Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

1996-03

Discrete asynchronous Kalman filtering of navigation data for the Phoenix autonomous underwater vehicle

McClarin, David W.

Monterey, California. Naval Postgraduate School

https://hdl.handle.net/10945/32182

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

DISCRETE ASYNCHRONOUS KALMAN FILTERING OF NAVIGATION DATA FOR THE PHOENIX AUTONOMOUS UNDERWATER VEHICLE

by

David W. McClarin

March 1996

Thesis Advisor:	Robert McGhee
Co-Advisor:	Anthony Healey

Approved for public release; distribution is unlimited.

19960620 115

DTIC QUALITY INSPECTED 1

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 1996	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE DISCRETE ASYNCHRONOUS KALMAN FILTERING OF NAVIGATION DATA FOR THE PHOENIX AUTONOMOUS UNDERWATER VEHICLE	5. FUNDING NUMBERS
6. AUTHOR(S) David W. McClarin	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

The Phoenix Autonomous Underwater Vehicle must be able to accurately determine its position at all times. This requires: 1) GPS and differential GPS for surface navigation, 2) short baseline sonar ranging system for submerged navigation, and 3) mathematical modeling of position.

This thesis describes a method of Kalman filtering to merge the GPS, differential GPS, short baseline sonar ranging, and the mathematical model to produce a single state vector of vehicle position and ocean currents. The filter operates in the extended mode for processing the non-linear sonar ranges, and in normal mode for the linear GPS/DGPS data. This required installation of a GPS system and the determination of the different variances and errors between these systems.

Phoenix now has a real time method of position determination using either position measuring system separately or combined. The results of this work have been validated by real world testing of the vehicle at sea, where position estimates accurate to within several meters were obtained.

14. SUBJECT TERMS NAVIGATION, KALMAN-FILTERING, AUTONOMOUS UNDERWATER VEHICLES			15. NUMBER OF PAGES 140
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

DISCRETE ASYNCHRONOUS KALMAN FILTERING OF NAVIGATION DATA FOR THE PHOENIX AUTONOMOUS UNDERWATER VEHICLE

David W. McClarin
Lieutenant, United States Navy
B.S., University of Florida, 1989

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1996

Author:

David W. McClarin

Approved by:

Robert McGhee, Thesis Advisor

Anthony Healey, Co-Advisor

Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The Phoenix Autonomous Underwater Vehicle must be able to accurately determine its position at all times. This requires: 1) GPS and differential GPS for surface navigation, 2) short baseline sonar ranging system for submerged navigation, and 3) mathematical modeling of position.

This thesis describes a method of Kalman filtering to merge the GPS, differential GPS, short baseline sonar ranging, and the mathematical model to produce a single state vector of vehicle position and ocean currents. The filter operates in the extended mode for processing the non-linear sonar ranges, and in normal mode for the linear GPS/DGPS data. This required installation of a GPS system and the determination of the different variances and errors between these systems.

Phoenix now has a real time method of position determination using either position measuring system separately or combined. The results of this work have been validated by real world testing of the vehicle at sea, where position estimates accurate to within several meters were obtained.

TABLE OF CONTENTS

I. INTRODUCTION 1
- A. BACKGROUND 1
- B. THE PHOENIX AUV 2
  - 1. Strategic Level 2
  - 2. Tactical Level 2
  - 3. Execution Level 4
- C. NAVIGATION MODULE 4
  - 1. Navigator1.C 4
  - 2. Kalman_filter.C 5
  - 3. Readgps.C 5
  - 4. Matrix.C 5
- D. THESIS CHAPTER SUMMARY 5

II. PHOENIX HARDWARE CHARACTERISTICS AND SHORTFALLS 7
- A. INTRODUCTION 7
- B. PHOENIX AUV HARDWARE OVERVIEW 7
- C. VOYAGER LAPTOP WORKSTATION 9
- D. MOTOROLA GPS/DGPS UNIT 9
- E. DIVETRACKER SYSTEM 10
- F. SUMMARY 11

III. KALMAN FILTERING 13
- A. INTRODUCTION 13
- B. PHOENIX IMPLEMENTATION 13
  - 1. Statistical Background 14
  - 2. Movement Model 14
- C. KALMAN FILTER FORMULAS 15
  - 1. Motion and Measurement Models 15
  - 2. Movement Step 17
  - 3. Measurement Step 17
- D. DIMENSIONLESS SHOCK 18
- E. EXTENDED KALMAN FILTERING 19
- F. SPEED/CURRENT ERROR MODEL 20
- G. SUMMARY 22

IV. NAVIGATION 23
- A. INTRODUCTION 23
- B. NAVIGATION OVERVIEW 23
- C. NAVIGATION CO-ORDINATES 24
- D. GPS/DGPS 25
  - 1. Phoenix GPS/DGPS Variances 26

2. Kalman Filtering of GPS/DGPS Data 29
3. GPS/DGPS Navigation 36
E. DIVETRACKER RANGE UTILIZATION 36
1. Divetracker Variance 37
2. Baseline Problem 37
F. FILTER RESPONSE VS VEHICLE STABILITY 38
G. FIX DETERMINATION 41
H. FIX POSITION TRANSLATION TO VEHICLE CENTER 42
I. NAVIGATION INITIALIZATION 43
J. OCEAN CURRENT (ERROR) ESTIMATION 43
K. WATER SPEED SENSOR CALIBRATION 44
L. SIMULATION MODE 44
M. SUMMARY 46

V. SOFTWARE 47
A. INTRODUCTION 47
B. NAVIGATOR1.C 47
1. Navigation Module Operation 47
2. Nav1_Initialize Function 50
3. My_Parse_Telemetry_String Function 50
4. Reset_Kalman Function 50
C. KALMAN_FILTER.C 50
1. Kalman_Filter Operation 51
2. Navtorad Function 52
3. Mysquare Function 52
D. READGPS.C 52
1. Get_GPS_Data Function 53
2. CheckSumCheck Function 53
3. Getmilsec Function 53
4. Getgpstime Function 54
5. Getgpsfixtype Function 54
6. Determine Fix Function 54
7. Gps_Serial_Read Function 55
8. Initialize_Serial Function 55
9. Open_Tty Function 55
10. Tty and Serial_Read Timeout Functions 56
11. Simulate_GPS_Data Function 56
E. MATRIX.C 56
1. Matrix_Multiply Function 57
2. Matrix_Add and Subtract Functions 57
3. Matrix_Transpose Function 57
4. Matrix_Inverse Function 57
5. Gauss_Elimination Function 57

6. Matrix_Rtransform Function 58
7. Output_Matrix Function 58
F. SUMMARY 58

VI. SUMMARY AND CONCLUSIONS 59
A. SUMMARY 59
B. FUTURE WORK 60
C. CONCLUSION 61

APPENDIX A. NAVIGATOR1.C 63

APPENDIX B. KALMAN_FILTER.H 81

APPENDIX C. KALMAN_FILTER.C 85

APPENDIX D. READGPS.H 93

APPENDIX E. READGPS.C 95

APPENDIX F. MATRIX.H 111

APPENDIX G. MATRIX.C 113

LIST OF REFERENCES 121

INITIAL DISTRIBUTION LIST 125

ACKNOWLEDGMENTS

During the course of my thesis work, there were many people who were instrumental in helping me. Without their guidance, help and patience, I would have never been able to accomplish the work of this thesis. I would like to take this opportunity to acknowledge some of them.

I would like to thank my thesis advisors, Professor Robert McGhee and Professor Anthony Healey, both of whom were driving factors to the successful completion of this phase of Phoenix development. Professor Alan Washburn introduced me to Kalman Filtering. A small MATLAB project required for his class served as the kernel of my entire project.

My fellow members of the Phoenix software team were Brad Leonhardt, Mike Campbell, Mike Burns and Duane Davis. Without their support, hard work and dedication none of this work would have been possible. A special thanks to Russ Whalen for efforts and support in this project.

I must give immense thanks to my wife Ruth and our children Lori, Jackie, Mark and Christopher. Their love and support during long nights of work away at the lab was of immeasurable value to me.

This research was supported in part by Grant BCS-9306252 from the National Science Foundation to the Naval Postgraduate School.

I. INTRODUCTION

For any vehicle to be truly autonomous requires that it have knowledge of its local world coordinate position. This thesis describes a method of discrete Kalman Filtering of short baseline sonar range data (DiveTracker) and satellite navigation data (GPS) to achieve accurate positioning of the NPS Phoenix AUV [MARC96].

A. BACKGROUND

An inherent difficulty in any precision navigation system is the accuracy of the measurements. No measurement system is perfect, just the amount of error in the system varies. Kalman Filtering is a method of filtering measurement data based on the known or approximated variance of the measurements and vehicle movements. [GELB88]

Previous and continuing related work in this area includes the Shallow-Water AUV Navigation System (SANS) [MCGH95],[BACH96]. SANS utilizes a twelve state continuous Kalman (complementary) filter of inertial measurement unit (IMU) data with differential GPS updating. SANS provides highly accurate dead reckoning utilizing IMU data. The SANS position is updated using raw DGPS data as the "Truth". SANS has no method of position updating other than dead reckoning when submerged, and only takes GPS measurements when surfaced. This system was used as a background for the work of this thesis.

Phoenix presently does not have an IMU, so all dead reckoning is performed using speeds developed via mass motion formulas, a vertical and heading gyro, and a water wheel speed measuring unit [MARC96]. Phoenix also has the DiveTracker system [FLAG94] which allows position measurements while submerged, and GPS for measurements while

surfaced. The work of this thesis utilizes filtered GPS and DiveTracker ranges for updating dead reckoned positions, versus using raw data as SANS does.

B. THE PHOENIX AUV

The Phoenix autonomous underwater vehicle is a shallow-water mine warfare test bed prototype (Figure 1). The vehicle is designed to act autonomously in searching for mine-like objects and accurately reporting their positions. This requires a complex software suite with a highly accurate method of navigation. The Phoenix runs on a unique three level software architecture, consisting of strategic, tactical and execution levels called the "Rational behavior Model" [BYRN96]. These levels are based on proven methods of actual U. S. Submarine control [HOLD95].

1. Strategic Level

The strategic level acts as the vehicle's Commanding Officer. This level holds the mission logic and controls the mission by giving orders to the tactical level. The strategic level only gives commands and awaits reports that the commands are accepted or completed. The tactical level responds with either a command accepted, command complete, or command aborted message. The strategic level then takes actions depending upon the command report. This level was written in Prolog, and treats the tactical level as a function call [MARC96],[LEON96].

2. Tactical Level

The tactical level acts as the vehicle's Officer of the Deck (OOD). It receives orders from the strategic level and takes the actions required to compete these actions, if possible. The tactical level OOD runs in parallel with the Sonar [CAMP96] and Navigation sub-levels,

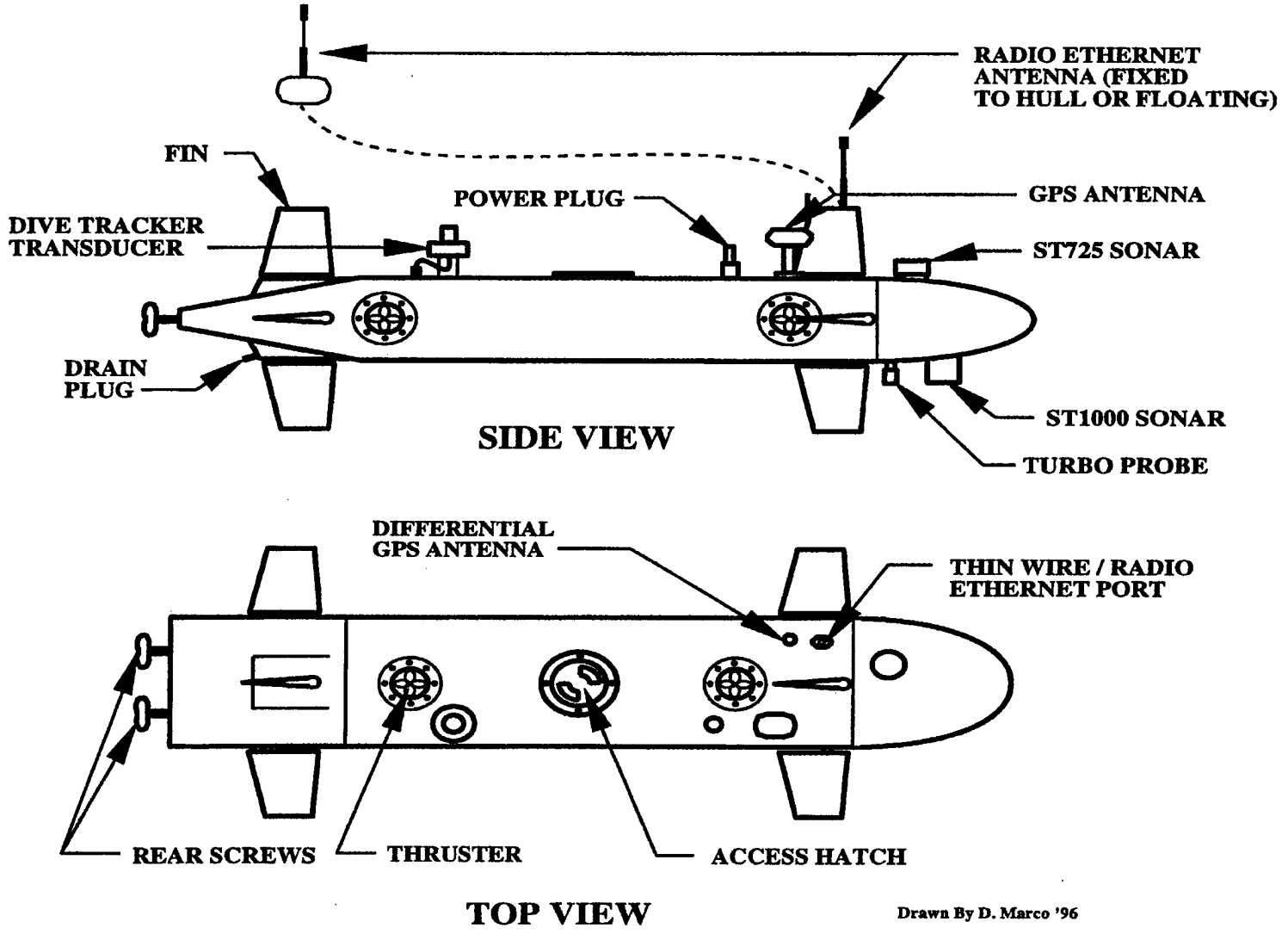

Figure 1: Phoenix AUV

and gives vehicle control commands to the execution level. Sonar and Navigation report directly to the tactical level OOD. The tactical level uses the sonar inputs to determine if an object has been encountered, and the navigation inputs to update the execution level's estimate of the vehicle's position. [LEON96]

3. Execution Level

The execution level acts as the ships crew; ie., it drives the vehicle from point to point, controls all control surfaces, and takes emergency actions [BYRN96]. The execution level can hover at a given point, maintain ordered depth, and take all actions required to conn the vehicle from point to point. The execution level communicates with the tactical level, updating vehicle parameters and receiving new orders and vehicle positions [BURN96].

C. NAVIGATION MODULE

The navigation module utilizes both discrete normal and extended Kalman Filtering of measured GPS/DGPS, or short baseline sonar ranges (DiveTracker System), to produce the best estimate of the vehicle's position. This level consists of four main functions: Navigator1.C, Kalman_Filter.C, ReadGps.C, and Matrix.C.

1. Navigator1.C

Navigator1.C is the driver of the navigation module. This section of code communicates with the tactical level via piped communications. It receives basic initialization information, and subsequent updated vehicle parameters, and returns the best estimate of the vehicles current position and N/S, E/W (X,Y) current estimations. It calls the Kalman filter routine to return the updated position estimate. This process also records to data for later analysis.

2. Kalman_filter.C

This code performs "dead-reckoning" (movement step) and filters the input navigation data (measurement step) to create an updated vehicle position estimate. It filters either linear data (GPS/DGPS) as a normal filter, or non-linear data (DiveTracker) as an extended filter. It also develops a combined estimate of "Ocean Currents/Errors" and determines if the filter has possibly lost track or has a bad measurement.

3. Readgps.C

This code reads the data from the Motorola GPS/DGPS receiver. It opens the Solaris serial port for communications with the GPS unit and then decodes the GPS binary data. It also has the routine that determines the best type of fix information to use based on input data.

4. Matrix.C

This code performs the basic matrix operations required by the Kalman filter to include addition, subtraction, and multiplication. It also computes a matrix inverse using Gausian elimination and constructs the rotation matrices required for body speed transformation to earth coordinates.

D. THESIS CHAPTER SUMMARY

Chapter II overviews the Phoenix, GPS and Dive-Tracker hardware. Chapter III provides an in-depth description of Kalman filtering, describing this implementation and variance determination. Chapter IV describes the navigation problem and its solutions. Chapter V covers pertinent factors of the developed software. Chapter VI summarizes the conclusions and results of this work and discusses possible future work to be performed.

II. PHOENIX HARDWARE CHARACTERISTICS AND SHORTFALLS

A. INTRODUCTION

The Phoenix AUV possesses the precise position control and sensing systems hardware required for mine hunting and localization. To achieve this capability requires complex multiple computer capability, a sonar system, navigation equipment, and the necessary position and control surface motors and controllers. The three major pieces of hardware used in the implementation of the navigation module are the on-board Solaris Voyager laptop workstation, the DiveTracker system, and a Motorola GPS/DGPS unit.

B. PHOENIX AUV HARDWARE OVERVIEW

The Phoenix AUV hardware layout is shown in Figure 2. The vehicle mission logic operates on an installed Solaris (SUN) Voyager laptop work station. The vehicle control systems operate on a GESPAC M68030 processor operating under an OS-9 system [MARC96]. These computers operate together over a LAN. Phoenix has two screws for forward propulsion, two vertical thrusters for depth control, two horizontal thrusters for station keeping, and eight control fins for vehicle attitude control during forward motion. To provide environmental data, the Phoenix has ST725 and ST1000 TRITECH sonars [TRITEC]. Phoenix uses a depth cell and turbine flow meter for water depth and speed determinations respectively. The Voyager has its own independent battery supply with a life of 1 ½ hours. All other vehicle power is supplied by four lead acid batteries, which provide a vehicle life of approximately four hours.

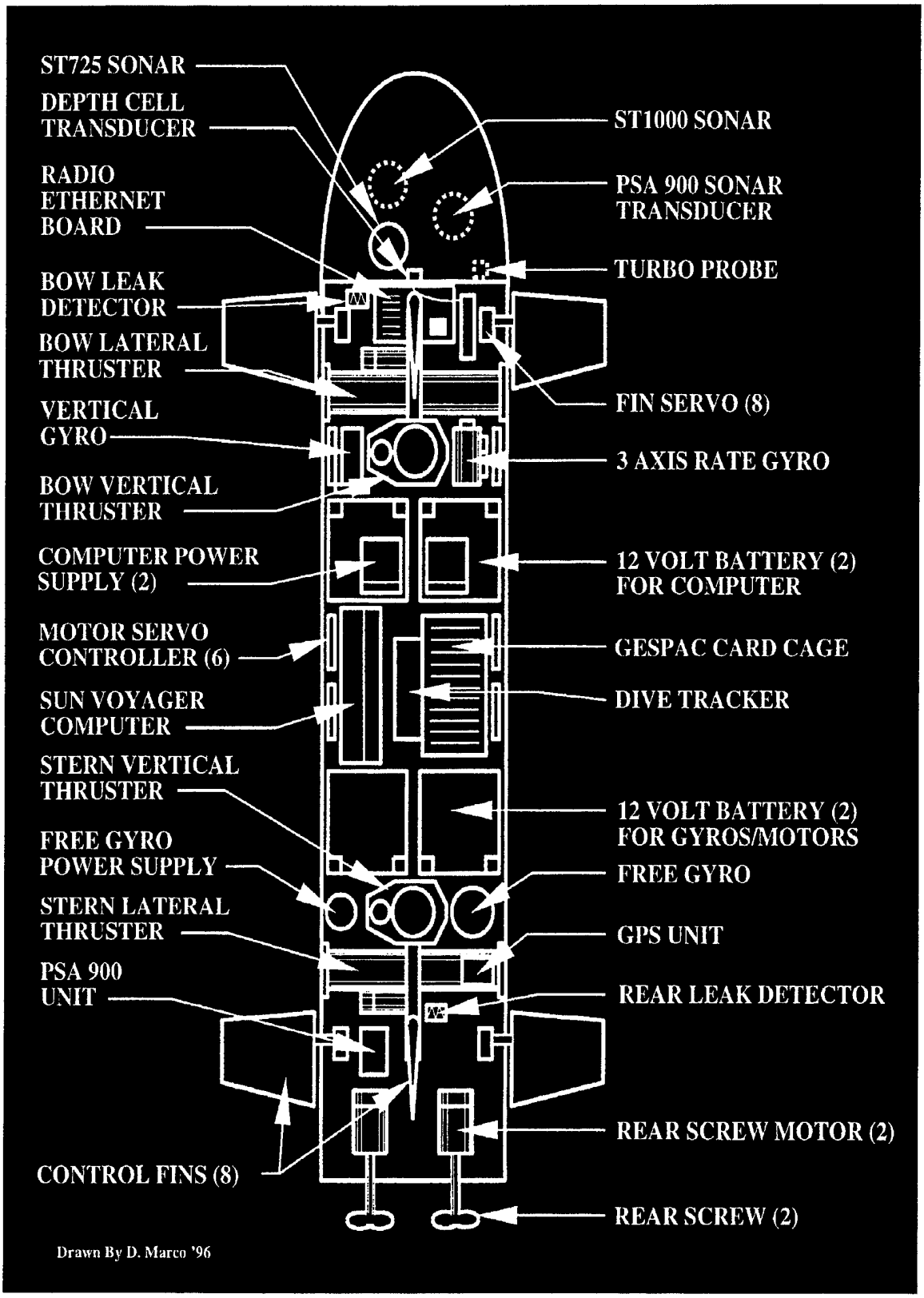

Figure 2: Phoenix Hardware LayOut

C. VOYAGER LAPTOP WORKSTATION

The voyager Solaris laptop workstation is the software host for the strategic and tactical levels. This is a new generation Solaris (SUN) workstation [SUN]. It has a 100 MHZ processor with 48 meg RAM and a 1.2 Gigabyte hard drive. It operates under the UNIX operating system.

The major shortfalls of this system was the poor battery capability. Trials of only short periods (a max of 1.5 hour) could be performed, with a two hour recharge rate. The Voyager had no real on/off switch. On/Off switching was keyboard controlled and the keyboard could not be installed due to space considerations in the vehicle. In addition, the battery was designed to be inserted into the unit, and it could not be removed once the VOYAGER was in the craft. The overall result was that upon a system lockup, there was no alternative to waiting until the battery died to shut down the system (overnight). To correct these problems the installed battery was removed and another battery was added in parallel to extend Voyager useful life. The new battery system is now wired directly into the Voyager with an on/off switch added. The SOLARIS system has the capability of serial port communication with other non-terminal devices. However, the operating system documentation did not include the required commands. After much trial and error, these commands were found in a non-SOLARIS source [SCSI].

D. MOTOROLA GPS/DGPS UNIT

The Motorola Eight Channel PVT8 GPS receiver is capable of both GPS and Differential GPS (DGPS) modes with a maximum speed of one fix per second [MOTORO]. This system is capable of simultaneously tracking up to eight satellites. The receiver output

data can be used in one of three formats: the MOTOROLA Binary Format, the National Marine Electronics Association (NMEA)-0183 Format, or the LORAN Emulation Format [MOTORO]. The GPS antenna is mounted on an four inch pedestal on the forward starboard side of the vehicle. The Differential antenna was eight inches long and mounted opposite on the port side. This system communicated to the VOYAGER serial port via a SCSI interface.

The main shortfall of this system was in its use of the Differential correction signal. In the absence of a new DGPS time correction, the receiver held the last DGPS time correction signal received for ninety seconds before changing fix status to standard (uncorrected) GPS. The Kalman filter requires that the variances of the system be known. The DGPS variance was approximately 45 ft^2 with uncorrected GPS variance being approximately 27900 ft^2. However, after 30 seconds of no DGPS signal, the DGPS variance increased to 207 ft^2 and after 60 seconds it grew to 17424 ft^2, while still reporting a DGPS fix [MOTORO]. The commands to modify this hold time to five seconds did not appear to work.

E. DIVETRACKER SYSTEM

The DiveTracker system is produced for divers' use by providing navigation and communications support [FLAG94]. On the Phoenix AUV, the DiveTracker hardware utilizes two base station sonar transducers combined with an onboard processor and transducer to provide independent ranges from each base station to the Phoenix. These ranges are processed by the Execution level and sent to the Navigation Module via the Tactical level. DiveTracker range standard deviation was determined to be approximately

six to eight inches by the manufacturer. Experimental data backed up this claim, but showed an occasional error of one to two feet. The baseline separation was a primary factor in determining the minimum and maximum useable navigation ranges. Ranges were reported at an approximate interval of 1 to 3 seconds.

This system worked only when the vehicle transducer was submerged. Unfortunately the Phoenix transducer was mounted on top of the vehicle. This caused a loss of DiveTracker data while surfaced or gaining a GPS/DGPS fix. After a subsequent submergence the system did not always restart. The system also had the problem of shadow zones where the transducer did not receive any data at all. To correct these problems the vehicle transducer has been mounted under the vehicle. Testing is in progress to determine the effectiveness of this solution.

F. SUMMARY

The Phoenix hardware configuration is highly complex and uses nearly all of the available space in the vehicle. There are still some hardware problems to work out and it seems that whenever the boat is opened a new problem develops. However, the overall hardware suite has proven to be very successful in meeting the requirements of supporting student thesis research and developing basic knowledge about the use of AUVs in mine hunting applications [BRUT96].

