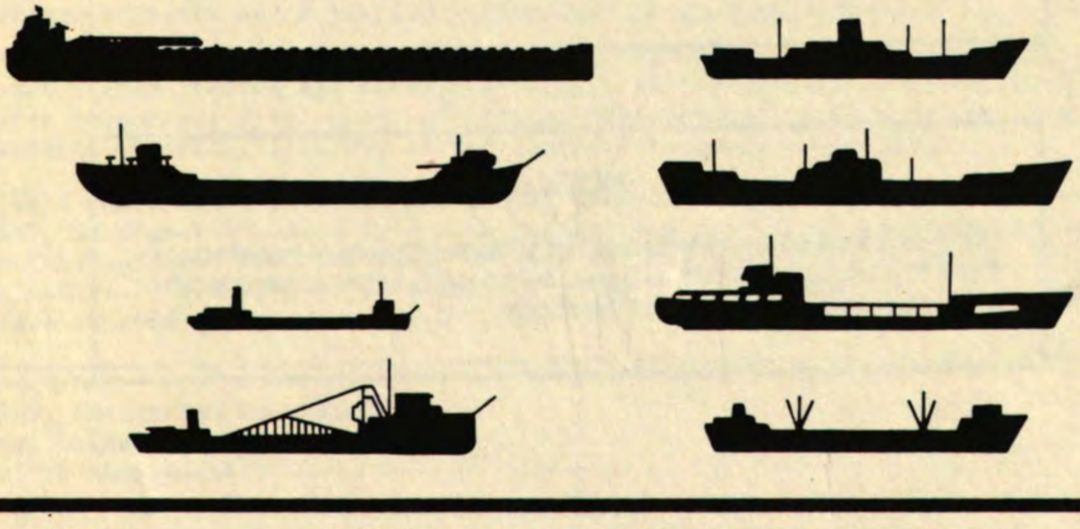
ent
ition

ion

Maritime Navigation/ Communications Program
VOLUME III: STATE OF THE ART SURVEY

Office of Technology Assessment
Washington, DC 20590

Transportation Systems Center
Center for Navigation



MA-RD-840-89005
DOT-TSC-RSPA-89-2

April 1989
Final Report

LEGAL NOTICE

This report was prepared as an account of government-sponsored work. Neither the United States, nor the Maritime Administration, nor any person acting on behalf of the Maritime Administration (A) Makes any warranty or representation, expressed or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or (B) Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report. As used in the above, "persons acting on behalf of the Maritime Administration" includes any employee or contract of the Maritime Administration to the extent that such employee or contractor prepares, handles, or distributes, or provides access to, any information pursuant to his employment or contract with the Maritime Administration.

NOTICE

This document is disseminated under the sponsorship of the Department of Transportation in the interest of information exchange. The United States Government assumes no liability for its contents or use thereof.

NOTICE

The United States Government does not endorse products of manufacturers. Trade of manufacturers'names appear herein solely because they are considered essential to the object of this report.

Technical Report Documentation Page

1. Report No.
MA-RD-840-89

2. Government Accession No.
PB89-190 953/GAR

3. Recipient's Catalog No.
78641

4. Title and Subtitle
MARITIME NAVIGATION/COMMUNICATIONS PROGRAM VOLUME III: STATE OF THE ART SURVEY

5. Report Date
March 1989

6. Performing Organization Code
DTS-52

7. Author(s)
Dr. J. W. Sennot*

8. Performing Organization Report No.
DOT-TSC-RSPA-89

9. Performing Organization Name and Address
U.S. Department of Transportation
Research and Special Programs Administration
Transportation Systems Center
Cambridge, MA 02142

10. Work Unit No. (TRAIS)
MA902-W9018

11. Contract or Grant No.
DOT-DTRS-57-85-C-0090

12. Sponsoring Agency Name and Address
U.S. Department of Transportation
Maritime Administration
Office of Technology Assessment
Washington, D.C. 20590

13. Type of Report and Period Covered
Final Report
March 1987 - March 1988

14. Sponsoring Agency Code
MAR-840

15. Supplementary Notes
*Department of Electrical and Computer Engineering
Bradley University
Peoria, IL 61625

16. Abstract

A MARAD/TSC team has been conducting a program to study navigation and communications systems on the Great Lakes and St. Lawrence River with the objective of defining technologies and systems which have the potential to increase economic benefits to the users, and support both National Defense and National Strategic Goals.

Volume I of this study identified shortcomings in existing systems, namely dependence on visual aids for navigation, congestion in the communications system, and lack of timely environmental data. Volume II contains the requirements for electronic navigation systems that supplement visual aids, additional communications capacity and a traffic information management system.

This report addresses navigation technology which could improve the efficiency of marine transportation and safety of vessels during transit of the waterways. The report recommends development of Federal Radionavigation Plan performance standards specifically aimed at the confined waterway environment.

A field measurement program, previously described in documentation submitted to the MARAD, is urgently needed to assure definitized navigation requirements on the Great Lakes-Saint Lawrence River network and to provide the framework for the evaluation of sensor configurations with the greatest potential for economic and safety related gains.

17. Key Words
MARAD, Navigation Communications,
Great Lakes, St. Lawrence River,
Traffic Management

18. Distribution Statement

19. Security Classif. (of this report)
UNCLASSIFIED

20. Security Classif. (of this page)
UNCLASSIFIED

21. No. of Pages
179

22. Price

Reproduction of completed page authorized

Preface

This report addresses navigation technology aspects of the MARAD Great Lakes - St. Lawrence Seaway Navigation/Communication Program. Recent navigation technology developments in the areas of differential GPS, RACON, and microwave ranging offer accuracies which could improve marine efficiency and vessel safety during transit of these waterways.

An overview and bibliography of ground and space-based electronic candidate nav aids suitable for confined waterways is given followed by a discussion of alternative methods of performance analysis, including sensor accuracy prediction and closed-loop pilotage simulation techniques. Selected for system assessment is a closed loop method which replaces the pilot/display portion of the system with a steering law tuned to obtain maximum performance from each candidate nav-aid combination. A steering algorithm and simulation software is then developed and verified in Z 20-20 turning maneuvers, using previous USCG and MARAD results.

A 14-state navigation filter is implemented for driving the steering algorithm. Inputs include a heading sensor and rudder position indicator, and up to four channels of radio nav-aid data. Sensor dynamics are modeled within both the environment and navigation filter. A ship footprint clearance width performance measure is developed, and used to compile channel width distributions for DGPS, differential LORAN-C, RACON and microwave systems. Distributions are obtained for two levels of ship disturbances, in both turn approach and turn recovery regions. The contribution of nav-aid subsystems to the overall waterway channel-width error budget is obtained.

Based upon DGPS and LORAN-C results, the report recommends development of Federal Radio Navigation Plan (FRP) performance standards specifically aimed at the confined waterway steering environment. Field measurements of the most promising sensor configurations is recommended, along with human factors display simulations incorporating the sensor models and navigation filter software of the present simulation system.

This work was sponsored under U.S. Department of Transportation contract DTRS-57-85-C-0090, in support of the U.S. Maritime Administration. The authors wish to thank the staff of DOT/TSC for their patience, guidance and support during this endeavor. Frank Mackenzie was the project monitor.

METRIC / ENGLISH CONVERSION FACTORS

ENGLISH TO METRIC

LENGTH (APPROXIMATE)

1 inch (in) = 2.5 centimeters (cm)
1 foot (ft) = 30 centimeters (cm)
1 yard (yd) = 0.9 meter (m)
1 mile (mi) = 1.6 kilometers (km)

AREA (APPROXIMATE)

1 square inch (sq in, in^2) = 6.5 square centimeters (cm^2)
1 square foot (sq ft, ft^2) = 0.09 square meter (m^2)
1 square yard (sq yd, yd^2) = 0.8 square meter (m^2)
1 square mile (sq mi, mi^2) = 2.6 square kilometers (km^2)
1 acre = 0.4 hectares (he) = 4,000 square meters (m^2)

MASS - WEIGHT (APPROXIMATE)

1 ounce (oz) = 28 grams (gr)
1 pound (lb) = .45 kilogram (kg)
1 short ton = 2,000 pounds (lb) = 0.9 tonne (t)

VOLUME (APPROXIMATE)

1 teaspoon (tsp) = 5 milliliters (ml)
1 tablespoon (tbsp) = 15 milliliters (ml)
1 fluid ounce (fl oz) = 30 milliliters (ml)
1 cup (c) = 0.24 liter (l)
1 pint (pt) = 0.47 liter (l)
1 quart (qt) = 0.96 liter (l)
1 gallon (gal) = 3.8 liters (l)
1 cubic foot (cu ft, ft^3) = 0.03 cubic meter (m^3)
1 cubic yard (cu yd, yd^3) = 0.76 cubic meter (m^3)

TEMPERATURE (EXACT)

[(x - 32)(5/9)] °F = y °C

METRIC TO ENGLISH

LENGTH (APPROXIMATE)

1 millimeter (mm) = 0.04 inch (in)
1 centimeter (cm) = 0.4 inch (in)
1 meter (m) = 3.3 feet (ft)
1 meter (m) = 1.1 yards (yd)
1 kilometer (km) = 0.6 mile (mi)

AREA (APPROXIMATE)

1 square centimeter (cm^2) = 0.16 square inch (sq in, in^2)
1 square meter (m^2) = 1.2 square yards (sq yd, yd^2)
1 square kilometer (km^2) = 0.4 square mile (sq mi, mi^2)
1 hectare (he) = 10,000 square meters (m^2) = 2.5 acres

MASS - WEIGHT (APPROXIMATE)

1 gram (gr) = 0.036 ounce (oz)
1 kilogram (kg) = 2.2 pounds (lb)
1 tonne (t) = 1,000 kilograms (kg) = 1.1 short tons

VOLUME (APPROXIMATE)

1 milliliter (ml) = 0.03 fluid ounce (fl oz)
1 liter (l) = 2.1 pints (pt)
1 liter (l) = 1.06 quarts (qt)
1 liter (l) = 0.26 gallon (gal)
1 cubic meter (m^3) = 36 cubic feet (cu ft, ft^3)
1 cubic meter (m^3) = 1.3 cubic yards (cu yd, yd^3)

TEMPERATURE (EXACT)

[(9/5) y + 32] °C = x °F

QUICK INCH-CENTIMETER LENGTH CONVERSION

INCHES 0 1 2 3 4 5 6 7 8 9 10

CENTIMETERS 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
25.40

QUICK FAHRENHEIT-CELCIUS TEMPERATURE CONVERSION

°F	-40°	-22°	-4°	14°	32°	50°	68°	86°	104°	122°	140°	158°	176°	194°	212°
°C	-40°	-30°	-20°	-10°	0°	10°	20°	30°	40°	50°	60°	70°	80°	90°	100°

For more exact and/or other conversion factors, see NBS Miscellaneous Publication 286, Units of Weights and Measures. Price $2.50. SD Catalog No. C13 10 286.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

This report addresses navigation technology aspects of the MARAD Great Lakes - St. Lawrence Seaway Navigation/Communication Program. Recent navigation technology developments in the areas of differential GPS, RACON, and microwave ranging offer accuracies which could improve marine efficiency and vessel safety during transit of the waterways of interest.

Several of the aids evaluated in this report have previously been evaluated using human factors simulators. However, sensor performance requirements are costly to develop with this approach. Errors in vessel cross-track and heading are difficult to apportion into human pilotage, display format, sensor dynamics and electromagnetic noise components.

Recently a methodology for standardizing sensor specifications and estimating vessel closed loop performance has been applied to the marine problem [A-2]. This analytical and computer simulation approach for accuracy assessment replaces the pilot/display portion of the system with a control law tuned to obtain maximum performance for each candidate nav-aid. While in an implemented system such a controller might only be utilized to display predictions of vessel footprint, an experienced pilot might approach the absolute performance obtained by the controller. Those sensor systems exhibiting poor control performance in simulation would likely prove unacceptable for driving pilot displays in the actual shipboard environment. Reducing the cost of comparing a large variety of sensor combinations, sensor candidates showing most promise may be tested in a complete human factors simulation.

The accuracy assessment model is shown in **Figure 1-1**. In the analysis and simulation work to follow the overall system is partitioned into the environment portion and the navigation and control gain portion:

- Ship and Sensor Environment Portion
    - Vessel Steering Dynamics and Disturbance Model
    - Water Current Model
    - Raw Sensor Dynamics and Sensor Noise Model
    - User Clock Model
    - Waypoint Steering Geometry
- Navigation and Control Portion
    - Ship and Sensor State Variable Model
    - Navigation Filter, Including Rudder Feedback

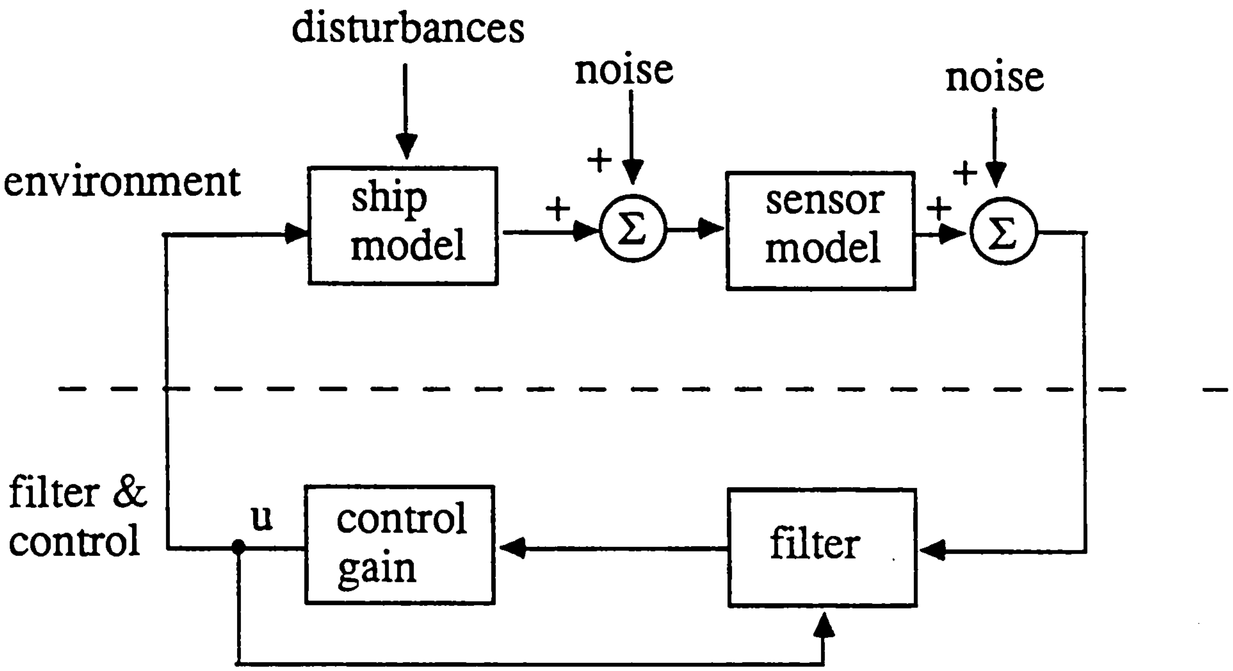


Figure 1-1   Accuracy Assessment Model

- Optimal Rudder Controller
- Waypoint Switching Logic

Implicit are multiple nav-aids, as well as a specified set of waypoint segments. The sensor portion of the model includes dynamical and noise models for RACON, differential GPS (DGPS), UHF/microwave multilateration, LORAN-C, and heading reference.

