

# Filter and Bounding Algorithm Development for a Helmet Mounted Micromechanical Inertial Sensor Array

by

Erik S. Bailey

S.B. Aeronautics and Astronautics  
Massachusetts Institute of Technology, 1998

Submitted to the Department of Aeronautics and Astronautics in  
partial fulfillment of the requirements for the degree of  
**MASTER OF SCIENCE IN AERONAUTICS AND ASTRONAUTICS**  
**AT THE**  
**MASSACHUSETTS INSTITUTE OF TECHNOLOGY**

September, 2000

© Erik Stephen Bailey, 2000. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly  
paper and electronic copies of this thesis document in whole or in part.

Author .....

Department of Aeronautics and Astronautics

Certified by .....

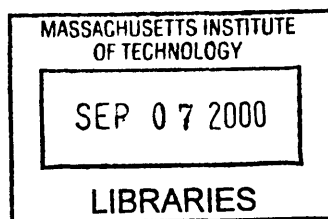
Tom P. Thorvaldsen  
Charles Stark Draper Laboratory  
Thesis Supervisor

Certified by .....

Professor John J. Deyst  
Aeronautics and Astronautics  
Thesis Advisor

Accepted by .....

Professor Nesbitt W. Hagood, IV  
Chairman, Departmental Graduate Committee



**Aero**



# **Filter and Bounding Algorithm Development for a Helmet Mounted Micromechanical Inertial Sensor Array**

by

Erik S. Bailey

Submitted to the Department of Aeronautics and Astronautics on  
June 30, 2000, in partial fulfillment of the requirements for the  
degree of Master of Science in Aeronautics and Astronautics

## **Abstract**

The technical evolution of head mounted displays (HMDs) and micromechanical inertial sensor arrays (MMISAs) have, until recently, occurred independently. This thesis details the development and simulation results of an inertial helmet-mounted head tracker for a T-38 jet aircraft flight environment. Primary focuses are the state error estimation filter and the bounding algorithms used to estimate the head position and orientation during various flight conditions. Also included is a discussion of the application of the Draper MMISA to a cueing system for an HMD in a military air vehicle environment. Particular attention is paid to necessary requirements to meet pointing accuracies for fire control system hand-off applications for the next generation of air to air missiles, such as the AIM-9X Sidewinder missile. A Markov process coupling technique used in this research is shown to achieve pointing accuracies of 4 to 11 milliradians. A generalization of the navigation filter to any number of inertial navigators with known relative positioning and attitude coupled using Markov process propagation matrices is also included, along with examples of future applications, in addition to helmet-mounted cueing systems, for vehicle applications.

Thesis Supervisor: Tom Thorvaldsen  
Title: Tactical Guidance and Navigation Group Leader,  
C. S. Draper Laboratory

Thesis Advisor: John J. Deyst  
Title: Professor, MIT Department of Aeronautics and Astronautics

## Acknowledgements

I would like to express my sincerest gratitude to the Charles Stark Draper Laboratory for hiring me as a Draper Fellow. Without the opportunity presented by them I would not be completing my Master of Science at MIT. I also wish to thank the staff of the Draper Laboratory for their daily influence and making their vast expanse of knowledge and experience available to me which has expanded and broadened my own horizons both as an engineer and as an individual.

Special thanks to my thesis supervisor at the CSDL, Tom P. Thorvaldsen, who provided invaluable guidance, insight, and additional personal time commitment into this project.

I would also like to thank my thesis advisor in the MIT Aeronautics and Astronautics Department, Professor John J. Deyst, who has assisted in the written presentation of my thesis content.

Dale Landis, of the CSDL has been of great assistance in both learning the existing MEMS IMU navigation code and adapting it for this application.

I'd also like to thank John Danis, Dave Hauger, Linda Leonard, Wade Hampton, Scott Berry, and Keith Mason of the Draper Simulation Lab for all of their assistance and guidance in writing the C-Sim simulation for this thesis.

I thank all my close friends —Adriane Faust, Carol Cheung, Chris Deards, Steve Conahan, Craig Henderson, Lisette Lyne, Todd Harrison, Elise Westmeyer, Scott Uebelhart, Rick Francis, and Celine Fauchon—for their help and support which have made the MIT environment not only enlightening but also enjoyable.

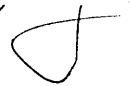
I also wish to thank Edward Ouellette, who has gone above and beyond the call of friendship. Without your hospitality and almost daily support over the past two years, this thesis would not be completed.

Finally, I wish to thank my family—especially my parents, John and Sharon Bailey—for their seemingly endless support, patience, and encouragement throughout my 4 years of undergraduate and 2 years of graduate studies at MIT. Without the two of you, I would never have been able to work on projects that fly and go into space—a dream of mine which I've had as long as I can remember. Thank you for everything, from the bottom of my heart.

This thesis was research and written at the Charles Stark Draper Laboratory under Internal Company Sponsored Research Project #200, MEMS Helmet Mounted Cueing System.

Publication of this thesis does not constitute approval by the Laboratory of the findings of conclusions contained herein. It is published for the exchange and stimulation of ideas.

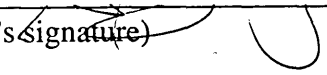
Erik S. Bailey  
June 29, 2000

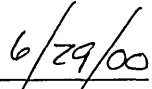


# Assignment

Draper Laboratory Report Number T-1382.

In consideration for the research opportunity and permission to prepare my thesis by and at The Charles Stark Draper Laboratory, Inc., I hereby assign my copyright of the thesis to The Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.

  
(author's signature)

  
(date)

# Table of Contents

Abstract.....	3
Acknowledgements.....	5
Table of Contents.....	7
List of Figures.....	11
List of Tables.....	13
<b>Chapter 1</b> Introduction and Background.....	15
1.1 Helmet Mounted Cueing Systems.....	15
1.1.1 Overview of Currently Available Systems.....	15
Magnetic Trackers.....	16
UltraSonic trackers.....	17
Optical Trackers.....	18
Inertial Trackers.....	18
Hybrid Systems.....	20
1.1.2 Current Applications of Head Trackers.....	20
Virtual Reality Applications.....	21
Helicopter and Jet Aircraft Applications.....	22
Foot Soldier Applications.....	24
1.1.3 Magnetic Tracking Issues.....	24
Comparative Reasoning for a Better Inertial HMCS.....	25
1.1.4 Motivation.....	26
Developing an Inertial HMCS (IHMCS) for the Flight Environment.....	26
1.2 Draper Micromechanical Inertial Sensors.....	27
1.2.1 Technology.....	28
MEMS Accelerometers.....	28
MEMS Gyroscopes.....	29
1.2.2 First Generation MMISA Application: The Competent Munitions Advanced Technology Demonstrator (CMATD).....	30
1.2.3 Sensor Applicability.....	31
1.2.4 Technology Road map.....	32
First Generation MMISA Sensors.....	32
Draper MEMS: The Next Generation.....	32
1.3 Helmet Mounted Inertial System Concept.....	33
1.3.1 Hardware.....	35
Placement of Accelerometers.....	36
Gyroscope Mounting.....	39
Vehicle IMU vs. HMCS IMU.....	40
Head Tracking Computer.....	40
1.3.2 Necessary System Algorithms.....	41
Kalman Filter.....	41
Bounding Algorithm.....	42
<b>Chapter 2</b> System Dynamics.....	43
2.1 Coordinate Frames.....	43
2.1.1 The ECEF Navigation Frame.....	43

2.1.2	The Body Frame .....	44
2.1.3	The Helmet Frame .....	44
2.2	Relative Dynamics .....	45
2.2.1	Rotating Frame Dynamics Applied to an Aircraft on the Earth .....	46
2.2.2	Rotating Frame Dynamics Applied to Helmet Acceleration .....	48
2.3	Vehicle Dynamic Model .....	49
2.3.1	Aerodynamic Model .....	49
	Control Surfaces.....	50
	Force Coefficients.....	50
	Moment Coefficients .....	51
2.3.2	Propulsion Model.....	52
2.3.3	Mass Model.....	53
2.3.4	Equations of Motion .....	54
	Aerodynamic Quantities and Earth Velocities.....	54
	Attitude .....	54
2.4	Pilot Dynamic Model.....	57
2.4.1	Stick-Figure Model .....	57
<b>Chapter 3</b>	<b>Simulation Architecture.....</b>	<b>59</b>
3.1	Overall Architecture.....	59
3.1.1	Concept .....	59
3.2	Structure .....	60
3.2.1	Implementation .....	61
	Trajectory and Hardware Simulation in Draper C-Sim Environment .....	64
	Navigation Algorithm in C-Sim .....	64
	Kalman Filter and Data Analysis in MATLAB.....	64
3.3	Physics and Dynamics Models .....	65
3.3.1	T-38 Flight Vehicle Model .....	65
	Aircraft.....	65
	Environment.....	68
3.3.2	Pilot Model .....	69
	Angular Position of the Pilot Torso and Head.....	69
	Angular Velocity and Acceleration of the Helmet .....	70
	Specific Force on the Helmet.....	71
3.4	Instrument Models .....	72
3.4.1	GPS Constellation/Receiver Model .....	72
3.4.2	MMISA & EGI Models .....	72
3.5	User Interface.....	73
3.5.1	Hardware.....	73
3.5.2	Graphics .....	75
3.5.3	Trajectory Generation and Control .....	76
	Aircraft Flight Path .....	76
	Pilot Motion .....	76
3.5.4	Data Storage & Analysis.....	77
<b>Chapter 4</b>	<b>Algorithms .....</b>	<b>79</b>
4.1	Navigation System Overview .....	79
4.1.1	Essence of the Navigation and Coupling Algorithms.....	79



4.1.2	Effect of the Gyros and the Attitude Algorithm .....	81
4.2	Navigation Equations.....	82
4.2.1	Attitude Algorithm.....	82
4.2.2	Velocity and Position Determination.....	84
	Additional Accelerometer Compensation.....	84
	Accelerometer Lever Arm Compensation .....	84
	Delta-Velocity and Attitude Accumulation (Rate Compensation) .....	85
	Earth Mass Attraction and Earth Rotation Navigation Corrections.....	86
	Velocity Summation and Position Integration.....	87
4.3	Error Estimation Kalman Filter .....	88
4.3.1	The Discrete Kalman Filter Equations.....	88
	Propagation and Measurement Equations.....	88
4.3.2	The Navigation Error Estimation Filter Formulation .....	91
	State Vector Description.....	91
	Propagation Matrix Formulation.....	92
	Measurements .....	95
4.3.3	Generalization to N Navigators With One Master Kalman Filter .....	97
	Parent Navigator vs. Internally Referenced (Markov) States .....	98
4.4	Bounding Algorithm .....	99
4.4.1	Description of a Markov Process.....	99
4.4.2	Attempt to Ensure Markov Process Remains Strictly Bounded .....	100
4.4.3	Implementation .....	101
	MATLAB Test.....	101
	Simulation Implementation.....	102
<b>Chapter 5</b>	<b>Data Analysis.....</b>	<b>103</b>
5.1	Simulated Truth Data.....	103
5.2	Filter Performance .....	104
5.2.1	Single Data Run Analysis .....	104
5.2.2	Twenty-Run Monte Carlo Analyses .....	107
5.3	Bounding Algorithm Performance.....	115
<b>Chapter 6</b>	<b>Conclusion and Recommendations.....</b>	<b>117</b>
6.1	Results.....	117
6.2	Future Work.....	117
6.2.1	Initial Transient and Error Reduction .....	117
6.2.2	Filter Reset Feedback.....	118
6.2.3	Sensor Placement and Lever Arm Sensitivity .....	118
6.3	Additional Technology Applications.....	119
6.3.1	Foot soldier HMD and Fire Control System.....	119
6.3.2	Tether Control.....	122
	Space Tethers .....	122
	Helicopter Tethers.....	122
6.3.3	Generic Application .....	122
References.....		125
<b>Appendix A</b>	<b>Measurement Matrices .....</b>	<b>127</b>
<b>Appendix B</b>	<b>Additional Monte Carlo Plots.....</b>	<b>129</b>
<b>Appendix C</b>	<b>Acronyms .....</b>	<b>135</b>

**Appendix D** Pilot Model Code.....137  
**Appendix E** Navigation Filter C Code.....151  
**Appendix F** Navigation Filter MATLAB Code.....187

# List of Figures

Figure 1.1 Polhemus IsoTrack II Magnetic Tracker .....	16
Figure 1.2 AH-64 Apache IHADSS Components .....	17
Figure 1.3 Logitech Head Tracker .....	18
Figure 1.4 The InterSense IS-300 InertiaCube™ System .....	19
Figure 1.5 The InterSense IS-600 InertiaCube™ and SoniDisc System .....	19
Figure 1.6 AH-64 and RAH-66 HMCS Helmets shown with HMDs .....	23
Figure 1.7 The VSI Joint Head Mounted Cueing System .....	23
Figure 1.8 MEMS Accelerometer Diagram.....	29
Figure 1.9 MEMS Gyroscope Diagram.....	30
Figure 1.10 the Draper MMISA utilized in CMATD with gyroscope enlarged.....	31
Figure 1.11 Simplified Relative IMU Geometry and Dynamics .....	35
Figure 1.12 Distributed in-plane sensor mounting example.....	37
Figure 1.13 Distributed Out-of-Plane Sensor Scheme (Intersecting, Centered Axes) ..	38
Figure 1.14 Co-located Mounting Examples (normal, intersecting sensor axes, offset)	39
Figure 1.15 MEMS instrument packaging for co-located placement.....	39
Figure 1.16 Overall system Hardware Concept.....	41
Figure 2.1 The Earth-Centered Earth-Fixed Navigation Frame .....	43
Figure 2.2 Body Axes Superimposed on image of T-38 .....	44
Figure 2.3 The Helmet Frame superimposed on image of JHMCS.....	45
Figure 2.4 Illustration of Relative Frame Dynamics and Position Vectors .....	46
Figure 2.5 Visualization of the Quaternion.....	55
Figure 2.6 Head and Torso Frames in Neutral Position Relative to the Ejection Seat ...	58
Figure 3.1 Simulation Module Architecture .....	61
Figure 3.2 The C-sim Framework.....	63
Figure 3.3 Joystick Geometry for Control of Pilot .....	70
Figure 3.4 Pilot Attitude, Angular Velocity and Angular Acceleration Generation .....	71
Figure 3.5 Author Running C-sim on an Onyx 2 with the Double Fly Panel.....	74
Figure 4.1 Navigation System Architecture.....	80
Figure 4.2 Kalman Filter Propagation and Update Timing Diagram [8].....	91
Figure 4.3 Generalized N-Navigator PHI Matrix .....	97
Figure 4.4 Generic Markov Process: $\tau = 10$ , $\sigma = 1$ , $E[x] = 0$ .....	100
Figure 4.5 Tau-constant and Tau-varying Markov Processes .....	102
Figure 5.1 Single Run T-38 and Helmet Navigation Errors .....	105
Figure 5.2 Single Run P-Matrix Attitude Calculated Standard Deviation.....	106
Figure 5.3 Monte Carlo Sample Statistics for Filter Markov $\tau = 10$ seconds.....	108
Figure 5.4 Monte Carlo sample RSS for filter Markov $\tau = 10$ seconds .....	109
Figure 5.5 Final Attitude Sigma vs Markov Time Constant.....	112
Figure 5.6 Maximum Attitude Error Sigma after settling vs. Markov Time Constant.	113
Figure 5.7 Maximum RSS of Attitude Error after settling vs. Markov Time Constant	114
Figure 5.8 Monte Carlo s Results of Azimuth Accuracy, $\tau$ varying.....	116
Figure 5.9 Monte Carlo RSS Results of Azimuth Accuracy, $\tau$ varying .....	116
Figure 6.1 Soldier Helmet with HMIMU Technology Applied to Ground Maneuvers	120
Figure 6.2 Soldier's Eye View of HMIMU used with HMD for Ground Forces.....	121

Figure B.1: Monte Carlo s Results of Azimuth Accuracy, $\tau = 5$ .....	129
Figure B.2: Monte Carlo RSS Results of Azimuth Accuracy, $\tau = 5$ .....	130
Figure B.3: Monte Carlo s Results of Azimuth Accuracy, $\tau = 50$ .....	131
Figure B.4: Monte Carlo RSS Results of Azimuth Accuracy, $\tau = 50$ .....	132
Figure B.5: Monte Carlo s Results of Azimuth Accuracy, $\tau = 100$ .....	133
Figure B.6: Monte Carlo RSS Results of Azimuth Accuracy, $\tau = 100$ .....	134

## List of Tables

Table 1.1	Draper MEMS Instrument Errors (one sigma) .....	33
Table 2.1	Interpolated Quantities in Force Coefficient Calculations .....	51
Table 2.2	Moment Coefficient Linear Coefficient Components .....	52
Table 2.3	T-38 Minimum (Empty) Moments of Inertia .....	53
Table 3.1	Simulation Modules .....	60
Table 3.2	GPS corruption parameters for simulated measurements.....	72
Table 3.3	Generic EGI Instrument Errors (one sigma).....	73
Table 3.4	BG Systems custom hardware panel .....	74
Table 4.1	Symbol Definitions for Figure 4.1 .....	81
Table 4.2	Generic Navigation Error States .....	92
Table 5.1	Summarized Sample Statistics Data at tfinal.....	110



# Chapter 1.0

## Introduction and Background

### 1.1 Helmet Mounted Cueing Systems

The purpose of a head tracker or Head Mounted Cueing System (HMCS) is to detect orientation and position of the user's head relative to his surroundings. Applications range from virtual reality systems to fire control system interfaces, allowing visual feedback via an "enhanced reality" Head Mounted Display (HMD).

#### 1.1.1 Overview of Currently Available Systems

Typically, HMCS systems consist of a relayed or transmitted signal between three or more sensors on the helmet and a group of emitters in the cockpit. These sensors and emitters rely on various ranging schemes, some of which use radio frequencies, optical, or ultra sonic (pressure wave) methods in various geometric configurations. One thing common to all of the aforementioned schemes is the involvement of emitters and sensors in a configuration such that there is a signal corresponding to each degree of freedom, propagating through the cockpit environment. For a more technical overview, see Velger's "Helmet Mounted Displays and Sights" [21]. The following paragraphs examine five technologies used to implement HMCS systems.

## Magnetic Trackers

Electromagnetic Trackers, such as those commercially available from Polhemus (Figure 1.1), or those designed for the AH-64 Apache (Figure 1.2), create three standing magnetic fields within the cockpit environment, which are mapped for deviations due to presence of metal objects. Sensors on the helmet detect their position and/or orientation by relating received signals, caused by the magnetic field, to the pre-stored magnetic field mapping. This is currently the most commonly used scheme used by the military in their head trackers. However, there are shortcomings of magnetic trackers, which are briefly outlined in Section 1.1.3.



**Figure 1.1** Polhemus IsoTrack II Magnetic Tracker



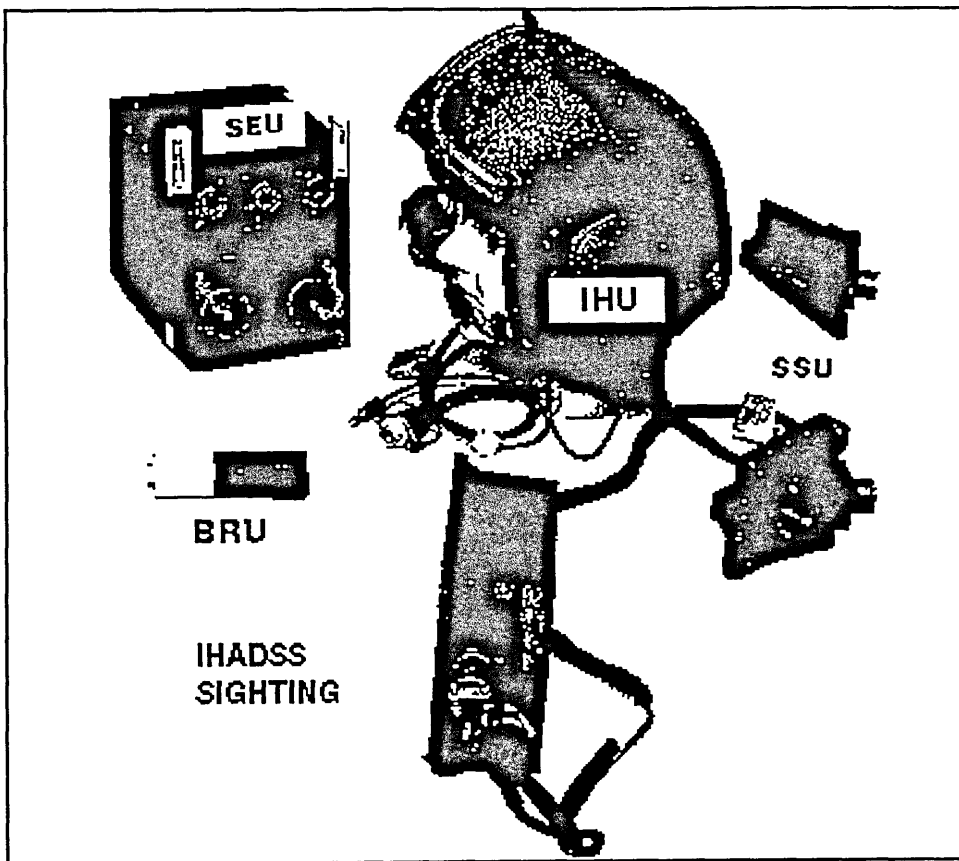
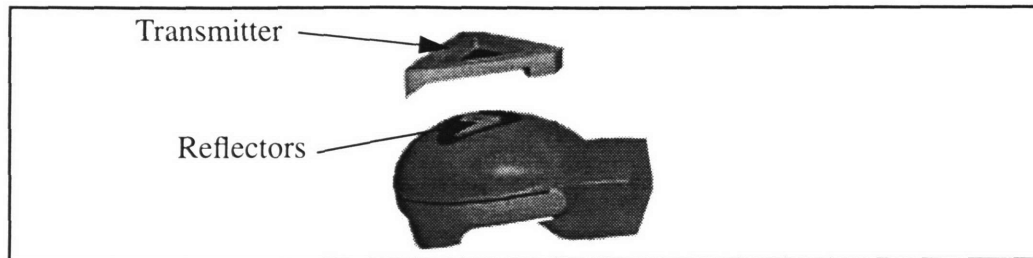


Figure 1.2 AH-64 Apache IHADSS Components

### UltraSonic trackers

Ultrasonic sensors, such as the standalone head tracker produced by Logitech (Figure 1.3) and the position measurement in the IS-600 by InterSense (Figure 1.5) for virtual reality applications, use three ultrasonic emitters and reflectors to detect position and orientation of the head. These types of sensors are not very applicable for a combat environment, as the reflectors are easily masked by extreme positioning of a pilot's head and the system can lose track during rapid movements.



**Figure 1.3** Logitech Head Tracker

### **Optical Trackers**

Optical sensors, such as those being investigated for ground vehicle head tracker, use either optical or infrared LEDs and a CCD detector to provide measurement updates to a Kalman filter. These sensors can be masked (similar to the aforementioned problems with ultrasonic trackers) and can lose tracking accuracy under rapid head motion, and their accuracy is highly dependent on camera stability, focus, and resolution. Alternate optical sensors have been proposed and tested, such as the one found in Kim, Richards, and Caudell [11]. Another publication with more detail about optical sensors is Velger's "Helmet Mounted Displays and Sights," [21] pp. 155-165.

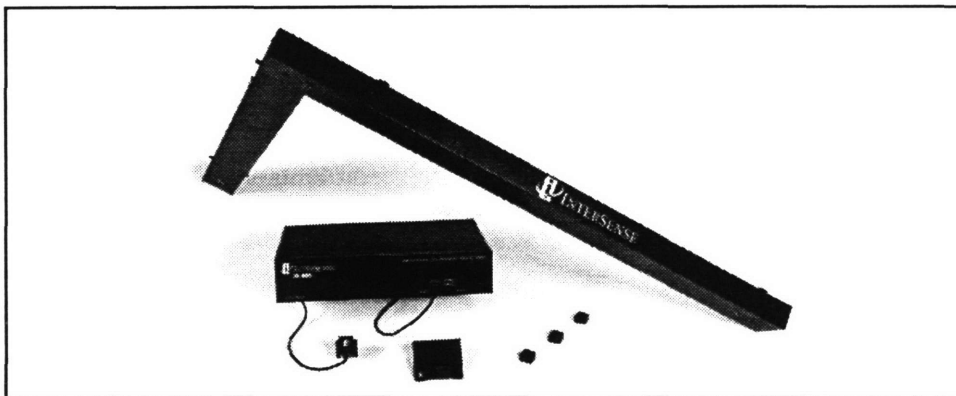
### **Inertial Trackers**

Inertial trackers, such as the one developed by InterSense (Figure 1.4 and Figure 1.5) in Burlington, MA, overcome the inherent masking and high-dynamics-tracking problems relating to signal transmission between sensors and emitters to detect head position. However, inertial sensor-based navigation solutions drift over time, and require periodic updates from other sensors. The InertiaCube™ from InterSense uses magnetometers and gravimeters to bound

the drift of the gyros and accelerometers, however both of those update sensors have large time constants relative to the inertial instruments. Therefore, to overcome those large time constants, the most accurate systems from InterSense (IS-600 series) use additional ultrasonic updates from their own ultrasonic position tracker to form a hybrid system.



**Figure 1.4** The InterSense IS-300 InertiaCube™ System



**Figure 1.5** The InterSense IS-600 InertiaCube™ and SoniDisc System

## Hybrid Systems

As mentioned in conjunction with the InterSense InertiaCube™ IS-600 (see Figure 1.5), many of the aforementioned schemes can be combined using one or more Kalman Filters to increase the overall confidence and accuracy of the position and orientation estimate generated by the head tracker. However, as the filter begins incorporating more measurements and states, the system requires a faster computer to generate the head state estimate. Therefore, the best system will incorporate a minimal number of measurements and the smallest allowable state vector to increase the performance of the computer dedicated to the filter of the head tracker.

### 1.1.2 Current Applications of Head Trackers

The aerospace research, simulation, entertainment, and military industries all have used head trackers for both research and in applications. In the aerospace research venue, a head tracker system can provide valuable data for the positioning of the head for motion perception experiments which desire to achieve data about pilot head motion in a dynamic environment. The simulation and entertainment venues both use head trackers for measurements of head position to display graphics to the user of a head mounted display, generating a virtual three-dimensional environment. Military applications are primarily concerned with detecting and providing head orientation and/or position to drive symbol generation for a helmet-mounted Heads Up Display (HUD) or Helmet-Mounted Display (HMD), resulting in an improved interface to both navigation and fire control systems. These devices are dubbed Head Mounted Cueing Sys-

tems (HMCSs) incorporating a head tracker with tighter pointing accuracy requirements, within the dynamic environment of a combat flight or ground vehicle.

### **Virtual Reality Applications**

The simulation and entertainment fields are rapidly integrating virtual environments into new and innovative applications. These environments aim to place the user in a believable environment generated by a computer and presented via a head mounted display. For user head motion to be an input source for determining the line of sight in the virtual world, the head must be tracked in orientation and/or position (depending on the simulation). Virtual Reality (VR) head tracking schemes are usually developed with the goal of reduced data latency (i.e. time lags caused by the necessary, sophisticated graphical computation) and minimizing visual artifacts such as slosh (i.e. unwanted roll or pitch due to improperly interpreted translational motion) or jitter (i.e. optical flow that is not smooth).

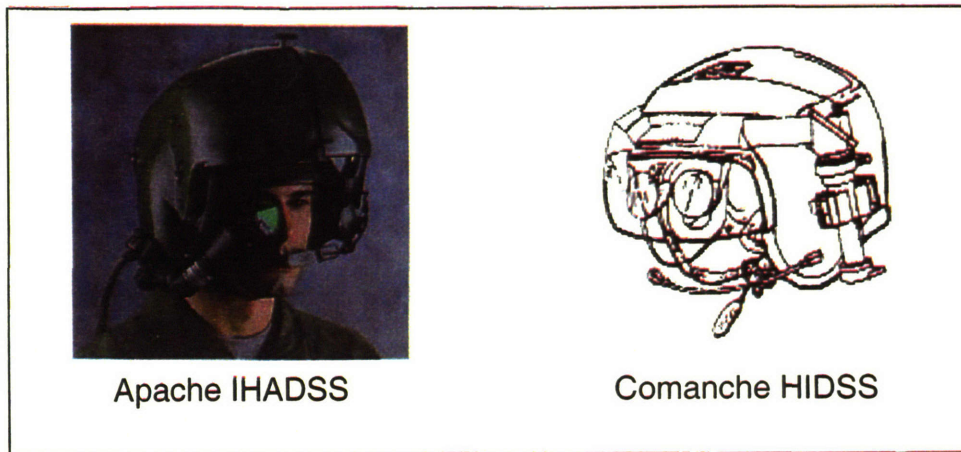
The traditional means of head tracking for VR applications uses inclinometers and magnetic compasses because of their simplicity and low cost. However these instruments usually introduce noticeable lags into the graphics display, which can cause the user to become nauseated. Therefore, InterSense has developed head trackers which use low-cost Coriolis gyroscopes and solid-state accelerometers in conjunction with magnetometers and acoustic sensors (i.e. the IS-300, IS-300 Pro, and IS-600 -- see Figure 1.4 and Figure 1.5). The

gyros and accelerometers allow high (>100 Hz) update rates to the simulation, while the magnetometers and acoustic sensors bound the drift caused by the inexpensive low-quality inertial components [7].

### **Helicopter and Jet Aircraft Applications**

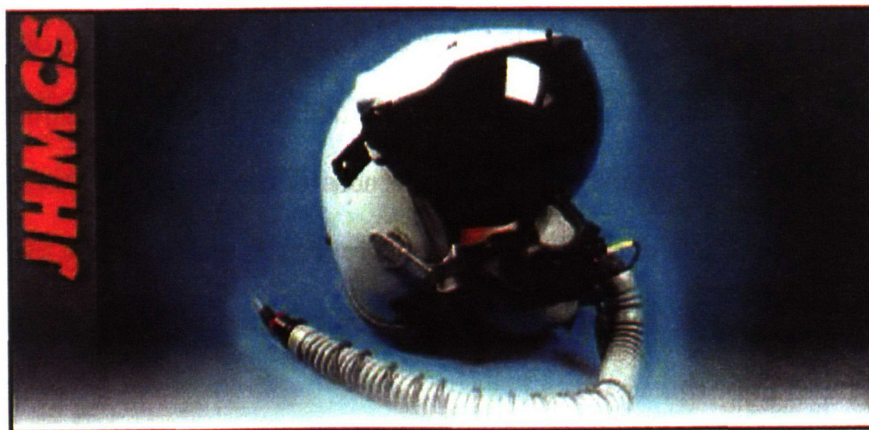
The application of a head tracker in a flight vehicle is somewhat different than the VR application, although many of the design issues are similar. The goal of a head tracker in a flight vehicle is to provide an HMD symbol generator with the proper line of sight data so that graphics can be overlaid on the pilot's view. System goals include better situation awareness, provision of navigational aids, and aided target acquisition for effective weapons deployment. Current HUD technology does all of the aforementioned tasks, but only in a fixed, small angular region about the boresight of the aircraft. HMDs and their associated head trackers drastically increase the area over which these tasks can be performed in modern jet and rotor aircraft.

Presently, the most common means of head tracking in the flight environment is the magnetic tracker. Actual integrated applications of a magnetic HMCS in combat vehicles can be found in the AH-64 Apache and the RAH-66 Comanche helicopters (see Figure 1.6), as well as tests performed by the RAF of off-boresight target acquisition systems using Jaguar and Hawk jet fighters equipped with AIM 9L Sidewinder missiles [5].



**Figure 1.6** AH-64 and RAH-66 HMCS Helmets shown with HMDs

Similar systems are currently under development for additional jet aircraft (the eventual target application of this study), as well as ground vehicles and troops [14]. The Joint Helmet Mounted Cueing System (JHMCS) is a system, currently being developed by Vision Systems International (VSI), which is intended for the F-15, F-16, F/A-18 and F-22 aircraft (See Figure 1.7) [13].



**Figure 1.7** The VSI Joint Head Mounted Cueing System

### **Foot Soldier Applications**

The Boeing Company has been working on the Integrated Maintenance and Logistics Soldier System (IMLSS) which is essentially a wearable computer for a foot soldier that implements an HMD, but no head tracker [4]. Either this particular system, or one similar to it, could be fitted with a miniature inertial system, similar to the one analyzed in this thesis. Its computer could be provided with the appropriate algorithms to provide similar functions (navigation, mission information, and target designation/tracking) as the aforementioned pilots' helmets. However, it would use GPS as a source of measurement updates to the error estimation filter. For a conceptual diagram of such a system, see Figure 6.1 and Figure 6.2.

### **1.1.3 Magnetic Tracking Issues**

The aircraft systems mentioned in Section 1.1.2 (IHADSS, HIDSS, and JHMCS) all utilize magnetic trackers to measure the helmet's orientation and position within the cockpit. This kind of head tracker requires that three mutually-perpendicular magnetic fields be generated within the cockpit and mapped. The mapping is necessary for the computer to interpret the readings from the sensors on the helmet during use, and must be done for each individual cockpit—a process that is simultaneously lengthy and tedious [21]. In addition to the long calibration procedure, the mapped magnetic field experiences distortion in the vicinity of metal objects not present during the mapping procedure, and accuracy also decreases with distance from the emitter. Other electronics may also interfere with the field, causing additional inaccuracy [21].

There is also an additional emerging issue surrounding the use of magnetic trackers: prolonged head exposure to strong magnetic fields. There is no litera-



ture stating whether power levels emitted by a magnetic tracker would be large enough to be detrimental to a pilot's health when frequently exposed to the magnetic tracker environment, but by removing the cause of the concern for the pilot's health (the standing magnetic field) this issue can be avoided entirely. Present concerns surrounding cell phone use point to a correlation of reduction in short term memory, as well as an increased tumor probability when exposed to EM radiation from an emitter in close proximity to the human head, as are most cockpit magnetic field emitters (mounted on or near the ejection seat).

As of June, 1999, Honeywell had successfully demonstrated its Advanced Metal Tolerant Tracker (AMTT) system on both the Apache Longbow helicopter and the F-16 Vesta testbed aircraft, and is already in production for the U.K. and Oman Jaguar fleets. The AMTT has the following capabilities:

... track head motion up to 180 deg. on either side and 90 deg. in elevation. It can also accommodate aircraft roll up to 180 deg. in either direction. Line of sight accuracy is less than 4 milliradians, averaged over the full motion box, in the Jaguar cockpit. Update rate is 240 Hz. for a single pilot aircraft, or 120Hz for dual-crew platforms using two helmet sights. Signal latency is 4.17 millisecc. or less... The system is also tolerant of high metal content of rotorcraft cockpits and ground combat-vehicle crew positions. [17]

### **Comparative Reasoning for a Better Inertial HMCS**

The main purpose for developing a better inertial head tracker is two-fold. First, it will allow an alternative tracking scheme to magnetic tracking, and will decrease the complexity of the hardware required to install and maintain by eliminating the transmitter and dependency on a mapped magnetic field in the

cockpit environment. Second, an inertial head tracker may be configured to be used with GPS, which could free the head tracker from its cockpit or virtual reality arenas. For example, a head mounted IMU system updated with GPS and integrated with an HMD could provide ground soldiers with the same navigational and targeting aids as vehicle operators. Metaphorically speaking, the inertial tracker removes the chains imposed by the magnetic tracker system in the form of the standing magnetic field emitter mounted to the surrounding environment.

#### **1.1.4 Motivation**

##### **Developing an Inertial HMCS (IHMCS) for the Flight Environment**

It might seem obvious to apply the existing inertial head trackers developed by InterSense for VR applications to HMDs in the flight environment. However upon closer inspection, the InterSense design relies upon particular dynamic nuances of head tracking for VR applications and would not be well suited for a flight environment for the following reasons:

1. The IS-300 (when used alone) relies on gravimetric measurements from the accelerometers to bound the gyro drift in roll and pitch, and magnetometer measurements to bound the gyro drift in yaw. This is under the assumption that Earth's gravity will always be the specific force vector of interest, which in the flight regime is a poor assumption, especially in combat situations. Furthermore, this arrangement only gives orientation estimates and not position, as well.
2. For both position and orientation, the additional acoustic ranging system is required (what InterSense markets as the IS-600). The goal of the IHMCS would be to eliminate dependency on any system that relies on transmitting signals through the cockpit, or one which has a limited range of motion.

3. The IS-300 gyro in-run bias stability rates are too large when there are no position update measurements available. Higher-grade MEMS sensors are desirable in the event of a long period without measurement updates to minimize the accumulated error in position and orientation.
4. The IS-300 has a dynamic accuracy of 3 degrees, which is much larger than AMTT's accuracy requirement of 4 milliradians, or 0.23 degrees [17].

The primary benefit of using an inertial tracker for head motion, as opposed to either visual, ultrasonic, or magnetic sensors is that the sensor components are self-contained with no reliance on optical, RF, acoustic wave, or mapped magnetic field signals transmitted/received within the cockpit. The system can be small, lightweight, and does not potentially obstruct integration with existing HMD systems. The HMIMU's low weight and volume are two additional benefits relative to pilot safety during ejection, which are driven by the desire to minimize loading (especially loads generated from a displaced helmet center of gravity), and wind blast on the head.

## **1.2 Draper Micromechanical Inertial Sensors**

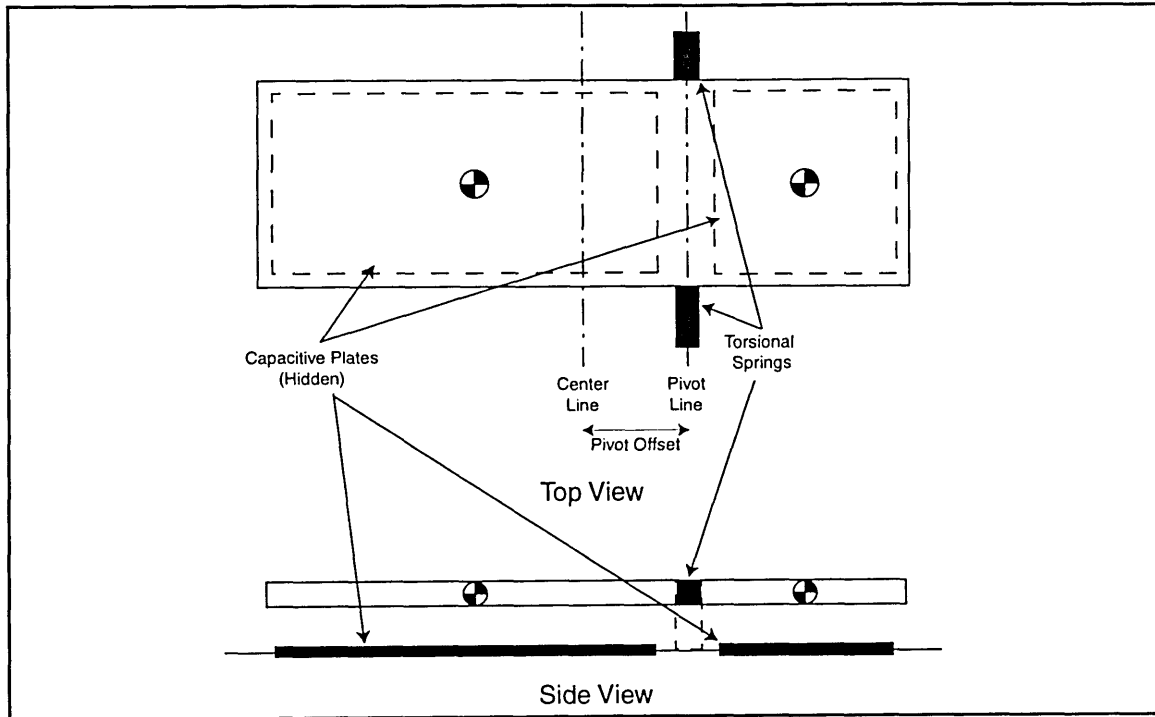
The sensors that will be modelled and investigated for use in the HMIMU are the MEMS gyroscopes and accelerometers developed at the Charles Stark Draper Laboratory. Initially designed to be low-cost, expendable inertial sensors for munitions, these sensors have the desirable characteristics of being small (each first generation sensor is 3cm square, in flat packaging with the necessary supporting ASICs [12]) and lightweight (approximately 10 grams per sensor per axis, totalling 60 grams for all sensors), which are important to head mounted systems [16].

### **1.2.1 Technology**

Both the MEMS accelerometer and gyroscope are manufactured from silicon using a dissolved wafer process, featuring silicon bonded to a glass substrate. The die size of the sensors are approximately 3 mm<sup>2</sup>.

#### **MEMS Accelerometers**

The MEMS Accelerometer is a pendulous mass, shaped like a flat plate attached to a torsional spring which is offset from the center of mass of the plate. It behaves like a mass attached to a clamped horizontal flexure. The need for the mass to be a flat plate is derived from the need to measure the distance between the plate and the substrate on which it is mounted via capacitive plates beneath the mass (see Figure 1.8). Therefore, the sensing axis is normal to the MEMS substrate plane since the torsional spring about which the pendulous mass rotates is parallel to the substrate plane. The MEMS Accelerometer is illustrated in Figure 1.8



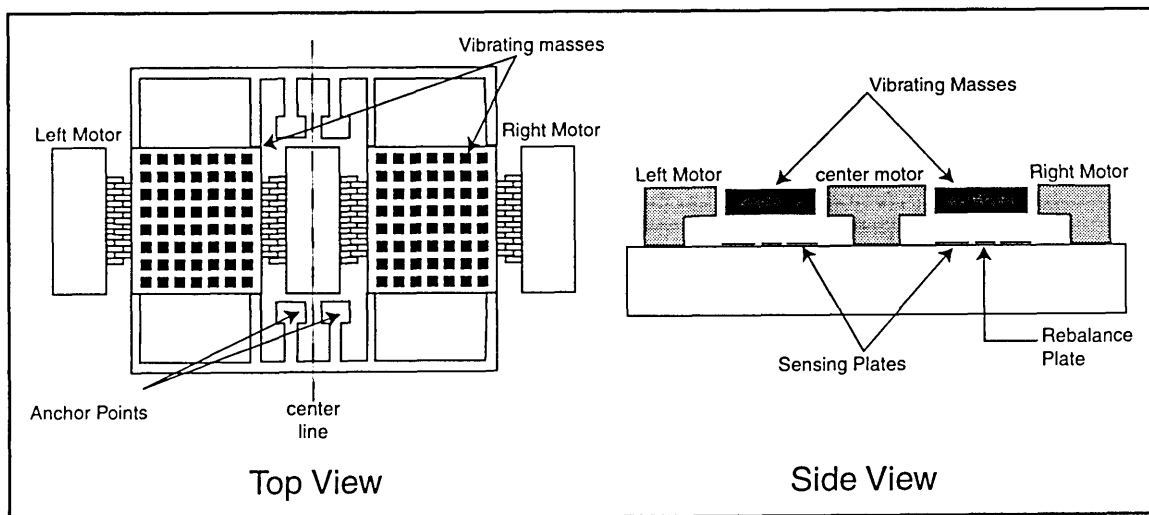
**Figure 1.8** MEMS Accelerometer Diagram

In-run performance is defined as the standard deviation of the residual error of a compensation model fit to the accelerometer data acquired during a thermal calibration. Present bias in-run stability performance (see Table 1.1) of the second generation accelerometers is at the 0.1 to 2.0 mg level, across a temperature range of -40 Celsius to +85 Celsius. This stability has been demonstrated over several days, with in-run scale factors from 30 to 160 ppm.

### **MEMS Gyroscopes**

The theory of operation for the gyroscope is that two masses, suspended by a sequence of beams anchored to the substrate, are vibrated electrostatically within the plane of the device. When an angular rate is applied about the input axis, perpendicular to the velocity vector of the masses, a Coriolis force pushes

the masses in and out of the plane of oscillation in an anti parallel manner. The resultant motion is measured by capacitive plates beneath the masses, providing a signal proportional to the angular rate input. A diagram of the Draper Laboratory MEMS gyro can be seen in Figure 1.9. A magnified photo of the sensor can be seen in Figure 1.10.



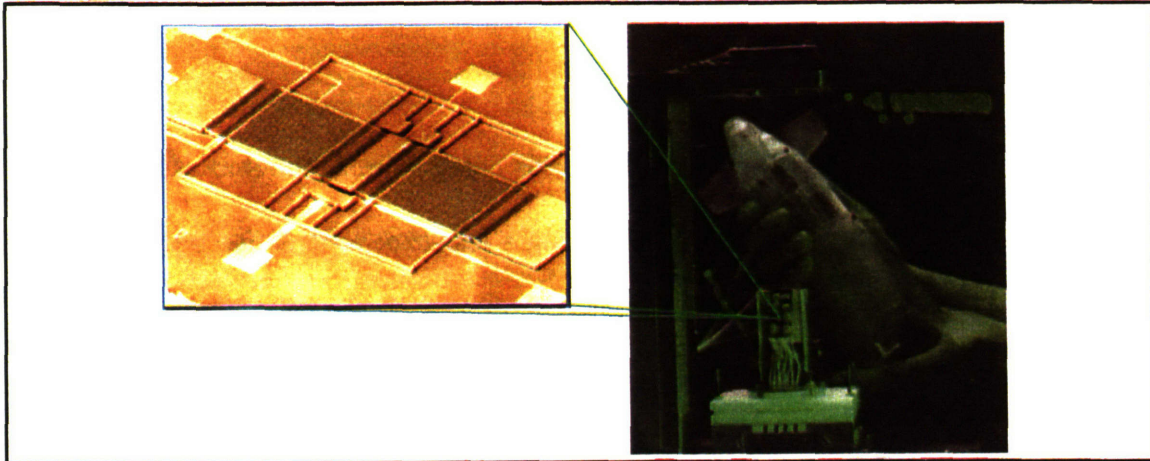
**Figure 1.9** MEMS Gyroscope Diagram

Present performance of in-run bias and scale factor stability range from 3 to 10 per hour and 30 to 100 ppm, respectively. Additional performance figures can be found in Table 1.1.

### **1.2.2 First Generation MMISA Application: The Competent Munitions Advanced Technology Demonstrator (CMATD)**

CMATD is a program which developed a replacement fuse for existing Naval gun-launched munitions. The CMATD Guidance Navigation & Control (GN&C) assembly contains a full six-degree of freedom (6-DoF) MEMS iner-

tial unit, GPS receiver, and Flight computer, packaged into eight cubic inches (see Figure 1.10). This electronics and sensor unit was designed to withstand 16,000 g's prior to activation and only produce 0.6 mrad of misalignment due to the extreme acceleration pulse generated upon firing the munition.



**Figure 1.10** the Draper MMISA utilized in CMATD with gyroscope enlarged

### **1.2.3 Sensor Applicability**

The Draper MMISA first generation MEMS IMU fits within a total of eight cubic inches, and the nearly completed second generation will fit within three cubic inches. Both the first and second generation systems have separate sensors for each axis. All gyros have input axes in the plane of the micro-machined sensor, and all accelerometers have input axes orthogonal to the plane of the sensor. The system has demonstrated drift rates better by an order of magnitude than the InertiaCube™ and does not need the magnetometers along each axis if it is used in conjunction with a higher-resolution measurement source such as the dual IMU solution suggested in this study. In addition, future generations of the Draper MMISA will fit all three axes onto a single chip as the ability to detect specific

force in the plane of the chip and angular rotation about a perpendicular axis to the chip become more refined. The light weight, low volume, and high accuracy of these MEMS components, relative to their counterparts in industry, makes them a primary candidate for application to head tracking in the flight vehicle environment.

## **1.2.4 Technology Road map**

### **First Generation MMISA Sensors**

The first generation Draper MEMS gyros being manufactured as of September, 1998 were being manufactured using a less-expensive 3-step process and had achieved a nominal 150deg/h turn-on to turn-on and 30 deg/h in-run bias stabilities within a 100 Hz operating bandwidth [12]. Present performance of the CMATD system is at the 10 to 30 deg/h level for the gyros, and the 1 to 7 mg level for the accelerometers across a temperature range of -40 to +85 degrees Celsius. This level of performance is the overall “inertial only” stability one could expect from the integrated system, should no measurement updates be available after initialization and calibration.

### **Draper MEMS: The Next Generation**

The current MEMS gyros and accelerometers are only the present step along the path to more accurate miniature inertial sensors achievable in the future. Like any development program, there are next generation sensors in development, whose errors can be seen compared to the current generation’s errors in Table 1.1.



<b>Instrument Error</b>	<b>Units</b>	<b>1st Generation System</b>	<b>2nd Generation System</b>
<b>Gyro</b>			
Bias Turn-on Repeatability	deg/hr	50-150	10-30
Bias In-run Stability	deg/hr	10-30	3-10
Scale Factor Turn-on Repeatability	ppm	300-1000	100-300
Scale Factor In-run Stability	ppm	100-300	30-100
Axis Misalignment	mrad	1	1
IA Repeatability	mrad	0.2	0.2
Maximum Input	deg/s	1000	1000
Bandwidth	Hz	100	<500
Angle Random Walk	deg/rt(hr)	0.15-0.30	0.03
<b>Accelerometer</b>			
Bias Turn-On Repeatability	mg	5-10	0.5-5
Bias In-Run Stability	mg	1-7	0.1-2
Scale Factor Turn-on Repeatability	ppm	500-1400	100-300
Scale Factor In-run Stability	ppm	300-500	30-160
Axis Misalignment	mrad	1	1
IA Repeatability	mrad	0.2	0.2
Maximum Input	#g's	15	30
Bandwidth	Hz	100	<500
Velocity Random Walk	cm/s/rt(hr)	33	1

**Table 1.1 Draper MEMS Instrument Errors (one sigma)**

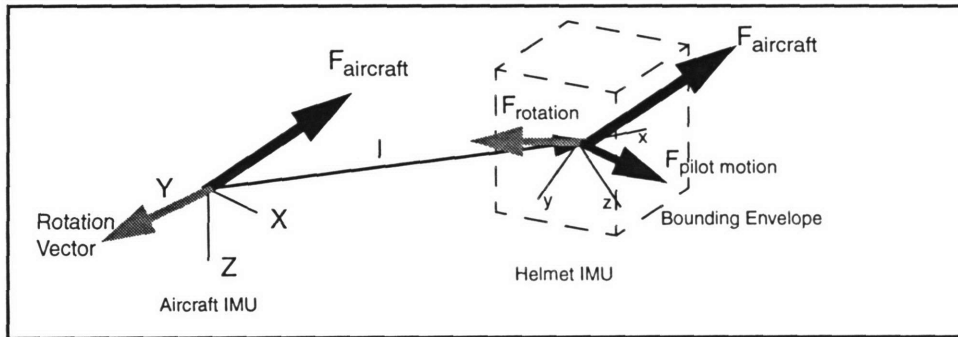
### **1.3 Helmet Mounted Inertial System Concept**

The primary focus of this study is the helmet-mounted inertial measurement unit (HMIMU), which is a full six-degree of freedom sensor array (consisting of three

gyros and three accelerometers) that is mounted to the helmet of the user for the purpose of tracking head motion within a cockpit environment. Three orthogonally oriented micromechanical tuning-fork gyroscopes and three orthogonally micro-mechanical accelerometers are mounted to or embedded within the helmet of a pilot to detect both orientation and position of the head in three dimensions. Inertial sensors, when used for vehicle navigation, traditionally utilize measurements from additional sensors and a Kalman Filter to estimate errors in both the inertial sensors and the navigation solution. These navigation measurements can come from a radio navigation system (which provides a position and/or velocity update) such as GPS, TACAN, VORTAC, VOR/DME, or LORAN, or they can be acceleration and orientation signals from another IMU. If the system is a cascaded IMU system, the relative positioning between the helmet-mounted IMU and the vehicle IMU must be either observable or stochastically determined. Any of the other sensor schemes outlined in Section 1.1.1 would also suffice as a measurement input to the Kalman Filter. To be a viable system, any of these cases must be able to accurately provide the angle between the head and the aircraft body for effective targeting hand off to the selected weapon system (missile, LASER, chain gun, etc.).

This study investigates a vehicle IMU and a helmet IMU with GPS to determine the orientation of a pilot's head with respect to an aircraft's body axes. Linear accelerations and angular velocities within the coordinate system are detected by the helmet IMU, and are compared to the aircraft IMU via a Kalman Filter which relates the position of the helmet to the position of the aircraft IMU via Markov processes representing displacement along each body axis. The vectors shown represent the specific force detected by both of the IMUs

(which is also mathematically detailed in Section 2.2). Note the additional component in the head IMU due to vehicle rotation (which can be sensed by the Aircraft IMU's gyros).



**Figure 1.11** Simplified Relative IMU Geometry and Dynamics

By removing the apparent force due to rotation at a lever arm, matching the orientation of the common components of these vectors in three-dimensional space, the relative orientations of the two IMUs can be determined. Such IMU transfer alignments have been successfully designed, tested, and implemented on various weapons systems, “using either natural or deliberately induced maneuvers of the vehicle” [10] to match the orientation and acceleration of the weapon’s IMU to the vehicle’s IMU. Essentially, the HMIMU must have algorithms that preferably use natural dynamics and *a priori* information to bound the drift that will inevitably occur. A more detailed discussion of the relative components can be found in Section 2.2.

### 1.3.1 Hardware

The HMCS system will need four key components:

1. Helmet-mounted IMU (Draper MMISA)
2. Vehicle-mounted IMU which utilizes precision gyros such as Ring Laser Gyros (RLGs) or Fiber Optic Gyros (FOGs) and accelerometers
3. GPS receiver for updating the Vehicle-mounted IMU
4. Processing unit (and associated software) for the Kalman Filter and other necessary algorithms

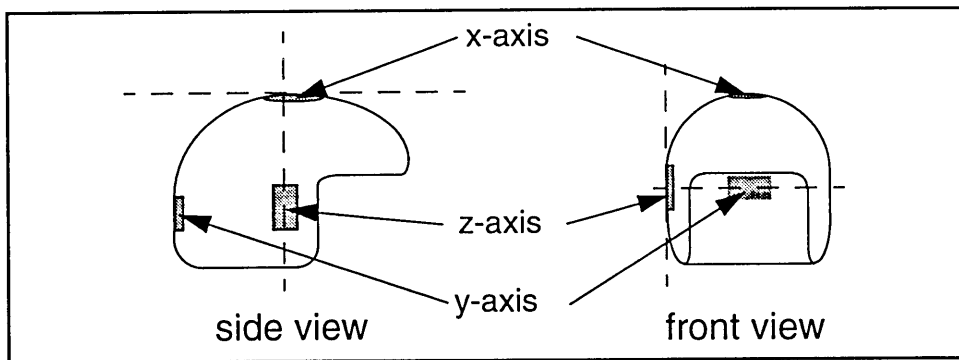
A Draper MMISA will be mounted on the helmet for detecting pilot head motions. This system will consist of a full 6-DoF array (angular rate and specific force in all three spacial dimensions), with three MEMS solid-state silicon tuning fork gyroscopes, three MEMS solid-state silicon accelerometers, and the associated ASICs electronics to support the sensors. Currently, one sensor with its ASICs is three square centimeters in a flat package [12], making them small enough to either distribute easily, or co-locate in one sensor mounting package.

### **Placement of Accelerometers**

Of the innumerable possible sensor mounting configurations, two primary mounting schemes appear practical. The first is a distributed scheme, where the center of rotation of the entire sensor package more closely matches that of the pilot's head. This particular configuration results in different moment arms from the head's center of rotation to each accelerometer, which must be known in order for the additional acceleration terms caused by those lever arms during angular rotation of the head. These lever arms may vary from unit to unit in the case of personalized fitted pilot helmets, and must be used to account for additional rotational dynamics terms within the head frame. The second scheme is

a co-located configuration, which attempts to make all of the lever arms within the head frame as small and similar as possible.

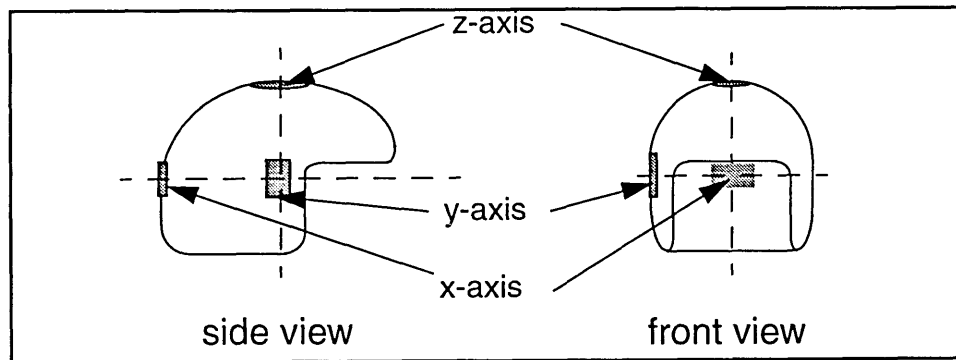
Two variations of the distributed scheme exist: one using in-plane sensor technology which currently exists, and one using out-of-plane sensor technology which is currently in development at Draper. The in-plane sensor technology poses an interesting mounting dilemma, since the sensing axis is confined to the plane in which the sensor is manufactured on the silicon wafer. This means that the MEMS accelerometers must be mounted such that the sensing axis is normal to the surface of the helmet to align the sensing axis with the helmet strap-down navigation axes. With the existing packaging, that would leave protrusions at three separate locations on the helmet, which would make those sensors prone to being damaged.