III. KALMAN FILTERING

A. INTRODUCTION

Kalman filtering is a method of recursively updating an estimate of a system state by processing a succession of measurements. The Kalman filter is model-based; each cycle of measured input data is compared with prior (model-based) estimates and are weighted by Kalman gains to obtain updated (output) state estimates. Kalman gains are computed during each cycle and are function's of the filter's covariances and models of the measurement process [GELB88]. In this chapter Kalman filtering will be discussed as implemented in the Phoenix AUV for navigation calculations.

B. PHOENIX IMPLEMENTATION

A discrete asynchronous Kalman Filter was used by the Phoenix navigation module. The use of DiveTracker range data required the addition of an Extended Kalman Filter mode of operation due to the non-linearity of range measurements. The Kalman filter used a non-zero mean movement model, where the input vehicle speed is assumed truth, and results in the filter solving for both an updated position data and estimates of ocean current. This filter also computes a Dimensionless shock quantity based on the received measurements to determine if the filter has possibly lost track or received bad measurements. The state vector ,U, was defined to be [Xpos Ypos Xdrift Ydrift]. The state was processed through the movement and measurement steps based on the previous position, measurements, Kalman gains, and system covariance.

1. Statistical Background

The Kalman computations are manipulations of (multi-variate) normal probability distributions [WASH94]. The computations are conducted in two separate stages consisting of motion and measurement step calculations. The symbol X represents a system state component and is a multi-variate normal with a mean of μ and a covariance of Σ, abbreviated as $X \sim N(\mu,\Sigma)$. V is the measurement noise, and is also a multi-variate normal with a mean of Uv and a variance of R abbreviated $V \sim N(Uv,R)$. W is the movement noise. It too is a multi-variate normal with a mean of Uw and a variance of Q, abbreviated $W \sim N(Uw,Q)$.

2. Movement Model

The movement model's X and Y position is based on standard dead-reckoning; i.e.,

$$Distance = Rate * Time \tag{3.1}$$

That is, Distance becomes the new X or Y position. Rates are computed using a rotational transform [CRAI86] of Phoenix u (longitudinal), v (sway) and w (heave) speeds to arrive with X (north/south), Y (east/west) and Z (up/down) speeds. The earth coordinates were set according to a right hand rule with north, east and down directions being positive. The movement model dead reckons in X and Y positions over a time Δ based on the following equations.

$$X_{i+1} = X_i + Xdrift^* \Delta + W_x \sim N(Xspeed^* \Delta, Q) \tag{3.2}$$

$$Y_{i+1} = Y_i + Ydrift^* \Delta + W_y \sim N(Yspeed^* \Delta, Q) \quad \textbf{(3.3)}$$

$$Xdrift_{i+1} = Xdrift_i + W_{Xdrift} \sim N(0, Q) \quad \textbf{(3.4)}$$

$$Ydrift_{i+1} = Ydrift_i + W_{Ydrift} \sim N(0, Q) \quad \textbf{(3.5)}$$

That is, The new X and Y positions are the sum of the old position, the distance covered by drift speeds, and an approximately normal non-zero mean random variable W, where W has a mean of Speed*Δ and a variance Q. The use of a non-zero mean random variable for the calculations of the X and Y positions is the primary driver for the solution of X and Y drift speeds. The X and Y drift calculations use a zero mean random variable W, with variance Q.

C. KALMAN FILTER FORMULAS

The Kalman filter uses Equations (3.6) and (3.7) for the motion modeling described by Equations (3.2-3.5). Equations (3.8-3.11) are used in the calculation of the measurement step. All operations are matrix operations. With the addition of Φ and H as movement and measurement matrices, Equations 3.2 and 3.5 are transformed to the Kalman filter formulas. For example if $X_{i+1} = \Phi X_i + W_x$ (simplification of Eq. 3.2) and $X \sim N(\mu, \Sigma)$, then, $\mu_{i+1} = \Phi\mu_i + Uw$ as demonstrated in Equation (3.6).

1. Motion and Measurement Models

The motion formulas are:

$$U(-)_{i+1} = \Phi_i^* U(+)_i + Uw_i \quad \textbf{(3.6)}$$

$$\Sigma(-)_{i+1} = \Phi_i * \Sigma(+)_i * \Phi^T{}_i + Q_i \quad \textbf{(3.7)}$$

The measurement and update formulas are:

$$K_{i+1} = \Sigma(-)_{i+1} * H_{i+1} / (H_{i+1} * \Sigma(-)_{i+1} * H_{i+1}{}^T + R) \quad \textbf{(3.8)}$$

$$U(+)_{i+1} = U(-)_{i+1} + K_{i+1} * Shock_{i+1} \quad \textbf{(3.9)}$$

$$Shock_{i+1} = Z_{i+1} - Uv - H_{i+1} * U(-)_{i+1} \quad \textbf{(3.10)}$$

$$\Sigma(+)_{i+1} = (I - K_{i+1} * H_{i+1}) * \Sigma(-)_i * (I - K_{i+1} * H_{i+1})^T \quad \textbf{(3.11)}$$

where:

U,Σ = The mean and covariance of the System State.

Φ = The movement Matrix, which describes how the state changes.

Uw,Q = The mean and covariance of the movement noise.

H = The measurement matrix (how the measurement depends on the state).

Uv,R = The mean and covariance of the measurement noise.

Z = The measurements (GPS/DGPS or DiveTracker).

K = Kalman Gains (a ratio of the filter Covariances)

I = Identity Matrix

and '+' indicates a measurement step while '-' indicates a movement step calculation.

2. Movement Step

The new movement step position given by Equation (3.6) is the sum of the product of the movement matrix Φ and state vector U(+), as shown in Equation (3.12).

$$NewPosition = \begin{bmatrix} 1 & 0 & \Delta & 0 \\ 0 & 1 & 0 & \Delta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Xdft \\ Ydft \end{bmatrix} \tag{3.12}$$

The addition of Uw results in Equation (3.6). The new value of Σ given by Equation (3.7) also depends on the movement matrix Φ and the addition of the covariance of the movement noise, and results in a new covariance matrix for the system state U.

3. Measurement Step

The measurement step computes a new state vector U based upon measurements and Kalman gains. Kalman gains given by Equation (3.8) are computed as a ratio of the state covariance, as it depends upon the measurement vector and the sum of the state covariance and the measurement Equation (3.11). The gains indicate how much the state vector U values depend upon the measurements Z1 and Z2. Specifically,

$$K = \begin{bmatrix} [X\ Z1gain & X\ Z2gain \\ Y\ Z1gain & Y\ Z2gain \\ Xdft\ Z1gain & Xdft\ Z2gain \\ Ydft\ Z1gain & Ydft\ Z2gain \end{bmatrix} \tag{3.13}$$

The computed gains are used as weights on the amount of change in the system based on the measurements. The difference between the estimated position based on the movement model

and the measured position Z is denoted as "Shock" [WASH94], or as equivalently to as "innovation" and is given by Equation (3.9). Where a measured position is from GPS/DGPS or is a position derived from DiveTracker ranges. The new system state U is a sum of the previous state and a gain weighted shock given by Equation (3.10). The new covariance, Equation (3.11), is the product of the "complement" of the state dependent gain (a measure of truth) and the old covariance. The complement is derived by subtracting the state dependent gain from an Identity matrix.

D. DIMENSIONLESS SHOCK

In a perfect system, the value of the shock would be zero. As the shock increases and becomes large, then the probability that the system has lost track also increases. A problem develops in determining what value of shock should be considered "large". Dimensionless shock (Eq. 3.14) is used to determine what value of shock relates to "large".

$$DimensionlessShock = Shock^{T} * (H * \Sigma(-) + R)^{-1} * Shock \qquad \textbf{(3.14)}$$

A large value of DimensionlessShock indicates a possible measurement problem or that the filter has lost track. DimensionlessShock can be gauged against the degrees of freedom of the shock [WASH94]. However, it has been found in the research of this thesis that an order of magnitude increase over the degrees of freedom provides better results.

An order of magnitude increase was determined to be required due to the shift in measurement methods. When using a consistent measurement method, a large shift in the DimensionlessShock value as gauged against the degrees of freedom of the shock does

indicated a possible loss of track. However, when shifting measurement methods it is possible to get a change in position that results in a higher value than expected of DimensionlessShock. To ensure that the new measurement is not ignored, an order of magnitude increase in the DimensionlessShock threshold level is used. This enables the filter to use the new measurement and maintain track.

E. EXTENDED KALMAN FILTERING

In the previous discussion of the Kalman Filter, the measurement was always a linear function of the system state. In the non-linear case, the relationship between the system state and the measurements must be linearized. In the Phoenix Kalman filter, the DiveTracker ranges are a non-linear function of the state. The DiveTracker ranges are two independent ranges from base station transducers to the Phoenix. In this case a non-linear filter (Extended Kalman Filter) must be used [WASH94]. This linearization is performed by taking the derivative of a calculated range, f(U), given by Equation (3.15). Where f(U) is a function of the X and Y components of the system state vector U. If Dx and Dy are distances between the Phoenix state position U and the DiveTracker base transponder positions, then

$$f(U) = CalcRange = \sqrt{((Dx)^2+(Dy)^2)} \qquad \textbf{(3.15)}$$

Since the values of the measurements are non-linear with respect to the state, the development of a new H (how the measurement depends upon the state) matrix is required. This new H (Equation 3.16) is now composed of the first partial derivatives of calculated measurements f(U), based upon the current values of the state, to form a Jacobian.

$$H = \begin{bmatrix} \partial Range1/\partial x & \partial Rang1/\partial y \\ \partial Range2/\partial x & \partial Range2/\partial y \end{bmatrix} \quad \textbf{(3.16)}$$

This H matrix represents a linearized relationship between the state and the measured ranges. The new H is used by Equations (3.8) and (3.9) to calculate Kalman gains and covariances as they relate to the measurements. The shock calculations must also change to reflect the amount of state change required. The new shock (Equation 3.17) is the difference between the actual measurements Z and the calculated measurements f(U) as based on the system state. Where Z holds the received ranges from the DiveTracker system.

$$Shock = Z - Fu - Uv \quad \textbf{(3.17)}$$

F. SPEED/CURRENT ERROR MODEL

If a measured Phoenix position does not agree with the motion model's position, then as the filter updates the system state the X and Y ocean current speed components will be increased to explain the difference. The ocean current speed components of the system state are actually a combination of ocean current and navigation errors caused by inaccurate vehicle speed and heading inputs. In the absence of measurements, the speed variances will slowly increase. In the long run, according to the movement model, vehicle speeds in excess of 1000 knots are not only possible but likely [WASH94]. Modeling these speeds as a discrete Ornstein-Uhlenbeck process (O-U) will correct this problem by exponentially decreasing the value of the ocean current speeds over time. This is useful for long term modeling of ocean or tidal currents. With this approach a value of C, where ($0 \leq C \leq 1$), is

used to decrease the value of the drift speed exponentially (Eq 3.18). That is,

$$C = \exp(-\Delta/T) \tag{3.18}$$

In this equation, Δ is the time step between cycles and T is the drift relaxation time. As an example for the case of Xdrift, the state component update equation changes to;

$$X_{i+1} = C*X_i + W_x \sim N(0,Q) \tag{3.19}$$

Consequently, the Xdrift variance changes to;

$$Var(Xdrift) = C^2 * Var(Xdrift) + Q \tag{3.20}$$

The limit of Var(Xdrift) as time approaches infinity is the average of $Xdrift^2$, so Q reduces to,

$$Q = Xdrift^2 * (1-C^2) \tag{3.21}$$

The final modification in the O-U process involves the Δ used in the Φ matrix. Now, the drift speeds not only fluctuate about zero, but they also decay toward zero at the rate specified by C. This results in a new term $\delta = T*(1-C)$, where δ is always smaller than Δ, although there is very little difference when Δ is small compared to T. The final result is a modified Φ matrix given by,

$$\Phi = \begin{bmatrix} 1 & 0 & \delta & 0 \\ 0 & 1 & 0 & \delta \\ 0 & 0 & C & 0 \\ 0 & 0 & 0 & C \end{bmatrix} \tag{3.22}$$

G. SUMMARY

Discrete Kalman filtering is a statistical method of calculating a new system state based on a series of measurements. The Phoenix navigation module uses a system state of [Xpos Ypos Xdrift Ydrift]^T, and measurements of GPS position and DiveTracker ranges. The use of DiveTracker ranges requires an Extended Kalman filter due to non-linearity of the measured ranges. Drift speeds are modeled as a Ornstein-Uhlenbeck process to keep the calculated speeds in bounds.

IV. NAVIGATION

A. INTRODUCTION

For the Phoenix AUV to be effective in mine warfare requires precision navigation with desired position estimates within several meters of actual positions. To solve this problem, a discrete Kalman filter was used to filter the GPS/DGPS and DiveTracker measurements and produce the most probable vehicle position. However, this filtering was only a means of utilizing measurement and dead reckoning to provide new positions. The Kalman-Filter by itself did not "solve" the navigation problem. Questions about initialization, accuracy of position fixing methods, which fix type to use under which conditions, and dead reckoning problems all must be solved before a fully functional filter can be implemented.

B. NAVIGATION OVERVIEW

The Phoenix navigation module works in a continuous loop as a forked process of the tactical level [LEON96]. The module receives the vehicle state string from the tactical level. From the state string, the values of speeds, vehicle attitude, heading and DiveTracker ranges are obtained. If the Phoenix depth is less than one foot, then an attempt is made to read GPS from the Motorola unit. If the Kalman filter has lost track for 15 seconds, the tactical level is informed, and the vehicle will surface to gain a GPS fix and reset the filter parameters. The Kalman filter is reset by re-initializing the gain and variance matrixes. Fix types are compared, and the appropriate fix position data type is selected for use. If there is no fix position data, the state vector drift values are manually updated using the computed

total drift resolved to Xdrift and Ydrift speed components using the vehicle's heading.

The Kalman filter routine is next called and passed the parameters for the selected fix type. The Kalman filter first performs the dead reckoning movement step. If no fix data was available, the filter returns the new dead reckoned state position estimate. If fix data was available the measured data is filtered and new Kalman gains are computed. Dimensionless Shock is calculated to determine if the measured data was reasonable. If the Dimensionless shock value is low, the state vector is updated using the computed Kalman gains and measurements. If the Dimensionless shock was too high, the measurements are ignored, the state vector is not updated and a loss track flag is set. The value of the root mean squared total drift is next calculated. The filter then returns the updated state, total drift and loss track flag data. Finally, the navigation module sends the new fix data back to the tactical level and records the fix data for later analysis, and the loop continues again. Loop timing is controlled by the time stamp in the state string received from the tactical level OOD. If there was no state string received, the loop performs a busy wait until a state string is received, if a state string is received then data processed by the loop uses the time passed in the state string.

C. NAVIGATION CO-ORDINATES

A right hand rule system of X, Y and Z measured in feet was used for Phoenix Navigation. In this system X is aligned along the North earth axis, with Y along the East axis and Z being down. This required the conversion of GPS/DGPS position data from latitude and longitude to X and Y in feet. The GPS/DGPS system raw data stream reported position data in milli-seconds of arc latitude and longitude. Before the GPS/DGPS data could be used by the Kalman filter, the data had to be converted both to feet and to the local coordinate

system. This was performed by first determining an origin (starting location) during the filter initialization phase. All subsequent fixes are referenced against this origin position to get a calculated difference in latitude and longitude milli-seconds of arc from the fix position to the origin position. The differences in milliseconds arc latitude are converted to feet by the relationship of 10 milli-seconds of arc per foot latitude. To convert longitude data a spherical world approximation was used (Eq. 4.1).

$$distancelongitude = longitude^* \cos(latitude) \quad \textbf{(4.1)}$$

Distance longitude was then converted to feet using the same 10 milli-seconds of arc per foot factor. These new X and Y distances in feet were then applied to the vehicles X, Y starting position to arrive at a new fix position in X,Y coordinates.

D. GPS/DGPS

GPS is a world wide satellite based system that provides highly accurate position data [MOTORO]. There are 26 satellites available, with a minimum of three satellites required to compute a fix position. The U.S. Department of Defense runs this system and intentionally perturbs the GPS signals so that accuracy of only approximately 180 feet RMS error in position can be achieved without special equipment. GPS operates on the measured time delays between the received satellite signals. To increase accuracy, a differential GPS (DGPS) system has been developed and is now widely commercially available [MOTORO]. DGPS receives the GPS signal at a surveyed land based site, and then broadcasts a correction time signal for GPS users to obtain accuracies of within 2 meters and more recently using carrier phase inversion methods to within 2 centimeters [LACH96].

1. Phoenix GPS/DGPS Variances

The Phoenix AUV received GPS/DGPS at an average rate of one fix per second when surfaced. This resulted in asynchronous data dependent upon the Phoenix depth and satellites availability. To determine the noise variances of the GPS/DGPS system, static (non-moving) testing of the unit was performed in the laboratory. Figures 3-6 detail the results of a 17 hour GPS and 7 hour DGPS test. Positions are recorded at five second intervals. Figure 3 shows the GPS latitude and longitude data (converted to feet) fluctuations. The standard deviation of this data was 100 feet latitude and 66 feet longitude. Figure 4 indicates the range of positions recorded over the 17 hour period. The pronounced gap in the data received around time 14 indicates when the minimum of three satellites apparently was not available.

Figure 5 and 6 show the same data for DGPS over a 7 hour period. The standard deviations of DGPS in latitude and longitude was 19 and 11 feet latitude and longitude for a tighter distribution. In this case the gap in data most probably resulted in a loss of DGPS correction signals. There is a notable increase in error before the gap which is consistent with the loss a correction signal.

Figures 5 and 6 also illustrate a loss of DGPS correction signal problem with our receiver. The Motorola receiver holds a received differential correction signal for 90 seconds before defaulting to uncorrected GPS mode. This results in increasing inaccuracies of up to 40.5 meters at ninety seconds [MOTORO]. These errors can be seen in Figure 6 as the occasional loop out from the bulk of the positions and the long spikes in Figure 5. In real world use, it is better to hold a correction signal as long as possible, because even the 40.5

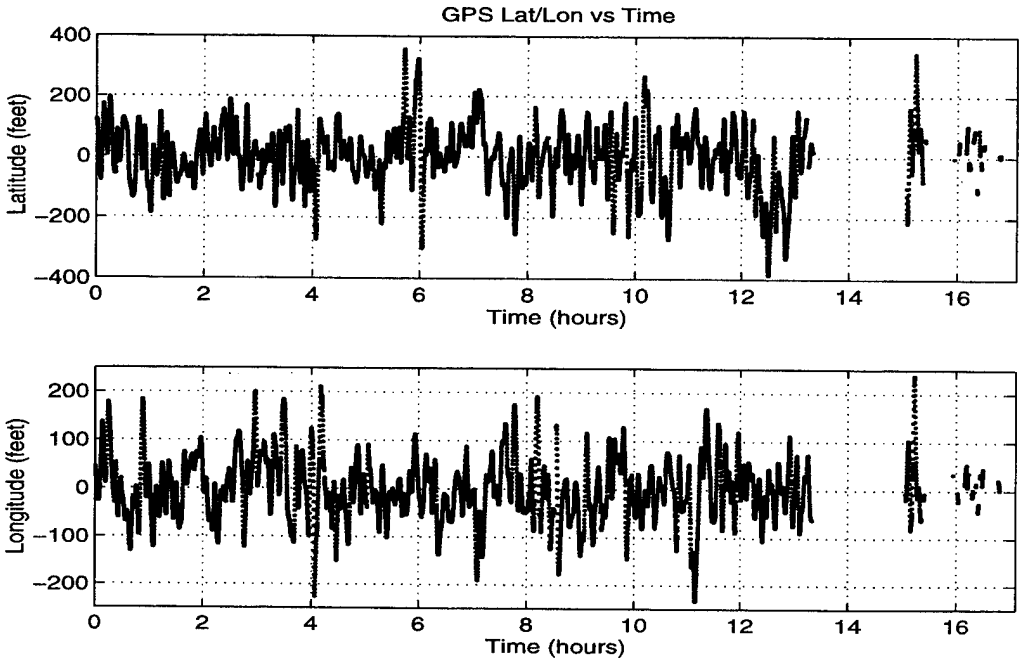

Figure 3: GPS Lat/Lon vs Time

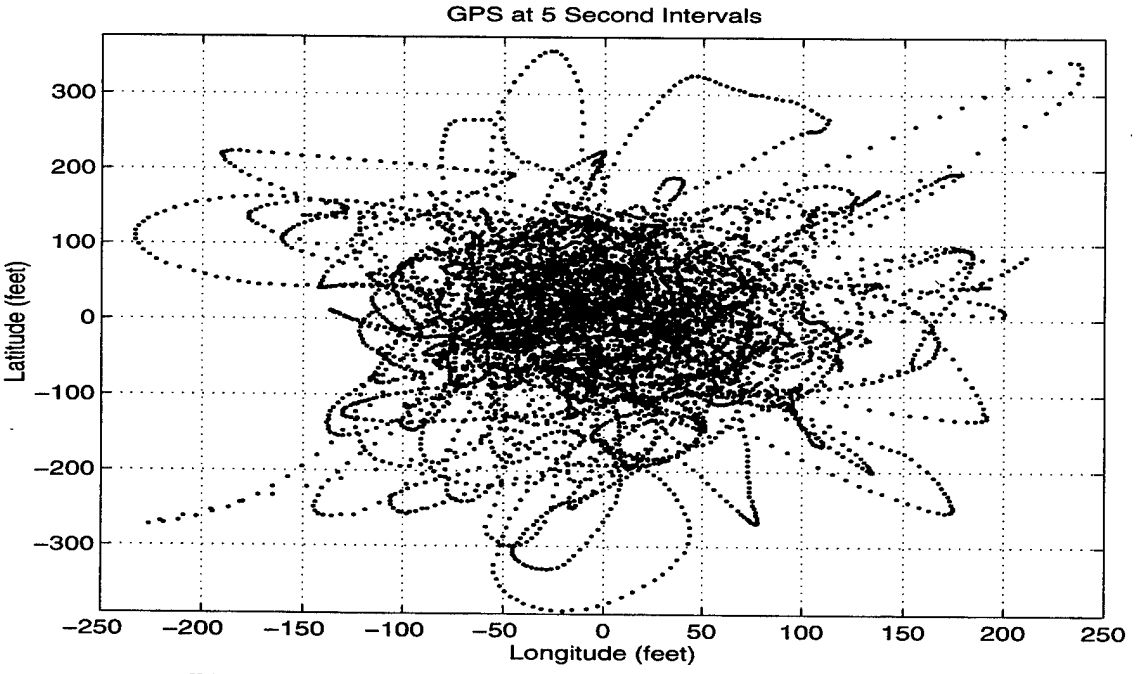

Figure 4: Plot of Raw GPS data Positions over 17 Hours

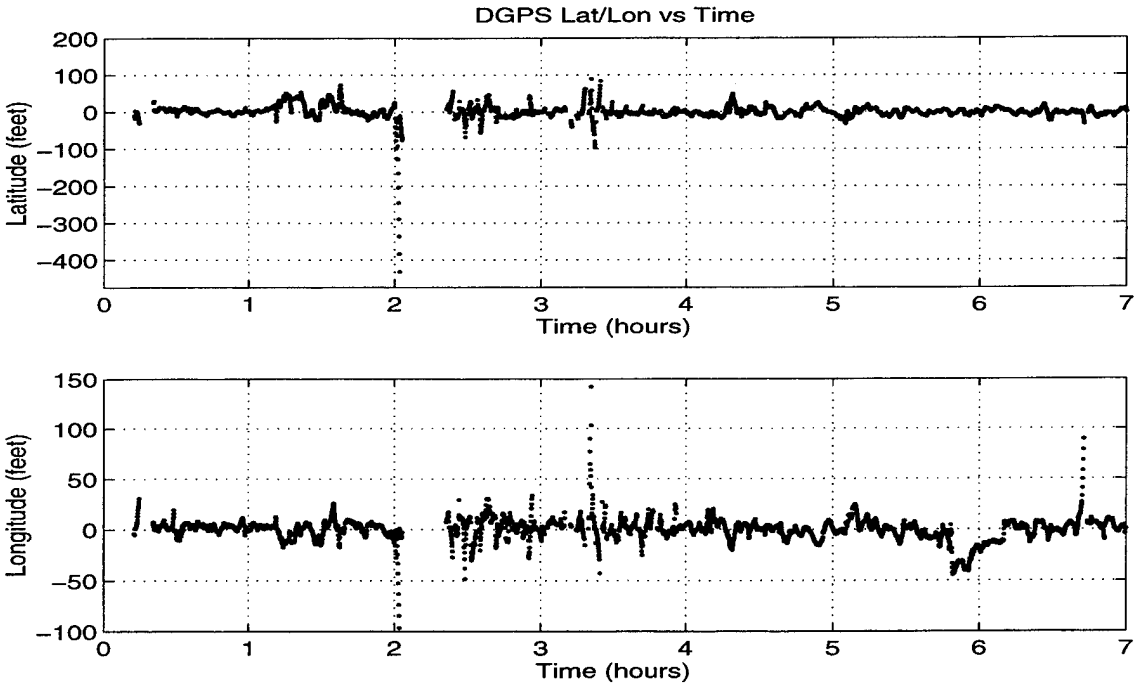

Figure 5: DGPS Lat/Lon vs Time

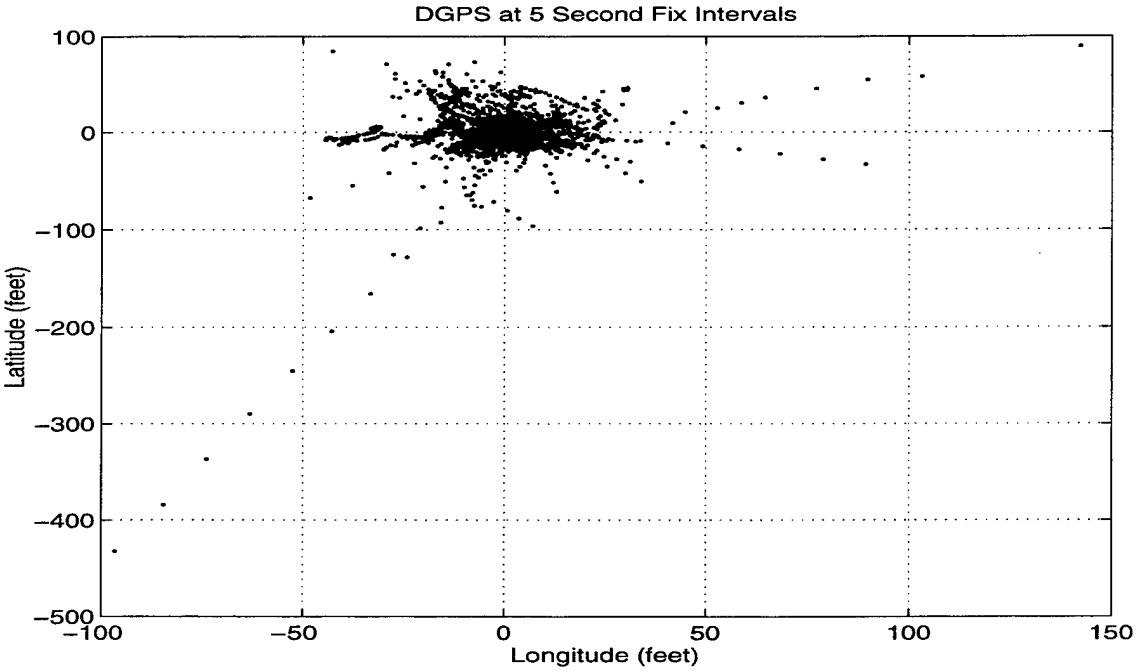

Figure 6: Plot of Raw DGPS Positions Over 7 Hours

meter inaccuracy after 90 seconds is better than the 60 meter inaccuracy of uncorrected GPS. However, the Kalman filter demands the knowledge of the measured variances, and after 30 seconds without correction, the estimated position error can exceed the calculated DGPS standard deviations and result in solutions exceeding the expected accuracy of position estimates.