Prior to establishing the model in detail, **Section 2** of the report gives a comprehensive overview of both ground and space-based electronic candidate nav aids suitable for confined waterways. Particular attention is given to the recent developments in marine differential GPS (DGPS). **Section 3** reviews the ship displays that have been investigated for precision ship guidance. In **Section 4** alternative methods of performance analysis are addressed, including a variety of sensor accuracy prediction and closed-loop pilotage techniques. Offering the best balance of model fidelity and computational efficiency, the approach based upon optimal stochastic rudder control is selected.

In **section 5** the ship and sensor environment state models are developed, including the effects of sensor tracking loop dynamics. Ship steering model dynamics are then verified in Z 20-20 turning maneuvers, using as a reference the turning results of previous USCG and MARAD simulations. A complete 14-state navigation filter is developed and incorporated into the ship navigation simulation. An optimal steering control law which seeks to minimize a weighted average of cross-track and heading error, relative to the local waypoint coordinate system, is then developed. This control performance index tends to minimize the channel width needed for safe ship passage or, equivalently, minimizes the probability of grounding to either side of the channel.

Section 6 presents the simulation results for four sensor families, DGPS, LORAN-C, RACON and microwave. A channel clearance width performance measure is developed, and used to compile cumulative distributions for required channel width. These distributions are compared against those obtained using perfect sensors, and in so doing the contribution of the sensor systems to the overall channel width budget is determined.
The report concludes with recommendations for development of nav-aid performance standards suited to the confined waterway environment, and for further simulation and field measurements of the most promising sensor configurations.

2. NAVIGATION SENSORS

Among the navigation sensors proposed for driving ship displays and rudder controllers are aided/differential LORAN-C, RACON radar beacon, primary radar with corner reflectors, primary radar with clutter maps, UHF and X-band multilateration, and differential GPS. The general characteristics of these candidate sensors are discussed below. In **section 6** representative sensors are assessed with a ship waypoint steering simulation.

Sensors are grouped into ground-based and space-based families. These are further subdivided according to the type of "raw" measurement produced. It should be kept in mind that most electronic aids are of the point positioning variety. Also vital for vessel control is ship orientation, or heading. Using dual antenna configurations it is possible to obtain heading with multilateration and GPS systems. Multiple RACON measurements can also be employed to derive heading. However, for all options an auxiliary heading reference sensor will be considered.

2.1 Overview of Ground-Based Candidate Systems

2.1.1 Precision Ranging Systems

These systems employ a round-trip measurement technique to determine range to several shore-side transponders. UHF and microwave ranging systems are available from many manufacturers to meet the specialized needs of local surveying and navigation. Commercially available UHF and microwave ranging systems are capable of 1 meter accuracy. Accuracy is limited primarily by uncertainty in the propagation medium, including the effects of multipath. Careful navigation filter sensor integration can reduce the effects of temporary signal blockage and strong multipath. However, difficult shore-side terrain may require a substantial investment in the transponder grid.

2.1.1.1 Microwave [E-3],[E-9],[E-11]

One of the most widely accepted systems, the Del Norte Trisponder, is available in both X-band and UHF configurations, with ranges to 80 Km for the UHF variant. At the microwave frequency the dominant error source is receiver pulse timing jitter, limiting accuracy to the one meter level. Range propagation biases are small, but special caution must be observed in protecting the navigation solution from large signal dropouts and multipath events.

2.1.1.2 UHF [E-3], [E-15]

These systems typically operate just above 400 MHz, and achieve maximum ranges several times that of microwave systems. Within LOS, UHF frequencies behave much as the previous systems and can produce stable measurements within one or two meters accuracy. Beyond LOS, a UHF signal may come into contact with a reflective temperature inversion layer just above the water or land that forms a captive passage or duct. Unable to escape, the signal must follow a path which approximates the curvature of the earth, and very consistently reaches vehicles at ranges two and three times LOS. At some cost in accuracy, the UHF systems provide far more efficient area-wide coverage than the microwave variants.

2.1.2 Rho-Theta Systems

Rho-theta systems share many of the features of the above ranging systems. In addition, they provide to the navigation filter a bearing angle measurement. As a family, these systems are easily integrated into the ship-board surveillance radar display. The systems are subdivided into primary and radar beacon categories. Primary radar requires little or no cooperation from shore-based elements, and is potentially very attractive from a cost standpoint. Techniques are under study which might provide sufficiently reliable ranging and bearing angle measurements for navigation filter use. Radar beacon variants are active in nature, requiring much the same shore-side support as the previously discussed ranging systems.

2.1.2.1 Primary Radar [E-1], [E-5],[E-6]

Radar is an invaluable navigation aid for shipping. In difficult weather with reduced visibility, the radar is often the only instrument that enables ships to move in busy waters. The range of a large ship's radar is usually over 20 nautical miles, which means that action can be taken at an early enough stage to avoid dangerous situations.

As of September 1, 1986, all ships over 40,000 tons had to conform with the International Maritime Organization requirements for automatic radar plotting aids (ARPA). This was the latest deadline in a progressive schedule for retrofitting older vessels with modern radar equipment. The rules already applied to all new vessels built since 1984, and to all tankers, old and new, over 10,000 grt.

Provided the channel boundaries can be unamiguously defined on the radar display, short range primary radar offer the potential for direct vessel control from the radar display.

Human factors simulations of vessel steerage using primary radar have been carried out [D-7]. However, it is desirable to automatically process range and bearing angles within a navigation filter. The resulting state estimates can then be used to drive digital map displays.

Two techniques have been suggested for precisely registering the primary radar with the shore datum. The first of these techniques employs polarized corner reflectors and a specially polarized ship radar antenna [E-5]. In early evaluations of this variant [E-6], acquisition and tracking of targets was judged not acceptable. Under dynamical conditions tracking biases were also observed. The second radar data registration technique, real-time correlation of the radar image with a previously stored radar clutter map [E-1], is derivative of image processing and tracking technologies employed in military systems. While preliminary results look encouraging, problems of mismatch between the clutter mapping radar and the ship radar, as well as the currency of the clutter map data base, are still outstanding problem areas.

2.1.2.2 Radar Beacon [E-7],[E-12],[E-14]

Primary radar echoes from small buoys and navigation markers are often masked by noise or sea clutter, and in complicated passages the display can be difficult to interpret. Ever since radar was introduced there has therefore been a need for equipment which can improve the possibility of detecting specific targets and also make identification easier. Passive reflectors, which give a larger target area, were available quite early on and are still widely used. However, reflectors only give a stronger echo and do not make identification any easier.

RACON is a secondary radar system which responds at the frequency used by the searching radar with a coded or modulated response, and is generally compatible with the standard short-range ship radar. The output of this system may be displayed on the ships PPI, and can also be interfaced with other displays and the navigation computer. Because of its coded response signature, the system overcomes the registration problems of primary radar. It also provides a nearly multipath-free response, enabling more precise estimates of range and bearing angle.

The second generation AGA-ERICON radar beacon, developed by the Defense and Space Systems Division of LM Ericsson, in close collaboration with the Swedish Administration of Shipping and Navigation, intercepts pulses from an X- or D-band radar and responds at the same frequency with a high degree of frequency accuracy and little delay [E-14]. The length of the response signal can be programmed to any desired value, and it can easily be coded in Morse or some other system. Since the RACON always responds at the frequency of the searching radar, the whole of the transmitted power is available in the radar receiver bandwidth. This reduces the sensitivity to different types of clutter. The waiting time between responses is short and the RACON response can be displayed during each antenna rotation.

In October 1980 a prototype of AGA-ERICON was installed on Trubaduren, a caisson lighthouse at the entrance to the harbor of Gothenburg, on the west coast of Sweden. It replaced an older, slow sweep X-band RACON and a fast sweep S-band RACON. It is programmed to respond for 30 seconds per minute and is equipped with a side lobe memory.

The Coast Guard presently has approximately 80 RACONs in operation at various locations and has 30 more on order. They currently operate various RACON types but will standardize on the frequency agile types. A U.S. Coast Guard final rule was published in the Federal Register on April 3, 1986 authorizing the operation of private RACONs. Requests from the offshore industry, and favorable experience with Federal RACONS, caused the Coast Guard to recognize the desirability of private RACONS. Previously, all private marine electronic aids to navigation, with the exception of shore based radar systems, were prohibited.

When used with a standard ships radar, RACON accuracy is primariliy limited by range jitter in the ship interrogator, and by azimuth measurement error. Typical azimuth angle measurements are good to about one degree. This translates into a 17 meter position error at 1000 meters, unacceptable in confined waterways. However, angular accuracy can be improved by an order of magnitude by fitting the ship with a more complex monopulse radar system [E-17].

2.1.3 Hyperbolic Systems

The hyperbolic family is based upon time-of-arrival measurements by a passive user. First developed in WWII, several generations of hyperbolic systems have been installed and operated by US and foreign governments.

2.1.3.1 LORAN-C

This system is under the control of the U.S. Coast Guard, and covers the entire U.S. coastline, as well as many other ocean areas, most notably the Mediterranean and large portions of the North Sea, Norwegian Sea and North Atlantic. Nominal geographical accuracy is 0.25 nautical miles, with repeatability accuracy (the ability to return to a given set of LORAN-C coordinates) on the order of 50-300 feet, according to the Coast Guard. Repeatability of time differences is maintained within each LORAN-C chain by a local area monitor, which transmits commands back to the master and slave stations. To promote cross-chain operation, and interoperability with other nav-aids such as GPS, efforts are underway to provide independant synchronization of LORAN masters and slaves, using a time transfer system such at GPS. This will somewhat degrade the repeatable accuracies presently obtainable with the system.

LORAN-C signals are subject to errors caused by distortion from irregular land masses, local weather conditions, seasonal and daily variations, system geometry and interference from other radio signals on nearby wavelengths. Some of these (such as the land-path distortion) are predictable and can be removed by readjusting the LORAN-C lines of position on the appropriate nautical charts, or by reprogramming the LORAN-C receivers themselves. Other sources of error can be cancelled out using the cited differential techniques, which involves setting up fixed monitoring stations at known locations. These monitors constantly compared the received signals against their known positions, and then rebroadcast the necessary corrections. The mobile receiver automatically receives the corrections and applies them to the received signals' anomalies.

The U.S. Coast Guard has just completed a research program evaluating the feasibility of differential techniques to enhance the accuracy of the LORAN-C radionavigation system. To demonstrate the system, the Coast Guard held a series of trials aboard the buoy tender Cowslip in Hampton Roads harbor in July 1986. For the differential LORAN-C demonstrations aboard the Cowslip, the Coast Guard set up three monitoring sites in Hampton Roads harbor. Each monitor consisted of a high-resolution, survey-grade

LORAN-C receiver, with two small Hewlett Packard computers. The LORAN-C corrections were automatically calculated and sent through a modem to a UHF radio transmitter, which broadcast these values very three minutes to the mobile receiver aboard the ship.

On the Cowslip, the differential LORAN-C signals were received through a similar UHF transceiver and modem. The ship was also fitted with a computer-based buoy positioning system, with which the differential LORAN-C was integrated. The ship's conning officer drove the vessel carefully through a moderate breeze and slight chop, nosing her buoy tending port up a buoy. The relative positions of the ship and buoy could be monitored on color CRT screens on each bridge wing, along with readouts of essential information. On the initial approach the differential LORAN-C inputs were used. Once the buoy was alongside, the LORAN-C position was compared against the position calculated by the computer from a round of horizontal sextant angles. The static difference was about 12 yards. To show the comparison between the differential LORAN-C position and the uncorrected LORAN-C coordinates, the differential LORAN-C computations were temporarily switched off. The offset was found to be about 60 yards.

Assuming all biases have been reduced to an acceptable level over a local area, the differential system is ultimately limited by the dynamical tracking accuracy of third cycle phase tracking, as influenced by ship motion and 100 kHz signal-to-noise ratio. This fundamental limit has been previously studied by theoretical means [F-7], [A-3]. While in many applications tracking loop jitter and dynamical lags can be ignored, for precision waterway applications these effects are of paramount importance, and are included in the simulation results of **section 6**.

2.2 Overview of Space-Based Candidate Systems

Historically, the first space-based navigation system was the TRANSIT system. Deployed by the U.S. Navy in the mid 60's, the TRANSIT system provides precise updates of position approximately every ninety minutes. In the early 70's a joint effort within the DOD led to the development of a much more powerful system, the Global Positioning System. Recently, numerous private sector activities aimed at real-time vehicle navigation and position reporting have appeared.