**Figure 1.12** Distributed in-plane sensor mounting example

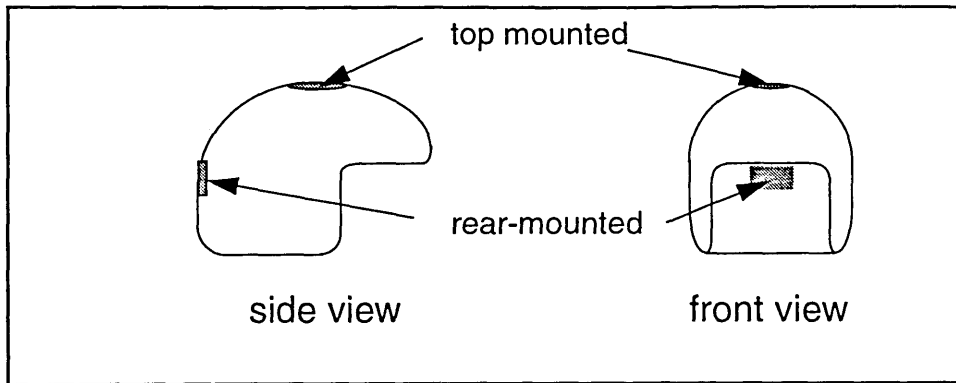
The more desirable alternative is the out-of-plane sensor technology, where the sensing axis is normal to the silicon wafer plane. With the existing packaging, that would allow the ASICs package to be mounted tangent to the surface of

the helmet at the normal, minimizing the protrusion and hence, the potential for damage. Note that the above argument assumes that adding thickness to the helmet to accommodate for the in-plane sensors' packaging footprint is not an option, since a standard lightweight helmet under high g-loading can have an apparent weight of multiple hundreds of pounds [16].



**Figure 1.13** Distributed Out-of-Plane Sensor Scheme (Intersecting, Centered Axes)

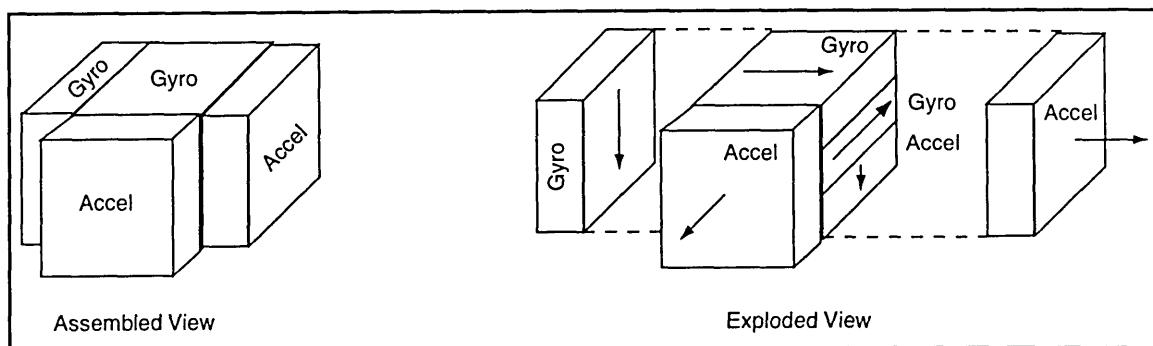
If the sensors were to be co-located, the moment arms to either the center or a mounting point of the sensor package are measured prior to mounting, and then the mounting point on the helmet is related to the head's center of rotation. In this configuration, the angular rotation effects can be made similar to one another, varying only by the inability to mount all the sensors at the same point in space, as opposed to varying locations on the helmet each with their own lever arms. This co-location scheme could be located anywhere on the helmet, but would most likely be mounted on the top of the helmet, to reduce the probability of cockpit or ejector seat impact on the sensor array assembly.



**Figure 1.14** Co-located Mounting Examples (normal, intersecting sensor axes, offset)

### Gyroscope Mounting

MEMS gyroscopes are insensitive to moment arms, and can be distributed anywhere on the helmet, provided that their primary sensing axes are aligned parallel with the x, y, and z axes of the helmet. One possible packaging arrangement for all of the MEMS instruments (with their required ASICs) is seen in Figure 1.15.



**Figure 1.15** MEMS instrument packaging for co-located placement

Other miscellaneous hardware on the helmet will consist of the connections and leads to and from the sensors to enable data transfer to the computer, and

for future designs, the computer may be embedded into the helmet as well with the advent of flexible trace boards.

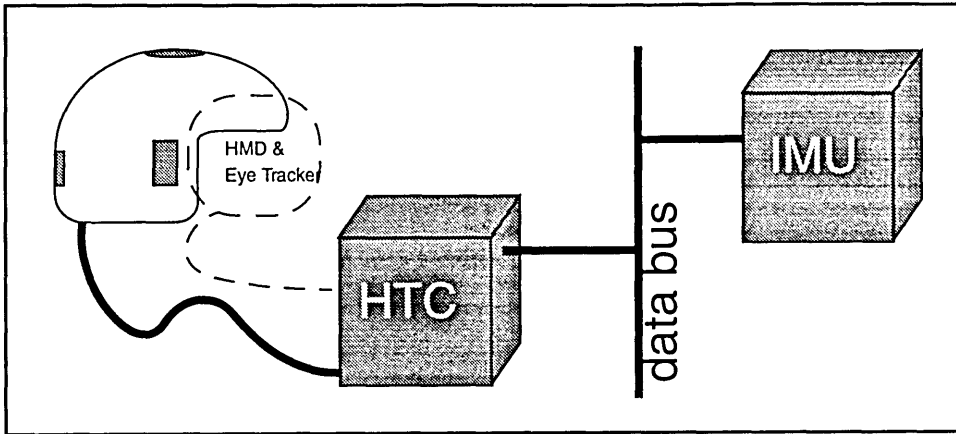
### **Vehicle IMU vs. HMCS IMU**

The vehicle IMU serves two functions: it acts as the primary IMU for the vehicle (it determines the body axes orientation with respect to the inertial, earth-fixed, and local-level frames) and it also acts as the measurement source for the HMCS to determine the HMCS's orientation with respect to the aircraft's body axes. The specific force and rotation vectors from the HMCS are compared to the vehicle IMU's specific force and rotation vectors, and the orientation and dynamics of the pilot's head can be determined therefrom.

### **Head Tracking Computer**

As shown in Figure 1.16, the Head Tracking Computer (HTC) accepts input from both the aircraft IMU, via the aircraft's data bus, and the HMCS. The HTC can also be used for symbology and control of the HMD, so it can also have output(s) to the HMD. For precision pointing, an eye-tracker should be used as well, and that must also output signals to the HTC for processing. For the scope of this thesis, however, the HMD and eye tracker interfaces should be secondary to HMCS MMISA development. The primary focus is on the accuracy of the HMCS MMISA and not its interface into the HMD video system and the eye tracker; however, when it is appropriate the research will point out how an eye tracker would work in consort with the HMCS MMISA.





**Figure 1.16** Overall system Hardware Concept

### 1.3.2 Necessary System Algorithms

#### **Kalman Filter**

For the HMCS to properly estimate the state of the helmet (position, velocity, orientation, and angular velocity) relative to the aircraft, there must be a Kalman Filter in the HTC which takes measurements from both IMUs. The filter allows the cross coupled probabilistic distributions (off-diagonal elements in the covariance matrix) to be implemented in coupling and bounding the two IMUs. In reality, the filter may have to operate in a cascaded mode, with the outputs from the aircraft IMU being already filtered by another navigation computer. This would make computation requirements on the HTC less cumbersome by reducing both dimensions of the state propagation matrices by a factor of approximately two. However, this would most likely introduce artifacts in the estimates of head position and orientation due to the time-varying nature of the measurement variances and covariances, as well as manifested jumps in the actual measurement readings themselves which occur after the

GPS updates on the aircraft IMU. These negative effects are due to the loss of the cross coupling information mentioned above. Therefore, this study will use a larger “dual IMU” state vector of 45 states to take advantage of the benefits of having the off-diagonal terms in the covariance matrix.

### **Bounding Algorithm**

In addition to the Kalman filter, a scheme for varying the time constant of the Markov processes relating the two IMU positions will be developed to ensure that the estimates for head position, velocity, and orientation remain within the realm of feasibility. Essentially, this algorithm modifies the variance of the noise in the Kalman Filter model based on *a priori* information (such as a nominal probabilistic lever arm to the helmet navigator’s allowed dynamic envelope), allowing for standard deviation reduction in the state estimates. For instance, the pilot’s head cannot be situated greater than 90 degrees in either direction relative to his torso’s x-axis (protruding from his chest), and his head cannot be removed from his body or outside the cockpit, or more specifically, from an ellipsoidal-shaped boundary governed by the pilot’s neck motion and spinal physiology. The bounding algorithm will work in parallel with the Kalman Filter, adjusting the measurement noise matrix (R-matrix) and the gains based on optimal control laws using a cost function for the head dynamics which is governed by the bounding envelope. The goal is to have a state estimator with low data latency, a narrow probability distribution of the state estimates, and minimal requirements on the pilot to maintain levels of high accuracy of his helmet-mounted targeting system.

# Chapter 2.0

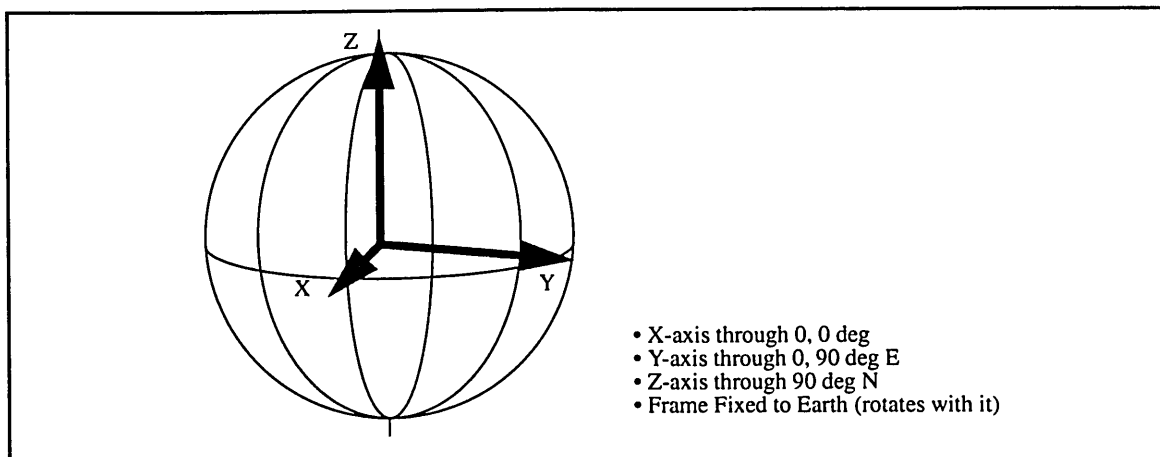
## System Dynamics

### 2.1 Coordinate Frames

The main coordinate frames used in this analysis are the Earth-Centered, Earth Fixed (ECEF) frame, the aircraft (or body) frame, and the head frame.

#### 2.1.1 The ECEF Navigation Frame

The ECEF frame is defined as being a cartesian frame with the origin at the center of the earth, the x-axis through zero latitude and zero longitude, the z-axis through the north pole, and the y-axis mutually orthogonal to both so as to form a right-handed coordinate system (x-axis vector crossed with the y-axis vector equals the z-axis vector) (see Figure 2.1).

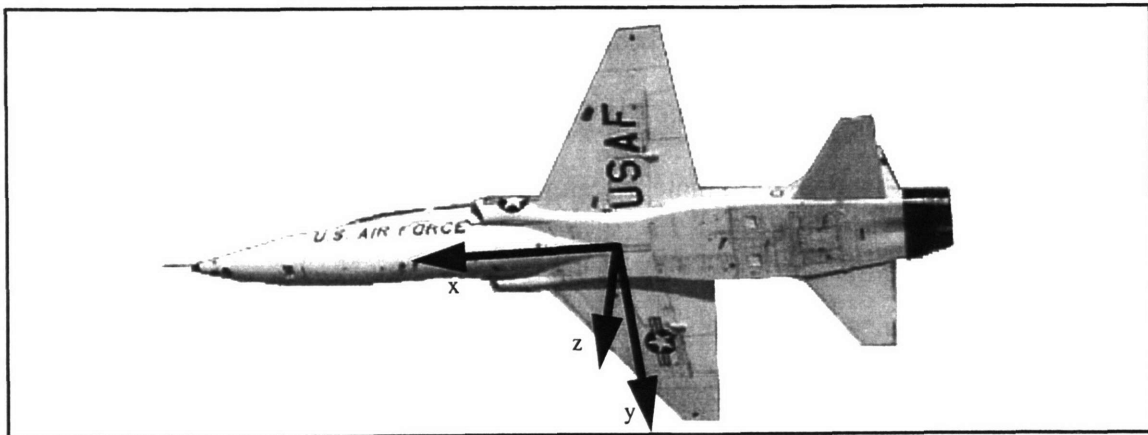


**Figure 2.1** The Earth-Centered Earth-Fixed Navigation Frame

The ECEF frame will serve as the navigation frame, and earth's rotation vector will be defined as being co-linear with the Z-axis of the frame.

### 2.1.2 The Body Frame

The body frame is fixed to the aircraft, which in this case is a T-38. The origin of the frame is located at the aircraft center of mass. The x-axis will be through the nose of the aircraft, the y-axis will be in the direction of the right wing, and the z-axis will be pointing mutually orthogonal to those axes through the belly of the aircraft (see Figure 2.2).



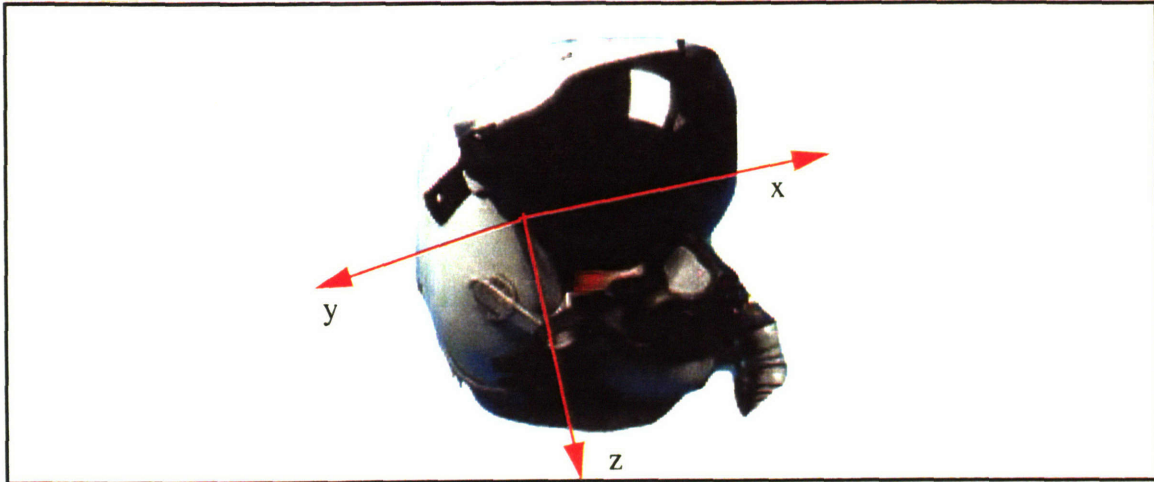
**Figure 2.2** Body Axes Superimposed on image of T-38

This aircraft body axis definition agrees with the derivations of force and moment as found in Stevens and Lewis [20].

### 2.1.3 The Helmet Frame

The helmet frame is defined as a frame similar to the aircraft body frame, with the origin located at the helmet's center of mass. Then, the x-axis is through the

center-line of the face opening, the y-axis is through the right side of the helmet at 90 degrees from the x-axis forming a plane parallel to the line formed by the centers of the eye sockets, and the z-axis is down, mutually orthogonal to the x- and y-axes (see Figure 2.3).

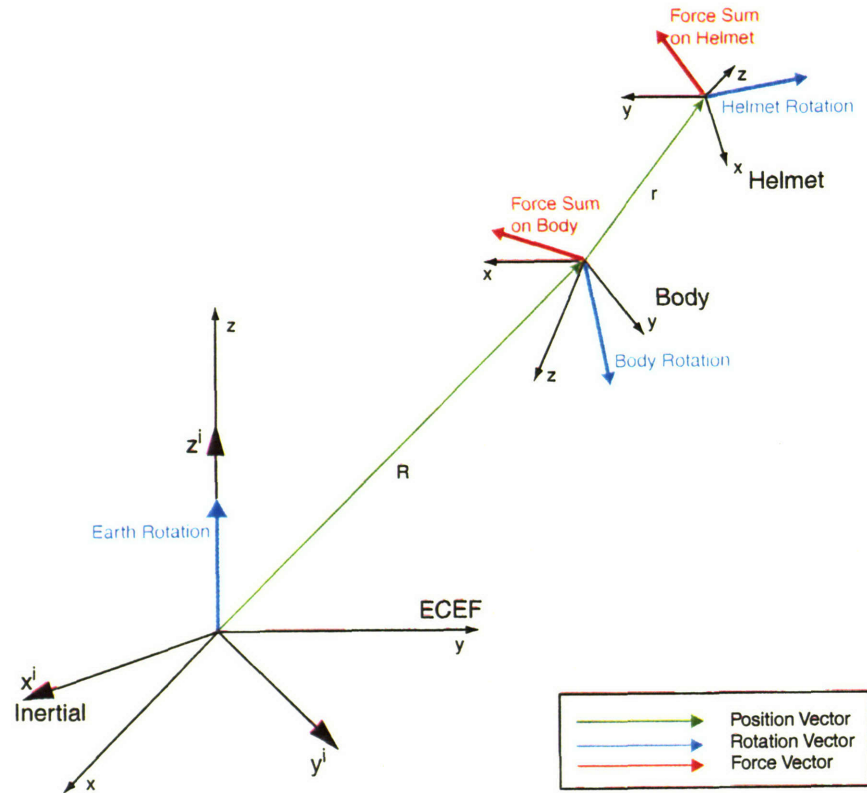


**Figure 2.3** The Helmet Frame superimposed on image of JHMCS

The helmet frame has been defined such that when the pilot is looking along the boresight of the aircraft, the Direction Cosine Matrix (DCM) between the helmet and body frames is the identity matrix.

## 2.2 Relative Dynamics

In general the dynamics of objects within rotating frames is covered quite well in Britting [3], which will be the source for the derivations in the following sections. The geometry discussed in the next two sections is shown in Figure



**Figure 2.4** Illustration of Relative Frame Dynamics and Position Vectors

### 2.2.1 Rotating Frame Dynamics Applied to an Aircraft on the Earth

The three frames to be considered are the inertial frame, the earth frame, and the body frame of the aircraft. The inertial frame is necessary as the frame in which all forces are calculated because the earth is rotating with respect to the inertial frame, causing centripetal, tangential, and Coriolis acceleration components. Therefore, with “i” denoting the inertial frame, “e” denoting the earth frame,  $R$  denoting the position vector,  $C$  denoting a DCM, and  $W$  denoting a skew-symmetric rotation matrix (as derived in Britting [3]) the total acceleration coordinatized in the ECEF frame is:

$$\ddot{\vec{R}}^e = C_i^e \left( \ddot{\vec{R}}^i + 2\Omega_{ie}^i \dot{\vec{R}}^i + (\dot{\Omega}_{ie}^i + (\Omega_{ie}^i)^2) \vec{R}^i \right) \quad (2.1)$$

since the angular acceleration of the earth frame with respect to the inertial frame is zero, we can look only at the rotational components as:

$$\ddot{\vec{R}}^e \Big|_{rot} = C_i^e \left( 2\Omega_{ie}^i \dot{\vec{R}}^i + (\Omega_{ie}^i)^2 \vec{R}^i \right) \quad (2.2)$$

Finally, the position of an aircraft in the earth's atmosphere is normally expressed within the rotating earth frame, not the inertial frame:

$$\ddot{\vec{R}}^e \Big|_{rot} = 2\Omega_{ie}^e \dot{\vec{R}}^e + (\Omega_{ie}^e)^2 \vec{R}^e \quad (2.3)$$

So the Differential Equation (2.3) represents the forces due to the earth's rotation experienced by a time-varying position vector  $\vec{R}$ , expressed in the ECEF frame. An application of this would be a jet at a given position  $\vec{R}$  in the ECEF frame, travelling with a velocity experiencing coriolis and centripetal forces due to the constant rotation of the ECEF frame with respect to inertial space. The full equation for the acceleration on the aircraft in the body frame would be given by:

$$\ddot{\vec{R}}^b = \vec{a}^b = \frac{\vec{F}^b}{m} + C_e^b \left( 2\Omega_{ie}^e \dot{\vec{R}}^e + (\Omega_{ie}^e)^2 \vec{R}^e + \vec{G}^e \right) \quad (2.4)$$

where  $F_b$ ,  $m$ , and  $G_b$  are the Forces applied by the aircraft (i.e. Lift, Drag, Side Force), the mass of the aircraft, and the mass attraction acceleration of the earth, respectively. Note that the non-rotational component in Equation (2.1) is

$$\ddot{\vec{R}}_i = C_b^i \left( \frac{\vec{F}^b}{m} + C_e^b \vec{G}^e \right) \quad (2.5)$$

from Equation (2.4).

## 2.2.2 Rotating Frame Dynamics Applied to Helmet Acceleration

The derivation for the helmet is similar to the derivation of velocity and acceleration in the earth frame for the aircraft body. However, the simplifications made for the aircraft cannot be made, which leaves extra terms in the solution.

The formulation of an equation which represents the acceleration on the head in the head frame, h, which is at a lever arm r from the center of gravity of a freely rotating T-38 aircraft frame (seen in Figure 2.4), b, with respect to the inertial frame, i, is as follows:

$$\ddot{\vec{r}}^h = C_b^h \left( \ddot{\vec{r}}^b + 2\Omega_{ib}^b \dot{\vec{r}}^b + (\dot{\Omega}_{ib}^b + (\Omega_{ib}^b)^2) \vec{r}^b \right) \quad (2.6)$$

In this case, however, “r” represents the position vector between the helmet and the T-38 center of gravity, which is subject to the aircraft’s dynamics. Therefore, the T-38’s total acceleration must be added to the first term on the right hand side, which includes the aircraft aerodynamic, Coriolis, and centripetal forces, as well as the local earth mass attraction gravity acceleration:

$$\begin{aligned} \ddot{\vec{r}}^h = C_b^h \left( \ddot{\vec{r}}^b + \frac{\vec{F}^b}{m} + C_e^b \left( 2\Omega_{ie}^e \dot{\vec{R}}^e + (\Omega_{ie}^e)^2 \vec{R}^e + \vec{G}^e \right) \right) + \\ C_b^h \left( 2\Omega_{ib}^b \dot{\vec{r}}^b + (\dot{\Omega}_{ib}^b + (\Omega_{ib}^b)^2) \vec{r}^b \right) \end{aligned} \quad (2.7)$$



Equation (2.7) represents the accelerations on the helmet, which has dynamics with respect to a vehicle moving over the surface of a rotating earth.

## **2.3 Vehicle Dynamic Model**

The vehicle dynamics used in this thesis will be those for a T-38 jet aircraft, as used in the CSDL T-38 avionics demonstrator simulation. As in all aircraft, the applied forces are Lift, Drag, Thrust, and the three applied moments about the principle body axes are generated by the vehicle aerodynamics, mass distribution, and propulsion system. Since the simulation code which generates the T-38 dynamics is not the primary focus of this thesis, a brief overview will be provided in this section.

### **2.3.1 Aerodynamic Model**

Traditionally, the Aerodynamic model is taken from tables of data collated from wind tunnel during design and aircraft testing. This simulation is no exception, using aerodynamic tables to interpolate the current values for the aerodynamic and moment coefficients for all aerodynamic surfaces, and drag on the entire vehicle over a wide range of state variables (i.e. angle of attack, velocity, angular rates, control surface deflections, etc.). These moment coefficients are then combined with the current aircraft speed relative to the surrounding fluid medium (atmosphere) as well as the local atmospheric temperature and density to create resulting forces and moments. These quantities can then be integrated and manipulated into linear and angular velocities and positions using a predictor/corrector integration scheme as found in Gersh-

enfeld. The flight, atmospheric, and ground models in addition to the integration scheme were all taken directly from the Draper T-38 avionics development demonstrator simulation, and integrated into the head tracker simulation with only minor modifications.

### **Control Surfaces**

The flight model portion of the code consists of a way of reading in commands from the simulated cockpit controls (stick, rudder pedals) generate deflections of the control surfaces (ailerons, horizontal stabilizer, and rudder). This simulation uses a three DoF joystick for elevator, aileron, and rudder command input. These command inputs are fed into control algorithms of the T-38 which output deflections of the control surfaces under the given flight loading conditions. These deflections are then translated into forces and moment components in and about each of the three primary axes of the aircraft via the previously mentioned tables of empirical data.

### **Force Coefficients**

The Coefficient of Lift, Drag, and Side Force are calculated from the interpolated quantities in the following table:

Force Coefficients	Interpolated Linear Coefficients
Lift	Horizontal Tail Moment Arm (speed), Main Wing (speed, AoA), Flaps (AoA), Speed Brakes (speed, AoA), Pitch Rate (speed), Inertial Bending of CL (AoA), Ground Effect (speed, altitude), Rate of Angle of Attack (speed)
Drag	Basic (speed, AoA, Coefficient of Lift), Sideslip (AoA), Flaps (Coefficient of Lift, flap pos'n), Speed Brake (speed, AoA), Wind Milling Engine (speed), Landing Gear (gear pos'n, flap pos'n, gear door pos'n)
Side Force	Sideslip (speed, AoA), Rudder (speed, AoA, sideslip), Aileron (speed), Roll Rate (speed, AoA), Yaw Rate (speed, AoA), Landing Gear (gear pos'n)

**Table 2.1 Interpolated Quantities in Force Coefficient Calculations**

The details of how these quantities are interpolated and from what actual simulation variables are not included in this document. The purpose for including the above information is to provide some idea as to the fidelity of the flight model used as the aircraft model for this simulation.

### **Moment Coefficients**

The Moment Coefficients about the primary axes of the aircraft (defined in Section 2.1.2) are calculated from interpolated values as the force coefficients

in the preceding table. Their individual components are listed (with the quantities of which they are dependent in parentheses) in the following table:

Moment Coefficient	Interpolated Linear Coefficients
Pitch	Basic (based on speed and AoA), Flaps (AoA), Speed Brakes (AoA, Landing Gear), Pitch Rate (based on speed and AoA), Horizontal Tail Deflection (speed), AoA Rate (speed), Landing Gear (Flaps, AoA)
Roll	Sideslip (speed & AoA), Flaps (speed), Ailerons (speed, AoA, & Flaps), Rudder (speed, sideslip, & AoA), Roll Rate (speed & AoA), Yaw Rate (speed & AoA)
Yaw	Sideslip (speed, AoA), Rudder (speed, AoA), Yaw Rate (speed, AoA, Rudder), Roll Rate (speed, AoA), Aileron (speed, AoA)

**Table 2.2 Moment Coefficient Linear Coefficient Components**

Again, the above table is included to give an idea of flight model fidelity only, and the detailed equations are omitted because they are not central to the topic of this thesis.

### 2.3.2 Propulsion Model

The propulsion model deals with how the thrust is generated in the aircraft model. The input is the position of the throttle in the cockpit, and its position relative to the idle and afterburner thresholds. The position of the throttle deter-

mines the fuel rate (which also affects the mass model, as to be discussed in Section 2.3.3), which in turn determines the RPM and thrust. If the afterburners are on, the fuel rate is adjusted accordingly, as well as the thrust. Airspeed and altitude are also accounted for in the thrust equations.

### 2.3.3 Mass Model

The mass model is primarily a static one, based on the geometry and known inertias of the empty (no fuel) T38 aircraft:

Inertial Component	Value
Ixx	11553.6 slug-ft <sup>2</sup>
Iyy	2.833372e+04 slug-ft <sup>2</sup>
Izz	2.926608e+04 slug-ft <sup>2</sup>
Ixz	47.25 slug-ft <sup>2</sup>

**Table 2.3 T-38 Minimum (Empty) Moments of Inertia**

The total weight of the aircraft is determined by the dry weight, 8140.0 Lbs., plus the weight of the fuel in the left and right fuel tanks, 1,1198.98 Lbs. each, for a total weight of  $1.05 \times 10^4$  Lbs. when the simulation starts. This weight decreases as fuel is burned until the tanks are empty, when the T-38 weighs its dry weight. Similarly, the inertias are modified as fuel is burned until they reach the values listed in Table 2.3. This variable inertia is calculated via interpolation tables (as was done in the aerodynamic force and moment calculations) which depend on the fuel quantity in the left and right fuel tanks.

As with the aerodynamic force and moment calculations, the actual equations for calculating the inertias and the weight have been omitted due to their ancillary nature.

### **2.3.4 Equations of Motion**

#### **Aerodynamic Quantities and Earth Velocities**

The computation of velocity into the earth frame, and the resolving of wind into the body frame are critical to computing angle of attack and sideslip, as they are defined by the orientation of the velocity vector in the body frame. This routine calculates the angle of attack (AoA), sideslip, and the rates for both for use in the aerodynamic calculations.

#### **Attitude**

Aircraft attitude is computed by means of the quaternion, which is a 4-element vector representing a single rotation which aligns two different frames by means of describing a unit vector in one frame and an angle about that vector to rotate one of the two frames.

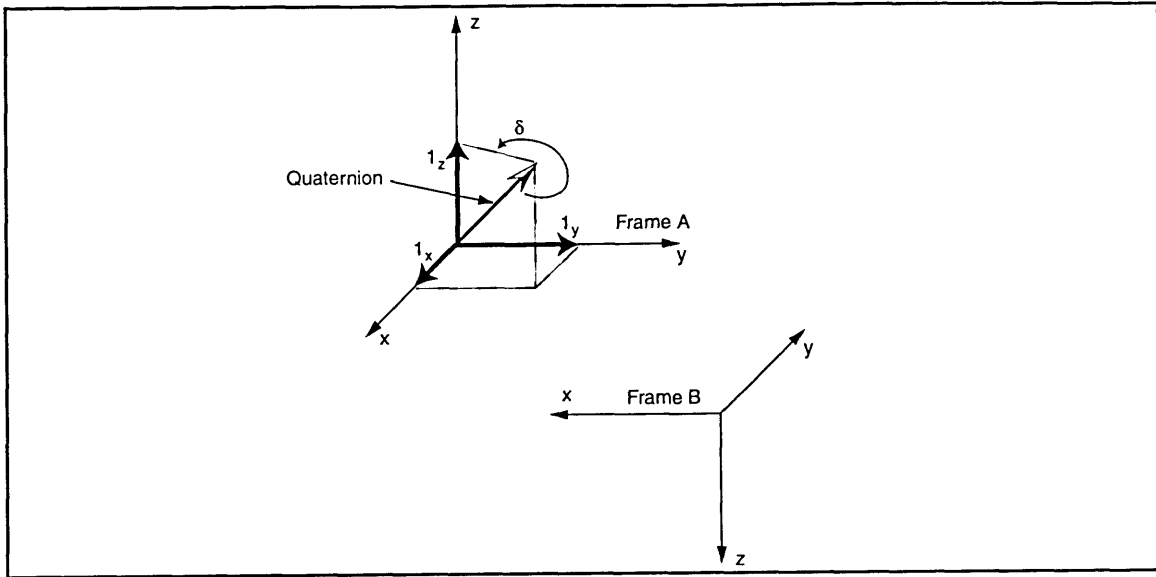


Figure 2.5 Visualization of the Quaternion

$$\vec{q} = \begin{bmatrix} \cos\left(\frac{\delta}{2}\right) \\ \sin\left(\frac{\delta}{2}\right) \begin{bmatrix} 1_x \\ 1_y \\ 1_z \end{bmatrix} \end{bmatrix} \quad (2.8)$$

The differential equations for the quaternion that relates the local level frame to the aircraft body frame are:

$$\dot{\vec{q}}_n^b = \begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} = -\frac{1}{2} \begin{bmatrix} 0 & p & q & r \\ -p & 0 & -r & q \\ -q & r & 0 & -p \\ -r & -q & p & 0 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \equiv -\frac{1}{2} \Omega_q \dot{\vec{q}}_n^b \quad (2.9)$$

where  $p$ ,  $q$ , and  $r$  are the angular rates about the  $x$ ,  $y$ , and  $z$  body axes, respectively.[20] Equation (2.9) is the means by which the quaternion is propagated based on the angular rates integrated from the moments in the body frame. However, the quaternion's magnitude (the norm) must always equal unity. In order to correct for the additive effects of small errors in computation, the norm is computed and used to normalize the quaternion to ensure unit magnitude:

$$\sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} = |\dot{\vec{q}}_n^b| \quad (2.10)$$

$$\frac{\dot{\vec{q}}_n^b}{|\dot{\vec{q}}_n^b|} = \dot{\vec{q}}_n^b|_{norm} \quad (2.11)$$

This normalized quaternion is what is used for the orientation of the aircraft with respect to the local level navigation frame. To use this information to transfer vectors from earth to body, the body-to-earth direction cosine matrix (DCM) is constructed from the body-to-local level quaternion DCM:

$$C_n^b = \begin{bmatrix} \frac{q_0^2 - q_1^2 - q_2^2 + q_3^2}{2} & \frac{2(-q_0q_1 + q_2q_3)}{2} & \frac{2(q_0q_2 + q_1q_3)}{2} \\ \frac{2(q_0q_1 + q_2q_3)}{2} & \frac{q_0^2 - q_1^2 + q_2^2 - q_3^2}{2} & \frac{2(-q_0q_3 + q_1q_2)}{2} \\ \frac{2(-q_0q_2 + q_1q_3)}{2} & \frac{2(q_0q_3 + q_1q_2)}{2} & \frac{q_0^2 + q_1^2 - q_2^2 - q_3^2}{2} \end{bmatrix} \quad (2.12)$$



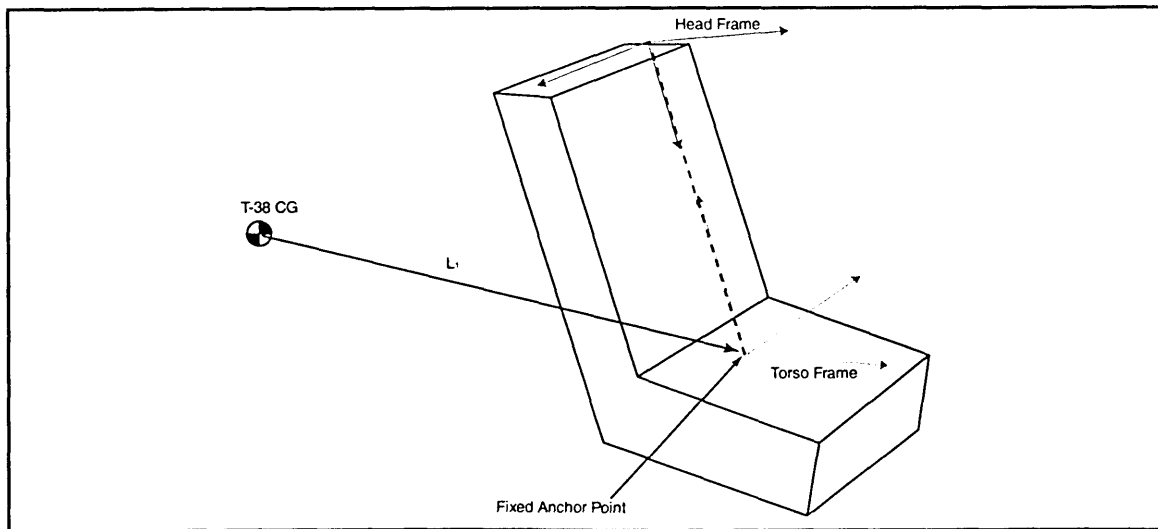
and the earth-to-local level DCM, which is calculated as in Equation (3.6).

## 2.4 Pilot Dynamic Model

The pilot's dynamics are a simplification of reality, however for the purposes of demonstrating the navigation algorithms for the head, they provide the necessary degrees of freedom in the simplest manner for analysis purposes.

### 2.4.1 Stick-Figure Model

The pilot is represented by a rigid torso which is fixed to the ejection seat in the virtual aircraft, and a head which is attached to the top of the torso, with the ability to rotate freely within normal physiological limits. A diagram of the construction can be seen in Figure 2.6. Both frames can rotate about all three axes independently of one another, however if the torso rotates, the head rotates the same amount, along with a translation due to the lever arm (depicted as the dotted line in Figure 2.6).



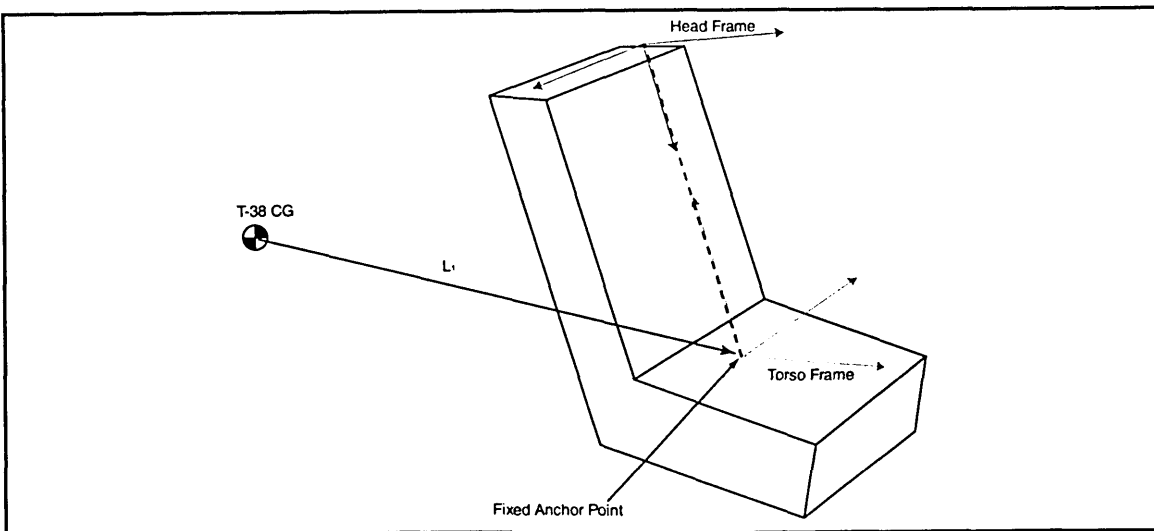
and the earth-to-local level DCM, which is calculated as in Equation (3.6).

## 2.4 Pilot Dynamic Model

The pilot's dynamics are a simplification of reality, however for the purposes of demonstrating the navigation algorithms for the head, they provide the necessary degrees of freedom in the simplest manner for analysis purposes.

### 2.4.1 Stick-Figure Model

The pilot is represented by a rigid torso which is fixed to the ejection seat in the virtual aircraft, and a head which is attached to the top of the torso, with the ability to rotate freely within normal physiological limits. A diagram of the construction can be seen in Figure 2.6. Both frames can rotate about all three axes independently of one another, however if the torso rotates, the head rotates the same amount, along with a translation due to the lever arm (depicted as the dotted line in Figure 2.6).



**Figure 2.6** Head and Torso Frames in Neutral Position Relative to the Ejection Seat

The detailed description of how the pilot dynamics are calculated can be found in Section 3.3.2.

# Chapter 3.0

## Simulation Architecture

### 3.1 Overall Architecture

#### 3.1.1 Concept

The simulation to represent the dynamic environment, instruments, and algorithms of the inertial head tracker is pieced together from existing parts of three separate simulations. The aircraft dynamics models are from the CSDL Simulation Laboratory T-38 avionics simulator. The micromechanical instrument models used to simulate both IMUs are from the CSDL Simulation Laboratory Micro Air Vehicle (MAV) simulation. The navigation code and filter implementation are based on the code used in the CSDL Competent Munitions Advanced Technology Demonstrator (CMATD) simulation and hardware. Finally, the GPS model which simulates the satellite constellation, transmitted signals, and the receiver messages was taken from the CMATD program as well.

The goal of the simulation was to create a dual-IMU system with a single Kalman Filter to bound the drift of the IMUs and estimate the dynamic lever arm between them. This simulation would have the ability to independently modify the relative position and accuracy of the IMUs and their individual sensor components, in addition to providing a flexible means of generating simulated

flight dynamics and pilot motion for performance analysis of the dual-IMU head tracker system.

## 3.2 Structure

As Section 3.1.1 suggests, the simulation has four independent modules which were inherited from previous projects, in addition to three new modules for this particular simulation.

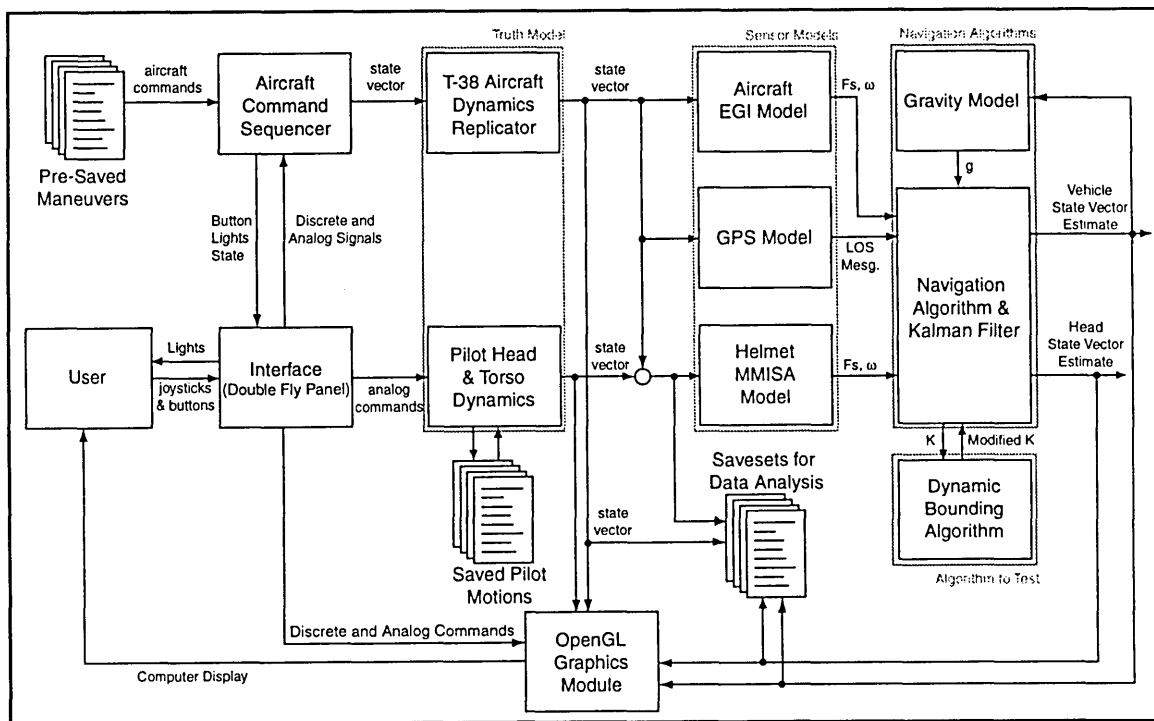
Module Name	New/Inherited	Function
T-38 Dynamics Module	Inherited with minor modifications	Provide Realistic and Flexible Flight Vehicle Dynamics
MMISA Module (x2)	Inherited	Provide Realistic and Flexible Representation of an IMU
GPS Module	Inherited	Simulate the GPS satellite constellation and a receiver unit
Navigation Algorithms	Inherited with modifications	Calculate Position, Velocity and Attitude when combined with an IMU
Kalman Filter Module	New (MATLAB)	Estimate Errors in the IMU instruments and the Navigation solution
Pilot Dynamics Model	New	Provide ample degrees of freedom for the relative lever arm between IMUs
Bounding Algorithm Model	New	Enhance the Kalman Filter and Nav module to ensure feasible head position estimates

**Table 3.1 Simulation Modules**

Module Name	New/Inherited	Function
Hardware and Graphics Interface	New with a few inherited components	Enable simple trajectory generation and control for data analysis and management

**Table 3.1 Simulation Modules**

In general, the way in which these modules interface with one another is shown in Figure 3.1, which represents the overall simulation architecture.



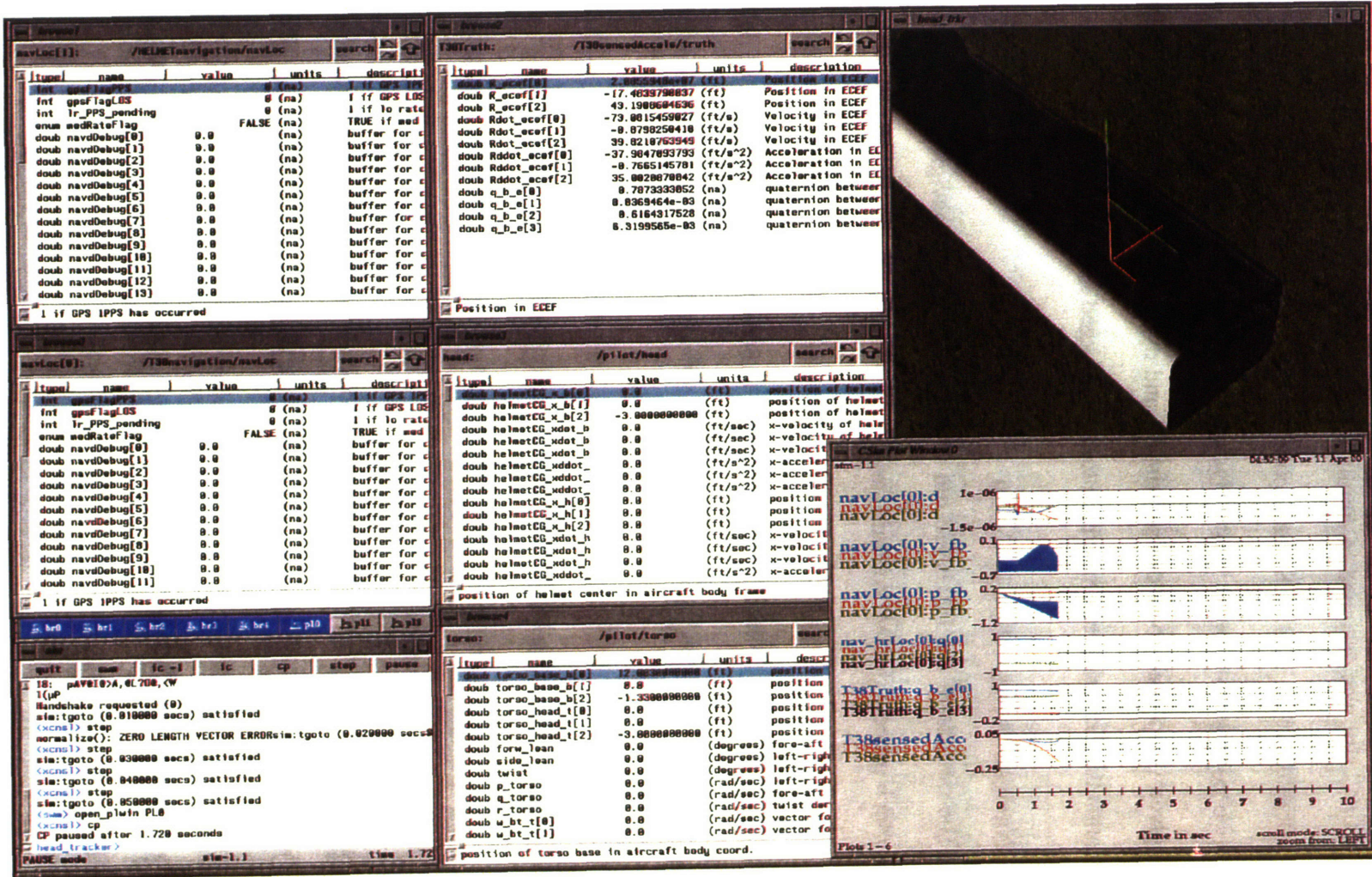
**Figure 3.1 Simulation Module Architecture**

### 3.2.1 Implementation

All of the modules except for the Kalman Filter were coded in the C programming language, and implemented within the CSDL C-Sim Framework. The Framework allows data visualization in multiple numeric and graphical for-

mats as data is processed and generated. A screenshot of the C-sim framework can be seen in Figure 3.2, with the shell, browse windows, OpenGL graphics window, and a plot window all visible. The upper right window contains the graphical representation of an aircraft cockpit, with the pilot torso and head coordinate frames in red and green, respectively. This particular screenshot was taken while debugging the quaternion calculations for the head truth algorithms.

Figure 3.2 The C-sim Framework





## **Trajectory and Hardware Simulation in Draper C-Sim Environment**

The C-sim was designed to simulate a T-38 flying within the atmosphere of a rotating earth. The T-38 has a controllable stick-figure pilot, represented by two coordinate frames. The IMUs are also emulated in the sim as well, and given the proper noise characteristics, and dynamic inputs. The true position and attitude variables are written to a time-tagged save-set file.

## **Navigation Algorithm in C-Sim**

The outputs of the IMUs are run through two parallel navigation algorithms which process the IMU outputs, and calculate attitude (via the third order algorithm developed by McKern in Section 4.2.1, equation (4.3)), velocity and position. These outputs are saved to a file, along with the compensated delta-velocity, delta-angles, the dynamic state/instrument state  $\phi_{12}$  sub-matrices (see Equation (4.23)).

## **Kalman Filter and Data Analysis in MATLAB**

The outputted variables mentioned in the above two sections are read into MATLAB, and parsed into the proper files. The Kalman Filter implementation is discussed in Chapter 4.0. The filter does not have reset capability, but the navigation solution could be corrected in MATLAB using error estimate. Without resets, the estimated errors will grow unbounded with time. However, for the purpose of this analysis, a filter without reset capabilities provides adequate evaluation of the concepts addressed in this study.

Data analysis is performed in the MATLAB environment after collection from the simulation. Details of this analysis are discussed in Chapter 5.0, “Data Analysis” .

### **3.3 Physics and Dynamics Models**

The simulation consists of a dual-set of algorithms, one generating truth, and another generating outputs and interface signals of system hardware. The primary purpose of this study is to generate a consistent dynamic environment which can be used as a benchmark to test the dual-IMU head tracker concept in an aircraft.

#### **3.3.1 T-38 Flight Vehicle Model**

This model, originally written by the CSDL simulation lab, is the dynamics engine for the flight vehicle. Its structure and functionality are overviewed in this section with particular attention to how features relate to testing an IMU head tracker in a jet aircraft environment.

##### **Aircraft**

The T-38 aerodynamics are summarized by a group of aerodynamic tables, relating individual force and moment elements. These elements are then summed up into the total moments and forces on the aircraft. Specifically, these forces and moments are calculated using the stability derivatives interpolated from the tables listed in Section 2.3.1, which are in turn multiplied by their applicable state vector quantities and summed to get the lift, drag, side force, and the three moment coefficients on the T-38. These forces and

moments are integrated into velocities and angular rates in the body frame, and a final integration takes them to position and attitude.

The Head Tracker simulation uses these generated dynamics as a baseline for motion within in the ECEF frame. These dynamics are then fed into the GPS constellation and receiver emulation module, as well as the MEMS instrument models. The MEMS modules, unlike their GPS module counterpart, require additional processing on the inputs to account for the earth's rotation, and angular rate due to motion over the earth's spheroidal surface as well as accelerations felt by lever arms from the sensors to the center of rotation. In particular, the additional angular rates are represented by the following equations:

The vector and corresponding skew-symmetric matrix of the earth's rotation in the ECEF frame with respect to inertial space are:

$$\vec{\omega}_{ie}^e = \begin{bmatrix} 0 \\ 0 \\ \Omega_{earth} \end{bmatrix} \rightarrow \Omega_{ie}^e = \begin{bmatrix} 0 & -\Omega_{earth} & 0 \\ \Omega_{earth} & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (3.1)$$

The angular rotation vector due to motion over the earth's surface in the local level navigation frame (n-frame) where  $\lambda$  denotes longitude and  $\phi$  denotes latitude is:

$$\vec{\omega}_{en}^n = \begin{bmatrix} \dot{\lambda} \cos \phi \\ -\dot{\phi} \\ -\dot{\lambda} \sin \phi \end{bmatrix} \quad (3.2)$$

and the body angular rotation rate vector, whose components are from the T-38 dynamics model as obtained from integrating the angular accelerations calculated from the moments in the body frame, is:

$$\omega_{nb}^b = \begin{bmatrix} p_{T38} \\ q_{T38} \\ r_{T38} \end{bmatrix}^b \quad (3.3)$$

Equations (3.1), (3.2), and (3.3) are combined using the appropriate DCMs in Equation (3.4).

$$\dot{\omega}_{ib}^b = \omega_{nb}^b + C_n^b \omega_{en}^n + C_e^b \omega_{ie}^e \quad (3.4)$$

The DCM between the n-frame and the body frame is calculated from the quaternion via Equation (2.12). The DCM between the ECEF frame and the body frame is obtained from the matrix product:

$$C_e^b = C_n^b C_e^n \quad (3.5)$$

where

$$C_e^n = \begin{bmatrix} -\cos \lambda \sin \phi & -\sin \lambda \sin \phi & \cos \phi \\ -\sin \lambda & \cos \lambda & 0 \\ -\cos \lambda \cos \phi & -\sin \lambda \cos \phi & -\sin \phi \end{bmatrix} \quad (3.6)$$

Equation 3.4 is the angular rate detected by the T-38 IMS gyros. The accelerometers, however, detect the acceleration of the body with respect to inertial space as felt by the IMS, which is at a known lever arm from the CG of the aircraft. That lever arm (l) in body coordinates (b), is denoted by

$$\lambda_{imu}^b = \begin{bmatrix} x_{imu} \\ y_{imu} \\ z_{imu} \end{bmatrix}^b \quad (3.7)$$

And the specific force felt on the IMU due to aircraft dynamics is

$$f_{imu}^b = \ddot{R}^b + 2C_e^b \dot{R}^e + C_e^b (\Omega_{ie}^e)^2 R^e + (\dot{\Omega}_{ib}^b + (\Omega_{ib}^b)^2) l_{imu}^b + G^b \quad (3.8)$$

Where  $G^b$  is the mass attraction acceleration of the Earth at position  $R^e$  in the body frame.

### Environment

The surrounding environment for the vehicle—the atmosphere and ground forces and effects models—came from the T38 sim. For the purposes of this simulation, only the atmospheric model’s density and temperature was used in computing the aerodynamic forces and moments, and no wind or turbulence was introduced into the simulation.

In early stages of development, the ground forces model was found to be faulty in the fact that it could not handle a rotating earth properly while the T-38 was sitting on the ground. Measures were taken to fix this modelling error so that the effects of acceleration and takeoff on the head tracker could be analyzed. Other than providing a means for determining if the T-38 had crashed during a simulation run and for providing a model for ground forces and interactions during takeoff, the ground model was not used extensively, and the original model for the Instrument Landing System (ILS) was disabled.

### 3.3.2 Pilot Model

The pilot model's purpose from its inception was not to be an anatomically accurate dynamic model of the pilot's head as a shock mount, but it was intended as a means to provide the necessary degrees of freedom to test the bounding algorithms used in the dual IMU head tracker.