2. Kalman Filtering of GPS/DGPS Data

The raw GPS/DGPS data shown in Figures 3-6 was input to the Kalman filter with the following results. Using the square of standard deviations of the raw GPS/DGPS data as variances for the Kalman filter proves the capability of this method. Figures 7 and 8 show the results of the filtering of the GPS data. The standard deviation of the filtered GPS data was reduced from 100 feet to 9 feet latitude and from 66 feet to 6 feet longitude. Increasing the variance by 100 in Figures 9 and 10 show a reduction in standard deviation to 5 and 4 feet latitude and longitude. Increasing the variance by 1000 in Figures 11 and 12 show a reduction in standard deviation to 3.8 feet and 2.5 feet latitude and longitude. The same type of result can be seen for the DGPS data in Figures 13-18, where the DGPS variance by itself produced standard deviations of 1.75 feet latitude and 1.1 feet longitude. Increasing these variances by 100 produced standard deviations of 1.1 feet latitude and .88 feet longitude and .78 feet latitude and .75 feet longitude for a 1000 fold increase. Of course these results apply only to a stationary receiver. The long time constants associated with large variances would not be suitable for use on a maneuvering vehicle such as Phoenix.

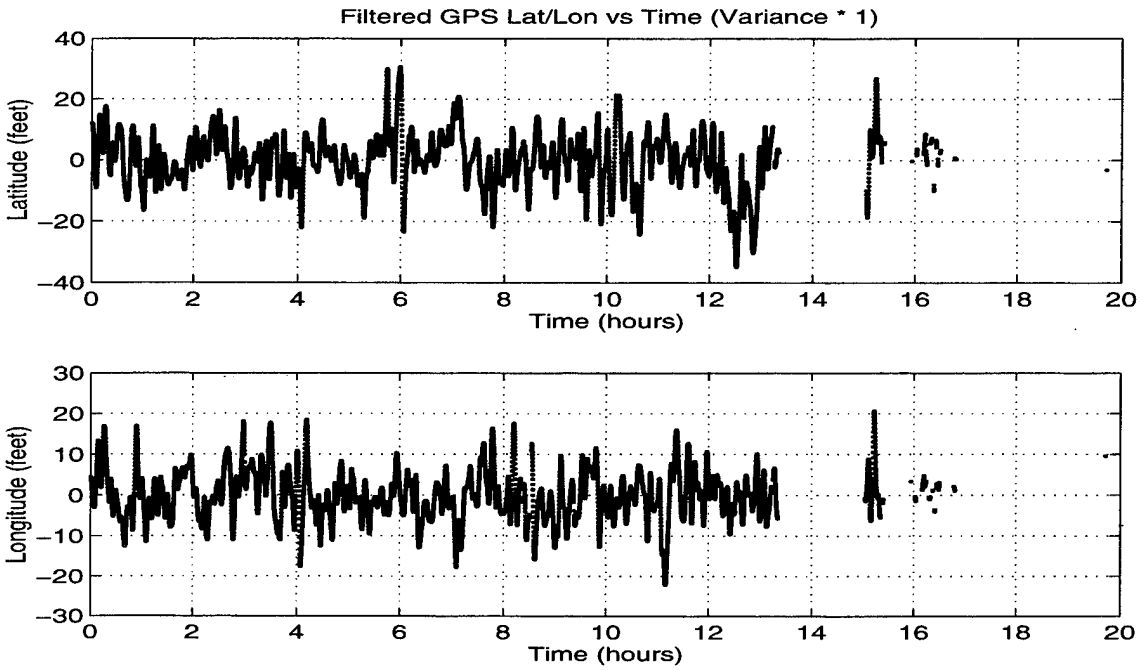

Figure 7: Filtered GPS Lat/Lon vs Time

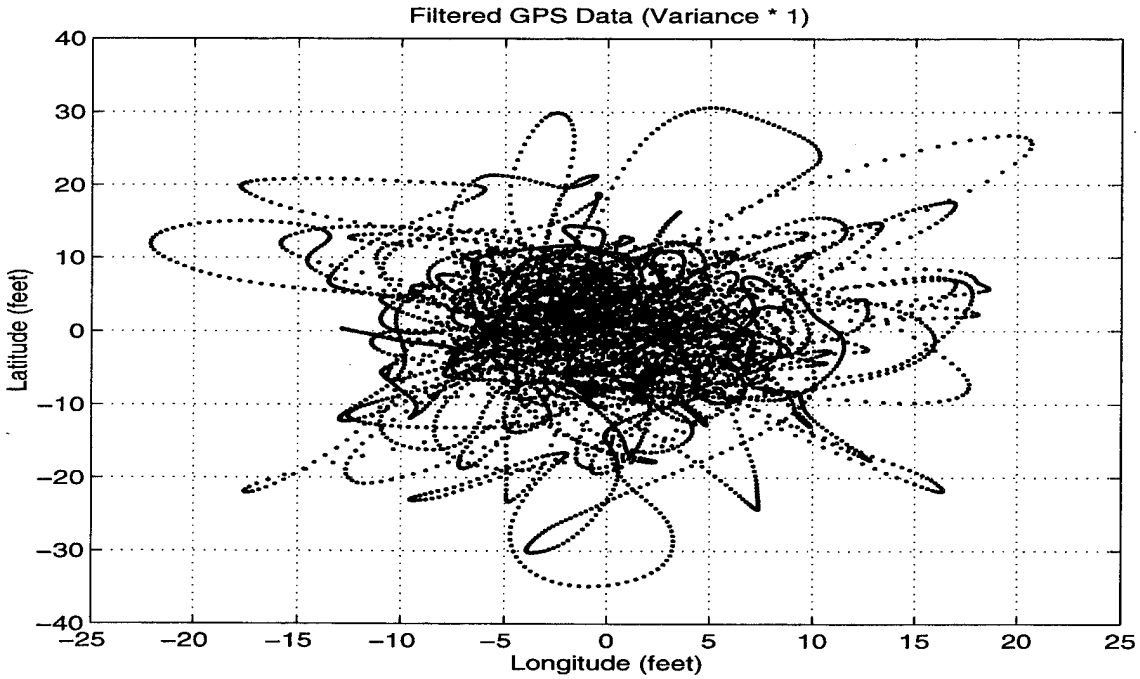

Figure 8: Plot of Filtered GPS Positions Over 17 Hours

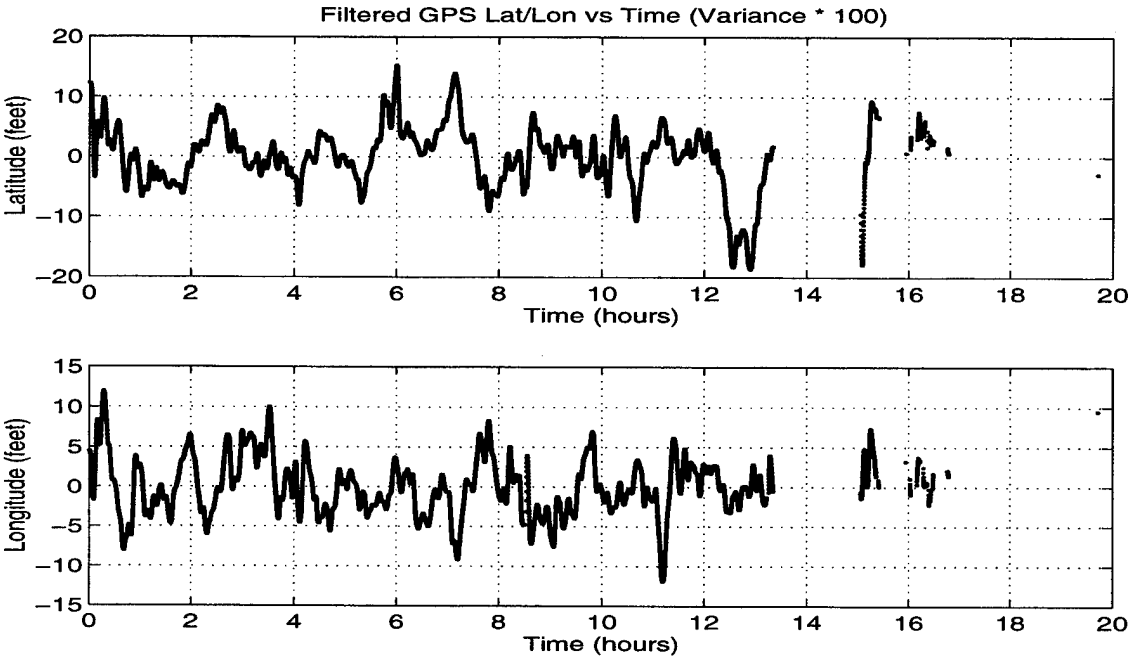

Figure 9: Increased Variance Filtered GPS Lat/Lon vs Time (Variance * 100)

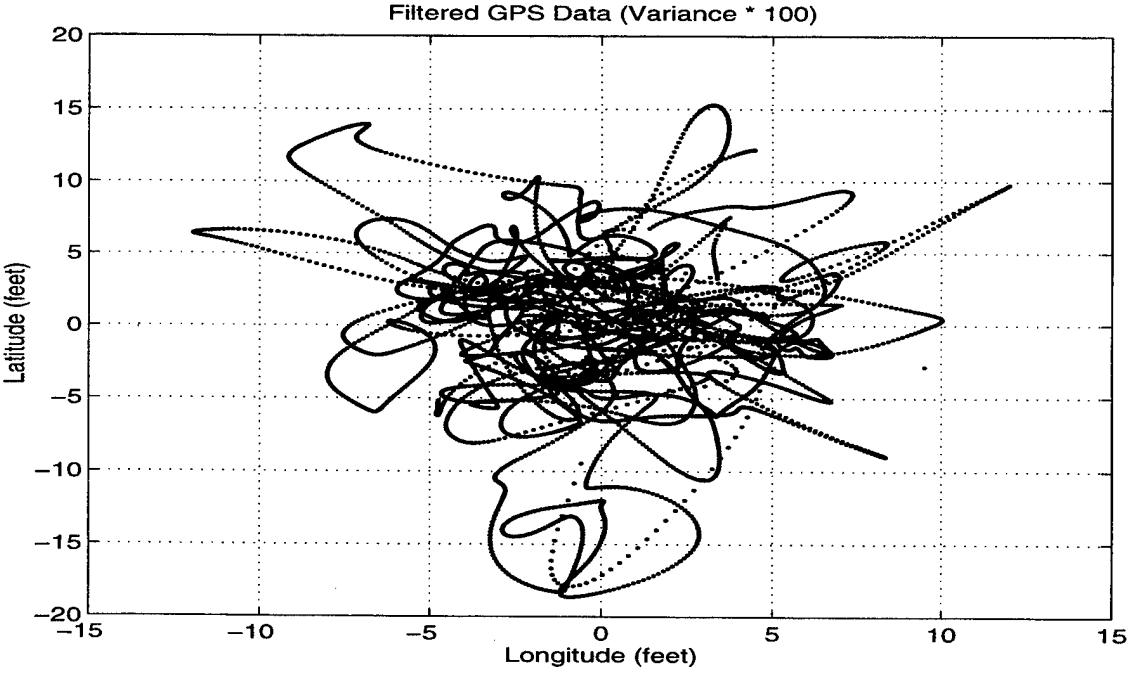

Figure 10: Plot of Increased Variance Filtered GPS Positions Over 17 Hours

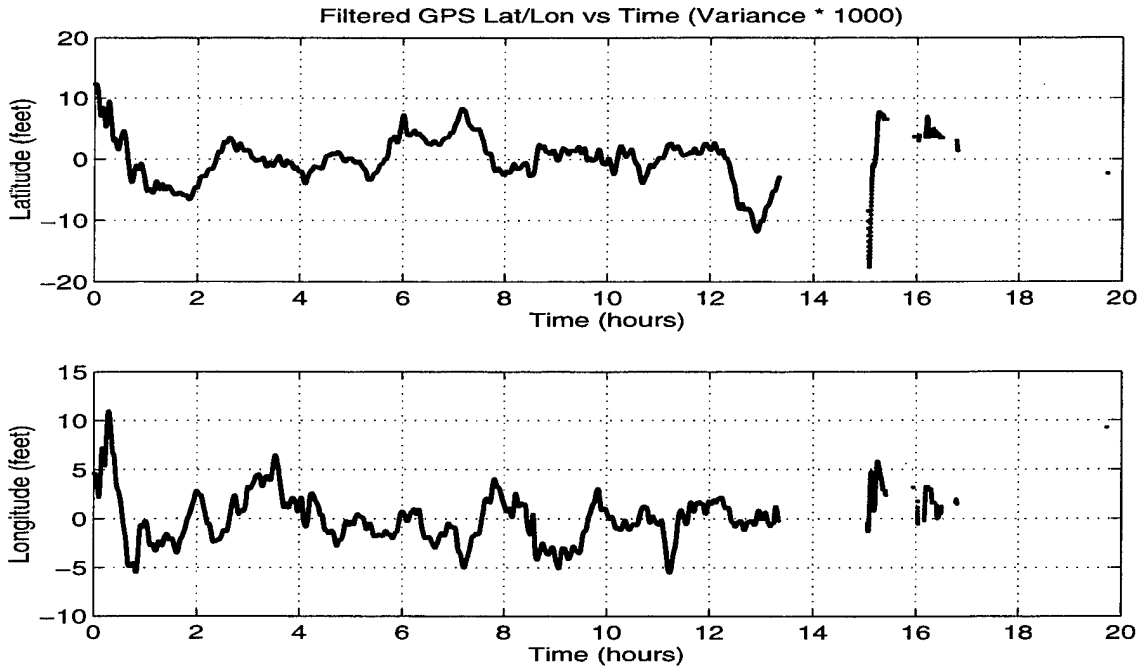

Figure 11: Increased Variance Filtered GPS Lat/Lon vs Time (Variance * 1000)

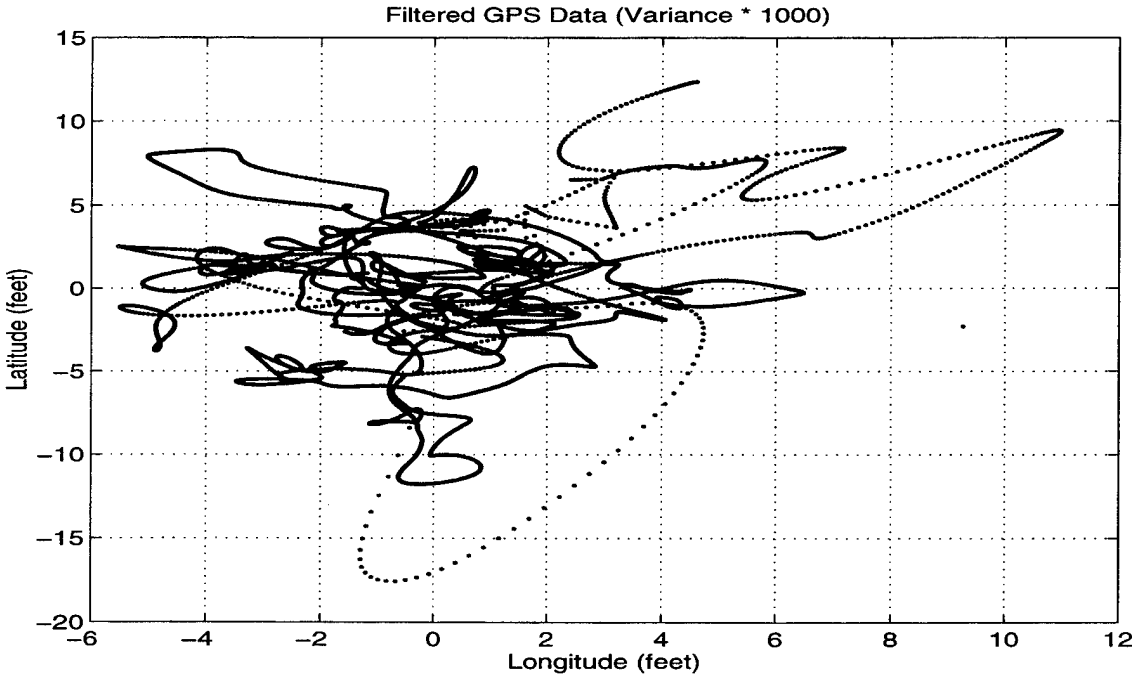

Figure 12: Plot of Increased Variance GPS Positions Over 17 Hours

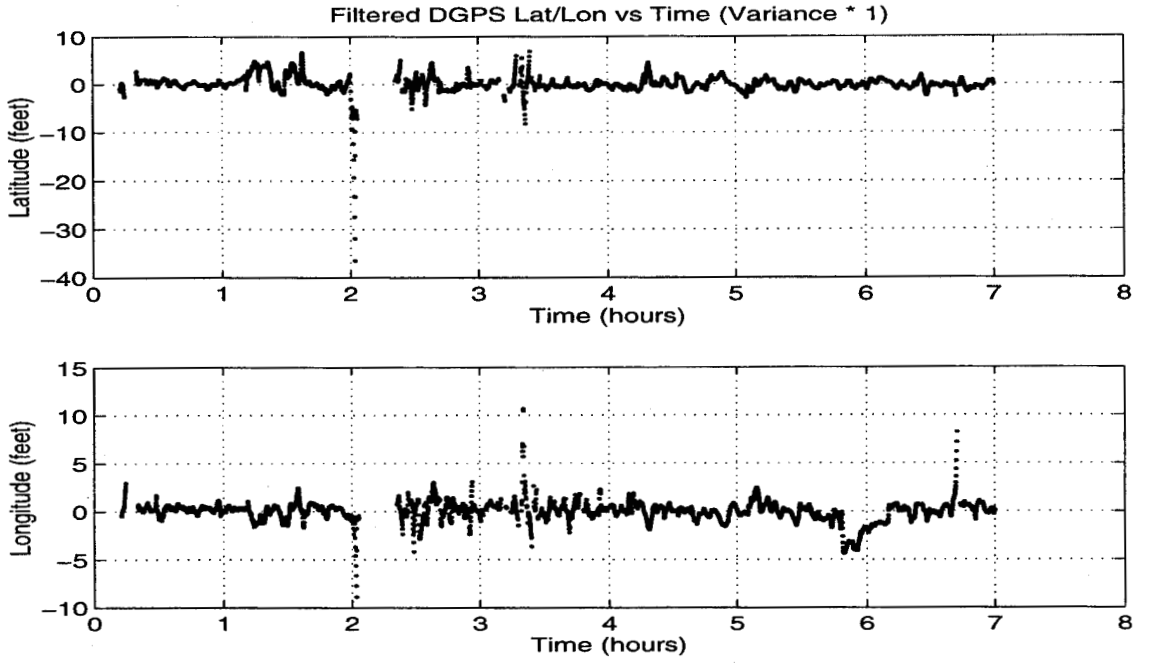

Figure 13: Filtered DGPS Lat/Lon vs Time

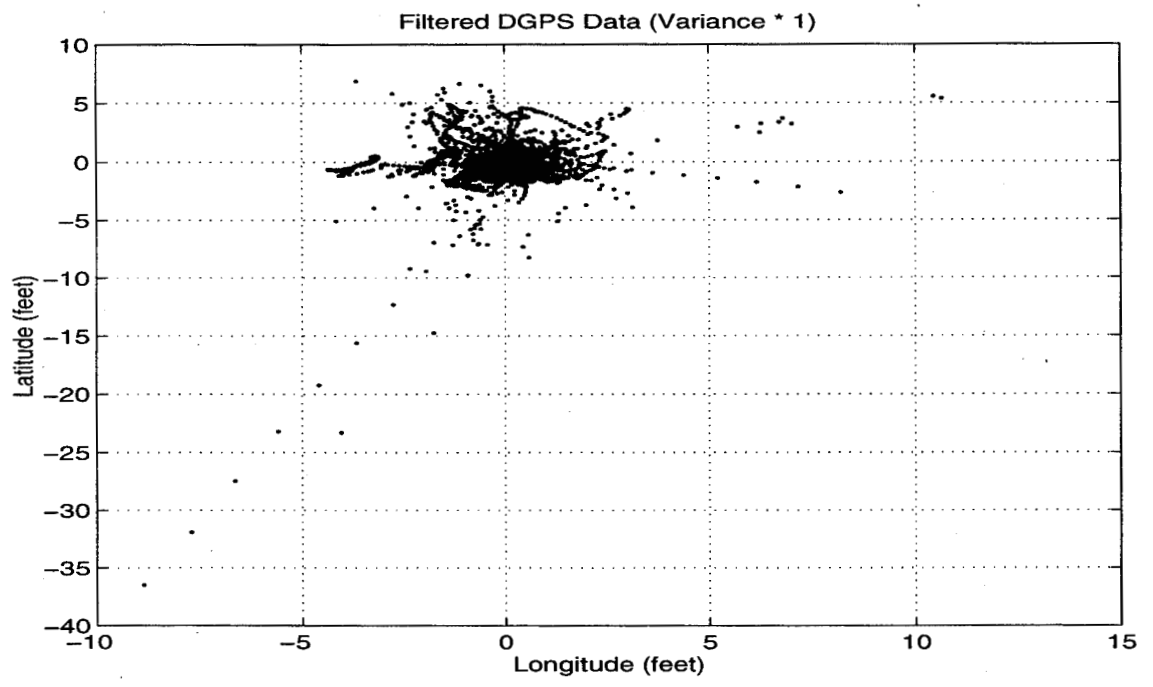

Figure 14: Plot of Filtered DGPS Positions Over 7 Hours

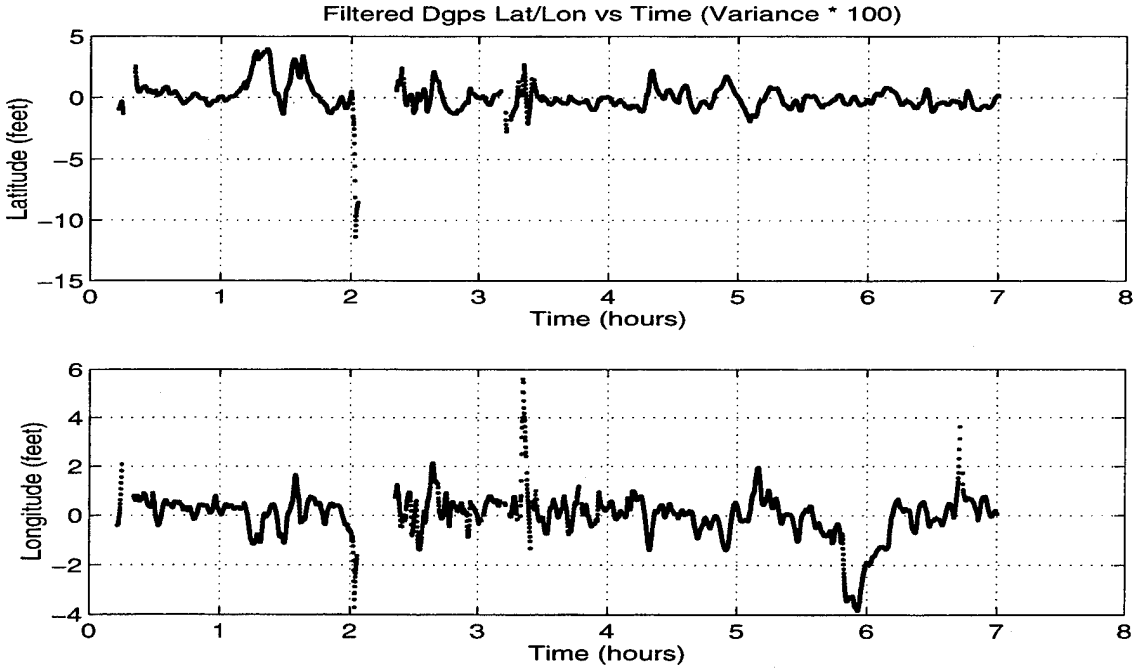

Figure 15: Increased Variance DGPS Lat/Lon vs Time (Variance * 100)

