(hObject,'String'));
    if fokay==0
        errordlg('Please load a .mat file',
        'Invalid Entry','modal')
        set(h,'BackgroundColor',[1 1 0])
    end
end

% --- Executes on button press in SimBrowse.
function SimBrowse_Callback(hObject, eventdata, handles)

[filename, pathname] = uigetfile( ...
       {'*.mdl', 'Simulink Model'}, ...
        'Select a Simulink Model');

set(handles.SimFile,'BackgroundColor',[1 1 1]) if filename==0
    % do nothing
else
```

```
    file=fullfile(pathname,filename);
    set(handles.SimFile,'String',file)
end

% --- Executes on button press in MatBrowse.
function MatBrowse_Callback(hObject, eventdata, handles)

[filename, pathname] = uigetfile( ...
       {'*.mat', 'Simulink Model'}, ...
        'Select a MATLAB File');

set(handles.MatFile,'BackgroundColor',[1 1 1]) if filename==0
    % do nothing
else
    file=fullfile(pathname,filename);
    set(handles.MatFile,'String',file)
    fokay=fileokay(file);
    if fokay==0
        set(handles.MatFile,'BackgroundColor',[1 1 0])
        errordlg('File is Not in Correct Form',
        'Invalid Entry','modal')
    end
end

% ----------------------------------------------------------------------
% checks if file is in readable form
function fokay=fileokay(filename)

try
    cont=load(filename);
    fokay=1;
    if isempty(cont)
        fokay=0;
    end
catch
    fokay=0;
end

% --- Executes during object creation, after setting all properties.
function no_CreateFcn(hObject, eventdata, handles)

if ispc
```

```
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function no_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1])
user_entry=str2num(get(hObject,'String')); if isempty(user_entry)
| user_entry<0
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('Value must be positive or 0','Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function Be_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function Be_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1]);

user_entry=str2num(get(hObject,'String')); [r,c]=size(user_entry);
no = str2num(get(handles.no,'String'));

if (~isempty(user_entry) & c~=5) | (~isempty(user_entry) & r~=no)
    set(hObject,'BackgroundColor',[1 1 0]);
    errordlg('Be-matrix must have dimension no X 5 or be [ ])',
    'Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function Ae_CreateFcn(hObject, eventdata, handles)

if ispc
```

```
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function Ae_Callback(hObject, eventdata, handles)
set(hObject,'BackgroundColor',[1 1 1]);

user_entry=str2num(get(hObject,'String')); [r,c]=size(user_entry);
no = str2num(get(handles.no,'String'));

if no>0
    if (~isempty(user_entry) & (no>0 & r~=c)) |
    (~isempty(user_entry) & (no>0 & r~=no))
        set(hObject,'BackgroundColor',[1 1 0]);
        errordlg('Ae-matrix must be noXno','Invalid Entry','modal')
    end

elseif no == 0
    if ~isempty(user_entry)
        set(hObject,'BackgroundColor',[1 1 0]);
        errordlg('If no = 0, Ae-matrix must be [ ]',
        'Invalid Entry','modal')
    end
end

% --- Executes during object creation, after setting all properties.
function Ce_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function Ce_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1]);

user_entry=str2num(get(hObject,'String'));
```

```
[r,c]=size(user_entry);
no = str2num(get(handles.no,'String'));

if (~isempty(user_entry) & c~=no) | (~isempty(user_entry) & r~=5)
    set(hObject,'BackgroundColor',[1 1 0]);
    errordlg('Ce-matrix must have dimension 5 X no or be [ ])',
    'Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function De_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function De_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1]);

user_entry=str2num(get(hObject,'String'));
[r,c]=size(user_entry);

if isempty(user_entry) | c~=5 | r~=5
    set(hObject,'BackgroundColor',[1 1 0]);
    errordlg('De-matrix must have dimension 5x5',
    'Invalid Entry','modal')
end

% --- Executes on button press in EstDef.
function EstDef_Callback(hObject, eventdata, handles)

if get(hObject,'Value') == 0
    set(handles.no,'Enable','Off');
    set(handles.Ae,'Enable','Off');
    set(handles.Be,'Enable','Off');
    set(handles.Ce,'Enable','Off');
    set(handles.De,'Enable','Off');
else
```