2.2.1 GPS

In a variety of applications, from high dynamics jet aircraft navigation, to baseline surveying to centimeter accuracies, the GPS has exceeded the expectation of its designers . The GPS system is operable in several modes. For military applications the system must be tolerant of heavy electronic jamming and be capable of performing well in a high-dynamics environment. This encripted P-code reception mode will not be available to civilian users, and will not be further considered. The civil community is expected to use the C/A signal transmissions, which offer excellent low-dynamics performance without excessive signal tracking noise and dynamical lags. The spread-spectrum signal structure provides resistance to RF interference and some resistance to multipath. However, signal blockage of low elevation satellites is of concern. In many locales it will be necessary to augment the satellite constellation with one or more local transmitters. These "pseudo-satellites" appear to the receiver as space satellites, and may be processed with the standard GPS receiver.

Of the three GPS modes the least accurate, "GPS/SA", gives an uncorrected position fix, using the measured arrival times on the C/A channel. The receiver processes these code arrival times and the broadcast satellite data, and outputs the vehicle state vector. The state vector may include position, velocity and acceleration components. In this mode the major errors are system biases, which are independent of receiver noise and dynamical lags. In phase II GPS spacecraft, the GPS ground tracking stations run by DOD will have the capability of introducing various errors into the C/A signal transmissions. To the user these errors will look like satellite clock and satellite ephemeris errors. Since these error sources are nearly identical for all receivers in a given region, local calibration techniques may be used to remove them, along with other errors due to ionospheric delay. This is the basis for the differential reception mode, DGPS. It is stated U.S. policy to deny the civilian community maximum performance in the non-differential C/A mode, limiting accuracy to 100 meters 2D rms, far outside acceptable limits for confined waterway operation.

2.2.1.1 Differential GPS, Code Mode

Using the "differential C/A code" reception mode, accuracies to several meters have been demonstrated on low-dynamics platforms [F-3]. Delta-range information is obtained by measuring the Doppler frequency shift of the L-band carrier. This enables velocity measurements good to several tenths of meters per second. For distances of up to several hundred km from the differential reference station the limiting factors on accuracy appear to be multipath and code loop tracking noise [F-5].

2.2.1.2 Differential GPS, Integrated Doppler Mode

The most accurate of the C/A GPS modes is differential/carrier [F-4]. This is based upon continuous tracking of signal phase on all satellites, and solution for the carrier phase ambiguity. At the L-band wavelength of 0.3 meters, the carrier phase observable is far more precise than the signal envelope measurement. In a multiple antenna receiver, the carrier tracking mode also enables very precise ship heading to be obtained. To date the carrier technique has been applied primarily in surveying applications, but preliminary real-time results rival the accuracies obtained with a microwave ranging system [F-3]. Real-time accuracies to the meter level, and better, appear possible. The most significant errors in this mode of operation arise from propagation path delays through the ionosphere, and satellite positioning errors, which tend to slowly decorrelate as the the separation between the reference station and the ship increases. The contribution from these sources is generally under 1 meter, out to distances of 100 km from the reference station [F-5].

3. Overview of Display Systems

[D-1]-[D-7]

A variety of pilot displays and navigation sensors have been suggested for use in confined waterways. Pilot displays ranging from simple digital readouts to accurate renditions of the pilot view seen from the ship's bridge have been examined with human factors simulations. Display formats have incorporated own-ship position and crosstrack error and velocity data, as well as predicted-track data.

4. Performance Evaluation Tools

Sensor selection involves recognition of many factors, including user and government capitalization, operation and maintenance costs, sensor reliability and sensor positioning performance. It is this last aspect which is of interest in this section of the report. Since the sensor system may consist of several devices which are integrated to provide the necessary estimates of vessel state, developing a suitably methodology for evaluating overall sensor performance is an important problem. Manufacturer specifications, such as data rate and 95 percentile cumulative error probability, are generally insufficient for predicting closed loop vessel control performance.

A variety of tools have been employed to evaluate and compare the performance of position-fixing devices. Not all of these methods are appropriate for predicting performance in the critical confined waterway application. Error sources which may be discounted in open ocean and harbor approach must be carefully modeled in confined waterways. It is therefor important to understand the rationale for selecting the optimal control simulation methodology over other available techniques. Evaluation options can be divided into two categories:

- Sensor Accuracy Prediction Methods
- Closed Loop Vessel Pilotage Performance Methods

With the first approach the sensor system of interest is analyzed in isolation from the vessel control. This leaves unanswered the level of pilotage accuracy which will ultimately be obtained, and may give misleading information as to the level of sensor accuracy required to meet overall vessel pilotage specifications. Three potential pitfalls of the isolated sensor accuracy approach are:

- Over-specification of sensor accuracy
- Excessive sensor smoothing/extrapolation to achieve optimistic accuracy and update-rate performance, and
- Incomplete picture of combined multi-sensor performance.

Over-specification of needed sensor accuracy occurs when the vehicle under control is inherently difficult to control. Even with perfect knowledge of vessel position, velocity and orientation, the vessel footprint cannot be positioned arbitrarily well. For example, a sensor accuracy specification of 3 meters makes little sense if, even with perfect knowledge of vessel state, pilotage error cannot be reduced to below 10 meters.

The second problem area relates to the application of excessive smoothing to achieve optimistic specifications. Utilizing long signal tracking time constants to achieve noise reduction will result in poor dynamical performance, and rapid sampling of smoothed sensor outputs will not provide independent estimates of ship position. While yielding impressive static performance figures, reliance on excessive smoothing and extrapolation may yield erroneous predictions of vessel dynamical pilotage performance.

Finally, care must be taken when predicting the performance of multi-sensor systems. Vessel state estimation performance must account for realistic dynamics, as encountered in the overall control problem. Before discussing the preferred method of analysis, it is worthwhile reviewing traditional methods of sensor accuracy prediction.

4.1 Sensor Accuracy Prediction Methods

4.1.1 Static GDOP Analysis

The geometric-dilution-of-precision (GDOP) method is the traditional means for predicting navigation and position-fixing performance. The position fix is of the "one-shot" variety. In essence, this method predicts performance by analysis of the crossing angles of lines-of-position (LOP). The underlying assumption is that observation errors (ranges, pseudo-ranges, bearing angles, and the like) are statistically independent. While the assumption of independent measurement noise is a reasonable one, the static nature of the analysis is too restrictive. In modern practice ship-board equipment employs navigation filtering in the position fixing process, leading in most circumstances to far better accuracies than predicted for the "one-shot" position fix.

4.1.2 Covariance Analysis

A more realistic prediction of sensor performance can be obtained by modeling the dynamics of the vessel with state-variables, permitting a prediction of navigation performance for the Kalman filter. Typical ship states utilized in such an exploration might be: sway velocity, yaw rate, geodetic heading, cross-track position, and along-track position. Sensor inputs are generally modeled as true ranges and bearings, plus additive noise. An underlying assumption is that the navigation filter knows precisely the parameters of the ship and sensor dynamical models.

The covariance method yields a statistical picture of accuracy averaged over a large number of vessels, for a series of random maneuvers. Little insight is gained regarding accuracies obtained for specific waypoint geometries and vessel turning maneuvers.

4.1.3 Sensor Simulation

The final method of sensor analysis treats specific waypoint geometries and turning maneuvers. A set of vessel trajectories are first obtained, either by simulation, or from field measurements of actual vessel maneuvers. For each trajectory of interest, a file of true range and bearing angle data is then generated. Depending upon the fidelity of the simulation, these "truth" files are then corrupted with simulated random noise, and may also be subjected to signal tracking lag effects. The resulting simulated raw sensor data is then used to drive an actual ship navigation filter, providing a very realistic picture of sensor performance. As explained previously, such an analysis fails to address the problem of overall vessel control accuracy. However, this approach is a basic building block employed in the closed-loop methods discussed below.

4.2 Closed Loop Vessel Pilotage Performance Methods

Vessel closed-loop performance analysis seeks to address the turning and course keeping accuracies obtainable from a given combination of sensors. Since achieved performance will depend upon such non-sensor factors as pilot skill level, pilot display type, vessel maneuverability, and wind and current disturbances, closed-loop performance analysis is more complex than the sensor-only techniques discussed above. Traditionally, such studies have been performed via human factors simulations, although recently computer simulation methods have been developed for exploration of sensor and vessel behavior in the closed loop mode.

4.2.1 Human Factors Simulation, Visual Nav-Aids

Simulation efforts have been carried out to study pilot and vessel response during various modes of pilotage. The influence of buoyage geometry and lighting on track-keeping and waypoint maneuvering had been extensively studied on the CAORF facility at the National Maritime Research Center, and at facilities operated for the U.S Coast Guard by Eclectech Associates [C-1]-[C-5],[C-6]-[C-10].

A basic scenario, typical of those studied at both facilities, involved a passage of two three mile legs, with a 35 degree turn. Channel width was 500 feet. The best buoy arrangement on the straight portion of course was found to be of the gated type, with gates spaced by

5/8 NM. Wind and current disturbance conditions were fixed between runs. During all reported tests the current on leg 1 was astern with a velocity of 11/2 knots, gradually decreasing on leg 2 to 3/4 knots on the port quarter. Winds on both simulators were 30 knots, gusting from astern on leg 1 and on the port quarter during leg 2. Pilot track-keeping results on both simulation facilities were very similar. Typical multi-run track-keeping standard deviations were 15 meters, with biases of 5 to 15 meters depending upon the relative wind and current vector. Under poorer visual aid conditions, staggered buoys 1 1/4 miles apart with 1 1/2 mile visibility, track-keeping standard deviation values increased to 25 meters.

On the turn to port from leg 1 to leg 2, best pilot visual results were obtained with three buoy cutoff turns and gated buoys on each leg. Vessel dynamics had an important effect during the turn, with the 80 dwt tanker experiencing more overshoot than the 30 dwt tanker. For the larger vessel typical multi-run standard deviations were 15 m, with a bias of 30 m to starboard. With staggered buoy conditions at the turn pullout, performance degraded to 25 m standard deviation, with a mean to starboard of 40 m. An important factor underlying these experiments was pilot familiarity with the vessel dynamics and bow image, and with the simulation test scenario.

The effect of underkeel clearance on turning performance received attention during some of these simulation efforts [10]. It was found, as clearances were decreased from 600 to one foot, that smaller clearance were helpful in reducing standard deviation on turns of under 35 degrees. The effect on larger turns was detrimental.

4.2.2 Human Factors Simulation, Electronic Displays [D-1]-[D-4],[D-6],[D-7]

4.2.3 At-Sea Evaluation, Visual Aids

In the Fall of 1980 an effort was made to measure track-keeping and turning performance of ships passing through the Craighill Channel and Craighill Channel Upper range of the Chesapeake Bay [C-4]. At this time a large number of vessels were continuously queued up waiting for coal loading in Baltimore, providing an excellent opportunity for *in situ* experimentation. The piloting conditions were similar to those reported in the simulation work. Channel width is approximately 600 feet on each leg, with a cutoff turn of 20 degrees. Buoyage is of the gated type with spacing of approximately 3/4 nm.

Data was taken on 21 ships, 14 of which were tracked by a properly calibrated Raydist low frequency position-fixing system. On 30 of the leg transits no traffic interference which could degrade track-following and turn performance was experienced. The ships were primarily coal carriers, but other vessel types were also included in the experiment. The aggregate track-keeping standard deviation for runs without vessel interference, including all weather and current conditions, was approximately 15 m. This compares favorably with the cited visual simulation results.

The data from the at-sea experiment were arranged into two wind speed groups, below and above 10 knots. Prevailing from the northwest, the winds typically were on the port bow during leg 1 and on the bow in leg 2. On the more difficult segment, leg 1, track-keeping standard deviations were about the same for each wind classification, indicating that on a straight course with gated buoyage pilots are easily able to adjust vessel helm to overcome wind disturbances. However, at the turn there was a very large difference between the two wind classification. In the under 10 knot group standard deviations of 20 m were obtained, consistent with simulation results. In the over 10 knot group, standard deviations increased to 90 m, much larger than the simulated results. Among the possible factors contributing to degraded pilotage may have been the variability of windage for different vessels, and the lack of vessel dynamics familiarity obtained on the brief passage from anchorage to the test range. The large difference between track-keeping and waypoint turning performance underscores the greater difficulty pilots have in estimating the vessel state and developing good rudder control inputs on turns.

4.2.4 Optimal Control Simulation

Recently a methodology for standardizing sensor specifications and estimating vessel closed loop performance has been applied to the marine problem [A-4]. Reducing the expense of comparing a large variety of sensor and display combinations, sensor candidates showing most promise may later be validated in a complete human factors simulation.

The analytical and computer simulation approach selected for accuracy assessment models the pilot/display portion of the system with a control law tuned to obtain maximum performance for each candidate nav-aid, for identical vessel dynamics and disturbances. While in an implemented system such a rudder controller might only be utilized to display predictions of vessel footprint, an experienced pilot should achieve an absolute performance level approaching that of the model. This is the methodology used in the remainder of this report.

5. The Ship Optimal Control Simulator

5.1 Objective

Comprising four major subsystems, ship model, sensor model, navigation filter, and rudder controller, the simulator offers an inexpensive means for comparing different electronic nav-aid combinations, as measured by ship footprint control accuracy. With an optimal control law as "stand in" for the human pilot, the rudder control and navigation filter are tuned to achieve the best performance from each sensor combination.

In the modeling process, considerable attention was given to the design of the navigation filter, and its integration with sensor and ship dynamics. Various rudder control methodologies were investigated, including both optimal deterministic and stochastic design methods. The ship dynamics model realized in both the environment and navigation filter was chosen with computational efficiency in mind. The selected model was tested against steering maneuver data published by MARAD and USCG simulation facilities.