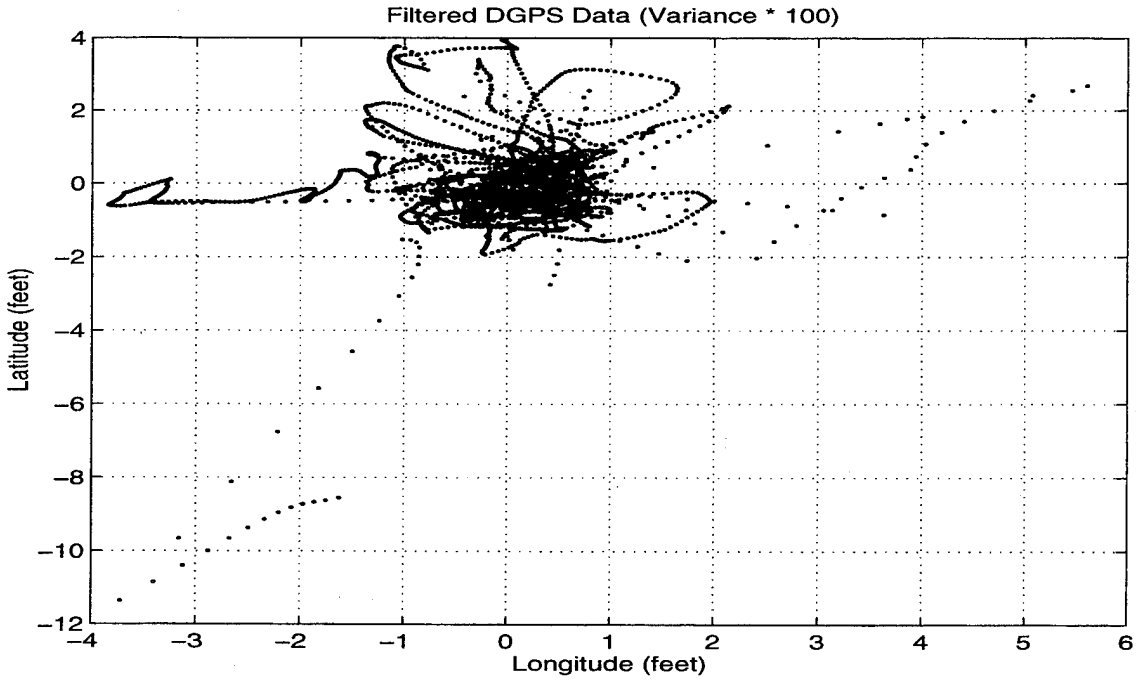

Figure 16: Plot of Increased Variance DGPS Positions Over 7 Hours

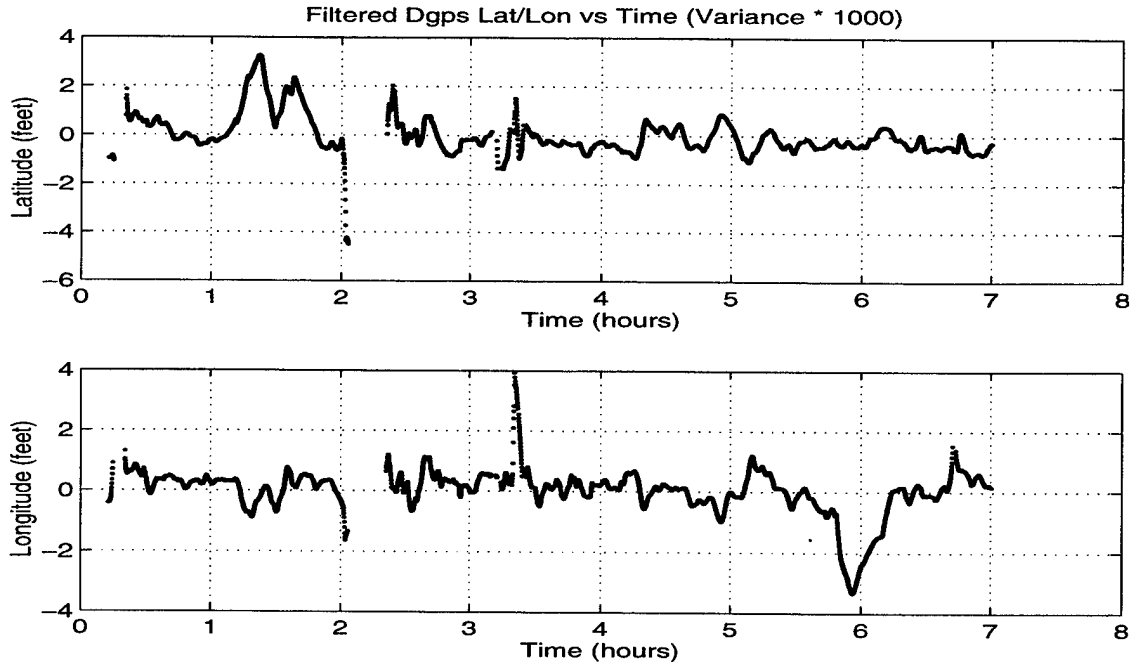

Figure 17: Increased Variance Filtered DGPS Lat/Lon vs Time (Variance * 1000)

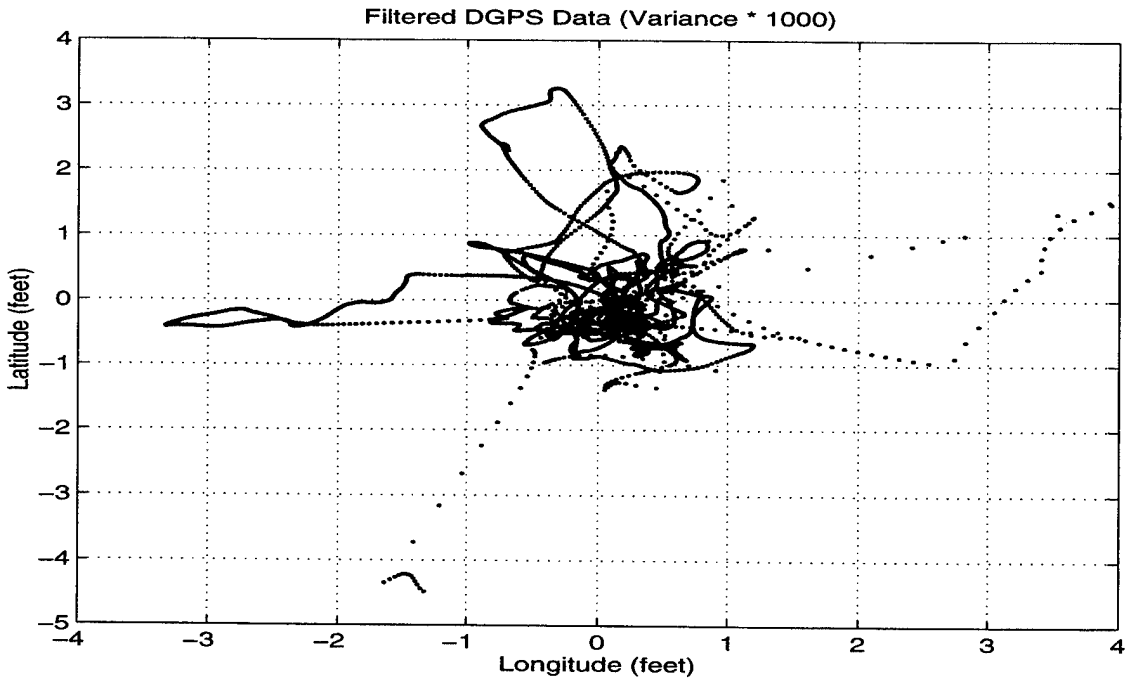

Figure 18: Plot of Increased Variance DGPS Positions Over 7 Hours

3. GPS/DGPS Navigation

The Phoenix almost always received GPS signals, but the DGPS correction signal could only be received during pier testing. When ready to launch, at sea level, a differential signal was not received at all, a suspected cause was local waterfront interference consisting of the pier itself or other ships in the vicinity. However, investigation revealed this problem to be the result of electromagnetic interference caused primarily by the Phoenix gyros, which are only started when ready to launch. Whenever the gyros were started the differential signal was lost. This created a problem of GPS/DGPS fix accuracy. In our sea trial area of operation, a raw GPS fix could position the Phoenix anywhere within the slip. The DGPS raw position was much more accurate, with two meters accuracy depending upon the receipt of the correction signal.

E. DIVETRACKER RANGE UTILIZATION

The DiveTracker sonar ranging system provided two independent ranges from base transducers to the Phoenix with accuracies within one foot. The DiveTracker data is asynchronous, and is normally received in 1 to 3 second intervals. The filtered range positions provided a much more accurate method for position fixing than our version of GPS/DGPS. However, the system only worked for ranges of up to 1000 feet, and the fix positions available were geometrically dependent upon the baseline locations of the transducers. For the DiveTracker system to be reliably used, the set up for transducer positions and calculations of optimum operating area must first be performed. This limits the Phoenix missions to these areas. In addition, care must be taken to avoid position ambiguity that can result from "crossing the baseline".

1. Divetracker Variance

DiveTracker variances were statically determined to be less than 1 foot^2, depending upon the range [SCRI96]. Figures 19 and 20 show an example of fix accuracy using a range error of ± 0.75 foot over 60 foot ranges with a 50 foot baseline. Figure 20 shows the generated error area of 2.97 feet^2. The second plot in Figure 20 shows the same area superimposed with the 2000 normally distributed positions based on the same range error. The normally distributed positions had a mean of the absolute position and a variance of 0.75 feet^2. In this example the normally distributed positions cover a larger area than the possible geometric error. A Kalman filter is designed to control normally distributed error, so in this case the possible geometric error is well under control. However, Figure 21 demonstrates the dependence of geometry in possible position errors. In this case 120 foot ranges with the same 50 foot base line is used to develop a geometric error area of 5.51 feet^2. Now the geometric error begins to grow larger than the normally distributed position errors. As ranges from the baseline increase, the possible geometric error continues to grow and exceeds the range of normally distributed position errors that the Kalman filter is designed to control. This can cause problems with position errors as ranges from the baseline increase.

2. Baseline Problem

DiveTracker range navigation introduced the problem of fix inconsistencies across the baseline between the transducers. Position determination while crossing the baseline is a problem because the ranges are identical from one side of the baseline to the other. This is normally not a problem because the baseline is normally set up so the Phoenix mission never crosses it. The Kalman filter may not be able to track which side of the baseline the

vehicle is on. Figure 22 illustrates an example of this problem that occurred during vehicle testing. The figure shows a track denoted by the solid line provided by the Kalman filter. The small dotted line segments show the dead reckoned movement. In this trial the vehicle started on the baseline at approximately X = 26 and Y = 0. When the trial started the vehicle proceeded as ordered along the positive Y axis, but as Figure 22 shows the filter tracked the vehicle running towards the negative Y axis. The vehicle dead reckoning traces show the vehicle moving in the desired direction, but every DiveTracker fix reset the vehicle position farther to the left.

F. FILTER RESPONSE VS VEHICLE STABILITY

The question of fix accuracy vice vehicle stability depends upon the actual variances used in the Kalman filter. If the variances are small, then the filter tends to believe the measurements more than the movement model. This results in a "high strung" behavior, where the fixes jump from location to location. Such behavior can create a serious problem in vehicle stability. For example; if the vehicle is attempting to hover at a designated point and the fixes keep jumping around, the vehicle may never achieve that point. If the variances are too high, the filter tends to ignore the measurements and follows the model. This gives a sluggish behavior to the filter where it isn't really following the measured positions. This has the least effect on vehicle stability, for it tends to believe its own model. Finding the proper values of variance is an ongoing effort, which will be studied farther in the thesis research of subsequent students working on Phoenix navigation and control. To overcome the inaccuracy problems with GPS/DGPS, the variances used by the Kalman filter for these measurements were increased a hundred fold from the experimental data. This kept the fix

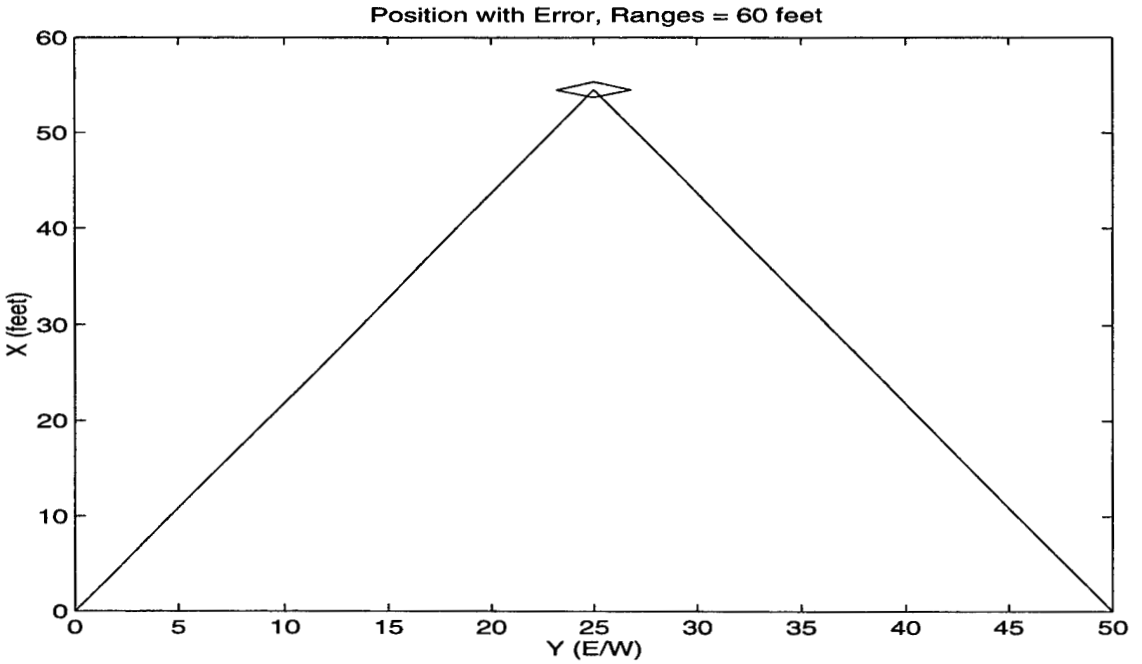

Figure 19: Plot of Position with Error for 60 Foot DiveTracker Ranges

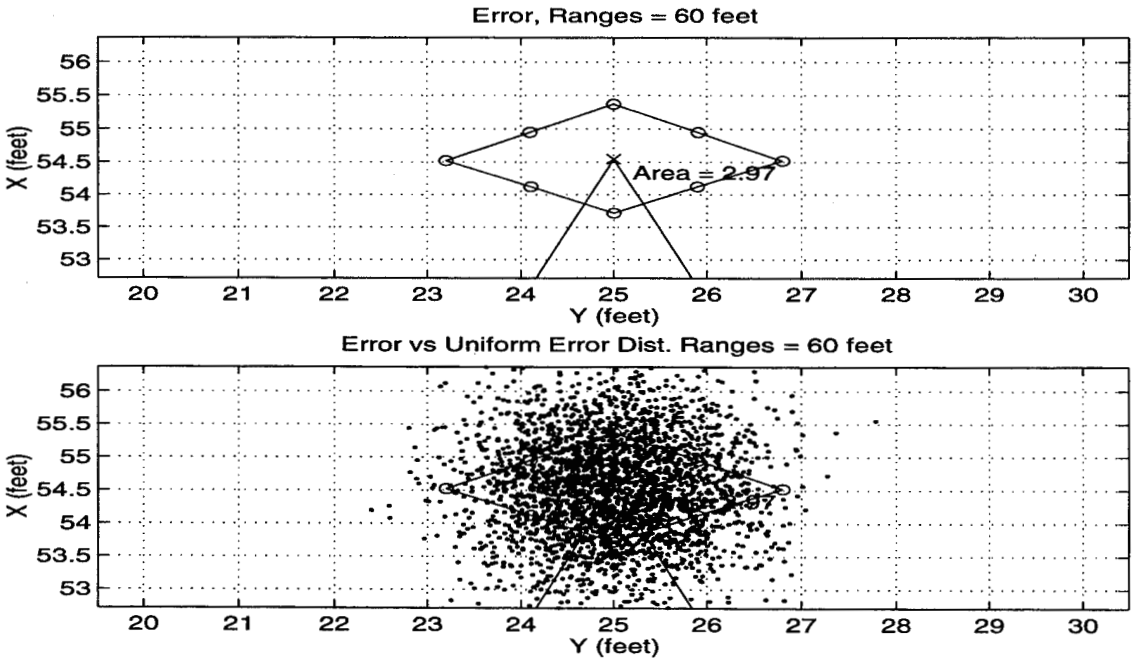

Figure 20: Top: Expanded View of Figure 19 Error Area
Bottom: Same Error Area with Random Error Superimposed

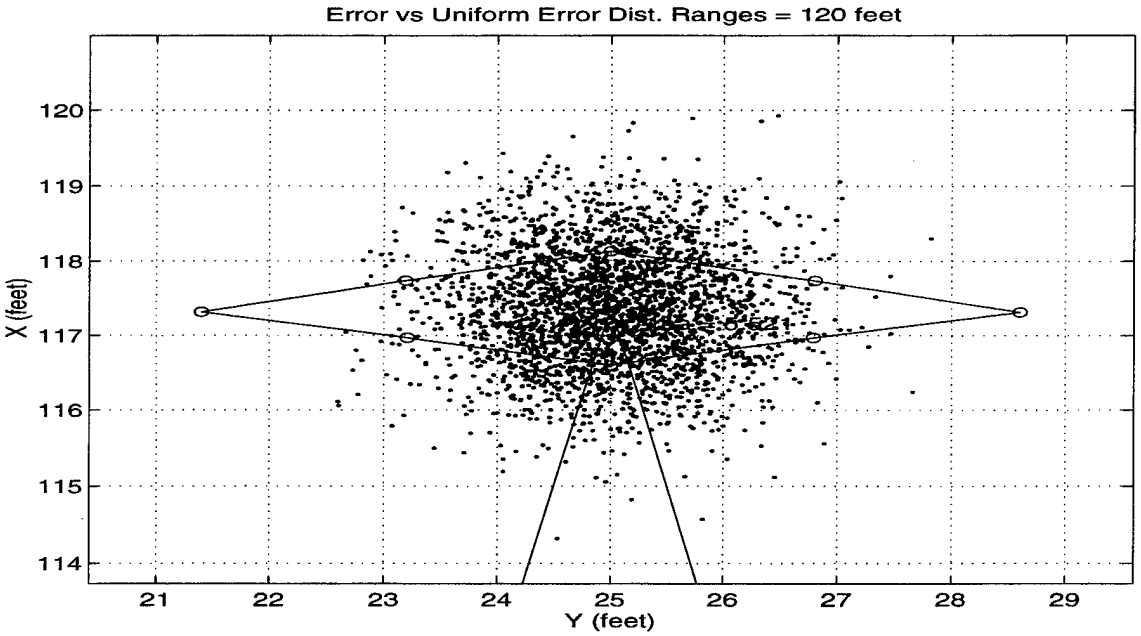

FIGURE 21: Error Area for 120 Foot Range, with Random Error Superimposed

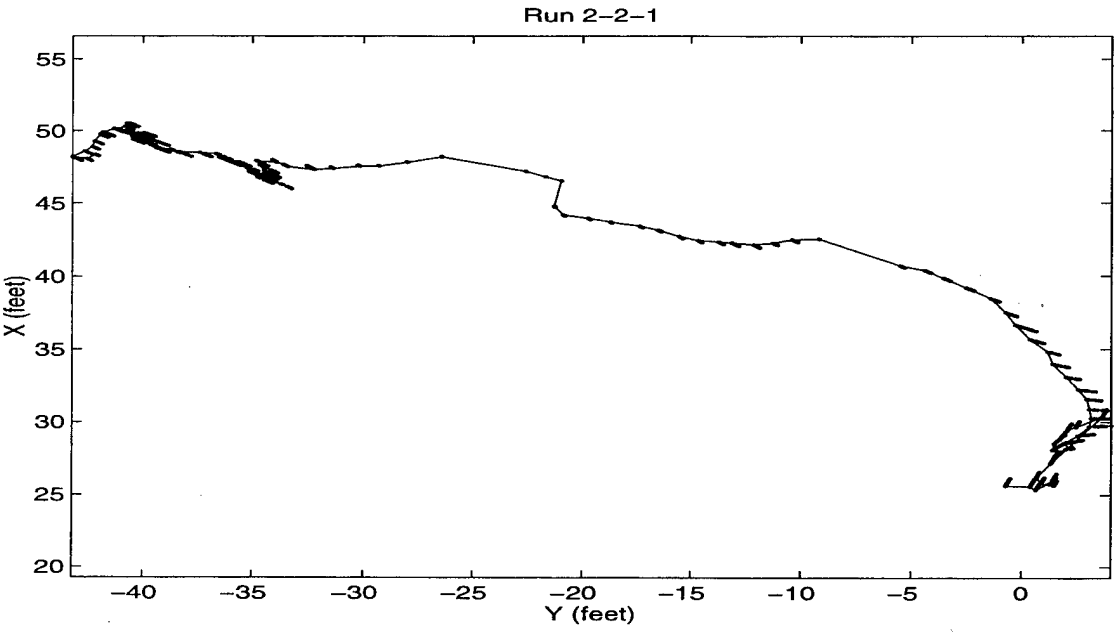

Figure 22: Example of Baseline Crossing, Showing Wrong Direction Tracking

data stable, but resulted in sluggish behavior by the filter when using only GPS/DGPS data. The variances for the DiveTracker ranges was increased by a factor of five, which seems to give good results.

G. FIX DETERMINATION

A fix determination routine was used to determine both what type of measurement data was available and which one to use for position fixing using the following metrics. If only one measurement was available, then it was selected. If no measurement was available, the system would dead reckon. The case when multiple measurements were available required the following metrics.

CASE 1: Both DiveTracker and GPS/DGPS are available, the lost track flag is set, and the last fix was by DiveTracker. In this case the system used DiveTracker as the last measurement and the system lost track. Since the DiveTracker data resulted in a loss of track, then fix data is switched to the GPS/DGPS (a more reliable but less accurate system).

CASE 2: Both DiveTracker and GPS/DGPS were available and the loss track flag was not set. In this case the system will use DiveTracker which provides more accurate position fixing.

CASE 3: Both GPS/DGPS and DiveTracker were available, the loss track flag was not set but the GPS/DGPS fix position is not within the State Estimate ± GPS/DGPS Standard Deviations. In this case the system has not lost track, but the vehicle state position vector U places the vehicle at a position more than the GPS fix plus it's standard deviations. Since GPS/DGPS is a reliable world wide system, the vehicle position is reset to the

GPS/DGPS fix.

H. FIX POSITION TRANSLATION TO VEHICLE CENTER

The measurements received by the GPS/DGPS antenna and the DiveTracker transducer result in fix positions at those locations rather than the center of the vehicle. This results in a fix inaccuracy of approximately 2.5 feet for a DiveTracker and 1 foot for GPS/DGPS fixes. This offset can cause a significant error when attempting to twist or rotate the vehicle about its center because the fix update position is not at the vehicle center. To correct this problem, the Phoenix state vector is centered on the vehicle. The motion model calculates movements based on the vehicle center. When a measurement is received, the vehicle center is translated to the antenna or transducer location as required. The vehicle state is now updated based on the new measurement, then translated back to the vehicle center. This translation (Eq 4.2 and 4.3) depends on two variables, the vehicle heading and the distances to the transducers. Only the fore/aft Phoenix offset distances are used; the slight athwart ships offset is ignored. Thus,

$$TranslatedX = OldX + Offset*\sin(\Psi) \tag{4.2}$$

$$TranslatedY = OldY + Offset*\cos(\Psi) \tag{4.3}$$

where the offset is the distance from the center of the vehicle to the antenna or transducer. Offset is a positive value for the GPS antenna and a negative value for the DiveTracker transducer.

I. NAVIGATION INITIALIZATION

Before the navigation module could successfully run, it required the following data inputs from the tactical level OOD. The DiveTracker base station transducer locations were needed for the Extended Kalman Filter module. The initial posture (starting location) of the vehicle was required to convert GPS/DGPS data to the local co-ordinates. A gyro error input was needed to compute accurate dead reckoning.

When a mission commenced the navigation initialization routine waited 30 seconds before reporting "Initialized" to the tactical level OOD to allow the GPS/DGPS positions to stabilize on first startup. When the tactical level OOD received reports that all modules had initialized, the first command to the Phoenix is to submerge and wait for another 30 seconds before transiting. This allowed the Kalman filter to stabilize and produce good fix data as it had now shifted from primarily GPS/DGPS position data while surfaced, to DiveTracker position data while submerged.

J. OCEAN CURRENT (ERROR) ESTIMATION

Accurate and efficient navigation from point to point requires the knowledge of the local ocean currents to prevent a "tail chase" to the desired location. If the vehicle fix position consistently does not agree with the modeled position, then current components are generated to overcome the error. The computed ocean currents are actually the combined sum of any ocean current, speed/heading, and model errors. Since the computed currents also include errors, the values may change with the vehicle heading, but the RMS value of the current will converge to a steady state number. This number is resolved into its X and Y components for dead reckoning use.

K. WATER SPEED SENSOR CALIBRATION

The DiveTracker system accuracy in position was used to calibrate the Phoenix water speed sensor. Water speed is the relative speed that the Phoenix moves through the water. The sensor is a small "water wheel" turbine, and speed is determined based on the wheel rotational speed (frequency). A Phoenix straight line run over a set distance and time was performed. By post processing, computed speeds were calculated based on the traveled distance and times between DiveTracker fixes. A graph of these calculated vehicle speeds compared to the recorded speed sensor output was plotted vs time. Figure 23 shows the result of the first speed sensor calibration run. The top figure shows the path followed by Phoenix. The bottom figure shows a graph of the calculated speeds vs the model speeds and the speed sensor speeds. An approximating polynomial describing the speed probe output was then modified to match the calculated speed curve. A second run was then performed to check the modification results. Figure 24 shows a much closer agreement between speed sensor and computed speeds after the modifications were performed.