#### Angular Position of the Pilot Torso and Head

Since the torso and the head are attached via a fixed lever arm between their origins, angles simply add between the two frames, relative to the aircraft body. These Euler angles are used for input purposes only, and are converted to a quaternion, which eliminates the singularity of Euler angles ( $\phi$  denotes roll,  $\theta$  denotes pitch, and  $\psi$  denotes yaw) at their local zenith. To construct the quaternion from Euler angles, both the torso and the head use:

$$q_0 = \pm \left( \cos\left(\frac{\phi}{2}\right) \cos\left(\frac{\theta}{2}\right) \cos\left(\frac{\psi}{2}\right) + \sin\left(\frac{\phi}{2}\right) \sin\left(\frac{\theta}{2}\right) \sin\left(\frac{\psi}{2}\right) \right) \quad (3.9)$$

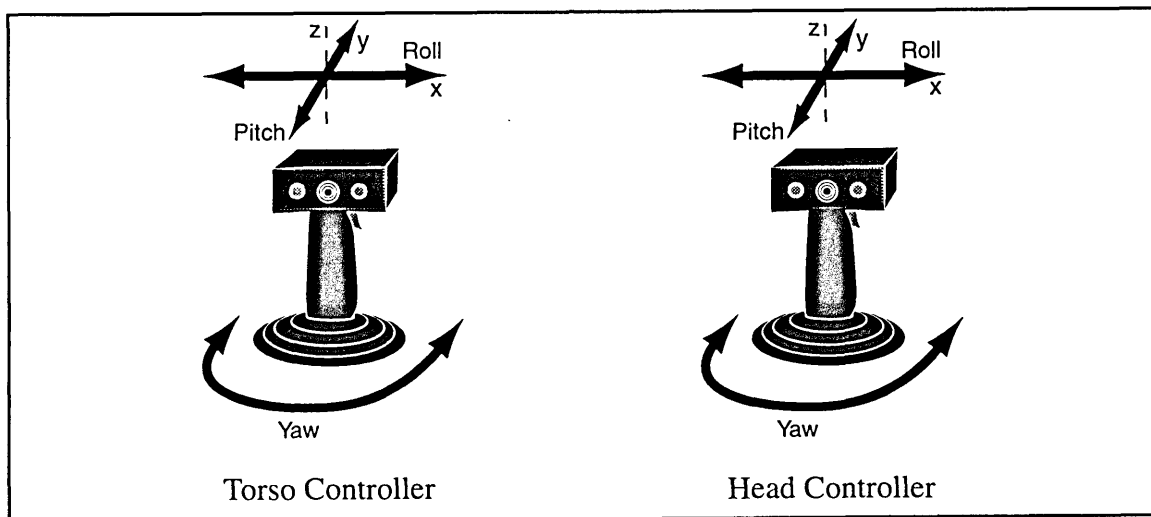
$$q_1 = \pm \left( \sin\left(\frac{\phi}{2}\right) \cos\left(\frac{\theta}{2}\right) \cos\left(\frac{\psi}{2}\right) - \cos\left(\frac{\phi}{2}\right) \sin\left(\frac{\theta}{2}\right) \sin\left(\frac{\psi}{2}\right) \right) \quad (3.10)$$

$$q_2 = \pm \left( \cos\left(\frac{\phi}{2}\right) \sin\left(\frac{\theta}{2}\right) \cos\left(\frac{\psi}{2}\right) + \sin\left(\frac{\phi}{2}\right) \cos\left(\frac{\theta}{2}\right) \sin\left(\frac{\psi}{2}\right) \right) \quad (3.11)$$

$$q_3 = \pm \left( \cos\left(\frac{\phi}{2}\right) \cos\left(\frac{\theta}{2}\right) \sin\left(\frac{\psi}{2}\right) - \sin\left(\frac{\phi}{2}\right) \sin\left(\frac{\theta}{2}\right) \cos\left(\frac{\psi}{2}\right) \right) \quad (3.12)$$

to generate the quaternion from torso to body and head to torso, respectively [20].

The three euler angles which generate each quaternion are controlled independently using two 3-Degree of Freedom joysticks, one for the torso and one for the head. Moving each joystick forward and backward (along the joystick's Y-axis and about it's X-axis) controls the pitch angle, moving it left and right (along the joystick's X-axis and about it's Y-axis) controls the roll angle, and twisting it (about the joystick's Z axis) controls the yaw angle. This geometric relationship of joystick to simulated torso motion is described by Figure 3.3.

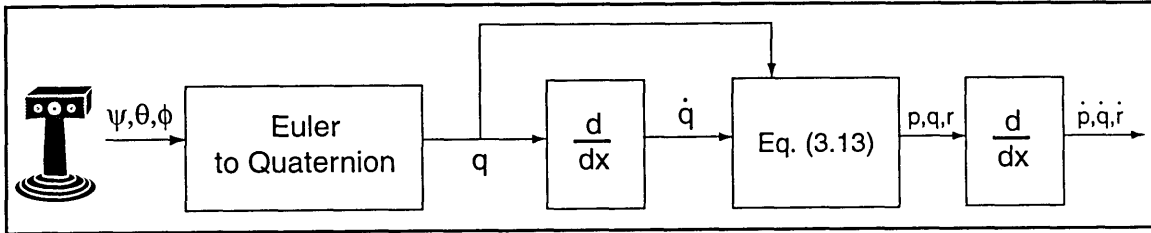


**Figure 3.3** Joystick Geometry for Control of Pilot

### Angular Velocity and Acceleration of the Helmet

In the simulation, position is directly controlled by either the operator or input files. Therefore, to obtain angular velocity the quaternion is differentiated and used in conjunction with the current quaternion to extract p, q, and r relative to the aircraft body using a modified form Equation (2.9), found in Equation

(3.13). The helmet rotation rates are then differentiated to get the angular acceleration. A flow diagram of this process is shown in Figure 3.4.



**Figure 3.4** Pilot Attitude, Angular Velocity and Angular Acceleration Generation

Where Equation (3.13) uses the four-parameter, non-commutative quaternion multiplication operation as defined in McKern [15].

$$\begin{bmatrix} 0 \\ p \\ q \\ r \end{bmatrix} = 2\dot{q}q^* \quad (3.13)$$

where  $q^*$  is the inverse of  $q$ :

$$q^* = \begin{bmatrix} q_0 \\ -q_1 \\ -q_2 \\ -q_3 \end{bmatrix} \quad (3.14)$$

### Specific Force on the Helmet

Since the simulation inputs control head angular position directly and the necessary first and second derivatives are calculated via the quaternion differential equation and a final numerical integration as in Figure 3.4, the helmet accelera-



tion equation (2.7) can now be used since all off the necessary variables have been provided.

## 3.4 Instrument Models

### 3.4.1 GPS Constellation/Receiver Model

To simplify the simulation code, the GPS model was simplified from the LOS measurement model to a simple Position and Velocity measurement, which is the true position and velocity, corrupted by the following delay and white noise parameters:

Corruption Parameter	Value
Delay	1 second (optional)
Position white noise	12 feet
Velocity white noise	0.1 feet per second

**Table 3.2 GPS corruption parameters for simulated measurements**

### 3.4.2 MMISA & EGI Models

The MMISA and EGI models take true angular rates and specific forces at the instrument locations and add noise, bias, and scaling factor errors. The only difference between the EGI and the MMISA is that the MMISA uses parameters as seen in Table 1.1, and the EGI, which uses more accurate instruments than MEMS instruments, has the following numbers:

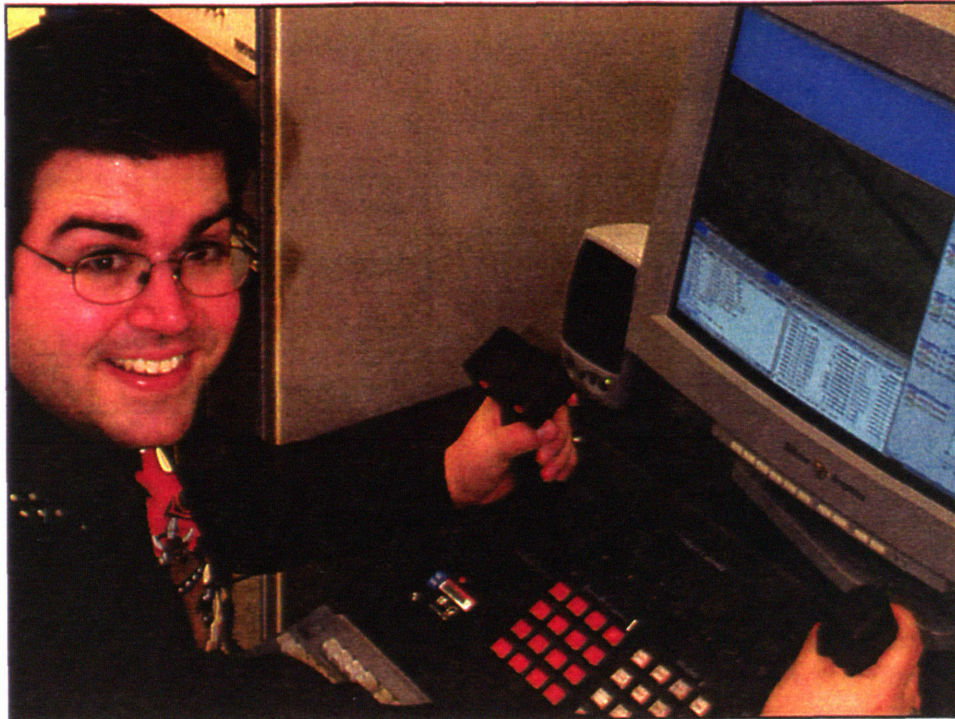
<b>Instrument Error</b>	<b>Units</b>	<b>Value</b>
<b>Gyroscopes</b>		
bias repeatability	degrees/hr	0.0035
bias stability	degrees/hr	0.0006
scale factor	ppm	2
input axis misalignment	arcseconds	4
angle random walk	deg/sqrt(hr)	0.0025
<b>Accelerometers</b>		
bias repeatability	$\mu\text{g}$	60
bias stability	$\mu\text{g}$	20
scale factor repeatability	ppm	100
scale factor stability	ppm	30
nonlinearity	$\mu\text{g}/\text{g}^2$	10
input axis misalignment	arcseconds	8
cross-coupling	$\mu\text{g}/\text{g}^2$	10

**Table 3.3 Generic EGI Instrument Errors (one sigma)**

## **3.5 User Interface**

### **3.5.1 Hardware**

A Silicon Graphics Onyx 2 workstation was used in conjunction with a custom built hardware interface manufactured by BG Systems, consisting of input devices listed in Table 3.4.



**Figure 3.5** Author Running C-sim on an Onyx 2 with the Double Fly Panel

<b>Hardware component</b>	<b>Signals Generated</b>	<b>Use in Sim</b>
Left Joystick	3 Analog stick positions 1 discrete trigger 2 discrete push buttons 1 4-way hat switch	Torso roll, pitch, yaw none none none
Right Joystick	4 Analog stick positions 1 discrete trigger	Roll/pitch/yaw of head or aircraft control inputs none
8 analog levers	8 analog channels	Camera Angle, throttle
32 momentary push buttons	32 discrete signals 32 lights for feedback	Aircraft and Pilot Mode Control Simulation Mode Awareness

**Table 3.4** BG Systems custom hardware panel

Hardware component	Signals Generated	Use in Sim
16 toggle push-button	16 discrete signals 16 lights for feedback	Graphics Components/View Control and Mode Awareness

**Table 3.4 BG Systems custom hardware panel**

The hardware is used primarily for data recording of control inputs to the T-38 or pilot motion, but the current configuration does not allow for simultaneous inputs to both the aircraft and the pilot models. However, the simulation was written with enough flexibility to control operating modes and implement an autopilot to command the aircraft while commanding the pilot motion. The mode controllers are tied to the inputs of the push buttons of the BG systems hardware, and their current state is displayed via lights which are integrated into the buttons on the central button panel.

### 3.5.2 Graphics

The core graphics of the simulation are a cockpit (modelled after a two-seat T-38 cockpit), with two sets of axes for the pilot torso (in red) and head (in green). The terrain is a simple texture map with an overlaid grid, with simple features such as a runway, a couple roads, trees, and mountains. The sky also has a texture map for clouds which are situated at 4000 feet. The camera view angle can be controlled via levers on the BG systems panel, modifying elevation, angle, and zoom with the view centered on the pilot inside the cockpit. Graphics were programmed in the OpenGL graphics language by Silicon

Graphics, and integrated into the Draper C-sim via the DraperGL standard.

The implementation of the OpenGL graphics can be seen in Figure 3.2.

### **3.5.3 Trajectory Generation and Control**

#### **Aircraft Flight Path**

For a trajectory to be completely saved for data analysis, a flight control inputs for the T-38 must be generated and saved as a C-sim “save set”, or a time-tagged data file with variables of interest. These saved files form a library that can be used to test various flight scenarios (bank turns, takeoff, loops, high-g maneuvers, etc.). To save a new set of aircraft control variables done by running the simulation with the aircraft control mode set to manual, so that the BG systems control inputs can be read in, and converted to T-38 stick deflections which the T-38 uses as control input variables to the aircraft simulation. These commands are saved, and then played back later with either manual or saved pilot motion inputs. The inputs recorded for the T-38 flight path are the lateral and longitudinal stick position, rudder pedal deflection, and throttle position.

#### **Pilot Motion**

The pilot motion is saved in a manner similar to the aircraft control data, but an aircraft flight path must be run using the control input autopilot while the pilot motion is being recorded. The two joysticks are used to record torso and head position, and their inputs are saved in a save set data file (just like the aircraft control inputs) to be re-played under various aircraft flight conditions for testing purposes.

### 3.5.4 Data Storage & Analysis

Input data is stored in files to allow the most flexibility in dynamic playback.

The pilot and aircraft control input data files can be combined in any permutation for generating truth and simulated sensor outputs for application of the navigation filter and bounding algorithm in MATLAB. The data which is passed to MATLAB is a binary “mat” file, consisting of a large matrix of time-tagged data, corresponding to truth, filter propagation matrix sub-sections, and navigation solutions.



# Chapter 4.0

## Algorithms

### 4.1 Navigation System Overview

An inertial navigation system uses outputs from gyroscopes and accelerometers to determine attitude, velocity, and position with respect to a navigation coordinate frame (i.e. ECEF, Local Level, or Inertial). Often, an inertial navigator also employs a Kalman Filter to estimate both the instrument and navigation errors of the system, so that the navigation errors can be reduced in magnitude. These errors are usually estimated via an outside position and/or velocity measurement source, such as a radio navigation unit (e.g. GPS, TACAN, VOR, DME, etc.), altimeter (e.g. radio, doppler, or barometric), air data system, sonar, or another non-inertial source. When another inertial unit is employed, it is usually for a transfer alignment, such as those used in submarine ICBMs, or inertially guided munitions deployed from aircraft (i.e. cruise missiles). The block diagram of a typical navigation system can be seen in Figure 4.1, and the symbol definitions can be found in Table 4.1.

#### 4.1.1 Essence of the Navigation and Coupling Algorithms

The essence of this thesis is to develop a filter which estimates the attitude of the pilot's helmet. This is modeled via a stochastically-coupled dual IMU system, where the movement of the helmet relative to the T-38 EGI is modeled by three first order Markov processes in position. The EGI develops accurate posi-



tion due to GPS updates, and accurate EGI attitude is achieved via changes in the direction of the specific force (with accurate position measurements). Transferring the EGI attitude to the helmet is achieved via a relative position update between the EGI and the helmet that includes a first order Markov process model. The Markov process model allows the head to move within a manifold as a time correlated process.

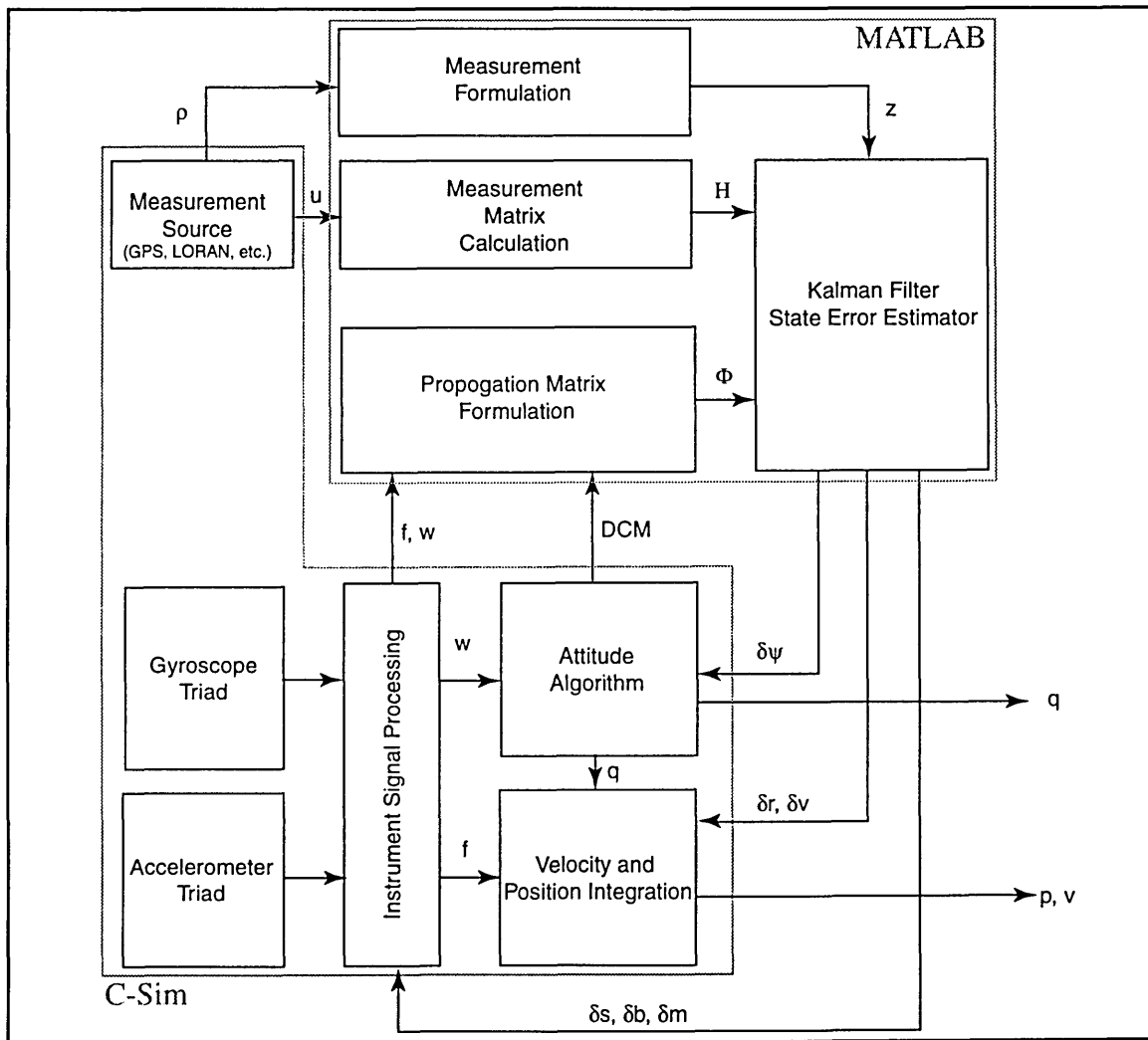


Figure 4.1 Navigation System Architecture

Symbol	Definition	Symbol	Definition
$\rho$	GPS LOS messages	$u$	GPS unit vector to satellite
$H$	Measurement Matrix	$z$	measurement
$\Phi$	Propagation Matrix	$p,v$	position and velocity
$q$	quaternion (attitude)	$w$	angular rate
$f$	specific force	$\delta\psi$	attitude error
$\delta r$	position error	$\delta v$	velocity error
$\delta m$	instrument misalign error	$\delta b$	instrument bias error
$\delta s$	instrument scale factor error		

**Table 4.1 Symbol Definitions for Figure 4.1**

#### 4.1.2 Effect of the Gyros and the Attitude Algorithm

The navigation equations are the “Attitude Algorithm” and “Velocity and Position Integration” blocks in Figure 4.1. An important block in the diagram in the figure is the “Attitude Algorithm” block. If the attitude is calculated incorrectly, then the accelerations will be resolved in erroneous directions resulting in both the incorrect summation of velocity and incorrect integration of position. Furthermore, the Kalman Filter error state propagation matrix will be formulated using the incorrect DCM to transform the instrument error states from the platform frame to the navigation frame. Therefore, it is important that the attitude (whether it is represented by either a DCM or a quaternion) be accurately determined.

## 4.2 Navigation Equations

This section describes the duties of the blocks labeled “Instrument Signal Processing,” “Attitude Algorithm,” and “Velocity and Position Integration” in Figure 4.1. The signal processing block in the figure scales and shifts the incoming gyro and accelerometer signals to both the proper units and values for computation.

### 4.2.1 Attitude Algorithm

The following attitude algorithm is a third order algorithm developed by McKern at the MIT Instrumentation Lab [15] which is typically run at 100 Hz or greater. A fourth order algorithm contains many more terms, and is not practical for embedded systems. Typically, if more accuracy is needed, a partitioned attitude algorithm is used, which runs a portion (the cross-product calculation) at a higher rate than the rate used for transformation of the delta velocity vector to the navigation frame.

A numerical error involving the transformation of delta velocity from the platform frame to navigation frame is called sculling. First-order sculling correction is achieved by averaging the old delta theta with the new delta theta before calculating the delta quaternion as in Equation (4.1) where  $n$  denotes the desired axis of rotation,  $t$  denotes the time of calculation, and  $\delta t$  denotes the time step.

$$\Delta\bar{\theta}_n|_t = \Delta\theta_n|_{t-\frac{\delta t}{2}} = \frac{\Delta\theta_n|_t + \Delta\theta_n|_{t-\delta t}}{2} \quad (4.1)$$

The equations for the quaternion update which utilize the above sculling correction are given in Equations (4.2) and (4.3):

$$\delta = \frac{(\Delta\bar{\theta}_x^2 + \Delta\bar{\theta}_y^2 + \Delta\bar{\theta}_z^2)}{4} \quad (4.2)$$

$$\Delta q|_t = \begin{bmatrix} 1 - \frac{\delta|_t}{2} \\ \frac{(\Delta\bar{\theta}_z|_t \cdot \Delta\bar{\theta}_y|_{t-\delta t}) + (\Delta\bar{\theta}_y|_t \cdot \Delta\bar{\theta}_z|_{t-\delta t}) - 2(\Delta\bar{\theta}_x|_t \cdot \delta|_t)}{24} + \frac{\Delta\bar{\theta}_x|_t}{2} \\ \frac{(\Delta\bar{\theta}_x|_t \cdot \Delta\bar{\theta}_z|_{t-\delta t}) + (\Delta\bar{\theta}_z|_t \cdot \Delta\bar{\theta}_x|_{t-\delta t}) - 2(\Delta\bar{\theta}_y|_t \cdot \delta|_t)}{24} + \frac{\Delta\bar{\theta}_y|_t}{2} \\ \frac{(\Delta\bar{\theta}_y|_t \cdot \Delta\bar{\theta}_x|_{t-\delta t}) + (\Delta\bar{\theta}_x|_t \cdot \Delta\bar{\theta}_y|_{t-\delta t}) - 2(\Delta\bar{\theta}_z|_t \cdot \delta|_t)}{24} + \frac{\Delta\bar{\theta}_z|_t}{2} \end{bmatrix} \quad (4.3)$$

Equation (4.3) is then multiplied by the previous quaternion [15] to produce the quaternion valid at the present time interval:

$$q|_t = q^e \cdot q|_{t-\delta t} \cdot \Delta q|_t \quad (4.4)$$

where  $q^e$  is the quaternion representing the rotation of the earth over one time interval.

Given gyro outputs from three axes, and an initial quaternion, Equations (4.1) through (4.4) represents a way to compute the present attitude. The quaternion

calculated represents the transformation from the platform frame to the ECEF (navigation) frame.

## 4.2.2 Velocity and Position Determination

### Additional Accelerometer Compensation

Further delta-velocity compensation is required (in addition to the instrument error corrections) which carefully accounts for the lever arms from the platform frame center of rotation to each instrument. Also, as part of the navigation solution, the specific force must have Earth mass attraction and the Earth rotational dynamics subtracted from it.

### Accelerometer Lever Arm Compensation

Once the delta-velocities are coordinatized in the platform frame, they must be compensated for tangential and centripetal forces caused by the lever arm for each instrument from the center of platform rotation (the platform coordinate frame's origin). If the lever arms of the three accelerometers from the center of rotation coordinatized in the platform frame are  $\dot{\rho}_1^P$ ,  $\dot{\rho}_2^P$ , and  $\dot{\rho}_3^P$ , then the additional sensed delta velocity components due to any rotation (expressed below as the average delta-theta (Equation (4.1)) about the subscripted axis of rotation) will be the following inner products:

$$\Delta v_1^P \Big|_{l.a.} = \begin{bmatrix} -(\overline{\Delta\theta}_2^2 + \overline{\Delta\theta}_3^2) \\ \overline{\Delta\theta}_1 \overline{\Delta\theta}_2 \\ \overline{\Delta\theta}_1 \overline{\Delta\theta}_3 \end{bmatrix}^T \dot{\rho}_1^P \quad (4.5)$$

$$\Delta v_2^p \Big|_{\text{l.a.}} = \begin{bmatrix} \overline{\Delta\theta}_1 \overline{\Delta\theta}_2 \\ -(\overline{\Delta\theta}_1^2 + \overline{\Delta\theta}_3^2) \\ \overline{\Delta\theta}_2 \overline{\Delta\theta}_3 \end{bmatrix}^T \dot{\rho}_2^p \quad (4.6)$$

$$\Delta v_3^p \Big|_{\text{l.a.}} = \begin{bmatrix} \overline{\Delta\theta}_1 \overline{\Delta\theta}_3 \\ \overline{\Delta\theta}_2 \overline{\Delta\theta}_3 \\ -(\overline{\Delta\theta}_1^2 + \overline{\Delta\theta}_2^2) \end{bmatrix}^T \dot{\rho}_3^p \quad (4.7)$$

which are all combined into the following vector:

$$\overrightarrow{\Delta v^p} \Big|_{\text{l.a.}} = \begin{bmatrix} \Delta v_1^p \Big|_{\text{l.a.}} \\ \Delta v_2^p \Big|_{\text{l.a.}} \\ \Delta v_3^p \Big|_{\text{l.a.}} \end{bmatrix} \quad (4.8)$$

Equation (4.8) represents the delta velocity correction to the accelerometer outputs due to the lever arm from the instrument platform's center of rotation to each accelerometer.

### **Delta-Velocity and Attitude Accumulation (Rate Compensation)**

The compensated delta-velocity vector is then provided to the velocity accumulator. Depending on whether or not the velocity and position algorithms are being calculated at the same rate as the attitude algorithm (in some embedded applications they run at a slower rate for computational reasons) the quaternion will be repeatedly calculated and the delta-velocities will be summed over the number of cycles until they are needed by the velocity and position algorithms.

For the IHMCS, the attitude algorithm is calculated at 100 Hz, while the velocity and position algorithm is calculated at 50 Hz. This means that for every velocity and position calculation, there are two instrument error and lever arm compensated delta-velocity and quaternion calculations. The attitude elements of the navigation solution are the result of multiple quaternion multiplications (one for each time step between navigation solution outputs) using the new delta-quaternions as calculated in Equation (4.3) on page 83. It is the resulting sum of the new compensated delta-velocities which are corrected for Earth Mass Attraction, Earth Rotation effects. These two final corrections are detailed in the following sections.

### Earth Mass Attraction and Earth Rotation Navigation Corrections

The current quaternion relating the platform and ECEF is used to transform the delta-velocity vector from the output of the accelerometers into the ECEF frame. This is done via:

$$\overrightarrow{\Delta v}^e = q_p^e \overrightarrow{\Delta v}^p (q_p^e)^* \quad (4.9)$$

Equation (4.9) must be corrected for Earth mass attraction and rotation. The mass attraction portion of the gravity correction from Britting [3] in geocentric coordinates, expanded to the  $J_4$  term is shown in Equation (4.10).

$$\overrightarrow{G}^c = \begin{bmatrix} -3\frac{\mu}{r^2}\left(\frac{r_e}{r}\right)^2 \sin\phi \cos\phi \left[ J_2 + \frac{1}{2}J_3\left(\frac{r_e}{r}\right) \sec\phi(5\cos^2\phi - 1) + \frac{5}{6}J_4\left(\frac{r_e}{r}\right)^2 (7\cos^2\phi - 3) \right] \\ 0 \\ -\frac{\mu}{r^2} \left[ 1 - \frac{3}{2}J_2\left(\frac{r_e}{r}\right)^2 (3\cos^2\phi - 1) - 2J_3\left(\frac{r_e}{r}\right)^3 \cos\phi(5\cos^2\phi - 3) - \frac{5}{8}J_4\left(\frac{r_e}{r}\right)^4 (35\cos^4\phi - 30\cos^2\phi + 3) \right] \end{bmatrix} \quad (4.10)$$

Note that in Equation (4.10)  $\phi$  represents co-latitude (which is equal to zero at north pole, 90 degrees at the equator, and 180 degrees at south pole), the terms  $J_n$  represent the modal correction terms for gravity deviation from geocentric normal,  $r$  represents the present position vector magnitude, and  $r_e$  is the radius of the earth at the equator (the semi-major axis of the WGS-84 ellipsoid).

The centripetal and Coriolis accelerations due to earth rotation are expressed in Equation (2.3) on page 47. The lever arm correction in Equation (4.8) on page 85 and the Earth rotational correction terms in Equation (2.3) are both used to correct the delta-velocity from Equation (4.9) as prescribed by Equation (4.11):

$$\overrightarrow{\Delta v^e} \Big|_{\text{total}} = \overrightarrow{\Delta v^e} - \Delta t \left( C_c^e \overrightarrow{G^c} + \ddot{R}^e \Big|_{\text{rot}} \right) - C_p^e \overrightarrow{\Delta v^p} \Big|_{\text{l.a.}} \quad (4.11)$$

where  $C_p^e$  and  $C_c^e$  represent the DCMs between platform and earth, and geocentric and earth coordinates, respectively. The result of Equation (4.11) is the final, compensated delta velocity in the ECEF navigation frame which is used to calculate the current velocity and position components of the navigation solution.

### **Velocity Summation and Position Integration**

The fully compensated delta-velocity in Equation (4.11) is then summed with the previous platform velocity solution to obtain the current platform velocity. The current platform velocity is then numerically integrated into the current



position, thus completing the full navigation solution consisting of attitude, velocity and position in the navigation frame.

## 4.3 Error Estimation Kalman Filter

To prevent the navigation solution from accumulating errors, both the navigation solution errors and the instrument errors (i.e. bias and scale factor) are estimated by a Kalman Filter. This section describes the Kalman Filter implemented for the IHMCS.

### 4.3.1 The Discrete Kalman Filter Equations

The Kalman Filter equations as developed in Gelb [8] and applied to the IHMCS are outlined in the following paragraphs.

#### Propagation and Measurement Equations

The propagation matrix formulation of the state equation is:

$$\dot{\hat{x}}_k = \Phi_{k-1} \dot{\hat{x}}_{k-1} + \dot{w}_{k-1} \quad (4.12)$$

and the measurement equation is:

$$z_k = H_k x_k + v_k \quad (4.13)$$

Equation (4.12) is the linear dynamic model for the state vector,  $x$ , formulated using a propagation matrix  $\Phi$ . The vector  $w$  contains the independent white process noise inherent to each state ( $w$  contains the same number of elements as  $x$ ). Equation (4.13) is the measurement equation, which relates the measurement vector,  $z$ , to the state vector via the measurement matrix  $H$ . The vector  $v$

is the white noise inherent to each element of the measurement vector ( $v$  contains the same number of elements as  $z$ ). For every different measurement vector formulation,  $H$  and  $v$  in Equation (4.13) should be re-evaluated to ensure that the measurement vector has both proper relation to the state matrix and proper noise content.

The Kalman Filter uses the Equation (4.14) to propagate the state estimate:

$$\hat{x}_k(-) = \Phi_{k-1} \hat{x}_{k-1}(+) \quad (4.14)$$

and Equation (4.15) to propagate the covariance matrix,  $P$ :

$$P_k(-) = \Phi_{k-1} P_{k-1}(+) \Phi_{k-1}^T + Q_{k-1} \quad (4.15)$$

utilizing the state transition matrix from Equation (4.12) and the process noise matrix,  $Q$  which is a diagonal matrix containing the variance of the white noise processes of  $w$ .

When measurement information is available, Equations (4.14) and (4.15) are used to propagate the state estimate and the covariance matrix, making them valid at the time of the measurement. The measurement matrix that is valid for the current measurement and the covariance matrix are both used to calculate a gain matrix via Equation (4.16), in conjunction with the measurement noise matrix,  $R$ , which is another diagonal matrix containing the variance of the white noise processes of  $v$  from Equation (4.13).

$$K_k = P_k(-) H_k^T [H_k P_k(-) H_k^T + R_k]^{-1} \quad (4.16)$$

This gain matrix is then used to update the state estimate, by multiplying  $K$  times the innovation. The innovation is the difference between the current measurement,  $z_k$ , and the predicted measurement,  $H_k \hat{x}_k(-)$ , as shown in Equation (4.17):

$$\hat{x}_k(+) = \hat{x}_k(-) + K_k [z_k - H_k \hat{x}_k(-)] \quad (4.17)$$

The gain matrix, measurement matrix, and measurement noise matrix are all used to update  $P$  as shown in Equation (4.18):

$$P_k(+) = [I - K_k H_k] P_k(-) [I - K_k H_k]^T + K_k R_k K_k^T \quad (4.18)$$

The associated timing diagram for use of Equations (4.14) through (4.18), which graphically explains the + and - notation, can be found in Figure 4.2. Equations (4.14) and (4.15) are used during the “propagation interval” to advance both the state and the covariance matrix to the pre-measurement time, as denoted by the horizontal red arrows in Figure 4.2. Equations (4.16), (4.17), and (4.18) are used during the “measurement and update” interval to update the state and the covariance matrix based on the new measurement data, represented by the vertical spike in Figure 4.2.



	Quantity	States
$x_{nav}$	Position Errors	$\delta x, \delta y, \delta z$
	Velocity Errors	$\delta V_x, \delta V_y, \delta V_z$
	Attitude Errors	$\psi_1, \psi_2, \psi_3$
$x_{inst}$	Gyro Bias Errors	$\delta b_{g1}, \delta b_{g2}, \delta b_{g3}$
	Gyro Scale Factor Errors	$\delta s_{g1}, \delta s_{g2}, \delta s_{g3}$
	Accelerometer Bias Errors	$\delta b_{a1}, \delta b_{a2}, \delta b_{a3}$
	Accelerometer Scale Factor Errors	$\delta s_{a1}, \delta s_{a2}, \delta s_{a3}$

**Table 4.2 Generic Navigation Error States**

Each of these two 21-element state vectors (one for each IMU) are split into  $x_{nav}$  and  $x_{inst}$  and arranged with an additional three-element relative position state vector,  $x_{rel}$ , into the state vector used by the Kalman filter:

$$\hat{x} = \begin{bmatrix} x_{nav_{T38}} \\ x_{nav_{helmet}} \\ x_{rel} \\ x_{inst_{T38}} \\ x_{inst_{helmet}} \end{bmatrix} \quad (4.19)$$

### Propagation Matrix Formulation

The state propagation matrix corresponding with the above state vector is:

$$\Phi = \begin{bmatrix} \phi_{11_{T38}} & 0 & 0 & \phi_{12_{T38}} & 0 \\ 0 & \phi_{11_{helmet}} & 0 & 0 & \phi_{12_{helmet}} \\ 0 & 0 & \phi_{rel} & 0 & 0 \\ 0 & 0 & 0 & I & 0 \\ 0 & 0 & 0 & 0 & I \end{bmatrix} \quad (4.20)$$

where for both the T-38 and the helmet IMUs the propagation matrices which relate the navigation error states within one IMU to one another is:

$$\phi_{11} = I + \begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}^* & \begin{bmatrix} 0 & 2\omega_e & 0 \\ -2\omega_e & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & -f_3 & f_2 \\ f_3 & 0 & -f_1 \\ -f_2 & f_1 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & \omega_e & 0 \\ -\omega_e & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{bmatrix} \Delta t \quad (4.21)$$

The  $f_j$  components of  $\phi_{11}$  are the elements of the instrument, misalignment and lever-arm-compensated specific force vector of the corresponding IMU for the given  $\phi_{11}$  matrix (the result of Equations (4.2) through (4.9), divided by the time stamp  $\Delta t$ ). The remaining constant,  $\omega_e$ , is the earth rotation rate. Note that the asterisk on the zero matrix in the second row, first column of Equation (4.21) represents that the 3 x 3 zero matrix is a replacement for the following rotation vector in skew symmetric matrix form:

$$\dot{\omega}_{ie}^e \times \omega_{ie}^e + \dot{\omega}_{ie}^e \quad (4.22)$$

which is on the order of  $2 \times 10^{-3}$  degrees per second per second, or  $6 \times 10^{-7}$  radians per second per second. For inertial systems with instruments of higher accuracy than those in the MMISA, this matrix should not be neglected, however due to the current level of MEMS sensor accuracy, it has been neglected for this system.

Also for both IMUs, the propagation matrix segments which relate the instrument errors to their corresponding navigation errors are both of the form:

$$\phi_{12} = \begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & C_a^e & C_a^e \cdot \begin{bmatrix} f_1 & 0 & 0 \\ 0 & f_2 & 0 \\ 0 & 0 & f_3 \end{bmatrix} \\ C_g^e & C_g^e \cdot \begin{bmatrix} \omega_1 & 0 & 0 \\ 0 & \omega_2 & 0 \\ 0 & 0 & \omega_3 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{bmatrix} \Delta t \quad (4.23)$$

where  $f_j$  represents the elements of the instrument specific force vector of the corresponding IMU for the given  $\phi_{12}$  matrix.  $\omega_j$  represents the instrument error compensated angular rate outputs of the corresponding IMU for the given  $\phi_{12}$  matrix.  $C_a^e$  and  $C_g^e$  represent the DCMs from the accelerometer and the gyro frames to the ECEF frame, respectively.

Finally, the relative propagation matrix represents a Markov process inside a three dimensional manifold, within which the pilot's head must remain due to physiological constraints. Equation (4.24) is the relative phi matrix:

$$\phi_{rel} = \begin{bmatrix} 1 - \frac{\Delta t}{\tau_1} & 0 & 0 \\ 0 & 1 - \frac{\Delta t}{\tau_2} & 0 \\ 0 & 0 & 1 - \frac{\Delta t}{\tau_3} \end{bmatrix} \quad (4.24)$$

The  $\tau_j$  variables represent the Markov process time constants for all three body axes. The Markov process is discussed in more detail in Section 4.4.

### Measurements

The GPS position and velocity updates are contained in the z vector:

$$\vec{z}_{gps} = \begin{bmatrix} x \\ y \\ z \\ V_x \\ V_y \\ V_z \end{bmatrix}_{nav} - \begin{bmatrix} x \\ y \\ z \\ V_x \\ V_y \\ V_z \end{bmatrix}_{gps} \quad (4.25)$$

which has the corresponding H-matrix in Appendix A:

The three element measurement vector used to couple the two navigators is calculated via:



$$\vec{z}_{rel} = \hat{x}_h - \hat{x}_b - \vec{r}_{rel} \quad (4.26)$$

where  $x_h$  denotes the estimated helmet position,  $x_b$  denotes the estimated aircraft body position, and  $r_{rel}$  denotes the nominal probabilistic lever arm which is the three-element expected value of the relative state Markov process vector in ECEF coordinates. The corresponding measurement matrix can be found in Appendix A.

If both a relative and a GPS measurement is conducted, the two measurement vectors are appended to one another:

$$\vec{z}_{combo} = \begin{bmatrix} \vec{z}_{gps} \\ \vec{z}_{rel} \end{bmatrix} \quad (4.27)$$

with the H matrix being appended similarly:

$$H_{combo} = \begin{bmatrix} H_{gps} \\ H_{rel} \end{bmatrix} \quad (4.28)$$

Equation (4.28) can be explicitly seen in Appendix A.

As can be seen by Equation (4.28), the relative measurements couple in the effects of the GPS updates into the helmet navigator, and add in the relative lever arm error (which is the relative state Markov process vector and part of the state vector).

### 4.3.3 Generalization to N Navigators With One Master Kalman Filter

One can begin to see a generalization in Equation (4.20), when multiple navigators are placed into a single Kalman Filter. The sub-matrices for relating the dynamic states to each other are on the diagonal of the upper left of the  $\Phi$  matrix, while the corresponding inertial instrument error states are directly to the right, in the upper right of the  $\Phi$  matrix, along the local diagonal. Relative states are beneath the dynamic states, and are along the main diagonal as Markov processes, and the lower right is an identity matrix. Figure 4.3 shows the partitioning for N-navigators:

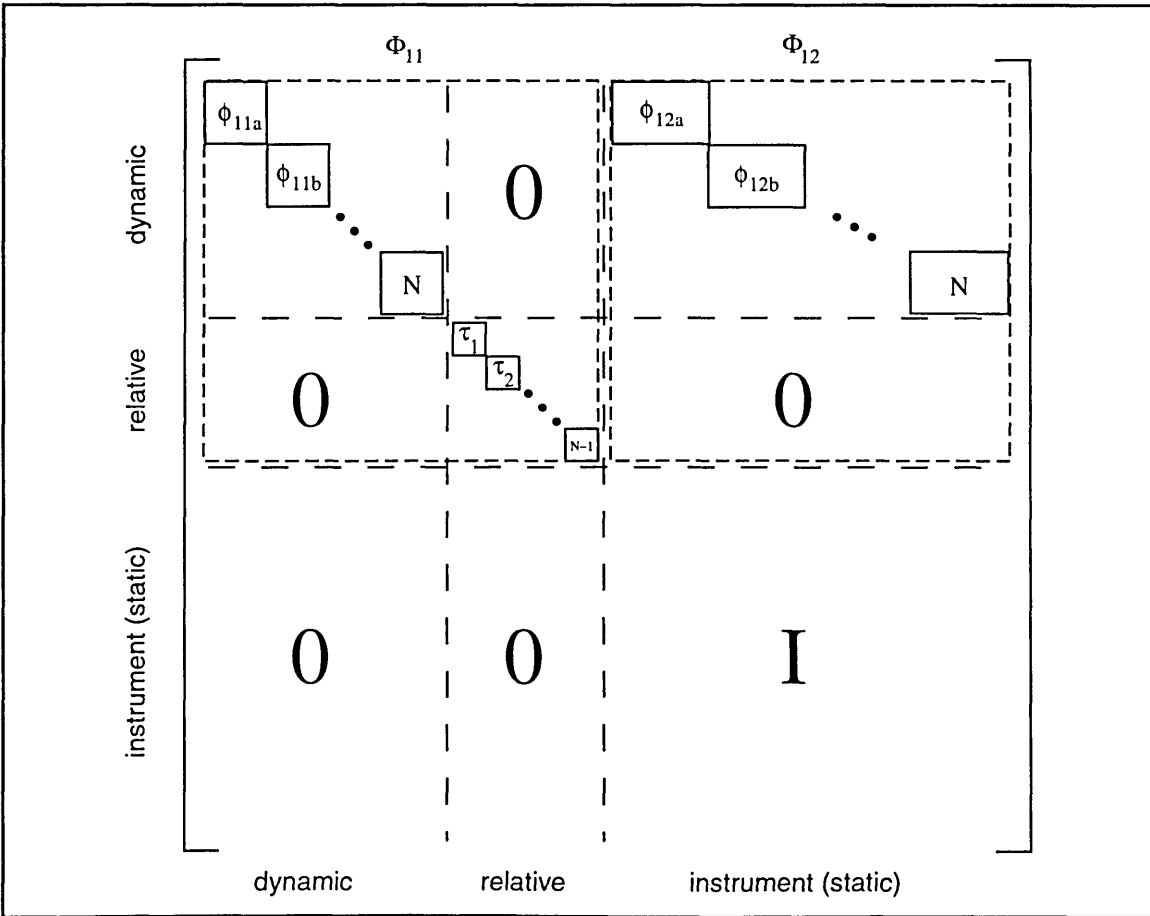


Figure 4.3 Generalized N-Navigator PHI Matrix

As can be seen by the above construction, the primary nature of the dynamic and static portions of the F matrix are conserved in the partitions marked  $\Phi_{11}$  and  $\Phi_{12}$ . This allows for sparse matrix calculations to be used when implementing the Kalman Filter in a program which ignore the zero sections of the matrix, as was implemented in the embedded filter for CMATD.

### **Parent Navigator vs. Internally Referenced (Markov) States**

Also note that the N-navigator formulation must have at least one navigator which is updated by an outside measurement (such as GPS or another radio navigation system—this will be known as the “parent navigator”), while the rest of the navigators can be dependent upon the relative Markov states for coupling to the parent navigator. This means that there will be a  $z_{\text{gps}}$  and  $H_{\text{gps}}$  for the parent navigator, and a number of  $z_{\text{rel}}$  and  $H_{\text{rel}}$  matrices for various permutations of relative measurements. The other internally bounded (via the *a priori* relative position knowledge and Markov) states can be referenced to one another, as well as to the parent navigator. For example, this would be implemented if the relative knowledge to another internally bounded state is known better than the same reference to the parent navigator, and that other internally bounded state has an equal-or-better accuracy and Markov dynamic with respect to the parent navigator. For certain applications, the relative state blocks (in the central region of the  $\Phi$  matrix) may outnumber the dynamic blocks (in the upper left region of the  $\Phi$  matrix).

## 4.4 Bounding Algorithm

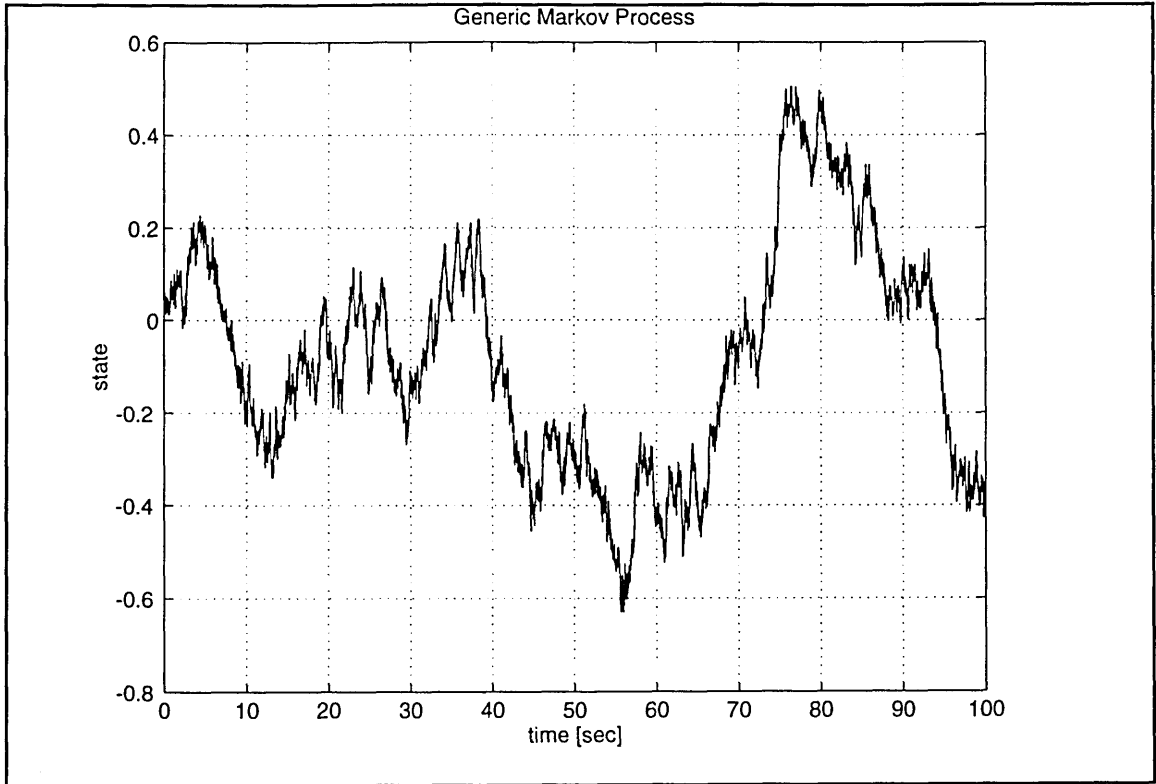
Since the system consists of two separate navigators with only one external measurement source, the two state error estimates have to be coupled with respect to one another to ensure that the position estimate of the helmet is physiologically feasible when compared to the T-38 EGI position. The methodology used for this is to navigate the two systems independently, but calculate their errors jointly in the aforementioned Kalman Filter. The lever arm between the two will be calculated, about which a Markov process will be used to allow for small, uncoupled head motion relative to the aircraft. The purpose of this thesis is to determine how well this stochastic coupling scheme works for the IHMCS.

### 4.4.1 Description of a Markov Process

A Markov process is mathematically described by the following differential equation:

$$\dot{x} = \frac{-1}{\tau}x + w \quad (4.29)$$

where the state  $x$  (the relative position error estimate in the case of the filter) is correlated in time and the noise exists in the state derivative, not the state itself. This has the effect of producing a gradually wandering variable about a given expected value, as seen in Figure 4.4 (where  $E[x] = 0.0$ ):



**Figure 4.4** Generic Markov Process:  $\tau = 10$ ,  $\sigma = 1$ ,  $E[x] = 0$

#### 4.4.2 Attempt to Ensure Markov Process Remains Strictly Bounded

By making the time constant a function of the Markov process state, no matter what the value of the state is within its variance limit, that the Markov process will not push it beyond that boundary by implementing the following linear relationship for  $\tau$ :

$$\tau = \left| \tau_{max} - \left( \frac{\tau_{max} - \tau_{min}}{x_{max}} \right) |x| \right| \quad (4.30)$$

The above relationship effects  $\tau$  when  $x$  approaches the boundary by decreasing  $\tau$  to  $\tau_{min}$ , thereby increasing the correlation in time at the present value of

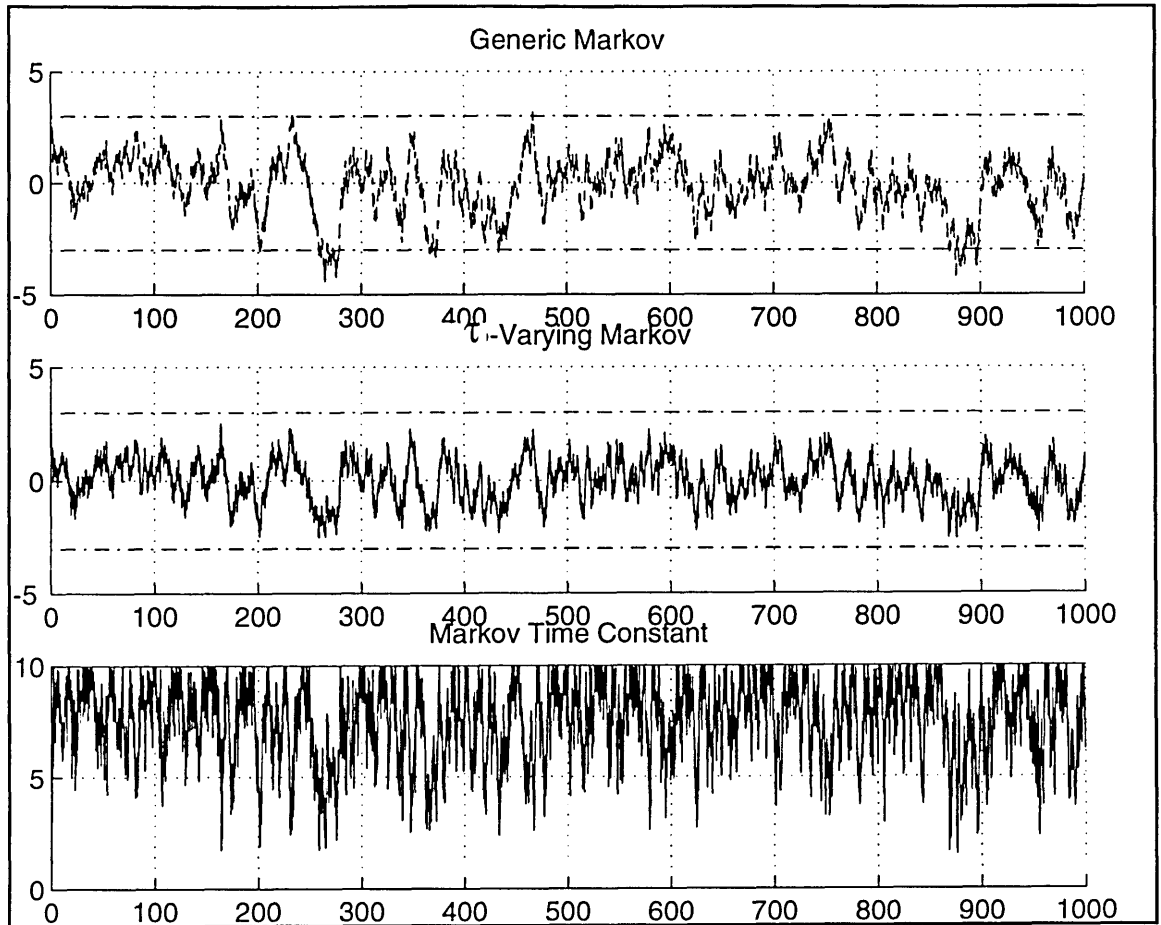
x. Alternatively, when  $x$  approaches  $E[x]$ ,  $\tau$  increases to  $\tau_{\max}$ , thus decreasing the correlation in time at the present value of  $x$ . The resulting behavior is that of a process which is more correlated closer to the boundary, making it more likely to return to the  $E[x]$ . Therefore, the algorithm was implemented to make sure that the Markov component doesn't add to the error, pushing the pilot's head outside the physiological limits. The algorithm performance was also examined to see if it improves the performance of our coupling algorithm.

A comparison of two Markov processes, one without Equation (4.30) implemented and one with it implemented, starting near the intended boundary can be seen in Figure 4.5 on the next page. It is clear that the  $\tau$ -varying scheme stays within the limits of  $x$  (in the figure at  $\pm 3$ ), while the constant- $t$  scheme strays outside of the limits multiple times.

### 4.4.3 Implementation

#### MATLAB Test

The relative state matrices will consist of the Markov time constant terms, as seen in Equation (4.24). These time constants will be calculated using Equation (4.30), and that should create the desired effect of bounding the errors of the head navigator with respect to the aircraft navigator to a box of predetermined dimensions.



**Figure 4.5** Tau-constant and Tau-varying Markov Processes

### Simulation Implementation

After implementing this algorithm which resulted in a tighter bound for the MATLAB trial above, the testing yielded no discernible improvement. See Section 5.3 for further details.

# Chapter 5.0

## Data Analysis

### 5.1 Simulated Truth Data

The trajectory for the aircraft generated for the data analysis starts with the aircraft in a free-fall nose-over maneuver, after which the aircraft enters a 2.5-g pull-up maneuver, followed by a sequence of pull-up and nose-over maneuvers (similar to a “nap-of-the-earth” terrain following trajectory). The trajectory reflects one of the worst conditions under which to initialize the system: no initial specific force vector. However, it is followed by a highly dynamic maneuver followed by smaller pull-up and nose-overs to create a specific force vector on both the pilot and the helmet to provide the information necessary to couple the two navigators’ attitude estimates, despite the pilot’s motion. Without the maneuvers (e.g. in straight and level flight) the roll and pitch attitude error would converge much like the data represented below, but the yaw (azimuth) error would increase at a rate equal to that of the yaw-axis gyro bias.

The pilot trajectory is a generic scan about the cockpit and the field of view, similar to the behavior of a pilot during visual search for and identification of a target. The same trajectories were used for all runs, with varying initial errors on all the navigator output states and inertial instrument error states.