As a human pilot navigates through a series of waypoints, **Figure 5-1**, a combination of factors are considered. If the potential for other traffic is low, the pilot generally applies rudder control so as to minimize the probability of violating the channel boundaries. In designing the rudder control and navigation filter elements of the simulator, analytical performance measures were reviewed. In the visual aid literature one finds a variety of measures including ship cross-track error [C-1]-[C-6] and relative risk factor [C-7],[C-8], and [C-10]. Unlike simple cross-track error, relative risk factor (RRF) considers the minimum clearance between the vessel footprint and the channel boundary, at specified along-track stations. Accounting for the vessel footprint orientation in the channel, the RRF measure is closely related to the probability of grounding and collision. In this report a similar measure is developed, Channel Clearance Width (CCW). The CCW function, plotted for the length of each simulation run, is the channel width needed such that the vessel would not have contacted either channel edge. This is depicted in **Figure 5-2**. In designing the ship navigation and rudder control system, the goal is to minimize CCW in one, or several, regions of the channel. CCW data can be used to compare the performance of the different sensor options, and can also be used to estimate the probability of safe vessel passage.

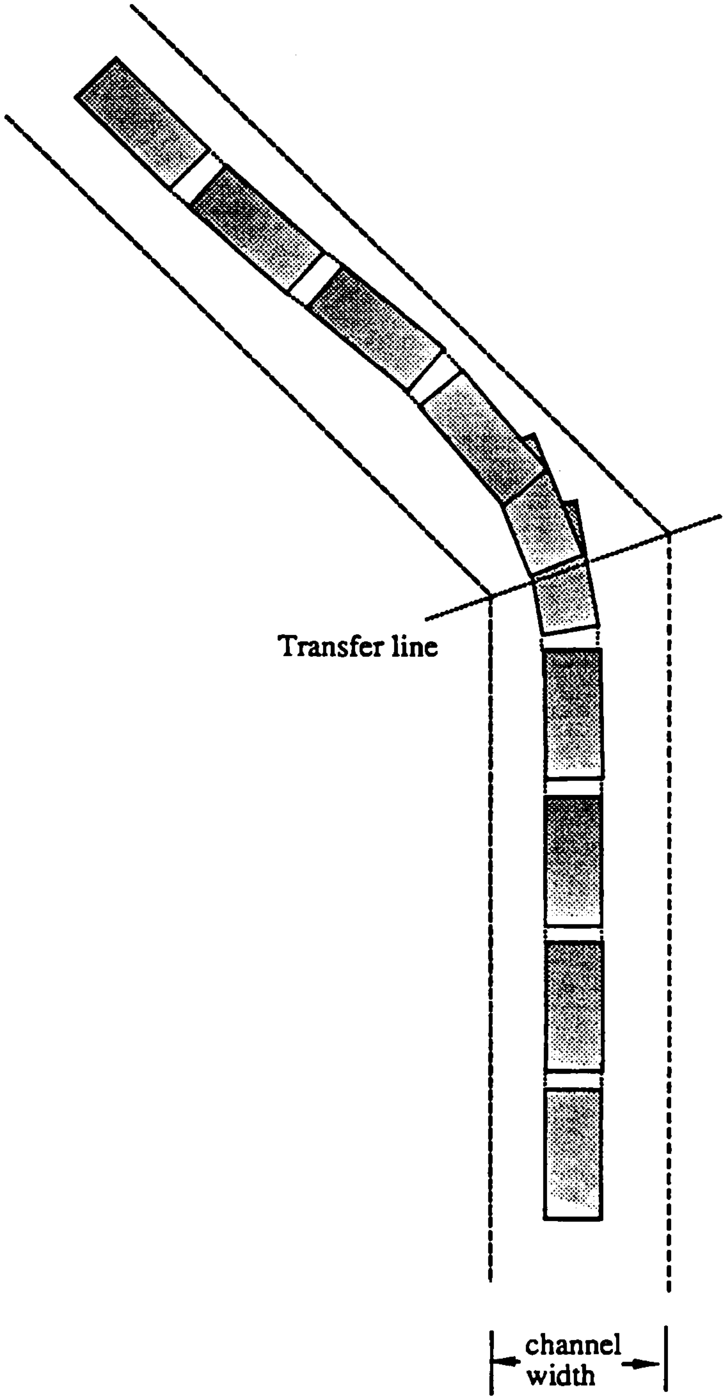
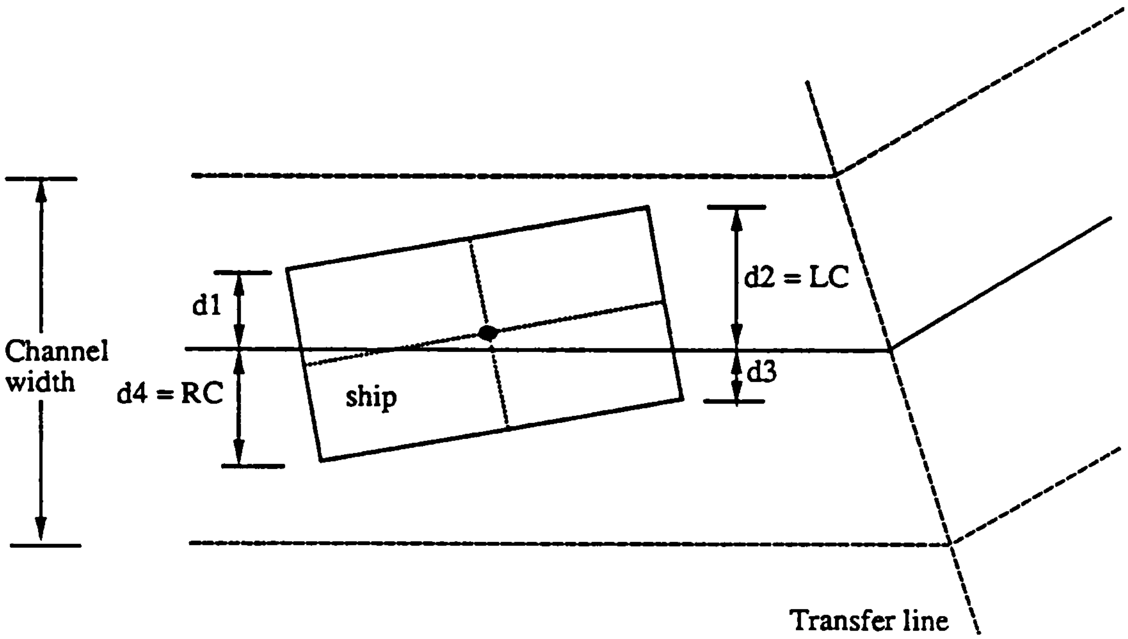


Figure 5-1 Pilot Waypoint Steering



LC = Left Clearance
RC = Right Clearance

Figure 5-2 Channel Clearance Width (CCW) Statistics

Figure 5-3 shows the coordinate systems and state variables used in both the truth environment and navigation filter and rudder control. The base of each waypoint leg defines a new waypoint coordinate system. Three states represent ship dynamics: sway velocity, yaw rate, and waypoint-referenced heading angle error, and two states register global position information: crosstrack, and alongtrack position. To add realism, two state variables represent easting and northing current components are included. Noise-like wind and ship modeling terms are injected on ship sway velocity and yaw rate axes.

Ship dynamics are recognized as being nonlinear and unstable. To implement the rudder control and navigation filter, a procedure for linearization is needed. To lessen the complexities of linearization a constant propeller thrust force and longitudinal velocity are assumed. In **section 5.2**, this ship state variable model will be developed in some detail.

In **section 5.3** a suitably general sensor model is developed. Including tracking loop dynamical lags, as encountered in such sensors ar LORAN-C, the model is linearized about a nominal ship state trajectory, leading to an overall state variable model for ship and sensors. **Section 5.4** gives an overview of the ship navigation filter, and **section 5.5** covers the rudder control design.

5.2 Ship Dynamics Model

From **Figure 5-3** the ship states x_i, $1 \leq i \leq 7$, are assigned as follows:

x_1 = sway velocity

x_2 = yaw rate

x_3 = heading angle error

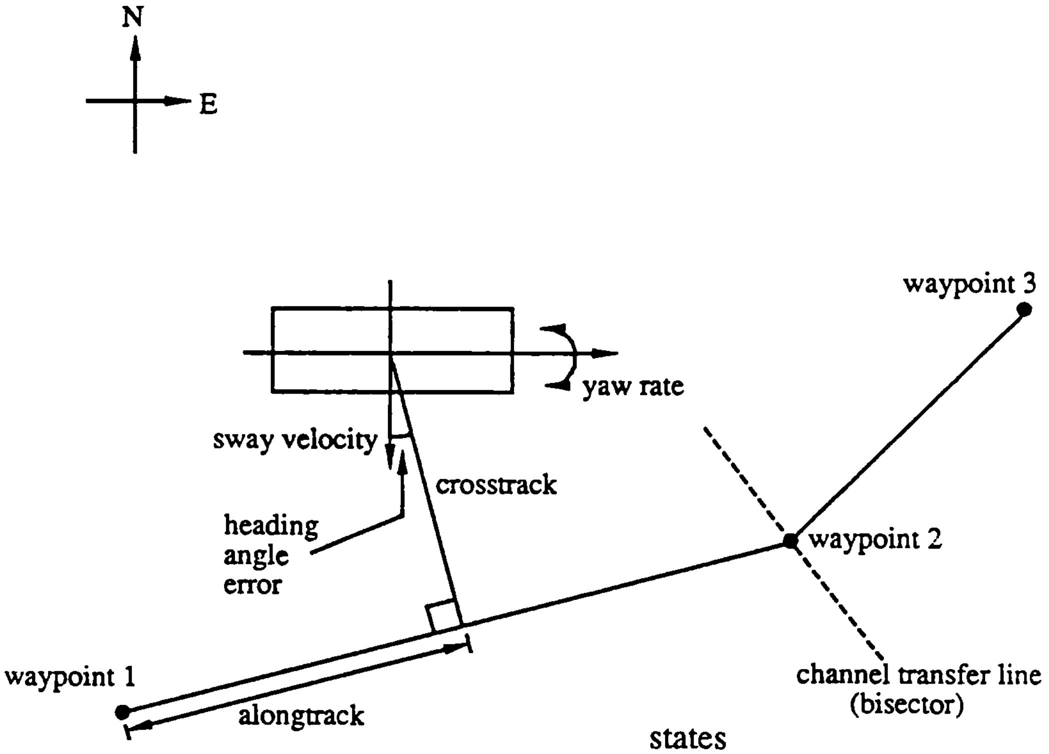
x_4 = crosstrack error on a waypoint segment

x_5 = alongtrack position on a waypoint segment

x_6 = easting current bias

x_7 = northing current bias

Then the ship dynamics can be written as below, as adapted from Goclowski and Gelb to accommodate the ship's alongtrack position in a local coordinate frame, in the presence of current **[B-9]**:



states

sway velocity
yaw rate
heading angle error
crosstrack
alongtrack
easting current
northing current

Figure 5-3 Coordinate System and State Definitions

$$\dot{x}_1 = -\frac{V}{L} C_{v\dot{v}}\, x_1 + V(C_{v\omega} - 1)\, x_2 - \frac{V^2}{L} C_{v\delta}\, \delta$$

$$\dot{x}_2 = -\frac{V}{L^2} C_{\omega\dot{v}}\, x_1 - \frac{V}{L} C_{\omega\omega} x_2 + \frac{V^2}{L^2} C_{\omega\delta} \delta$$

$$\dot{x}_3 = x_2$$

$$\dot{x}_4 = x_1 \cos x_3 + \cos\theta\, x_6 - \sin\theta\, x_7 + V \sin x_3$$

$$\dot{x}_5 = -x_1 \sin x_3 + \sin\theta\, x_6 + \cos\theta\, x_7 + V \cos x_3$$

$$\dot{x}_6 = -\alpha\, x_6 + \text{noise}$$

$$\dot{x}_7 = -\alpha\, x_7 + \text{noise}$$

where V is the ship's velocity in meter/second, L the ship length in meters, δ is the rudder control input in radian angle, C's are the hydrodynamic coefficients of the ship, **Appendix A**, θ is the heading of the present course leg, and α is the time constant for the first-order current model. In states x_1 and x_2 there is one positive eigenvalue. Hence, the rudder controller design must provide ship stability.

As a prelude to obtaining the rudder controller and navigation filter, a state space representation for the ship must be developed from the above non-linear state equations. In general the ship longitudinal velocity relative to the water could be included as a state. However, with the velocity V fixed, only the heading angle error x_3 must be linearized.

Linearization

A continuous-time incremental linear model can be derived by linearizing dynamics with respect to x_3 at a nominal operating point. Because of the rudder control, x_3 is maintained around zero, and zero can be used as the nominal operating point. Taking partial derivatives with respect x_3, at the operating point, the following state matrix coefficients are obtained:

$$a_{41} = \cos x_3 \quad , \quad a_{43} = - x_1 \sin x_3 + V \cos x_3$$

$$a_{51} = -\sin x_3 \quad , \quad a_{53} = - x_1 \cos x_3$$

and

$$b_{52} = \cos x_3$$

Then the incremental continuous-time linear model may be written in matrix form as:

$$\Delta\dot{\underline{x}} = A\,\Delta\underline{x} + B\underline{u}$$

where

$$\Delta\underline{x} = [\,\Delta x_1 \;\; \Delta x_2 \;\; \Delta x_3 \;\; \Delta x_4 \;\; \Delta x_5 \;\; \Delta x_6 \;\; \Delta x_7\,]^T \quad \text{, where T = transpose}$$

$$\underline{u} = [\,\delta \;\; V\,]^T$$

and A and B matrices are given by

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ a_{41} & 0 & a_{43} & 0 & 0 & \cos\theta & -\sin\theta \\ a_{51} & 0 & a_{53} & 0 & 0 & \sin\theta & \cos\theta \\ 0 & 0 & 0 & 0 & 0 & -\alpha & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\alpha \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & 0 \\ b_{21} & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & b_{52} \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

with

$$a_{11} = -\frac{V}{L} C_{v\dot{v}} \quad , \quad a_{12} = V(C_{v\omega} - 1)$$

$$a_{21} = -\frac{V}{L^2} C_{\omega\dot{v}} \quad , \quad a_{22} = -\frac{V}{L} C_{\omega\omega}$$

$$b_{11} = -\frac{V^2}{L} C_{v\delta} \quad , \quad b_{21} = \frac{V^2}{L^2} C_{\omega\delta}$$

and with the hydrodynamic coefficient C's as given in **Appendix A**.