L. SIMULATION MODE

The simulate flag can be set to allow for developmental code testing utilizing the Virtual World AUV simulator [BRUT94]. This simulator provides a full mathematical simulation of the Phoenix AUV with estimated hydro-dynamic effects and a visual animated display. The use of this simulator allowed all the developed AUV code to be run, debugged and tested prior to the first Phoenix deployment. This was an enormous time saver. However the simulator does not provide simulated GPS/DGPS data or DiveTracker ranges.

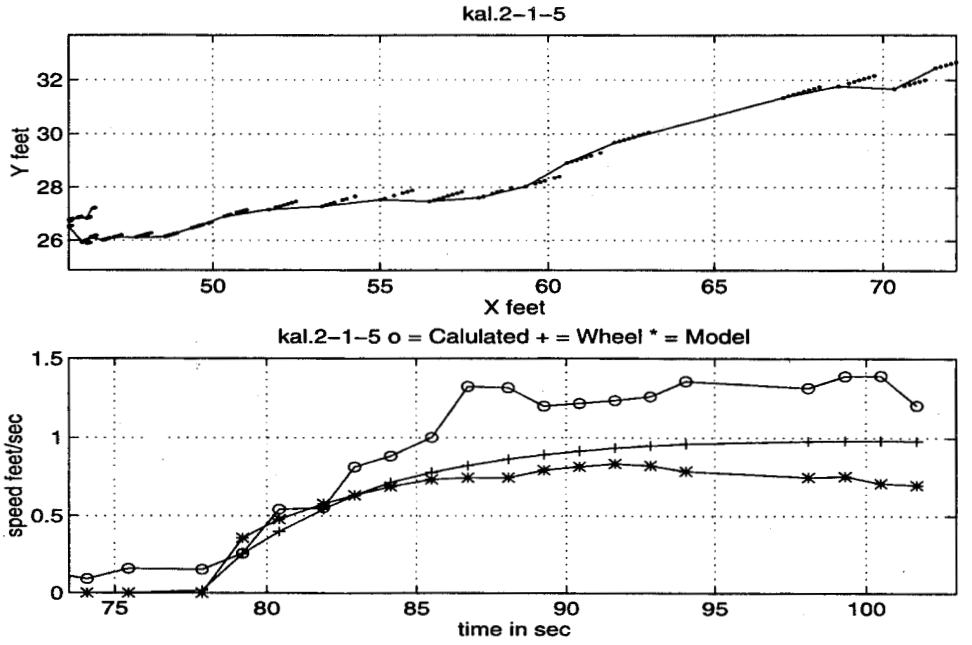

Figure 23: Top: First Straight Line Speed Calibration Run Bottom: Plot of Speed Calibration Run, o = Calculated Speeds Based on Measured Position, + = Speed Wheel Output, * = Mathematical Model Calculated Speeds

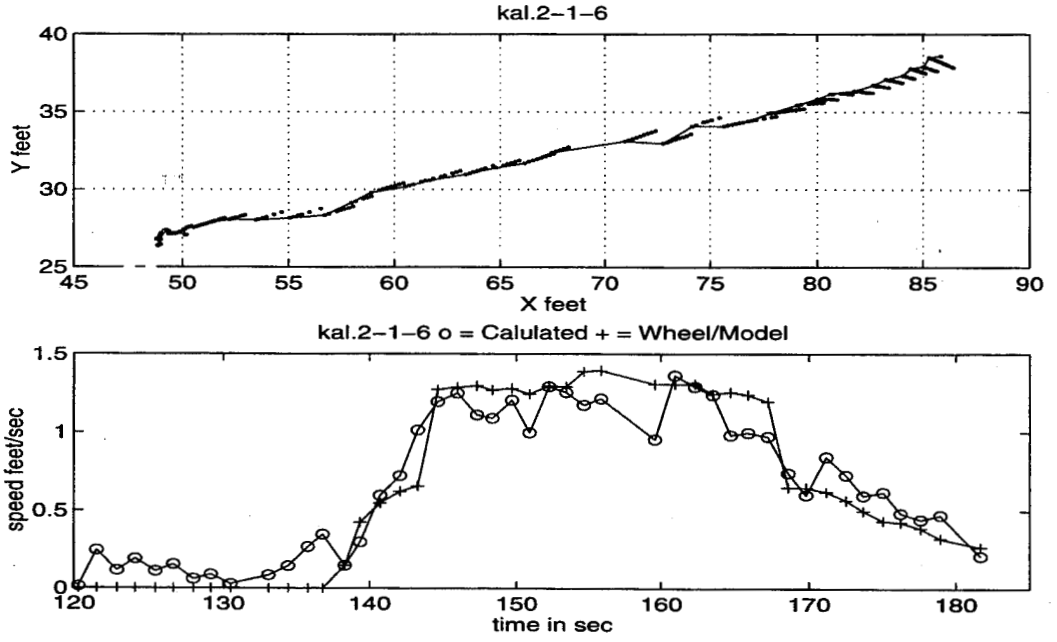

Figure 24: Top: Second Straight Line Speed Calibration Run Bottom: Plot of Speed Calibration Run, o = Calculated Speeds Based on Measured Position, + = Speed Wheel Output

When the simulate flag is set, the Navigation module computes simulated DiveTracker ranges and GPS/DGPS locations. To more accurately simulate the DiveTracker, random uniformly distributed noise with a range of 0.75 feet is placed on calculated Ranges, and the range arrival time is a random variable uniformly distributed from 1 to 4 seconds. The GPS/DGPS is simulated in the DGPS mode with uniformly distributed random noise with a range of 6 feet placed on the positions, with fixes arriving randomly from 1 to 2 seconds. The Kalman filter uses these simulated measurements to track the vehicle in a simulated runs.

M. SUMMARY

The navigation module uses a discrete Kalman filter to process GPS/DGPS and DiveTracker measurements to produce updated estimates of position. The navigation module is a forked process of the tactical level OOD and runs in a continuous loop. The module can use either real or simulated measurements. The module is first initialized, then processes measurements or dead reckons as required. Measurements are examined and the best measurement method available to produce a position estimate is used. A local coordinate system in feet aligned with the earths meridians is used for positioning. The GPS/DGPS and DiveTracker standard deviations are converted to variations and were used in the filter with good results. More work needs to be performed to optimize these variations for best positioning. The navigation module fix determinations were used to calibrate the Phoenix speed sensor to increase the position accuracy while dead reckoning.

V. SOFTWARE

A. INTRODUCTION

The navigation module software is written in C and consists of the following four modules; Navigator1.C, Kalman_Filter.C, ReadGps.C and Matrix.C. These modules define global variables and set function prototypes in the files Kalman_Filter.H, ReadGps.H and Matrix.H. A simulation flag can be set in Kalman_Filter.H which causes simulated measurement to be calculated for test tank or bench testing. In addition, all trouble shooting print statements are wrapped inside 'If' statements that require a local variable 'TRACE' to be set to true before any print statements will be performed.

B. NAVIGATOR1.C

Navigator1.C was the main driver for the code. It receives system state inputs from the tactical level, and GPS data from the Motorola GPS unit. It returns position data to the tactical level and records this data in a file for later analysis. The Navigator1.C module consists of the following sub-routines; Nav1_Initialize, My_Parse_Telemetry_String and Reset_Kalman. This module has function calls to Kalman_Filter.C, ReadGps.C and Matrix.C. The Navigator1.C code is located in Appendix A.

1. Navigation Module Operation

Navigator1.C first declares and initializes the variables and Kalman filter matrices required for operation. These matrices include three Kalman gains (K), three system covariance matrices (Sigma) and three measurement covariances matrixes (R), one for each measurement method (DGPS, DIVETRACKER and GPS). After the declarations, the file

'kal.dat' is opened for data recording. If simulate is FALSE, the GPS unit serial port communications are initialized. This is followed by a function call to Nav1_Initialize which reads inputs from the tactical level to properly set up the Kalman_Filter and navigation parameters. The navigation module then enters an infinite 'For' loop for navigation data processing. Once the loop is entered, the module reads data from the tactical level command and telemetry data pipes. If a 'Quit' command is present, the module terminates. If an 'AUV_STATE' message is present, the filter process commences. If neither 'Auv_State' or 'Quit' has been read, then the loop performs 'busy' cycles and loops until either a Quit command or an AUV_STATE message is received.

The AUV_STATE message provides all vehicle telemetry data to include roll, pitch, azimuth, speeds, thruster rpm, DiveTracker ranges, and a time stamp. If the Simulate flag is set, then the DiveTracker data is overwritten with calculated ranges and the GPS data is simulated. Both simulated measurements have uniform noise added. If DiveTracker data is available, the DiveTracker timer flag is advanced 15 seconds. The Determine_Fix routine is then called which takes the received measurements and returns the proper measurement Fix type to use (0 = No_Fix, 1 = DGPS, 2 = DiveTracker and 3 = GPS), and sets flags denoting what measurements were available.

The IOU velocity model values are now set. If the Fix type was 0, then the state X and Y drift speeds are computed and set, the No_Fix flag is set to TRUE, and the Fix_Type flag is set to the most recent Fix_Type available. This ensures that the dead reckoning to be performed by the Kalman filter movement step will use the state covariances of the last measurement received.

The movement (Φ), movement noise covariance (Q), and mean movement noise (Uw) matrices are now set with the IOU computed values. The local coordinate vehicle speeds u, v and w are now converted using a rotational transformation matrix of Course, Pitch and Roll to produce the Earth X, Y and Z speeds required for navigation.

The Loss_Track flag is next checked. If the vehicle has lost track and a GPS or DGPS fix is available, the Reset Kalman routine is called. The Reset Kalman routine resets the DGPS or GPS system covariance matrix (Sigma) so track can be regained with the new fix data. The Kalman filter can now be called, with the passed parameters depending upon what Fix type was present. This routine returns updated values of the system state (U), updates the appropriate Kalman gain and Sigma matrices, and sets the Loss_Track flag. If the Loss_Track flag was not set, then the Loss_Track timer is advanced thirty seconds. The vehicle speed is now calculated using the state vector drift and vehicle speeds. The calculated speed is used to determine if there is a measured speed error which must be accounted for while dead reckoning.

The Loss_Track and DiveTracker timers are now checked. If the current time exceeds either timer, then the system has lost track for 30 seconds or received no DiveTracker data for 15 seconds. In this case the DiveTracker avail flag is set to 2, which is used to indicate a problem to the tactical level. The tactical level uses this data to determine if the Phoenix should surface to obtain a GPS fix.

Fix data is now sent to the tactical level in the form of the updated system state (U), and the Flags indicating what measurement types were available. This fix data is recorded in a file for analysis and the loop continues.

2. Nav1_Initialize Function

The initialize function reads inputs from the tactical level and sets variables for navigation use. This function runs as a 'Do' loop reading data from the tactical level and only exits normally when all required inputs are received. The required inputs consist of an initial vehicle posture (position and heading), the base DiveTracker transducer locations, a Gyro Error input, and the receipt of a GPS/DGPS fix. If 45 seconds have elapsed and all required inputs were not received, this function reports Initialization Failed to the tactical level. The receipt of a GPS/DGPS fix is used to set origin position globals, all further GPS/DGPS fixes are based on the origin position.

3. My_Parse_Telemetry_String Function

This function parses the command and telemetry strings received through the tactical level pipes into its component data. The receipt of a DiveTracker location command is used to set the DiveTracker transducer location globals. The Gyro Error and Posture commands set their associated globals.

4. Reset_Kalman Function

The reset Kalman function is used to reset the state covariance matrix, Σ, associated with the received measurement. This function resets the covariance matrix to an identity matrix to use as new starting point covariances.

C. KALMAN_FILTER.C

The Kalman_Filter function performed the movement and measurement steps for the discrete Kalman filter in normal and extended modes. This function is called by Navigator1.C and returns the updated system state U, updates the Kalman gains and state

covariances, and sets the system Loss_Track flag. The majority of the function calls used by this routine are matrix operations defined in Matrix.C. Included functions are the Nav_to_Rad and My_Square functions. Kalman_Filter.H code is in Appendix B and Kalman_Filter.C code is presented in Appendix C of this thesis.

1. Kalman_Filter Operation

This function first sets the variables and matrices required uniquely for the movement and measurement steps that were not required for the Navigation module. The movement step is now performed using the matrix operations defined in Matrix.C The measurement step only takes place if No_Fix is set to FALSE. If it is TRUE, then the system state U is returned.

The measurement step first checks if extended or normal filtering is required by checking the Fix_Type. If the Fix_Type indicates DiveTracker measurements will be used, then the extended Kalman filtering steps are taken. If the Fix_Type indicates GPS/DGPS measurements are used, then regular filtering takes place.

Extended filtering entails the calculation of Ranges from the system state estimate U to the DiveTracker base transducer stations to be placed in the f(U) matrix. The first partial derivatives of these calculated ranges are placed in the H matrix and the measured ranges in the Z matrix. For non-extended filtering, the GPS values are just placed in the Z matrix.

The Kalman gains are now calculated. The Shock value method of calculation depends upon if extended or regular filtering is being conducted. The Dimensionless Shock value is now calculated. If Dimensionless Shock exceeds 50, then the Loss_Track flag is set

to true and the measurement steps are not performed, with an end result of ignoring the measurement. Otherwise, the measurement step is performed, updating U and Sigma. In addition the value of Total Drift is calculated which is used by the navigation module when dead reckoning to update the Drift Components. Finally the new value of U is returned to Navigator1.C.

2. Navtorad Function

The NavtoRad function converts vehicle headings in degrees as used for navigation to the proper radian values. This is used for the updating of the ocean current drift/error components while dead reckoning.

3. Mysquare Function

This function performs a simple computation of the square of an input number. It is used to save space and because somewhere in one of the Phoenix software modules the Pow function was over written by other code not compatible with the needs of the navigation software.

D. READGPS.C

The READGPS.C module opens the Voyager (Solaris/Sun) serial port for communications with the Motorola GPS/DGPS. It decodes the Motorola data stream, and contains the Determine Fix routine. This module utilized code from previous work with a six channel Motorola GPS/DGPS for decoding the data stream [BACH95]. The functions described below were included in this module. The ReadGps.H code is located in Appendix D and the ReadGps.C code is in Appendix E.

1. Get_GPS_Data Function

This is the driving function for the GPS/DGPS data reading and decoding. It calls the Gps_Serial_Read function to read the Motorola data. If there is new data read, then the data stream is decoded. If not, then the GPS fix type is set to 0. This function returns the GPS/DGPS data structure.

To fully decode the data stream, first the number of satellites detected was decoded and the message checksum was computed. If there were three or more satellites available and the checksum was valid, then the message was fully decoded. To fully decode the message requires calls to GetMilSec for latitude and longitude information, GetGpsTime for the fix time data, and GetGpsFixType which determines if the fix was computed using GPS or DGPS data.

2. CheckSumCheck Function

This function computes the exclusive OR of bytes 2 through 73 of the Motorola data stream. The XOR'd data is then compared to the data in byte 73. If equal then TRUE is returned.

3. Getmilsec Function

This function extracts the Latitude and Longitude data from the Motorola data stream in milliseconds of arc. Low level bit shifting is required to conduct this operation. Specifically, the data in bytes 15 - 18 are shifted and combined to produce the Latitude value. The same calculation is performed on bytes 19 - 20 to produce the Longitude data.

4. Getgpstime Function

This function works along the lines of GetMilSec, where the data is shifted and manipulated to decode the time values. Bytes 8 and 9 hold the hours and minutes values. Byte 10 holds the integer value for seconds. However, milliseconds are held in bytes 10 - 14 which must be shifted and combined and then added to the value of byte 10 to arrive at the total seconds value. This function returns the time of day in seconds, where hours and minutes are converted to seconds and added to the seconds value. The computed time is not used in the Navigation Module. It is used only for testing and data analysis.

5. Getgpsfixtype Function

This function performs bit level comparison of byte 72 of the data stream to check for DGPS use. A logical AND of Byte 72 and bit stream 0100 is performed to check if bit 3 is set. If set, then the differential signal is used in fix computation.

6. Determine Fix Function

This function inputs were the system measurements, the Loss_Track flag, the state vector U, and the DiveTracker timer variable. These inputs are used to set the Fix_By variable for Navigator1.C and Kalman_Filter.C use. In addition, this function sets flags for determining which measurement types are available. The measurement type available flags are first set by examining the DiveTracker ranges and the GPS fix types. Fix Types are then determined by comparing the GPS/DGPS, the DiveTracker, and the Loss_Track flags. The DiveTracker measurements are used if both DiveTracker and GPS/DGPS are available, unless the DiveTracker Loss_Track flag is set, in which case GPS/DGPS measurements will be used. If the Loss_Track flag is not set, and DiveTracker and GPS/DGPS are available,

then DiveTracker is used, unless the system state position places the vehicle outside the standard deviation of the GPS/DGPS fix. In this case, Phoenix is assumed lost and GPS/DGPS is used. If only a single measurement type is available, then that type is returned. As a last check for stability purposes, if a GPS/DGPS fix is received and there has been a DiveTracker measurement within the last 15 seconds, the GPS/DGPS fix is ignored. This prevents bouncing that may occur between a GPS/DGPS fix and DiveTracker fix when the Phoenix is near the surface.

7. Gps_Serial_Read Function

This function actually reads the raw data message from the Motorola GPS through the Voyager serial port. It uses blocking reads, with a signal alarm timeout to prevent system lockup. This function first sets up the signal handler for the signal alarm timeout system. The function can then perform reads without danger of lockup. The function reads to clear the serial port until it reaches the beginning of a Motorola message. The first 4 bytes of this message are then read in. If they are consistent with a data message header, the rest of the message is read into the raw message data structure and returned. If the bytes indicate a differential signal message, the differential message is read to clear the port.

8. Initialize_Serial Function

This function opens the serial port and sets the necessary flags for the Solaris/Sun port. It also returns the serial port path number after the port has been successfully opened.

9. Open_Tty Function

This function is called by Initialize_Serial to actually open the port. It uses a signal alarm to prevent lockup while opening the port to prevent a system lockup if another process

is using the port (this will not happen in the Phoenix application as the Navigator is the only process that uses the port).

10. Tty and Serial_Read Timeout Functions

These functions are used as alarm handlers for the Open_tty and the Read_serial functions. Each simply provides an error message indicating a timeout has occurred. The Serial_Read_Timeout also sets the TIMEOUT flag to true for use in the Serial Read function.

11. Simulate_GPS_Data Function

If the simulate flag is set in the Navigator1.C function, then this function is called to provide simulated GPS data. This function's inputs are the current vehicle X and Y position. Ten feet of uniformly distributed random noise is placed on this position, and the position is converted to milliseconds of arc. The new simulated fix position is now returned in the GPS fix structure.

E. MATRIX.C

The file Matrix.C provides all the required matrix operations used by the Kalman filter. Matrix.H sets the matrix data structure. The matrix data structure consist of a double 4 X 4 array with row and column place holders. For example, the state vector U is a 4 X 1 vector, the data structure for U is a double 4 X 4 with row set to 4 and columns set to 1. All matrix operations index through the input structure row and columns as set in their data structure. The Matrix.H code is located in Appendix F and the Matrix.C code is in Appendix G.

1. Matrix_Multiply Function

This function receives two matrices and returns their product. The returned matrix will have the proper row and column values set.

2. Matrix_Add and Subtract Functions

These functions perform element by element addition and subtraction of two input matrices and return the result.

3. Matrix_Transpose Function

This function simply indexes through a matrix and returns its transpose.

4. Matrix_Inverse Function

This function returns the inverse of an input matrix by performing Gausian elimination. The premise is that any matrix multiplied by its inverse results in an identity matrix. This simplifies to the Gausian elimination problem of $AX = I$, where A is the input matrix, I is the identity matrix and X is the inverse to be solved for. The function first creates an identity matrix of the input matrix size. The identity matrix and the input matrix are now used as inputs for the Gauss_Elimination function. The inverse matrix is returned upon completion of the Gausian elimination.

5. Gauss_Elimination Function

The Gauss_Elimination function takes as inputs a matrix to be inverted and an identity matrix of the same size. This function first concatenates the input matrix and the identity matrix together. Row eliminations are performed on the new concatenated matrix, followed by back substitution. The resulting answer is the desired inverse.

6. Matrix_Rtransform Function

This function constructs a rotation matrix for use in transforming body coordinates into Earth coordinates. The inputs for this function are vehicle roll, pitch and azimuth. The output is the required rotation matrix.

7. Output_Matrix Function

This function is used to print the contents of an input matrix. The output is printed in row, column form.

F. SUMMARY

The navigation module is written in 'C' and consists of the following functions: Navigator1.C, Kalman_Filter.C, Kalman_Filter.H, ReadGps.C, ReadGps.H, Matrix.C and Matrix.H.

Navigator1.C is the main driver for the Phoenix navigation module, it initializes the navigation system, reads and writes data from the tactical level OOD, calls the Kalman_Filter functions, and records data for later analysis. Kalman_Filter.C performs the Kalman filter movement and measurement to produce a new state position estimate. It determines if a measurement was bad by using Dimensionless Shock and sets a Loss_Track flag.

ReadGps.C and Matrix.C are support functions for the navigation module. ReadGps.C opens the Voyager serial port, reads and decodes GPS/DGPS data. It also contains the function that determines the best available measurement to use to compute a position. Matrix.C functions perform all matrix mathematics required by a Kalman filter. It includes addition, subtraction, multiplication, inverse and transpose. It also computes a rotation matrix to resolve body coordinates to Earth coordinates.

VI. SUMMARY AND CONCLUSIONS

A. SUMMARY

The overall conclusion of this thesis is that the method of Discrete Asynchronous Kalman Filtering used in this implementation can provide accurate real time vehicle positioning. To produce a common vehicle position state using different unrelated measurements requires accurate data of the measurement systems, variances, and accurate speed inputs. If any of the variances are wrong, or the input speeds are incorrect, the resulting error grows fast and is hard to isolate. In this implementation, the GPS measurements did not follow the Gaussian distribution required by Kalman filtering, and yet produced excellent results.

The use of a simulator greatly reduces the development time of software. The graphical simulator allowed for troubleshooting and tests that otherwise would have had to be performed in the vehicle, in the water. However, as discovered during actual trials, the real world is much noisier and more complicated than the clean simulated version of reality. Trials that had been performed numerous times in the simulator did not work in reality. Problems with the speed wheel calibration and vehicle control constants were readily visible during actual testing. Only real world runs revealed these problems, they never appeared in the Simulator.

The combination of local coordinates and global coordinates requires accurate positioning. This was especially true for the GPS and DiveTracker measurement systems. To convert the Global position to the local position is error prone. These errors become very visible when shifting from one form of measurement to another.

The use of manufacturer document's are not always complete or correct. This was found while attempting to read from the Solaris/Sun serial ports.

B. FUTURE WORK

The addition of an IMU to provide accurate vehicle accelerations would greatly increase the value of the motion model. Measured accelerations could be added to the Kalman Filter process and be used to provide speed inputs. This addition would result in the filter operating as an Extended Kalman Filter at all times. The system state, movement, and measurement matrices would have to be modified to account for the additional inputs. The motion and measurement models would also have to be changed to reflect the use of accelerations.

The virtual world simulator needs to be modified to fully simulate both DiveTracker and GPS data. In addition, the effect of simulated random noise on these simulated measurements combined with updating the simulated vehicle position based on fix data causes a random walk behavior of the simulated vehicle. At present the only solution to this is having no noise on the simulated measurements. To totally simulate the effects of the navigation module tracking would require the addition of a stable reference vehicle that would not be updated with the solved for fix information.

Matrix multiplication analysis should be performed on the multiple series matrix multiplications used in the Kalman Filter. This may lead to a reduction of the multiplications required and consequently reduce the computation time.

More work with Differential GPS receiver is required. A DGPS fix was never obtained on the water. This was due to high electromagnetic interference cause primarily by

the Phoenix gyros. Better shielding of the receiver and use of a double shielded coax antenna cable are possible solutions to this problem.

C. CONCLUSION

Kalman filtering provides an accurate method of solving the navigation problem. This thesis has proved that two dissimilar measurement systems can be combined and used in one module to produce excellent results. It is important to note that the results produced by this software are useable in 'real-time'. This is critical for close maneuvering where late data can cause serious problems.