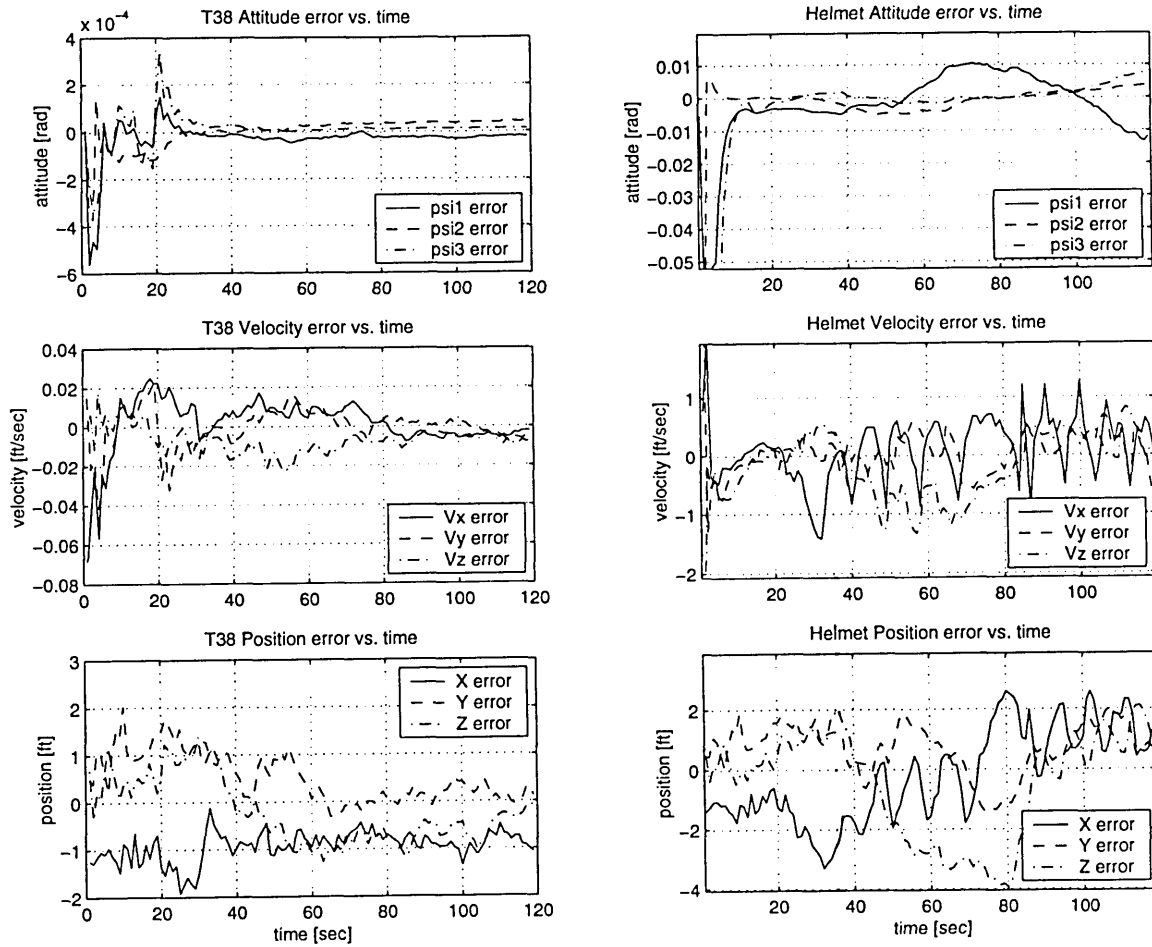


## 5.2 Filter Performance

To analyze filter performance, MATLAB scripts were written to operate in two situations: single data run and multiple data runs which are in turn analyzed as a Monte Carlo data set. The Single Data Run is used to demonstrate pointing accuracy, position update accuracy, and velocity estimate accuracy on both IMUs, primarily for debugging purposes. The Monte Carlo analyses determine how well the system works under various noise sequences and initial error conditions, and produce the most useful data. Both analyses are described in the following paragraphs.

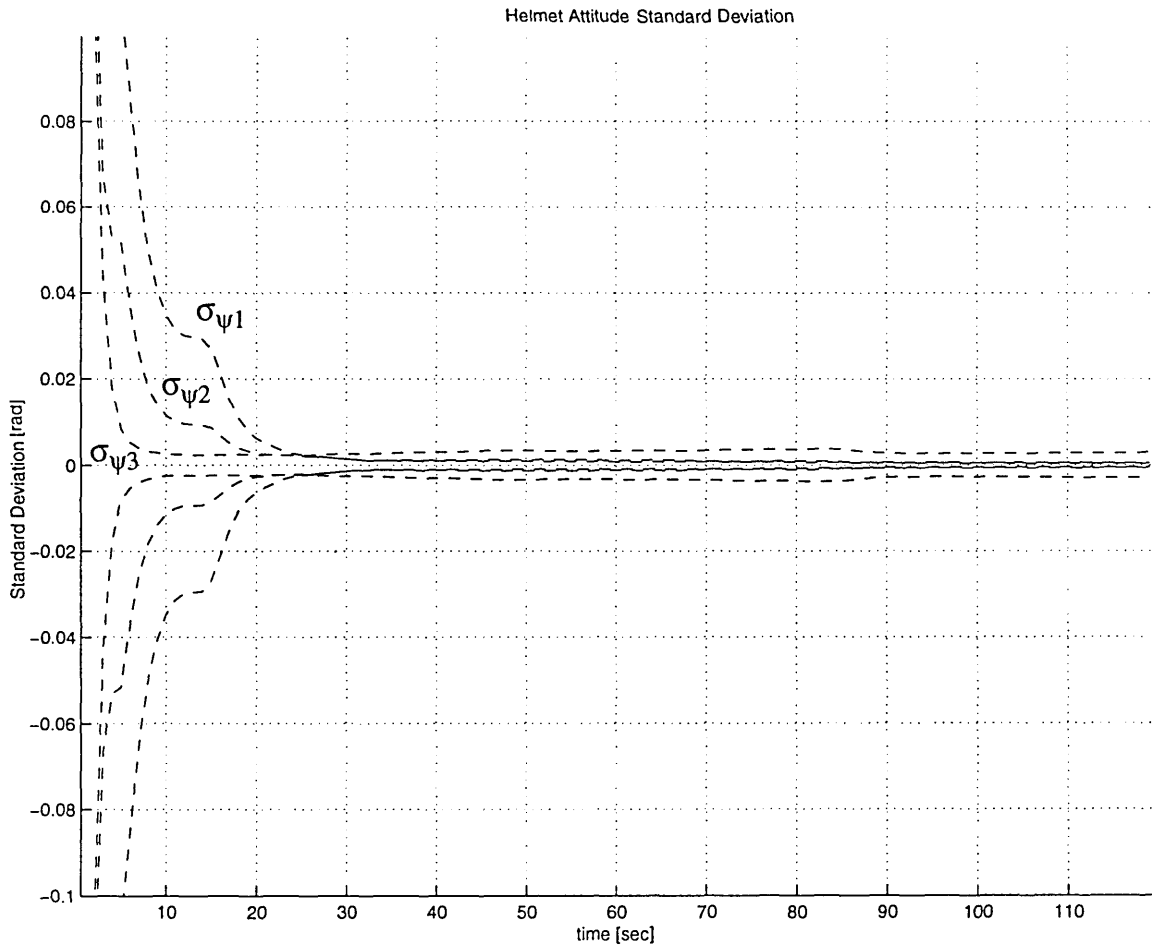
### 5.2.1 Single Data Run Analysis

A single data run consists of running the Kalman Filter described in Chapter 4.0 on the data coming from the simulation described in Chapter 3.0. The updates in the filter were performed at 1Hz, the frequency at which the GPS measurements are obtained. Each GPS update consists of the difference between GPS position and velocity and the navigated T-38 position and velocity. Each relative update between the IMUs consists of the difference between the navigation position of the EGI and the MMISA which is further differenced with the stochastic lever arm (containing the Markov process used to couple the two navigators) between the EGI and the MMISA. To evaluate the navigation solution, the estimates of position, velocity, and attitude errors were combined with the navigated position, velocity, and attitude and compared to the true position, velocity and attitude values, producing a final result of the position velocity and attitude error as seen in Figure 5.1.



**Figure 5.1** Single Run T-38 and Helmet Navigation Errors

Also of significance is the calculated standard deviation of each of the error states, as found in the P-matrix of the filter. The calculated filter standard deviation of the attitude error for the helmet MMISA which correspond to the run in Figure 5.1 are presented in Figure 5.2.



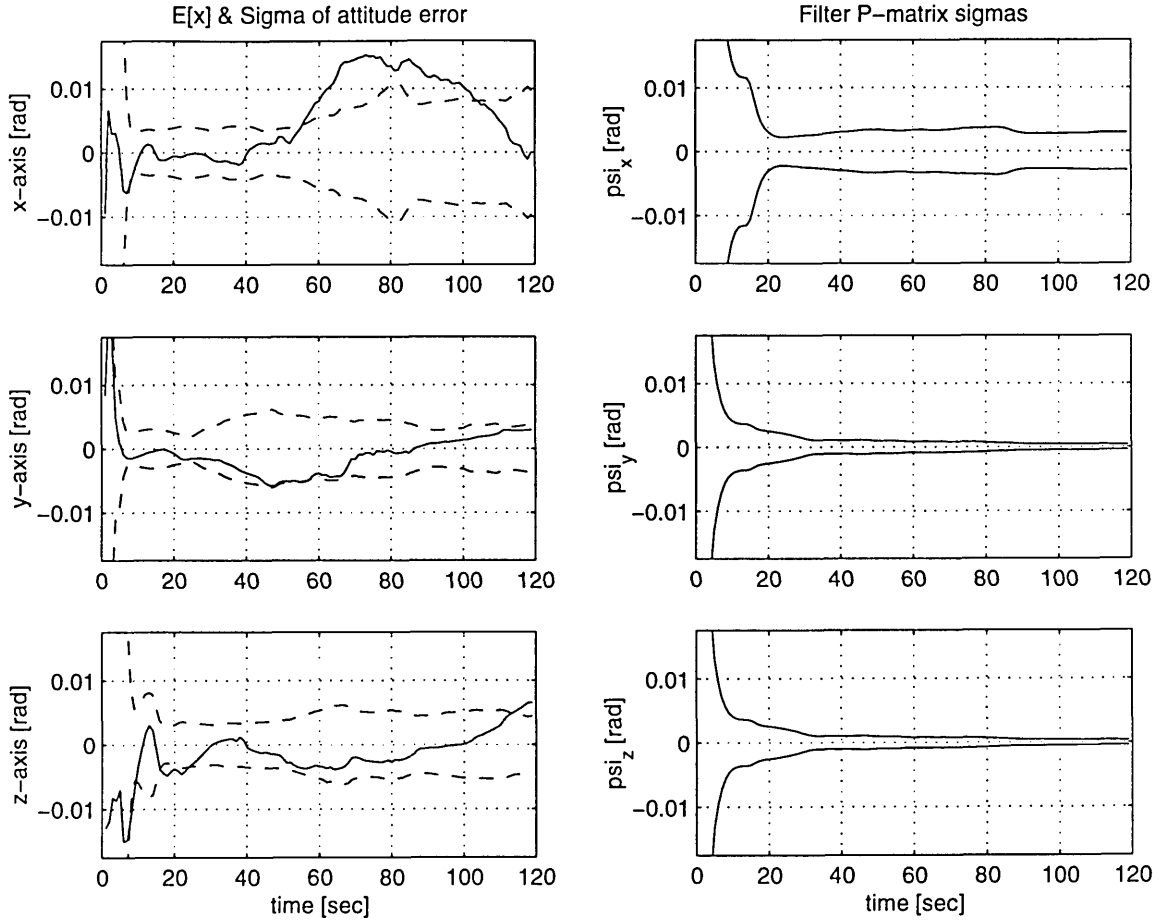
**Figure 5.2** Single Run P-Matrix Attitude Calculated Standard Deviation

The single runs are particularly useful for debugging, but they do not adequately represent the system performance under varying initial errors and conditions. Since this system is dependent upon a stochastic coupling between the T-38 and helmet navigation solutions (in the form of the Markov relative position states in the PHI and H-matrices), an analytical procedure known as a Monte Carlo analysis, which uses sample random variables for the error states represented in the filter, was used to quantify the performance of the system.

### 5.2.2 Twenty-Run Monte Carlo Analyses

For a better performance evaluation of the Markov process linking the helmet navigator to the aircraft navigator, a 20-sample Monte Carlo analysis was performed, with random initial errors based on the sigmas specified for all state variables. The “roller coaster” trajectory described in Section 5.1 was used, as well as the general scan pilot motion trajectory, for a 120 second duration in simulation time.

The three relative-state Markov process time constants in the Kalman Filter were varied using 5, 10, 50, and 100 seconds to analyze sensitivity to this parameter caused by the coupling methodology (see Appendix B for full results). Since the filter post-processed the simulation data, the same 20-sample Monte Carlo set was used for each value assigned to the Markov process time constants. The resulting runs were then analyzed to determine the attitude error sampled means and standard deviations about each axis as a function of time. These quantities are plotted adjacent to the filter’s calculated standard deviation (the square root of the diagonal of the P-matrix) for comparison. The resulting plot for filter Markov time constant of  $\tau = 10$  seconds is shown in Figure 5.3.



**Figure 5.3** Monte Carlo Sample Statistics for Filter Markov  $\tau = 10$  seconds

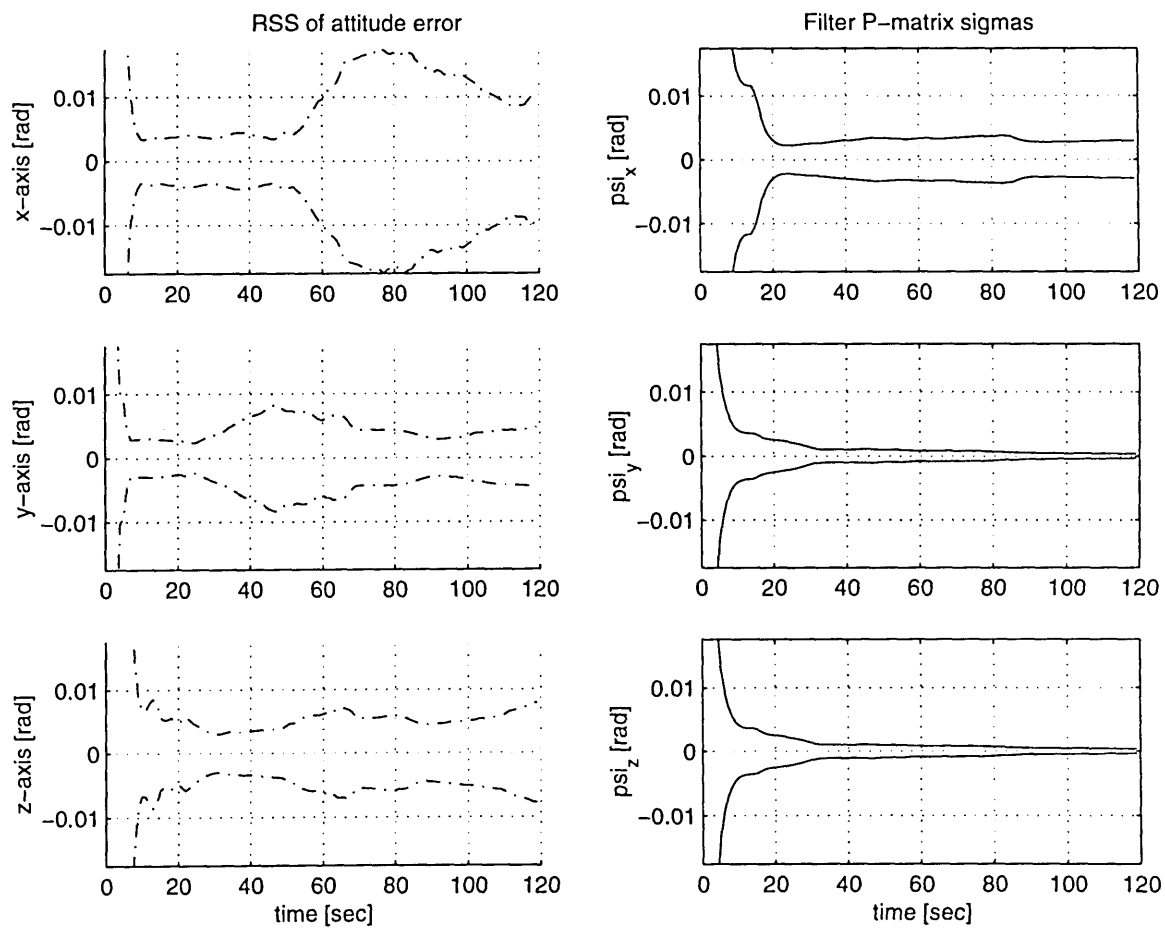
In addition, the root sum squared (RSS) was calculated from the sample data at each time step as shown in Equation (5.1):

$$\sqrt{\frac{\sum x_n^2}{n}} \quad (5.1)$$

which assumes the expected value is zero in its calculation, unlike the standard deviation (shown in Equation (5.2) for comparison).

$$\sqrt{\frac{\sum (x_n - E[x])^2}{n}} \quad (5.2)$$

The RSS shown in Figure 5.4 was also compared to the same attitude error standard deviation estimated by the Kalman Filter shown in the previous figure.



**Figure 5.4** Monte Carlo sample RSS for filter Markov  $\tau = 10$  seconds

For both the sample statistics and the RSS, the P-matrix standard deviations create a more tightly bounded envelope for the expected value of the attitude

error. Since the calculated standard deviation from the P matrix does not match the Monte Carlo error standard deviation, this discrepancy suggests that the filter is sub-optimal, so a gain modification scheme may improve the errors and reduce the discrepancy. The plots for the time constants other than  $\tau = 10$  seconds can be found in Appendix B.

To analyze the trend of the sample standard deviation and RSS versus the time constant chosen for the Markov process in the post processing Kalman Filter, their final values (at  $t = 120$  seconds) and maximum values after  $t = 15$  seconds (the settling time for the system errors) were tabulated and plotted versus the filter's Markov time constant. The table comparing the sample statistics at  $t = 120$  seconds from both the covariance matrix and the averaged data can be seen in Table 5.1.

$\tau$	Quantity	$\Psi_x$ [rad]	$\Psi_y$ [rad]	$\Psi_z$ [rad]
<b>5</b>	final $\sigma$ of Monte Carlo data	0.0104	0.0039	0.0045
	final RSS of Monte Carlo data	0.0102	0.0048	0.0079
	mean of Monte Carlo data	0.0001	0.0030	0.0066
	mean $\sigma$ of Kalman Filter P	0.0029	0.0004	0.0005
<b>10</b>	final $\sigma$ of Monte Carlo data	0.0097	0.0038	0.0044
	final RSS of Monte Carlo data	0.0094	0.0047	0.0078
	mean of Monte Carlo data	0.0	0.0029	0.0065
	mean $\sigma$ of Kalman Filter P	0.0029	0.0004	0.0005
<b>50</b>	final $\sigma$ of Monte Carlo data	0.0126	0.0038	0.0047
	final RSS of Monte Carlo data	0.0122	0.0045	0.0082
	mean of Monte Carlo data	0.0003	0.0025	0.0069

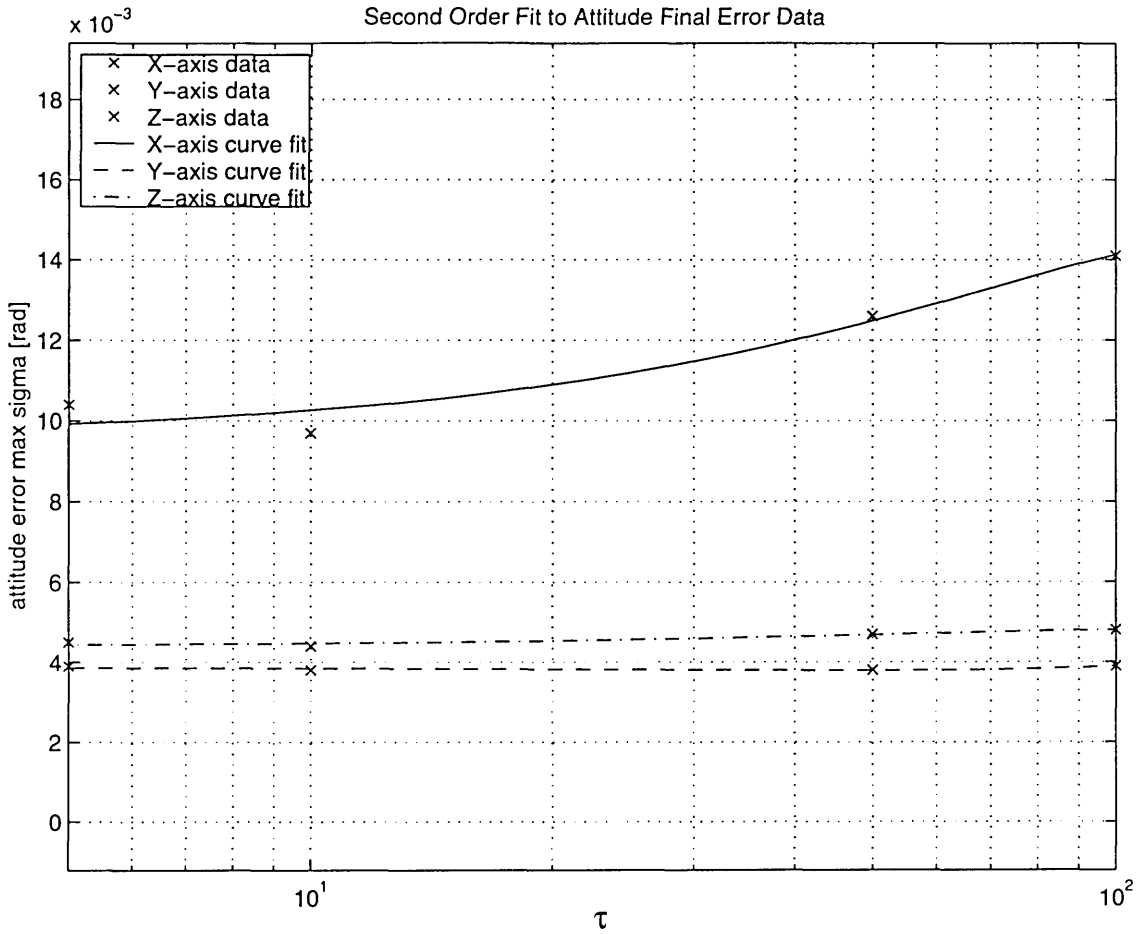
**Table 5.1 Summarized Sample Statistics Data at  $t_{final}$**

$\tau$	Quantity	$\Psi_x$ [rad]	$\Psi_y$ [rad]	$\Psi_z$ [rad]
	average $\sigma$ of Kalman Filter P	0.0028	0.0004	0.0005
<b>100</b>	final $\sigma$ of Monte Carlo data	0.0141	0.0039	0.0048
	final RSS of Monte Carlo data	0.0137	0.0044	0.0085
	mean of Monte Carlo data	0.0004	0.0023	0.0071
	mean $\sigma$ of Kalman Filter P	0.0027	0.0003	0.0005
<b>Eq. (4.30)</b>	final $\sigma$ of Monte Carlo data	0.0119	0.0033	0.0048
	final RSS of Monte Carlo data	0.0117	0.0041	0.0086
	mean of Monte Carlo data	0.0015	0.0025	0.0073
	mean $\sigma$ of Kalman Filter P	0.0028	0.0004	0.0005

**Table 5.1 Summarized Sample Statistics Data at  $t_{\text{final}}$**

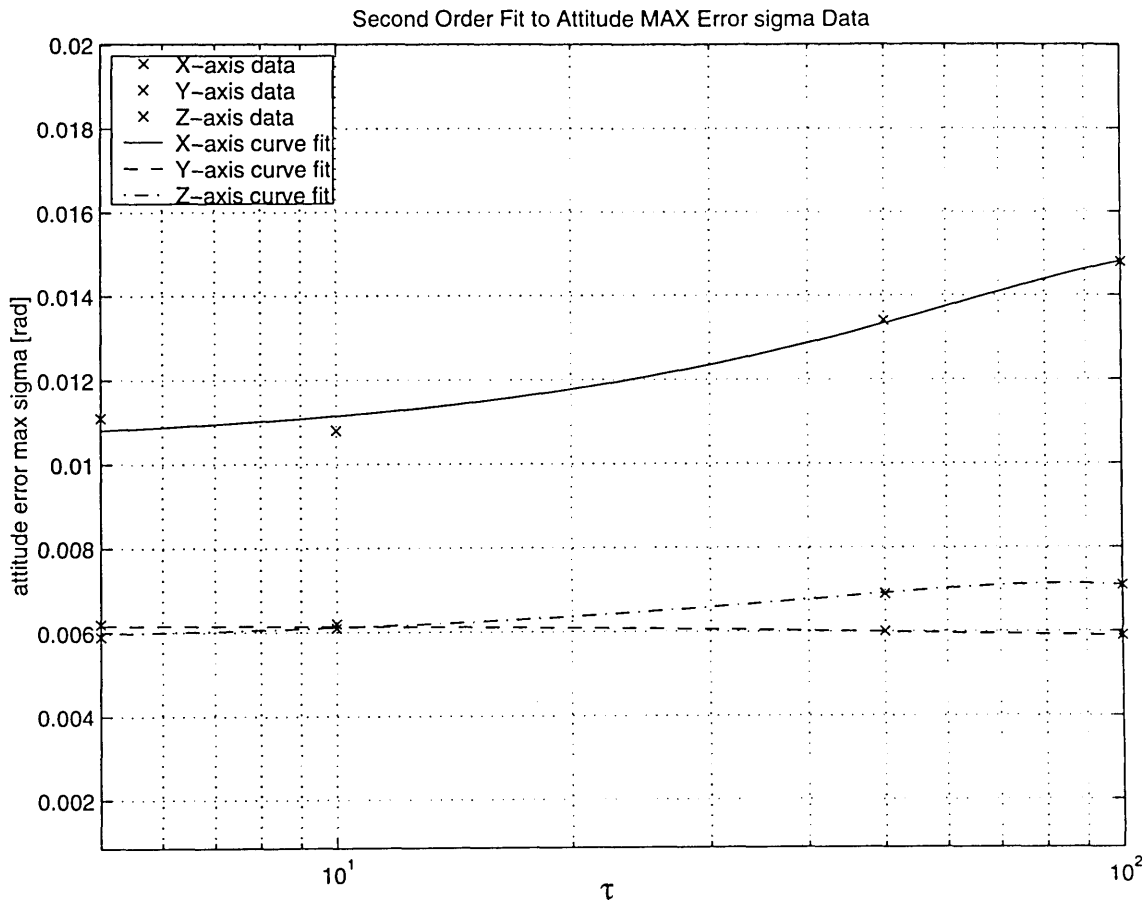
From the above tabulated data and the following plot, it is apparent that the bounding algorithm in Equation(4.30) does not add any additional performance over a constant  $\tau$ , and that  $\tau = 10$  resulted in the minimal  $\sigma_\psi$  (the results plotted in Figure 5.3 and Figure 5.4). A plot of the sample statistics at  $t = 120$  seconds can be seen in Figure 5.5.





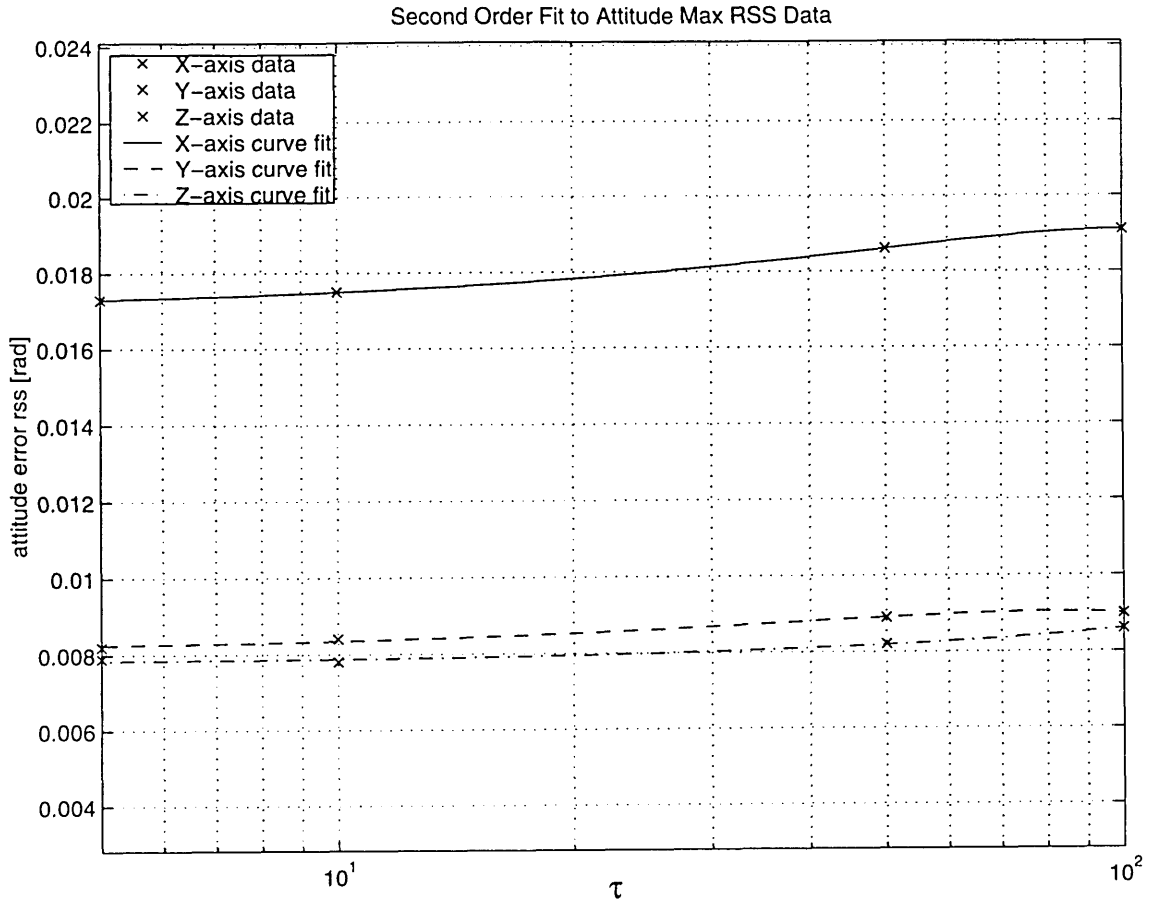
**Figure 5.5** Final Attitude Sigma vs Markov Time Constant

The trend can clearly be seen to increase as the time constant is increased to 50 and 100 seconds in the Markov Process. Figure 5.6 clearly shows the trend exists for the maximum sigma after the calculated error settling time ( $t = 15$  seconds):



**Figure 5.6** Maximum Attitude Error Sigma after settling vs. Markov Time Constant

The maximum RSS after the attitude error settling time, as the Markov Time constant varies, can be seen to have a similar trend to the two previous plots as well in Figure 5.7, although the trend is less pronounced.



**Figure 5.7** Maximum RSS of Attitude Error after settling vs. Markov Time Constant

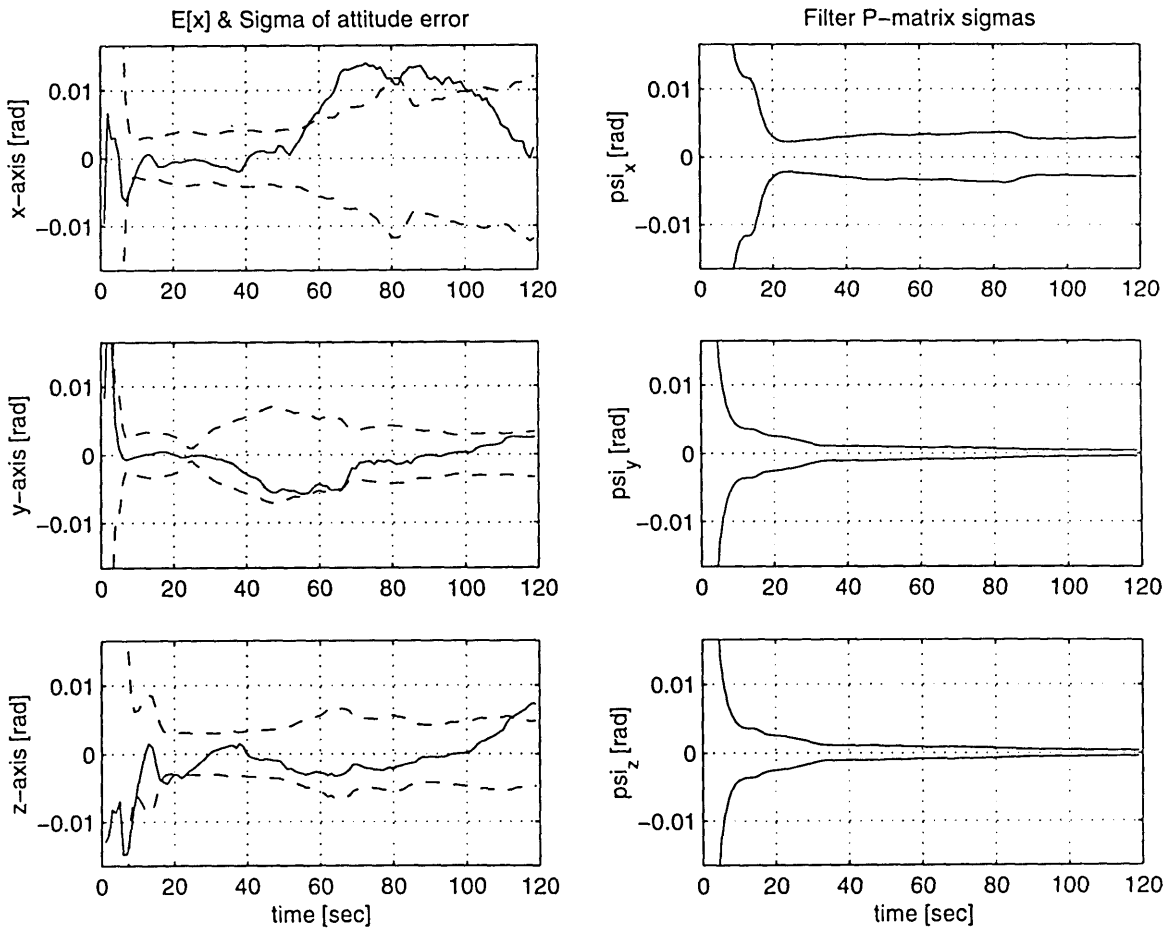
The *a priori* information and the Markov process provides enough information for the attitude error to be bounded, thus indicating that a lower-grade IMU with a stochastically defined relative position to another higher grade IMU can be configured using relative Markov states in a Kalman Filter to determine attitude (with an appropriate trajectory).

It is apparent from these plots that under the given conditions (pitch over and pullout maneuver with pilot scanning), it takes 10 to 15 seconds for the attitude error to be estimated within reasonable limits. The other trials with different

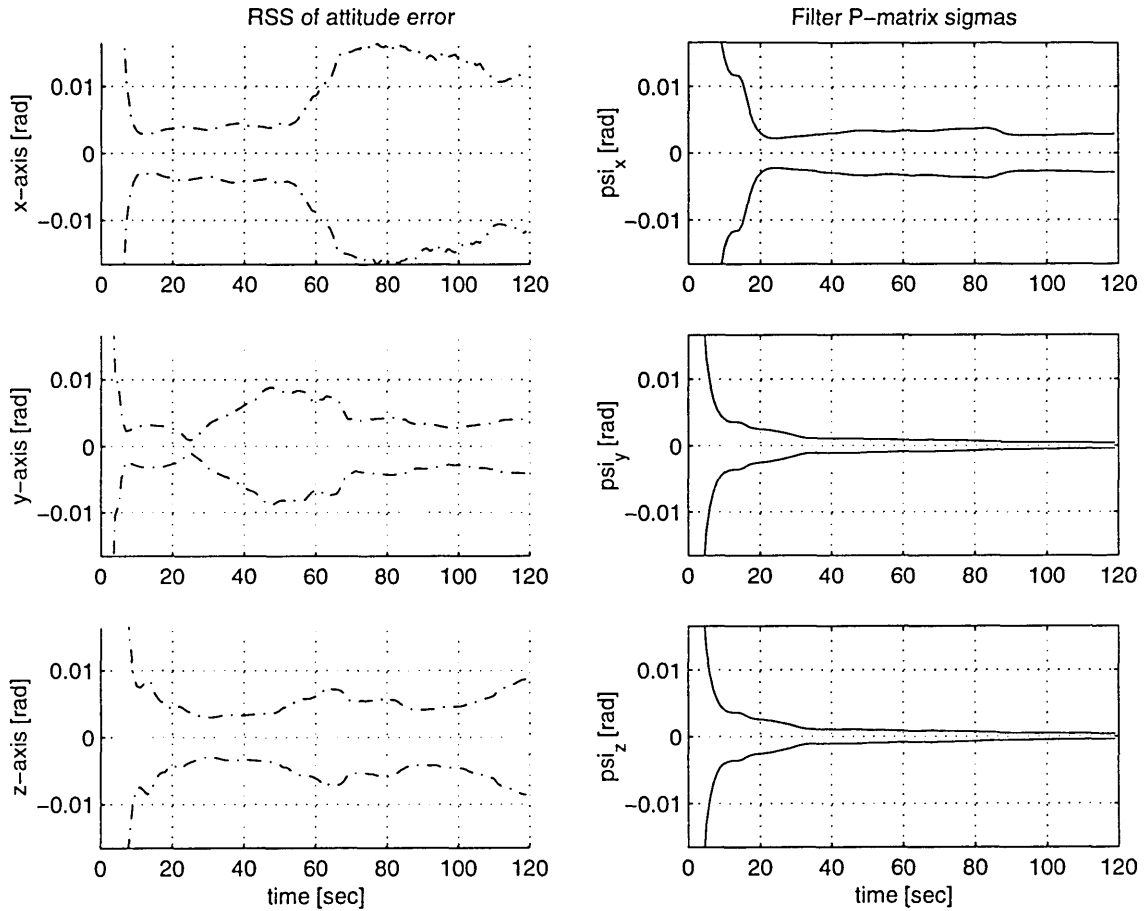
and varying time constants in the relative Markov bounding states confirms this settling time as well. However, one feature of the data worth noting is the settling time on the y-axis attitude error. As  $t$  is increased, the settling time after 10 to 15 seconds to get back into the calculated sigma of the filter is shortened as shown in Appendix B.

### 5.3 Bounding Algorithm Performance

Monte Carlo runs were also analyzed using the Markov process bounding algorithm stated in Section 4.4, with  $\tau_{\max} = 100$  and  $\tau_{\min} = 1$ . The results are presented in Figure 5.8 and Figure 5.9.



**Figure 5.8** Monte Carlo  $\sigma$  Results of Azimuth Accuracy,  $\tau$  varying



**Figure 5.9** Monte Carlo RSS Results of Azimuth Accuracy,  $\tau$  varying

Both by comparing the previous two plots with Figure 5.3 and Figure 5.4 and by examining the tabulated data in Table 5.1, it is evident that the implementation of Equation (4.30) produces no apparent benefits regarding attitude error reduction.

# Chapter 6.0

## Conclusion and Recommendations

### 6.1 Results

The coupling of the HMIMU to the aircraft IMU via the Kalman Filter described in Chapter 4.0, which provides position coupling information, has been shown to be effective after a 10-15 second settling period under poor initial dynamic conditions, depending on desired accuracy (see previous chapter for specific results). The goal of the simulation was to show that the same level of accuracy could be achieved as that of the Honeywell AMTT, which was 4 milliradians pointing accuracy. This system, after the settling period, has shown repeatable accuracies between 4 and 11 milliradians under the given testing conditions, which suggests that this technology can eventually be competitive with magnetic trackers or replace them pending additional development.

### 6.2 Future Work

#### 6.2.1 Initial Transient and Error Reduction

Additional work should focus on reducing the initial 10 to 15 second transient as well as reducing the overall error to obtain acceptable pointing accuracy that is competitive with other head tracking systems. Possible solutions would be an algorithm to introduce gain matrix modifications so that the initial transient could be reduced to less than 5 seconds, by analyzing the aircraft dynamics

during system initialization. The goal of this work should be to get the duration of error transients at system startup as close to other technologies (e.g. sonic, optical, and magnetic) as possible, to prove that the IMU on the helmet can be a viable competitor in the vehicle-based head tracker marketplace. Additionally, the dual-IMU setup can be used by a foot soldier, where the other types of trackers are impractical to use because they rely on equipment mounted to the surrounding environment (i.e. cockpit or room). A shorter settling time for pointing accuracy makes the system more versatile for the foot soldier application as well.

### **6.2.2 Filter Reset Feedback**

The filter as it exists now has no resets-meaning that the navigator is not corrected by the error estimate of the filter, thus the filter output error can grow unbounded under certain conditions. Resets prevent this unbounded growth from occurring and would be desirable to implement in a hardware system. This implementation would require the MATLAB filter be ported to the C programming language and implemented as part of the C-sim simulation.

### **6.2.3 Sensor Placement and Lever Arm Sensitivity**

Additional work on packaging and sensitivity to mounting on a helmet would be desirable to determine where best to mount the sensors, and whether or not to use in-plane, normal-to-plane, or a mixture of both types of sensors. The results of this work should provide a map of best sensor locations on the hel-

met for both vehicle and foot soldier applications, based on pointing accuracy and an error sensitivity study of sensor placement.

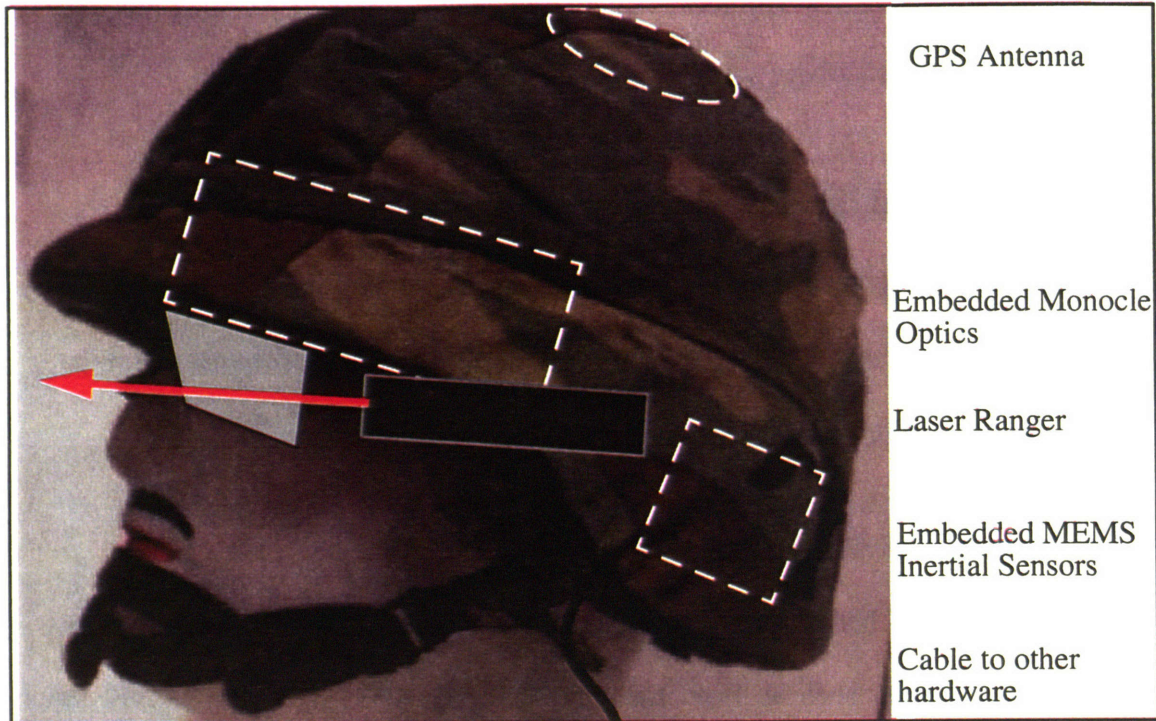
## **6.3 Additional Technology Applications**

The following technology areas are suggestions for additional applications of multiple IMU filters with appropriate *a priori* knowledge to formulate a similar filter as to the one developed in this thesis.

### **6.3.1 Foot soldier HMD and Fire Control System**

This type of head tracker would be ideally suited for operation in a non-enclosed area, such as head tracking of a foot soldier. An IMU could be mounted within a helmet, with a GPS antenna, and attached to a GPS receiver and a wearable computer. This would enable the soldier to have an HMD overlaying targeting, communication, and navigation symbology over his field of view. When combined with a LASER ranging device (also rigidly mounted to the helmet) location of targets would be as simple as looking at them. The soldier would have to physically move around to bound azimuth drift as shown by the Personal Inertial Navigator System (PINS) project at the CSDL. One possible implementation is shown in Figure 6.1:





**Figure 6.1** Soldier Helmet with HMIMU Technology Applied to Ground Maneuvers

For a view through the monocular with navigation and targeting symbology, see Figure 6.2.



**Figure 6.2** Soldier's Eye View of HMIMU used with HMD for Ground Forces

As the previous two figures suggest, the HMIMU applied to the foot soldier would be a welcome addition to the information network enhanced battlefield equipped with precision guided munitions. Soldiers could locate, target and relay targeting coordinates to precision munitions, which could clear the way for the foot soldiers with minimal threat to the soldiers themselves, and minimal ammunition use. The targets could be pinpointed within the navigation coordinates of choice in real time, and could be eliminated or disabled shortly thereafter.

## 6.3.2 Tether Control

### Space Tethers

Tethers for space missions have become thrust into the limelight lately, with the possibility of generating power and controlling orbital dynamics in unique and more conservative ways. The tethers' dynamics, however, become quite complex in the orbital environment, especially when the tethers are quite long due to the nature of orbital dynamics and planetary magnetic fields. It would be possible to place small IMUs distributed throughout the tether, to aid in observability of the modes and other overall dynamics of the tether and adequate control of the thrusting systems at both ends of the tether.

### Helicopter Tethers

Helicopter tethers on autonomous vehicles is a simpler problem than the space tethers, and is yet another application to which this multiple IMU solution could be applied. By placing an IMU in the helicopter, and placing an IMU at the "claw" end of the tether, the helicopter could be controlled to accurately extend the tether and pick up a tracked object. Such systems could be placed on autonomous or remotely piloted vehicles to recover items from hazardous environments.

## 6.3.3 Generic Application

Any application where there is an *a priori* knowledge of dynamic bounds on additional inertial navigators relative to a main "parent" navigator (which is updated via another navigation system such as GPS) is appropriate for the implementation of this filter. This technique will become more feasible and

applicable to additional systems as inertial sensors continue to reduce in both size and cost and as computers increase their computational capacity compared to their required physical volume. When an entire IMU can fit on a single piece of silicon and be manufactured as an all-in-one chip is when this technique will really see its greatest application base blossom due to the availability of less-expensive components.

For the present time, however, examples such as those states above can be used as test beds to further develop low-cost multiple-IMU filters with bounded relative dynamics.



## References

- [1] Axt, W. E. "Head Tracking Accuracy in View of Boresighting and Parallax Compensation." *SPIE International Society for Optical Engineering Helmet Mounted Displays II*. Vol. 1290, April 19-20, 1990. pp. 192-203.
- [2] Barbour, N. "Inertial Sensor Technology Trends." *Draper Technology Digest*. Vol. 3, 1999. pp. 5-13.
- [3] Britting, K.R. *Inertial Navigation Systems Analysis*. John Wiley & Sons: New York, NY, 1971.
- [4] Burcham, M. A. "Head-Mounted Display Technology For Use on the Advanced Concept Technology (ACT) II- Integrated Maintenance & Logistics Soldier System (IMLSS)." *SPIE Conference on Helmet and Head-Mounted Displays III*. Vol. 3362, April 13-14, 1998. pp. 276-283.
- [5] Chapman, F. W. "The Advent of Helmet-Mounted Devices in the Combat Aircraft Cockpit." *SPIE International Society for Optical Engineering: Helmet Mounted Displays III*. Vol. 1695, April 21-22, 1992. pp. 26-37.
- [6] Foxlin, E. "Inertial Head-Tracker Sensor Fusion by a Complimentary Separate-Bias Kalman Filter." *IEEE Proceedings of the Virtual Reality Annual International Symposium*. March 30-April 3, 1996. pp. 185-194.
- [7] Foxlin, E. "Miniature 6-DOF inertial system for tracking HMDs." *SPIE Conference on Helmet and Head-Mounted Displays III*. Vol. 3362, April 13-14, 1998. pp. 214-228.
- [8] Gelb, A. *Applied Optimal Estimation*. M.I.T. Press: Cambridge, MA, 1974.
- [9] Gershenfeld, N. *The Nature of Mathematical Modeling*. Cambridge University Press: Cambridge, England, 1999.
- [10] Kayton, M. "Avionics Navigation Systems." Second Edition. Wiley & Sons, New York, 1997.
- [11] Kim, D., Richards, S. W., and Caudell, T. P. "An Optical Tracker for Augmented Reality and Wearable Computers." *IEEE Proceedings of the Virtual Reality Annual International Symposium*. March 1-5, 1997. pp. 146-150.
- [12] Kourepenis, A. "Performance of Small, Low-Cost Rate Sensors for Military and Commercial Applications." *Draper Technology Digest, Vol. 2*. 1988. pp. 85-92.

- [13] Kranz, Y. "Implementation of Lessons Learned in Design and Evaluation of Displays for Helmet Mounted Display Systems." *SPIE Conference on Helmet and Head-Mounted Displays III*. Vol. 3362, April 13-14, 1998. pp. 156-163.
- [14] Loyd, R. B. "Head-Mounted Display Systems and the Special Operations Soldier." *SPIE Conference on Helmet and Head-Mounted Displays III*. Vol. 3362, April 13-14, 1998. pp. 244-251.
- [15] McKern, R.A.. "A Study of Transformation Algorithms for Use in a Digital Computer." S.M. Thesis, MIT Department of Aeronautics and Astronautics, 1968.
- [16] Melzer, J. E. *Head Mounted Displays: Designing for the User*. McGraw-Hill: New York, 1997. pp. 147-173.
- [17] Proctor, P. "Head Tracker Advances 'Look and Shoot' Tactics." *Aviation Week & Space Technology*. June 14, 1999. p. 194.
- [18] Osgood, R. K. "JSF Integrated Helmet Audio-Visual System Technology Demonstration Results." *SPIE International Society for Optical Engineering*. Vol. 3058, April 21-27, 1997. pp. 332-334.
- [19] Schmidt, G. T. "INS/GPS Technology Trends for Military Systems." *Draper Technology Digest*. Vol. 2, 1998. pp. 143-154.
- [20] Stevens, Brian L., Lewis, Frank L.. *Aircraft Control and Simulation*. John Wiley and Sons: New York, 1992.
- [21] Velger, Mordekhai. *Helmet Mounted Displays and Sights*. Artech House: Boston, 1998.



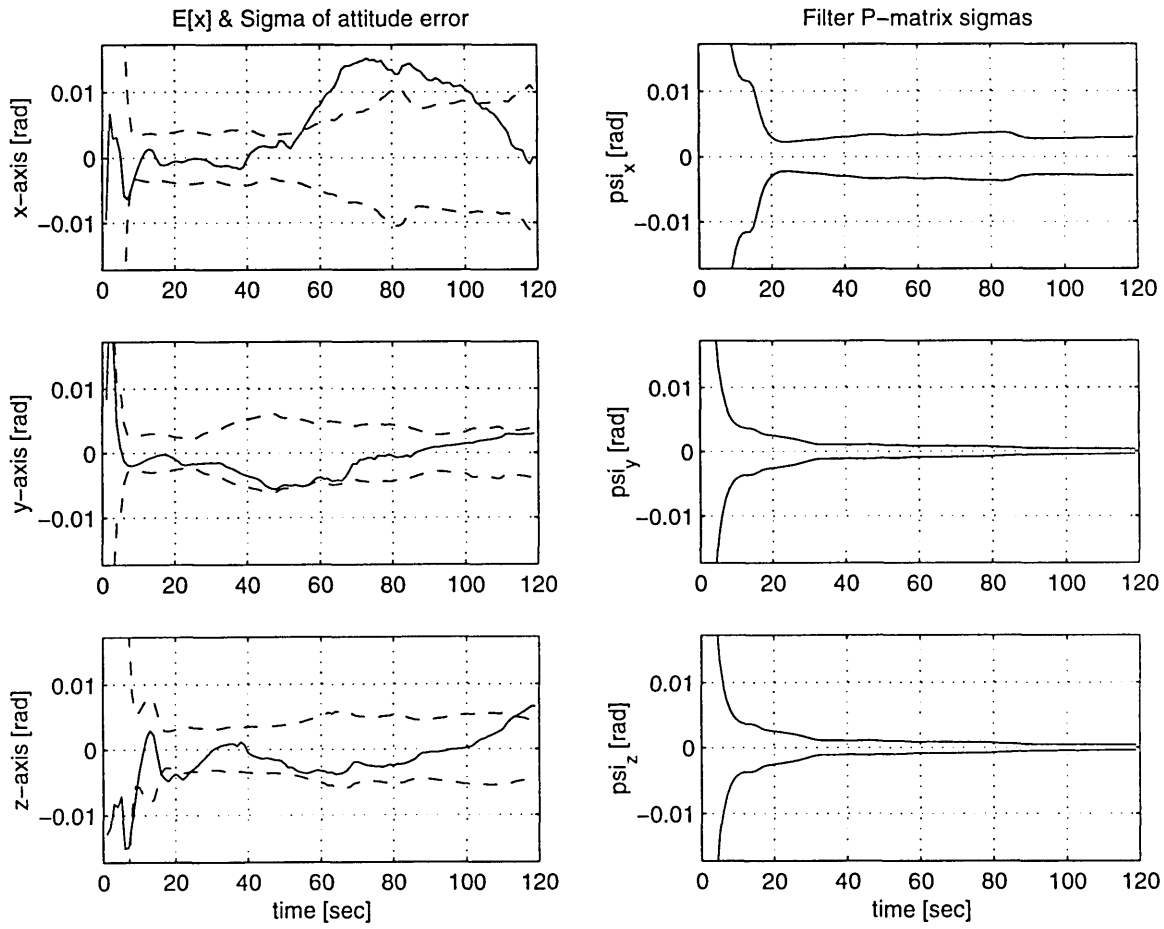


Note that in equations (A.2) and (A.3),  $r_j^s$  represents the  $j$ -th component of the lever arm from the T-38 center of gravity to the three-dimensional stochastic mean of the bounding manifold, coordinatized in ECEF coordinates. Also note that  $c_{ij}$  are the elements of the direction cosine matrix from the body frame to the ECEF frame.