```
    set(handles.no,'Enable','On');
    set(handles.Ae,'Enable','On');
    set(handles.Be,'Enable','On');
    set(handles.Ce,'Enable','On');
    set(handles.De,'Enable','On');
    set(handles.MatLoadE,'Value',0)
    set(handles.MatFileE,'Enable','Off')
    set(handles.MatBrowseE,'Enable','Off')
end

% --- Executes on button press in MatLoadE.
function MatLoadE_Callback(hObject, eventdata, handles)

if get(hObject,'Value') == 0
    set(handles.MatFileE,'Enable','Off')
    set(handles.MatBrowseE,'Enable','Off')
else
    set(handles.MatFileE,'Enable','On')
    set(handles.MatBrowseE,'Enable','On')
    set(handles.EstDef,'Value',0)
    set(handles.no,'Enable','Off');
    set(handles.Ae,'Enable','Off');
    set(handles.Be,'Enable','Off');
    set(handles.Ce,'Enable','Off');
    set(handles.De,'Enable','Off');
end

% --- Executes during object creation, after setting all properties.
function MatFileE_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function MatFileE_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1]) if
isempty(get(hObject,'String'))
    % do nothing
```

```
elseif exist(get(hObject,'String'))==0
    set(hObject,'BackgroundColor',[1 1 0])
    errordlg('File Does Not Exist As Entered',
    'Invalid Entry','modal')
else
    fokay=fileokay(get(hObject,'String'));
    if fokay==0
        errordlg('Please load a .mat file',
        'Invalid Entry','modal')
        set(h,'BackgroundColor',[1 1 0])
    end
end

% --- Executes on button press in MatBrowseE.
function MatBrowseE_Callback(hObject, eventdata, handles)

[filename, pathname] = uigetfile( ...
       {'*.mat', 'Simulink Model'}, ...
        'Select a MATLAB File');

set(handles.SimFile,'BackgroundColor',[1 1 1]) if filename==0
    % do nothing
else
    file=fullfile(pathname,filename);
    set(handles.MatFileE,'String',file)
    fokay=fileokay(file);
    if fokay==0
        set(handles.MatFileE,'BackgroundColor',[1 1 0])
        errordlg('File is Not in Correct Form',
        'Invalid Entry','modal')
    end
end

% --- Executes on button press in DefContButton.
function DefContButton_Callback(hObject, eventdata, handles)
set(handles.ContText,'Visible','On')
set(handles.ControllerOn,'Visible','On')
set(handles.ContDef,'Visible','On')
set(handles.CEFrame,'Visible','On')
set(handles.ContDefText,'Visible','On')
set(handles.statesText,'Visible','On')
set(handles.nc,'Visible','On')
set(handles.pcText,'Visible','On')
set(handles.pc,'Visible','On')
set(handles.Atext,'Visible','On')
set(handles.A,'Visible','On')
set(handles.B1text,'Visible','On')
set(handles.B1,'Visible','On')
set(handles.B2text,'Visible','On')
set(handles.B2,'Visible','On')
set(handles.Ctext,'Visible','On')
set(handles.C,'Visible','On')
set(handles.D1text,'Visible','On')
set(handles.D1,'Visible','On')
set(handles.D2text,'Visible','On')
set(handles.D2,'Visible','On')
set(handles.ContMat,'Visible','On')
set(handles.MatBrowse,'Visible','On')
set(handles.MatFile,'Visible','On')
set(handles.ContSim,'Visible','On')
set(handles.SimBrowse,'Visible','On')
set(handles.SimFile,'Visible','On')

set(handles.EstText,'Visible','Off')
set(handles.EstDef,'Visible','Off')
set(handles.EstDefText,'Visible','Off')
set(handles.statesEtext,'Visible','Off')
set(handles.no,'Visible','Off')
set(handles.AeText,'Visible','Off')
set(handles.Ae,'Visible','Off')
set(handles.BeText,'Visible','Off')
set(handles.Be,'Visible','Off')
set(handles.CeText,'Visible','Off')
set(handles.Ce,'Visible','Off')
set(handles.DeText,'Visible','Off')
set(handles.De,'Visible','Off')
set(handles.MatLoadE,'Visible','Off')
set(handles.MatBrowseE,'Visible','Off')
set(handles.MatFileE,'Visible','Off')