APPENDIX A. NAVIGATOR1.C

```
/*************************************************************************
  FILENAME:       navigator1.c

  AUTHOR:         Dave McClarin

  DATE:           8 March 1996

  PURPOSE:  Performs kalman_filtering of dive_tracker, Gps and Dgps
            data to create a valid state estimate of location, and
            the ocean current estimates, initializes and resets the
            Kalman_filter if required

  FUNCTIONS:  navigator1()
        nav1_initialize()
        my_parse_telemetry_string()
        reset_kalman()
*************************************************************************/

#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include "matrix.h"
#include "readgps.h"
#include "kalman_filter.h"
#include "../execution/globals.h"
#include "../execution/defines.h"
#include "../execution/statevector.h"

/*************************************************************************
  FUNCTION:       navigator1.c

  AUTHOR:         Dave McClarin

  DATE:           5 March 1996

  PURPOSE:  Performs kalman_filtering of dive_tracker, Gps and Dgps
            data to create a valid state estimate of location, and
            the ocean current estimates.
```

```
  RETURNS: none, sends nav data through socket comms to calling
           function. (tactical.c)
*****************************************************************/
void navigator1(){

/* defines external pipes for socket coms to and from tactical.c  */
extern  int Nav1_to_OOD_fd[2],OOD_to_Nav1_fd[2];
extern  int Nav1_telemetry_fd[2];
extern  int Simulate;

/* strings used for socket coms */
char Nav_Dat[MAXBUFFERSIZE];
char Nav_Read_to_clear[MAXBUFFERSIZE];
char Nav_String_read[MAXBUFFERSIZE];

/* command string for gps initialization */
char Gpscmd_8[20] = {'@','@','E','a',1,25,13,10};

int Fix_Type;                 /* holds fix type flag */
int No_Fix = FALSE;           /* Current Fix Flag status */
int Most_Recent_Fix = 1;      /* Holds last fix type, init to DGPS */
int TRACE = FALSE;            /* flag for trouble-shooting printf's */
int Path;                     /* holds file Id for serial port opening*/
int ti = 0;

/* fix status flags */
int Dt_Avail = 0;
int Fix_By = 0;
int Dgps_Avail = 0;
int Gps_Avail = 0;
int Loss_Track = 0;
int Fix_Concur = 0;

double Del;                   /* time between New AUV-States received */
double Gps_X, Gps_Y;          /* Gps Posits in X,Y feet */
double Dt_X, Dt_Y;            /* Dive-Track Posits in X,Y feet */
double Cos_Lat;               /* cosine of present latitude */
double Qns,Qew,Qlat,Qlon;     /* lat/lon and NS/EW drift covariance factors*/
double C;                     /* shrinkage factor for IOU velocity model */
double Gamma;                 /* velocity multiplier for IOU velocity model*/
double Calc_Speed;            /* calculated total speed */
double Old_Time;              /* time of last auv-state */
```

```
double Pitch,Roll,Course;     /* degree versions of theta, phi and psi */
double Last_R1 = 0.0;         /* holder for last divetrack 1 range recieved*/
double Last_R2 = 0.0;         /* holder for last divetrack 2 range recieved*/
double Speed_Sign = 1.0;      /* used for Dead Reckoning */
double DriftNS = 0.0;         /* NS drift speed in Feet/second */
double DriftEW = 0.0;         /* EW drift speed in Feet/second */
double Total_Drift = 0.0;     /* Total drift speed in Feet/second */
double DtSim_Time = 0.0;      /* Used to simulate divetracker arrival time */
double GpsSim_Time = 0.0;     /* Used to simulate Gps arrival times */
double DT_Timer =99999.0;     /* time in secs until a DT fix timeout*/
double LT_Timer = 99999.0;    /* time in secs until a Loss of Track timeout*/
double DSim_Error = 0.0;      /* Sets a holder for DT range error simulation */
double FGps_X, Fgps_Y;        /* filtered Gps Positions */

gps Gps_Fix;                  /* structure for GPS fix data */
transponder Dt1, Dt2;         /* structures for Dive Tracker data */

matrix  K,Kdt,Kg;             /* matrix's for kalman gains Dgps,dt and Gps */

/* Matrix for speed vector Y speed, X speed and Z speed */
matrix  Speed_Vect = {{{0.0} ,{0.0}, {0.0}},3,1};

/* Matrix for holding state values X, Y, XDrift, YDrift */
matrix  U =  {{{0.0}, {0.0}, {0.0}, {0.0}},4,1};

/* means of movement noise (state values) */
matrix  Uw = {{{0.0}, {0.0}, {0.0}, {0.0}},4,1};

/* movement matrix, how state changes between measurements */
matrix  Phi ={{{1.0, 0.0, 0.0, 0.0},
          {0.0, 1.0, 0.0, 0.0},
          {0.0, 0.0, 0.0, 0.0},
          {0.0, 0.0, 0.0, 0.0}},4,4};

/* covariance matrix of movement noise */
matrix  Q =  {{{0.0, 0.0, 0.0, 0.0},
          {0.0, 0.0, 0.0, 0.0},
          {0.0, 0.0, 0.0, 0.0},
          {0.0, 0.0, 0.0, 0.0}},4,4};
```

```
/* covariance of DGPS, DiveTrack and Gps system state, */
matrix  Sigma = {{{1.0, 0.0, 0.0, 0.0},
                  {0.0, 1.0, 0.0, 0.0},
                  {0.0, 0.0, 1.0, 0.0},
                  {0.0, 0.0, 0.0, 1.0}},4,4};

matrix  Sigma_dt = {{{1.0, 0.0, 0.0, 0.0},
                     {0.0, 1.0, 0.0, 0.0},
                     {0.0, 0.0, 1.0, 0.0},
                     {0.0, 0.0, 0.0, 1.0}},4,4};

matrix  Sigma_g = {{{1.0, 0.0, 0.0, 0.0},
                    {0.0, 1.0, 0.0, 0.0},
                    {0.0, 0.0, 1.0, 0.0},
                    {0.0, 0.0, 0.0, 1.0}},4,4};

/* covariance of Dgps, DiveTrack and Gps measurement noise */
matrix  R  = {{{Dgps_Lat_Var,   0.0},
               {0.0,   Dgps_Lon_Var}},2,2};

matrix  Rdt  = {{{DvTrk_R1_Var,   0.0},
                 {0.0,   DvTrk_R2_Var}},2,2};

matrix  Rg  = {{{Gps_Lat_Var,   0.0},
                {0.0, Gps_Lon_Var}},2,2};

/* file handling for data recording */
FILE *Fw;
Fw = fopen("kal.dat","w");

/* if in Simulate Mode, do not perform any serial port comms. */
 if (!Simulate){
   Path = initialize_serial();
   if (write(Path,Gpscmd_8,8) != 8)
         printf("Gpscmd write error \n");
         printf("GPS initialized\n");
}

/* initialize filter, sets all globals from kalman_filter.h and
   those imported from ../execution/statevector.h"*/
nav1_initialize(Path,Simulate);
Dt1.Range = divetracker_range1;
Dt2.Range = divetracker_range2;
```

```
Dt1.Xloc = DT1X;
Dt1.Yloc = DT1Y;
Dt1.Zloc = DT1Z;
Dt2.Xloc = DT2X;
Dt2.Yloc = DT2Y;
Dt2.Zloc = DT2Z;

/* gets initial fix and sets Gps_X and Y */
if (Simulate){
  /* converts x y to milliseconds of arc and sets fix
  type to 1 (dgps) */
  Gps_Fix = simulate_gps_data(x,y,1);
}else
  Gps_Fix = get_gps_data(Path);

/* converts milliseconds of arc to feet X, Y */
Cos_Lat = cos(DegToRad(Gps_Fix.lat/3600000));
Gps_X = (Feet_Conv * (Gps_Fix.lat - Orig_Lat)) + Posture_X;
Gps_Y = (Feet_Conv * (Gps_Fix.lon - Orig_Lon) * Cos_Lat)
                              + Posture_Y;

/* converts Degrees inputs to radians */
Course = NavtoRad(psi,Init_Heading);

/* sets initial values for startup */
Old_Time = t;
U.m[0][0] = Posture_X;
U.m[1][0] = Posture_Y;
Dt_X = Posture_X;
Dt_Y = Posture_Y;
FGps_Y = Posture_Y;
FGps_X = Posture_X;

for (;;){

  /* Reads OOD command Strings via socket, Do nothing if no input */
  if (read(OOD_to_Nav1_fd[0],Nav_String_read,MAXBUFFERSIZE)== -1){}
  else{
    if (strcmp (Nav_String_read,"QUIT") == 0){
      printf("Terminating Navigator1 Module \n");
      exit(0);
    }
    else
```

```
    my_parse_telemetry_string(Nav_String_read);
}
/* Reads the telemetry string, if AUV_STATE the process data,
  otherwise skip everything 'busy wait' */
if (read(Nav1_telemetry_fd[0],Nav_String_read,MAXBUFFERSIZE) == -1){}
else{
    my_parse_telemetry_string(Nav_String_read);

    if (strcmp (keyword,"AUV_STATE") == 0){

    /* trouble-shooting only */
    if (TRACE)
      printf("D1X %lf D1Y %lf D1z %lf D2X %lf D2Y %lf D2Z %lf\n",
      Dt1.Xloc, Dt1.Yloc, Dt1.Zloc, Dt2.Xloc, Dt2.Yloc, Dt2.Zloc);

    /* just for gps testing only, so only non-moving
      gps tracking data is produced.
    divetracker_range1 = -1.0;
    divetracker_range2 = -1.0;
    z = 0.0;
    speed = 0.0;
    u = 0.0;
    v = 0.0;
    */

    /* produces simulated Dt ranges with some noise for
      bench testing */
    if ((Simulate) && (DtSim_Time < t) && (z > .60)){
      DSim_Error = 0.0 * drand48();
      if (DSim_Error > 0.5) DSim_Error *= -1.0;
      Dt1.Range = sqrt(my_square(Dt1.Xloc - x)+
                  my_square(Dt1.Yloc - y)+
                    my_square(Dt1.Zloc - z)) + DSim_Error;

      DSim_Error = 0.0 * drand48();
      if (DSim_Error > 0.5) DSim_Error *= -1.0;

      Dt2.Range = sqrt(my_square(Dt2.Xloc - x)+
                  my_square(Dt2.Yloc - y) +
                   my_square(Dt2.Zloc - z)) + DSim_Error;

        DtSim_Time = 4.0*drand48() + t;
    }
```

```
/* this is required because the virtual world and the
   real execution level don't work the same */
else if ((Simulate) && (DtSim_Time > t)){
   Dt1.Range = -1.0;
   Dt2.Range = -1.0;
}
/* dive tracking ranges are only valid if they change
   because execution sends last data until new data
   comes in */
else if ((!Simulate) &&
          ((Last_R1 != divetracker_range1) &&
                (Last_R2 != divetracker_range2))){
   Dt1.Range = divetracker_range1;
   Dt2.Range = divetracker_range2;
}
else{
   Dt1.Range = -1.0;
   Dt2.Range = -1.0;
}
Last_R1 = divetracker_range1;
Last_R2 = divetracker_range2;

/* sets DT_Timer to t + 15 sec if there was valid divetrack
   data, or depth was less than 1 ft*/
if (((Dt1.Range >= 0.0) && (Dt2.Range >= 0.0)) || (z <= 1.0))
   DT_Timer = t + 15.0;

/* get gps fix  data*/
if (z <= 1.0){
   /* if in simulate provides a fix about 1 per second */
   if ((Simulate) && (GpsSim_Time < t)){
      /* converts x y to milliseconds of arc and sets fix
         type to 1 (dgps) */
      Gps_Fix = simulate_gps_data(x,y,1);
      GpsSim_Time = t + 1.5*drand48();
   }
   else if ((Simulate) && (GpsSim_Time > t))
      Gps_Fix.type = 0;
   else
      Gps_Fix = get_gps_data(Path);

   /* converts milliseconds of arc to feet X, Y */
   Cos_Lat = cos(DegToRad(Gps_Fix.lat/3600000));
```

```
  Gps_X = (Feet_Conv * (Gps_Fix.lat - Orig_Lat)) + Posture_X;
  Gps_Y = (Feet_Conv * (Gps_Fix.lon - Orig_Lon) * Cos_Lat)
                                    + Posture_Y;
  /* for troubleshooting only */
  if (TRACE){
    printf("GPSX = %lf  GPSY = %lf \n",
                        Gps_X,Gps_Y);
    printf("ORIG_LAT = %lf ORIG_LON = %lf \n",
                        Orig_Lat,Orig_Lon);
    printf("Gps_Lat = %lf, GPS_Lon = %lf \n",
                        Gps_Fix.lat,Gps_Fix.lon);
    printf("PostureX = %lf PostureY = %lf\n",
                        Posture_X, Posture_Y);
  }
}
else{
  /* no gps available due to depth greater than 1 foot */
  Gps_Fix.type = 0;
}

/* determines fix type based on ranges, and gps fix data
  also sets fix avail flags */
Fix_Type = determine_fix(Dt1.Range, Dt2.Range, Gps_Fix,
                &Gps_Avail, &Dgps_Avail,
                &Dt_Avail, &Fix_By, &Fix_Concur,
                U, Gps_X, Gps_Y, Loss_Track,
                t, DT_Timer);

/* for troubleshooting only */
if (TRACE){
  printf("X %lf GPS_X %lf GPS_Lat %lf Orig Lat %lf\n",
          x,Gps_X,Gps_Fix.lat,Orig_Lat);
  printf("Y %lf GPS_Y %lf GPS_Lon %lf Orig_Lon %lf \n",
          y,Gps_Y,Gps_Fix.lon, Orig_Lon);
}
/* sets the time between received auv-states */
Del = t - Old_Time;
Old_Time = t;

/* sets the shrinkage factor and velocity multiplier for
IOU velocity model */
C = exp(-(Del/Tau));
Gamma = Tau * (1.0-C);
```

```
/* fix type = 0 then updates NS/EW drift number for any
   new course, also sets no-fix flag, note.. fix_type
   is set to the previous type for movement-step
   considerations*/
if (Fix_Type == 0){
   Fix_Type = Most_Recent_Fix;
   No_Fix = TRUE;
   U.m[2][0] = Total_Drift * (Speed_Sign) * sin(Course);
   U.m[3][0] = Total_Drift * (Speed_Sign) * cos(Course);
}
else
   No_Fix = FALSE;

Most_Recent_Fix = Fix_Type;

/* sets movement noise covariance factors based on
   the IOU velocity model Qlat/lon 1 =  (1 ft/sec)^2
   variance for speed */
Qns = DriftNS*DriftNS*(1.0-C*C);
Qew = DriftEW*DriftEW*(1.0-C*C);
Qlat = Gamma * Gamma * 1.0;
Qlon = Gamma * Gamma * 1.0;
Q.m[0][0] = Qlat;
Q.m[1][1] = Qlon;
Q.m[2][2] = Qns;
Q.m[3][3] = Qew;

/* sets movement matrix based on the IOU velocity model*/
Phi.m[0][2] = Gamma;
Phi.m[1][3] = Gamma;
Phi.m[2][2] = C;
Phi.m[3][3] = C;

/* sets the mean movement noise */
Uw.m[0][0] = Speed_Vect.m[0][0] * Gamma;
Uw.m[1][0] = Speed_Vect.m[1][0] * Gamma;

/* converts degrees in theta, phi, and psi to radians */
Pitch = DegToRad(theta);
Roll = DegToRad(phi);
Course = NavtoRad(psi,Init_Heading);
```

```
if (Simulate)
  speed = u;

/* uses u, v. w for speeds if thrusters are on,
  otherwise uses speed, v w. */
if ((fabs(AUV_bow_vertical) + fabs(AUV_bow_lateral)
    + fabs(AUV_stern_vertical) +
                fabs(AUV_stern_lateral)) > 0.0){
  Speed_Vect.m[0][0] = u;
  Speed_Vect.m[1][0] = v;
  Speed_Vect.m[2][0] = w;
}
else{
  Speed_Vect.m[0][0] = speed;
  Speed_Vect.m[1][0] = v;
  Speed_Vect.m[2][0] = w;
}
/* converts the speed vector from local to earth co-ords,
  result speed vect[1][0]=Y(E/W), [0][0] = X(N/S)
  and [2][0] = z */
Speed_Vect = matrix_multiply(rtransform(DegToRad(psi),Pitch,Roll),
          Speed_Vect);

/* Resets the Kalman_Filter if There is a Loss track condition
      and a Dgps or Gps fix available */
if ((Loss_Track) && (Fix_By == 1))
  reset_kalman(&Sigma);
else if ((Loss_Track) && (Fix_By == 3))
  reset_kalman(&Sigma_g);

/* decides which type of filtering extended or normal to perform
  and what parameters to use based on fix type */
switch (Fix_Type){
  /* case 1 = DGPS Fix, normal filter */
  case 1 : U =  kalman_filter(Q, U, No_Fix, &Total_Drift,
                    &Sigma, Phi, R, &K, Uw, Dt1,
                    Dt2, Fix_Type, &Loss_Track,
                    Gps_X, Gps_Y,Course);
       break;
```

```
  /* case 2 = Dive Track Fix, extended filter*/
  case 2 : U = kalman_filter(Q, U, No_Fix, &Total_Drift,
                    &Sigma_dt, Phi,Rdt, &Kdt, Uw, Dt1,
                    Dt2, Fix_Type, &Loss_Track,
                    Gps_X, Gps_Y,Course);
        break;

  /* case 3 = GPS Fix normal filter*/
  case 3 : U = kalman_filter(Q, U, No_Fix, &Total_Drift,
                    &Sigma_g, Phi, Rg, &Kg, Uw, Dt1,
                    Dt2, Fix_Type, &Loss_Track,
                    Gps_X, Gps_Y,Course);
        break;
}

/* sets loss track timer to t + 30 if no loss of track */
if (Loss_Track == FALSE)
  LT_Timer = t + 30.0;

/* keeps track of calculated speed to set the speed sign
  used in dead-reckoning */
Calc_Speed = sqrt(my_square(Speed_Vect.m[0][0]+U.m[2][0])+
          my_square(Speed_Vect.m[1][0]+U.m[3][0]));
if (TRACE)
  printf("calc_speed = %lf speed = %lf\n",Calc_Speed, speed);

if (Calc_Speed > speed)
  Speed_Sign = 1.0; else Speed_Sign = -1.0;

/* updates the drift speed after filtering */
DriftNS = U.m[2][0];
DriftEW = U.m[3][0];

/* if t > Dt_Timer then there was no divetrack avail for 15
  seconds or if t > LT_Timer there was loss of track for
  30 seconds and Dt_Avail is set to 2 to indicate a problem*/
if ((t  > DT_Timer) || (t > LT_Timer)) Dt_Avail = 2;

/* OUTPUT TO TACTICAL - "FIX X Y Z NSdrift EWdrift Dt_Avail
  Dgps_Avail Gps_Avail Loss_Track" */
sprintf(Nav_Dat,"FIX %lf %lf %lf %lf %lf %d %d %d %d\n",
    U.m[0][0], U.m[1][0], z, U.m[2][0], U.m[3][0], Dt_Avail,
    Dgps_Avail, Gps_Avail, Loss_Track);
```

```
        if (TRACE)
          printf("FIX %lf %lf %lf %lf %lf %lf %d %d %d %d %d %d\n",t,
            U.m[0][0], U.m[1][0], z, U.m[2][0], U.m[3][0], Dt_Avail,
            Dgps_Avail,Gps_Avail, Loss_Track, Fix_By, Fix_Concur);

        /* writes fix data to socket to OOD level */
        if (Dt_Avail && (!Loss_Track)){
            if (TRACE)printf("%d nav1 fix time = %lf\n",ti++,t);
          write(Nav1_to_OOD_fd[1],Nav_Dat,MAXBUFFERSIZE);
        }
        else if (Dgps_Avail || Gps_Avail){
          write(Nav1_to_OOD_fd[1],Nav_Dat,MAXBUFFERSIZE);
        }
           else if (Dt_Avail == 2)
          write(Nav1_to_OOD_fd[1],Nav_Dat,MAXBUFFERSIZE);

        /*sets variable FGps_X and Fgps_Y for data recording purposes */
        if ( (((Dgps_Avail) && (Fix_By == 1))
              || ((Gps_Avail) && (Fix_By == 3))) && (!Loss_Track)){
          FGps_X = U.m[0][0];
          FGps_Y = U.m[1][0];
        }
        /*sets variable FGps_X and FGps_Y for data recording purposes */
        if ((Dt_Avail) && (Fix_By == 2) && (!Loss_Track)){
          Dt_X = U.m[0][0];
          Dt_Y = U.m[1][0];
        }
        /* data recording information */
        fprintf(Fw,"%lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf %lf",
              t,x, y, z,
              U.m[0][0], U.m[1][0], U.m[2][0], U.m[3][0],
              Gps_Fix.lat, Gps_Fix.lon, FGps_X, FGps_Y);
        fprintf(Fw," %lf %lf %lf %lf %d %d %d %d %d %d \n",
              Dt1.Range, Dt2.Range, Dt_X, Dt_Y,
              Dt_Avail, Dgps_Avail, Gps_Avail, Loss_Track,
              Fix_By, Fix_Concur);
      }
    }
  }
fclose(Fw);
close(Path);
exit(0);
}
```

```
/*************************************************************************
  FUNCTION:  nav1_initialize()

  AUTHOR:    Dave McClarin

  DATE:      9 February 1996

  PURPOSE:   Reads initialization data from OOD via socket and set
             the appropriate data to initialize the filter

  RETURNS:   none,
*************************************************************************/
void nav1_initialize(int Path, int Simulate){

/* data strings for socket comms */
char Nav_String_read[MAXBUFFERSIZE];
char Nav_Read_to_clear[MAXBUFFERSIZE];

int Init_Posture = FALSE;     /* init flag */
int Dive1 = FALSE;            /* init flag */
int Dive2 = FALSE;            /* init flag */
int Gps  = TRUE;              /* init flag */
int Time_OK = FALSE;          /* init flag */
int Gyro_Error = FALSE;       /* init flag */
int Continue = TRUE;          /* init flag */
int Dgps = FALSE;             /* init flag */
int TRACE = FALSE;            /* init flag */
int Nav1_Time_Out = FALSE;    /* init flag */

gps First_Fix;                /* first fix obtained */

/* used for bench testing */
if (Simulate){
  srand48;
  First_Fix.lat = Orig_Lat;
  First_Fix.lon = Orig_Lon;
  First_Fix.type = 1;
}
```

```
/* loops until all init flags are true */
do{

  /* gets a gps fix */
  if (!Simulate){
    First_Fix = get_gps_data(Path);
  }
  /* Reads the telemetry string from the tactical socket*/
  if (read(Nav1_telemetry_fd[0],Nav_String_read,MAXBUFFERSIZE)== -1){}
  else{
    my_parse_telemetry_string(Nav_String_read);
    if (t > 15.0){
      printf("\n Nav 1 init time = %lf\n",t);
      Time_OK = TRUE;
    if (t > 45.0)
      Nav1_Time_Out = TRUE;
    }
  }
  /* reads the command strings from the tactical socket and
    parsed commands for initialization */
  if (read(OOD_to_Nav1_fd[0],Nav_String_read,MAXBUFFERSIZE) == -1){}
  else{
    if (strcmp (Nav_String_read,"QUIT") == 0){
      printf("Terminating Navigator1 Module \n");
      exit(0);
    }
    else
      my_parse_telemetry_string(Nav_String_read);

    /* sets flags as keywords are recieved */
    if (strcmp(keyword,"POSTURE") == 0)
      Init_Posture = TRUE;
    if (strcmp(keyword,"DIVE-TRACKER1") == 0)
      Dive1 = TRUE;
    if (strcmp(keyword,"DIVE-TRACKER2") == 0)
      Dive2 = TRUE;
    if (strcmp(keyword,"GYRO-ERROR") == 0)
      Gyro_Error = TRUE;

    /* sets orig_lat and lon from gps fixes, preference is
      to Dgps fix */
    if ((First_Fix.type == 1) || (First_Fix.type == 3)){
        Gps = TRUE;
```

```
        if (First_Fix.type == 1)
                Dgps = TRUE;

        /* will only update orig_lat/lon if fix is Dgps
          or if no Dgps was ever recieved */
        if ((Dgps) && (First_Fix.type == 1)){
          Orig_Lat = First_Fix.lat;
          Orig_Lon = First_Fix.lon;
        }
        else if (!Dgps){
          Orig_Lat = First_Fix.lat;
          Orig_Lon = First_Fix.lon;
        }
      }
    }

    /* sets loop exit flag */
    if ((Dive1) && (Dive2) && (Gps) &&
        (Gyro_Error) && (Init_Posture) && (Time_OK)){
            Continue = FALSE;
    }
  }while((Continue) && (!Nav1_Time_Out));

  /* sends init success to OOD */
  if ((Time_OK) && (!Nav1_Time_Out)){
    read(Nav1_to_OOD_fd[0],Nav_Read_to_clear,MAXBUFFERSIZE);
    write(Nav1_to_OOD_fd[1],"NAV1_INITIALIZED",MAXBUFFERSIZE);
    printf("NAV1 INITIALIZED **********\n\n");
  }
  else if (!Gyro_Error){
      printf("WARNING: Gyro_Error not received, assume 0 \n");
      Gyro_Error = TRUE;
      read(Nav1_to_OOD_fd[0],Nav_Read_to_clear,MAXBUFFERSIZE);
      write(Nav1_to_OOD_fd[1],"NAV1_INITIALIZED",MAXBUFFERSIZE);
      printf("NAV1 INITIALIZED **********\n\n");
      }
  else{
    /* sends init fail to OOD and outputs reason */
    read(Nav1_to_OOD_fd[0],Nav_Read_to_clear,MAXBUFFERSIZE);
    printf("NAV1 INITIALIZATION FAILED ON TIME OUT BECAUSE");
    if (!Gps)
      printf(" Gps not received \n");
    if (!Dive1)
```

```
    printf(" Dive Track 1 position not received \n");
  if (!Dive2)
    printf(" Dive Track 2 position not received \n");
 if (!Init_Posture)
    printf(" Posture not received \n");
  write(Nav1_to_OOD_fd[1],"Nav1_Init_Fail",MAXBUFFERSIZE);
}
}

/*************************************************************************
  FUNCTION:  my_parse_telemetry_string

  AUTHOR:    Dave McClarin

  DATE:      8 February 1996

  PURPOSE:   Parses out data obtained in telemetry and command
             strings from the OOD.

  RETURNS:   But does set the globals DT1 and DT2 X Y & Z
             and gyro_error
*************************************************************************/
void my_parse_telemetry_string(char Nav_String_read[MAXBUFFERSIZE]){

/* used for keyword uppercase conversions */
int index;
char  lower_key[MAXBUFFERSIZE];

/* parses keyword out of nav_string and converts it to upper case */
sscanf(Nav_String_read,"%s",keyword);
for (index = 0; index <= (int) strlen (keyword); index++)
  keyword [index] = toupper (keyword [index]);

/* parses dive-tracker1 command and set location globals */
if (strcmp(keyword,"DIVE-TRACKER1") == 0){
  sscanf(Nav_String_read,"%s %lf %lf %lf",lower_key,&DT1X,&DT1Y,&DT1Z);
}
/* parses dive-tracker2 command and set location globals */
else if (strcmp(keyword,"DIVE-TRACKER2") == 0){
  sscanf(Nav_String_read,"%s %lf %lf %lf",lower_key,&DT2X,&DT2Y,&DT2Z);
}
```

```
/* parses gyro-error command and sets global */
else if (strcmp(keyword,"GYRO-ERROR") == 0){
  sscanf(Nav_String_read,"%s %lf",lower_key,&Init_Heading);
}
/* parses posture command and sets globals */
else if (strcmp(keyword,"POSTURE") == 0){
  sscanf(Nav_String_read,"%s %lf %lf",lower_key,&Posture_X,&Posture_Y);
}
else{
  /* must be a telemetry string or something I don't care about
    so use an external parse function */
  parse_telemetry_string(Nav_String_read);
  for (index = 0; index <= (int) strlen (keyword); index++)
      keyword [index] = toupper (keyword [index]);
}
}


/*************************************************************************
  FUNCTION:       reset_kalman()

  AUTHOR:         Dave McClarin

  DATE:           8 February 1996

  PURPOSE:        Resets the kalman filter when called with gps data

  RETURNS:        None, but resets the Sigma, State Covariance Matrix
                  passed in.
*************************************************************************/
void reset_kalman(matrix *Sigma){

if (TRACE) printf("** In RESET KALMAN **\n");

Sigma->m[0][0] = 1.0;
Sigma->m[1][0] = 0.0;
Sigma->m[2][0] = 0.0;
Sigma->m[3][0] = 0.0;

Sigma->m[0][1] = 0.0;
Sigma->m[1][1] = 1.0;
Sigma->m[2][1] = 0.0;
Sigma->m[3][1] = 0.0;
```

```
Sigma->m[0][2] = 0.0;
Sigma->m[1][2] = 0.0;
Sigma->m[2][2] = 1.0;
Sigma->m[3][2] = 0.0;

Sigma->m[0][3] = 0.0;
Sigma->m[1][3] = 0.0;
Sigma->m[2][3] = 0.0;
Sigma->m[3][3] = 1.0;

}
```