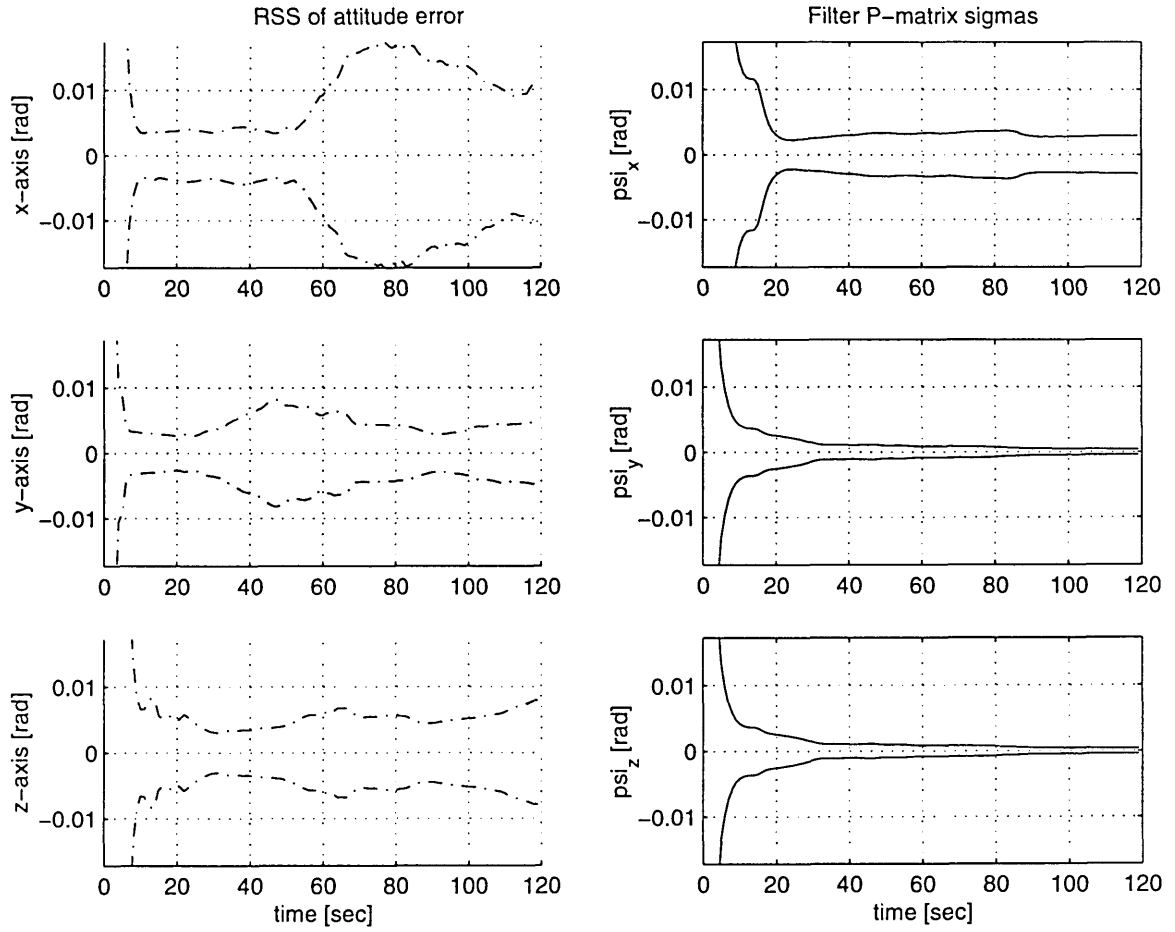
# Appendix B

## Additional Monte Carlo Plots

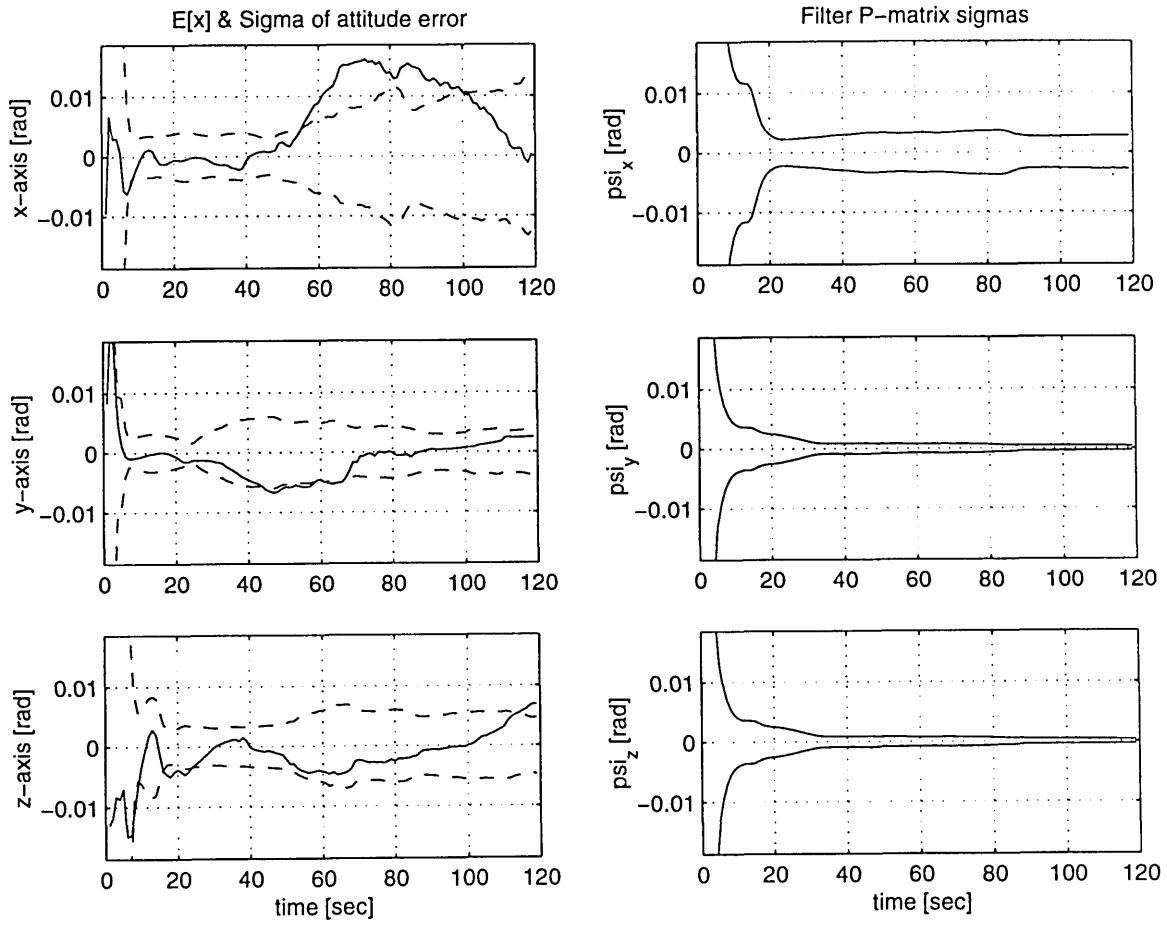
The plots for the other chosen non-varying time constants are as follows:



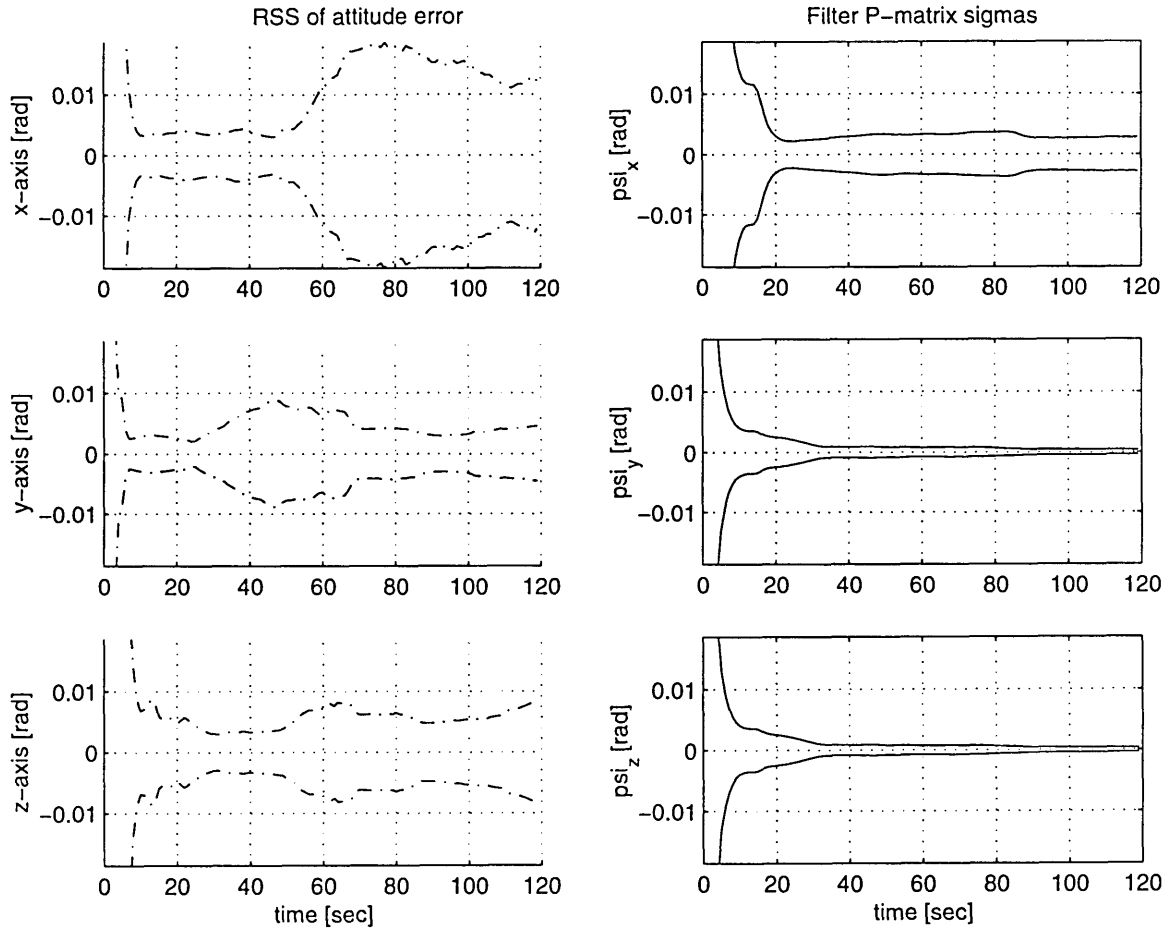
**Figure B.1:** Monte Carlo  $\sigma$  Results of Azimuth Accuracy,  $\tau = 5$



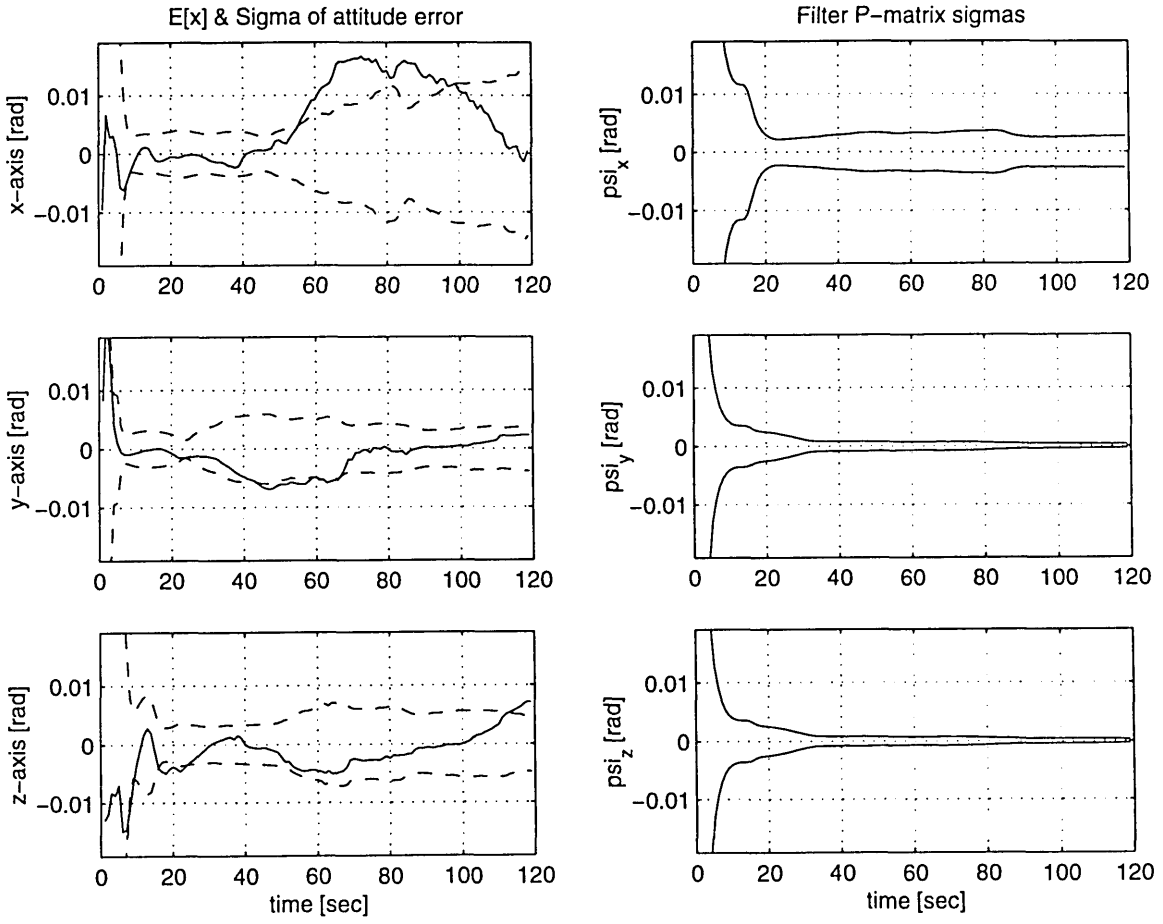
**Figure B.2:** Monte Carlo RSS Results of Azimuth Accuracy,  $\tau = 5$



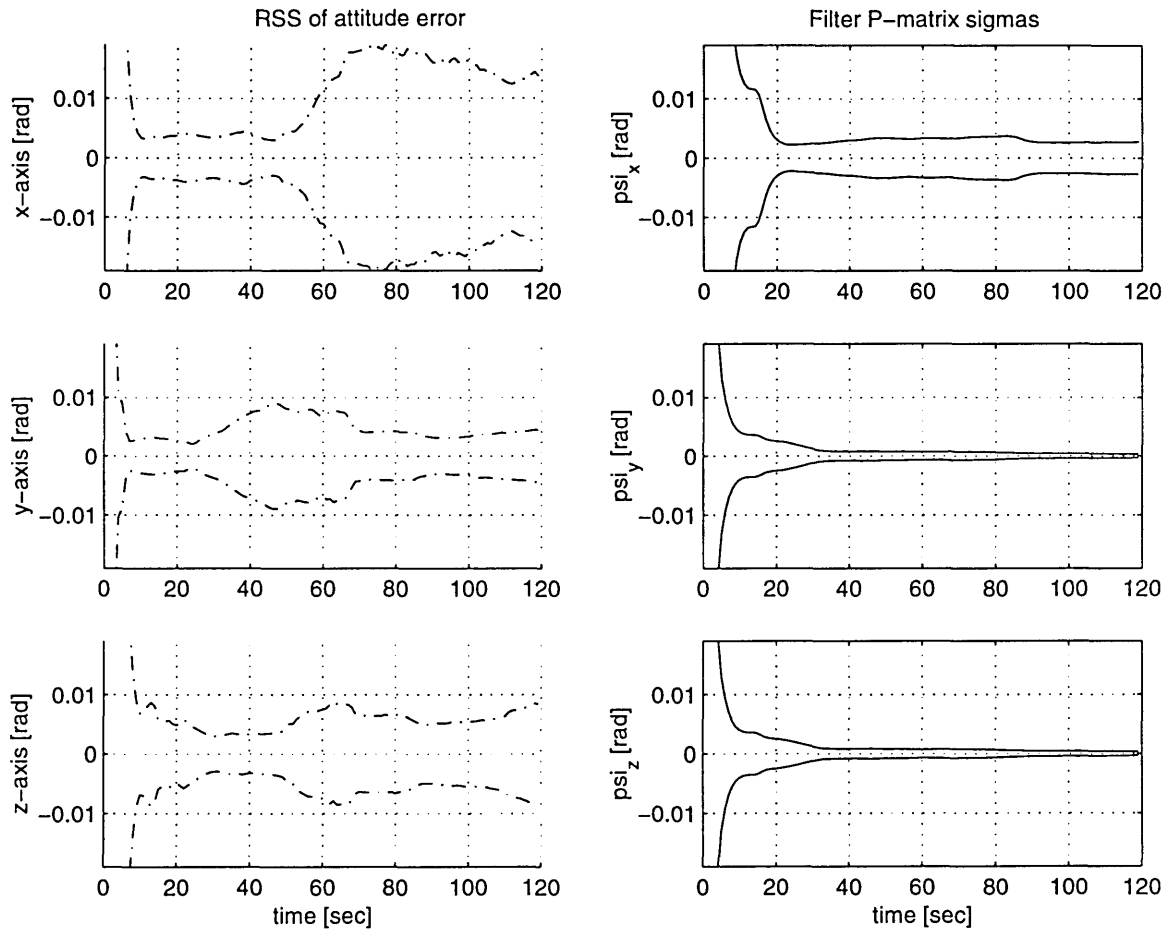
**Figure B.3:** Monte Carlo  $\sigma$  Results of Azimuth Accuracy,  $\tau = 50$



**Figure B.4:** Monte Carlo RSS Results of Azimuth Accuracy,  $\tau = 50$



**Figure B.5:** Monte Carlo  $\sigma$  Results of Azimuth Accuracy,  $\tau = 100$



**Figure B.6:** Monte Carlo RSS Results of Azimuth Accuracy,  $\tau = 100$

# Appendix C

## Acronyms

AMTT .....	Advanced Metal Tolerant Tracker
AoA.....	Angle of Attack
ASIC .....	Application Specific Integrated Circuit
CCD .....	Charged Coupled Device
$C_L$ .....	Coefficient of Lift
$C_D$ .....	Coefficient of Drag
CMATD.....	Competent Munitions Advanced Tactical Demonstrator
CPU.....	Central Processing Unit
CSDL .....	Charles Stark Draper Laboratory
DCM .....	Direction Cosine Matrix
DoF .....	Degrees of Freedom
ECEF.....	Earth Centered Earth Fixed
EGI.....	Embedded GPS/INS
EM.....	Electro Magnetic
GNC .....	Guidance Navigation & Control
GPS .....	Global Position System
HMCS .....	Head Mounted Cueing System
HMD .....	Head Mounted Display
HMIMU .....	Head Mounted Inertial Measurement Unit
HOT .....	Higher Order Terms
HTC.....	Head Tracking Computer
HUD.....	Heads Up Display
ICBM .....	Inter Continental Ballistic Missile
IHMCS.....	Inertial Head Mounted Cueing System
IMU.....	Inertial Measurement Unit
IMS .....	Inertial Measurement System
INS .....	Inertial Navigation System
JHMCS.....	Joint Helmet Mounted Cueing System
LED.....	Light Emitting Diode
LORAN.....	LONG RANGE Navigation System
LOS.....	Line of Sight
MAV.....	Micro Air Vehicle
MEMS.....	Micro ElectroMechanical System
MMISA .....	MicroMechanical Inertial Sensor Array
PPS.....	Precise Positioning Service
RAF.....	United Kingdom Royal Air Force
RF.....	Radio Frequency
RLG.....	Ring Laser Gyro
RPM .....	Revolutions Per Minute
SGI.....	Silicon Graphics Inc.
TOPART.....	TOMahawk Precision Accurate Rapid Targeting
VR.....	Virtual Reality
VSI.....	Vision Systems International





## Appendix D

### Pilot Model Code

The following code simulates a stick-figure pilot torso and head moving in a seated position.



```

/*-----*
 *      ** pilot.h **      *
 * Author: E. Bailey      *
 * Contents: + prototypes of pilot.c *
 *-----*/

/* prototypes for use in other files */
void pilot_dynamics(void);
void pilot_stick_inputs(void);
void pilot2axes(void);

```

```

/*-----*
 *      ** pilot.c **      *
 * Author: E. Bailey      *
 * Contents: + pilot torso & head *
 *           modelling      *
 *-----*/

#include "pilotmodel_ref.h"
#include "axes_ref.h"
#include "torso.h"
#include "head.h"
#include "pilot.h"

/* Copy Position and Orientation Data to Graphics Variables */
void pilot2axes(void)
{
    struct torso_ref *torso_ptr = &torso;
    struct head_ref *head_ptr = &head;
    struct axes_ref *ax = &axes;
    struct the_axes_ref *t_ax = &the_axes;

    t_ax->torso_psi = torso_ptr->twist;
    t_ax->torso_theta = torso_ptr->forw_lean;
    t_ax->torso_phi = torso_ptr->side_lean;

    t_ax->hd_rel_phi = head_ptr->lean;
    t_ax->hd_rel_theta = head_ptr->nod;
    t_ax->hd_rel_psi = head_ptr->turn;

    t_ax->head_phi = t_ax->torso_phi + t_ax->hd_rel_phi;
    t_ax->head_theta = t_ax->torso_theta + t_ax->hd_rel_theta;
    t_ax->head_psi = t_ax->torso_psi + t_ax->hd_rel_psi;
}

/* Convert Stick Movements to Head and Torso Angles */
void pilot_stick_inputs(void)
{
    torso_orientation();
    head_orientation();
}

/* Calculate Orientation and Position truth quantities */
void pilot_dynamics(void)
{
    torso_quaternion();
    torso_DCM();
    torso_pqr();

    head_quaternion();
    head_DCM();
    head_pqr();
    mckern_w();
    head_position();
    head_velocity();
    head_acceleration();
}

```

```
)
```

```
/* Function Prototypes in torso.c */  
void torso_orientation(void);  
void torso_quaternion(void);  
void torso_DCM(void);  
void torso_pqr(void);
```

```

/-----*
*      ** torso.c **      *
* Author: E. Bailey      *
* Contents: + pilot torso & head *
*           + modelling   *
*-----*/

#include <math.h>
#include "simio.h"
#include "sim_ref.h"
#include "bg_hardware_ref.h"
#include "pilotmodel_ref.h"

#define DEG2RAD 3.14159265359/180.0
#define RAD2DEG 180.0/3.14159265359

/*****
* Static Function: LaGrange 3-Point Differentiation *
* * * * *
* static double three_pt_lagrange_diff(double *var_hist, double dt) *
* * * * *
* This function implements the LaGrange 3-point Differentiation as *
* found in Kreyszig's "Advanced Engineering Mathematics", p. 790 *
* with an added twist: the three derivatives at the three previous *
* function values are averaged (summed and the result divided by 3) *
*****/

static double three_pt_lagrange_diff(double *var_hist, double dt)
{
    double LaGrange[3];
    double average;
    int i;

    /* Calculate the three z-axis rotation rates */
    LaGrange[0] = (-3.0 * var_hist[0] + 4.0 * var_hist[1] - var_hist[2])/(2.0 * dt);
    LaGrange[1] = (-var_hist[0] + var_hist[2])/(2.0 * dt);
    LaGrange[2] = (var_hist[0] - 4.0 * var_hist[1] + 3.0 * var_hist[2])/(2.0 * dt);

    /* Average the three z-axis rotation rates */
    average = 0.0;
    for(i = 0; i<3; i++)
        average += LaGrange[i]/3.0;

    return average;
}

static void pqr_quat_calc(double *q1, double *q2, double *pqr_prod)
{
    double q_skew_matrix[3][4];

    int i,j;

    q_skew_matrix[0][0] = q1[1];
    q_skew_matrix[0][1] = q1[0];
    q_skew_matrix[0][2] = -q1[3];
    q_skew_matrix[0][3] = q1[2];
    q_skew_matrix[1][0] = q1[2];
    q_skew_matrix[1][1] = q1[3];
    q_skew_matrix[1][2] = q1[0];

```

```

    q_skew_matrix[1][3] = -q1[1];
    q_skew_matrix[2][0] = q1[3];
    q_skew_matrix[2][1] = -q1[2];
    q_skew_matrix[2][2] = q1[1];
    q_skew_matrix[2][3] = q1[0];

    for(i = 0; i<3; i++)
    {
        pqr_prod[i] = 0.0;
        for(j = 0; j<4; j++)
            pqr_prod[i] += q_skew_matrix[i][j] * q2[j];
        pqr_prod[i] = -2.0 * pqr_prod[i];
    }

    static void quat_inv(double *q, double *q_inv)
    {
        int i;

        for(i = 0; i<4; i++)
            if(i == 0)
                q_inv[i] = q[i];
            else
                q_inv[i] = -q[i];
    }

    static double euler_diff(double var, double var_delayed, double dt)
    {
        double diffed_val;

        diffed_val = (var - var_delayed)/dt;

        return diffed_val;
    }

/*****
* Function: Generate Torso Euler Angles from Analog Joystick *
* * * * *
* void torso_orientation() *
* * * * *
* This function takes analog joystick values (in 3-axes) which range *
* from -1.0 to 1.0 and scales them via a gain to a suitable range of *
* motion for a pilot torso. *
*****/

void torso_orientation(void)
{
    struct torso_ref *torso_ptr = &torso;

    /* Torso Axis control via left hand controller */
    torso_ptr->twist = Filter.l_tws_avg * 50.0;

    if (Filter.l_lon_avg > 0)
        torso_ptr->forw_lean = Filter.l_lon_avg * -90.0;
    else
        torso_ptr->forw_lean = 0.0;

```

```

torso_ptr->side_lean = Filter.l_lat_avg * 45.0;
)

/*****
 * Function: Generate Torso Quaternion relative to Aircraft Body
 *
 * void torso_quaternion(void)
 *
 * This function generates the torso quaternion relative to the AC
 * body, given the Euler Angles of the torso are already calculated
 * for the given pass. The equations used are from Stevens and Lewis,
 * "Aircraft Simulation and Control", p. 41. This function also
 * ensures that the quaternion 2-norm = 1.0.
 *****/

void torso_quaternion(void)
{
    struct torso_ref *torso_ptr = &torso;
    struct sim_ref   *sim_dir = &sim;

    int i,j;
    double quat_norm = 1.0;

    for(i = 0; i<4; i++)
        torso_ptr->q_t_b_delay[i] = torso_ptr->q_t_b[i];

    torso_ptr->q_t_b[0] = (cos(torso_ptr->side_lean * DEG2RAD /2.0) *
        cos(torso_ptr->forw_lean * DEG2RAD/2.0) *
        cos(torso_ptr->twist * DEG2RAD/2.0) +
        sin(torso_ptr->side_lean * DEG2RAD/2.0) *
        sin(torso_ptr->forw_lean * DEG2RAD/2.0) *
        sin(torso_ptr->twist * DEG2RAD/2.0));
    torso_ptr->q_t_b[1] = (sin(torso_ptr->side_lean * DEG2RAD/2.0) *
        cos(torso_ptr->forw_lean * DEG2RAD/2.0) *
        cos(torso_ptr->twist * DEG2RAD/2.0) -
        cos(torso_ptr->side_lean * DEG2RAD/2.0) *
        sin(torso_ptr->forw_lean * DEG2RAD/2.0) *
        sin(torso_ptr->twist * DEG2RAD/2.0));
    torso_ptr->q_t_b[2] = (cos(torso_ptr->side_lean * DEG2RAD/2.0) *
        sin(torso_ptr->forw_lean * DEG2RAD/2.0) *
        cos(torso_ptr->twist * DEG2RAD/2.0) +
        sin(torso_ptr->side_lean * DEG2RAD/2.0) *
        cos(torso_ptr->forw_lean * DEG2RAD/2.0) *
        sin(torso_ptr->twist * DEG2RAD/2.0));
    torso_ptr->q_t_b[3] = (cos(torso_ptr->side_lean * DEG2RAD/2.0) *
        cos(torso_ptr->forw_lean * DEG2RAD/2.0) *
        sin(torso_ptr->twist * DEG2RAD/2.0) -
        sin(torso_ptr->side_lean * DEG2RAD/2.0) *
        sin(torso_ptr->forw_lean * DEG2RAD/2.0) *
        cos(torso_ptr->twist * DEG2RAD/2.0));

    /* Ensure that the 2-norm of the quaternion is 1.0 */
    quat_norm = sqrt(torso_ptr->q_t_b[0] * torso_ptr->q_t_b[0] +
        torso_ptr->q_t_b[1] * torso_ptr->q_t_b[1] +
        torso_ptr->q_t_b[2] * torso_ptr->q_t_b[2] +
        torso_ptr->q_t_b[3] * torso_ptr->q_t_b[3]);

    if (quat_norm != 0.0) /* avoid floating point exception */
        for (i = 0; i < 4; i++)

```

```

torso_ptr->q_t_b[i] = torso_ptr->q_t_b[i] / quat_norm;
)

/*****
 * Function: Generate Torso/Body Direction Cosine Matrices
 *
 * void torso_DCM(void)
 *
 * This function generates the Torso->Body DCM (c_t_b) from the
 * equations found in Stevens and Lewis, "Aircraft Simulation and
 * Control", p. 41. It then transposes that matrix to form its
 * inverse (a property of DCMs) which translates from Body->Torso
 * (c_b_t).
 *****/

void torso_DCM(void)
{
    struct torso_ref *torso_ptr = &torso;
    struct sim_ref   *sim_dir = &sim;

    int i,j;
    double q_t_b_avg[4];
    double q_mag;

    for(i = 0; i<4; i++)
        q_t_b_avg[i] = (torso_ptr->q_t_b[i] + torso_ptr->q_t_b_delay[i])*0.5;

    q_mag = sqrt(q_t_b_avg[0]*q_t_b_avg[0] + q_t_b_avg[1]*q_t_b_avg[1] +
        q_t_b_avg[2]*q_t_b_avg[2] + q_t_b_avg[3]*q_t_b_avg[3]);

    if (q_mag != 0.0)
        for(i = 0; i<4; i++)
            q_t_b_avg[i] = q_t_b_avg[i] / q_mag;

    torso_ptr->c_t_b[0][0] = q_t_b_avg[0] * q_t_b_avg[0] +
        q_t_b_avg[1] * q_t_b_avg[1] -
        q_t_b_avg[2] * q_t_b_avg[2] -
        q_t_b_avg[3] * q_t_b_avg[3];

    torso_ptr->c_t_b[0][1] = 2.0 * (q_t_b_avg[1] * q_t_b_avg[2] +
        q_t_b_avg[0] * q_t_b_avg[3]);

    torso_ptr->c_t_b[0][2] = 2.0 * (q_t_b_avg[1] * q_t_b_avg[3] -
        q_t_b_avg[0] * q_t_b_avg[2]);

    torso_ptr->c_t_b[1][0] = 2.0 * (q_t_b_avg[1] * q_t_b_avg[2] -
        q_t_b_avg[0] * q_t_b_avg[3]);

    torso_ptr->c_t_b[1][1] = q_t_b_avg[0] * q_t_b_avg[0] -
        q_t_b_avg[1] * q_t_b_avg[1] +
        q_t_b_avg[2] * q_t_b_avg[2] -
        q_t_b_avg[3] * q_t_b_avg[3];

    torso_ptr->c_t_b[1][2] = 2.0 * (q_t_b_avg[2] * q_t_b_avg[3] +
        q_t_b_avg[0] * q_t_b_avg[1]);

    torso_ptr->c_t_b[2][0] = 2.0 * (q_t_b_avg[1] * q_t_b_avg[3] +
        q_t_b_avg[0] * q_t_b_avg[2]);

```

```

torso_ptr->c_t_b[2][1] = 2.0 * (q_t_b_avg[2] * q_t_b_avg[3] -
q_t_b_avg[0] * q_t_b_avg[1]);

torso_ptr->c_t_b[2][2] = q_t_b_avg[0] * q_t_b_avg[0] -
q_t_b_avg[1] * q_t_b_avg[1] -
q_t_b_avg[2] * q_t_b_avg[2] +
q_t_b_avg[3] * q_t_b_avg[3];

/* Calculate the DCM's Inverse/Transpose */

for(i=0; i<3; i++)
  for(j=0; j<3; j++)
    torso_ptr->c_b_t_delay[i][j] = torso_ptr->c_b_t[i][j];

for (i = 0; i < 3; i++)
  for(j = 0; j < 3; j++)
    torso_ptr->c_b_t[i][j] = torso_ptr->c_t_b[j][i];
)

/*****
* Function: Calculates -the torso rotation rate vector (P,Q,R) *
*                   -the torso rotation acceleration vector *
*                   -OMEGA and OMEGAdot skew-symmetric matrices *
* void torso_pqr(void) *
* *
* This function calculates the rotation rate by differentiating the *
* quaternion, then multiplying that derivative times the current *
* quaternion's inverse and a factor of 2. The last three elements *
* of the resulting 4-element vector are P, Q, and R, the rotation *
* rates about the torso x, y, and z axes, respectively, relative to *
* the body frame, expressed in the torso frame. *
* *
* The torso PQR is then differentiated using the LaGrange 3-point *
* method to get Pdot, Qdot, and Rdot. *
* (P, Q, R) and (Pdot, Qdot, and Rdot) are then used to form the two *
* skew-symmetric cross-product matrices, OMEGA and OMEGAdot, *
* respectively. *
*****/

void torso_pqr(void)
{
  struct torso_ref *torso_ptr = &torso;
  struct sim_ref *sim_dir = &sim;

  double q_inv[4];
  double q_mag;
  double q_avg[4];
  double pqr[3];

  int i,j;

  for(i = 0; i < 4; i++){
    torso_ptr->q_dot[i] = euler_diff(torso_ptr->q_t_b[i], torso_ptr-
>q_t_b_delay[i], sim_dir->dt);

    q_avg[i] = (torso_ptr->q_t_b[i] + torso_ptr->q_t_b_delay[i])/2.0;
  }

```

```

  q_mag = sqrt(q_avg[0] * q_avg[0] + q_avg[1] * q_avg[1] + q_avg[2] * q_avg[2] +
q_avg[3] * q_avg[3]);

  if(q_mag != 0.0)
    for(i = 0; i < 4; i++)
      q_avg[i] = q_avg[i] / q_mag;

  quat_inv(q_avg, q_inv);

  pqr_quat_calc(torso_ptr->q_dot, q_inv, pqr);

  torso_ptr->p_torso = pqr[0];
  torso_ptr->q_torso = pqr[1];
  torso_ptr->r_torso = pqr[2];

  torso_ptr->w_bt_t[0] = torso_ptr->p_torso;
  torso_ptr->w_bt_t[1] = torso_ptr->q_torso;
  torso_ptr->w_bt_t[2] = torso_ptr->r_torso;

  /* set delay variables for this differentiation pass */
  for( i = 0; i < 3; i++)
  {
    if(i < 2)
    {
      torso_ptr->p_torso_delay[i] = torso_ptr->p_torso_delay[i + 1];
      torso_ptr->q_torso_delay[i] = torso_ptr->q_torso_delay[i + 1];
      torso_ptr->r_torso_delay[i] = torso_ptr->r_torso_delay[i + 1];
    }

    else if (i == 2)
    {
      torso_ptr->p_torso_delay[i] = torso_ptr->p_torso;
      torso_ptr->q_torso_delay[i] = torso_ptr->q_torso;
      torso_ptr->r_torso_delay[i] = torso_ptr->r_torso;
    }
  }

  /* calculate derivatives using Lagrange 3-point method, averaging the three
derivs.*/
  torso_ptr->pdot_torso = three_pt_lagrange_diff(torso_ptr->p_torso_delay,
sim_dir->dt);
  torso_ptr->qdot_torso = three_pt_lagrange_diff(torso_ptr->q_torso_delay,
sim_dir->dt);
  torso_ptr->rdot_torso = three_pt_lagrange_diff(torso_ptr->r_torso_delay,
sim_dir->dt);

  torso_ptr->wd_bt_t[0] = torso_ptr->pdot_torso;
  torso_ptr->wd_bt_t[1] = torso_ptr->qdot_torso;
  torso_ptr->wd_bt_t[2] = torso_ptr->rdot_torso;

  /* Form Skew-Symmetric Rotation Matrices */
  torso_ptr->OMEGA_bt_t[0][1] = -torso_ptr->r_torso ;
  torso_ptr->OMEGA_bt_t[0][2] = torso_ptr->q_torso;
  torso_ptr->OMEGA_bt_t[1][2] = -torso_ptr->p_torso;

  for(i = 0; i<3; i++)
    for(j = 0; j<3; j++)
    {
      if(i == j)
        torso_ptr->OMEGA_bt_t[i][j] = 0.0;
      else if(i > j)

```



```
torso_ptr->OMEGA_bt_t[i][j] = -torso_ptr->OMEGA_bt_t[j][i];
)

torso_ptr->OMEGAdot_bt_t[0][1] = -torso_ptr->rdot_torso;
torso_ptr->OMEGAdot_bt_t[0][2] = torso_ptr->qdot_torso;
torso_ptr->OMEGAdot_bt_t[1][2] = -torso_ptr->pdot_torso;

for(i = 0; i<3; i++)
  for(j = 0; j<3; j++)
  {
if(i == j)
  torso_ptr->OMEGAdot_bt_t[i][j] = 0.0;
else if(i > j)
  torso_ptr->OMEGAdot_bt_t[i][j] = -torso_ptr->OMEGAdot_bt_t[j][i];
  }
}
```

```
/* Function Prototypes in head.c */
```

```
void head_orientation(void);
void head_quaternion(void);
void head_DCM(void);
void head_pqr(void);
void head_w_xform_h2b(void);
void mckern_w(void);
void head_position(void);
void head_velocity(void);
void head_acceleration(void);
```

```

/-----*
*      ** head.c **      *
* Author: E. Bailey      *
* Contents: + pilot torso & head *
*           modelling     *
*-----*/

#include <math.h>
#include "simio.h"
#include "sim_ref.h"
#include "quaternion_algebra.h"
#include "v_state_ref.h"
#include "bg_hardware_ref.h"
#include "pilotmodel_ref.h"

#define DEG2RAD 3.14159265359/180.0
#define RAD2DEG 180.0/3.14159265359
#define EARTH_RATE 0.0000727220521663

/*****
* Static Function: LaGrange 3-Point Differentiation *
* *
* static double three_pt_lagrange_diff(double *var_hist, double dt) *
* *
* This function implements the LaGrange 3-point Differentiation as *
* found in Kreyszig's "Advanced Engineering Mathematics", p. 790 *
* with an added twist: the three derivatives at the three previous *
* function values are averaged (summed and the result divided by 3) *
*****/

static double three_pt_lagrange_diff(double *var_hist, double dt)
{
    double LaGrange[3];
    double average;
    int i;

    /* Calculate the three z-axis rotation rates */
    LaGrange[0] = (-3.0 * var_hist[0] + 4.0 * var_hist[1] - var_hist[2]) / (2.0 * dt);
    LaGrange[1] = (-var_hist[0] + var_hist[2]) / (2.0 * dt);
    LaGrange[2] = (var_hist[0] - 4.0 * var_hist[1] + 3.0 * var_hist[2]) / (2.0 * dt);

    /* Average the three z-axis rotation rates */
    average = 0.0;
    for(i = 0; i < 3; i++)
        average += LaGrange[i] / 3.0;

    return average;
}

static void pqr_quat_calc(double *q1, double *q2, double *pqr_prod)
{
    double q_skew_matrix[3][4];

    int i, j;

    q_skew_matrix[0][0] = q1[1];
    q_skew_matrix[0][1] = q1[0];
    q_skew_matrix[0][2] = -q1[3];
    q_skew_matrix[0][3] = q1[2];

```

```

    q_skew_matrix[1][0] = q1[2];
    q_skew_matrix[1][1] = q1[3];
    q_skew_matrix[1][2] = q1[0];
    q_skew_matrix[1][3] = -q1[1];
    q_skew_matrix[2][0] = q1[3];
    q_skew_matrix[2][1] = -q1[2];
    q_skew_matrix[2][2] = q1[1];
    q_skew_matrix[2][3] = q1[0];

    for(i = 0; i < 3; i++)
    {
        pqr_prod[i] = 0.0;
        for(j = 0; j < 4; j++)
            pqr_prod[i] += q_skew_matrix[i][j] * q2[j];
        pqr_prod[i] = -2.0 * asin(pqr_prod[i] * sim.dt) / sim.dt;
    }
}

```

```

static void quat_inv(double *q, double *q_inv)
{
    int i;

    for(i = 0; i < 4; i++)
        if(i == 0)
            q_inv[i] = q[i];
        else
            q_inv[i] = -q[i];
}

```

```

static double euler_diff(double var, double var_delayed, double dt)
{
    double diffed_val;

    diffed_val = (var - var_delayed) / dt;

    return diffed_val;
}

```

```

/*****
* Function: Generate Head Euler Angles from Analog Joystick *
* *
* void head_orientation() *
* *
* This function takes analog joystick values (in 3-axes) which range *
* from -1.0 to 1.0 and scales them via a gain to a suitable range of *
* motion for a pilot head. *
*****/

void head_orientation(void)
{
    struct head_ref *head_ptr = &head;

    /* Head Axis control via right hand controller */

    head_ptr->lean = Filter.r_lat_avg * 40.0;

    if (Filter.r_lon_avg > 0) {

```

```

    head_ptr->nod = Filter.r_lon_avg * -45.0;
}
else if (Filter.r_lon_avg < 0) {
    head_ptr->nod = (Filter.r_lon_avg) * -80.0;
}
else {
    head_ptr->nod = 0.0;
}

head_ptr->turn = Filter.r_tws_avg * 85.0;
}

/*****
 * Function: Generate Head Quaternion relative to Torso & AC Body
 *
 * void head_quaternion(void)
 *
 * This function generates the head quaternion relative to both the
 * torso and the AC body, given the Euler Angles of the torso are
 * already calculated for the given pass. The equations used are
 * from Stevens and Lewis, "Aircraft Simulation and Control", p. 41.
 * This function also ensures that the quaternions' 2-norm = 1.0.
 *****/

void head_quaternion(void)
{
    struct head_ref *head_ptr = &head;
    struct torso_ref *torso_ptr = &torso;
    struct sim_ref *sim_dir = &sim;
    struct cmat_ref *dcms = &cmat;

    int i,j;
    double quat_norm = 1.0;
    double temp_q[4];

    for(i = 0; i<4; i++)
        head_ptr->q_h_t_delay[i] = head_ptr->q_h_t[i];

    head_ptr->q_h_t[0] = (cos(head_ptr->lean * DEG2RAD /2.0) *
        cos(head_ptr->nod * DEG2RAD/2.0) *
        cos(head_ptr->turn * DEG2RAD/2.0) +
        sin(head_ptr->lean * DEG2RAD/2.0) *
        sin(head_ptr->nod * DEG2RAD/2.0) *
        sin(head_ptr->turn * DEG2RAD/2.0));
    head_ptr->q_h_t[1] = (sin(head_ptr->lean * DEG2RAD/2.0) *
        cos(head_ptr->nod * DEG2RAD/2.0) *
        cos(head_ptr->turn * DEG2RAD/2.0) -
        cos(head_ptr->lean * DEG2RAD/2.0) *
        sin(head_ptr->nod * DEG2RAD/2.0) *
        sin(head_ptr->turn * DEG2RAD/2.0));
    head_ptr->q_h_t[2] = (cos(head_ptr->lean * DEG2RAD/2.0) *
        sin(head_ptr->nod * DEG2RAD/2.0) *
        cos(head_ptr->turn * DEG2RAD/2.0) +
        sin(head_ptr->lean * DEG2RAD/2.0) *
        cos(head_ptr->nod * DEG2RAD/2.0) *
        sin(head_ptr->turn * DEG2RAD/2.0));
    head_ptr->q_h_t[3] = (cos(head_ptr->lean * DEG2RAD/2.0) *
        cos(head_ptr->nod * DEG2RAD/2.0) *
        sin(head_ptr->turn * DEG2RAD/2.0) -

```

```

    sin(head_ptr->lean * DEG2RAD/2.0) *
    sin(head_ptr->nod * DEG2RAD/2.0) *
    cos(head_ptr->turn * DEG2RAD/2.0));

/* re-normalize quaternion */

quat_norm = sqrt(head_ptr->q_h_t[0] * head_ptr->q_h_t[0] +
    head_ptr->q_h_t[1] * head_ptr->q_h_t[1] +
    head_ptr->q_h_t[2] * head_ptr->q_h_t[2] +
    head_ptr->q_h_t[3] * head_ptr->q_h_t[3]);

if (quat_norm != 1.0 && quat_norm != 0.0) /* avoid floating point exception */
    for (i = 0; i < 4; i++)
        head_ptr->q_h_t[i] = head_ptr->q_h_t[i] / quat_norm;

for(i = 0; i<4; i++)
    head_ptr->q_h_e_delay[i] = head_ptr->q_h_e[i];

quat_mult(temp_q, torso_ptr->q_t_b, dcms->q_b_e);
quat_mult(head_ptr->q_h_e, head_ptr->q_h_t, temp_q);
}

/* Calc Head DCM components from Quaternions */

void head_DCM(void)
{
    struct head_ref *head_ptr = &head;
    struct torso_ref *torso_ptr = &torso;
    struct cmat_ref *T38dcms = &cmat;
    struct sim_ref *sim_dir = &sim;

    double *temp_ptr;

    double q_h_t_avg[4];
    double q_mag;

    int i,j,k;

    for(i = 0; i<4; i++)
        q_h_t_avg[i] = (head_ptr->q_h_t[i] + head_ptr->q_h_t_delay[i])*0.5;

    q_mag = sqrt(q_h_t_avg[0]*q_h_t_avg[0] + q_h_t_avg[1]*q_h_t_avg[1] +
        q_h_t_avg[2]*q_h_t_avg[2] + q_h_t_avg[3]*q_h_t_avg[3]);

    if (q_mag != 0.0)
        for(i = 0; i<4; i++)
            q_h_t_avg[i] = q_h_t_avg[i] / q_mag;

/*****
 * Head to Torso DCM
 *****/

head_ptr->c_h_t[0][0] = q_h_t_avg[0] * q_h_t_avg[0] +
    q_h_t_avg[1] * q_h_t_avg[1] -
    q_h_t_avg[2] * q_h_t_avg[2] -
    q_h_t_avg[3] * q_h_t_avg[3];

```

```

head_ptr->c_h_t[0][1] = 2.0 * (q_h_t_avg[1] * q_h_t_avg[2] +
q_h_t_avg[0] * q_h_t_avg[3]);

head_ptr->c_h_t[0][2] = 2.0 * (q_h_t_avg[1] * q_h_t_avg[3] -
q_h_t_avg[0] * q_h_t_avg[2]);

head_ptr->c_h_t[1][0] = 2.0 * (q_h_t_avg[1] * q_h_t_avg[2] -
q_h_t_avg[0] * q_h_t_avg[3]);

head_ptr->c_h_t[1][1] = q_h_t_avg[0] * q_h_t_avg[0] -
q_h_t_avg[1] * q_h_t_avg[1] +
q_h_t_avg[2] * q_h_t_avg[2] -
q_h_t_avg[3] * q_h_t_avg[3];

head_ptr->c_h_t[1][2] = 2.0 * (q_h_t_avg[2] * q_h_t_avg[3] +
q_h_t_avg[0] * q_h_t_avg[1]);

head_ptr->c_h_t[2][0] = 2.0 * (q_h_t_avg[1] * q_h_t_avg[3] +
q_h_t_avg[0] * q_h_t_avg[2]);

head_ptr->c_h_t[2][1] = 2.0 * (q_h_t_avg[2] * q_h_t_avg[3] -
q_h_t_avg[0] * q_h_t_avg[1]);

head_ptr->c_h_t[2][2] = q_h_t_avg[0] * q_h_t_avg[0] -
q_h_t_avg[1] * q_h_t_avg[1] -
q_h_t_avg[2] * q_h_t_avg[2] +
q_h_t_avg[3] * q_h_t_avg[3];

/* Calculate the DCM's Inverse/Transpose */

for (i = 0; i < 3; i++)
for(j = 0; j < 3; j++)
head_ptr->c_t_h[i][j] = head_ptr->c_h_t[j][i];

/*****
* Head to Body DCM *
*****/

for (i = 0; i < 3; i++)
for (j = 0; j < 3; j++)
{
head_ptr->c_h_b[i][j] = 0.0;
for (k = 0; k < 3; k++)
head_ptr->c_h_b[i][j] += torso_ptr->c_t_b[i][k] * head_ptr->c_h_t[k][j];
}

/* Calculate the DCM's Inverse/Transpose */

for (i = 0; i < 3; i++)
for(j = 0; j < 3; j++)
head_ptr->c_b_h[i][j] = head_ptr->c_h_b[j][i];

/*****
* Head to ECEF *
*****/

for (i = 0; i < 3; i++)

```

```

for (j = 0; j < 3; j++)
{
head_ptr->c_h_e[i][j] = 0.0;
for (k = 0; k < 3; k++)
head_ptr->c_h_e[i][j] += T38dcm->c_b_e[i][k] * head_ptr->c_h_b[k][j];
}

for (i = 0; i < 3; i++)
for(j = 0; j < 3; j++)
head_ptr->c_e_h[i][j] = head_ptr->c_h_e[j][i];
}

/* Calculate Head PQR angular rates and OMEGA matrices */

void head_pqr(void)
{
struct head_ref *head_ptr = &head;
struct torso_ref *torso_ptr = &torso;
struct sim_ref *sim_dir = &sim;

double q_inv[4];
double q_avg[4];
double q_mag;
double q_dot[4];
double pqr[3];
double temp_vctr[3];

int i,j;

/* HEAD -> TORSO ANGLE DYNAMICS */

for(i = 0; i < 4; i++){
q_dot[i] = euler_diff(head_ptr->q_h_t[i], head_ptr->q_h_t_delay[i], sim_dir-
>dt);
q_avg[i] = (head_ptr->q_h_t[i] + head_ptr->q_h_t_delay[i])/2.0;
}

q_mag = sqrt(q_avg[0] * q_avg[0] + q_avg[1] * q_avg[1] + q_avg[2] * q_avg[2] +
q_avg[3] * q_avg[3]);

if(q_mag != 0.0)
for(i = 0; i < 4; i++)
q_avg[i] = q_avg[i] / q_mag;

quat_inv(q_avg, q_inv);

pqr_quat_calc(q_dot, q_inv, pqr);

for(i = 0; i < 3; i++)
head_ptr->w_th_h[i] = pqr[i];

/* set delay variables for this pass */
for(i = 0; i<3; i++)
for(j = 0; j<3; j++)
if (j < 2)

```

```

head_ptr->w_th_h_delay[i][j] = head_ptr->w_th_h_delay[i][j+1];
    else if (j == 2)
head_ptr->w_th_h_delay[i][j] = head_ptr->w_th_h[i];

    for(i = 0; i<3; i++)
        head_ptr->wd_th_h[i] = three_pt_lagrange_diff(head_ptr->w_th_h_delay[i],
sim_dir->dt);

    head_ptr->OMEGA_th_h[0][1] = -head_ptr->w_th_h[2];
    head_ptr->OMEGA_th_h[0][2] = head_ptr->w_th_h[1];
    head_ptr->OMEGA_th_h[1][2] = -head_ptr->w_th_h[0];

    for(i = 0; i<3; i++)
        for(j = 0; j<3; j++){
            if(i == j)
head_ptr->OMEGA_th_h[i][j] = 0.0;
            else if(i > j)
head_ptr->OMEGA_th_h[i][j] = -head_ptr->OMEGA_th_h[j][i];
        }

    /* HEAD -> BODY ANGLE DYNAMICS */

    /* Calculate total angular rate between head and body in head frame (including
torso w) */
    for(i = 0; i < 3; i++) {
        head_ptr->w_bh_h[i] = head_ptr->w_th_h[i];
        for(j = 0; j < 3; j++)
            head_ptr->w_bh_h[i] += head_ptr->c_t_h[i][j] * torso_ptr->w_bt_t[j];
    }

    /* Calculate w_th_t for calculation of wd_bh_h */

    for(i = 0; i<3; i++){
        head_ptr->w_th_t[i] = 0.0;
        for(j = 0; j<3; j++)
            head_ptr->w_th_t[i] += head_ptr->c_h_t[i][j] * head_ptr->w_th_h[j];
    }

    head_ptr->OMEGA_th_t[0][1] = -head_ptr->w_th_t[2];
    head_ptr->OMEGA_th_t[0][2] = head_ptr->w_th_t[1];
    head_ptr->OMEGA_th_t[1][2] = -head_ptr->w_th_t[0];

    for(i = 0; i<3; i++)
        for(j = 0; j<3; j++){
            if(i == j)
head_ptr->OMEGA_th_t[i][j] = 0.0;
            else if(i > j)
head_ptr->OMEGA_th_t[i][j] = -head_ptr->OMEGA_th_t[j][i];
        }

    for(i = 0; i<3; i++){
        temp_vctr[i] = 0.0;
        for(j = 0; j<3; j++)
            temp_vctr[i] += head_ptr->OMEGA_th_t[i][j] * torso_ptr->w_bt_t[j];
    }

```

```

for(i = 0; i < 3; i++) {
    head_ptr->wd_bh_h[i] = head_ptr->wd_th_h[i];
    for(j = 0; j < 3; j++)
        head_ptr->wd_bh_h[i] += head_ptr->c_t_h[i][j] * (temp_vctr[j] + torso_ptr-
>wd_bt_t[j]);
}

for(i = 0; i < 3; i++) {
    head_ptr->wd_bh_b[i] = 0.0;
    for(j = 0; j < 3; j++)
        head_ptr->wd_bh_b[i] += head_ptr->c_h_b[i][j] * head_ptr->wd_bh_h[j];
}

head_ptr->OMEGA_bh_h[0][1] = -head_ptr->w_bh_h[2] * DEG2RAD;
head_ptr->OMEGA_bh_h[0][2] = head_ptr->w_bh_h[1] * DEG2RAD;
head_ptr->OMEGA_bh_h[1][2] = -head_ptr->w_bh_h[0] * DEG2RAD;

for(i = 0; i<3; i++)
    for(j = 0; j<3; j++){
        if(i == j)
head_ptr->OMEGA_bh_h[i][j] = 0.0;
        else if(i > j)
head_ptr->OMEGA_bh_h[i][j] = -head_ptr->OMEGA_bh_h[j][i];
    }

    head_ptr->OMEGAdot_bh_b[0][1] = -head_ptr->wd_bh_h[2] * DEG2RAD;
    head_ptr->OMEGAdot_bh_b[0][2] = head_ptr->wd_bh_h[1] * DEG2RAD;
    head_ptr->OMEGAdot_bh_b[1][2] = -head_ptr->wd_bh_h[0] * DEG2RAD;

    for(i = 0; i<3; i++)
        for(j = 0; j<3; j++){
            if(i == j)
head_ptr->OMEGAdot_bh_h[i][j] = 0.0;
            else if(i > j)
head_ptr->OMEGAdot_bh_h[i][j] = -head_ptr->OMEGAdot_bh_h[j][i];
        }
}

void mckern_w(void)
{
    struct head_ref *hd = &head;

    double q_inv[4];
    double q_h_i[4];
    double q_e_i[4];
    double delta;
    double dt = sim.dt;
    double dt_inv;
    double mag;

    int i,j;

    dt_inv = 1.0/dt;

    delta = EARTH RATE * sim.t;

    q_e_i[0] = cos(delta/2);
    q_e_i[1] = 0.0;
    q_e_i[2] = 0.0;

```

```

q_e_i[3] = -sin(delta/2);
quat_mult(q_h_i, hd->q_h_e, q_e_i);

mag =
sqrt(q_h_i[0]*q_h_i[0]+q_h_i[1]*q_h_i[1]+q_h_i[2]*q_h_i[2]+q_h_i[3]*q_h_i[3]);
if(mag != 0.0)
    for(i = 0; i<4; i++)
        q_h_i[i] = q_h_i[i]/mag;

quat_mult(hd->dq, q_h_i, hd->q_h_i_hist[0]);

mag = sqrt(hd->dq[0]*hd->dq[0]+hd->dq[1]*hd->dq[1]+hd->dq[2]*hd->dq[2]+hd->dq[3]*hd->dq[3]);
if(mag != 0.0)
    for(i = 0; i<4; i++)
        hd->dq[i] = hd->dq[i]/mag;

quat_mult(hd->dqm1, q_h_i, hd->q_h_i_hist[1]);

mag = sqrt(hd->dqm1[0]*hd->dqm1[0]+hd->dqm1[1]*hd->dqm1[1]+hd->dqm1[2]*hd->dqm1[2]+hd->dqm1[3]*hd->dqm1[3]);
if(mag != 0.0)
    for(i = 0; i<4; i++)
        hd->dqm1[i] = hd->dqm1[i]/mag;

quat_mult(hd->dqm2, q_h_i, hd->q_h_i_hist[2]);

mag = sqrt(hd->dqm2[0]*hd->dqm2[0]+hd->dqm2[1]*hd->dqm2[1]+hd->dqm2[2]*hd->dqm2[2]+hd->dqm2[3]*hd->dqm2[3]);
if(mag != 0.0)
    for(i = 0; i<4; i++)
        hd->dqm2[i] = hd->dqm2[i]/mag;

for(i = 2; i >= 0; i--)
    for(j = 0; j<4; j++)
        if(i == 0)
            if(j == 0)
                hd->q_h_i_hist[i][j] = q_h_i[j];
            else
                hd->q_h_i_hist[i][j] = -q_h_i[j];
            else
                hd->q_h_i_hist[i][j] = hd->q_h_i_hist[i-1][j];

for(i = 0; i<4; i++){
    hd->wdot[i] = 4.0*((-15.0*hd->dq[i]+12.0*hd->dqm1[i]-3.0*hd->dqm2[i])/
6.0)*dt_inv*dt_inv;
    hd->west[i] = ((-18.0*hd->dq[i]+9.0*hd->dqm1[i]-2.0*hd->dqm2[i])/
6.0)*2.0*dt_inv;
    hd->wtemp[i] = (-3.0*hd->dq[i]+3.0*hd->dqm1[i]-hd->dqm2[i])/6.0;
    if(i == 0){
        hd->qwd[i] = 0.0;
        hd->qw[i] = 0.0;
    }
    else {
        hd->qwd[i] = hd->wdot[i];
        hd->qw[i] = hd->west[i];
    }
}
}

```

```

quat_mult(hd->qww, hd->qw, hd->qw);
quat_mult(hd->qwww, hd->qww, hd->qw);
quat_mult(hd->qwdw, hd->qwd, hd->qw);
quat_mult(hd->qw_wd, hd->qw, hd->qwd);

for(i = 0; i<4; i++){
    hd->wddot[i] = 12.0*(dt_inv*dt_inv*dt_inv*hd->wtemp[i] - hd->qwww[i]/48.0 -
hd->qwdw[i]/24.0 - hd->qw_wd[i]/12.0);
    if(i != 0)
        hd->w_ih_h[i-1] = hd->west[i] - dt * hd->wdot[i]/2.0 + dt * dt * hd->wddot[i]/6.0;
}
}

/*****
* Calculate Head Position in Body Frame *
*****/

void head_position(void)
{
    struct torso_ref *torso_ptr = &torso;
    struct head_ref *head_ptr = &head;

    int i,j;

    for(i=0; i<3; i++){
        head_ptr->helmetCG_x_b[i] = torso_ptr->torso_base_b[i];
        for(j=0; j<3; j++)
            head_ptr->helmetCG_x_b[i] += torso_ptr->c_t_b[i][j] * torso_ptr->torso_head_t[j];
    }
}

/*****
* Calculate Head Velocity in the Body Frame *
*****/

void head_velocity(void)
{
    struct torso_ref *torso_ptr = &torso;
    struct head_ref *head_ptr = &head;

    double temp_vctr[3] = {0.0, 0.0, 0.0};

    int i,j;

    /* In the Body Frame */

    for(i=0; i<3; i++){
        temp_vctr[i] = 0.0;
        for(j = 0; j<3; j++)
            temp_vctr[i] += -torso_ptr->OMEGA_bt_t[i][j] * torso_ptr->torso_head_t[j];
    }

    for(i = 0; i<3; i++){
        head_ptr->helmetCG_xdot_b[i] = 0.0;
        for(j = 0; j<3; j++)