% --- Executes on button press in DefEstButton.
function DefEstButton_Callback(hObject, eventdata, handles)
set(handles.EstText,'Visible','On')
set(handles.EstDef,'Visible','On')
set(handles.CEFrame,'Visible','On')
set(handles.EstDefText,'Visible','On')
set(handles.statesEtext,'Visible','On')
set(handles.no,'Visible','On')
set(handles.AeText,'Visible','On')
set(handles.Ae,'Visible','On')
set(handles.BeText,'Visible','On')
set(handles.Be,'Visible','On')
set(handles.CeText,'Visible','On')
set(handles.Ce,'Visible','On')
set(handles.DeText,'Visible','On')
set(handles.De,'Visible','On')
set(handles.MatLoadE,'Visible','On')
set(handles.MatBrowseE,'Visible','On')
set(handles.MatFileE,'Visible','On')

set(handles.ContText,'Visible','Off')
set(handles.ControllerOn,'Visible','Off')
set(handles.ContDef,'Visible','Off')
set(handles.ContDefText,'Visible','Off')
set(handles.statesText,'Visible','Off')
set(handles.nc,'Visible','Off')
set(handles.pcText,'Visible','Off')
set(handles.pc,'Visible','Off')
set(handles.Atext,'Visible','Off')
set(handles.A,'Visible','Off')
set(handles.B1text,'Visible','Off')
set(handles.B1,'Visible','Off')
set(handles.B2text,'Visible','Off')
set(handles.B2,'Visible','Off')
set(handles.Ctext,'Visible','Off')
set(handles.C,'Visible','Off')
set(handles.D1text,'Visible','Off')
set(handles.D1,'Visible','Off')
set(handles.D2text,'Visible','Off')
set(handles.D2,'Visible','Off')
set(handles.ContMat,'Visible','Off')
set(handles.MatBrowse,'Visible','Off')
set(handles.MatFile,'Visible','Off')
set(handles.ContSim,'Visible','Off')
set(handles.SimBrowse,'Visible','Of')
set(handles.SimFile,'Visible','Off')
```

```
% --- Executes on button press in TargetS.
function TargetS_Callback(hObject, eventdata, handles)

if get(hObject,'Value') == 1
    set(handles.SineWave,'Value',0)
    set(handles.ThetaD,'Enable','On')
    set(handles.OmegaD,'Enable','On')
    set(handles.Amp,'Enable','Off')
    set(handles.Freq,'Enable','Off')
else
    set(handles.ThetaD,'Enable','Off')
    set(handles.OmegaD,'Enable','Off')
end

% --- Executes on button press in SineWave.
function SineWave_Callback(hObject, eventdata, handles)

if get(hObject,'Value') == 1
    set(handles.TargetS,'Value',0)
    set(handles.Amp,'Enable','On')
    set(handles.Freq,'Enable','On')
    set(handles.ThetaD,'Enable','Off')
    set(handles.OmegaD,'Enable','Off')
else
    set(handles.Amp,'Enable','Off')
    set(handles.Freq,'Enable','Off')
end

% --- Executes during object creation, after setting all properties.
function Amp_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

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

```
set(hObject,'BackgroundColor',[1 1 1]);

user_entry=str2num(get(hObject,'String'));

if isempty(user_entry) | user_entry<0
    set(hObject,'BackgroundColor',[1 1 0]);
    errordlg('Please enter a positive Amplitude',
    'Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function Freq_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function Freq_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1]);

user_entry=str2num(get(hObject,'String'));

if isempty(user_entry)
    set(hObject,'BackgroundColor',[1 1 0]);
    errordlg('Please enter a Frequency','Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function ThetaD_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

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

```
set(hObject,'BackgroundColor',[1 1 1]);

user_entry=str2num(get(hObject,'String')); w =
str2num(get(handles.OmegaD,'String'));

if isempty(user_entry)
    set(hObject,'BackgroundColor',[1 1 0]);
    errordlg('Please enter a desired target angle',
    'Invalid Entry','modal')
elseif user_entry~=0 & w~=0
    set(hObject,'BackgroundColor',[1 1 0]);
    errordlg('Cannot have non-zero target rate and target angle',
    'Invalid Entry','modal')
end

% --- Executes during object creation, after setting all properties.
function OmegaD_CreateFcn(hObject, eventdata, handles)