APPENDIX B. KALMAN_FILTER.H

```
/*****************************************************************************
  FILENAME:  Kalman_Filter.h

  AUTHOR:    Dave McClarin

  DATE:      8 March 1996

  PURPOSE:   'H' file for kalman_filter and navigator1 routines
*****************************************************************************/
#ifndef KALMAN_FILTER_H
#define KALMAN_FILTER_H

#include "matrix.h"
#include <math.h>
#include <time.h>

#define TRUE 1
#define FALSE 0
#define DegToRad(x) ((double) (x * M_PI/180.0))
#define RadToDeg(x) ((double) (x * 180.0/M_PI))

/* div-tracker measurement variances */
#define DvTrk_R1_Var 10.0
#define DvTrk_R2_Var 10.0

/* dgps measurement variances */
#define Dgps_Lat_Var 64000.0
#define Dgps_Lon_Var 36000.0

/* gps measurement variances */
#define Gps_Lat_Var  62500000.0
#define Gps_Lon_Var  14400000.0

/* dgps measurement standard deviations */
#define Dgps_Lat_Dev 800.0
#define Dgps_Lon_Dev 600.0

/* gps measurement standard deviations */
#define Gps_Lat_Dev 250.0
#define Gps_Lon_Dev 120.0
```

```
/* mili-seconds of arc to feet conversion factor */
static double Feet_Conv = 0.1;

/* divetracker data storage structure */
typedef struct{
    double  Xloc;
    double  Yloc;
    double  Zloc;
    double  Range;
}transponder;

/*****  SIMULATE SET POINT *****/
int Simulate = TRUE;

/* divetracker location default values */
double DT1X = 15.0;
double DT1Y =  10.0;
double DT1Z = 40.0;
double DT2X = 15.0;
double DT2Y =  -10.0;
double DT2Z = 40.0;

/* default values */
double Orig_Lat = 130000000.0;
double Orig_Lon = -440000000.0;
double Init_Heading = 0.0;
double Posture_X = 0.0;
double Posture_Y = 0.0;

/* Sensor Distances from Sensor to Center of Phoenix */
DiveTrackerXducer_Dist = -2.3;
GpsAntenna_dist = 1.0;
To_Center = -1.0;
To_Sensor = 1.0;

/* setting for ocean current relaxation time 7200 seconds (2 hours) */
const double Tau = 7200.0;

/* kalman_filter.c  prototypes */
matrix kalman_filter(matrix Q, matrix U, int No_Fix, double *Total_Drift,
        matrix *Sigma, matrix Phi, matrix R, matrix *K,
        matrix Uw, transponder Dt1, transponder Dt2, int Fix_Type,
        int *Loss_Track, double Gps_X, double Gps_Y, double Course);
```

```
double NavtoRad(double Degrees, double Init_Heading);
double  my_square(double xx);
matrix translate_position(double Direction, matrix U, double Course, int Fix_Type);


/* navigator1.c prototypes */
void navigator1(void);
void nav1_initialize(int Path, int Simulate);
void reset_kalman(matrix *Sigma);
void my_parse_telemetry_string(char *Nav_String_read);

#endif
```

APPENDIX C. KALMAN_FILTER.C

```
/*****************************************************************
  FILENAME:       kalman_filter.c

  AUTHOR:         Dave McClarin

  DATE:           7 February 1996

  PURPOSE:        Performs the movement and measurment steps for both the
                  extended and regular discrete kalman_filter.

  FUNCTIONS:  kalman_filter()
              NavtoRad()
              my_square()
              translate_position();
*****************************************************************/

#include "matrix.h"
#include "kalman_filter.h"
#include <stdio.h>
#include <math.h>
#include <time.h>
#include "../execution/statevector.h"
#include <stdlib.h>


/*****************************************************************
  FUNCTION:       kalman_filter()

  AUTHOR:         Dave Mcclarin

  DATE:           7 February 1996

  PURPOSE:    Performs the movement and measurment steps for both a
              extended and regular discrete kalman_filter. Determines
              the dimensionless shock value and sets the loss_track
              flag if the d-shock exceeds 50.

  RETURNS:    State Matrix U
*****************************************************************/
```

```
matrix kalman_filter(matrix Q, matrix U, int No_Fix, double *Total_Drift,
        matrix *Sigma, matrix Phi, matrix R, matrix *K,
        matrix Uw, transponder Dt1, transponder Dt2, int Fix_Type,
        int *Loss_Track, double Gps_X, double Gps_Y, double Course){

int Trace;
double Calc_Dist1, Calc_Dist2;
matrix InverseMat, Ds;
matrix Shock;

/* matrix of actual measurments */
matrix Z = {{{0.0}, {0.0}},2,1};

/* extended filter calculated value of z */
matrix Fu = {{{0.0}, {0.0}},2,1};

/* Id matrix */
matrix ID = {{{1.0, 0.0, 0.0, 0.0},
              {0.0, 1.0, 0.0, 0.0},
              {0.0, 0.0, 1.0, 0.0},
              {0.0, 0.0, 0.0, 1.0}},4,4};

/* measurment matrix, how the measurement depends on the state */
matrix H  = {{{1.0, 0.0, 0.0, 0.0},
              {0.0, 1.0, 0.0, 0.0}},2,4};

/* mean of the measurement noise */
matrix Uv = {{{0.0},{0.0}},2,1};

/* Trace feature allows for printing of various values for use
  in trouble shooting, initilized to false */
Trace = FALSE;
*Loss_Track = FALSE;


/* conducting movement step 'Dead Reckoning'
  U = Phi*U + Uw; -> updates state values via movement
  Sigma = Phi*Sigma*Transpose(Phi) + Q  -> updates state covariance */
U = matrix_add(matrix_multiply(Phi,U),Uw);
*Sigma = matrix_add(matrix_multiply(matrix_multiply(Phi,*Sigma),
                matrix_transpose(Phi)),Q);

if (Trace){
```

```
  printf("DT1X %lf DT1Y %lf DT1Z %lf DT2X %lf DT2Y %lf DT2Z %lf \n",
        Dt1.Xloc, Dt1.Yloc, Dt2.Zloc, Dt2.Xloc, Dt2.Yloc, Dt2.Zloc);
  output_matrix(U);
  printf("X %lf Y %lf Z %lf \n",x,y,z);
}

/*measurement phase
  K = Sigma*Transpose(H)*Inverse(H*Sigma*Transpose(H) + R) ->
        Updates Kalman Gain
  U = U + K(Z - Uv - H*U) ->
        updates state for linear GPS non-extended K-filter
  U = U + K(Z- Fu - Uv) ->
        update state for non linear DT extended K-Filter
  Sigma = (Id - K*H)*Sigma -> updates state covariance */
if (No_Fix == FALSE){

  /* translate position of center of Phoenix to the Sensor */
  U = translate_position(To_Sensor, U, Course, Fix_Type);

  /* fix_type of 2 = dive track, which is non-linear and requires a
     extended kalman_filtering */
  if (Fix_Type == 2){

    /* calculated measurment function */
    Calc_Dist1 = sqrt(my_square(Dt1.Xloc - U.m[0][0])+
                      my_square(Dt1.Yloc - U.m[1][0])+
                        my_square(Dt1.Zloc - z));
    Calc_Dist2 = sqrt(my_square(Dt2.Xloc - U.m[0][0])+
                      my_square(Dt2.Yloc - U.m[1][0]) +
                        my_square(Dt2.Zloc - z));

    Fu.m[0][0] = Calc_Dist1;
    Fu.m[1][0] = Calc_Dist2;

    /* 1st partial deriviatives of the measurment fuction with respect
       to the associated state variable 'the Jacobian' */
    H.m[0][0] = -(Dt1.Xloc - U.m[0][0])/Calc_Dist1;
    H.m[0][1] = -(Dt1.Yloc - U.m[1][0])/Calc_Dist1;
    H.m[1][0] = -(Dt2.Xloc - U.m[0][0])/Calc_Dist2;
    H.m[1][1] = -(Dt2.Yloc - U.m[1][0])/Calc_Dist2;
```

```
  Z.m[0][0] = Dt1.Range;
  Z.m[1][0] = Dt2.Range;

  if (Trace){
      printf("X = %lf Y = %lf Z = %lf\n",x,y,z);
      printf("Dt1X = %lf Dt1y = %lf Dt1z = %lf\n",Dt1.Xloc,
                                  Dt1.Yloc, Dt1.Zloc);
      printf("Dt2X = %lf Dt2y = %lf Dt2z = %lf\n",Dt2.Xloc,
                                  Dt2.Yloc, Dt2.Zloc);
      printf("Calc1 %lf DT1-RANGE %lf Calc2 %lf DT2-RANGE %lf\n",
            Calc_Dist1,Dt1.Range ,Calc_Dist2, Dt2.Range);
  }
}
else{
    /* a gps fix which is linear and uses non-extened filtering */
    Z.m[0][0] = Gps_X;
    Z.m[1][0] = Gps_Y;
}

/* K gain calculations */
InverseMat = matrix_inverse(matrix_add(matrix_multiply(
            matrix_multiply(H,*Sigma),matrix_transpose(H)),R));
*K = matrix_multiply(matrix_multiply(*Sigma,matrix_transpose(H)),
                    InverseMat);

/* calculate shock for extended or non extended filtering */
if (Fix_Type == 2)
    Shock = matrix_subtract(matrix_subtract(Z,Fu),Uv);
else
    Shock = matrix_subtract(matrix_subtract(Z,Uv),
            matrix_multiply(H,U));

/* calculates dimensionless shock */
Ds = matrix_multiply(matrix_multiply(matrix_transpose(Shock),
                InverseMat),Shock);

if (Trace)
  printf("Dimensionless Shock = %lf\n",Ds.m[0][0]);


/* only perform measurment steps if DS < 50, to ensure no bad
  measurements */
if (Ds.m[0][0] < 50){
```

```
    if (Trace){
      printf("shock \n");
      output_matrix(Shock);
      printf("k \n");
      output_matrix(*K);
      printf("k * Shock\n");
      output_matrix(matrix_multiply(*K,Shock));
    }

    U = matrix_add(U,matrix_multiply(*K,Shock));
    *Sigma = matrix_multiply(matrix_subtract(
                    ID,matrix_multiply(*K,H)),*Sigma);

    /* updates the total amount of error or drift if the measurement
      was good */
    *Total_Drift = sqrt(U.m[2][0]*U.m[2][0]+U.m[3][0]*U.m[3][0]);
  }
  else {
    /* sets loss track to TRUE for bad measurement */
    if (Trace) printf("DS = %lf Ignoring last measurement\n",
          Ds.m[0][0]);
    *Loss_Track = TRUE;
  }

  /* Put translated fix from sensor back to center of Phoenix */
  U = position_translate(To_Center, U, Course, Fix_Type);

}

return U;
}
```

```
/*************************************************************************
  FUNCTION:  NavtoRad()

  AUTHOR:    Dave Mcclarin

  DATE:      7 February 1996

  PURPOSE:   Computes the Radian equivilant of a Naval Degree
             Measurement

  RETURNS:   Rads as a Double
*************************************************************************/
double NavtoRad(double Degrees, double Gyro_Error){

double Rad;

/* adds Gyro_Error to input degrees */
Degrees = Degrees + Gyro_Error;

/* normalizes degrees */
if (Degrees < 0.0)
  Degrees = Degrees + 360.0;
if (Degrees >= 360.0)
  Degrees = Degrees - 360.0;

/* assigns the proper rads to Naval degrees */
if ((0.0 <= Degrees) & (Degrees <= 90.0))
  Rad = (90.0-Degrees)*M_PI/180.0;

else if ((90.0 < Degrees) & (Degrees <= 180.0))
  Rad = M_PI/180.0*(180.0-Degrees)+3.0*M_PI/2.0;

else if ((180.0 < Degrees) & (Degrees <= 270.0))
  Rad = M_PI/180.0*(270.0-Degrees)+M_PI;

else if ((270.0 < Degrees) & (Degrees <= 360.0))
  Rad = M_PI/180.0*(360.0-Degrees)+M_PI/2.0;

return Rad;
}
```

```
/*************************************************************************
  FUNCTION:  my_square()

  AUTHOR:    Dave Mcclarin

  DATE:      7 February 1996

  PURPOSE:   Computes square of a double (for some reason Pow stopped
             working when integrated with others code)

  RETURNS:   Double * Double
*************************************************************************/
double  my_square(double xx){
return xx*xx;
}

/*************************************************************************
  FUNCTION: translate_position()

  AUTHOR:    Dave Mcclarin

  DATE:      7 February 1996

  PURPOSE:  Translates the Pheonix center to the DiveTracker transducer or the
            Gps Antenna, or translate from sensor to center of Phenoex.

  RETURNS:  matrix
*************************************************************************/
matrix translate_position(double Direction, matrix U, double Course, int Fix_Type){

/* translate the divetracker transducer to center and vice versa depending upon
   The direction 1 = to transducer, -1 = to center */
if (Fix_Type == 2){
   U.m[0][0] += Direction*DiveTrackerXducer_Dist*sin(Course);
   U.m[1][0] += Direction*DiveTrackerXducer_Dist*cos(Course);
}
else{ /* translates the GPS antenna to center and vice versa */
   U.m[0][0] += Direction*GpsAntenna_Dist*sin(Course);
   U.m[1][0] += Direction*GpsAntenna_Dist*cos(Course);
}
}
```

APPENDIX D. READGPS.H

```
/*************************************************************************
  FILENAME:  readgps.h

  AUTHOR:    Dave McClarin

  DATE:      15 March 1996

  PURPOSE:   'H' file for Opens and reads of Gps Data through a Solaris
             serial port,then parses and returns the desired gps data.

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

#ifndef READGPS_H
#define READGPS_H

#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
#include "kalman_filter.h"
#include "matrix.h"

#define  GPSBLOCKSIZE 76  /* size of motorola @@Ea position message */
#define  New_Data   1
#define  Old_Data   0
#define  GPS_STR_SIZE     GPSBLOCKSIZE-1
#define  SATELITE_DATA    39

/* sets number of channels of the gps reciever */
const int CHANNELS = 8;

/* used in raw gps data decoding */
typedef long FOURBYTE;

/* defines the decoded gps data storage type */
typedef struct{
    double lat;
    double lon;
    double time;
    int type;
}gps;
```

```
/* defines the raw gps data storage type */
typedef struct{
  unsigned char GPSdata[2*GPSBLOCKSIZE];
  int         data_status;
}raw_gps;

/* Defines raw gps data storage 'GLOBAL' */
raw_gps Gps_Message =  {"none",Old_Data};

/*serial read timeout variable */
int TIMEOUT = FALSE;

/* headers for 6 and 8 channel motorola messages */
char header_6[5] = "@@Ba";  /* 6 channel */
char header_8[5] = "@@Ea";  /* 8 channel */

/* function prototypes for readgps.c */
gps get_gps_data(const int path);
int CheckSumCheck(void);
gps GetMilSec(gps temp);
gps GetGpsTime(gps temp);
gps GetGpsFixType(gps temp);
int determine_fix(double Range1, double Range2, gps Gps_Fix,
            int *Gps_Avail,int *Dgps_Avail, int *Dt_Avail,
            int *Fix_By, int *Fix_Concur, matrix U,
            double Gps_X, double Gps_Y, int Loss_Track,
            double t, double DT_Timer);
int Gps_Serial_Read(int path);
int initialize_serial(void);
int open_tty(char *device_name);
void tty_open_timeout(int arg);
void serial_read_timeout(int arg);
gps simulate_gps_data(double x, double y, int fix_type);

#endif
```