```

```
    head_ptr->helmetCG_xdot_b[i] += torso_ptr->c_t_b[i][j] * temp_vctr[j];
}
}
/*****
 * Calculate Head Acceleration in the Body Frame *
 *****/
void head_acceleration(void)
{
    struct head_ref *head_ptr = &head;
    struct torso_ref *torso_ptr = &torso;

    double temp_vctr[3] = {0.0, 0.0, 0.0};
    double temp_mtrx[3][3] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};

    int i,j,k;

    for(i = 0; i<3; i++){
        for(j = 0; j < 3; j++){
            temp_mtrx[i][j] = -torso_ptr->OMEGAdot_bt_t[i][j];
            for(k = 0; k<3; k++){
                temp_mtrx[i][j] += -torso_ptr->OMEGA_bt_t[i][k] * -torso_ptr->OMEGA_bt_t[k][j];
            }

            for(i = 0; i<3; i++){
                temp_vctr[i] = 0.0;
                for(j = 0; j<3; j++)
                    temp_vctr[i] += temp_mtrx[i][j] * torso_ptr->torso_head_t[j];
            }

            for(i=0;i<3;i++){
                head_ptr->helmetCG_xddot_b[i] = 0.0;
                for(j=0; j<3; j++)
                    head_ptr->helmetCG_xddot_b[i] += torso_ptr->c_t_b[i][j] * temp_vctr[j];
            }
        }
    }
}
```

## Appendix E

### Navigation Filter C Code

The following code is the code for generating the navigation solution for both IMUs





```

/*
 * $Source: /hosts/dc2/users5/gab3/simlab/source/head_tracker/navigation/navigation.h,v $
 *
 * $Author: esb2110 $
 *
 * $Date: 2000/06/10 17:55:07 $
 *
 * $Revision: 1.3 $
 *
 * $Source: /hosts/dc2/users5/gab3/simlab/source/head_tracker/navigation/navigation.h,v $
 *
 * $Log: navigation.h,v $
 * Revision 1.3 2000/06/10 17:55:07 esb2110
 * Sim ready for MonteCarlo runs
 *
 * Revision 1.2 2000/04/11 19:43:25 esb2110
 * CVS hang-ups fixed, head navigator works with minor errors, quaternion time
  shifting corrected.
 *
 * Revision 1.1 1998/04/01 14:18:05 ejl2588
 * import from CMATD 4/1/98
 *
 */

#ifndef NAVIGATION_HDR
#define NAVIGATION_HDR

static char RCSid_navigation[] = "$Header: /hosts/dc2/users5/gab3/simlab/source/
head_tracker/navigation/navigation.h,v 1.3 2000/06/10 17:55:07 esb2110 Exp $";

/* declaration of the primary interface functions in the navigation module */
void navigation(int nav_num, int LR_flag);

#endif /* NAVIGATION_HDR */

```

```

/*****
 *      NAVIGATION.C
 *****/

/* Standard headers */
#include <math.h>

/* Headers generated from .spec files */
#include "sim_ref.h" /* global sim variables (time, init, etc.) */
#include "nav_ref.h" /* navIn, navOut, navLoc */
#include "navFC_ref.h" /* rate module variables */
#include "mmisa_ref.h" /* MMISA and other sensor variables */
#include "v_state_ref.h" /* T38 State Variables */
#include "pilotmodel_ref.h"
#include "quaternion_algebra.h" /* For Quaternion Multiply */

/* navigation code header files */
#include "WGS84.h" /* Earth Model */
#include "navigation.h" /* DataVault for data copying between modules */
#include "return_code.h"
#include "nav_highrate.h" /* High Rate routines & hr/mr, hr/lr data copying */
#include "nav_medrate.h" /* Med Rate routines & mr/hr, mr/lr data copying */
#include "nav_lorate.h" /* Low Rate routines & lr/hr, lr/mr data copying */

/* forward declarations */
static void NavProcessFC (int nav_num, int LR_flag);
static double angle_limit (double);

/* external function definitions */
extern void DoInit (int nav_num);

extern int sginap(long);

/* definitions for the simplified navigation code */
#define NUM_OF_NAV_STATES 9

/*
 * navigation -- primary interface for the navigation system from the
 * simulation
 */

void navigation (int nav_num, int LR_flag)
{
    /* pointers to relevant simulation directories */
    struct navIn_ref *nav_in = &navIn[nav_num];
    struct navLoc_ref *nav_loc = &navLoc[nav_num];
    struct navOut_ref *nav_out = &navOut[nav_num];
    struct simpLoc_ref *simp_loc = &simpLoc[nav_num];
    struct simpOut_ref *simp_out = &simpOut[nav_num];
    struct qnavOut_ref *qnav_out = &qnavOut[nav_num];
    struct nav_mrOut_ref *mrOut = &nav_mrOut[nav_num];
    struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];
    struct nav_mrParm_ref *mrParm = &nav_mrParm[nav_num];
    struct nav_hrOut_ref *hrOut = &nav_hrOut[nav_num];
    struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];
    struct nav_ctlLoc_ref *navFC_ctlLoc = &nav_ctlLoc[nav_num];

```

```

struct nav_ctlParm_ref *navFC_ctlParm = &nav_ctlParm[nav_num];
struct head_ref *hd = &head;

int ii, jj, kk; /* local counter/index */
struct cmat_ref *dcms = &cmat;
double q_b_e_star[4];
double q_mag;

static enum NavType output_mode;

/* initialization */
if (sim.init) {

    /* initialize flight nav */
    NavProcessFC (nav_num, LR_flag);
    nav_out->tSinceFire = sim.t;

}

/* normal (CP) execution */
else {

    NavProcessFC (nav_num, LR_flag);

}

/* select the appropriate module for output data, depending on mode */
if (nav_in->navSwitch == FC_NAV) {

    for (ii = 0; ii < 3; ii++) {
        nav_out->p_fb_e_e[ii] = mrOut->p_fb_e_e[ii];
        nav_out->v_fb_e_e[ii] = mrOut->v_fb_e_e[ii];
        nav_out->v_fb_e_fb[ii] = mrOut->v_fb_e_fb[ii];
        nav_out->w_fb_i_fb[ii] = mrOut->w_fb_i_fb[ii];
        nav_out->a_fb_i_fb[ii] = mrOut->a_fb_i_fb[ii];
        nav_out->wsen_fb_i_fb[ii] = hrOut->w_fb_i_fb[ii];
        nav_out->asen_fb_i_fb[ii] = hrOut->asen_fb_i_fb[ii];
    }
    for(ii=0; ii<4; ii++)
        nav_out->q_b_e[ii] = mrOut->q_pw[ii];

    nav_out->lat = mrLoc->lat;
    nav_out->lon = mrLoc->lon;
    nav_out->alt = mrLoc->alt;
    nav_out->speed = mrOut->speed;

    nav_out->tSinceFire = mrOut->time;
    nav_out->gdc_rdy = mrOut->gdc_rdy;

    /* only update the error when we transition to navigation mode */
    if (mrOut->gdc_rdy) {

        for (ii = 0; ii < 3; ii++) {
nav_loc->p_fb_e_e_err[ii] =
            mrOut->p_fb_e_e[ii] - nav_in->p_fb_e_e[ii];
nav_loc->v_fb_e_e_err[ii] =
            mrOut->v_fb_e_e[ii] - nav_in->v_fb_e_e[ii];

```

```

/* quaternion calc */
if(nav_num == 0){
    q_b_e_star[0] = dcms->q_b_e[0];
    for (ii = 1; ii < 4; ii++)
        q_b_e_star[ii] = dcms->q_b_e[ii];
}

else if(nav_num == 1){
    q_b_e_star[0] = (hd->q_h_e[0] + hd->q_h_e_delay[0]) * 0.5;
    for (ii = 1; ii < 4; ii++)
        q_b_e_star[ii] = (hd->q_h_e[ii] + hd->q_h_e_delay[ii]) * 0.5;

    q_mag = sqrt(q_b_e_star[0] * q_b_e_star[0] + q_b_e_star[1] * q_b_e_star[1] +
        q_b_e_star[2] * q_b_e_star[2] + q_b_e_star[3] * q_b_e_star[3]);

    if(q_mag != 0.0)
        for(ii = 0; ii<4; ii++)
            q_b_e_star[ii] = q_b_e_star[ii]/q_mag;
}

quat_mult(nav_loc->delta_q_fb_e_err, hrLoc->q, q_b_e_star);
}

/* if we haven't gotten to the point where the nav is activated,
don't calculate the errors (no nav output is available yet) */
else {
    nav_loc->p_fb_e_e_err[0] = 0.0;
    nav_loc->p_fb_e_e_err[1] = 0.0;
    nav_loc->p_fb_e_e_err[2] = 0.0;

    nav_loc->v_fb_e_e_err[0] = 0.0;
    nav_loc->v_fb_e_e_err[1] = 0.0;
    nav_loc->v_fb_e_e_err[2] = 0.0;

    nav_loc->w_fb_e_fb_err[0] = 0.0;
    nav_loc->w_fb_e_fb_err[1] = 0.0;
    nav_loc->w_fb_e_fb_err[2] = 0.0;

}

/* for QUICK_NAV mode, update output data only as often as med_rate would */
else if ((nav_in->navSwitch == QUICK_NAV) && (nav_loc->medRateFlag)) {

    for (ii = 0; ii < 3; ii++) {
        nav_out->p_fb_e_e[ii] = nav_in->p_fb_e_e[ii];

        nav_out->v_fb_e_e[ii] = nav_in->v_fb_e_e[ii];
        nav_out->v_fb_e_fb[ii] = 0.0;

        nav_out->w_fb_i_fb[ii] = nav_in->w_fb_i_fb[ii];
        nav_out->a_fb_i_fb[ii] = nav_in->a_fb_i_fb[ii];
    }

    for(ii=0;ii<4;ii++)
        nav_out->q_b_e[ii] = nav_in->q_pwInit[ii];

    RectangularToGeodetic (&nav_out->lat, &nav_out->lon, &nav_out->alt,

```

```

nav_out->p_fb_e_e);

nav_out->tSinceFire = mrOut->time;
nav_out->gdc_rdy = mrOut->gdc_rdy;

/* by definition, the errors are zero */
nav_loc->p_fb_e_e_err[0] = 0.0;
nav_loc->p_fb_e_e_err[1] = 0.0;
nav_loc->p_fb_e_e_err[2] = 0.0;
nav_loc->v_fb_e_e_err[0] = 0.0;
nav_loc->v_fb_e_e_err[1] = 0.0;
nav_loc->v_fb_e_e_err[2] = 0.0;
nav_loc->v_fb_e_fb_err[0] = 0.0;
nav_loc->v_fb_e_fb_err[1] = 0.0;
nav_loc->v_fb_e_fb_err[2] = 0.0;
}
}

/*
 * NavProcessFC -- function that performs the "executive" functions
 * for the flight code operational modules (nav_highrate, nav_medrate,
 * nav_lorate, plus other), including module scheduling
 */

static void NavProcessFC (int nav_num, int LR_flag)
{
/* pointers to relevant simulation directories */
struct navIn_ref *nav_in = &navIn[nav_num];
struct navLoc_ref *nav_loc = &navLoc[nav_num];
struct navOut_ref *nav_out = &navOut[nav_num];
struct simpLoc_ref *simp_loc = &simpLoc[nav_num];
struct simpOut_ref *simp_out = &simpOut[nav_num];
struct qnavOut_ref *qnav_out = &qnavOut[nav_num];
struct nav_mrOut_ref *mrOut = &nav_mrOut[nav_num];
struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];
struct nav_mrParm_ref *mrParm = &nav_mrParm[nav_num];
struct nav_hrOut_ref *hrOut = &nav_hrOut[nav_num];
struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];
struct nav_ctlLoc_ref *navFC_ctlLoc = &navFC_ctlLoc[nav_num];
struct nav_ctlParm_ref *navFC_ctlParm = &nav_ctlParm[nav_num];

/* temporary, for test */
struct nav_lrIn_ref *navFC_lrIn = &nav_lrIn;
struct cmat_ref *dcms = &cmat;
struct head_ref *hd = &head;
double q_b_e_star[4];
double q_mag;
short ii;

/* if we're in first pass of initialization mode, don't process anything */
if (sim.init && (sim.ic_j == 0))
    return;

/* for the remainder of the initialization passes */
if (sim.init && (sim.ic_j >= 1)) {

```

```

/* do the navigation code initialization */
DoInit (nav_num);
return;
}

/* get the flags from the GPS system */
if (sim.t >= navFC_ctlParm->first_gps) {
nav_loc->gpsFlagPPS = nav_in->gpsFlagPPS;
nav_loc->gpsFlagLOS = nav_in->gpsFlagLOS;
if (nav_loc->gpsFlagPPS)
    nav_loc->lr_PPS_pending = 1;
}
else {
nav_loc->gpsFlagPPS = 0;
nav_loc->gpsFlagLOS = 0;
}

/* control med rate flag */
if (navFC_ctlLoc->hi_cnt_med >= navFC_ctlParm->del_c_med) {
nav_loc->medRateFlag = TRUE;
navFC_ctlLoc->hi_cnt_med = 0;
}
else if (nav_loc->medRateFlag)
    nav_loc->medRateFlag = FALSE;

/* always process high rate loop */
HR_Reads (&mmisaOut[nav_num], nav_num);
if (nav_loc->gpsFlagPPS) {
    DoHiRate (nav_loc->medRateFlag, nav_loc->gpsFlagPPS,
              (unsigned long) (nav_in->timeGPS / FET_TO_SECONDS),
              nav_num);
}
else
    DoHiRate (nav_loc->medRateFlag, nav_loc->gpsFlagPPS, 0, nav_num);
HR_Writes (nav_num);
navFC_ctlLoc->hi_cnt++; /* advance the counters */
navFC_ctlLoc->hi_cnt_med++;

/* (if necessary) process med rate loop */
if (nav_loc->medRateFlag) {

    MR_Reads (nav_num);
    DoMedRate (nav_num);
    MR_Writes (nav_num);
    navFC_ctlLoc->med_cnt++;
    /* only update the error when we transition to navigation mode */
    if (mrOut->gdc_rdy) {

        for (ii = 0; ii < 3; ii++) {
nav_loc->p_fb_e_e_err[ii] =
mrOut->p_fb_e_e[ii] - nav_in->p_fb_e_e[ii];
nav_loc->v_fb_e_e_err[ii] =
mrOut->v_fb_e_e[ii] - nav_in->v_fb_e_e[ii];
        }

        /* quaternion calc */

        if(nav_num == 0){
q_b_e_star[0] = dcms->q_b_e[0];
for (ii = 1; ii < 4; ii++)

```

```

    q_b_e_star[ii] = dcms->q_b_e[ii];
    }
    else if(nav_num == 1){
q_b_e_star[0] = (hd->q_h_e[0] + hd->q_h_e_delay[0]) * 0.5;
for (ii = 1; ii < 4; ii++)
    q_b_e_star[ii] = (hd->q_h_e[ii] + hd->q_h_e_delay[ii]) * 0.5;

q_mag = sqrt(q_b_e_star[0] * q_b_e_star[0] + q_b_e_star[1] * q_b_e_star[1] +
    q_b_e_star[2] * q_b_e_star[2] + q_b_e_star[3] * q_b_e_star[3]);

if(q_mag != 0.0)
    for(ii = 0; ii<4; ii++)
        q_b_e_star[ii] = q_b_e_star[ii]/q_mag;
    }

    quat_mult(nav_loc->delta_q_fb_e_err, hrLoc->q, q_b_e_star);
}
/* if we haven't gotten to the point where the nav is activated,
   don't calculate the errors (no nav output is available yet) */
else {
    nav_loc->p_fb_e_e_err[0] = 0.0;
    nav_loc->p_fb_e_e_err[1] = 0.0;
    nav_loc->p_fb_e_e_err[2] = 0.0;

    nav_loc->v_fb_e_e_err[0] = 0.0;
    nav_loc->v_fb_e_e_err[1] = 0.0;
    nav_loc->v_fb_e_e_err[2] = 0.0;

    nav_loc->v_fb_e_fb_err[0] = 0.0;
    nav_loc->v_fb_e_fb_err[1] = 0.0;
    nav_loc->v_fb_e_fb_err[2] = 0.0;
    nav_loc->delta_q_fb_e_err[0] = 0.0;
    nav_loc->delta_q_fb_e_err[1] = 0.0;
    nav_loc->delta_q_fb_e_err[2] = 0.0;
    nav_loc->delta_q_fb_e_err[3] = 0.0;
}

/* (if necessary) process lo rate (1PPS) loop */
if (LR_flag && nav_loc->lr_PPS_pending) {
    LR_Reads (0, nav_num);
    DoLoRate (0);
    LR_Writes ();
    navFC_ctlLoc->lo_cnt_pps++;
    nav_loc->lr_PPS_pending = 0;
}
}

/*
   local function for limiting the range of angles
*/

static double angle_limit (double angle_in)
{
    double angle_local; /* temporary buffer for input data */

    /* copy input data to local buffer */

```

```

angle_local = angle_in;

/* perform the limiter operation */
while (angle_local > PI) {
    angle_local -= (2 * PI);
}
while (angle_local < -PI) {
    angle_local += (2 * PI);
}

/* return the result */
return (angle_local);
}

```

```

/*****
 *
 *          nav_highrate.h
 *
 * Author: Erik Bailey
 * Purpose: Defines functions called by other code
 * modules.
 *****/

#ifndef __NAV_HIGHRATE_H_
#define __NAV_HIGHRATE_H_

/* function prototypes called by other .c files */

/* called by nav_init.c */
void filter_init_hi (int nav_num);

/* called by navigation.c */
void DoHiRate (int medRateFlag, int gpsFlag, unsigned long gpsTime, int nav_num);
Return_Code_Type HR_Reads (struct mmisaOut_ref *ISAOut, int nav_num);
Return_Code_Type HR_Writes (int nav_num);

#endif /* __NAV_HIGHRATE_H_ */

```

```

/*****
 *          Nav HighRate
 *****/

/*
 * Description: This file contains the functions which comprise the
 * high-rate (nominally 100Hz) component of the precision strike
 * navigation system.
 */

#if 0
typedef int Return_Code_Type;
#define GOOD_RETURN_CODE 0
#endif

#include "simio.h"
#include "navFC_ref.h"
#include "matrixx.h"
#include "matrix_plus.h"
#include "quaternion_algebra.h"
#include "nav_ref.h"
#include "v_state_ref.h"
#include "envr_ref.h"
#include "pilotmodel_ref.h"

#include <math.h>

#include "sim_ref.h"
#include "mmisa_ref.h"
#include "return_code.h"
#include "nav_highrate.h"
#include "nav_data_vault.h"

#include "T38sens_accel_ref.h"
#include "HELMETSsens_accel_ref.h"

/* highrate functions */

static void att_alg (int nav_num);
static void dv_trans (int nav_num);
static void q_trans (int nav_num);
static void q_norm (int nav_num);
static void accum_sums (int nav_num);
static void init_sums (int nav_num);
static void filter_dstm_prp (int nav_num);
static void filter_dstm_init (int nav_num);
static void sysmat (int nav_num);
static void quat_dcm (int nav_num);
static void gyro_comp (int nav_num);
static void accel_comp (int nav_num);
static void lever_arm_comp (int nav_num);
static void hirate_reset (int nav_num);
static void StartGpsSum (int nav_num);
static void head_tracker_PHIPHI_l2 (int nav_num);
static void sensor_output (int nav_num);

```

```

/* external declarations */
extern int matrix_err_handler (int code, int src);

/*
  filter_init_hi -- all high rate initializations; flags filter
  compensation
*/

void filter_init_hi (nav_num)
{
  struct nav_hrIn_ref *hrIn = &nav_hrIn[nav_num];
  struct nav_hrOut_ref *hrOut = &nav_hrOut[nav_num];
  struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];
  struct nav_hrParm_ref *hrParm = &nav_hrParm[nav_num];
  struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];
  struct nav_ctlLoc_ref *ctlLoc = &nav_ctlLoc[nav_num];

  /* Truth Structures for Initialization Kluge */
  struct cmat_ref *dcms = &cmat;
  struct T38sensedAccelsOut_ref *T38dynOut = &T38sensedAccelsOut;
  struct head_ref *hd = &head;
  struct HELMETsensedAccelsOut_ref *HELMETdynOut = &HELMETsensedAccelsOut;

  int ii, jj; /* local counters/indices */
  int err;
  double init_q_err[4], q_inv[4];

  /* initialize the matrix entities */
  /* (note: typecast shouldn't be necessary, but the SGI compiler
  gives errors without them... */
  err = init_matrix (hrLoc->f11, N_DYN, N_DYN,
    hrLoc->f11_dp, (MATRIX_P) hrLoc->f11_d);
  if (err) {
    matrix_err_handler (err, HR_INIT);
  }
  err = init_matrix (hrLoc->f12, N_DYN, N_STAT,
    hrLoc->f12_dp, (MATRIX_P) hrLoc->f12_d);
  if (err) {
    matrix_err_handler (err, HR_INIT);
  }
  err = init_matrix (hrLoc->phil2x, N_DYN, N_STAT,
    hrLoc->phil2x_dp, (MATRIX_P) hrLoc->phil2x_d);
  if (err) {
    matrix_err_handler (err, HR_INIT);
  }
  err = init_matrix (hrLoc->phil2z, N_DYN, N_STAT,
    hrLoc->phil2z_dp, (MATRIX_P) hrLoc->phil2z_d);
  if (err) {
    matrix_err_handler (err, HR_INIT);
  }
  err = init_matrix (hrLoc->phil2z_out, N_DYN, N_STAT,
    hrLoc->phil2z_out_dp, (MATRIX_P) hrOut->phil2z);
  if (err) {
    matrix_err_handler (err, HR_INIT);
  }
}

```

```

err = zero_matrix (hrLoc->f11);
if (err) {
  matrix_err_handler (err, HR_INIT);
}
err = zero_matrix (hrLoc->f12);
if (err) {
  matrix_err_handler (err, HR_INIT);
}
err = zero_matrix (hrLoc->phil2x);
if (err) {
  matrix_err_handler (err, HR_INIT);
}
err = zero_matrix (hrLoc->phil2z);
if (err) {
  matrix_err_handler (err, HR_INIT);
}
err = zero_matrix (hrLoc->phil2z_out);
if (err) {
  matrix_err_handler (err, HR_INIT);
}

/* velocity driving position */
for (ii = 0; ii < 3; ii++) {
  hrLoc->f11->data[ii][ii+3] = ctlParm->del_t_hi;
}
/* diagonals are all 1 for the present model */
for (ii = 0; ii < hrLoc->f11->rows; ii++)
  hrLoc->f11->data[ii][ii] = 1.0;

hrLoc->f11->data[3][4] = -2.0 * hrParm->we * ctlParm->del_t_hi;
hrLoc->f11->data[4][3] = 2.0 * hrParm->we * ctlParm->del_t_hi;
hrLoc->f11->data[6][7] = -hrParm->we * ctlParm->del_t_hi;
hrLoc->f11->data[7][6] = hrParm->we * ctlParm->del_t_hi;

sysmat (nav_num); /* first call to sysmat */

filter_dstm_init (nav_num); /* initialize PHI12 */

/* initialize the output/state variables */
for (ii = 0; ii < 3; ii++) {

  /* accelerometer data */
  hrLoc->dvvc[ii] = 0.0;
  hrOut->dvvc_sum[ii] = 0.0;

  /* gyro data */
  hrLoc->dthc[ii] = 0.0;
  hrLoc->dthc_prev[ii] = 0.0;
  hrOut->dthc_sum[ii] = 0.0;
  hrLoc->dthq[ii] = 0.0;
  hrLoc->dthqp[ii] = 0.0;

  /* inertial velocity */
  hrLoc->dvi[ii] = 0.0;
  hrOut->dvi_sum[ii] = 0.0;
  hrOut->dvi_sum_left[ii] = 0.0;
  hrLoc->dvi_left[ii] = 0.0;
  hrLoc->dvi_right[ii] = 0.0;
}

```

```

/* initialize remaining local variables */
hrLoc->modeFlag = Navigate;
hrLoc->first_pass = 1;
hrLoc->restartHiRateFlag = 0;
hrLoc->filterReinit = 0;
hrLoc->mmisaInterTime = 0.0;
hrLoc->gpsDTime = 0.0;
hrLoc->temperFirstPasses = 1;
hrLoc->temp_proc_state = 0;
hrLoc->temp_blk_cnt = 0;
hrLoc->qmag_sq = 0.0;
hrLoc->qmag_inv = 0.0;

for (ii = 0; ii < 3; ii++) {
    hrLoc->dv_la[ii] = 0.0;
    hrLoc->dvv_lac[ii] = 0.0;
    hrLoc->dthc_accum[ii] = 0.0;

    for (jj = 0; jj < 3; jj++)
        hrLoc->ww[ii][jj] = 0.0;
}

if(nav_num == 0){
    hrLoc->q[0] = dcms->q_b_e[0];
    hrLoc->q_int[0] = dcms->q_b_e[0];
    for (ii = 1; ii < 4; ii++) {
        hrLoc->q[ii] = -dcms->q_b_e[ii];
        hrLoc->q_int[ii] = -dcms->q_b_e[ii];
    }
}
else if (nav_num == 1){
    q_inv[0] = hd->q_h_e[0];
    init_q_err[0] = 1.0;
    for (ii = 1; ii < 4; ii++) {
        q_inv[ii] = -hd->q_h_e[ii];
        init_q_err[ii] = hrIn->init_psi_err[ii-1]*0.5;
    }

    quat_mult (hrLoc->q,q_inv,init_q_err);

    for(ii = 0; ii<4; ii++)
        hrLoc->q_int[ii] = hrLoc->q[ii];
}

/* initialize remaining output variables */
hrOut->medValidTime = 0.0;
hrOut->loValidTime = 0.0;
hrOut->gpsInterTime = 0.0;
hrOut->gpsUpdateFlag = 0;
hrOut->update_time = 0.0;
hrOut->reset_timestamp = 0.0;

for (ii = 0; ii < 3; ii++) {
    hrOut->rwCor[ii] = 0.0;
    hrOut->vwCor[ii] = 0.0;
    if(nav_num == 0){
        hrOut->w_fb_i_fb[ii] = T38dynOut->w_ib_b[ii];
        hrOut->asen_fb_i_fb[ii] = T38dynOut->f_b_sensed[ii];
    }
}

```

```

}
else if(nav_num == 1){
    hrOut->w_fb_i_fb[ii] = HELMETdynOut->w_ib_b[ii];
    hrOut->asen_fb_i_fb[ii] = HELMETdynOut->f_h_sensed[ii];
}
}

/* most parameter values are initialized in spec file, updated by GCC */

/* just to be sure that the inputs are clean */
hrIn->qCor[0] = 1.0;
for (ii = 0; ii < 3; ii++) {
    hrIn->rwCor[ii] = 0.0;
    hrIn->vwCor[ii] = 0.0;
    hrIn->qCor[ii+1] = 0.0;
    hrIn->gbiasCor[ii] = 0.0;
    hrIn->abiasCor[ii] = 0.0;
}
hrIn->gsfe_xCor = 0.0;
hrIn->asfe2_xCor = 0.0;

nav_hrParm->we = earth.we;

/* q_trans initialization (dq is quat/w(t)w(t-dt)) */
hrParm->angl = 0.5 * ctlParm->del_t_hi * hrParm->we;
hrParm->dq0 = 1.0; /* cos angl */
hrParm->dq3 = - hrParm->angl; /* sin angl */

#if 0
/* temp_cal smoothing factor initialization */
hrParm->sm_fact = 15;
hrParm->sm_fact_inv = 1.0 / (double) hrParm->sm_fact;
#endif

return;
}

/*
DoHiRate -- executive for high rate processing module; primary entry
point for high rate processing, called from rate group control module

high rate inputs from executive:
medRateFlag  1 : current pass last one before next medium rate call
gpsFlag      1 : GPS 1PPS interrupt has occurred since
               prev hi rate pass
gpsTime      : time of GPS 1PPS (earlier than "time" is now)
               (only valid when gpsFlag is TRUE)

*/

void DoHiRate (int medRateFlag, int gpsFlag, unsigned long gpsTime, int nav_num)
{
    struct nav_hrIn_ref *hrIn = &nav_hrIn[nav_num];
    struct nav_hrOut_ref *hrOut = &nav_hrOut[nav_num];
    struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];
    struct nav_hrParm_ref *hrParm = &nav_hrParm[nav_num];
}

```



```

struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];
struct nav_ctlLoc_ref *ctlLoc = &nav_ctlLoc[nav_num];

double time_int[2];
double time_fract[2];

int ii;

/* store the input data */
hrIn->medRateFlag = medRateFlag;
hrIn->gpsFlag = gpsFlag;
hrIn->gpsTime = gpsTime;
/* Note: This data copy may seem redundant, but it's part of a
long-term strategy to move all rate group input/output data to
relevant simulation variable structures. */

/* deal with the time issues here */
hrLoc->mmisaInterTime = FET_TO_SECONDS * (double) hrIn->MMISA_time;

/* if necessary, restart the high rate filter */
if (hrLoc->restartHiRateFlag == 1) {

    /* clear the restart flag */
    hrLoc->restartHiRateFlag = 0;

    /* initialize the accumulators */
    init_sums (nav_num);

    /* clear the GPS update flag */
    hrOut->gpsUpdateFlag = 0;
}

/* navigation mode */
if (hrLoc->modeFlag == Navigate) {

    /* here is the guts of the high rate processing in nav mode */
    /* Pre-processing deletion by E. Bailey for TPD program */
    accel_comp (nav_num);      /* error compensation for accelerometers */
    gyro_comp (nav_num);      /* error compensation for gyros */
    lever_arm_comp (nav_num); /* accelerometer lever arm compensation */
    att_alg (nav_num);        /* calculate new quaternion */
    q_trans (nav_num);        /* calculate quaternion relative to earth frame */
    dv_trans (nav_num);       /* get current delta velocity to earth */
    q_norm (nav_num);         /* normalize the quaternion */
    accum_sums (nav_num);     /* accumulate accel/gyro sums (for med rate) */

    if(sim.t >= 2.0){
        time_fract[nav_num] = modf(sim.t, &time_int[nav_num]);
        if(!time_fract[nav_num])
            hrOut->gpsUpdateFlag = 1;
    }

    /* initialize or propagate phil2 matrix, as appropriate */
    if (hrLoc->filterReinit == 1) {
        filter_dstm_init (nav_num);
        hrLoc->filterReinit = 0;
    }
    else
        filter_dstm_prp (nav_num);
}

```

```

/* DLL note: consider halting phi propagation at gps interrupt */

/* if there's new GPS data... */
if (hrIn->gpsFlag) {

    /* transform the GPS interrupt time from MMISA ticks to seconds */
    hrOut->gpsInterTime = FET_TO_SECONDS * (double) hrIn->gpsTime;
    hrOut->loValidTime = hrLoc->mmisaInterTime;

    /* calculate the GPS delta velocity sum */
    StartGpsSum (nav_num);

    /* copy the phil2z data to output area */
    matrix_cpy (hrLoc->phil2z_out, hrLoc->phil2z);
    /* note: phil2z should only be copied to the output buffer,
    phil2z_out, when a 1PPS interrupt has been received, to avoid
    the possibility of the low rate module received an update of
    the phil2 data too early */

    /* set the update flag */
    hrOut->gpsUpdateFlag = 1;
}

/* if med rate is going to happen immediately after... */
if (hrIn->medRateFlag) {

    /* signal restart for next pass through DoHiRate */
    hrLoc->restartHiRateFlag = 1;
}

if (hrOut->gpsUpdateFlag) { /* measurement update now */
    hrLoc->filterReinit = 1; /* restart filter next pass */
    hirate_reset (nav_num);
}

sensor_output (nav_num);
}

/* error -- mode is neither Down_Det or Navigate */
else {
    therr("Highrate Error: The Medrate Mode is not Navigate\n");
    /* insert error handler here */
}

return;
}

/*
accel_comp -- convert dvv from counts to f/s, compensate for known
errors, convert to platform coordinates, return result as dvvc
*/

static void accel_comp (int nav_num)
{
    struct nav_hrIn_ref *hrIn = &nav_hrIn[nav_num];
}

```

```

struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];
struct nav_hrParm_ref *hrParm = &nav_hrParm[nav_num];
struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];
struct nav_ctlLoc_ref *ctlLoc = &nav_ctlLoc[nav_num];

int ii, jj;          /* local counter/index */
double comp_dvv[3]; /* compensated accelerometer data */
double dvvii;       /* scaled input data (delta velocity) */
double dvviisq;     /* square of dvvii */
double asfe_total; /* total scale factor compensation parameter */
double abias_total; /* total bias compensation parameter */

#if 0

/* cycle through three axes */
for (ii = 0; ii < 3; ii++) {

    /* do all compensation except misalignment */
    /* 1. read input, subtract offset, coarse scaling */
    /* 2. calculate total scale factor parameter */
    /* 3. calculate total bias parameter */
    switch (ii) {
    case 0:
        dvvii = hrParm->ax_scale_factor * ctlParm->del_t_hi *
(double) ((int) hrIn->accel_out[0] - hrParm->accel_offset);
        asfe_total = hrParm->asfe[0] + hrParm->sfaccel_x;
        abias_total = hrParm->abias[0] + hrParm->baccel_x;
        break;
    case 1:
        dvvii = hrParm->ay_scale_factor * ctlParm->del_t_hi *
(double) ((int) hrIn->accel_out[1] - hrParm->accel_offset);
        asfe_total = hrParm->asfe[1] + hrParm->sfaccel_y;
        abias_total = hrParm->abias[1] + hrParm->baccel_y;
        break;
    case 2:
        dvvii = hrParm->az_scale_factor * ctlParm->del_t_hi *
(double) ((int) hrIn->accel_out[2] - hrParm->accel_offset);
        asfe_total = hrParm->asfe[2] + hrParm->sfaccel_z;
        abias_total = hrParm->abias[2] + hrParm->baccel_z;
        break;
    }

    /* subtract the total bias, then calculate the square */
    dvvii -= abias_total;
    dvviisq = dvvii * dvvii;

    /* compute the compensated delta velocity */
    comp_dvv[ii] = dvvii -
        (asfe_total * dvvii) -
        (hrParm->asfe2[ii] * dvviisq) -
        (hrParm->asfe3[ii] * dvvii * dvviisq);
    }

/* apply transformation -- accelerometer frame to body (platform) frame */
for (ii = 0; ii < 3; ii++) {

    /* multiplying a matrix and a vector */
    hrLoc->dvvvc[ii] = 0.0;

```

```

        for (jj = 0; jj < 3; jj++)
            hrLoc->dvvvc[ii] += (comp_dvv[jj] * hrParm->c_a_to_p[ii][jj]);
    }
#endif
    if (nav_num == 1)
        for (ii = 0; ii < 3; ii++) {
            hrLoc->dvvvc[ii] = 0.5 * (navIn[nav_num].accel_old[ii] +
navIn[nav_num].a_fb_i_fb[ii]) * ctlParm->del_t_hi;
        }
    else
        for (ii = 0; ii < 3; ii++) {
            hrLoc->dvvvc[ii] = navIn[nav_num].a_fb_i_fb[ii] * ctlParm->del_t_hi;
        }
    return;
}

/*
gyro_comp -- gyro compensation
*/

static void gyro_comp (int nav_num)
{
    struct nav_hrIn_ref *hrIn = &nav_hrIn[nav_num];
    struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];
    struct nav_hrParm_ref *hrParm = &nav_hrParm[nav_num];
    struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];
    struct nav_ctlLoc_ref *ctlLoc = &nav_ctlLoc[nav_num];

    /* NB -- assumes accelerometer compensation has already been done */
    int ii, jj;          /* local counter/index */

#if 0
double comp_dth[3]; /* compensated gyro data */
double dthii;       /* scaled input data (delta theta) */
double dthiisq;     /* square of dthii */
double dvv_ia;      /* inline axis acceleration */
double dvv_xa1;     /* cross axis 1 acceleration */
double dvv_xa2;     /* cross axis 2 acceleration */
double cc1, cc2;    /* local computation buffers */
double gsfe_total; /* total scale factor compensation parameter */
double gbias_total; /* total bias compensation parameter */
#endif

/* cycle through three axes */
for (ii = 0; ii < 3; ii++) {

    /* push down delta theta */
    hrLoc->dthc_prev[ii] = hrLoc->dthc[ii];
}

#if 0
/* do all compensation except misalignment */
/* 1. read input, subtract offset, coarse scaling */
/* 2. obtain compensated acceleration data */
/* 3. calculate total scale factor parameter */

```

```

/* 4. calculate total bias parameter */
switch (ii) {
case 0:
    dthii = hrParm->gx_scale_factor * ctlParm->del_t_hi *
(double) ((int) hrIn->gyro_out[0] - hrParm->gyro_offset);

    dvv_ia = hrLoc->dvvvc[0];          /* del v along input axis */
    dvv_xa1 = hrLoc->dvvvc[1];
    dvv_xa2 = hrLoc->dvvvc[2];          /* del v along cross axes */

    gsfe_total = hrParm->gsfe[0] + hrParm->sfgyro_x;
    gbias_total = hrParm->gbias[0] + hrParm->bgyro_x;
    break;
case 1:
    dthii = hrParm->gy_scale_factor * ctlParm->del_t_hi *
(double) ((int) hrIn->gyro_out[1] - hrParm->gyro_offset);

    dvv_ia = hrLoc->dvvvc[1];          /* del v along input axis */
    dvv_xa1 = hrLoc->dvvvc[2];
    dvv_xa2 = hrLoc->dvvvc[0];          /* del v along cross axes */

    gsfe_total = hrParm->gsfe[1] + hrParm->sfgyro_y;
    gbias_total = hrParm->gbias[1] + hrParm->bgyro_y;
    break;
case 2:
    dthii = hrParm->gz_scale_factor * ctlParm->del_t_hi *
(double) ((int) hrIn->gyro_out[2] - hrParm->gyro_offset);

    dvv_ia = hrLoc->dvvvc[2];          /* del v along input axis */
    dvv_xa1 = hrLoc->dvvvc[0];
    dvv_xa2 = hrLoc->dvvvc[1];          /* del v along cross axes */

    gsfe_total = hrParm->gsfe[2] + hrParm->sfgyro_z;
    gbias_total = hrParm->gbias[2] + hrParm->bgyro_z;
    break;
}

/* subtract the total bias, then calculate the square */
dthii -= gbias_total;
dthiisq = dthii * dthii;

/* compute the compensated delta theta */
comp_dth[ii] = dthii -
    (gsfe_total * dthii) -
    (hrParm->gsfe2[ii] * dthiisq) -
    (hrParm->gsfe3[ii] * dthii * dthiisq);

/* - (hrParm->g_ia_gsens[ii] * dvv_ia) -
 * (hrParm->g_xa1_gsens[ii] * dvv_xa1) -
 * (hrParm->g_xa2_gsens[ii] * dvv_xa2) -
 * (hrParm->sfe_xg[ii] * dvv_xa2 * dthii);
*/

/* adjust the compensated delta theta */
cc1 = comp_dth[1] - (hrParm->dth_g1_z * comp_dth[0]);
cc2 = comp_dth[2] + (hrParm->dth_g2_y * comp_dth[1]) -
    (hrParm->dth_g2_x * comp_dth[2]);
comp_dth[1] = cc1;

```

```

comp_dth[2] = cc2;

/* apply transformation -- gyro frame to body (platform) frame */
for (ii = 0; ii < 3; ii++) {

    /* multiplying a matrix and a vector */
    hrLoc->dthc[ii] = 0.0;
    for (jj = 0; jj < 3; jj++)
        hrLoc->dthc[ii] += (comp_dth[jj] * hrParm->c_g_to_p[ii][jj]);
}
#endif

hrLoc->dthc[0] = navIn[nav_num].w_fb_i_fb[0] * ctlParm->del_t_hi;
hrLoc->dthc[1] = navIn[nav_num].w_fb_i_fb[1] * ctlParm->del_t_hi;
hrLoc->dthc[2] = navIn[nav_num].w_fb_i_fb[2] * ctlParm->del_t_hi;

/* integrate delta thetas */
for (ii = 0; ii < 3; ii++)
    hrLoc->dthc_accum[ii] += hrLoc->dthc[ii];

return;
}

/*
lever_arm_comp -- lever arm compensation
*/

static void lever_arm_comp (int nav_num)
{
    struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];
    struct nav_hrParm_ref *hrParm = &nav_hrParm[nav_num];
    struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];

    int ii, jj;          /* local counter/index */
    double xx;          /* local computation buffer */
    double wdt[3];      /* interpolated delta theta */
    double w_sq[3];     /* square of interpolated delta theta */

    /* cycle through three axes, calculating the rates and squares */
    for (ii = 0; ii < 3; ii++) {

        /* interpolate the delta theta (angular rate multiplied by dt) */
        wdt[ii] = xx = 0.5 * (hrLoc->dthc[ii] + hrLoc->dthc_prev[ii]);

        /* compute the square (angular rate squared multiplied by dt squared) */
        w_sq[ii] = xx * xx;
    }

    /* compute the ww matrix, part 1: compute the diagonal elements */
    for (ii = 0; ii < 3; ii++) {
        xx = w_sq[ii];
        for (jj = 0; jj < 3; jj++)
            xx -= w_sq[jj];
        hrLoc->ww[ii][ii] = xx;
    }
}

```

```

/* compute the ww matrix, part 2: compute the non-diagonal elements */
for (ii = 0; ii < 2; ii++) {
  for (jj = ii + 1; jj < 3; jj++) {
    hrLoc->ww[jj][ii] = wdt[ii] * wdt[jj];
    hrLoc->ww[ii][jj] = hrLoc->ww[jj][ii];
  }
}

/* assume accelerometer input axes are along the body frame */
for (ii = 0; ii < 3; ii++) { /* for each axis... */

  /* ...sum the row... */
  xx = 0.0;
  for (jj = 0; jj < 3; jj++)
    xx += (hrLoc->ww[ii][jj] * hrParm->rho[ii][jj]);

  /* ...compute the lever arm compensation... */
  hrLoc->dv_la[ii] = xx * ctlParm->freq_hi;
  /* equivalent to divide by ctlParm->del_t_hi */

  /* ...and apply it to the compensated delta velocity */
  hrLoc->dvv_lac[ii] = hrLoc->dvvc[ii] - hrLoc->dv_la[ii];
}

return;
}

/*
att_alg -- The attitude algorithm will use the compensated delta
thetas and the previous compensated delta thetas to calculate the
attitude of the platform with respect to inertial space. The
algorithm is the third order algorithm developed by R. McKern.
*/

static void att_alg (int nav_num)
{
  struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];

  int ii; /* local counter/index */
  double aa; /* local computation buffer */
  double dq[4]; /* delta quaternion */

  /* calculate interpolated delta theta values (compensated) */
  for (ii = 0; ii < 3; ii++)
    hrLoc->dthq[ii] = 0.5 * (hrLoc->dthc[ii] + hrLoc->dthc_prev[ii]);

  /* if first time, initialize previous delta theta with first value */
  if (hrLoc->first_pass == 1) {
    for (ii = 0; ii < 3; ii++)
      hrLoc->dthqp[ii] = hrLoc->dthq[ii];
    hrLoc->first_pass = 0;
  }

  #if 1
  /* compute the delta quaternion */
  aa = ((hrLoc->dthq[0] * hrLoc->dthq[0]) +
        (hrLoc->dthq[1] * hrLoc->dthq[1]) +

```

```

        (hrLoc->dthq[2] * hrLoc->dthq[2])) / 4;
  dq[0] = 1 - (aa / 2);
  dq[1] = (hrLoc->dthq[2] * hrLoc->dthqp[1] -
          hrLoc->dthq[1] * hrLoc->dthqp[2] -
          2 * hrLoc->dthq[0] * aa) / 24 + (hrLoc->dthq[0] / 2);
  dq[2] = (hrLoc->dthq[0] * hrLoc->dthqp[2] -
          hrLoc->dthq[2] * hrLoc->dthqp[0] -
          2 * hrLoc->dthq[1] * aa) / 24 + (hrLoc->dthq[1] / 2);
  dq[3] = (hrLoc->dthq[1] * hrLoc->dthqp[0] -
          hrLoc->dthq[0] * hrLoc->dthqp[1] -
          2 * hrLoc->dthq[2] * aa) / 24 + (hrLoc->dthq[2] / 2);
  #endif
  #if 0
  /* KLUGE FIX compute the delta quaternion */
  aa = ((hrLoc->dthq[0] * hrLoc->dthq[0]) +
        (hrLoc->dthq[1] * hrLoc->dthq[1]) +
        (hrLoc->dthq[2] * hrLoc->dthq[2])) / 4;
  dq[0] = 1 - (aa / 2);
  dq[1] = (-2 * hrLoc->dthq[0] * aa) / 24 + (hrLoc->dthq[0] / 2);
  dq[2] = (-2 * hrLoc->dthq[1] * aa) / 24 + (hrLoc->dthq[1] / 2);
  dq[3] = (-2 * hrLoc->dthq[2] * aa) / 24 + (hrLoc->dthq[2] / 2);
  #endif

  /* multiply previous quaternion by delta quaternion */
  quat_mult (hrLoc->q_int, hrLoc->q, dq);

  /* note: the result of this multiplication of the previous
  quaternion value with the delta quaternion proshowvides an angle
  between the body (platform) and the ECEF coordinates at the
  previous time step; this result is saved in a distinct array
  (q_int, for interim quaternion), and will subsequently be
  transformed in the function q_trans */

  /* save previous delta thetas (radians) */
  for (ii = 0; ii < 3; ii++)
    hrLoc->dthqp[ii] = hrLoc->dthq[ii];

  return;
}

/*
q_trans -- quaternion from attitude algorithm is updated to give
body-to current world (LL-NED?) transformation
*/

static void q_trans (int nav_num)
{
  struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];
  struct nav_hrParm_ref *hrParm = &nav_hrParm[nav_num];

  /* calculate new quaternion values */
  hrLoc->q[0] =
    (hrParm->dq0 * hrLoc->q_int[0]) - (hrParm->dq3 * hrLoc->q_int[3]);

  hrLoc->q[1] =
    (hrParm->dq0 * hrLoc->q_int[1]) - (hrParm->dq3 * hrLoc->q_int[2]);

```

```

hrLoc->q[2] =
    (hrParm->dq0 * hrLoc->q_int[2]) + (hrParm->dq3 * hrLoc->q_int[1]);

hrLoc->q[3] =
    (hrParm->dq0 * hrLoc->q_int[3]) + (hrParm->dq3 * hrLoc->q_int[0]);

return;
}

/*
 * q_norm -- normalize the quaternion
 */
static void q_norm (int nav_num)
{
    struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];

    int ii;          /* local counter/index */

    /* calculate the sum of the squares of the quaternion elements */
    hrLoc->qmag_sq = 0.0;
    for (ii = 0; ii < 4; ii++) {
        hrLoc->qmag_sq += hrLoc->q[ii] * hrLoc->q[ii];
    }

    /* calculate quaternion correction */
    hrLoc->qmag_inv = 1.5 - (0.5 * hrLoc->qmag_sq);

    /* correct the quaternion */
    for (ii = 0; ii < 4; ii++) {
        hrLoc->q[ii] = hrLoc->q[ii] * hrLoc->qmag_inv;
    }

    return;
}

/*
 dv_trans -- calculate inertial delta velocity (dvi) from
 compensated body delta velocity (dvv)
 */
static void dv_trans (int nav_num)
{
    struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];

    double tmp[3];    /* local computation buffer */

    /* calculate the temporary values */
    tmp[0] =
        (hrLoc->q[2] * hrLoc->dvv_lac[2]) - (hrLoc->q[3] * hrLoc->dvv_lac[1]);

    tmp[1] =
        (hrLoc->q[3] * hrLoc->dvv_lac[0]) - (hrLoc->q[1] * hrLoc->dvv_lac[2]);

```

```

tmp[2] =
    (hrLoc->q[1] * hrLoc->dvv_lac[1]) - (hrLoc->q[2] * hrLoc->dvv_lac[0]);

/* now calculate the transformed delta velocities */
hrLoc->dvi[0] = hrLoc->dvv_lac[0] + 2.0 * (hrLoc->q[0] * tmp[0]
    + hrLoc->q[2] * tmp[2]
    - hrLoc->q[3] * tmp[1]);

hrLoc->dvi[1] = hrLoc->dvv_lac[1] + 2.0 * (hrLoc->q[0] * tmp[1]
    + hrLoc->q[3] * tmp[0]
    - hrLoc->q[1] * tmp[2]);

hrLoc->dvi[2] = hrLoc->dvv_lac[2] + 2.0 * (hrLoc->q[0] * tmp[2]
    + hrLoc->q[1] * tmp[1]
    - hrLoc->q[2] * tmp[0]);

return;
}

/*
 accum_sums -- accumulate accel and gyro data (accumulated over
 medium rate period, initialized/re-initialized by init_sums)
 */
static void accum_sums (int nav_num)
{
    struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];
    struct nav_hrOut_ref *hrOut = &nav_hrOut[nav_num];

    int ii;    /* local counter/index */

    /* set the time of validity for accumulator data */
    hrOut->medValidTime = hrLoc->mmisaInterTime;

    for (ii = 0; ii < 3; ii++) {

        /* inertial delta velocities */
        hrOut->dvi_sum[ii] += hrLoc->dvi[ii];

        /* delta angles */
        hrOut->dthc_sum[ii] += hrLoc->dthc[ii];

        /* delta velocities */
        hrOut->dvvc_sum[ii] += hrLoc->dvvc[ii];

    }

    return;
}

/*
 init_sums -- initialize accel and gyro data accumulation registers
 */
static void init_sums (int nav_num)

```

```

(
  struct nav_hrOut_ref *hrOut = &nav_hrOut[nav_num];

  int ii; /* local counter/index */

  for (ii = 0; ii < 3; ii++) {
    hrOut->dvi_sum[ii] = 0.0;
    hrOut->dthc_sum[ii] = 0.0;
    hrOut->dvvc_sum[ii] = 0.0;
  }

  return;
)

/*
  filter_dstm_init -- get the next system model matrix
*/

static void filter_dstm_init (int nav_num)
(
  struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];

  int err,ii,jj;

  /* calculate the discrete state transition matrix */
  sysmat (nav_num);

  for(ii = 0; ii<N_DYN; ii++)
    for(jj = 0; jj< N_STAT; jj++)
      hrLoc->phil2z_d[ii][jj] = hrLoc->f12_d[ii][jj];

  #if 0
  /* copy f12 matrix to phil2z */
  err = matrix_cpy (hrLoc->phil2z, hrLoc->f12);
  if (err) {
    /* error handler here, if desired */
  }
  #endif

  return;
)

/*
  filter_dstm_prp -- advance discrete system given continuous system
  (keeps track of phil2z only)
*/

static void filter_dstm_prp (int nav_num)
(
  struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];

  int ii, jj; /* local indices */
  int err; /* local error code buffer */
  double temp_mtrx[N_DYN][N_STAT];

  /* PHIPHI code -- calculate phil2z only */

```

```

head_tracker_PHIPHI_12 (nav_num);
/* new matrix is calculated in phil2x */

/* then we reassign the data portion to phil2z */
for(ii = 0; ii<N_DYN; ii++)
  for(jj = 0; jj<N_STAT; jj++)
    temp_mtrx[ii][jj] = hrLoc->phil2z_d[ii][jj];

for(ii = 0; ii<N_DYN; ii++)
  for(jj = 0; jj<N_STAT; jj++)
    hrLoc->phil2z_d[ii][jj] = hrLoc->phil2x_d[ii][jj];

for(ii = 0; ii<N_DYN; ii++)
  for(jj = 0; jj<N_STAT; jj++)
    hrLoc->phil2x_d[ii][jj] = temp_mtrx[ii][jj];

/* phil2z always points to the current phil2 data */

/* get the next system model matrix */
sysmat (nav_num);

return;
)

/*
  sysmat -- calculate the current state transition matrix
  models for instrument states :
  index in phil2      state
  0-2                  gyro bias errors
  3-5                  accel bias errors
  6                    accel g**2 error
  note: placement in full state vector is 11 plus above index
*/

static void sysmat (int nav_num)
(
  struct nav_hrIn_ref *hrIn = &nav_hrIn[nav_num];
  struct nav_hrOut_ref *hrOut = &nav_hrOut[nav_num];
  struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];
  struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];
  struct cmat_ref *dcms = &cmat;
  struct head_ref *hd = &head;

  /* NB -- f is transition matrix, not continuous model matrix */
  int ii,jj,kk;
  double fsq_dt;

  static double temp_mtrx[3][3];
  static double diag_mtrx[3][3];
  static double force_mtrx[3][3];
  static double ang_v_mtrx[3][3];

  for(ii = 0; ii<3; ii++)
    for (jj = 0; jj < 3; jj++) {
      if(nav_num == 1)
        hrOut->c_e_fb[ii][jj] = hd->c_h_e[ii][jj];
      else if (nav_num == 0)

```

```

hrOut->c_e_fb[ii][jj] = dcms->c_b_e[ii][jj];
}

/* set up the f11 matrix */
hrLoc->f11_d[3][7] = - hrLoc->dvi[2];
hrLoc->f11_d[3][8] = hrLoc->dvi[1];
hrLoc->f11_d[4][8] = - hrLoc->dvi[0];
hrLoc->f11_d[4][6] = hrLoc->dvi[2];
hrLoc->f11_d[5][6] = - hrLoc->dvi[1];
hrLoc->f11_d[5][7] = hrLoc->dvi[0];
/* diagonals (=1) and top 3 rows stay as initialized */

/* get the latest body-to-ECEF transform matrix */
quat_dcm (nav_num);

for(ii = 0; ii<3; ii++)
    force_mtrx[ii][ii] = hrLoc->dvvc[ii];

for(ii = 0; ii<3; ii++)
    ang_v_mtrx[ii][ii] = hrLoc->dthc[ii];

/* f12 matrix */
/* gyro bias elements drive attitude error states 6-8 */
for (ii=0; ii<3; ii++)
    for (jj=0; jj<3; jj++) {
        hrLoc->f12_d[6+ii][jj] = ctlParm->del_t_hi * hrOut->c_e_fb[ii][jj];
    }

/* gyro scale factor elements drive attitude error states 6-8*/
for (ii=0; ii<3; ii++)
    for (jj=0; jj<3; jj++){
        temp_mtrx[ii][jj] = 0.0;
        for (kk = 0; kk<3; kk++) {
temp_mtrx[ii][jj] += hrOut->c_e_fb[ii][kk] * ang_v_mtrx[kk][jj];
        }
    }

for (ii=0; ii<3; ii++)
    for(jj=0; jj<3; jj++)
        hrLoc->f12_d[6+ii][3+jj] = temp_mtrx[ii][jj];

/* accelerometer bias elements drive vel error states 3-5 */
for (ii=0; ii<3; ii++) {
    for (jj=0; jj<3; jj++) {
        hrLoc->f12_d[3+ii][6+jj] = ctlParm->del_t_hi * hrOut->c_e_fb[ii][jj];
    }
}

for (ii=0; ii<3; ii++)
    for (jj=0; jj<3; jj++){
        temp_mtrx[ii][jj] = 0.0;
        for (kk = 0; kk<3; kk++) {
temp_mtrx[ii][jj] += hrOut->c_e_fb[ii][kk] * force_mtrx[kk][jj];
        }
    }

for (ii=0; ii<3; ii++)
    for(jj=0; jj<3; jj++)