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',
    get(0,'defaultUicontrolBackgroundColor'));
end

function OmegaD_Callback(hObject, eventdata, handles)

set(hObject,'BackgroundColor',[1 1 1]);

user_entry=str2num(get(hObject,'String')); theta =
str2num(get(handles.ThetaD,'String'));

if isempty(user_entry)
    set(hObject,'BackgroundColor',[1 1 0]);
    errordlg('Please enter a desired target rate',
    'Invalid Entry','modal')
elseif user_entry~=0 & theta~=0
    set(hObject,'BackgroundColor',[1 1 0]);
    errordlg('Cannot have non-zero target rate and target angle',
    'Invalid Entry','modal')
end
```

Appendix C

TableSat/MATLAB Communications Code

```
function handleOut = handle_uplinkStateData(flags,sock)

global tsat_msg_waits

  % Request more yummy data
tsat_send_msg(81,{uint8(0)},TSsock);

% Read the data out of the receive buffer
x=pnet(sock,'read',6,'double','intel');

handleOut=[81;x(2:4)'];

function handleErr = handle_ackMesg(flags,sock)

global TSgui tsat_msg_waits

theMesg=pnet(sock,'read',1,'char','intel');

msgNum = ceil(theMesg); tsat_msg_waits(msgNum)=0;

%if (msgNum==6)
%    if (TSgui.controllerOn)
%        set(TSgui.controlButton,'Value',0.0);
%        disp('Setting button to 0');
%    else
%        set(TSgui.controlButton,'Value',1.0);
%        disp('Setting button to 1');
%    end
%end

handleErr = [2;0];
```

```
function handleErr = handle_initOK(flags,sock)

% Stub which does nothing except return error!

handleErr = 1;

function handleErr = handle_uplinkRawData(flags,sock)

global tsat_msg_waits oldRawT oldRawCSS oldRawTAM oldRawRate
oldRawFan TSgui

% Read the data out of the receive buffer
x=pnet(sock,'read',11,'double','intel') handleErr=0; return;

% If not showing the data, just return
if (~TSgui.showData(2))
    handleErr=0;
    return
end

% Otherwise, draw the graphs
figure(2); if (TSgui.firstTime(2))
    oldRawT =    x(1);
    oldRawCSS=  x(2:5);
    oldRawRate= x(6);
    oldRawTAM= x(7:9);
    oldRawFan = x(10:11);
    clf;
end

function handleOut = handle_uplinkStateData(flags,sock)

global tsat_msg_waits numrec oldT dttot firstCall

% Read the data out of the receive buffer
x=pnet(sock,'read',6,'double','intel');

if (firstCall)
    firstCall=0;
    oldT=x(1);
end
```

```
dttot = dttot+(x(1)-oldT); numrec=numrec+1; oldT=x(1);

handleOut=[81;x(1:6)'];

function handleErr = handle_uplinkStatusData(flags,sock)

global recvState

theMesg=pnet(sock,'read',5,'char','intel');

recvState = 1;
reply = sprintf(' Run State is %d\n Controller Mode is
%d\n Fan Mode is %d\n',theMesg(1),theMesg(2),theMesg(3));
reply2 = sprintf(' Actuator Mode is %d\n Gyro Mode is
 %d\n',theMesg(4),theMesg(5));
disp([reply,reply2]);

handleErr = 0;

function handleOut = handle_uplinkStateData(flags,sock)

global tsat_msg_waits

% Read the data out of the receive buffer
x=pnet(sock,'read',6,'double','intel');

handleOut=[81;x(2:5)'];

function sock=tsat_init(port,tsat_addr)

global tsat_msg_headers tsat_msg_handlers tsat_msg_flags
tsat_msg_waits

MAXMSG = 256;  % Maximum number of messages (arbitrary!)

bsize = 1;          % Size of byte (single char)
dsize = 8;          % Size of double
vsize = 32767;   % Flag for variably-sized messages

% Open socket on recognized port for listening
sock=pnet('udpsocket',port);
```

```
% All outgoing connections will be to tsat_addr
pnet(sock,'udpconnect',tsat_addr,port);

% Null out all the arrays

for i=1:MAXMSG
    tsat_msg_headers{i}  = [];
    tsat_msg_handlers{i} = @handle_NULL;
    tsat_msg_flags(i)      = 0;
    tsat_msg_waits(i)     = 0;
end