Note that the term for crosstrack error is arranged in a subtly different way to avoid numerical instability problems. There are two forcing terms - rudder control and the velocity of the ship. The ship velocity is fixed. The rudder optimum control input is obtained by solving the Riccati gain equation for the rudder regulator **Appendix B**.

5.3 Sensor Model

To meaningfully compare different sensor combinations for the precise navigation problem of interest, a sensor state variable model is next developed. Among the basic sensors which may be selected in the simulation are range and pseudorange tracking loops, bearing angle measurements, and heading sensors.

As an example of the sensor modeling employed in the simulator, consider a basic range sensor which tracks distance from the ship to a ground-based transmitter. Typically this device is combined with an onboard heading angle sensor which indicates the ship's orientation with respect to the geodetic system. **Figure 5-4** depicts such a sensor geometry.

The angle x_3 is the true heading angle error, and the R_i, $1 \leq i \leq 4$, represent true distances from the ship to each of the sensors. These true range and angle values are <u>not</u> directly available to the navigation filter. The navigation filter only has access to these quantities via the navigation sensor hardware. For example, in the GPS system the ranges are provided as outputs from the delay-lock loop, and in LORAN-C the third cycle zero-crossing tracker provides the raw sensor output. In each case, substantial noise and dynamic lag terms can distort the true range values. It is therefore essential that these effects be included in the simulation models.

Considering the range-sensor model, let R_i, $1 \leq i \leq 4$, be the actual distance to the ith sensor. Then

$$R_i(t) = \sqrt{\{e_{ship}(t) - e_{sensor,i}(t)\}^2 + \{n_{ship}(t) - n_{sensor,i}(t)\}^2} \quad , \quad 1 \leq i \leq 4$$

where $e_{ship}(t)$ and $n_{ship}(t)$ denote the ship's easting and northing positions at time t, respectively. Similarly, $e_{sensor,i}(t)$ and $n_{sensor,i}(t)$ denote the ith sensor's easting and northing positions at time t, respectively. In this expression, $e_{ship}(t)$ and $n_{ship}(t)$ are non-linear functions of the ship along-track and cross-track states.

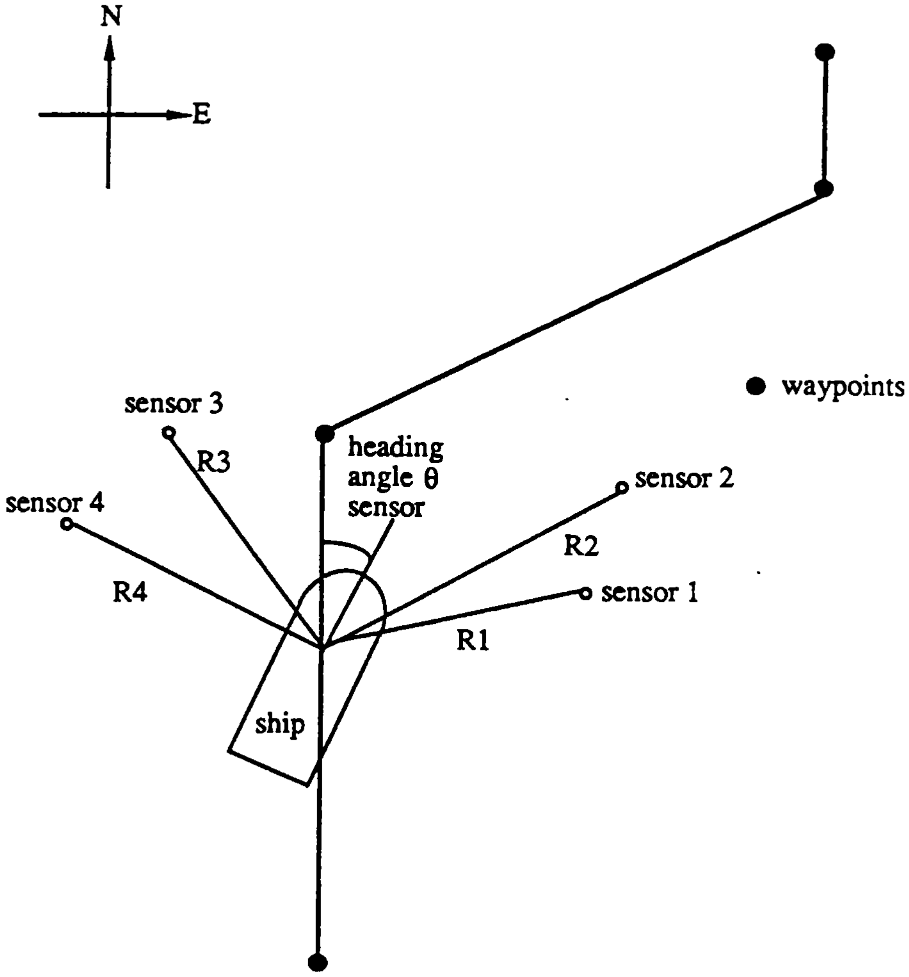


Figure 5-4 Typical Sensor Geometry

Now consider the sensor output as seen by the navigation filter. Assuming simple first-order tracking loop dynamics for each of four sensor ranging channels, the sensor state variable model is as follows:

$$\dot{x}_1(t) = -\alpha_1 x_1(t) + \alpha_1 R_1(t) + w_1(t) \quad , \quad x_1(0) = R_1(0)$$
$$\dot{x}_2(t) = -\alpha_2 x_2(t) + \alpha_2 R_2(t) + w_2(t) \quad , \quad x_2(0) = R_2(0)$$
$$\dot{x}_3(t) = -\alpha_3 x_3(t) + \alpha_3 R_3(t) + w_3(t) \quad , \quad x_3(0) = R_3(0)$$
$$\dot{x}_4(t) = -\alpha_4 x_4(t) + \alpha_4 R_4(t) + w_4(t) \quad , \quad x_4(0) = R_4(0)$$

where x_i, $1 \le i \le 4$, is the sensor output seen by the navigation filter during measurement incorporation, α_i, $1 \le i \le 4$, is the tracking loop time constant, and w_i, $1 \le i \le 4$, is the tracking loop atmospheric and/or receiver front-end noise.

For the heading angle sensor the model is likewise given by:

$$\dot{x}_5(t) = -\alpha_5 x_5(t) + \alpha_5 \dot{x}_3(t) + w_5(t) \quad , \quad x_5(0) = x_3(0)$$

To form a complete model, for both the simulation environment and the navigation filter, the above state equations must in some fashion be combined with the previously derived ship state equations. To do so requires that the range nonlinearities in the above sensor dynamics be linearized about the nominal ship state solution.

Nominal ship solution

Periodically in the simulation the nominal ship solution is updated and the incremental ship state reset to zero.

$$\underline{x}_{\text{total}} = \underline{x}_{\text{nominal}} + \Delta\underline{x}^{\text{ship}}$$

where Δx^{ship} is the incremental solution discussed in **Section 5.2**. Corresponding to this division, the sensor range forcing terms are also separated into two terms. The incremental term provides the desired connection between the ship dynamics and sensor dynamics. In continuous time, the combined sensor and incremental ship state equations are given by:

Sensor and Ship Continuous-time model

$$\begin{bmatrix} \Delta \underline{x}^{ship} \\ \underline{x}^{sensor} \end{bmatrix} = \begin{bmatrix} A_{ship} & 0 \\ A_{ship,sensor} & A_{sensor} \end{bmatrix} \begin{bmatrix} \Delta \underline{x}^{ship} \\ \underline{x}^{sensor} \end{bmatrix} + \begin{bmatrix} B_{ship} & 0 \\ 0 & B_{sensor} \end{bmatrix} \begin{bmatrix} \underline{u}^{ship} \\ \underline{u}^{sensor} \end{bmatrix} + \text{noise}$$

where

$$\underline{u}^{ship} = [\,\delta \quad V\,]^T$$

$$\underline{u}^{sensor} = [\,R_1\, R_2\, R_3\, R_4\, \theta\,]^T \Big|_{nominal}$$

The nominal ship state dynamics are treated as an unforced system with nonzero initial conditions.

Discretization

From the continuous-time system a discrete-time linear system is obtained by discretization, with a simulation step size Δt. In the discrete time system, the incremental ship state vector at time $t = k\,\Delta t$ is given by

$$\Delta \underline{x}(k+1) = \Phi\, \Delta \underline{x}(k) + \underline{u}(k)$$

where the state transition matrix Φ is given by

$$\Phi = e^{A\,\Delta t}$$

and

$$\Gamma = \int_0^{\Delta t} e^{A\xi}\, B \;\; d\xi$$

where A and B are as given in Section 5.2. In discrete time the state equations are given by:

Sensor and Ship Discrete-time model

$$\begin{bmatrix} \Delta \underline{x}^{ship}(k+1) \\ \underline{x}^{sensor}(k+1) \end{bmatrix} = \begin{bmatrix} \Phi_{ship} & 0 \\ \Phi_{ship,sensor} & \Phi_{sensor} \end{bmatrix} \begin{bmatrix} \Delta \underline{x}^{ship}(k) \\ \underline{x}^{sensor}(k) \end{bmatrix} + \begin{bmatrix} \Gamma_{ship} & 0 \\ 0 & \Gamma_{sensor} \end{bmatrix} \begin{bmatrix} \underline{u}^{ship} \\ \underline{u}^{sensor} \end{bmatrix} + \text{noise}$$

Figure 5-5 illustrates how in the navigation filter portion of the simulator the above state equations are periodically updated. Updating takes place whenever a significant change in the ship dynamics matrix is detected, as sensed by changes in ship heading angle, or when a sensor

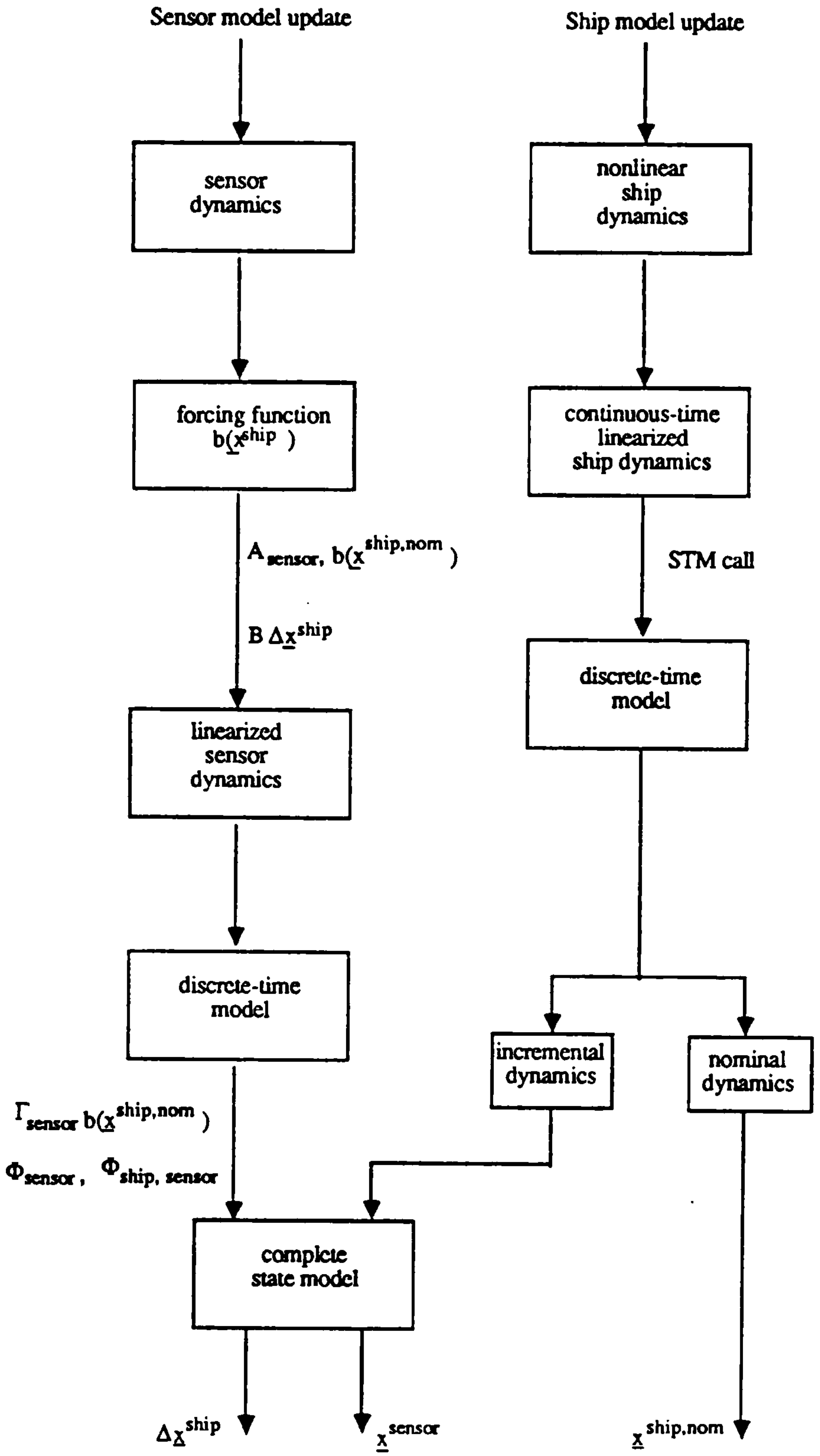

Sensor model update
Ship model update
sensor dynamics
nonlinear ship dynamics
forcing function $b(\underline{x}^{ship})$
continuous-time linearized ship dynamics
A_{sensor}, $b(\underline{x}^{ship,nom})$
$B\,\Delta\underline{x}^{ship}$
STM call
linearized sensor dynamics
discrete-time model
discrete-time model
incremental dynamics
nominal dynamics
$\Gamma_{sensor}\, b(\underline{x}^{ship,nom})$
Φ_{sensor}, $\Phi_{ship,\,sensor}$
complete state model
$\Delta\underline{x}^{ship}$
$\underline{x}^{sensor}$
$\underline{x}^{ship,nom}$

Figure 5-5 Sensor and Ship Model Update

geometry non-linearity threshold is exceeded. In the updating step the nominal ship solution is set to the present estimate of total ship state, and the ship incremental state is reset to zero. After ship dynamics matrix updating, the routine STM is called to convert to a discrete time model. The truth environment is updated in a similar way.