```

```

        hrLoc->f12_d[3+ii][9+jj] = temp_mtrx[ii][jj];
    }
    return;
}

/*
quat_dcm -- quaternion (vector format) to direction cosine matrix
(mccconley format)
*/

static void quat_dcm (int nav_num)
{
    struct nav_hrOut_ref *hrOut = &nav_hrOut[nav_num];
    struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];

    /* NB -- assume that direction cosine matrix is 3x3 */

    /* first row */
    hrOut->c_e_fb[0][0] = ((hrLoc->q[0] * hrLoc->q[0]) +
(hrLoc->q[1] * hrLoc->q[1]) -
                        (hrLoc->q[2] * hrLoc->q[2]) -
(hrLoc->q[3] * hrLoc->q[3]));
    hrOut->c_e_fb[0][1] = 2.0 * ((hrLoc->q[1] * hrLoc->q[2]) -
(hrLoc->q[0] * hrLoc->q[3]));
    hrOut->c_e_fb[0][2] = 2.0 * ((hrLoc->q[1] * hrLoc->q[3]) +
(hrLoc->q[0] * hrLoc->q[2]));

    /* second row */
    hrOut->c_e_fb[1][0] = 2.0 * ((hrLoc->q[1] * hrLoc->q[2]) +
(hrLoc->q[0] * hrLoc->q[3]));
    hrOut->c_e_fb[1][1] = ((hrLoc->q[0] * hrLoc->q[0]) -
(hrLoc->q[1] * hrLoc->q[1]) +
                        (hrLoc->q[2] * hrLoc->q[2]) -
(hrLoc->q[3] * hrLoc->q[3]));
    hrOut->c_e_fb[1][2] = 2.0 * ((hrLoc->q[2] * hrLoc->q[3]) -
(hrLoc->q[0] * hrLoc->q[1]));

    /* third row */
    hrOut->c_e_fb[2][0] = 2.0 * ((hrLoc->q[1] * hrLoc->q[3]) -
(hrLoc->q[0] * hrLoc->q[2]));
    hrOut->c_e_fb[2][1] = 2.0 * ((hrLoc->q[2] * hrLoc->q[3]) +
(hrLoc->q[0] * hrLoc->q[1]));
    hrOut->c_e_fb[2][2] = ((hrLoc->q[0] * hrLoc->q[0]) -
(hrLoc->q[1] * hrLoc->q[1]) -
                        (hrLoc->q[2] * hrLoc->q[2]) +
(hrLoc->q[3] * hrLoc->q[3]));

    return;
}

```

```

/*
  hirate_reset -- update the quaternion and several compensation
  parameters, using data from low rate
*/
static void hirate_reset (int nav_num)
{
  struct nav_hrIn_ref *hrIn = &nav_hrIn[nav_num];
  struct nav_hrOut_ref *hrOut = &nav_hrOut[nav_num];
  struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];
  struct nav_hrParm_ref *hrParm = &nav_hrParm[nav_num];

  double q_comp[4]; /* local buffer for compensated quaternion */
  int ii; /* local counter/index */

  hrIn->qCor[0] = 1.0;
  for(ii = 0; ii < 3; ii++)
    hrIn->qCor[ii+1] = 0.0;

  /* calculate the actual compensated values, store in local buffer */
  quat_mult (q_comp, hrIn->qCor, hrLoc->q);

  /* copy compensated values into output buffer */
  for (ii = 0; ii < 4; ii++)
    hrLoc->q[ii] = q_comp[ii];

  /* update the remaining compensation parameters */
  for (ii = 0; ii < 3; ii++)
    hrParm->gbias[ii] -= hrIn->gbiasCor[ii];
  hrParm->gsfe[0] -= hrIn->gsfe_xCor;
  for (ii = 0; ii < 3; ii++)
    hrParm->abias[ii] += hrIn->abiasCor[ii];
  hrParm->asfe2[0] += hrIn->asfe2_xCor;

  /* copy the medium rate reset data for subsequent use */
  for (ii = 0; ii < 3; ii++) {
    hrOut->rwCor[ii] = hrIn->rwCor[ii];
    hrOut->vwCor[ii] = hrIn->vwCor[ii];
  }
  /* pass to med rate the time ID matching rw and vw Cor */
  hrOut->update_time = hrIn->update_time;

  /* save the timestamp of the reset data (for low rate) */
  hrOut->reset_timestamp = hrIn->update_time;

  return;
}

/*
  StartGpsSum -- calculate the GPS delta velocity sum
*/
static void StartGpsSum (int nav_num)
{

```

```

  struct nav_hrOut_ref *hrOut = &nav_hrOut[nav_num];
  struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];
  struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];

  int ii; /* local counter/index */

  /* calculate the time between the previous MMISA interrupt and
  the GPS interrupt */
  hrLoc->gpsDTime =
    hrOut->gpsInterTime - (hrLoc->mmisaInterTime - ctlParm->del_t_hi);

  /* calculate the partial delta velocities */
  for (ii = 0; ii < 3; ii++) {

    /* portion of delta velocity from last high rate to GPS interrupt */
    hrLoc->dvi_left[ii] =
      (hrLoc->gpsDTime * ctlParm->freq_hi) * hrLoc->dvi[ii];

    /* note: multiplying by frequency is equivalent to dividing by
    the time period, but saves us a division operation */

    /* portion of delta velocity from GPS interrupt to current high rate */
    hrLoc->dvi_right[ii] = hrLoc->dvi[ii] - hrLoc->dvi_left[ii];

    /* calculate delta velocity from last med rate to GPS interrupt */
    hrOut->dvi_sum_left[ii] = hrOut->dvi_sum[ii] - hrLoc->dvi_right[ii];

  }

  return;
}

/*
  head_tracker_PHIPHI_12 -- PHI x PHI code
*/
static void head_tracker_PHIPHI_12 (int nav_num)
{
  /* calculates PHI12 = f11 x PHI12 + f12 for special case only */

  /* declare pointers to the pertinent data areas */
  struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];

  int ii,jj,kk;

  for(ii = 0; ii<9; ii++)
    for(jj = 0; jj<12; jj++) {
      hrLoc->phil2x_d[ii][jj] = hrLoc->f12_d[ii][jj];
      for(kk = 0; kk<9; kk++)
        hrLoc->phil2x_d[ii][jj] += hrLoc->f11_d[ii][kk] * hrLoc->phil2z_d[kk][jj];
    }

  return;
}

```



```

/*
 * sensor_output -- copy compensated sensor data to output buffer
 * for use by guidance and control systems
 */

static void sensor_output (int nav_num)
{
    struct nav_hrOut_ref *hrOut = &nav_hrOut[nav_num];
    struct nav_hrLoc_ref *hrLoc = &nav_hrLoc[nav_num];
    struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];

    int ii; /* local counter/index */

    for (ii = 0; ii < 3; ii++) {
        hrOut->w_fb_i_fb[ii] = hrLoc->dthc[ii] * ctlParm->freq_hi;
        hrOut->asen_fb_i_fb[ii] = hrLoc->dvvc[ii] * ctlParm->freq_hi;
    }
    /* note: multiplication by frequency is functionally equivalent to
     division by time period, but computationally much more efficient */

    return;
}

/*****
 *
 * FUNCTION NAME:          HR_Reads
 *
 * DESCRIPTION:          Reads data from the common into HR 'local' variables
 *
 * ARGUMENTS:            int cycle - cycle number of HR pass that is running
 *
 * RETURNS:              GOOD_RETURN_CODE
 *
 *****/

Return_Code_Type HR_Reads (struct mmisaOut_ref *ISAOut, int nav_num)
{
    struct nav_hrIn_ref *hrIn = &nav_hrIn[nav_num];

    int ii;

    DV1r2hr 1r2hr;
    DVmr2hr mr2hr;

    Get_HR_In (&1r2hr, &mr2hr, nav_num);

    /* get MMISA data for high rate module */
    hrIn->MMISA_time = (unsigned long) (sim.t / FET_TO_SECONDS);
    for (ii = 0; ii < 3; ii++) {
        hrIn->accel_out[ii] = ISAOut->mmisaAccl[ii];
        hrIn->gyro_out[ii] = ISAOut->mmisaGyro[ii];
    }
    /* note: the accelerometer and gyro outputs (inputs to nav) should
     really be funneled through the navIn structure in nav.spec... */
}

```

```

/* data copies from local to high rate module */

/* data from medium rate module to high rate module */
for (ii = 0; ii < 4; ii++)
    hrIn->q_pw[ii] = mr2hr.q_pw[ii];

/* data from low rate module to high rate module */
hrIn->update_time = 1r2hr.update_time;
hrIn->gsfe_xCor = 1r2hr.gsfe_xCor;
hrIn->asfe2_xCor = 1r2hr.asfe2_xCor;
for (ii = 0; ii < 3; ii++) {
    hrIn->rwCor[ii] = 1r2hr.rwCor[ii];
    hrIn->vwCor[ii] = 1r2hr.vwCor[ii];
    hrIn->gbiasCor[ii] = 1r2hr.gbiasCor[ii];
    hrIn->abiasCor[ii] = 1r2hr.abiasCor[ii];
    hrIn->qCor[ii] = 1r2hr.qCor[ii];
}
hrIn->qCor[ii] = 1r2hr.qCor[ii];
/* (once extra on qCor, to get the fourth element) */

return GOOD_RETURN_CODE;
} /* end HR_Reads */

/*****
 *
 * FUNCTION NAME:          HR_Writes
 *
 * DESCRIPTION:          Writes data from the HR 'local' variables into the common area
 *
 * ARGUMENTS:            none
 *
 * RETURNS:              GOOD_RETURN_CODE
 *
 *****/

Return_Code_Type HR_Writes (int nav_num)
{
    struct nav_hrOut_ref *hrOut = &nav_hrOut[nav_num];

    int ii, jj;

    DVhr21r hr21r;
    DVhr2mr hr2mr;

    /* data copies from high rate module to local */

    /* data from high rate module to medium rate module */
    hr2mr.medValidTime = hrOut->medValidTime;
    hr2mr.gpsInterTime = hrOut->gpsInterTime;
    hr2mr.gpsUpdateFlag = hrOut->gpsUpdateFlag;
    hr2mr.update_time = hrOut->update_time;
    for (ii = 0; ii < 3; ii++) {
        hr2mr.dvvc_sum[ii] = hrOut->dvvc_sum[ii];
        hr2mr.dthc_sum[ii] = hrOut->dthc_sum[ii];
        hr2mr.dvi_sum[ii] = hrOut->dvi_sum[ii];
        hr2mr.dvi_sum_left[ii] = hrOut->dvi_sum_left[ii];
        hr2mr.rwCor[ii] = hrOut->rwCor[ii];
    }
}

```

```

hr2mr.vwCor[ii] = hrOut->vwCor[ii];
for (jj = 0; jj < 3; jj++)
    hr2mr.c_e_fb[ii][jj] = hrOut->c_e_fb[ii][jj];
}

/* data from high rate module to low rate module */
hr2lr.loValidTime = hrOut->loValidTime;
hr2lr.gpsInterTime = hrOut->gpsInterTime;
for (ii = 0; ii < N_DYN; ii++)
    for (jj = 0; jj < N_STAT; jj++)
        hr2lr.phil2z[ii][jj] = hrOut->phil2z[ii][jj];
hr2lr.last_reset_time = hrOut->reset_timestamp;

Put_HR_Out(&hr2lr, &hr2mr, nav_num);

return GOOD_RETURN_CODE;
} /* end HR_Writes */

```

```

/*****
*
*          nav_medrate.h
*
* Author: Erik Bailey
* Purpose: Defines functions called by other code
* modules.
*****/

#ifndef __NAV_MEDRATE_H_
#define __NAV_MEDRATE_H_

/* function prototypes called by other .c files */

/* called by nav_init.c */
void filter_init_med (int nav_num);

/* called by navigation.c */
void DoMedRate (int nav_hum);
Return_Code_Type MR_Reads (int nav_num);
Return_Code_Type MR_Writes (int nav_num);

#endif /* __NAV_MEDRATE_H_ */

```

```

/*****
/* NAV MEDIUM RATE */
/*****

/*
 * Description: This file contains the functions which comprise the
 * medium-rate (nominally 50Hz) component of the precision strike
 * navigation system.
 */

#define DEBUG_PERFECT_NAV 0

#include <math.h>
#include "navFC_ref.h"
#include "matrixx.h"
#include "matrix_plus.h"
#include "consts_ref.h"
#include "WGS84.h"
#include "v_state_ref.h"
#include "sim_ref.h"
#include "nav_ref.h"
#include "return_code.h"
#include "nav_medrate.h"
#include "nav_data_vault.h"
#include "T38sens_accel_ref.h"
#include "HELMETSsens_accel_ref.h"
/* forward declarations */

static void rotate_dcm (struct matrix_ref *, double, int);
static void vel_conversion (int nav_num);
static void nav_init (int nav_num);
static void grav_init (int nav_num);
static void setup_telemetry (int nav_num);
static void nav_alg (int nav_num);
static void med_accum_sums (int nav_num);
static void addNavHistory (int nav_num);
static void nav_output (int nav_num);
static void medrate_reset (int nav_num);
static void prepGpsMeasmt (int nav_num);
static void receiver_aid (int nav_num);

/* external references */
extern int matrix_err_handler (int code, int src);

#define Square(X) ((X)*(X))

/*

note: Included in the data passed from the high rate group to the
medium rate group is the body-to-ECEF transformation matrix, c_e_fb.
In the high rate group, this data is treated as a simple two
dimensional array, and is stored as such in the output section of
the high rate shared variable area. It is copied into a similar two
dimensional array in the medium rate input section. However, the
medium rate processing uses this data as a matrix, and performs

```

```

matrix operations on the data using the matrix library functions.
In order to rectify the references, the solution chosen was as
follows:

```

- a) define a 2D array named c\_e\_fb in the high rate output data area
- b) define a 2D array named c\_e\_fb\_d in the medium rate input data area
- c) define a matrix entity named c\_e\_fb in the medium rate local data area
- d) within the matrix entity named c\_e\_fb, reference the input data c\_e\_fb\_d as the associated data storage for the matrix

```

Using this design, the data is element-wise copied from the high
rate output section to the medium rate input section, and
subsequently referenced within the medium rate code as a matrix
entity.

```

```

*/

/*
 * filter_init_med -- initialization of medium rate group data
 * storage entities
 */

```

```

void filter_init_med (int nav_num)
{
    /* pointers to shared data area */
    struct nav_mrIn_ref *mrIn = &nav_mrIn[nav_num];
    struct nav_mrOut_ref *mrOut = &nav_mrOut[nav_num];
    struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];
    struct nav_mrParm_ref *mrParm = &nav_mrParm[nav_num];
    struct nav_ctlLoc_ref *ctlLoc = &nav_ctlLoc[nav_num];
    struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];

    struct cvars_ref *cv = &cvars;
    struct cmat_ref *dcms = &cmat;
    struct st_vector_ref *sv = &st_vector;
    struct T38Truth_ref *Ttruth = &T38Truth;
    struct HELMETTruth_ref *Htruth = &HELMETTruth;

    /* Local Variables*/
    int ii, jj;          /* local counter/index */
    double alt;         /* local buffer for altitude */
    double lat, lon;    /* local buffers for latitude, longitude (dummy) */
    double r_ref[3];    /* local buffer for aimpoint position */
    int err;            /* buffer for error code */

    mrLoc->rec_aid_count = 0;
    mrLoc->timeLastGps = 0.0;
    mrLoc->step_change = 0;

    /* initialize output and local state variables */
    mrLoc->restartMedRateFlag = 0;
    if(nav_num == 0)
        for (ii = 0; ii < 3; ii++) {
            mrOut->dvi_sum_med[ii] = 0.0;
        }

```

```

    mrOut->dvi_sum_sum[ii] = 0.0;
    mrLoc->dvi_s[ii] = 0.0;
    mrLoc->dvi_ss[ii] = 0.0;
    mrOut->vwGpsTime[ii] = 0.0;
    mrLoc->r0[ii] = Ttruth->R_ecef[ii];
    mrLoc->rw[ii] = 0.0;
    mrLoc->vw[ii] = Ttruth->Rdot_ecef[ii];
    mrLoc->accel_sum[ii] = 0.0;
    mrLoc->gyro_sum[ii] = 0.0;
}
else if(nav_num == 1)
for (ii = 0; ii < 3; ii++) {
    mrOut->dvi_sum_med[ii] = 0.0;
    mrOut->dvi_sum_sum[ii] = 0.0;
    mrLoc->dvi_s[ii] = 0.0;
    mrLoc->dvi_ss[ii] = 0.0;
    mrOut->vwGpsTime[ii] = 0.0;
    mrLoc->r0[ii] = Htruth->R_ecef[ii] + mrIn->init_p_err[ii];
    mrLoc->rw[ii] = 0.0;
    mrLoc->vw[ii] = Htruth->Rdot_ecef[ii] + mrIn->init_v_err[ii];
    mrLoc->accel_sum[ii] = 0.0;
    mrLoc->gyro_sum[ii] = 0.0;
}

if (mrOut->hist_len > MAX_NAV_HIST) {
    /* error handler here, if desired */
    mrOut->hist_len = MAX_NAV_HIST;
}
else if (mrOut->hist_len <= 0) {
    /* error handler here, if desired */
    mrOut->hist_len = 1;
}
mrOut->hist_last = -1;
mrOut->hist_full = 0;

/* initialize the matrix/vector entities */
err = init_vector (mrLoc->g_e, 3, (MATRIX_P) mrLoc->g_e_d);
if (err) {
    matrix_err_handler (err, MR_INIT);
}
err = init_vector (mrLoc->g_fb, 3, (MATRIX_P) mrLoc->g_fb_d);
if (err) {
    matrix_err_handler (err, MR_INIT);
}
err = init_matrix (mrLoc->c_e_ln, 3, 3,
    mrLoc->c_e_ln_dp, (MATRIX_P) mrLoc->c_e_ln_d);
if (err) {
    matrix_err_handler (err, MR_INIT);
}
err = init_matrix (mrLoc->c_ln_fb, 3, 3,
    mrLoc->c_ln_fb_dp, (MATRIX_P) mrLoc->c_ln_fb_d);
if (err) {
    matrix_err_handler (err, MR_INIT);
}
err = init_matrix (mrLoc->c_e_fb, 3, 3,
    mrLoc->c_e_fb_dp, (MATRIX_P) mrIn->c_e_fb_d);
if (err) {
    matrix_err_handler (err, MR_INIT);
}
}

```

```

/* calculate the fore-body-to-sensor rotation matrix */
for (ii = 0; ii < 3; ii++) {
    for (jj = 0; jj < 3; jj++) {
        if (ii == jj)
            mrLoc->c_s_fb[ii][jj] = 1.0;
        else
            mrLoc->c_s_fb[ii][jj] = 0.0;
    }
}

/* initialize remaining local variables */

mrLoc->restartMedRateFlag = 0;
mrLoc->timeNavMode = 0.0;
mrLoc->nwt = 0.0;
mrLoc->slt = cv->slat;
mrLoc->clt = cv->clat;
mrLoc->sln = cv->slon;
mrLoc->cln = cv->clon;
mrLoc->lat = sv->latitude;
mrLoc->lon = sv->longitude;
mrLoc->alt = -sv->z;
mrLoc->tResetPrev = 0.0;

for (ii = 0; ii < 3; ii++) {
    mrLoc->rw_prev[ii] = 0.0;
    mrLoc->vw_prev[ii] = 0.0;
    mrLoc->g0[ii] = 0.0;
    mrLoc->grv[ii] = 0.0;
    mrLoc->rrr[ii] = 0.0;
    mrLoc->dvi_s[ii] = 0.0;
    mrLoc->dvi_ss[ii] = 0.0;
    mrLoc->rwCorPrev[ii] = 0.0;
    mrLoc->vwCorPrev[ii] = 0.0;
    mrLoc->rwDeltaCor[ii] = 0.0;
    mrLoc->vwDeltaCor[ii] = 0.0;
    mrLoc->w_fb_i_s[ii] = 0.0;
    mrLoc->a_fb_i_s[ii] = 0.0;

    for (jj = 0; jj < 3; jj++)
        mrLoc->gamma0[ii][jj] = 0.0;
}

/* initialize remaining output variables */
mrOut->modeFlag = Navigate;
mrOut->gdc_rdy = 0;
mrOut->time = 0.0;
mrOut->altitude = 0.0;
mrOut->speed = 0.0;

for (ii = 0; ii < MAX_NAV_HIST; ii++) {
    mrOut->nav_history[ii]->time = 0.0;

    for (jj = 0; jj < 3; jj++) {
        mrOut->nav_history[ii]->pos[jj] = 0.0;
        mrOut->nav_history[ii]->vel[jj] = 0.0;
    }
}
}

```

```

if(nav_num == 0){
    for (ii = 0; ii < 3; ii++) {
        mrOut->pos0[ii] = 0.0;
        mrOut->p_fb_e_e[ii] = Ttruth->R_ecef[ii];
        mrOut->v_fb_e_e[ii] = Ttruth->Rdot_ecef[ii];
        mrOut->v_fb_e_fb[ii] = 0.0;
        mrOut->e_fb_ln[ii] = 0.0;
        mrOut->w_fb_i_fb[ii] = 0.0;
        mrOut->a_fb_i_fb[ii] = 0.0;
        mrOut->g_fb[ii] = 0.0;

        for (jj = 0; jj < 3; jj++) {
            mrOut->c_fb_e[ii][jj] = 0.0;
            mrOut->c_fb_ln[ii][jj] = 0.0;
        }
    }
} else if(nav_num == 1){
    for (ii = 0; ii < 3; ii++) {
        mrOut->pos0[ii] = 0.0;
        mrOut->p_fb_e_e[ii] = Htruth->R_ecef[ii];
        mrOut->v_fb_e_e[ii] = Htruth->Rdot_ecef[ii];
        mrOut->v_fb_e_fb[ii] = 0.0;
        mrOut->e_fb_ln[ii] = 0.0;
        mrOut->w_fb_i_fb[ii] = 0.0;
        mrOut->a_fb_i_fb[ii] = 0.0;
        mrOut->g_fb[ii] = 0.0;

        for (jj = 0; jj < 3; jj++) {
            mrOut->c_fb_e[ii][jj] = 0.0;
            mrOut->c_fb_ln[ii][jj] = 0.0;
        }
    }
}

for (ii = 0; ii < 4; ii++)
    mrOut->q_pw[ii] = dcms->q_b_e[ii];

return;
}

/*
 * rotate_dcm -- apply a rotation to existing direction cosine matrix
 */

static void rotate_dcm (struct matrix_ref *dcm, double angle, int axis)
{
    /* axis code: 1 = x-axis, 2 = y-axis, 3 = z-axis (Euler convention) */

    double in_data[3][3]; /* buffer for input data */
    double xform[3][3]; /* buffer for transformation matrix */
    int ii, jj, kk; /* local counter/index */

```

```

    /* copy data to input array */
    for (ii = 0; ii < 3; ii++) {
        for (jj = 0; jj < 3; jj++) {
            in_data[ii][jj] = dcm->data[ii][jj];
        }
    }

    /* populate transformation matrix */
    switch (axis) {
    case 1: /* x-axis */
        xform[0][0] = 1.0;
        xform[0][1] = 0.0;
        xform[0][2] = 0.0;
        xform[1][0] = 0.0;
        xform[1][1] = cos (angle);
        xform[1][2] = sin (angle);
        xform[2][0] = 0.0;
        xform[2][1] = - sin (angle);
        xform[2][2] = cos (angle);
        break;
    case 2: /* y-axis */
        xform[0][0] = cos (angle);
        xform[0][1] = 0.0;
        xform[0][2] = - sin (angle);
        xform[1][0] = 0.0;
        xform[1][1] = 1.0;
        xform[1][2] = 0.0;
        xform[2][0] = sin (angle);
        xform[2][1] = 0.0;
        xform[2][2] = cos (angle);
        break;
    case 3: /* z-axis */
        xform[0][0] = cos (angle);
        xform[0][1] = sin (angle);
        xform[0][2] = 0.0;
        xform[1][0] = - sin (angle);
        xform[1][1] = cos (angle);
        xform[1][2] = 0.0;
        xform[2][0] = 0.0;
        xform[2][1] = 0.0;
        xform[2][2] = 1.0;
        break;
    default:
        /* if an illegal axis is specified, do nothing */
        return;
        break;
    }

    /* calculate new matrix, write to dcm structure */
    for (ii = 0; ii < 3; ii++) {
        for (jj = 0; jj < 3; jj++) {
            dcm->data[ii][jj] = 0.0;
            for (kk = 0; kk < 3; kk++) {
                dcm->data[ii][jj] += (xform[ii][kk] * in_data[kk][jj]);
            }
        }
    }
}

```

```

/*
DoMedRate -- executive for medium rate processing module; primary
entry point for medium rate processing, called from rate group
control module
*/

void DoMedRate (int nav_num)
{
    /* pointers to shared data area */
    struct nav_mrIn_ref *mrIn = &nav_mrIn[nav_num];
    struct nav_mrOut_ref *mrOut = &nav_mrOut[nav_num];
    struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];
    struct nav_mrParm_ref *mrParm = &nav_mrParm[nav_num];
    struct nav_ctlLoc_ref *ctlLoc = &nav_ctlLoc[nav_num];
    struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];

    int ii;

    /* execute the navigation algorithm */
    nav_alg (nav_num);
    addNavHistory (nav_num);

    /* if necessary, reinitialize the medium rate accumulator registers */
    if (mrLoc->restartMedRateFlag) {
        mrLoc->restartMedRateFlag = 0;
        for (ii = 0; ii < 3; ii++) {
            mrLoc->dvi_s[ii] = 0.0;
            mrLoc->dvi_ss[ii] = 0.0;
        }
    }

    /* accumulate the sums */
    med_accum_sums (nav_num);

    /* if we've received a GPS update... */
    if (mrIn->gpsUpdateFlag) {

        /* record the time */
        mrLoc->timeLastGps = mrIn->medValidTime;
        /* lOut->updateTime = mrLoc->timeLastGps;--needs to be done in lorate */

        /* add position/velocity corrections from low rate Kalman filter */
        medrate_reset (nav_num);

        /* record reset time for guidance */
        mrLoc->tResetPrev = mrIn->update_time;

        mrLoc->step_change = 1;
        mrLoc->restartMedRateFlag = 1;

        /* compute velocity at time of GPS interrupt */
        prepGpsMeasmt (nav_num);
        for (ii = 0; ii < 3; ii++) {
            mrOut->dvi_sum_med[ii] = mrLoc->dvi_s[ii];
            mrOut->dvi_sum_sum[ii] = mrLoc->dvi_ss[ii];
        }
    }
}

```

```

/* note: the cumulative sums of dvi are copied here to stable
locations, for subsequent copy to the low rate module; the
cumulative sums are complete only at time of an interrupt,
from the perspective of the low rate module */
}

/* prep data for output to guidance system */
nav_output (nav_num);

/* send out the GPS receiver aiding data once every fifth medrate call */
if (++mrLoc->rec_aid_count >= mrParm->rcvr_aid_freq) {
    receiver_aid (nav_num);
    mrLoc->rec_aid_count = 0;
}

/* prep the output data for telemetry */
setup_telemetry (nav_num);

return;
}

/*
nav_init -- initialize the navigation algorithm data, using the
first GPS data
*/

static void nav_init (int nav_num)
{
    /* pointers to shared data area */
    struct nav_mrIn_ref *mrIn = &nav_mrIn[nav_num];
    struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];
    struct nav_mrParm_ref *mrParm = &nav_mrParm[nav_num];

    int ii;                /* local counter/index */
    double time_since_pvt; /* delta time since first GPS message */

    /* note: Due to considerations of dynamic range in the
representation of floating point numbers, the position vector has
been divided into two components. The fixed component, r0, is
the position at the first valid GPS data. The other component,
rw, is the subsequent offset from r0. */

    /* calculate the delta time since GPS PVT data was valid */
    time_since_pvt = mrIn->medValidTime - mrIn->gpsInterTime;

    /* get the GPS position data */
    for (ii = 0; ii < 3; ii++)
        mrLoc->r0[ii] = mrIn->GPS_pos[ii] * M2FT;

    /* mrLoc->vw[ii] calculated in vel_conversion */
    vel_conversion (nav_num);

    /* extrapolate to the current time (arbitrarily use rw[] for storage) */
    for (ii = 0; ii < 3; ii++)
        mrLoc->rw[ii] = time_since_pvt * mrLoc->vw[ii];

    /* (may not need this...) */
}

```

```

mrLoc->nwt = mrParm->mu / (mrParm->r_e * mrParm->r_e * mrParm->r_e);

/* initialize the gravity vector data */
grav_init (nav_num);

return;
)

/*
vel_conversion -- converts initial gps velocity (LL-ENU frame)
to ECEF frame velocity
*/
static void vel_conversion (int nav_num)
{
    struct nav_mrIn_ref *mrIn = &nav_mrIn[nav_num];
    struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];

    int ii, jj;          /* local counter indices */
    double xx, xxx;     /* local computation buffers */
    double rri, pri;    /* length, projected length of velocity vector */
    double slt, clt;    /* sine/cosine of latitude */
    double sln, cln;    /* sine/cosine of longitude */
    double c_enu_w[3][3]; /* direction cosine matrix, ENU to ECEF */

    xx = Square (mrIn->GPS_pos[0] * M2FT) + Square (mrIn->GPS_pos[1] * M2FT);
    xxx = xx + Square (mrIn->GPS_pos[2] * M2FT);
    rri = sqrt (xxx);
    pri = sqrt (xx);
    slt = mrIn->GPS_pos[2] * M2FT / rri;
    clt = pri / rri;
    sln = mrIn->GPS_pos[1] * M2FT / pri;
    cln = mrIn->GPS_pos[0] * M2FT / pri;

    /* create direction cosine matrix, LL-ENU to ECEF frame */
    c_enu_w[0][1] = - slt * cln;
    c_enu_w[0][0] = - sln;
    c_enu_w[0][2] = clt * cln;
    c_enu_w[1][1] = - slt * sln;
    c_enu_w[1][0] = cln;
    c_enu_w[1][2] = clt * sln;
    c_enu_w[2][1] = clt;
    c_enu_w[2][0] = 0.00;
    c_enu_w[2][2] = slt;

    /* calculate the transformed velocity vector */
    for (ii = 0; ii < 3; ii++) {
        mrLoc->vw[ii] = 0.0;
        for (jj = 0; jj < 3; jj++)
            mrLoc->vw[ii] += c_enu_w[ii][jj] * mrIn->GPS_vel[jj] * M2FT;
    }

    return;
}

```

```

/*
grav_init -- from r0, calculate g0 and gg0 for gravity calculation
*/

static void grav_init (int nav_num)
{
    struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];
    struct nav_mrParm_ref *mrParm = &nav_mrParm[nav_num];

    double rmag, rmag_sq; /* magnitude, squared magnitude of position */
    double ww, ff;       /* local calculation buffers */
    int ii, jj;          /* local counters */

    /* calculate the squared magnitude of the position vector */
    rmag_sq = (mrLoc->r0[0] * mrLoc->r0[0]) +
              (mrLoc->r0[1] * mrLoc->r0[1]) + (mrLoc->r0[2] * mrLoc->r0[2]);

    /* calculate the
    /* (should be double precision operation...) */
    rmag = sqrt (rmag_sq);

    ww = mrParm->mu / (rmag_sq * rmag);
    for (ii = 0; ii < 3; ii++)
        mrLoc->g0[ii] = ww * mrLoc->r0[ii];

    ff = 3.0 * ww / rmag_sq;
    for (ii = 0; ii < 3; ii++) {

        /* set the basic values */
        for (jj = 0; jj < 3; jj++)
            mrLoc->gamma0[ii][jj] = - ff * mrLoc->r0[ii] * mrLoc->r0[jj];

        /* augment the diagonal terms */
        mrLoc->gamma0[ii][ii] += ww;
    }

    return;
}

/*
setup_telemetry -- prep the gyro and accelerometer summations
for output through the telemetry stream (bounded)
*/

static void setup_telemetry (int nav_num)
{
    struct nav_mrIn_ref *mrIn = &nav_mrIn[nav_num];
    struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];

    int ii; /* local counter */
#define GYRO_SUM_BOUND 6300.0
#define ACCEL_SUM_BOUND 120.0

    /* cycle through the three axes... */
    for (ii = 0; ii < 3; ii++) {

        /* calculate cumulative gyro sum with threshold */

```

```

mrLoc->gyro_sum[ii] += mrIn->dthc_sum[ii];
while (mrLoc->gyro_sum[ii] > GYRO_SUM_BOUND)
  mrLoc->gyro_sum[ii] -= GYRO_SUM_BOUND;
while (mrLoc->gyro_sum[ii] < (- GYRO_SUM_BOUND))
  mrLoc->gyro_sum[ii] += GYRO_SUM_BOUND;

/* calculate cumulative accel sum with threshold */
mrLoc->accel_sum[ii] += mrIn->dvvvc_sum[ii];
while (mrLoc->accel_sum[ii] > ACCEL_SUM_BOUND)
  mrLoc->accel_sum[ii] -= ACCEL_SUM_BOUND;
while (mrLoc->accel_sum[ii] < (- ACCEL_SUM_BOUND))
  mrLoc->accel_sum[ii] += ACCEL_SUM_BOUND;
}

return;
}

/*
  nav_alg -- primary navigation algorithm
*/

static void nav_alg (int nav_num)
{
  /* pointers to shared data area */
  struct nav_mrIn_ref *mrIn = &nav_mrIn[nav_num];
  struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];
  struct nav_mrParm_ref *mrParm = &nav_mrParm[nav_num];
  struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];

  /* Computation Gravity Vector in Geocentric Coordinates */
  /* in addition to apparent forces and position/velocity integration */

  int ii,jj; /* counter */

  double R_geo_sq = mrLoc->R_geoc * mrLoc->R_geoc;
  double Re_div_R; /* Radius of Earth divided by local dist to Earth's Ctr. */
  double Re_div_R_sq;

  /* pre-calc of square of cos(lat) to
   * to save time in P[] and g0[] calcs
   */
  double clt_sq = mrLoc->clt * mrLoc->clt;

  /* pre-calc of cube of cos(lat) to
   * save time in P[] and g0[] calcs
   */
  double clt_3 = clt_sq * mrLoc->clt;

  /* preset of geocentric gravity */
  double g0_geoc[3] = {0.0, 0.0, -32.179};

  double we; /* earth angular velocity */

  /* save the previous values */
  for (ii = 0; ii < 3; ii++) {
    mrLoc->rw_prev[ii] = mrLoc->rw[ii];
    mrLoc->vw_prev[ii] = mrLoc->vw[ii];
  }
}

```

```

#if 0
/* set magnitude of the position from the Earth's Center using ECEF coords */
if (nav_ctlLoc.med_cnt < 1)
  mrLoc->R_geoc = sqrt((mrLoc->r0[0]*mrLoc->r0[0]) +
(mrLoc->r0[1]*mrLoc->r0[1]) +
(mrLoc->r0[2]*mrLoc->r0[2]));
else
  mrLoc->R_geoc = sqrt((mrLoc->rw[0]*mrLoc->rw[0]) +
(mrLoc->rw[1]*mrLoc->rw[1]) +
(mrLoc->rw[2]*mrLoc->rw[2]));

R_geo_sq = mrLoc->R_geoc * mrLoc->R_geoc;

/* Calculate the Pk terms (dependend on latitude)
 * equations from Britting, K. "Inertial Navigation Systems Analysis"
 */

mrLoc->P[0] = 1.0;
mrLoc->P[1] = 0.5*(3.0*clt_sq)-1.0;
mrLoc->P[2] = 0.5*(5.0*clt_3)-(3.0*mrLoc->clt);
mrLoc->P[3] = 0.125*((35.0*clt_3*mrLoc->clt)-(3.0*clt_sq)+3.0);

/* Sum up higher order terms which include J[k] and P[k].
 * This particular formulation goes to 4th order term as
 * taken from Britting, K. "Inertial Navigation Systems Analysis"
 * for G[r] this is not a problem, since the P[k] terms are unaffected
 * by the gradient operator (see Britting)--thus the use of the for loop
 * as a summation after initializing G[2]. However, for the G[phi]
 * term, the P[k] terms are altered by the gradient, so the entire
 * function is written out below as G[0].
 */

g0_geoc[2] = -1.0 * (mrParm->mu / R_geo_sq);

for (ii=1;ii<4;ii++) {
  g0_geoc[2] = g0_geoc[2] +
  (((double) (ii+2)) * mrParm->J[ii] *
  pow((mrLoc->R_geoc / physics.Req), (double) (ii+1)) *
  mrLoc->P[ii]);
}

Re_div_R = physics.Req / mrLoc->R_geoc;
Re_div_R_sq = Re_div_R * Re_div_R;

g0_geoc[0] = (double) -3.0 *
  ((mrParm->mu/R_geo_sq)*
  Re_div_R_sq * mrLoc->slt * mrLoc->clt *
  (mrParm->J[1] + 0.5 *
  (mrParm->J[2] * Re_div_R / mrLoc->clt * (5.0*clt_sq-1.0)) +
  ((5.0/6.0)*mrParm->J[3] * Re_div_R_sq * ((7.0*clt_sq-3.0)))));

g0_geoc[1] = 0.0;
#endif

/* *** Compute ECEF Gravity Vector *** */
mrLoc->g0[0] = -(mrLoc->slt*mrLoc->cln)*g0_geoc[0] + mrLoc->clt*g0_geoc[2];
mrLoc->g0[1] = -mrLoc->sln*g0_geoc[0];
mrLoc->g0[2] = -(mrLoc->clt*mrLoc->cln)*g0_geoc[0] -
  mrLoc->clt*mrLoc->sln*g0_geoc[1] - mrLoc->slt * g0_geoc[2];

```



```

/* calculate apparent forces in rotating w frame */
for(ii = 0; ii<3; ii++)
  mrLoc->G_vTerm[ii] = mrLoc->g0[ii] * ctlParm->del_t_med;

we = nav_hrParm[nav_num].we; /* (local copy of high rate parameter value) */

mrLoc->rrr[0] = we * ctlParm->del_t_med *
  (- we * (mrLoc->rw_prev[0] + mrLoc->r0[0]) -
  (2.0 * mrLoc->vw_prev[1]));
mrLoc->rrr[1] = we * ctlParm->del_t_med *
  (- we * (mrLoc->rw_prev[1] + mrLoc->r0[1]) +
  (2.0 * mrLoc->vw_prev[0]));
mrLoc->rrr[2] = 0.0;

/* calculate earth relative vel and pos in w frame */
for (ii = 0; ii <= 2; ii++) {

  mrLoc->vw[ii] += (mrIn->dvi_sum[ii] + mrLoc->G_vTerm[ii] - mrLoc->rrr[ii]);
  mrLoc->rw[ii] += (ctlParm->del_t_med * 0.5 * (mrLoc->vw[ii]+mrLoc-
  >vw_prev[ii]));
}
return;
}

/*
med_accum_sums -- medium rate accumulation of delta velocity sums
*/
static void med_accum_sums (int nav_num)
{
  /* pointers to shared data area */
  struct nav_mrIn_ref *mrIn = &nav_mrIn[nav_num];
  struct nav_mrOut_ref *mrOut = &nav_mrOut[nav_num];
  struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];
  struct nav_mrParm_ref *mrParm = &nav_mrParm[nav_num];
  struct nav_ctlLoc_ref *ctlLoc = &nav_ctlLoc[nav_num];
  struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];

  int ii;

  for (ii = 0; ii < 3; ii++) {

    /* accumulate the delta velocities over the GPS period */
    mrLoc->dvi_s[ii] += mrIn->dvi_sum[ii];

    /* also accumulate the sum of the sums */
    mrLoc->dvi_ss[ii] += mrLoc->dvi_s[ii];

  }

  return;
}

```

```

/*
addNavHistory -- append a navigation data set to the history array
*/
static void addNavHistory (int nav_num)
{
  /* pointers to shared data area */
  struct nav_mrIn_ref *mrIn = &nav_mrIn[nav_num];
  struct nav_mrOut_ref *mrOut = &nav_mrOut[nav_num];
  struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];
  struct nav_mrParm_ref *mrParm = &nav_mrParm[nav_num];
  struct nav_ctlLoc_ref *ctlLoc = &nav_ctlLoc[nav_num];
  struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];

  static int hist_len_prev; /* memory of previous length parameter */
  struct nav_hist_ref *nav_el; /* pointer to history element */
  int ii; /* local counter/index */
  int first_time; /* flag used for setting offset position */

  /* it is permissible to dynamically change the size of the circular
  buffer; however, to ensure data consistency, the buffer will be
  flushed when the size is changed */
  if (hist_len_prev != mrOut->hist_len) {

    /* sanity checks */
    if (mrOut->hist_len > MAX_NAV_HIST) {
      /* error handler here, if desired */
      mrOut->hist_len = MAX_NAV_HIST;
    }
    else if (mrOut->hist_len <= 0) {
      /* error handler here, if desired */
      mrOut->hist_len = MAX_NAV_HIST;
    }

    /* reinitialize parameters */
    mrOut->hist_last = -1;
    mrOut->hist_full = 0;

    /* remember this new size */
    hist_len_prev = mrOut->hist_len;

  }

  /* is this the first entry in the buffer? */
  if (mrOut->hist_last == -1)
    first_time = 1;
  else
    first_time = 0;

  /* calculate the buffer index */
  mrOut->hist_last++;

  /* if we get to the end of the buffer, circle around */
  if (mrOut->hist_last >= mrOut->hist_len) {
    mrOut->hist_last = 0;
    if (mrOut->hist_full == 0) {
      mrOut->hist_full = 1; /* mark it full */
    }
  }
}

```

```

/* now that the index is set, we'll actually load the data */
nav_el = mrOut->nav_history[mrOut->hist_last];
nav_el->time = mrIn->medValidTime;
for (ii = 0; ii < 3; ii++) {
    if (first_time)
        mrOut->pos0[ii] = mrLoc->r0[ii];
    nav_el->pos[ii] = mrLoc->rw[ii];
    nav_el->vel[ii] = mrLoc->vw[ii];
}
}

/*
nav_output -- copy calculated data to output buffer area for use by
guidance subsystem
*/

static void nav_output (int nav_num)
{
    /* pointers to shared data area */
    struct nav_mrIn_ref *mrIn = &nav_mrIn[nav_num];
    struct nav_mrOut_ref *mrOut = &nav_mrOut[nav_num];
    struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];
    struct nav_mrParm_ref *mrParm = &nav_mrParm[nav_num];
    struct nav_ctlLoc_ref *ctlLoc = &nav_ctlLoc[nav_num];
    struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];

    int ii, jj;          /* local counter/index */
    int err;            /* local error code buffer */
    double xx, xxx;     /* projection/radius squared (calculation buffer) */
    double pri, rri;    /* projection/radius */
    double vxx;         /* velocity squared (calculation buffer) */
    MATRIX_PP c_tmp_data; /* local pointer to matrix data */
    double lambda;      /* ??? angle */
    double cl, sl;      /* cosine/sine of lambda */
    double mu;          /* ??? angle */
    double cm, sm;      /* cosine/sine of mu */

    /* compare time of current position and velocity corrections
    to those used in the most recent navigation reset.  If current
    corrections are more recent, adjust guidance outputs so that they
    reflect the additional accuracy */

    mrLoc->dtGuidCor = mrIn->tIdGuidCor - mrLoc->tResetPrev;
    if (mrLoc->dtGuidCor > 0.5e0) {
        /* guidance corrs refer to later filter timethan last nav rest */
        for (ii = 0; ii < 3; ii++) {
            mrLoc->rwDeltaCor[ii] = mrIn->rwGuidCor[ii];
            mrLoc->vwDeltaCor[ii] = mrIn->vwGuidCor[ii];
        }
    }
    else if (mrLoc->dtGuidCor > 1.0e-6) {
        /* last nav reset refers to same filter time as guid corrs */
        for (ii = 0; ii < 3; ii++) {
            mrLoc->rwDeltaCor[ii] = mrIn->rwGuidCor[ii] - mrLoc->rwCorPrev[ii];
            mrLoc->vwDeltaCor[ii] = mrIn->vwGuidCor[ii] - mrLoc->vwCorPrev[ii];
        }
    }
}

```

```

else{
    /* guid coors are same as nav resets */
    for (ii = 0; ii < 3; ii++) {
        mrLoc->rwDeltaCor[ii] = 0.0e0;
        mrLoc->vwDeltaCor[ii] = 0.0e0;
    }
}

/* copy the navigation system time (for guidance and control) */
mrOut->time = mrIn->medValidTime;

/* note: multiplying by the frequency is the same as dividing by the
time period, and it avoids multiple division operations */

/* angular velocity and acceleration, in sensor frame */
for (ii = 0; ii < 3; ii++) {
    mrLoc->w_fb_i_s[ii] = mrIn->dthc_sum[ii] * ctlParm->freq_med;
    mrLoc->a_fb_i_s[ii] = mrIn->dvvc_sum[ii] * ctlParm->freq_med;
}

/* transform to fore-body frame */
for (ii = 0; ii < 3; ii++) {
    mrOut->w_fb_i_fb[ii] = 0.0;
    mrOut->a_fb_i_fb[ii] = 0.0;
    for (jj = 0; jj < 3; jj++) {
        mrOut->w_fb_i_fb[ii] += mrLoc->c_s_fb[jj][ii] * mrLoc->w_fb_i_s[jj];
        mrOut->a_fb_i_fb[ii] += mrLoc->c_s_fb[jj][ii] * mrLoc->a_fb_i_s[jj];
    }
}
#endif
/* if we're in down determination mode, don't do anything else */
if (mrOut->modeFlag == Down_Det) {
    mrOut->gdc_rdy = 0;
    return;
}
endif
/* if we got here, then we're in Navigate mode */
mrOut->gdc_rdy = 1;

/* position and velocity outputs */
/* use guidance delta corrections */
for (ii = 0; ii < 3; ii++) {
    mrOut->p_fb_e_e[ii] =
        mrLoc->rw[ii] + mrLoc->r0[ii] - mrLoc->rwDeltaCor[ii];
    mrOut->v_fb_e_e[ii] = mrLoc->vw[ii] - mrLoc->vwDeltaCor[ii];
}

/* calculate gravity vector in body frame */
err = MTv (mrLoc->g_fb, mrLoc->c_e_fb, mrLoc->g_e);
if (err) {
    /* error handler here, if desired */
}

/* (c_fb_e used instead of c_e_fb, per 20 feb 96 memo by Dowdle) */
for (ii = 0; ii < 3; ii++) {
    for (jj = 0; jj < 3; jj++) {
        mrOut->c_fb_e[ii][jj] = mrIn->c_e_fb_d[jj][ii];
    }
}
}

```

```

/* outputs for autopilot */
/* calculate roll, pitch, and yaw angles */

xx = Square (mrOut->p_fb_e_e[0]) + Square (mrOut->p_fb_e_e[1]);
xxx = xx + Square (mrOut->p_fb_e_e[2]);
rri = sqrt (xxx);
pri = sqrt (xx);
mrLoc->slt = mrOut->p_fb_e_e[2] / rri;
mrLoc->clt = pri / rri;
mrLoc->sln = mrOut->p_fb_e_e[1] / pri;
mrLoc->cln = mrOut->p_fb_e_e[0] / pri;

c_tmp_data = mrLoc->c_e_ln->data;
c_tmp_data[0][0] = - mrLoc->slt * mrLoc->cln;
c_tmp_data[0][1] = - mrLoc->sln;
c_tmp_data[0][2] = - mrLoc->clt * mrLoc->cln;
c_tmp_data[1][0] = - mrLoc->slt * mrLoc->sln;
c_tmp_data[1][1] = mrLoc->cln;
c_tmp_data[1][2] = - mrLoc->clt * mrLoc->sln;
c_tmp_data[2][0] = mrLoc->clt;
c_tmp_data[2][1] = 0.0;
c_tmp_data[2][2] = - mrLoc->slt;

/* calculate the transformation matrix for body to LL-NED */
err = MTM (mrLoc->c_ln_fb, mrLoc->c_e_ln, mrLoc->c_e_fb);
if (err) {
    /* error handler here, if desired */
}

/* attitude angles */
c_tmp_data = mrLoc->c_ln_fb->data;
mrOut->e_fb_ln[1] = - asin (c_tmp_data[2][0]); /* pitch */
mrOut->e_fb_ln[2] = atan2 (c_tmp_data[1][0], c_tmp_data[0][0]); /* yaw */
mrOut->e_fb_ln[0] = atan2 (c_tmp_data[2][1], c_tmp_data[2][2]); /* roll */

/* calculate the lat/lon/alt form of the current vehicle position */
RectangularToGeodetic (&mrLoc->lat, &mrLoc->lon, &mrLoc->alt, mrOut->p_fb_e_e);

/* calculate output altitude, relative to mean sea level */
mrOut->altitude = mrLoc->alt;

/* speed */
vxx = 0.0;
for (ii = 0; ii < 3; ii++)
    vxx += Square (mrLoc->vw[ii]);
mrOut->speed = sqrt (vxx);

/* transform output to guidance frame */

/* transform the fore-body velocity vector in the fore-body frame */
for (ii = 0; ii < 3; ii++) {
    mrOut->v_fb_e_fb[ii] = 0.0;
    for (jj = 0; jj < 3; jj++) {
        mrOut->v_fb_e_fb[ii] += (mrOut->c_fb_e[ii][jj] * mrOut->v_fb_e_e[jj]);
    }
}

/* feed through the quaternion from body to ecef as calculated by hirate */
/* Mod by E. Bailey 4/4/00 */
for(ii=0; ii<3; ii++)

```

```

mrOut->q_pw[ii] = mrIn->q_pwInit[ii];

/* feed through the gravity vector as calculated by nav_alg()
 * converted by the calculated DCM c_fb_e, transposed */
for(ii=0; ii<3; ii++){
    mrOut->g_fb[ii] = 0.0;
    for(jj=0; jj<3; jj++)
        mrOut->g_fb[ii] += mrOut->c_fb_e[jj][ii] * mrLoc->g0[jj];
}

/*
medrate_reset -- add the position and velocity corrections from
the low rate filter
*/

static void medrate_reset (int nav_num)
{
    /* pointers to shared data area */
    struct nav_mrIn_ref *mrIn = &nav_mrIn[nav_num];
    struct nav_mrOut_ref *mrOut = &nav_mrOut[nav_num];
    struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];
    struct nav_mrParm_ref *mrParm = &nav_mrParm[nav_num];
    struct nav_ctlLoc_ref *ctlLoc = &nav_ctlLoc[nav_num];
    struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];

    int ii, jj; /* local counter/index */
    struct nav_hist_ref *nav_el; /* pointer to history element */

    /* apply corrections to position and velocity */
    for (ii = 0; ii < 3; ii++) {
        mrLoc->rw[ii] -= mrIn->rwCor[ii];
        mrLoc->vw[ii] -= mrIn->vwCor[ii];
        mrLoc->vw_prev[ii] -= mrIn->vwCor[ii];
    }

    /* also apply corrections to navigation data history */
    for (jj = 0; jj < mrOut->hist_len; jj++) {

        nav_el = mrOut->nav_history[jj];
        for (ii = 0; ii < 3; ii++) {
            nav_el->pos[ii] -= mrIn->rwCor[ii];
            nav_el->vel[ii] -= mrIn->vwCor[ii];
        }
    }
    for(ii=0; ii<3; ii++) {
        mrLoc->rwCorPrev[ii] = mrIn->rwCor[ii];
        mrLoc->vwCorPrev[ii] = mrIn->vwCor[ii];
    }

    return;
}

```

```

/*
  prepGpsMeasmt -- compute the velocity at the time of GPS interrupt
*/

static void prepGpsMeasmt (int nav_num)
{
  /* pointers to shared data area */
  struct nav_mrIn_ref *mrIn = &nav_mrIn[nav_num];
  struct nav_mrOut_ref *mrOut = &nav_mrOut[nav_num];
  struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];
  struct nav_mrParm_ref *mrParm = &nav_mrParm[nav_num];
  struct nav_ctlLoc_ref *ctlLoc = &nav_ctlLoc[nav_num];
  struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];

  double s; /* proportion of medium rate period before GPS interrupt */
  int ii; /* local counter/index */

  /* calculate proportion of medium rate interval before GPS interrupt */
  /* (multiply by frequency instead of dividing by time period) */
  s = (mrIn->gpsInterTime + ctlParm->del_t_med - mrIn->medValidTime)
    * ctlParm->freq_med;

  /* calculate the velocity at time of GPS interrupt */
  for (ii = 0; ii < 3; ii++) {
    mrOut->vwGpsTime[ii] =
      mrLoc->vw_prev[ii] + mrIn->dvi_sum_left[ii] +
      (s * (- mrLoc->grv[ii] - mrLoc->rrr[ii]));
  }
}

/*
  receiver_aid -- sends inertial information to aid GPS
*/

static void receiver_aid (int nav_num)
{
  /* pointers to shared data area */
  struct nav_mrIn_ref *mrIn = &nav_mrIn[nav_num];
  struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];

  double posToRcvr[3]; /* buffer for calculating position */
  int ii; /* local counter/index */
  unsigned int inertialStatus; /* buffer for constructing status word */
#define AIDING_VELOCITY_VALID 0x00008000
#define AIDING_POSITION_VALID 0x00004000
#define AIDING_STEP_CHANGE 0x00002000
#define AIDING_DATA_DEGRADED 0x00001000

  /* calculate status word */
  inertialStatus = AIDING_VELOCITY_VALID | AIDING_POSITION_VALID;
  if (mrLoc->step_change) {
    inertialStatus = inertialStatus | AIDING_STEP_CHANGE;
    mrLoc->step_change = 0;
  }
  if ((mrIn->medValidTime - mrLoc->timeLastGps) > 2.0)
    inertialStatus = inertialStatus | AIDING_DATA_DEGRADED;
}

```

```

/* calculate current position */
for (ii = 0; ii < 3; ii++)
  posToRcvr[ii] = mrLoc->rw[ii] + mrLoc->r0[ii];

/* send to receiver */
#if (sun || sgi)
#else
  SendInertialAidingRcvrMsg (inertialStatus, mrIn->medValidTime,
    posToRcvr, mrLoc->vw);
#endif

return;
}

/*****
 *
 * FUNCTION NAME:      MR_Reads
 *
 * DESCRIPTION:      Reads data from the common into MR 'local' variables
 *
 * ARGUMENTS:        none
 *
 * RETURNS:          GOOD_RETURN_CODE
 *
 *****/

Return_Code_Type MR_Reads (int nav_num)
  /* procedure to copy data from common.h to needed HR variables */
{
  /* pointers to shared data area */
  struct nav_mrIn_ref *mrIn = &nav_mrIn[nav_num];

  int ii, jj;
  DVhr2mr hr2mr;
  DVlhr2mr lr2mr;

  Get_MR_In(&hr2mr, &lr2mr, nav_num);

  /* data copies from local to medium rate module */

  /* data from high rate module to medium rate module */
  mrIn->medValidTime = hr2mr.medValidTime;
  mrIn->gpsInterTime = hr2mr.gpsInterTime;
  mrIn->gpsUpdateFlag = hr2mr.gpsUpdateFlag;
  /* dll june 97 read time ID matching rw and vw Cor */
  /* note on update time : low rate sets time ID of each update, and
  passes it to high rate . High sets med rate copy of this only
  at time of a high rate reset. med rate gets it as time ID of
  most recent high rate reset */
  mrIn->update_time = hr2mr.update_time;
  for (ii = 0; ii < 3; ii++) {
    mrIn->dvvc_sum[ii] = hr2mr.dvvc_sum[ii];
    mrIn->dthc_sum[ii] = hr2mr.dthc_sum[ii];
    mrIn->dvi_sum[ii] = hr2mr.dvi_sum[ii];
    mrIn->dvi_sum_left[ii] = hr2mr.dvi_sum_left[ii];
    mrIn->rwCor[ii] = hr2mr.rwCor[ii];
    mrIn->vwCor[ii] = hr2mr.vwCor[ii];
  }
}

```

```

    for (jj = 0; jj < 3; jj++)
        mrIn->c_e_fb_d[ii][jj] = hr2mr.c_e_fb[ii][jj];
}
for (ii=0; ii<3; ii++) {
    mrIn->rwGuidCor[ii] = lr2mr.rwGuidCor[ii];
    mrIn->vwGuidCor[ii] = lr2mr.vwGuidCor[ii];
}
mrIn->tIdGuidCor = lr2mr.tIdGuidCor;

/* data passed from GPS receiver -- time mark message (msg 4) */
if (mrIn->gpsUpdateFlag) {
    for (ii = 0; ii < 3; ii++) {
        mrIn->GPS_pos[ii] = navIn[nav_num].posGPS[ii];
        mrIn->GPS_vel[ii] = navIn[nav_num].velGPS[ii];
    }
}

/* initial attitude */
mrIn->q_pwInit[0] = navIn[nav_num].q_pwInit[0];
mrIn->q_pwInit[1] = navIn[nav_num].q_pwInit[1];
mrIn->q_pwInit[2] = navIn[nav_num].q_pwInit[2];
mrIn->q_pwInit[3] = navIn[nav_num].q_pwInit[3];

/* data passed from control module (and other places) */
mrIn->roll_rdy = navIn[nav_num].roll_rdy;

return GOOD_RETURN_CODE;
} /* end MR_Reads */

/*****
 *
 * FUNCTION NAME:          MR_Writes
 *
 * DESCRIPTION: Writes data from the MR 'local' variables into the common area
 *
 * ARGUMENTS:             none
 *
 * RETURNS:               GOOD_RETURN_CODE
 *
 *****/

Return_Code_Type MR_Writes (int nav_num)
/* procedure to copy data from common.h to needed MR variables */
{
    /* pointers to shared data area */
    struct nav_mrIn_ref *mrIn = &nav_mrIn[nav_num];
    struct nav_mrOut_ref *mrOut = &nav_mrOut[nav_num];
    struct nav_mrLoc_ref *mrLoc = &nav_mrLoc[nav_num];
    struct nav_mrParm_ref *mrParm = &nav_mrParm[nav_num];
    struct nav_ctlLoc_ref *ctlLoc = &nav_ctlLoc[nav_num];
    struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];

    int ii, jj;
    struct nav_hist_ref *src, *dest;
    DVmr2hr mr2hr;
    DVmr2lr mr2lr;