          % Receive Acknowledgements
Msgs = { {2,   @handle_ackMesg,bsize},  ...
           % Set run Mode (send 0 to shutdown)
              {4,   @handle_NULL,bsize}, ...
           % Set Control Mode (0=off (default), 1=on)
              {6,   @handle_NULL,bsize}, ...
           % Set Fan Mode (0=no fan comp, 1 = fric/speed comp)
              {8,   @handle_NULL,bsize}, ...
           % Set Actuator Mode (0=Cont (default), 1=PWM, 2=BANGBANG,
           % 2.x -> x is deadzone)
              {10, @handle_NULL,dsize}, ...
           % Set Gyro autocal  (0 = off (default), 1 = on)
              {12, @handle_NULL,bsize}, ...
           % Set controller definition (structure)
              {15, @handle_NULL,vsize}, ...
           % Set Fan speed data (double npts,vector wpts,
           % vector forcepts)
              {18, @handle_NULL,vsize}, ...
           % Set Fan friction data (double npts, vector wpts,
           % vector fricpts)
              {21, @handle_NULL,vsize}, ...
           % Set controller target (double Mode
           % (0 = hold,1=sine track), double xd[2])
              {24, @handle_NULL,3*dsize}, ...
           % Set estimator definition (structure)
              {27, @handle_NULL,vsize}, ...
           % Set gyro autocal adaptation gains
              {30, @handle_NULL,2*dsize}, ...
           % Set controller/estimator sample rates (2 values in Hz)
              {33, @handle_NULL,2*dsize}, ...
```

```
            % Receive current state estimate from vehicle
               {60, @handle_StateData,6*dsize},...
            %  Receive raw sensor readings from vehicle
               {63, @handle_RawData,11*dsize}, ...
            % Receive status data flags from vehicle
               {66, @handle_StatusData,5*bsize},...
            % Request state data from vehicle
               {81, @handle_NULL, bsize}, ...
            % Request raw sensor readings from vehicle
               {84, @handle_NULL, bsize}, ...
            % Request status data from vehicle
               {87, @handle_NULL, bsize}, ...
           % Request recalibration of sensors
               {99, @handle_NULL, bsize}, ...
           };


%  Report success

for i=1:size(Msgs,2)
    tsat_register_msg(Msgs{i}{1},0,Msgs{i}{3},Msgs{i}{2});
end

reply=sprintf('Tablesat communications initialized
 on port %d\n',port);
disp(reply);

function handleOut=tsat_recv_msg(sock)

global tsat_msg_headers tsat_msg_handlers tsat_msg_flags
tsat_msg_watch tsat_msg_waits

handleOut=0;

size=pnet(sock,'readpacket',65500,'noblock');
tsat_msg_waits(tsat_msg_watch) =
    tsat_msg_waits(tsat_msg_watch)+1;

if (size>0)
     % Get the header and message number
    header=pnet(sock,'read',5,'uint8','intel');
    msgnum = header(1);
    handler = tsat_msg_handlers{msgnum};

    % Look for errors
  % Check if message is registered
    if (isequal(handler,@handle_NULL))
        reply=sprintf('Message %d received but
        not registered for receipt\n',char(msgnum));
        disp(reply);
        return;
    end
      % Check for correct payload length
    if (header(3)~=tsat_msg_headers{msgnum}(3) | ...
        header(4)~=tsat_msg_headers{msgnum}(4))
        reply=sprintf('Message %d received but with
        incorrect payload size\n',char(msgnum));
        disp(reply);
        return;
    end

    % Get message flags and invoke handler
    flags = header(2);
    handleOut=feval(handler,flags,sock);

    %if (handleErr)
        %reply=sprintf('Handler for message
        %d reports error %d\n',char(msgnum),handleErr);
        %disp(reply);
        %end
end

function tsat_register_msg(msgnum,flags,msgsize,handler)

global tsat_msg_headers tsat_msg_handlers tsat_msg_flags


%handlerAssign = 'handleErr=';
%handlerArgs = '(flags,sock);';
%tsat_msg_handlers{msgnum}=
    strcat(handlerAssign,handler,handlerArgs);

tsat_msg_handlers{msgnum} = handler; tsat_msg_flags(msgnum) =
flags;
```

```
tsat_msg_headers{msgnum}=[uint8(msgnum),uint8(flags),
    uint8(floor(msgsize/256)),uint8(mod(msgsize,256)),uint8(0)];