APPENDIX E. READGPS.C

```
/*****************************************************************************
  FILENAME: readgps.c

  AUTHOR:   Dave McClarin

  DATE:     15 March 1996

  PURPOSE:  Opens and reads Gps Data through a Solaris serial ports,
            then parses and returns the desired gps data.

  FUNCTIONS: get_gps_data()
             CheckSumCheck()
             GetMilSec()
             GetGpsTime()
             GetGpsFixType()
             determine_fix()
             Gps_Serial_Read()
             initialize_serial()
             tty_open_timeout()
             open_tty()
             serial_read_timeout()
             simulate_gps_data()
*****************************************************************************/
#include <stdio.h>
#include <ctype.h>
#include <errno.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include "readgps.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>
/* #include <sys/termiox.h> */
#include "termiox.h"
#include <sys/uio.h>
#include <termios.h>
#include <termio.h>
#include "matrix.h"
#include "kalman_filter.h"
```

```
/*************************************************************************
  FUNCTION:  get_gps_data()

  AUTHOR:    Dave Mcclarin

  DATE:      6 February 1996

  PURPOSE:   Determines if an updated gps position message is available
             and copies it into the input argument 'rawMessage'. If the
             message has a valid checksum and was obtained with at least
             three satelites in view,  a 'TRUE' is returned to the
             caller, indicating that the message is valid.


  RETURNS:   GPS Data Structure.
*************************************************************************/
gps get_gps_data(const int path){

unsigned char tempchar;
int satelites;

/* returned gps values stored in temp and initialied to zeroes */
gps temp;
temp.lat = 0.0;
temp.lon = 0.0;
temp.time = 0.0;
temp.type = 0;

/* Global that keeps track of the serial read has timed out */
TIMEOUT = FALSE;

/* calls read that places data in Gps_Message global */
Gps_Serial_Read(path);

if (Gps_Message.data_status == New_Data){


  /* finds the number of satelites available */
  tempchar = Gps_Message.GPSdata[39];
  satelites = (int)tempchar;

 /* ensures there is a valid checksum and 3 satelites for data
    places data into the gps temp structure */
```

```
  if ((CheckSumCheck() == TRUE) && (satelites > 3)){
     temp = GetMilSec(temp);
     temp = GetGpsTime(temp);
     temp = GetGpsFixType(temp);
  }

}
else
   temp.type = 0;


/* sets flag to indicated data has been read */
Gps_Message.data_status = Old_Data;

return temp;
}


/*********************************************************************
   FUNCTION:  checkSumCheck

   AUTHOR:    Dave Mcclarin

   DATE:      6 February 1996

   MODIFIED:  From code by Dave Gay and Eric Bachman 11 July 95

   PURPOSE:   Takes an exclusive or of bytes 2 through 78 in a Motorola
              format (@@EA) position message and compares it to the
              checksum of the message of the message.

  RETURNS:    TRUE, if the message contains a valid checksum
*********************************************************************/
int CheckSumCheck()
{
   unsigned short chkSum;
   unsigned short temp;
   int i;

   /* gets first element of message */
   chkSum = Gps_Message.GPSdata[2];
```

```
  /* XORs bytes 2 through 78 to get the checksum */
  for (i = 3; i < (GPS_STR_SIZE-2); i++) {
   temp = Gps_Message.GPSdata[i];
   chkSum = chkSum ^ temp;
  }
  /* returns a TRUE of valid checksum */
  return (chkSum == Gps_Message.GPSdata[GPS_STR_SIZE-2]);
}


/*************************************************************************
  FUNCTION:  getMilSec

  AUTHOR:    Dave Mcclarin

  DATE:      6 February 1996

  MODIFIED:  From code by Dave Gay and Eric Bachman 11 July 95

  PURPOSE:   Extracts position in mili-seconds of arc from a Motorola
             (@@Ea) data string.

  RETURNS:   The latitude and longitude in milli-seconds arc
             in a gps data structure
*************************************************************************/
gps GetMilSec(gps temp) {

  FOURBYTE temps4byte;

  /* gets the latitude from the raw data message */
  temps4byte     = Gps_Message.GPSdata[15];
  temps4byte     = (temps4byte<<8) + Gps_Message.GPSdata[16];
  temps4byte     = (temps4byte<<8) + Gps_Message.GPSdata[17];
  temps4byte     = (temps4byte<<8) + Gps_Message.GPSdata[18];
  temp.lat = temps4byte;
    /* gets the longitude from the raw data message */
  temps4byte     = Gps_Message.GPSdata[19];
  temps4byte     = (temps4byte<<8) + Gps_Message.GPSdata[20];
  temps4byte     = (temps4byte<<8) + Gps_Message.GPSdata[21];
  temps4byte     = (temps4byte<<8) + Gps_Message.GPSdata[22];
  temp.lon = temps4byte;

  return temp;}
```

```
/*************************************************************************
  FUNCTION:  getGpsTime

  AUTHOR:    Dave Mcclarin

  DATE:      6 February 1996

  MODIFIED:  From code by Dave Gay and Eric Bachman 11 July 95

  PURPOSE:   Extracts the position time in seconds from a Motorola (@@Ea)
             data string.

  RETURNS:   The time of the gps message in seconds stored
             in a gps data structure.
*************************************************************************/
gps GetGpsTime(gps temp){

  unsigned char    tempchar, hours, minutes;
  unsigned long    tempu4byte;
  double seconds;

  /* gets hours and minutes from raw data message */
  hours   = Gps_Message.GPSdata[8];
  minutes = Gps_Message.GPSdata[9];

  /* gets seconds from raw data message */
  tempchar      = Gps_Message.GPSdata[10];
  tempu4byte    = Gps_Message.GPSdata[11];
  tempu4byte    = (tempu4byte<<8) + Gps_Message.GPSdata[12];
  tempu4byte    = (tempu4byte<<8) + Gps_Message.GPSdata[13];
  tempu4byte    = (tempu4byte<<8) + Gps_Message.GPSdata[14];

  seconds = (double)tempchar + (((double)tempu4byte)/1.0E+9);

  /* converts hours minutes and seconds to total seconds for the day */
  temp.time = (double)hours * 3600.0 + (double)minutes * 60.0 + seconds;
  return  temp;
}
```

```
/*************************************************************************
  FUNCTION:  GetGpsFixType

  AUTHOR:    Dave Mcclarin

  DATE:      6 February 1996

  PURPOSE:   Extracts the position type from a Motorola (@@Ea)
             data string, 1 = DGPS, 3 = GPS

  RETURNS:   The type of the gps fix stored in a gps data structure.
*************************************************************************/
gps GetGpsFixType(gps temp){

  unsigned long    tempu4byte, MASK;

  tempu4byte      = Gps_Message.GPSdata[GPS_STR_SIZE-3];
  MASK = 4;


  /* checks bit 2 of tempu4byte, if set then DGPS avail, else GPS avail*/
  if ((tempu4byte & MASK) == MASK){
       temp.type = 1;
  }
  else {
       temp.type = 3;
  }
  return  temp;
}


/*************************************************************************
  FUNCTION:  determine_fix()

  AUTHOR:    Dave Mcclarin

  DATE:      16 February 1996

  PURPOSE:   Determines the type of position fix to be used by the
             kalman_filter.

  RETURNS:   The type of the position fix to be used 1 = DGPS, 2 = Dive
             Track and 3 = GPS.  Also sets Fix_By (what the fix was
```

```
        computed by), sets the Dgps, Gps Dt avail flags and sets
        the Fix_Concur flag if a dive track fix and gps fix concur
        in position
*************************************************************************/

int determine_fix(double Range1, double Range2, gps Gps_Fix, int *Gps_Avail,
            int *Dgps_Avail, int *Dt_Avail, int *Fix_By,
            int *Fix_Concur, matrix U, double Gps_X, double Gps_Y,
            int Loss_Track, double t, double DT_Timer){

/* Sets the fix availabilty flags */
if ((Range1 >= 0.0) && (Range2 >= 0.0))
   *Dt_Avail = TRUE;
else
   *Dt_Avail = FALSE;

if (Gps_Fix.type == 1){
   *Dgps_Avail = TRUE;
   *Gps_Avail = FALSE;
}else
   *Dgps_Avail = FALSE;

if (Gps_Fix.type == 3){
   *Dgps_Avail = FALSE;
   *Gps_Avail = TRUE;
}else
   *Gps_Avail = FALSE;

if (Gps_Fix.type == 0){
   *Dgps_Avail = FALSE;
   *Gps_Avail = FALSE;
}

/* determines what to use for the fix if both Dgps and
  dive track is avail with a loss_track flag set */
if ((*Dgps_Avail && *Dt_Avail) && (Loss_Track)){

  /* if last fix by DT then use DGPS, or vice versa */
  if (*Fix_By == 2){
        *Fix_Concur = FALSE;
        *Fix_By = 1;
        return *Fix_By;
  }
```

```
  else{
        *Fix_Concur = FALSE;
        *Fix_By = 2;
        return *Fix_By;
  }
}


/* determines what to use for the fix if both Gps and
  dive track is avail with a loss_track flag set */
if ((*Gps_Avail && *Dt_Avail) && (Loss_Track)){

   /* if last fix by DT then use GPS, or vice versa */
  if (*Fix_By == 2){
        *Fix_Concur = FALSE;
        *Fix_By = 3;
        return *Fix_By;
  }
  else{
        *Fix_Concur = FALSE;
        *Fix_By = 2;
        return *Fix_By;
  }
}


/* determines what to use for the fix if both DGps and
  dive track is avail with no loss_track flag set */
if (*Dgps_Avail && *Dt_Avail){

  /* if the difference between the DGPS posit and filter Posit
    is > the STD DEV of the GPS Posits then reset fix to
    Dgps Position, and set fix concur to false else vice versa */
  if ((fabs(U.m[0][0] - Gps_X) > Dgps_Lat_Dev) ||
          (fabs(U.m[1][0] - Gps_Y) > Dgps_Lon_Dev)){
        *Fix_Concur = FALSE;
        *Fix_By = 1;
        return *Fix_By;
  }else
  {
        *Fix_Concur = TRUE;
        *Fix_By = 2;
        return *Fix_By;
```

```
  }
}


/* determines what to use for the fix if both Gps and
  dive track is avail with no loss_track flag set */
if (*Gps_Avail && *Dt_Avail){

  /* if the difference between the DGPS posit and filter Posit
    is > the STD DEV of the GPS Posits then reset fix to
    gps Position, and set fix concur to false else vice versa */
  if ((fabs(U.m[0][0] - Gps_X) > Gps_Lat_Dev) ||
                (fabs(U.m[1][0] - Gps_Y) > Gps_Lon_Dev)){
        *Fix_Concur = FALSE;
        *Fix_By = 3;
        return *Fix_By;
      }else{
        *Fix_Concur = TRUE;
        *Fix_By = 2;
        return *Fix_By;
      }
}

/* if none of the above are true then just return what
  fix typs is avail and set fix_concur to false */
if (*Dt_Avail == TRUE)
   *Fix_By = 2;
else if (*Dgps_Avail == TRUE)
   *Fix_By = 1;
else if (*Gps_Avail == TRUE)
   *Fix_By = 3;
else
   *Fix_By = 0;
*Fix_Concur = FALSE;

if ((t < DT_Timer) && ((*Fix_By == 1) || (*Fix_By == 3)))
   *Fix_By = 0;

return *Fix_By;}


/*************************************************************************
  FUNCTION:  Gps_Serial_Read()
```

```
  AUTHOR:    Dave Mcclarin

  DATE:      6 February 1996

  PURPOSE:   Reads the serial port for the raw gps Position

  RETURNS:   An integer to keep the compiler happy, and places the
             Gps Raw data into the GLOBAL Gps_Message
*************************************************************************/
int Gps_Serial_Read(int npath)
{
 unsigned char Gps_string[GPSBLOCKSIZE], ch[5];
 int j;
 char  data_header[5];

 /* set signal handler for open watchdog */
 signal(SIGALRM, serial_read_timeout);

 /* set watchdog timer */
  alarm(2);

 /* if old dat stored in message then get new data */
 if (Gps_Message.data_status == Old_Data){

    ch[0] = 0;
    j = 0;


    /*clears the port of all chars to beginning of data stream '@'
      j counts to 5 for a differential gps data message */
    do{

       if ( read(npath,ch,1) < 0)
            perror("do read error");

       if (TIMEOUT)
             return 0;
       j++;

    }while ((ch[0] != '@') && (j < 5));


    /* keeps first char */
```

```
Gps_string[0] = ch[0];


/* reads message header */
for(j=1;j<=3;j++){

  if (read(npath,ch,1)<0)
        perror("main string error");

  Gps_string[j] = ch[0];

  if (TIMEOUT)
       return 0;
}

/* puts global message header into local var data_header */
strcpy(data_header,header_8);

if ((Gps_string[0]==data_header[0])&&
    (Gps_string[1]==data_header[1])&&
    (Gps_string[2]==data_header[2])&&
    (Gps_string[3]==data_header[3])){


  /* if valid header reads the entire message string */
  for(j=4;j<=GPS_STR_SIZE;j++){
      if (read(npath,ch,1)<0)
         perror("main string error");

      if (TIMEOUT)
          return 0;

      Gps_string[j]=ch[0];
  }

  /* copies data into the GLOBAL Gps_Message */
  for(j=0;j<=GPS_STR_SIZE;j++){

    Gps_Message.GPSdata[j]=Gps_string[j];
    Gps_Message.data_status = New_Data;

    if (TIMEOUT)
      return 0;
```

```
        }
    }
    else{

        /* clear the differential gps data string */
        if ((Gps_string[0]=='@')&&(Gps_string[1]=='@')
              &&(Gps_string[2]=='C')&& (Gps_string[3]=='k')){

          for(j=0;j<=2;j++){
            if (read(npath,ch,1)<0)
              perror("main string error");

            if (TIMEOUT)
              return 0;
          }
        }
      }
    }

    /* clear the timer */
    alarm(0);

    /* go back to default alarm handler */
    signal(SIGALRM, SIG_DFL);

return 0;
}


/*************************************************************************
  FUNCTION:  initialize_serial()

  AUTHOR:    Dave Mcclarin

  DATE:      6 February 1996

  MODIFIED:  From Dinc.c Software source included in SCSII hardware
             package as an example of solaris serial port coms.

  PURPOSE:   Initilizes the serial port for reading the raw gps data

  RETURNS:   An integer that is used as the file descriptor
*************************************************************************/
```

```
int initialize_serial(){

  int i, path, stat;
  unsigned short flag = 6;
  struct termiox *termsetup;
  struct termios tty_termios;
  termsetup = (struct termiox *)calloc(50,sizeof(short));

  /*Get path number */
  path = open_tty("/dev/sts/ttyc50");
  if (path < 0)
     printf("error in initilization open tty, bad argument? \n");

  /* sets the io-control flag */
  ioctl(path,TCGETX,termsetup);
  termsetup->x_hflag = flag;
  ioctl(path,TCSETX,termsetup);

  /* sets the termios data structure to 'good' starting points */
  stat = tcgetattr(path,&tty_termios);
  tty_termios.c_lflag = 0;
  tty_termios.c_oflag &= ~OPOST;
  tty_termios.c_iflag &=
   ~(INPCK | PARMRK | BRKINT | INLCR | ICRNL | IUCLC | IXANY );
  tty_termios.c_iflag |= IGNBRK;
  tty_termios.c_cflag &= ~(CSIZE | PARODD | PARENB | CSTOPB);
  tty_termios.c_cflag |= (CREAD | CLOCAL);
  tty_termios.c_cflag |= CS8;
     tty_termios.c_iflag &= ~(IXON | IXOFF);
  stat = tcsetattr(path, TCSANOW, &tty_termios);


  return path;
}



/*************************************************************************
  FUNCTION:  open_tty()

  AUTHOR:    Dave Mcclarin

  DATE:      6 February 1996
```

```
  MODIFIED:  From Dinc.c Software source included in SCSII hardware
             package as an example of solaris serial port coms.

  PURPOSE:   Opens the TTY port non-blocking style. Runs a watchdog
             timer in case another process is blocking for carrier
             preventing us from proceeding with open.

  RETURNS:   An integer that is used as the file descriptor
*****************************************************************/
int open_tty(char *tty_name){

  int tty_fd;

  /* set signal handler for open watchdog */
  signal(SIGALRM, tty_open_timeout);

  /* set watchdog timer */
  alarm(10);

  /* open the tty port */
  tty_fd = open(tty_name,O_RDWR | O_NDELAY);
  if(tty_fd < 0) {
    perror(tty_name);

  }

  /* clear the timer */
  alarm(0);

  /* go back to default alarm handler */
  signal(SIGALRM, SIG_DFL);

  /* restore normal blocking operation on port */
  fcntl(tty_fd, F_SETFL, 0);

  return tty_fd}

/*****************************************************************
  FUNCTION:  tty_open_timeout()

  AUTHOR:    Dave Mcclarin

  DATE:      6 February 1996
```

```
  MODIFIED:   From Dinc.c Software source included in SCSII hardware
              package as an example of solaris serial port coms.

  PURPOSE:    Just prints a message saying the device is busy.

  RETURNS:    Void
*****************************************************************************/
void tty_open_timeout(int arg){

  /* note: only passing 'arg' to shut up compiler */
  printf("Timed out: port busy path = %d\n",arg);
  exit(1);
}


/*****************************************************************************
  FUNCTION:   serial_read_timeout()

  AUTHOR:     Dave Mcclarin

  DATE:       6 February 1996

  MODIFIED:   From Dinc.c Software source included in SCSII hardware
              package as an example of solaris serial port coms.

  PURPOSE:    Just prints a message saying the device is busy, and
              sets TIMEOUT to TRUE.

  RETURNS:    Void
*****************************************************************************/
void serial_read_timeout(int arg){

  printf("Serial_read_Timeout \n");
  TIMEOUT = TRUE;
}


/*****************************************************************************
  FUNCTION:   simulate_gps_data()

  AUTHOR:     Dave Mcclarin

  DATE:       9 February 1996
```

```
  PURPOSE:    Just prints a message saying the device is busy, and
              sets TIMEOUT to TRUE.

  RETURNS:    Gps data structure containing simulated fix data
*************************************************************************/
gps simulate_gps_data(double x, double y, int fix_type){

  gps Gps_Fix;

  /* Computes simulated fix data in lat and lon milliseconds of arc
     and puts up to 10 feet of noise on the posit */

  x = x + (10.0 * drand48());
  y = y + (10.0 * drand48());

  Gps_Fix.lat = ((x-Posture_X)/Feet_Conv) + Orig_Lat;
  Gps_Fix.lon = ((y-Posture_Y) /(Feet_Conv *
                cos(DegToRad(Gps_Fix.lat/3600000))))
                 + Orig_Lon;
  Gps_Fix.type = fix_type;
  return Gps_Fix;
}
```

APPENDIX F. MATRIX.H

```
/*********************************************************************
  FILENAME:  matrix.h

  AUTHOR:    Dave McClarin

  DATE:      6 February 1996

  PURPOSE:   'H' file for matrix operators to include addition,
             subtraction, multiplication, and inverse.
*********************************************************************/

#ifndef MATRIX_H
#define MATRIX_H

/* defines the matrix data structure type */
typedef struct{
      double m[4][4];
      int row, col;
}matrix;

/* function prototypes for matrix.c */
matrix matrix_multiply(matrix mat1, matrix mat2);
matrix matrix_add(matrix mat1, matrix mat2);
matrix matrix_subtract(matrix mat1, matrix mat2);
matrix matrix_transpose(matrix mat1);
matrix matrix_inverse(matrix mat1);
matrix gauss_elimination(matrix mat1, matrix mat2);
matrix rtransform(double azimuth, double pitch, double roll);
void output_matrix(matrix input_matrix);

#endif
```

APPENDIX G. MATRIX.C

```
/*************************************************************************
  FILENAME:  matrix.c

  AUTHOR:    Dave McClarin

  DATE:      19 February 1996

  PURPOSE:   Create matrix operators to include addition, subtraction,
             multiplication, inverse and gauss_elimination, and
             create a rotation matrix.

  FUNCTIONS: matrix_multiply()
             matrix_add()
             matrix_subtract()
             matrix_transpose()
             matrix_inverse()
             gauss_elimination()
             matrix rtransform()
             output_matrix()
*************************************************************************/


#include <stdio.h>
#include <math.h>
#include "matrix.h"
#include "kalman_filter.h"



/*************************************************************************
  FUNCTION:  matrix_multiply()

  AUTHOR:    Dave Mcclarin

  DATE:      6 February 1996

  PURPOSE:   Multiplies two matrix's together

  RETURNS:   Matrix1 * Matrix2 in a matrix data structure
*************************************************************************/
matrix matrix_multiply(matrix mat1, matrix mat2)
{
```

```
int row, col, i;
matrix answer;

/* conducts multiplication */
for (row=0; row<mat1.row; row++) {
  for (col=0; col<mat2.col; col++) {
      answer.m[row][col]=0.0;
      for (i=0; i <mat1.col; i++){
          answer.m[row][col] += mat1.m[row][i] * mat2.m[i][col];
      }
  }
}

/* assigns new row and col number to matrix data structure */
answer.row = mat1.row;
answer.col = mat2.col;
return answer;
}



/*****************************************************************************
  FUNCTION:  matrix_add()

  AUTHOR:    Dave Mcclarin

  DATE:      6 February 1996

  PURPOSE:   Adds two matrix's together

  RETURNS:   Matrix1 + Matrix2 in a matrix data structure
*****************************************************************************/
matrix matrix_add(matrix mat1, matrix mat2)
{
int row, col;
matrix answer;

/* conducts addition */
for (row=0; row<mat1.row; row++) {
   for (col=0; col<mat1.col; col++) {
       answer.m[row][col] = mat1.m[row][col] + mat2.m[row][col];
   }
```

```
}

/* assigns new row and col number to matrix data structure */
answer.row = mat1.row;
answer.col = mat1.col;
return answer;
}


/*****************************************************************
  FUNCTION:  matrix_subtract()

  AUTHOR:    Dave Mcclarin

  DATE:      6 February 1996

  PURPOSE:   Subtracts two matrix's from each other

  RETURNS:   Matrix1 - Matrix2 in a matrix data structure
*****************************************************************/
matrix matrix_subtract(matrix mat1, matrix mat2)
{
int row, col;
matrix answer;

/* conducts subtraction */
for (row=0; row<mat1.row; row++) {
   for (col=0; col<mat1.col; col++) {
        answer.m[row][col] = mat1.m[row][col] - mat2.m[row][col];
   }
}

/* assigns new row and col number to matrix data structure */
answer.row = mat1.row;
answer.col = mat1.col;
return answer;
}
```

```
/*************************************************************************
  FUNCTION:   matrix_transpose()

  AUTHOR:     Dave Mcclarin

  DATE:       6 February 1996

  PURPOSE:    Creates the transpose of a matrix

  RETURNS:    transpose(Matrix) in a matrix data structure
*************************************************************************/
matrix matrix_transpose(matrix mat1)
{
int row, col;
matrix answer;

/* conducts transpose */
for (row=0; row<mat1.row; row++) {
   for (col=0; col<mat1.col; col++) {
       answer.m[col][row] = mat1.m[row][col];
   }
}
/* assigns new row and col number to matrix data structure */
answer.row = mat1.col;
answer.col = mat1.row;
return answer;
}


/*************************************************************************
  FUNCTION:   matrix_inverse()

  AUTHOR:     Dave Mcclarin

  DATE:       6 February 1996

  PURPOSE:    Creates the inverse of a matrix using gausing elimination

  RETURNS:    inverse(Matrix) in a matrix data structure
*************************************************************************/
matrix matrix_inverse(matrix mat1)
{
int row, col;
```

```
matrix Idmat;

/*creates Id matrix of size mat 1*/
for (row=0;row<mat1.row;row++){
   for (col=0;col<mat1.col;col++){
      if (row==col){
         Idmat.m[row][col] = 1.0;
      }else   Idmat.m[row][col] = 0.0;
   }
}

/* assigns new row and col number to matrix data structure */
Idmat.row=mat1.row;
Idmat.col=mat1.col;

return(gauss_elimination(mat1,Idmat));
}


/*************************************************************************
  FUNCTION:   gauss_elimination()

  AUTHOR:     Dave Mcclarin

  DATE:       6 February 1996

  PURPOSE:    returns the solution x of Ax=B

  RETURNS:    inverse(Matrix) in a matrix data structure
*************************************************************************/
matrix gauss_elimination(matrix mat1, matrix mat2)
{
int row, col, max, i ,j, k;
matrix answer;
double a[4][8], t;

/*copies the input matrix into the temp solution matrix*/
for (row=0; row<mat1.row; row++) {
   for (col=0; col<mat1.col; col++) {
       a[row][col] = mat1.m[row][col];
   }
}
/*tacks on a 2nd matrix into the temp solution matrix*/
```

```
for (row=0;row<mat2.row;row++){
   for (col=mat2.col;col<2*mat2.col;col++){
      a[row][col]=mat2.m[row][col-mat2.col];
   }
}
/* performs row eliminations */
for (i=0;i<mat1.row;i++){
    max = i;
    for (j=i+1;j<mat1.row;j++){
      if (fabs(a[j][i]) > fabs(a[max][i]))
         max = j;
    }
    for (k=i;k<2*mat1.col;k++){
      t = a[i][k];
      a[i][k] = a[max][k];
      a[max][k]= t;
    }
    for (j=i+1;j<mat1.row;j++){
      for (k=2*mat1.col;k>i-1;k--){
        if (fabs(a[i][i]) < 0.000001){
           printf("this is becomming a singular matrix");
           exit(0);
        }
        else a[j][k] = a[j][k]-a[i][k]*a[j][i]/a[i][i];
      }
    }
}
/*performs back substitution*/
for (i=0;i<mat1.col;i++){
 for (j=mat1.row-1;j>-1;j--){
    t = 0.0;
    for (k=j+1;k<mat1.row;k++){
      t = t+a[j][k]*answer.m[k][i];
    }
    answer.m[j][i] = (a[j][(mat1.col)+i]-t)/a[j][j];
 }
}

/* assigns new row and col number to matrix data structure */
answer.row = mat1.row;
answer.col = mat1.col;
return answer;
}
```

```
/*************************************************************************
  FUNCTION:  matrix rtransform()

  AUTHOR:    Dave Mcclarin

  DATE:      19 February 1996

  PURPOSE:   Construct a rotation matrix to use in transforming
             body co-ordinates into Earth co-ordinates, using right
             hand rule.
             body co-ords u = nose
                          v = right side
                          w = bottom (belly)
             earch co-ords x = North
                           y = East
                           z = down

  RETURNS:   Rotation matrix in a matrix data structure
*************************************************************************/
matrix rtransform(double azimuth, double pitch, double roll)
{
matrix answer1;

double spsi = sin(azimuth);
double cpsi = cos(azimuth);
double sphi = sin(pitch);
double cphi = cos(pitch);
double sth = sin(roll);
double cth = cos(roll);

answer1.row = 3;
answer1.col = 3;

answer1.m[0][0] = cpsi * cphi;
answer1.m[0][1] = cpsi * sphi * sth - spsi * cth;
answer1.m[0][2] = cpsi * sphi * cth + spsi * sth;
answer1.m[1][0] = spsi * cphi;
answer1.m[1][1] = spsi * sphi * sth + cpsi * cth;
answer1.m[1][2] = spsi * sphi * cth - cpsi * sth;
answer1.m[2][0] = -sphi;
answer1.m[2][1] = cphi * sth;
answer1.m[2][2] = cphi * cth;
```

```
return answer1;
}


/*************************************************************************
  FUNCTION:   output_matrix()

  AUTHOR:     Dave Mcclarin

  DATE:       6 February 1996

  PURPOSE:    Prints the contents of a matrix

  RETURNS:    Void
*************************************************************************/
void output_matrix(matrix input_matrix)
{
int i,j;

for (i=0;i<input_matrix.row;i++){
  for(j=0;j<input_matrix.col;j++){
     printf("%4.4f  ",input_matrix.m[i][j]);
  }
  printf("\n");
}
printf("\n");
}
```

LIST OF REFERENCES

[BACH95] Bachmann, E.R. and Gay, D.L., "Design and Evaluation of an Integrated GPS/INS System for Shallow-Water AUV Navigation," M.S. Thesis, Naval Postgraduate School, Monterey, CA 93943, September, 1995.

[BACH96] Bachmann, E.R., et al., "Evaluation of an Integrated GPS/INS System for Shallow-Water AUV Navigation (SANS)," Symposium on Autonomous Underwater Vehicle Technology, Monterey, California, June 3-6, 1996

[BRUT92] Brutzman, Donald P. et al., "Autonomous Sonar Classification Using Expert Systems," Proceedings of the IEEE Oceanic Engineering Society Conference OCEANS 92, Newport Rhode Island, October 26-29, 1992, pp 554-559. Available at, ftp://taurus.cs.nps.navy.mil/pub/auv/oceans92.ps.Z

[BRUT95] Brutzman, Donald P., "Virtual World Visualization for an Autonomous Underwater Vehicle," Proceedings of the IEEE Oceanic Engineering Society Conference OCEANS 95, San Diego California, October 12-15 1995, pp 1592-1600. Available at, ftp://taurus.cs.nps.navy.mil/pub/auv/oceans95.ps.Z

[BRUT96] Brutzman, Donald P., et al., "NPS Phoenix AUV Software Integration and In-Water Testing," Center for Autonomous Underwater Vehicle Research, Code UW/Br, Naval Postgraduate School, Monterey California. Available at, http://www.cs.nps.navy.mil/research/auv

[BURN96] Burns, Mike, An Experimental Evaluation and Modification of Simulator-based Vehicle Control Software for the Phoenix Autonomous Underwater Vehicle (AUV)," M.S. thesis, Naval Postgraduate School, Monterey, CA 93943, March, 1996. Available at http://www.cs.nps.navy.mil/research/auv

[BYRN96] Byrnes, R.B. et al., "The Rational Behavior Model Software Architecture for Intelligent Ships," Naval Engineers Journal, March 1996, pp. 43-55

[CAMP96] Campbell, Mike, Real-Time Sonar Classification for Autonomous Underwater Vehicles, " M.S. Thesis, Naval Postgraduate School, Monterey, CA 93943, March, 1996. Available at, http://www.cs.nps.navy.mil/research/auv

[CRAI86] Craig, John, J., "Introduction to Robotics, Mechanics and Control, Second Edition,"Addison-Wesley Publishing, Company, Reading, Massachusetts, 1986.

[FLAG94] Flagg, Marco, "Submersible Computer for Divers, Autonomous Applications," Sea technology, vol. 35, February 1994, pp. 33-37.

[GELB88] Gelb, A., "Applied Optimal Estimation," MIT Press, 1988.

[HOLD95] Holden, Michael J., "ADA Implementation of Concurrent Execution of Multiple Tasks in the Strategic and Tactical Levels of the Rational Behavior Model for the NPS Phoenix Autonomous Underwater Vehicle (AUV),"M.S. thesis, Naval Postgraduate School, Monterey, CA 93943, September, 1995. Available at, http://www.cs.nps.navy.mil/research/auv

[LACH96] Lachapelle, G., et al., "Shipboard Attitude Determination During MMST-93," IEEE Journal of Oceanic engineering, Vol 21 January 1996, pp 100-105.

[LEON96] Leonhardt, Bradley J., "Mission Planning and Mission Control software for the Phoenix Autonomous Underwater Vehicle (AUV): Implementation and Experimental Study," M.S. thesis, Naval Postgraduate School, Monterey, CA 93943, March, 1996. Available at, http://www.cs.nps.navy.mil/research/auv

[MARC96] Marco, D.B., et al, "Autonomous Underwater Vehicles: Hybrid Control of Mission and Motion," Journal of Autonomous Robots, 1996.

[MCGH95] McGhee, R.B., et al., "An Experimental Study of an Integrated GPS/INS System for Shallow-Water AUV Navigation (SANS)", Proceedings of the Ninth International Symposium on Unmanned Untethered Submersible Technology (UUST), September 25-27 1995, Durham, NH. pp 153-167.

[MOTORA] Anon., Oncore 8 Channel GPS Receiver, Motorola, Inc., Schaumburg, Ill 60196.

[SCRI96] Scrivener Art., "Acoustic Navigation of the Phoenix Autonomous Underwater Vehicle using the DiveTracker System," M.S. Thesis, Naval Postgraduate School, Monterey, CA 93943, March, 1996.

[SCSII] Anon., "scsiServer Solari 2 Software Installation Guide," Central Data Corporation, Champaign, Ill., 1995.

[SUN] Anon., SPARCstation Voyager, Sun Microsystems, Inc., Mountain View, CA., 94043.

[TRITEC] Anon., TriTech ST750, ST1000 Sonars, Tritech Int'l Ltd., Aberdeen AB32 6JL, UK.

[WASH94] Washburn, Alan, OA4607: Lecture Notes, A Short Introduction to Kalman Filters, Naval Postgraduate School, Monterey California, July 1994.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center. 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Computer Technology Programs, Code CS 1
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Dr. Ted Lewis, Code CS/Lt 1
   Chair, Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5100

5. Dr. Robert McGhee, Code CS/Mz 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5100

6. Dr. Donald P. Brutzman, Code UW/Br 2
   Undersea Warfare Academic Group
   Naval Postgraduate School
   Monterey, CA 93943-5100

7. Dr. Anthony J. Healey, Code ME/Hy 1
   Mechanical Engineering Department
   Naval Postgraduate School
   Monterey CA, 93943-5100

8. CDR Michael J. Holden, USN, Code CS/Hm. 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5100

9. Dr. Alan Washburn, Code OR/WS 1
Operation Research Department
Naval Postgraduate School
Monterey CA, 93943-5100

10. David Marco, Code ME/MA 1
Mechanical Engineering Department
Naval Postgraduate School
Monterey, California 93943-5000

11. Russell Whalen, Code CS 1
Computer Science Department
Naval Postgraduate School
Monterey, California 93943-5000

12. Dr. Richard Blidberg, Director 1
Marine Systems Engineering Laboratory, Marine Science Center
Northeastern University
East Point, Nahant, Massachusetts 01908

13. Dr. James Bellingham 1
Underwater Vehicles Laboratory, MIT Sea Grant College Program
292 Main Street
Massachusetts Institute of Technology
Cambridge Massachusetts 02142

14. Mr. Norman Caplan 1
National Science Foundation
BES, Room 565
4201 Wilson Blvd.
Arlington, Virginia 22230

15. LT Eric Bachmann, USN, Code CS 1
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5100

16. Dr. James Eagle, Code OR/ER 1
Chair, Undersea Warfare Department
Naval Postgraduate School
Monterey, CA 93943-5100

17. LT, David McClarin 1
3605 E. Florence
P.O. Box 163
Mead, Washington, 99201