5.4 Navigation Filter Design

The time-discrete ship and sensor model derived above provides the foundation for the navigation filter and rudder control design to follow. Noteworthy features of the ship navigation filter developed for this project include: a) sensor signal tracking loop modeling with an augmented state vector, as described above, and b) an option for navigation filter aiding with rudder position feedback.

Mirroring the truth environment model, the navigation filter has knowledge of fourteen states. Five ship states are included: two in the body-fixed frame, sway velocity and yaw rate, and three in the local waypoint coordinate system frame, alongtrack position, crosstrack error and heading error. Two water current velocity states, easting and northing, are also estimated. These seven states portray the ship motion more accurately than would a simpler easting-northing or ECEF based filter. However, as discussed previously, such a model entails non-linearities in the system dynamics matrix, requiring a periodic linearization about the estimated ship trajectory.

The seven remaining states are associated with the sensor system. As discussed above, division of the ship state vector into nominal and incremental terms permits the sensor and ship states to be combined in a linear way. Two of the sensor states model the navigation receiver clock bias and bias rate, as required in DGPS and differential LORAN-C versions of the navigation filter. The remaining five states portray sensor smoothing lags, as encountered in the range, pseudo-range, RACON bearing angle and heading sensor hardware of the navigation system.

Editing the filter segment of the simulation control file, the user is able to configure three filter variants:

- ranging (up to 4 sites) plus heading sensor,
- pseudo-ranging (up to 4 sites) plus heading sensor, and

- range and bearing (2 sites) plus heading sensor.

The simulation control file permits mismatching between the ship and sensor environment parameters, and the parameters used by the navigation filter.

5.5 Optimal Stochastic Control Design

In the simulation, navigation filter estimates of all ship states are passed on to the rudder controller, whose job it is to control the ship footprint relative to the specified channel boundaries. As discussed previously, such a control seeks to minimize the CCW, defined as the minimum channel width that will clear the ship footprint. To be viable, such a control solution must tolerate random disturbances of current, wind and model mismatch, making best use of the complete navigation filter state vector. This is in contrast to deterministic approaches that have been previously investigated [B-4]. The minimum CCW solution for control involves minimization of a performance index involving both the ship cross-track error and heading error. In particular, the control design problem can be cast as a stochastic regulator problem, whose goal is to drive these waypoint referenced states to zero. This problem formulation is similar to that studied for minimum energy loss steering in a seaway [B-20]. However, in the present setting the controller must regulate both crosstrack and heading errors, and also provide steering commands for transfer to new course legs.

To find the ship controller, the following performance index is used:

$$\text{Performance index } J_i = \frac{1}{2} \sum_{k=i}^{N-1} \{ q_{33}\, \Delta x_3^2(k) + q_{44}\, \Delta x_4^2(k) + r_{11}\, \delta^2(k) \}$$

$$= \frac{1}{2} \sum_{k=i}^{N-1} \{ \Delta \underline{x}^T(k)\, Q\, \Delta \underline{x}(k) + \underline{u}^T(k)\, R\, \underline{u}(k) \}$$

where

$$\Delta \underline{x} = [\, \Delta x_1 \ \Delta x_2 \ \Delta x_3 \ \Delta x_4 \ \Delta x_5 \ \Delta x_6 \ \Delta x_7 \,]^T \quad \text{, where T = transpose}$$

$$\underline{u} = [\, \delta \ \ V \,]^T$$

$$Q = \begin{bmatrix} q_{11} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & q_{22} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & q_{33} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & q_{44} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & q_{55} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad , \quad R = \begin{bmatrix} r_{11} & 0 \\ 0 & r_{22} \end{bmatrix}$$

The weighting matrix Q consists of diagonal elements q_{33} and q_{44}, and other arbitrarily small positive elements along the diagonal to make Q positive semidefinite. This choice of Q implies emphasis on heading angle error and crosstrack error, possibly with some weighting of sway velocity and yaw rate. The matrix R for the control input is positive definite with r_{11} for the rudder control and r_{22} for the velocity control. But the velocity is assumed to be constant, and r_{22} may be set to a very small positive number to maintain the positive-definiteness. Using the above Q and R matrices the optimal control u is obtained by solving the Riccati equation, **Appendix B**.

5.6 Simulator Operation Overview

Prior to closed loop operation with the sensor system and navigation filter, ship hydrodynamic coefficients were calibrated against published models. Examined were response time delay, overshoot, damping, and so on. The general vessel parameters considered were as follows [B-3]:

length of ship	305 meters
width of ship	38 meters
hull type	tanker, 80,000 tons
speed of ship	5.14 meters/second (10 knots)

Beginning with the hydrodynamic coefficients of [B-2], values were trimmed to give dynamics similar to report [B-3]. Final coefficients are as listed in **Table 5-1**.

Table 5-1 Hydrodynamic coefficients

coefficients of the current model	coefficients taken from [B-2]
$C_{v\omega} = 0.518$	$C_{v\omega} = 0.518$
$C_{vv'} = 0.863$	$C_{vv'} = 0.863$
$C_{v\delta} = 0.175$	$C_{v\delta} = 0.175$
$C_{\omega\omega} = 5.32$	$C_{\omega\omega} = 2.45$
$C_{\omega v'} = 5.25$	$C_{\omega v'} = 5.25$
$C_{\omega\delta} = 1.38$	$C_{\omega\delta} = 1.38$

A typical indicator of ship performance is the 20/20 Z maneuver test [B-3]. The test can be described as follows: From a straight line path with rudder amidships, the helm (rudder command) is deflected 20 degrees to the right. When the heading changes 20 degrees to the right of the initial heading, the helm is reversed to 20 degrees left. When the heading changes to 20 degrees left of the initial heading, the helm is reversed to 20 degrees right. The sequence may be repeated many times. For typical 20/20 Z maneuver data recorded on the simulator model, see **Figure 5-6**. From the figure, we can define variables for turn response, heading response, crosstrack response, and so on. For the exact definition of the variables, refer to [B-3]. The Z test itself reveals many inherent characteristics of the ship, such as delay between the rudder input and actual response, heading excursion, overall damping, and so on. Following a preliminary effort at adjustment of the $C_{\omega\omega}$ parameter, the values of **Table 5-2** were obtained. Further work on sway velocity dynamics is warranted.

Table 5-2 20/20 Z Maneuver

	final model	USCG model [B-3]
Turn Response Variables		
Rise time: T_{20}	137 seconds	92 seconds
Slew rate: ψ_{20}	0.21 deg/sec	0.3 deg/sec
Heading lag: $T_{\psi lag}$	41 seconds	38 seconds
Displacement lag: T_{Dlag}	263 seconds	168 seconds
Heading Response Variables		
Max heading excursion: ψ_{max}	23 degrees	25.8 degrees
% heading overshoot: % ψ_{os}	15 %	29 %
Track Response Variables		
Max crosstrack displacement: D_{max}	385 meters	260 meters

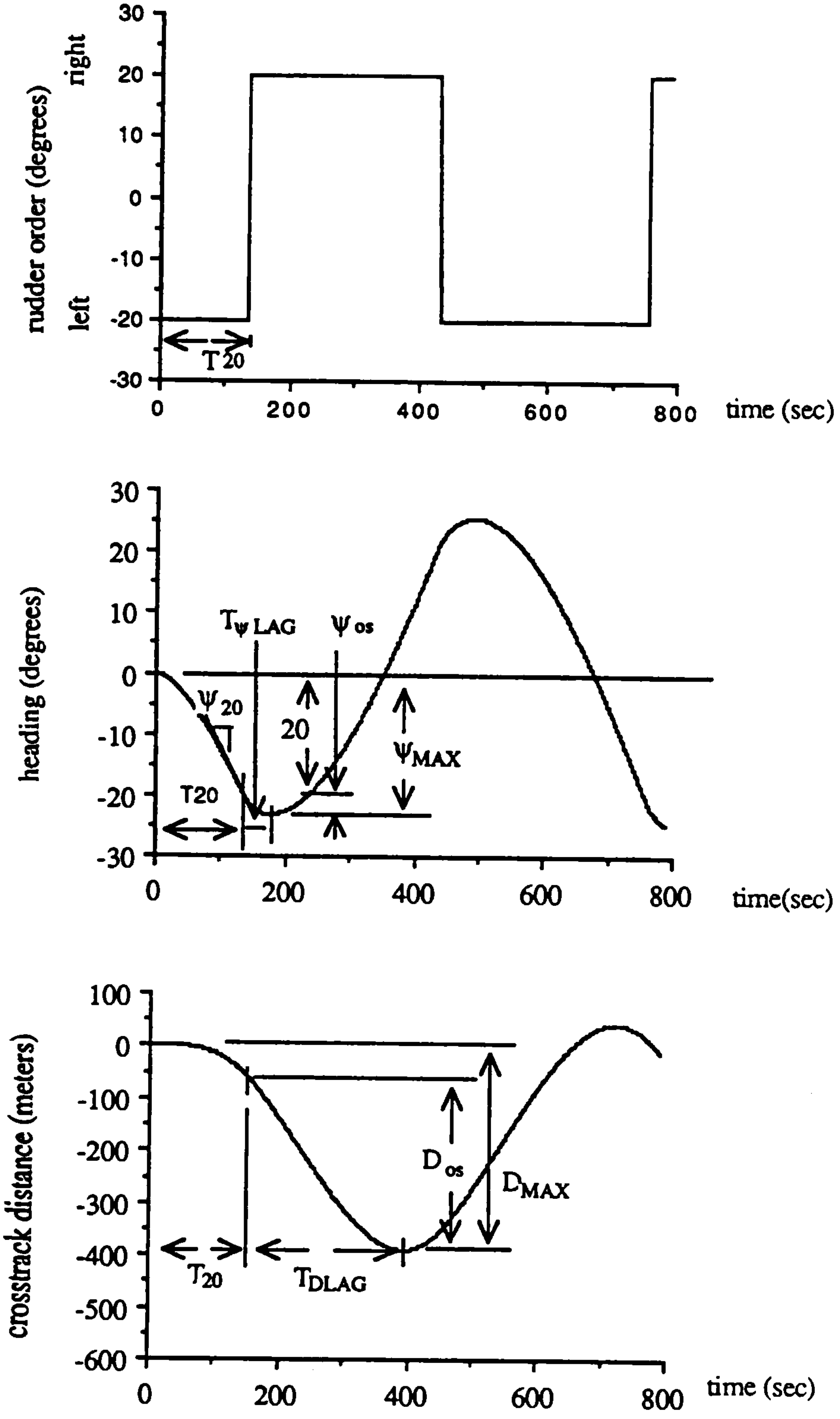


Figure 5-6 20/20 Z Maneuver test

In the closed loop simulations to follow, disturbances are introduced on sway and yaw rate dynamics, on water current states sources, and on sensor channel outputs. Typically, simulations are run in two phases, with perfect state knowledge, and with navigation filter state estimates. For each sensor combination, data is gathered on ship control performance as exemplified by such measures as ship cross-track and heading error, relative to the currently defined course leg, and by channel clearance width as a function of accumulated along-track position. Various measures of navigation sensor performance are also gathered. Sensor faults may be introduced at any point.

6. Performance of Representative Candidate Systems

6.1 Test Scenarios Overview

From the survey of sensors discussed in **Sections 2.1 and 2.2**, the candidates chosen for evaluation with the simulator were DGPS, LORAN-C, RACON, and microwave. As shown in **Figure 6-1**, the basic test scenario consisted of a strait approach segment of 1500 meters, a 35 degree course change to port, and another 2500 meters of course keeping. Ship velocity was 10 knots (5.2 meters per second). A diminishing southwesterly current of 1.5 knots was applied early in each run.

No attempt was made to replicate geometry specific to a particular waterway. Rather, for the microwave, DGPS, and LORAN-C systems emitter geometry was established in a generic way, with all sensors sites lying at fixed radial distances from the test area. Microwave transponders and LORAN-C transmitters were deployed at radial distances of 20 km and 1000 km respectively, while GPS sub-satellite points were located 10000 km from the test site. RACON sites were located at the extension of each course leg, with RACON1 placed so as to support good two-unit geometry in the turn region. Note that for GPS and LORAN-C the minimal three emitter geometry is assumed, while the microwave ranging and RACON systems operate with over-determined geometry.

Table 6-1 summarizes those parameters common to all the test scenarios. To simulate the effect of wind and unmodeled hydrodynamical forces on the ship, random disturbances were introduced on the ship sway and yaw rate axes. The higher of the two disturbance level scenarios was selected to model storm conditions likely to be encountered only infrequently in river and harbor areas. The low-disturbance scenario is more typical of every-day pilotage conditions.