reply=sprintf('Message %d registered\n',msgnum);
disp(reply);

function tsat_send_msg(msgnum,data,sock)

global tsat_msg_headers tsat_msg_handlers tsat_msg_flags

if (length(tsat_msg_headers{msgnum})==5)
    pnet(sock,'write',tsat_msg_headers{msgnum},'intel');
    if (isstruct(data))  % flatten structures into cell array
        data=struct2cell(data);
    end
    for i=1:length(data)  % loop through cells
        if (length(data{i})>0)
            if (size(data{i},1)>1)
              pnet(sock,'write',data{i}','intel');
            else
              pnet(sock,'write',data{i},'intel');
          end
        end
    end
    pnet(sock,'writepacket');
else
    reply=sprintf('Message %d has not been
        registered\n',msgnum);
    disp(reply);
end
```

Appendix D

TableSat Truth Model Code

```
%TableSat Time Domain Sim Initilization
clear all

T_Cont = 0.02; T_Est = 0.02; sigmaTam = 2.2;
sigmaGyro = 0.09;

%Note for the Controller, B = [B2 B1], D = [D2 D1]
% Acd = [];
% Bcd = [];
% Ccd = [];
% Dcd = [1 0 0 0 0 0 0];

load('E:\Table Sat\TableSat\Controllers\MBCi_t.mat') Acd =
TS_Con.A; Bcd = [TS_Con.B2 TS_Con.B1]; Ccd = TS_Con.C; Dcd =
[TS_Con.D2 TS_Con.D1];

% load('E:\Table Sat\TableSat\Estimators\TSatFilter_2Sample.mat')
% Ae = TS_Est.Ae;
% Be = TS_Est.Be;
% Ce = TS_Est.Ce;
% De = TS_Est.De;

Ae = []; Be = []; Ce = []; De = eye(5,5);

function [sys,x0,str,ts] = sfuntmpl(t,x,u,flag,params)
%SFUNTMPL General M-file S-function template
%   With M-file S-functions, you can define you own ordinary
%   differential equations (ODEs), discrete system equations,
%   and/or just about any type of algorithm to be used within
%   a Simulink block diagram.
%
```

```
%   The general form of an M-File S-function syntax is:
%       [SYS,X0,STR,TS] = SFUNC(T,X,U,FLAG,P1,...,Pn)
%
%   What is returned by SFUNC at a given point in time, T,
%   depends on the, value of the FLAG, the current state
%   vector, X, and the current input vector, U.
%
%   FLAG   RESULT             DESCRIPTION
%   -----  ------             ------------------------------
%   0      [SIZES,X0,STR,TS]  Initialization, return system
%                             sizes in SYS, initial state in
%                             X0, state ordering strings in
%                             STR, and sample times in TS.
%   1      DX                 Return continuous state
%                             derivatives in SYS.
%   2      DS                 Update discrete states
%                             SYS = X(n+1)
%   3      Y                  Return outputs in SYS.
%   4      TNEXT              Return next time hit for variable
%                             step sample time in SYS.
%   5                         Reserved for future (root finding).
%   9      []                 Termination, perform any cleanup
%                             SYS=[].
%
%
%   The state vectors, X and X0 consists of continuous
%   states followed by discrete states.
%
%   Optional parameters, P1,...,Pn can be provided to
%   the S-function and used during any FLAG operation.
%
%   When SFUNC is called with FLAG = 0, the following
%   information should be returned:
%
%      SYS(1) = Number of continuous states.
%      SYS(2) = Number of discrete states.
%      SYS(3) = Number of outputs.
%      SYS(4) = Number of inputs.
%               Any of the first four elements in SYS
%               can be specified as -1 indicating that
%               they are dynamically sized. The actual
%               length for all other flags will be equal
%               to the length of the input, U.
%       SYS(5) = Reserved for root finding. Must be zero.
%       SYS(6) = Direct feedthrough flag (1=yes, 0=no).
%               The s-function has direct feedthrough if
%               U is used during the FLAG=3 call. Setting
%               this to 0 is akin to making a promise that
%               U will not be used during FLAG=3. If you
%               break the promise then unpredictable
%               results will occur.
%       SYS(7) = Number of sample times. This is the
%               number of rows in TS.
%
%
%       X0     = Initial state conditions or [] if no states.
%
%       STR    = State ordering strings which is generally
%               specified as [].
%
%       TS     = An m-by-2 matrix containing the sample
%               time (period, offset) information.
%               Where m = number of sample times. The
%               ordering of the sample times must be:
%
%               TS = [0      0,
%               : Continuous sample time.
%                     0      1,
%               : Continuous, but fixed in minor step
%               sample time.
%               PERIOD OFFSET, : Discrete sample
%               time where PERIOD > 0 & OFFSET < PERIOD.
%                     -2     0];
%               : Variable step discrete sample time
%                 where FLAG=4 is used to get time of
%                 next hit.
%
%               There can be more than one sample
%               time providing they are ordered such
%               that they are monotonically
%               increasing. Only the needed sample
%               times should be specified in TS.
%               When specifying than one
%               sample time, you must check for sample
%               hits explicitly by seeing if
%        abs(round((T-OFFSET)/PERIOD) - (T-OFFSET)/PERIOD)
%               is within a specified tolerance,
%               generally 1e-8. This tolerance is
%               dependent upon your model's sampling times
%               and simulation time.
%
%               You can also specify that the sample time
%               of the S-function is inherited from the
%               driving block. For functions which
%               change during minor steps, this is done by
%               specifying SYS(7) = 1 and TS = [-1 0].
%               For functions which are held during minor
%               steps, this is done by specifying
%               SYS(7) = 1 and TS = [-1 1].