```

```

/* write guidance data from medium rate module */

for (ii = 0; ii < 3; ii++) {
    navOut[nav_num].p_fb_e_e[ii] = (double) mrOut->p_fb_e_e[ii];
    navOut[nav_num].v_fb_e_e[ii] = (double) mrOut->v_fb_e_e[ii];
    for (jj = 0; jj < 3; jj++) {
        navOut[nav_num].c_fb_e_e[ii][jj] = (double) mrOut->c_fb_e_e[ii][jj];
    }
    navOut[nav_num].q_b_e_e[ii] = (double) mrOut->q_pw[ii];
    navOut[nav_num].w_fb_i_fb[ii] = (double) mrOut->w_fb_i_fb[ii];
    navOut[nav_num].a_fb_i_fb[ii] = (double) mrOut->a_fb_i_fb[ii];
}
navOut[nav_num].alt = (double) mrOut->altitude;
navOut[nav_num].speed = (double) mrOut->speed;

/* data copies from medium rate module to local */

/* data from medium rate module to high rate module */
for (ii = 0; ii < 4; ii++)
    mr2hr.q_pw[ii] = mrOut->q_pw[ii];

/* data from medium rate module to low rate module */
mr2lr.modeFlag = mrOut->modeFlag;
mr2lr.hist_len = mrOut->hist_len;
for (ii = 0; ii < 3; ii++) {
    mr2lr.dvi_sum_med[ii] = mrOut->dvi_sum_med[ii];
    mr2lr.dvi_sum_sum[ii] = mrOut->dvi_sum_sum[ii];
    mr2lr.pos0[ii] = mrOut->pos0[ii];
}

for (ii = 0; ii < 3; ii++)
    mr2lr.pos0[ii] = mrOut->pos0[ii];

for (ii = 0; ii < mrOut->hist_len; ii++) {
    src = nav_mrOut[nav_num].nav_history[ii];
    dest = &mr2lr.nav_history[ii];
    dest->time = src->time;
    dest->pos[0] = src->pos[0];
    dest->pos[1] = src->pos[1];
    dest->pos[2] = src->pos[2];
    dest->vel[0] = src->vel[0];
    dest->vel[1] = src->vel[1];
    dest->vel[2] = src->vel[2];
}

mr2lr.hist_last = mrOut->hist_last;
mr2lr.hist_full = mrOut->hist_full;
mr2lr.hist_len = mrOut->hist_len;

Put_MR_Out(&mr2hr, &mr2lr, nav_num);

return GOOD_RETURN_CODE;
} /* end MR_Writes */

```

```

/*****
 *
 *      nav_data_vault.h
 *
 * Author: Erik S. Bailey & Keith Mason
 * Purpose: Defines the data vault sub-structures
 *****/

#ifdef __NAV_DATA_VAULT_H__
#define __NAV_DATA_VAULT_H__

#include "return_code.h"
#include "navFC_ref.h"

/* data from high rate module to medium rate module */
struct hr2mr_ref {
    double medValidTime;
    double gpsInterTime;
    double c_e_fb[3][3];
    double dvvc_sum[3];
    double dthc_sum[3];
    double dvi_sum[3];
    double dvi_sum_left[3];
    int    gpsUpdateFlag;
    double update_time;
    double rwCor[3];
    double vwCor[3];
};
typedef struct hr2mr_ref DVhr2mr;

/* data from high rate module to low rate module */
struct hr2lr_ref {
    double loValidTime;
    double gpsInterTime;
    MATRIX_TYPE phil2z[N_DYN][N_STAT];
    double last_reset_time;
};
typedef struct hr2lr_ref DVhr2lr;

/* data from medium rate module to high rate module */
struct mr2hr_ref {
    double q_pw[4];
};
typedef struct mr2hr_ref DVmr2hr;

/* data from medium rate module to low rate module */
struct mr2lr_ref {
    enum NavMode modeFlag;
    double dvi_sum_med[3];
    double dvi_sum_sum[3];
    struct nav_hist_ref nav_history[MAX_NAV_HIST];
    double pos0[3];
    int hist_len;
    int hist_last;
    int hist_full;
};
typedef struct mr2lr_ref DVmr2lr;

/* data from low rate module to high rate module */
struct lr2hr_ref {

```

```

    double update_time;
    double qCor[4];
    double gbiasCor[4];
    double gsfe_xCor;
    double abiasCor[3];
    double asfe2_xCor;
    double rwCor[3];
    double vwCor[3];
};

typedef struct lr2hr_ref DVlr2hr;

/* data from low rate module to medium rate module */
struct lr2mr_ref {
    int dummy; /* nothing here now, but dummy variable as a placeholder */
    double rwGuidCor[3];
    double vwGuidCor[3];
    double tIdGuidCor;
};

typedef struct lr2mr_ref DVlr2mr;

struct data_vault_ref {
    DVhr2mr hr2mr;
    DVhr2lr hr2lr;
    DVmr2hr mr2hr;
    DVmr2lr mr2lr;
    DVlr2hr lr2hr;
    DVlr2mr lr2mr;
};

Return_Code_Type Get_HR_In (DVlr2hr *p_lr2hr, DVmr2hr *p_mr2hr, int nav_num);
Return_Code_Type Put_HR_Out (DVhr2lr *p_hr2lr, DVhr2mr *p_hr2mr, int nav_num);
Return_Code_Type Get_MR_In (DVhr2mr *p_hr2mr, DVlr2mr *p_lr2mr, int nav_num);
Return_Code_Type Put_MR_Out (DVmr2hr *p_mr2hr, DVmr2lr *p_mr2lr, int nav_num);
Return_Code_Type Get_LR_In (DVhr2lr *p_hr2lr, DVmr2lr *p_mr2lr, int nav_num);
Return_Code_Type Put_LR_Out (DVlr2hr *p_lr2hr, DVlr2mr *p_lr2mr, int nav_num);

#endif /* __NAV_DATA_VAULT_H__ */

```

```

/*
 * $Source: /hosts/dc2/users5/gab3/simlab/source/head_tracker/navigation/
nav_init.c,v $
 *
 * $Author: esb2110 $
 *
 * $Date: 2000/06/10 17:55:06 $
 *
 * $Revision: 1.3 $
 */

static char RCSid[] = "$Header: /hosts/dc2/users5/gab3/simlab/source/
head_tracker/navigation/nav_init.c,v 1.3 2000/06/10 17:55:06 esb2110 Exp $";

#include "navFC_ref.h"
#include "matrixx.h"
#include "simio.h"
#include "return_code.h"
#include "nav_highrate.h"
#include "nav_medrate.h"
#include "nav_lorate.h"

/* forward declarations */
void DoInit (int nav_num);

/*
 * DoInit -- executive for all rate group initialization
 */

void DoInit (int nav_num)
{
    /* pointers to shared data area */
    struct nav_ctlLoc_ref *ctlLoc = &nav_ctlLoc[nav_num];
    struct nav_ctlParm_ref *ctlParm = &nav_ctlParm[nav_num];

    int ii;

    ctlParm->freq_med = ctlParm->freq_hi / ((double) ctlParm->del_c_med);
    ctlParm->del_t_hi = 1.0 / ctlParm->freq_hi;
    ctlParm->del_t_med = 1.0 / ctlParm->freq_med;
    ctlParm->del_t_gps = 1.0 / ctlParm->freq_gps;

    /* initialize counters */
    ctlLoc->hi_cnt = 0;
    ctlLoc->med_cnt = 0;
    ctlLoc->lo_cnt_pps = 0;
    ctlLoc->lo_cnt_los = 0;
    ctlLoc->hi_cnt_med = 1;

    /* internal initialization for each rate function */
    filter_init_hi (nav_num);
    filter_init_med (nav_num);
    filter_init_lo ();

    return;
}

```

```

)

/*
 * matrix_err_handler -- general utility function for handling errors
 * that occur during access of matrix and kalman library routines;
 * err_code is the error code returned from the library function,
 * while source_group is a coded ID of the routine from which the
 * library call is made
 */

int matrix_err_handler (int err_code, int source_group)
{
    /* identify the calling function */
    switch (source_group) {
    case HR_INIT:
        therr ("Matrix error during high rate initialization: ");
        break;
    case MR_INIT:
        therr ("Matrix error during medium rate initialization: ");
        break;
    case LR_INIT:
        therr ("Matrix error during low rate initialization: ");
        break;
    default:
        therr ("Matrix error during initialization in unknown function: ");
        break;
    }

    /* mask to get only the pertinent bits */
    switch (err_code & 0x7) {
    case ERR_MXX_TYPE:
        therr2 ("wrong matrix element type\n");
        break;
    case ERR_MXX_MATCH:
        therr2 ("mismatched element sizes\n");
        break;
    case ERR_MXX_EXC:
        therr2 ("dimension exceeds capacity\n");
        break;
    case ERR_MXX_PTR:
        therr2 ("NULL pointer\n");
        break;
    default:
        therr2 ("unexpected error code %d\n", err_code);
        break;
    }
    if (err_code & ERR_MXX_LWR)
        therr2 (" (from a lower level call)\n");
}

```

```
/* return_code.h
 * contains the typedef for the rate group read & writes from the data vault
 */

#ifndef __RETURN_CODE_H__
#define __RETURN_CODE_H__

typedef int Return_Code_Type;
#define GOOD_RETURN_CODE 0

#endif /* __RETURN_CODE_H__ */
```



```
/* Status.h
 * Contains the typedef and defines the variables of "STATUS"
 */
```

```
#ifndef __STATUS_H_
#define __STATUS_H_

typedef int STATUS;
#define OK 0
#define ERROR (-1)
#define TRUE 1
#define FALSE 0

#endif /* __STATUS_H_ */
```

```
/******
 *
 * WGS84.h
 *
 * Prototypes for routines programmed by Matt Bottkol
 * that are in WGS84.c, adapted for the TPD/CGT sim
 * by E. Bailey 17 August, 1999
 *****/
```

```
void GeodeticToRectangular(double *x, double phi, double lambda, double h);
void RectangularToGeodetic(double *phi, double *lambda, double *h, double *x);
double Geodetic_N(double phi);
double GeodeticLat(double x, double y, double z);
```

```

/*****
*
*                               WGS84.c
*
* Geometric conversion routines programmed by Matt
* Bottkol adapted for the TPD/CGT sim
* by E. Bailey 17 August, 1999
*****/

#include "WGS84.h"
#include "navFC_ref.h"
#include <math.h>

/* B_WGS84 and E_WGS84 from DMA TR 8350.2   */
/* 30 September 1987                        */
/*                                          */
/* other geometrical constants are derived */
/*                                          */

/* Note: the commented numbers below are in SI units, whereas the TOPART
* code uses ft/slug/sec units. Therefore, the pointers below to the
* values defined in navFC.spec are in ft/slug/sec units and do not
* equal their SI counterparts, which are commented out.
*/

#define eSquared_WGS84nav_mrParm[0].sqellip /*0.0066943799901378*/
#define axisRatio_WGS84nav_mrPar[0].m.axis_ratio /*0.99664718933525*/
#define axisRatioSquared_WGS84      nav_mrParm[0].sqaxis_ratio /
*0.99330562000936*/
#define semiMajor_WGS84nav_mrParm[0].smajax /*6378136.9999547*/
#define semiMinor_WGS84nav_mrParm[0].sminax /*6356752.3142*/
#define ellip_WGS84nav_mrParm[0].ellip /*0.0818191908426*/
#define earthRadius_WGS842.08905664602e7 /*6367444.6570775*/

#define B_WGS84      nav_mrParm[0].sminax /*6356752.3142*/
#define E_WGS84      nav_mrParm[0].ellip /*0.0818191908426*/

#define ESQUARED_WGS84nav_mrParm[0].sqellip /*0.0066943799901378*/
#define AXISRATIO_WGS84nav_mrParm[0].axis_ratio /*0.99664718933525*/
#define A_WGS84nav_mrParm[0].smajax /*6378136.9999547*/

/* MU, OMEGA_E values from GPS ICD */
#define MU      nav_mrParm[0].mu /*3.986005e14*/
#define J2      nav_mrParm.J[1] /*1082.64e-6*/
#define J4      nav_mrParm.J[2] /*-2.4e-6*/
#define G      physics.gc /*9.8066352*/
#define OMEGA_E      physics.we /*7292115.1467e-11*/

#define LIGHTC      nav_mrParm[0].lightc /*299792458.0*/
#define LIGHT_MILLISEC      nav_mrParm[0].light_milsec /*299792.458*/

#define GEODETTIC_LAT_TOL 1.e-8

/*****
*
* RectangularToGeodetic(x,&phi,&lambda,&h)
* INPUT: earth relative rectangular coordinates in vector x[0],...x[2]
* OUTPUT: geodetic latitude, longitude, altitude (radians & meters)
*
*****/

```

```

*****/
void RectangularToGeodetic(double *phi,double *lambda,double *h,double *x)
{
    *phi = GeodeticLat(x[0],x[1],x[2]);
    *lambda = atan2(x[1],x[0]);
    /* *h = abs(x)/cos(*phi) - Geodetic_N(*phi); */
    *h = sqrt(x[0]*x[0]+x[1]*x[1])/cos(*phi) - Geodetic_N(*phi);
}/* end RectangularToGeodetic */

/*****
*
* GeodeticToRectangular(phi,lambda,h,x)
* INPUT: geodetic latitude, longitude, altitude (radians & meters)
* OUTPUT: earth relative rectangular coordinates in vector x[0],...x[2]
* (allocated by caller)
*
*****/

void GeodeticToRectangular(double *x, double phi,double lambda,double h)
{
    double N;
    N = Geodetic_N(phi);
    x[0] = (N+h)*cos(phi)*cos(lambda);
    x[1] = (N+h)*cos(phi)*sin(lambda);
    x[2] = (axisRatioSquared_WGS84*N+h)*sin(phi);
}/* end GeodeticToRectangular */

/*****
*
* phi = GeodeticLat(x,y,z)
* computes geodetic latitude of point with
* rectangular coordinates (x,y,z)
*
*****/

double GeodeticLat(double x,double y,double z)
{
    double phi0,phi1,tan_geocentricPhi;

    /*iterationCount=1;*/
    tan_geocentricPhi = z/sqrt(x*x+y*y);
    phi0 = 0.;
    phi1 = atan(tan_geocentricPhi/axisRatioSquared_WGS84);

    /* Picard iteration */
    while(fabs(phi1-phi0) > GEODETTIC_LAT_TOL)
    {
        phi0 = phi1;
        phi1 = atan(tan_geocentricPhi *
            (1.0 + eSquared_WGS84*Geodetic_N(phi0)*sin(phi0)/z));
    }
}

```

```
    return phi;
}/*    end GeodeticLat    */

/*.....
 *
 * N = Geodetic_N(phi)
 *computes the geodetic parameter N,
 *one of the principal radii of curvature of WGS84 ellipsoid
 *
 *.....*/

double Geodetic_N(double phi)
{
    return semiMajor_WGS84/sqrt(1-eSquared_WGS84*sin(phi)*sin(phi));
}/*    end GeodeticLat    */
```

## Appendix F

### Navigation Filter MATLAB Code

The following code is the code for implementing the Kalman Filter and performing the Monte Carlo Analysis.



```

function tao = bounding_alg(q_b_e,R_T38_nav,R_HELMET_nav, R_T38err_est,
R_HELMETerr_est, RelPos_Err_Est, dt)

% Generates the proper tao depending on proximity to the bounding
% box which relates the two

%-----%
% Initial Conditions %
%-----%

%-- Bounding Box Dimensions --%
twosigma = [1.5, 1.5, 1.5];% [ft] (in both directions)

%-- Nominal Lever Arm --%
nom_la_b = [12.083,0.0,-4.33]';% [ft] (in body frame)

%-- Limits on Tao --%

tao_lim = [10, 100;
           10, 100;
           10, 100];

%-----%
% Main Calculations %
%-----%

%-- Calculate C_b_e from q_b_e --%
C_b_e = q2C(q_b_e);

%-- Calculate navigated relative position --%

RelPos_nav = (R_T38_nav-R_T38err_est) - (R_HELMET_nav-R_HELMETerr_est); % - (C_b_e
* RelPos_Err_Est) + (C_b_e * nom_la_b);

%keyboard

%-- Calculate Tao --%

for (i = 1:1:3)
var = (0.5 * twosigma(i))^2;
tao(i) = abs(-(tao_lim(i,2) - tao_lim(i,1))/var * ...
             abs(RelPos_nav(i)) + tao_lim(i,2));
end

```

```

function K = calc_K(P_pre, H, R);

% K = calc_K(P_pre, H, R);
%
% Calculates the gain matrix for a particular measurement update given
% H and R for that particular update.
%
% Inputs:
% P_pre = propogated state estimate covariance matrix prior to next update
% x_pre = propagated state estimate prior to next update
%
% Outputs:
% K = Kalman Gain Matrix
%
% Written by E. Bailey February 26, 2000.

K = P_pre * H' * inv(H * P_pre * H' + R);

```

```
%-----%
%--- ERROR_CALC.M ---%
%-----%
```

```
q_h_e_avg = (q_h_e + q_h_e_old) * 0.5;
q_h_e_star = q_h_e_avg ./ norm(q_h_e_avg,2);
```

```
qmultT38 = qmulq(q_b_e_T38,q_b_e);
qmultHELMET = qmulq(q_h_e_HELMET,q_h_e_star);
```

```
error(i,:) = [time,(R_T38-R_T38_nav + x_est(1:3))', ...
(R_HELMET-R_HELMET_nav + x_est(10:12))'...
(Rdot_T38-Rdot_T38_nav + x_est(4:6))', ...
(Rdot_HELMET-Rdot_HELMET_nav + x_est(13:15))', ...
(2.0 * qmultT38(2:4) + x_est(7:9))', ...
(2.0 * qmultHELMET(2:4) + asin(x_est(16:18)))'];
```

```
%-----%
%--- FORM_Q.M ---%
%-----%
```

```
q_R_EGI = 0.0;% [feet]
q_R_MMISA = 0.0;% [feet]
q_V_EGI = 0.00032808 / 60;% [ft/sec @ 1 sec]
q_V_MMISA = 0.03280 / 60;% [ft/sec @ 1 sec]
q_att_EGI = 4.363323e-5 / 60; % [rad @ 1 sec]
q_att_MMISA = 5.23598e-4 / 60; % [rad @ 1 sec]
q_gb_EGI = 0.0; % [rad/sec]
q_gsf_EGI = 0.0; % [rad/sec]
q_ab_EGI = 0.0; % [rad/sec]
q_asf_EGI = 0.0; % [rad/sec]
q_gb_MMISA = 0.0;% [rad/sec]
q_gsf_MMISA = 0.0;% [na]
q_ab_MMISA = 0.0;% [ft/s^2]
q_asf_MMISA = 0.0;% [ft/s^2]
q_rel = sqrt(1.0^2 * (1-exp(-2*meas_rate_dt/tao)));% [feet]
```

```
q_diag = [ones(1,3).*q_R_EGI, ones(1,3).*q_V_EGI, ones(1,3).*q_att_EGI, ...
ones(1,3).*q_R_MMISA, ones(1,3).*q_V_MMISA, ones(1,3).*q_att_MMISA, ...
ones(1,3).*q_rel, ones(1,3).*q_gb_EGI, ones(1,3).*q_gsf_EGI, ...
ones(1,3).*q_ab_EGI, ones(1,3).*q_asf_EGI, ones(1,3).*q_gb_MMISA, ...
ones(1,3).*q_gsf_MMISA, ones(1,3).*q_ab_MMISA, ones(1,3).*q_asf_MMISA];
```

```
clear q_R_EGI q_R_MMISA q_V_EGI q_V_MMISA q_att_EGI q_att_MMISA q_gb_EGI
q_gsf_EGI ...
q_ab_EGI q_asf_EGI q_gb_MMISA q_gsf_MMISA q_ab_MMISA q_asf_MMISA
```

```
Q = diag(q_diag.^2,0);
```

```
clear q_diag ii
```

```
function list = gen_mcVals(num_runs)

% list = generate_mcarlo_vals(num_runs, sigmax2)
% Generates a list of values within a 2-sigma normal distribution
% to be used for initial attitude errors for the Head Tracker Sim

sigmas = [13.2 * ones(1,3), 1.0 * ones(1,3), 0.001 * ones(1,3), ...
10.0 * ones(1,3), 2.0 * ones(1,3), 0.02 * ones(1,3)];

for(i = 1:num_runs)
    list(i,:) = [i, sigmas .* randn(1,length(sigmas))];
end
```

```
save mcarlo_vals.txt list -ASCII -DOUBLE
```





```

%-----%
%--- HELMET_KF.M ---%
%-----%

%-----%
%-- INITIALIZE VARIABLES --%
%-----%

data_dt = data(2,1) - data(1,1);
start_time = data(1,1);
time = start_time;
end_time = data(end,1);

high_rate_dt = 0.01;
med_rate_dt = 0.02;
meas_rate_dt = 1.0;

tao = 5 * ones(3,1);

i = 0;

phill_T38 = eye(9);
phill_HELMET = eye(9);
x_est = zeros(45,1);
init_P;
form_Q;

%-----%
%-- MAIN FILTER ROUTINE --%
%-----%

for (time = start_time:data_dt:end_time-data_dt)

    if( (abs(round(time) - time) < 0.000001))
        i = i+1; % increment row index for output and error data matrices
        fprintf('Updating at t=%4.2f\n',time);

        %-- unlog data at valid measurement time --%
        q_h_e_old = q_h_e;
        [q_b_e, q_h_e, q_b_e_T38, q_h_e_HELMET] =
unlog_quaternions(data,time,data_dt);
        [R_T38, R_HELMET, R_T38_nav, R_HELMET_nav] = unlog_R(data,time,data_dt);
        [Rdot_T38, Rdot_HELMET, Rdot_T38_nav, Rdot_HELMET_nav] =
unlog_Rdot(data,time,data_dt);

        %-- Bounding Algorithm --%

        %   tao = bounding_alg(q_b_e_T38, R_T38_nav, R_HELMET_nav, x_est(1:3), ...
        %   x_est(10:12), x_est(19:21), meas_rate_dt);

        phi_rel = expm(diag(-1./tao,0));

        [dvv_b_i_T38, dvv_b_i_HELMET] = unlog_del_velo(data,time,data_dt);

        %-- formulate sub-phi matrices along diagonal --%
        prev_phill_T38 = phill_T38;
        phill_T38 = make_phill(dvv_b_i_T38, med_rate_dt);
        phill_T38 = phill_T38 * prev_phill_T38;
        prev_phill_HELMET = phill_HELMET;
        phill_HELMET = make_phill(dvv_b_i_HELMET, med_rate_dt);
        phill_HELMET = phill_HELMET * prev_phill_HELMET;
    end
end


```

```

phill_HELMET = make_phill(dvv_b_i_HELMET, med_rate_dt);
phill_HELMET = phill_HELMET * prev_phill_HELMET;

%-- note that time is at its final value for unlogging phil2's --%
[phill2_T38, phill2_HELMET] = unlog_phi12(data,time,data_dt);

%-- construct main PHI matrix for entire system --%
PHI = make_PHI(phill_T38, phill_HELMET, phill2_T38, phill2_HELMET, phi_rel);

%-- reset the phill matrices to I --%
phill_T38 = eye(9);
phill_HELMET = eye(9);

%-- propogate filter estimate from previous update --%
[x_est, P] = propagate(PHI,x_est,P,Q);

%-- get measurement data --%
[z_mHxhat,z] = measurement('both', x_est, data, time, data_dt);
[H,R] = H_R_matrix('both', q_b_e_T38);

%-- calculate kalman gain matrix --%
K = calc_K(P,H,R);

%-- update state estimate and covariance matrix --%
[x_est,P] = update(x_est,P,K,H,R,z);

out(i,:) = [time, x_est',z_mHxhat', z'];
P_out(i,:) = [time, sqrt(diag(P))'];

error_calcs % MATLAB script for calculating errors ./error_calcs.m

else
    [q_b_e, q_h_e, q_b_e_T38, q_h_e_HELMET] =
unlog_quaternions(data,time,data_dt);

    if(~mod(time, med_rate_dt))
        [dvv_b_i_T38, dvv_b_i_HELMET] = unlog_del_velo(data,time,data_dt);

        %-- formulate sub-phi matrices along diagonal --%
        prev_phill_T38 = phill_T38;
        phill_T38 = make_phill(dvv_b_i_T38, med_rate_dt);
        phill_T38 = phill_T38 * prev_phill_T38;
        prev_phill_HELMET = phill_HELMET;
        phill_HELMET = make_phill(dvv_b_i_HELMET, med_rate_dt);
        phill_HELMET = phill_HELMET * prev_phill_HELMET;
    end
end

plot_filter_out % MATLAB script for plotting output
                % and errors ./plot_filter_out.m


```

```

%-----%
%----  INIT_P.M  ----%
%-----%

% RENAME BELOW VARS TO "VARIANCE" VARIABLES FOR CONSISTENCY
% IN NAMING VARIABLES

sig_R_EGI      = 3.0;% [feet]
sig_R_MMISA    = 10.0;% [feet]
sig_V_EGI      = 1.0;% [ft/sec]
sig_V_MMISA    = 5.0;% [ft/sec]
sig_att_EGI    = 0.001; % [rad]
sig_att_MMISA  = 0.17;% [rad]
sig_gb_EGI     = 1.69684788388e-8; % [rad/sec]
sig_gsf_EGI    = 2e-6; % [rad/sec]
sig_ab_EGI     = 0.0000322; % [ft/sec/sec]
sig_asf_EGI    = 0.0001; % [rad/sec]
sig_gb_MMISA   = 0.000097;% [rad/sec]
sig_gsf_MMISA  = 0.0002;% [na]
sig_ab_MMISA   = 0.08855;% [ft/s^2]
sig_asf_MMISA  = 0.0002;% [na]
sig_rel        = 0.75;% [feet]

p_diag = [ones(1,3).*sig_R_EGI, ones(1,3).*sig_V_EGI, ones(1,3).*sig_att_EGI, ...
ones(1,3).*sig_R_MMISA, ones(1,3).*sig_V_MMISA, ones(1,3).*sig_att_MMISA, ...
ones(1,3).*sig_rel, ones(1,3).*sig_gb_EGI, ones(1,3).*sig_gsf_EGI, ...
ones(1,3).*sig_ab_EGI, ones(1,3).*sig_asf_EGI, ones(1,3).*sig_gb_MMISA, ...
ones(1,3).*sig_gsf_MMISA, ones(1,3).*sig_ab_MMISA, ones(1,3).*sig_asf_MMISA];

clear sig_R_EGI sig_R_MMISA sig_V_EGI sig_V_MMISA sig_att_EGI sig_att_MMISA
sig_gb_EGI sig_gsf_EGI ...
sig_ab_EGI sig_asf_EGI sig_gb_MMISA sig_gsf_MMISA sig_ab_MMISA sig_asf_MMISA
sig_rel

P = diag(p_diag.^2,0);

clear p_diag

```

```

function PHI = make_PHI(philla, phil1b, phil2a, phil2b, phi_rel)

% PHI = make_PHI(philla, phil1b, phil2a, phil2b, phi_rel)
%
% formulates PHI from the five sub matrices

PHI = eye(45,45);

%-- The PHI_11 components --%

PHI(1:9,1:9)      = philla;
PHI(10:18, 10:18) = phil1b;
PHI(19:21, 19:21) = phi_rel;

%-- The PHI_12 components --%

% PHI(1:9, 22:33) = zeros(9,12);
% PHI(10:18, 34:45) = zeros(9,12);

PHI(1:9, 22:33) = phil2a;
PHI(10:18, 34:45) = phil2b;

```

```

function phill = make_phill(delta_v,dt)

% phill = make_phill(delta_v,dt)
%
% makes the phill sub-matrix for the propagation
% matrix.

w_e = 7.2921e-5; % [rad/sec]

phill = eye(9);

phill(1:3,4:6) = eye(3,3) * dt;
phill(4,5) = -2 * w_e * dt;
phill(5,4) = 2 * w_e * dt;
phill(4:6,7:9) = [0 -delta_v(3) delta_v(2);
  delta_v(3) 0 -delta_v(1);
  -delta_v(2) delta_v(1) 0];
phill(7,8) = -w_e * dt;
phill(8,7) = w_e * dt;

```

```

%-----%
%--- MC_HELMET_KF.m ---%
%-----%

set = 10;
numruns = 20;

for(run = 1:numruns)

%-----%
%-- Batch DATA LOADER --%
%-----%

fprintf('loading data for set %d, run number %d ...\n',set,run)
eval(['data' num2str(run) ' = MC_load_setrun(' num2str(set) ', ' num2str(run)
');'])
fprintf('data saved as matrix data%d.\n',run)

%-----%
%-- INITIALIZE VARIABLES --%
%-----%
eval(['data = data' num2str(run) '; '])
data_dt = data(2,1) - data(1,1);
start_time = data(1,1);
time = start_time;
end_time = data(end,1);

high_rate_dt = 0.01;
med_rate_dt = 0.02;
meas_rate_dt = 1.0;

tao = 10.0 * ones(3,1);

i = 0;

phill_T38 = eye(9);
phill_HELMET = eye(9);
x_est = zeros(45,1);
init_P;
form_Q;

%-----%
%-- MAIN FILTER ROUTINE --%
%-----%

for (time = start_time:data_dt:end_time-data_dt)

  if( (abs(round(time) - time) < 0.000001))
    i = i+1; % increment row index for output and error data matrices
    fprintf('Run Number %d: Update at t=%4.2f\n', run, time);

    %-- unlog data at valid measurement time --%
    q_h_e_old = q_h_e;
    [q_b_e, q_h_e, q_b_e_T38, q_h_e_HELMET] =
unlog_quaternions(data,time,data_dt);
    [R_T38, R_HELMET, R_T38_nav, R_HELMET_nav] = unlog_R(data,time,data_dt);
    [Rdot_T38, Rdot_HELMET, Rdot_T38_nav, Rdot_HELMET_nav] =
unlog_Rdot(data,time,data_dt);

    %-- Bounding Algorithm --%

```

```

tao = bounding_alg(q_b_e_T38, R_T38_nav, R_HELMET_nav, x_est(1:3), ...
  x_est(10:12), x_est(19:21), meas_rate_dt);

phi_rel = expm(diag(-1./tao,0));

[divv_b_i_T38, divv_b_i_HELMET] = unlog_del_velo(data,time,data_dt);

%-- formulate sub-phi matrices along diagonal --%
prev_phill_T38 = phill_T38;
phill_T38 = make_phill(divv_b_i_T38, med_rate_dt);
phill_T38 = phill_T38 * prev_phill_T38;
prev_phill_HELMET = phill_HELMET;
phill_HELMET = make_phill(divv_b_i_HELMET, med_rate_dt);
phill_HELMET = phill_HELMET * prev_phill_HELMET;

%-- note that time is at its final value for unlogging phil2's --%
[phil2_T38, phil2_HELMET] = unlog_phil2(data,time,data_dt);

%-- construct main PHI matrix for entire system --%
PHI = make_PHI(phil1_T38, phil1_HELMET, phil2_T38, phil2_HELMET, phi_rel);

%-- reset the phill matrices to I --%
phill_T38 = eye(9);
phill_HELMET = eye(9);

%-- propogate filter estimate from previous update --%
[x_est, P] = propagate(PHI,x_est,P,Q);

%-- get measurement data --%
[z_mHxhat,z] = measurement('both', x_est, data, time, data_dt);
[H,R] = H_R_matrix('both', q_b_e_T38);

%-- calculate kalman gain matrix --%
K = calc_K(P,H,R);

%-- update state estimate and covariance matrix --%
[x_est,P] = update(x_est,P,K,H,R,z);

rtP_diag = sqrt(diag(P));

eval(['P_out' num2str(run) '(i,:) = [time, rtP_diag];'])

error_calcs % MATLAB script for calculating errors ./error_calcs.m
eval(['att_error' num2str(run) ' = [error(:,1),error(:,17:19)];'])

x_est = x_est';
z_mHxhat = z_mHxhat';
z = z';

eval(['filter_out' num2str(run) '(i,:) = [time, x_est, z_mHxhat, z];'])
x_est = x_est';

else
[q_b_e, q_h_e, q_b_e_T38, q_h_e_HELMET] = unlog_quaternions(data,time,data_dt);

if(~mod(time, med_rate_dt))
  [divv_b_i_T38, divv_b_i_HELMET] = unlog_del_velo(data,time,data_dt);

```

```

%-- formulate sub-phi matrices along diagonal --%
prev_phill_T38 = phill_T38;
phill_T38 = make_phill(divv_b_i_T38, med_rate_dt);
phill_T38 = phill_T38 * prev_phill_T38;
prev_phill_HELMET = phill_HELMET;
phill_HELMET = make_phill(divv_b_i_HELMET, med_rate_dt);
phill_HELMET = phill_HELMET * prev_phill_HELMET;
end
end

fprintf('Covariance output saved in P_out%d.\n',run)
fprintf('filter output data saved in filter_out%d.\n',run)

eval(['clear data' num2str(run)])

end

% cleanup unwanted variables
disp('cleaning up unwanted variables...')
clear C_b_e H K P PHI Q R R_HELMET R_HELMET_nav
clear R_T38 R_T38_nav Rdot_HELMET Rdot_HELMET_nav
clear Rdot_T38 Rdot_T38_nav
clear data data_dt divv_HELMET_sum divv_T38_sum divv_b_i_HELMET
clear divv_b_i_T38 end_time error error2 error3 error4
clear high_rate_dt i meas_rate_dt med_rate_dt nom_la
clear phill_HELMET phill_T38 phil2_HELMET phil2_T38 phi_rel
clear q_b_e q_b_e_T38 q_h_e q_h_e_HELMET q_rel qmultHELMET
clear qmultREL qmultREL_tr qmultT38 rtP_diag run set start_time
clear tao time x_est z z_mHxhat

clear numruns i

disp('saving processed data as "processed_data.mat"...')
save processed_data

disp('performing statistical analysis...')
stat_run

```

```

function data = MC_load_setrun(set, run)
% data = MC_load_setrun(set, run)
%
% loads data from given Monte Carlo Set and run number

if(run < 10)
    eval(['load /disk02/esb2110/matlab/Thesis_Data/monte_carlo/set_' num2str(set)
    '/MC_run_0' num2str(run) '.mat'])
    eval(['data = MC_run_0' num2str(run) ';''])
else
    eval(['load /disk02/esb2110/matlab/Thesis_Data/monte_carlo/set_' num2str(set)
    '/MC_run_' num2str(run) '.mat'])
    eval(['data = MC_run_' num2str(run) ';''])
end

data = data(:,2:end)

```

```

function [z_mHxhat, z] = measurement(meas_type, x_est, data, time, dt)

% [z_mHxhat, z] = measurement(meas_type, x_est, data, time, dt)
%
% Given inputted types (gps, relative, or both) generates
% proper z-vector for the Kalman Filter.

if(meas_type == 'gpsM')

    [GPS_R, GPS_V] = unlog_GPS(data, time, dt);

    [R_T38, R_HELMET, R_T38_nav, R_HELMET_nav] = unlog_R(data, time, dt);
    R_cor_T38 = x_est(1:3);

    [Rdot_T38, Rdot_HELMET, Rdot_T38_nav, Rdot_HELMET_nav] =
    unlog_Rdot(data, time, dt);
    V_cor_T38 = x_est(4:6);
    z = [(R_T38_nav - GPS_R)', (Rdot_T38_nav - GPS_V)'];
    z_mHxhat = [(R_T38_nav - R_cor_T38) - GPS_R)', ((Rdot_T38_nav - V_cor_T38) -
    GPS_V)']';

elseif(meas_type == 'relM')

    [R_T38, R_HELMET, R_T38_nav, R_HELMET_nav] = unlog_R(data, time, dt);
    [q_b_e, q_h_e, q_b_e_T38, q_h_e_HELMET] = unlog_quaternions(data, time, dt);
    R_cor_T38 = x_est(1:3);
    R_cor_HELMET = x_est(10:12);
    markov = x_est(19:21);
    nom_la = [12.083, 0.0, -4.33]';
    C_b_e = q2C(q_b_e_T38);
    z = R_HELMET_nav - R_T38_nav - C_b_e' * nom_la;
    z_mHxhat = ((R_HELMET_nav - R_cor_HELMET) - (R_T38_nav - R_cor_T38) - ...
    (C_b_e' * (nom_la + markov)));

elseif(meas_type == 'both')

    [GPS_R, GPS_V] = unlog_GPS(data, time, dt);
    [q_b_e, q_h_e, q_b_e_T38, q_h_e_HELMET] = unlog_quaternions(data, time, dt);
    [R_T38, R_HELMET, R_T38_nav, R_HELMET_nav] = unlog_R(data, time, dt);
    R_cor_T38 = x_est(1:3);
    R_cor_HELMET = x_est(10:12);
    [Rdot_T38, Rdot_HELMET, Rdot_T38_nav, Rdot_HELMET_nav] =
    unlog_Rdot(data, time, dt);
    V_cor_T38 = x_est(4:6);
    markov = x_est(19:21);
    nom_la = [12.083, 0.0, -4.33]';
    C_b_e = q2C(q_b_e_T38);

    z = [ (R_T38_nav - GPS_R)', ...
    (Rdot_T38_nav - GPS_V)', ...
    (R_HELMET_nav - R_T38_nav - C_b_e' * nom_la)']';

    z_mHxhat = [ ((R_T38_nav - R_cor_T38) - GPS_R)', ...
    ((Rdot_T38_nav - V_cor_T38) - GPS_V)', ...
    ((R_HELMET_nav - R_cor_HELMET) - (R_T38_nav - R_cor_T38) - ...
    (C_b_e' * (nom_la + markov)))']';

end

```

```

function phill = make_phill(delta_v,dt)

% phill = make_phill(delta_v,dt)
%
% Forms phill matrix component for given
% delta-v inputted to it.

phill = eye(9,9);

phill(1:3,4:6) = eye(3,3) * dt;
phill(4,5) = -2 * w_e * dt;
phill(5,4) = 2 * w_e * dt;
phill(4:6,7:9) = [0 -delta_v(3) delta_v(2);
    delta_v(3) 0 -delta_v(1);
    -delta_v(2) delta_v(1) 0];
phill(7,8) = -w_e * dt;
phill(8,7) = w_e * dt;

```

```

%-----%
%---- PLOT_FILTER_OUT.M ----%
%-----%

figure(1)
clf

subplot(3,2,5),
zoom off
plot(out(:,1), out(:,2), 'b-', out(:,1), out(:,3), 'r--', out(:,1), out(:,4), 'k-
.')
```

```

legend('psi1 error','psi2 error','psi3 error', 4)
ylabel('attitude [rad]')
xlabel('time [sec]')
title('Helmet Filter Attitude error estimate vs. time')

```

```

%-----%
-----%

```

```

figure(2)
clf

```

```

subplot(3,2,5),
zoom off
plot(error(:,1),error(:,2),'b-',error(:,1),error(:,3),'r--',
error(:,1),error(:,4),'k-.')
hold on
grid on
legend('X error','Y error','Z error',1)
ylabel('position [ft]')
xlabel('time [sec]')
title('T38 Position error vs. time')

```

```

subplot(3,2,3),
zoom off
plot(error(:,1),error(:,8),'b-',error(:,1),error(:,9),'r--',
error(:,1),error(:,10),'k-.')
hold on
grid on
legend('Vx error','Vy error','Vz error',4)
ylabel('velocity [ft/sec]')
title('T38 Velocity error vs. time')

```

```

subplot(3,2,1),
zoom off
plot(error(:,1),error(:,14),'b-',error(:,1),error(:,15),'r--',
error(:,1),error(:,16),'k-.')
hold on
grid on
legend('psi1 error','psi2 error','psi3 error',4)
ylabel('attitude [rad]')

title('T38 Attitude error vs. time')

```

```

subplot(3,2,6),
zoom off
plot(error(:,1),error(:,5),'b-',error(:,1),error(:,6),'r--',
error(:,1),error(:,7),'k-.')
hold on
grid on
legend('X error','Y error','Z error',4)
ylabel('position [ft]')
xlabel('time [sec]')
title('Helmet Position error vs. time')

```

```

subplot(3,2,4),
zoom off
plot(error(:,1),error(:,11),'b-',error(:,1),error(:,12),'r--',
error(:,1),error(:,13),'k-.')

```

```

hold on
grid on
legend('Vx error','Vy error','Vz error',4)
ylabel('velocity [ft/sec]')
title('Helmet Velocity error vs. time')

```

```

subplot(3,2,2),
zoom off
plot(error(:,1),error(:,17),'b-',error(:,1),error(:,18),'r--',
error(:,1),error(:,19),'k-.')

```

```

hold on
grid on
legend('psi1 error','psi2 error','psi3 error',4)
ylabel('attitude [rad]')
title('Helmet Attitude error vs. time')

```

```

%-----%
-----%

```

```

figure(3)
clf

```

```

hold on
plot(P_out(:,1),P_out(:,17),'b--',P_out(:,1),-1.*P_out(:,17),'b--')
plot(P_out(:,1),P_out(:,18),'g--',P_out(:,1),-1.*P_out(:,18),'g--')
plot(P_out(:,1),P_out(:,19),'r--',P_out(:,1),-1.*P_out(:,19),'r--')

```

```

grid on
title('Helmet Attitude Covariance')
xlabel('time [sec]')
ylabel('covariance [rad]')

```



```
function [x_pre, P_pre] = propagate(PHI, x_post, P_post, Q)

% [x_pre, P_pre] = propagate(PHI, x_post, P_post, Q)
%
% This function propogates the state estimate and covariace matrix.
%
% Inputs:
% PHI = propagation matrix for state & state estimate
% x_post = either initial condition, or post-measurement-update state estimate
% P_post = either initial condition, or post-measurement-update covariance matrix
% Q = inherent noise matrix for state estimate vector (w in diagonal matrix)
%
% Outputs:
% x_pre = propogated state estimate prior to next update
% P_pre = propogated state estimate covariance matrix prior to next update
%
% Written by E. Bailey February 26, 2000.

x_pre = PHI * x_post;

P_pre = PHI * P_post * PHI' + Q;
```

```

function C = q2C(q)
% C = q2C(q)
%
% Quaternion to DCM

C = eye(3,3); % Initialize Matrix

C(1,1) = q(1)^2 + q(2)^2 - q(3)^2 - q(4)^2;
C(2,1) = 2*(q(2)*q(3)-q(1)*q(4));
C(3,1) = 2*(q(2)*q(4)+q(1)*q(3));
C(1,2) = 2*(q(2)*q(3)+q(1)*q(4));
C(2,2) = q(1)^2 - q(2)^2 + q(3)^2 - q(4)^2;
C(3,2) = 2*(q(3)*q(4)-q(1)*q(2));
C(1,3) = 2*(q(2)*q(4)-q(1)*q(3));
C(2,3) = 2*(q(3)*q(4)+q(1)*q(2));
C(3,3) = q(1)^2 - q(2)^2 - q(3)^2 + q(4)^2;

```

```

function qout=qmulq(q1,q2)
% qout=qmulq(q1,q2)
%
% Quaternion Multiplication

qout(1,:) = q1(1,:).*q2(1,:)-q1(2,:).*q2(2,:)-q1(3,:).*q2(3,:)-q1(4,:).*q2(4,:);
qout(2,:) = q1(1,:).*q2(2,:)+q1(2,:).*q2(1,:)+q1(3,:).*q2(4,:)-q1(4,:).*q2(3,:);
qout(3,:) = q1(1,:).*q2(3,:)-q1(2,:).*q2(4,:)+q1(3,:).*q2(1,:)+q1(4,:).*q2(2,:);
qout(4,:) = q1(1,:).*q2(4,:)+q1(2,:).*q2(3,:)-q1(3,:).*q2(2,:)+q1(4,:).*q2(1,:);

```

```

%-----%
%----- STAT_RUN.M -----%
%-----%

%-----%
% INIT VALUES %
%-----%

FLAGshow_all_errors = 0;

%-----%
% Statistical Calcs %
%-----%

psi_x(:,1) = att_error1(:,1);
psi_y(:,1) = att_error1(:,1);
psi_z(:,1) = att_error1(:,1);

for(i= 1:20)
    eval(['psi_x(:, ' num2str(i) '+1) = att_error' num2str(i) '(:,2);'])
    eval(['psi_y(:, ' num2str(i) '+1) = att_error' num2str(i) '(:,3);'])
    eval(['psi_z(:, ' num2str(i) '+1) = att_error' num2str(i) '(:,4);'])
end

psi_stat(:,1) = psi_x(:,1);

for (i = 1:length(psi_stat(:,1)))
    psi_stat(i,2) = mean(psi_x(i,2:end));
    psi_stat(i,3) = mean(psi_y(i,2:end));
    psi_stat(i,4) = mean(psi_z(i,2:end));
    psi_stat(i,5) = sqrt(var(psi_x(i,2:end)));
    psi_stat(i,6) = sqrt(var(psi_y(i,2:end)));
    psi_stat(i,7) = sqrt(var(psi_z(i,2:end)));
    psi_stat(i,8) = sqrt(sum(psi_x(i,2:end).^2)/20);
    psi_stat(i,9) = sqrt(sum(psi_y(i,2:end).^2)/20);
    psi_stat(i,10) = sqrt(sum(psi_z(i,2:end).^2)/20);
end

Psig_psix(:,1) = P_out1(:,1);
Psig_psiy(:,1) = P_out1(:,1);
Psig_psiz(:,1) = P_out1(:,1);

for (i = 1:20)
    eval(['Psig_psix(:, ' num2str(i) '+1) = P_out' num2str(i) '(:,17);'])
    eval(['Psig_psiy(:, ' num2str(i) '+1) = P_out' num2str(i) '(:,18);'])
    eval(['Psig_psiz(:, ' num2str(i) '+1) = P_out' num2str(i) '(:,19);'])
end

Psig_stat(:,1) = Psig_psix(:,1);

for (i = 1:length(psi_stat(:,1)))
    Psig_stat(i,2) = mean(Psig_psix(i,2:end));
    Psig_stat(i,3) = mean(Psig_psiy(i,2:end));
    Psig_stat(i,4) = mean(Psig_psiz(i,2:end));
end

%-----%
% PLOTTING ROUTINES %
%-----%

```

```

figure(1)
clf
subplot(3,2,1),
plot(psi_stat(:,1),psi_stat(:,2),'r')
hold on
plot(psi_stat(:,1),psi_stat(:,5),'--')
plot(psi_stat(:,1),-psi_stat(:,5),'--')
ylabel('x-axis [rad]')
grid on
axis_vals = [0,120,-
max([max(psi_stat(10:end,8)),max(psi_stat(10:end,9)),max(psi_stat(10:end,10))]),
...
max([max(psi_stat(10:end,8)),max(psi_stat(10:end,9)),max(psi_stat(10:end,10))])];
title('E[x] & Sigma of attitude error')
axis(axis_vals);

subplot(3,2,2),
plot(Psig_stat(:,1),Psig_stat(:,2))
hold on
plot(Psig_stat(:,1),-Psig_stat(:,2))
ylabel('psi_x [rad]')
title('Filter P-matrix sigmas')
grid on
axis(axis_vals);

subplot(3,2,3),
plot(psi_stat(:,1),psi_stat(:,3),'r')
hold on
plot(psi_stat(:,1),psi_stat(:,6),'--')
plot(psi_stat(:,1),-psi_stat(:,6),'--')
ylabel('y-axis [rad]')
grid on
axis(axis_vals);

subplot(3,2,4),
plot(Psig_stat(:,1),Psig_stat(:,3))
hold on
plot(Psig_stat(:,1),-Psig_stat(:,3))
ylabel('psi_y [rad]')
grid on
axis(axis_vals);

subplot(3,2,5),
plot(psi_stat(:,1),psi_stat(:,4),'r')
hold on
plot(psi_stat(:,1),psi_stat(:,7),'--')
plot(psi_stat(:,1),-psi_stat(:,7),'--')
ylabel('z-axis [rad]')
xlabel('time [sec]')
grid on
axis(axis_vals);

subplot(3,2,6),
plot(Psig_stat(:,1),Psig_stat(:,3))
hold on
plot(Psig_stat(:,1),-Psig_stat(:,3))
ylabel('psi_z [rad]')
xlabel('time [sec]')
grid on
axis(axis_vals);

```

```

figure(2)
clf
subplot(3,2,1),
%plot(psi_stat(:,1),psi_stat(:,2),'r')
hold on
plot(psi_stat(:,1),psi_stat(:,8),'k-.')
plot(psi_stat(:,1),-psi_stat(:,8),'k-.')
axis(axis_vals);
grid on;
ylabel('x-axis [rad]')
title('E[x] & RSS of attitude error')

subplot(3,2,3),
%plot(psi_stat(:,1),psi_stat(:,3),'r')
hold on
plot(psi_stat(:,1),psi_stat(:,9),'k-.')
plot(psi_stat(:,1),-psi_stat(:,9),'k-.')
axis(axis_vals);
grid on;
ylabel('y-axis [rad]')

subplot(3,2,5),
%plot(psi_stat(:,1),psi_stat(:,4),'r')
hold on
plot(psi_stat(:,1),psi_stat(:,10),'k-.')
plot(psi_stat(:,1),-psi_stat(:,10),'k-.')
axis(axis_vals);
grid on;
ylabel('z-axis [rad]')
xlabel('time [sec]')

subplot(3,2,2),
plot(Psig_stat(:,1),Psig_stat(:,2))
hold on
plot(Psig_stat(:,1),-Psig_stat(:,2))
ylabel('psi_x [rad]')
title('Filter P-matrix sigmas')
grid on
axis(axis_vals);

subplot(3,2,4),
plot(Psig_stat(:,1),Psig_stat(:,3))
hold on
plot(Psig_stat(:,1),-Psig_stat(:,3))
ylabel('psi_y [rad]')
grid on
axis(axis_vals);

subplot(3,2,6),
plot(Psig_stat(:,1),Psig_stat(:,3))
hold on
plot(Psig_stat(:,1),-Psig_stat(:,3))
ylabel('psi_z [rad]')
xlabel('time [sec]')
grid on
axis(axis_vals);

if(FLAGshow_all_errors)

```

```

figure(3)
clf
for(i = 1:5)
axis_vals = [0,psi_x(end,1), ...
             min([min(psi_x(:,i+1)), min(psi_y(:,i+1)), min(psi_z(:,i+1))]), ...
             max([max(psi_x(:,i+1)), max(psi_y(:,i+1)), max(psi_z(:,i+1))])]);
subplot(5,3,3*(i-1)+1),
plot(psi_x(:,1),psi_x(:,i+1))
hold on
grid on
axis(axis_vals);
if(i == 1)
title('x-axis psi-error');
elseif(i == 5)
xlabel('time [sec]');
end
ylabel('error [rad]')
subplot(5,3,3*(i-1)+2),
plot(psi_y(:,1),psi_y(:,i+1))
hold on
grid on
axis(axis_vals);
if(i == 1)
title('y-axis psi-error');
elseif(i == 5)
xlabel('time [sec]');
end
subplot(5,3,3*(i-1)+3),
plot(psi_z(:,1),psi_z(:,i+1))
hold on
grid on
axis(axis_vals);
if(i == 1)
title('z-axis psi-error');
elseif(i == 5)
xlabel('time [sec]');
end
end

figure(4)
clf
for(i = 6:10)
axis_vals = [0,psi_x(end,1), ...
             min([min(psi_x(:,i+1)), min(psi_y(:,i+1)), min(psi_z(:,i+1))]), ...
             max([max(psi_x(:,i+1)), max(psi_y(:,i+1)), max(psi_z(:,i+1))])]);
subplot(5,3,3*(i-6)+1),
plot(psi_x(:,1),psi_x(:,i+1))
hold on
grid on
axis(axis_vals);
if(i == 6)
title('x-axis psi-error');
elseif(i == 10)
xlabel('time [sec]');
end
ylabel('error [rad]');
subplot(5,3,3*(i-6)+2),
plot(psi_y(:,1),psi_y(:,i+1))
hold on

```

```

grid on
axis(axis_vals);
if(i == 6)
    title('y-axis psi-error');
elseif(i == 10)
    xlabel('time [sec]');
end
subplot(5,3,3*(i-6)+3),
    plot(psi_z(:,1),psi_z(:,i+1))
    hold on
    grid on
    axis(axis_vals);
if(i == 6)
    title('z-axis psi-error');
elseif(i == 10)
    xlabel('time [sec]');
end
end

figure(5)
clf
for(i = 11:15)
    axis_vals = [0,psi_x(end,1), ...
        min([min(psi_x(:,i+1)), min(psi_y(:,i+1)), min(psi_z(:,i+1))]), ...
        max([max(psi_x(:,i+1)), max(psi_y(:,i+1)), max(psi_z(:,i+1))])]);
    subplot(5,3,3*(i-11)+1),
        plot(psi_x(:,1),psi_x(:,i+1))
        hold on
        grid on
        axis(axis_vals);
    if(i == 11)
        title('x-axis psi-error');
    elseif(i == 15)
        xlabel('time [sec]');
    end
    ylabel('error [rad]');
    subplot(5,3,3*(i-11)+2),
        plot(psi_y(:,1),psi_y(:,i+1))
        hold on
        grid on
        axis(axis_vals);
    if(i == 11)
        title('y-axis psi-error');
    elseif(i == 15)
        xlabel('time [sec]');
    end
    subplot(5,3,3*(i-11)+3),
        plot(psi_z(:,1),psi_z(:,i+1))
        hold on
        grid on
        axis(axis_vals);
    if(i == 11)
        title('z-axis psi-error');
    elseif(i == 15)
        xlabel('time [sec]');
    end
end

figure(6)
clf

```

```

for(i = 16:20)
    axis_vals = [0,psi_x(end,1), ...
        min([min(psi_x(:,i+1)), min(psi_y(:,i+1)), min(psi_z(:,i+1))]), ...
        max([max(psi_x(:,i+1)), max(psi_y(:,i+1)), max(psi_z(:,i+1))])]);
    subplot(5,3,3*(i-16)+1),
        plot(psi_x(:,1),psi_x(:,i+1))
        hold on
        grid on
        axis(axis_vals);
    if(i == 16)
        title('x-axis psi-error');
    elseif(i == 20)
        xlabel('time [sec]');
    end
    ylabel('error [rad]');
    subplot(5,3,3*(i-16)+2),
        plot(psi_y(:,1),psi_y(:,i+1))
        hold on
        grid on
        axis(axis_vals);
    if(i == 16)
        title('y-axis psi-error');
    elseif(i == 20)
        xlabel('time [sec]');
    end
    subplot(5,3,3*(i-16)+3),
        plot(psi_z(:,1),psi_z(:,i+1))
        hold on
        grid on
        axis(axis_vals);
    if(i == 16)
        title('z-axis psi-error');
    elseif(i == 20)
        xlabel('time [sec]');
    end
end
end

```

```

function [x_post, P_post] = update(x_pre, P_pre, K, H, R, z);
% [x_post, P_post] = update(x_pre, P_pre, K, H, z);
%
% Performs the Kalman Update using the gain matrix calculated
% prior to calling this function using calc_K.m
%
% Inputs:
% H = measurement matrix for update
% z = measurement for update
% x_pre = state estimate prior to update
% P_pre = state estimate covariance matrix prior to update
%
% Outputs:
% x_post = state estimate after measurement update
% P_post = state estimate covariance matrix after update
%
% Written by E. Bailey, February 26, 2000

I = eye(length(x_pre));

z_mHxhat = (z - H*x_pre);
x_post = x_pre + K * z_mHxhat;

%-- K-varying code --%
%
% x_post_2 = z - H*x_post;
%
% if (abs(x_post_2(7)) > 3)
% K(:,7) = K(:,7)*2;
% end
%
% if (abs(x_post_2(8)) > 3)
% K(:,8) = K(:,8)*2;
% end
%
% if (abs(x_post_2(9)) > 3)
% K(:,9) = K(:,9)*2;
% end
%
% x_post = x_pre + K * z_mHxhat;

ImKH = (I-K*H);

P_post = ImKH * P_pre * ImKH' + K*R*K';

```