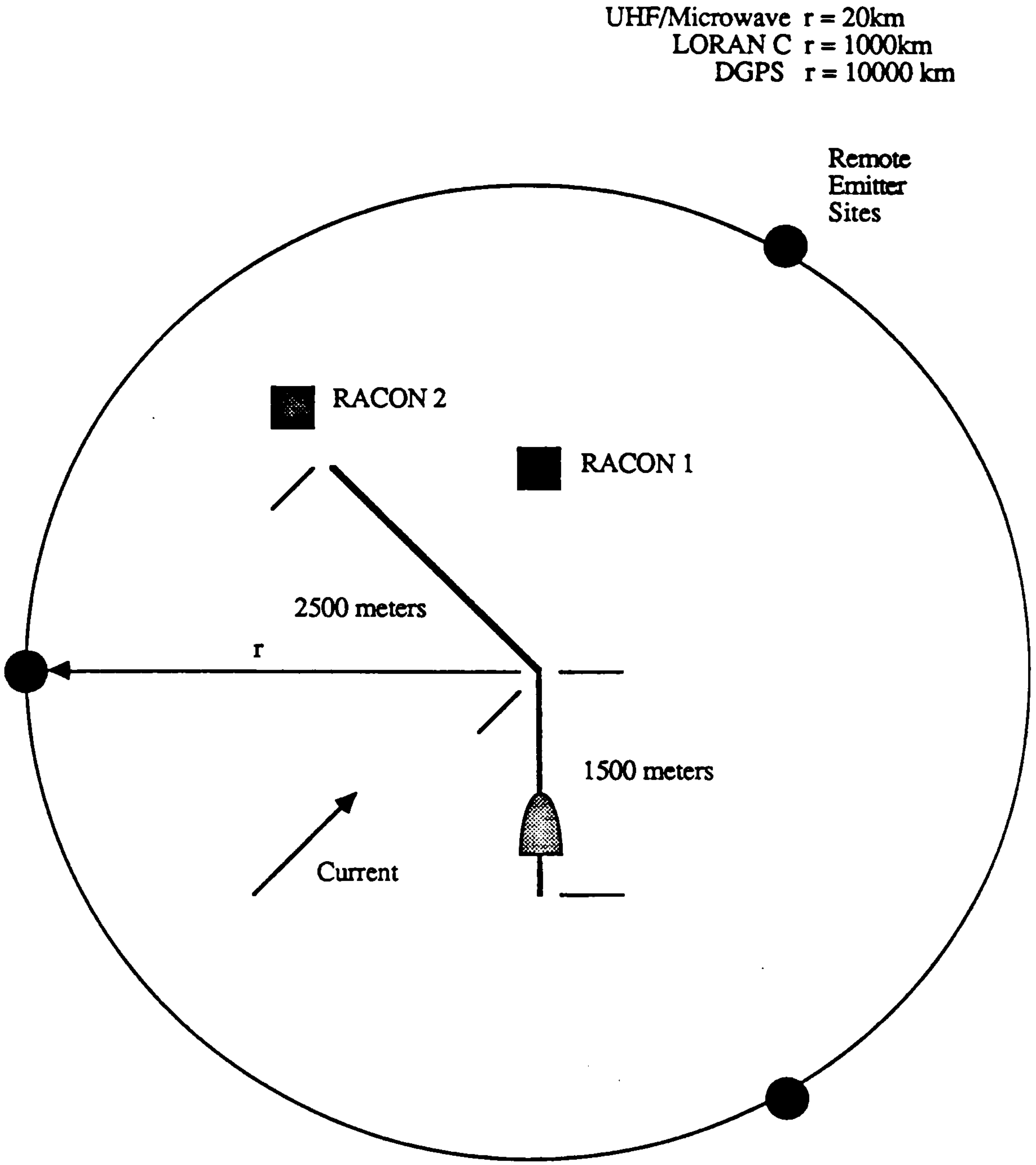


Figure 6-1 Site Geometry

Table 6-1
Common Scenario Ship and Sensor Parameters

Ship Parameters

sway axis velocity disturbance, high setting .32 meters/second, 1 sigma
sway axis velocity disturbance, low setting .097 meters/second, 1 sigma

yaw axis rate disturbance, high setting .11 degrees/second, 1 sigma
yaw axis rate disturbance, low setting .035 degrees/second, 1 sigma

longitudinal ship velocity 5.14 meters/seconds (10 knots)

easting and northing current components .5 meters/second

Sensor Parameters

heading sensor output standard deviation 0.5 degrees, 1 sigma
heading sensor time constant 1 second
heading sensor navigation filter sample rate 1 Hz

user clock drift rate 1 nanosecond/second, 1 sigma
drift rate process correlation time 100 seconds

Parameters common to all sensor scenarios are listed in **Table 6-1**. The simulated heading sensor dynamics/noise model consists of a first-order lowpass filter of one second time constant forced by the true ship heading. A white noise term is also introduced such that the sensor jitter is 0.5 degrees. The heading sensor output is sampled by the navigation system at a 1 Hz rate.

For LORAN-C and DGPS systems, the simulation environment also provides a two-state clock model. The clock rate state is Gauss Markov, with a time constant of 100 seconds and a clock drift rate standard deviation of one nanosecond per second. This clock model is mirrored in the ship navigation filter.

6.2 Tests With Perfect Navigation State Knowledge

Even with perfect knowledge of ship position, velocity, heading and yaw rate states, some deviation from the desired waypoint trajectory is unavoidable. Therefore, before beginning specific sensor configuration runs, it was important to determine the aggregate contribution, to ship control error, from wind and water disturbances and ship handling. To this end the optimal ship controller was driven with a perfect ship navigation state vector, under a variety of disturbance conditions. This performance baseline is useful in establishing an overall error budget for ship navigation and control systems.

During this preliminary testing phase, the course leg switching point was optimized to obtain best channel clearance width (CCW) performance in the turn region, and baseline values for sway and yaw rate disturbances were established.

Figure 6-2 shows disturbance-free left and right clearance width results obtained after switch-point optimization. The graph ordinate is accumulated along-track position of the ship, and the abscissa is the minimum channel width needed to just clear the extremity of the ship. As expected, during the turn approach and after turn recovery the clearance width values are nearly the same as the ship half width, 19 meters. In the turn region however, the right clearance width must be increased to 150 meters, and the left clearance width must be increased to 80 meters to avoid boundary contact.

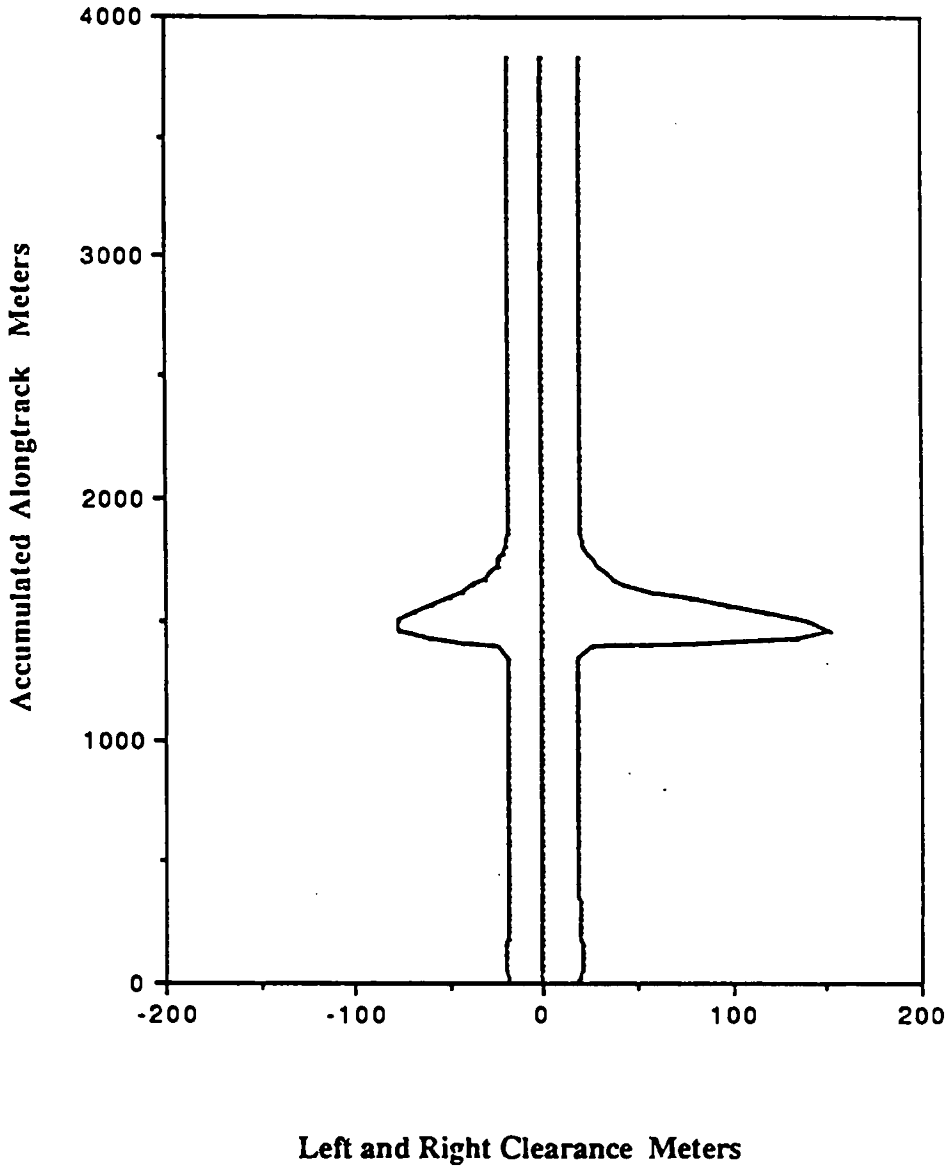


Figure 6-2 CCW, Perfect State Inputs Without Disturbance

Figure 6-3, depicts a typical run with the lower of the two disturbance levels of **Table 6-1**. Even with perfect knowledge of ship state, increased clearance values are observed. **Figure 6-4** is a typical channel clearance plot with the larger ship disturbance level. The clearance values required in the turn region are similar to those in the no-disturbance case, but turn approach and recovery regions show substantial increases in the channel width needed for safe passage. One way of further quantifying this behavior is to tabulate cumulative probability distributions for channel clearance values at sample along-track points, as obtained by passing the maximum of left and right clearance values at each sample point to the cumulative distribution routine. Because the distributions are likely to differ in turn approach and recovery regions, they are computed separately in each region. Using a five second sample interval, each distribution is the result of several runs and thousands of data points. Distributions are tallied for high and low ship disturbance cases.

Figure 6-5 compares the channel clearance width cumulative distributions in the pre-turn region, for both low and high disturbance conditions. For the low disturbance runs 95% of the clearance width samples are under 28 meters. For the high disturbance group the 95% clearance half width is 55 meters. Higher or lower probability thresholds can be estimated from these curves. Not shown is the distribution for zero ship disturbance, which consists of a step jump to probability 1.0 at the 19 meter ship half-width.

Figure 6-6 compares the channel clearance width distributions in the turn recovery region for both disturbance conditions. The 95% values are nearly the same as in the pre-turn region. However, as will be seen in later simulation runs, turn recovery requirements are generally greater when the ship is steered with imperfect sensor data.

6.3 Differential GPS Performance

As discussed in Section 2.2.1, DGPS user and reference station equipment can operate in code mode and integrated Doppler mode. The most accurate results are obtained with both sites in the integrated Doppler mode. However, in this report we shall assume that aboard ship the less accurate equipment is available. Furthermore, delta range data, useful in estimating velocity states, will be ignored. Thus, the DGPS configuration investigated constitutes the bottom rung in the DGPS equipment ladder. In this configuration the sensor