%   Copyright 1990-2002 The MathWorks, Inc.
%   $Revision: 1.18 $

%
% The following outlines the general structure of an
% S-function.
%
switch flag,

  %%%%%%%%%%%%%%%%%%
  % Initialization %
  %%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts]=mdlInitializeSizes;

  %%%%%%%%%%%%%%%
  % Derivatives %
  %%%%%%%%%%%%%%%
  case 1,
    sys=mdlDerivatives(t,x,u,params);

  %%%%%%%%%%
  % Update %
  %%%%%%%%%%
  case 2,
    sys=mdlUpdate(t,x,u);
  %%%%%%%%%%%
  % Outputs %
  %%%%%%%%%%%
  case 3,
    sys=mdlOutputs(t,x,u);

  %%%%%%%%%%%%%%%%%%%%%%%
  % GetTimeOfNextVarHit %
  %%%%%%%%%%%%%%%%%%%%%%%
  case 4,
    sys=mdlGetTimeOfNextVarHit(t,x,u);

  %%%%%%%%%%%%%
  % Terminate %
  %%%%%%%%%%%%%
  case 9,
    sys=mdlTerminate(t,x,u);

  %%%%%%%%%%%%%%%%%%%%
  % Unexpected flags %
  %%%%%%%%%%%%%%%%%%%%
  otherwise
    error(['Unhandled flag = ',num2str(flag)]);

end

% end sfuntmpl

%
%============================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for
% the S-function.
%============================================================
%
function [sys,x0,str,ts]=mdlInitializeSizes global fric1 fric2
%
% call simsizes for a sizes structure, fill it in and convert
% it to a sizes array.
%
% Note that in this example, the values are hard coded.
% This is not a recommended practice as the characteristics
% of the block are typically defined by the S-function
% parameters.
%
sizes = simsizes;

sizes.NumContStates  = 4; sizes.NumDiscStates  = 0;
sizes.NumOutputs     = 4; sizes.NumInputs      = 1;
sizes.DirFeedthrough = 0;
sizes.NumSampleTimes = 1; % at least one sample time is needed

sys = simsizes(sizes);

%
% initialize the initial conditions
%
x0  = [(180)*pi/180; 0.0*pi/180; 0; 0];

%
% str is always an empty matrix
%
str = [];

%
% initialize the array of sample times
%
ts  = [0 0];

% end mdlInitializeSizes

%
%=============================================================
% mdlDerivatives
% Return the derivatives for the continuous states.
%=============================================================
%
function sys=mdlDerivatives(t,x,u,params)

% Parse out state variables
th=x(1); % angle
w = x(2); % rate
nu1 = x(3); % fan 1
```

```
nu2 = x(4); % fan 2

% Get compensation parameters
fcomp = params(10); fmag = params(9);

% Implement friction compensation
if (fcomp)
    if (w>=0)
        v = u + fmag;
    else
        v = u - fmag;
    end
else
    v = u;
end

% Implement actuator deadzone compensation
vmag = abs(v); if (vmag>=3)
    v1 = vmag;
    v2 = 0.0;
elseif (vmag>=.1)
    v2 = 3;
    v1 = vmag+3;
else
    v1 = 0.0;
    v2 = 0.0;
end

% "Jet select" logic for +/- fans
if (v>0)
    vp = v1;
    vm = v2;
else
    vp = v2;
    vm = v1;
end

% -----------Compute system dynamics-------------

sys=zeros(4,1);

% Tablesat physical constants
```

```
It = params(1); l = params(2); ft = params(3); ff = params(4);
alpha = params(5);
Kwf = params(6); %speed to force constant
Kvw = params(7); %voltage to speed constant
fanTC = params(8); eps = 0.01;

% Model actuator saturation
if (vp>12)
    vp = 12;
elseif (vp<0)
    vp = 0.0;
end if (vm>12)
    vm = 12;
elseif (vm<0.0)
    vm = 0.0;
end

% Model table friction
if (w>0)
    Ft = ft;
elseif (w<0)
    Ft = -ft;
else
    Ft=0;
end

% Model fan 1 friction
if (nu1>eps)
    Ff1 = ff*exp(-nu1/fanTC);
else
    Ff1 = 0;
end

% Model fan 2 friction
if (nu2>eps)
    Ff2 = ff*exp(-nu2/fanTC);
else
    Ff2 = 0;
end

sys(1) = w;
sys(2) = (l*Kwf*nu1 - l*Kwf*.99*nu2 - Ft)/It;

sys(3)=-alpha*nu1 + Kvw*(vp - Ff1);

sys(4) = -alpha*nu2 + Kvw*(vm-Ff2);

% end mdlDerivatives

%
%=====================================================
% mdlUpdate
% Handle discrete state updates, sample time hits,
% and major time step requirements.
%=====================================================
%
function sys=mdlUpdate(t,x,u)

sys = [];

% end mdlUpdate

%
%======================================================
% mdlOutputs
% Return the block outputs.
%======================================================
%
function sys=mdlOutputs(t,x,u)

sys = [mod(180*x(1)/pi,360)-180;180*x(2)/pi;x(3);x(4)];

% end mdlOutputs

%
%======================================================
% mdlGetTimeOfNextVarHit
% Return the time of the next hit for this block.
% Note that the result is absolute time.  Note that
% this function is only used when you specify a
% variable discrete-time sample time [-2 0] in the
% sample time array in mdlInitializeSizes.
%======================================================
%
function sys=mdlGetTimeOfNextVarHit(t,x,u)

%  Example, set the next hit to be one second later.
sampleTime = 1;
sys = t + sampleTime;

% end mdlGetTimeOfNextVarHit

%
%=================================================
% mdlTerminate
% Perform any end of simulation tasks.
%=================================================
%
function sys=mdlTerminate(t,x,u)

sys = [];

% end mdlTerminate
```

BIBLIOGRAPHY

[1] E. Atkins, B. Roberts, D. Bawek, and R. Besser. A high-speed, extensible hardware system for the TableSat 1-dof satellite simulator. Technical report, University of Maryland, College Park, Space Systems Lab, College Park, MD 20742, 2005 (in preparation).

[2] BEI Technologies, Inc., 2700 Systron Drive, Concord, CA 94518. *BEI Gyrochip Horizon Micromachined Angular Rate Sensor*, 1998.

[3] Diamond Systems Corporation, 8430-D Central Ave. Newark, CA 94560. *PROMETHEUS: High Integration PC/104 CPU with Ethernet and Data Acquisition User Manual V1.44*, 2003.

[4] Honeywell, Solid State Electronics Center, www.magneticsensors.com. *HMC2003 Three-Axis Magnetic Sensor Hybrid*.

[5] Lawrence L. Lapin. *Modern Engineering Statistics*. Wadsworth Publishing Company, Belmont, CA, 1997.

[6] LINKSYS, 17401 Armstrong Ave. Irvine, CA 92614. *Wireless-B Broadband Router*, 2003.

[7] OTC Wireless, www.otcwireless.com. *ASR102 802.11b Access Point*.

[8] Sunon, www.sunon.com. *Sunon DC Brushless Fan*, 2004.

[9] Greg Welch and Gary Bishop. An introduction to the Kalman Filter. Technical report, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599, 2004.