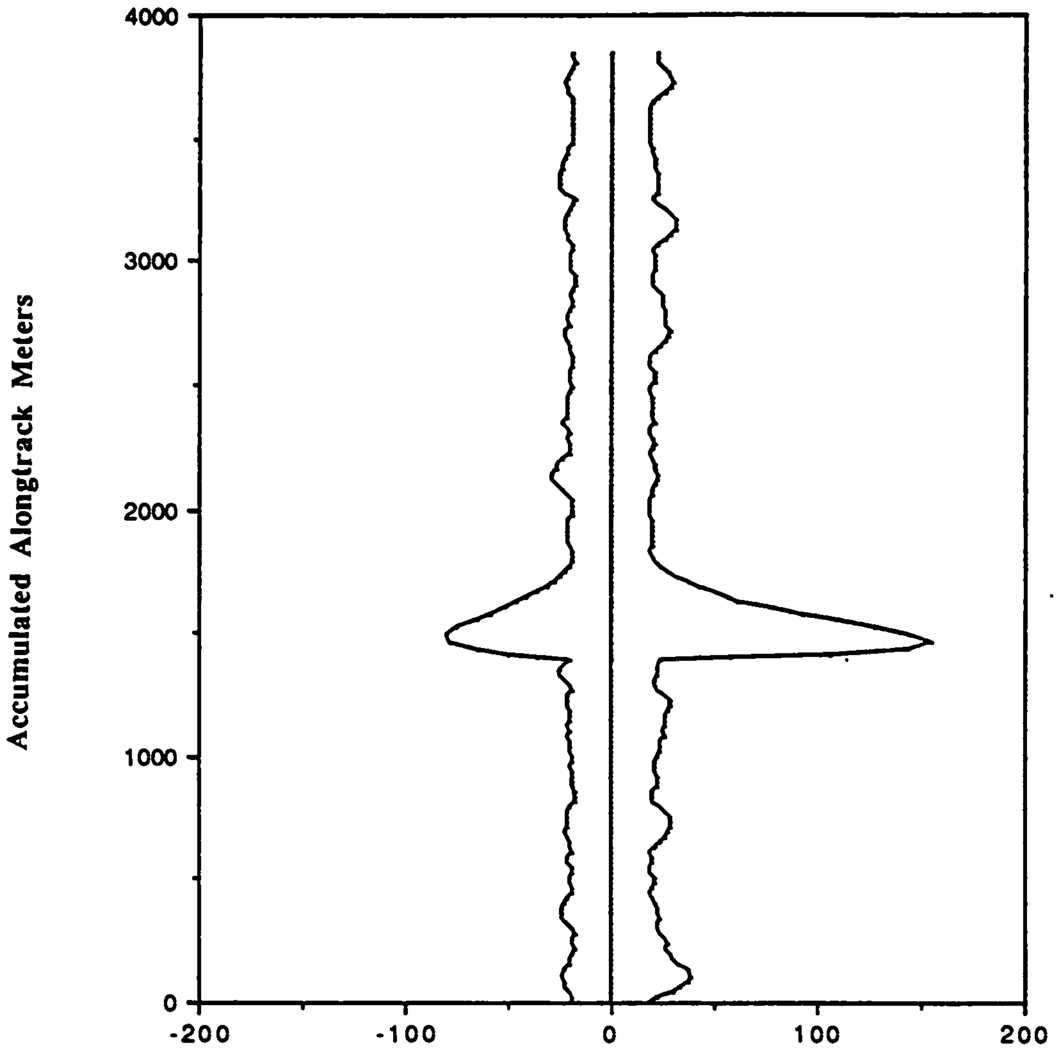


Figure 6-3 CCW, Perfect State Inputs, Low Disturbance

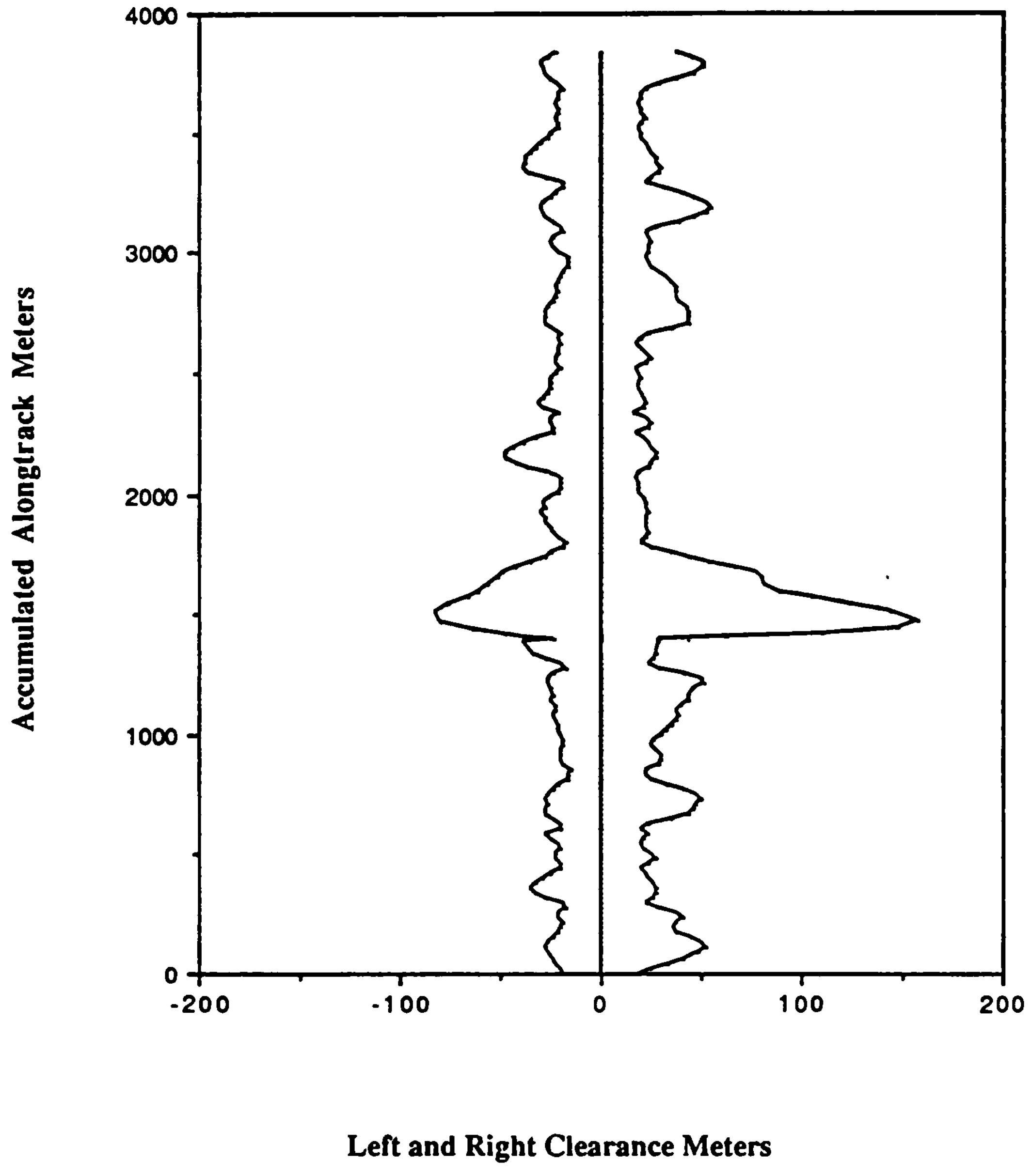


Figure 6-4 CCW, Perfect State Inputs, High Disturbance

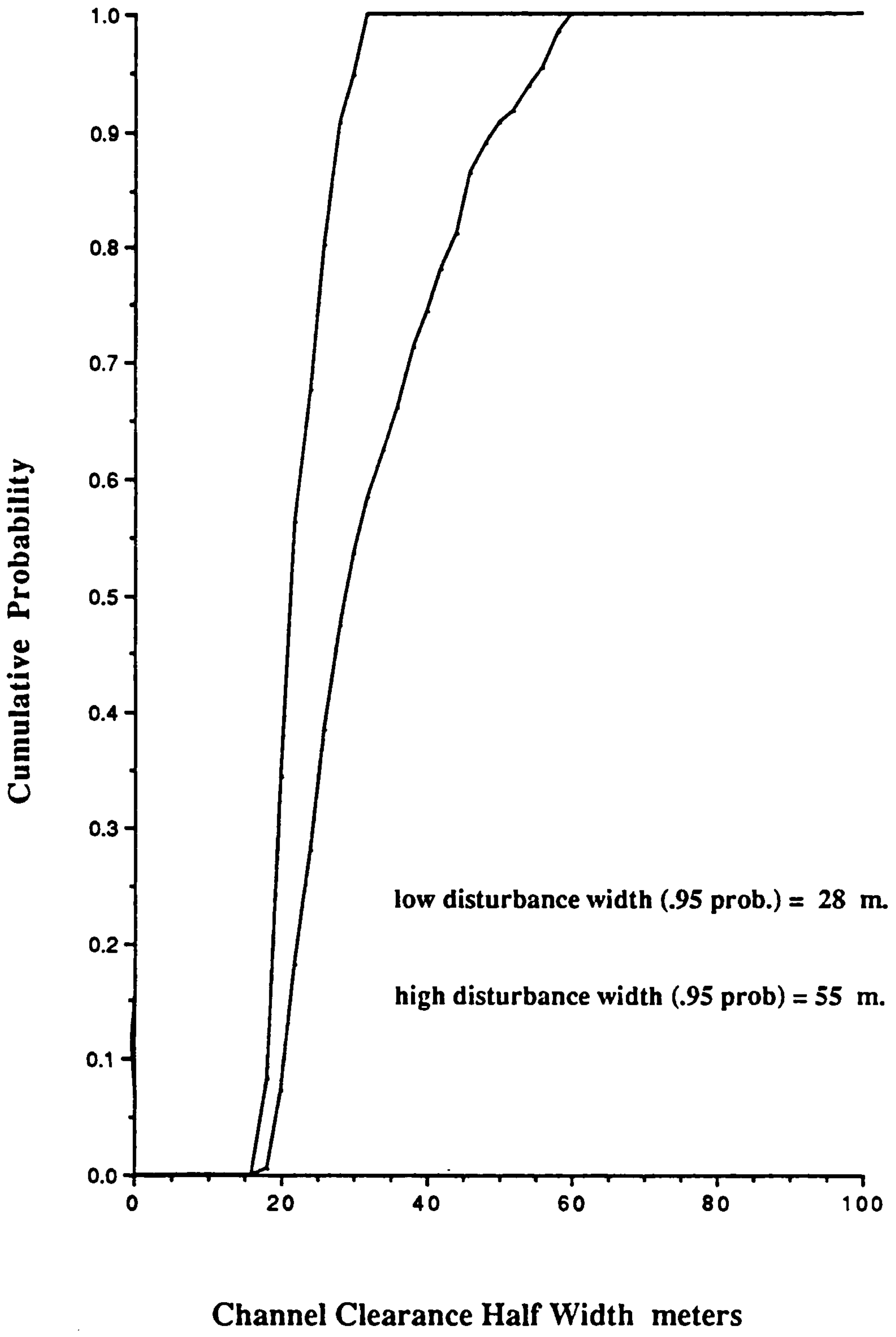
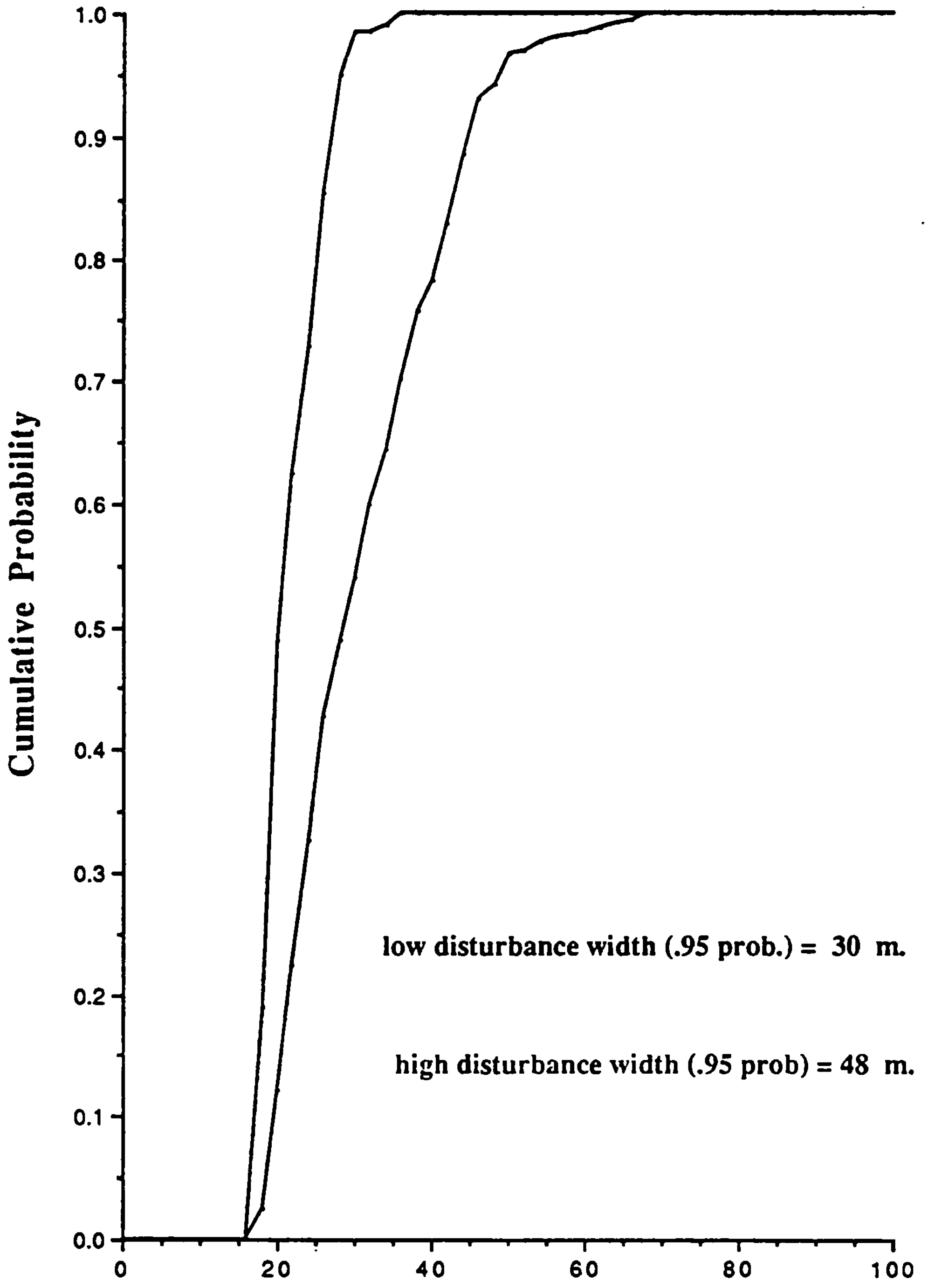


Figure 6-5 Channel Clearance Width Distributions Before Turn, Perfect Nav



Channel Clearance Half Width meters

Figure 6-6 Channel Clearance Width Distributions After Turn, Perfect Nav

error budget is dominated by code tracking loop noise, multipath and tracking loop dynamical errors.

Corresponding to operation with a received carrier-to-noise ratio of 40 dB-Hz, and a static tracking loop jitter of 5 meters, the simulated code tracking loop time constant was set at 0.125 seconds. This loop model is employed as well in the ship navigation filter, where an augmented state vector is employed for compensation purposes.

Satellite visibility in certain areas of the waterway system may restrict the user to fewer than four satellites. For this reason all simulations are carried out with just three satellite channels. The navigation filter is aided with the previously described heading sensor, as well as with rudder position knowledge. All data is sampled at a one Hz rate.

Figures 6-7 and **6-8** are CCW plots of typical runs, under low and high disturbance conditions respectively. These are generally similar to those obtained for operation with perfect ship state knowledge. It is very instructive to compare cumulative distributions for CCW, obtained over several runs, against the perfect state knowledge case. These comparisons in before and after turn regions are as shown in **Figures 6-9** and **6-10**, with perfect-state distributions depicted as lighter plot lines. Particularly in the before turn region, the the DGPS distribution tracks very closely with the perfect-state distribution. At the 95% level an increase of about five meters in CCW is contributed by the DGPS sensor system over the perfect-state case. The turn recovery region distributions, **Figure 6-10**, are less good. Although the performance loss attributable to the sensor system is only about 2 meters for the low disturbance condition, a 16 meter increase in CCW is observed in the large disturbance scenario.

In summary, the DGPS system contributed a modest increase in CCW under most operating conditions. However, dynamical performance could be greatly improved upon in the turn recovery region by employing delta range or integrated Doppler GPS data in the navigation filter.

6.4 Differential LORAN-C Performance

As with the GPS system, LORAN-C may be operated in a differential mode. The contribution of the reference station corrections to the overall error budget will be neglected. This assumption may be more problematical for LORAN-C, since local grid

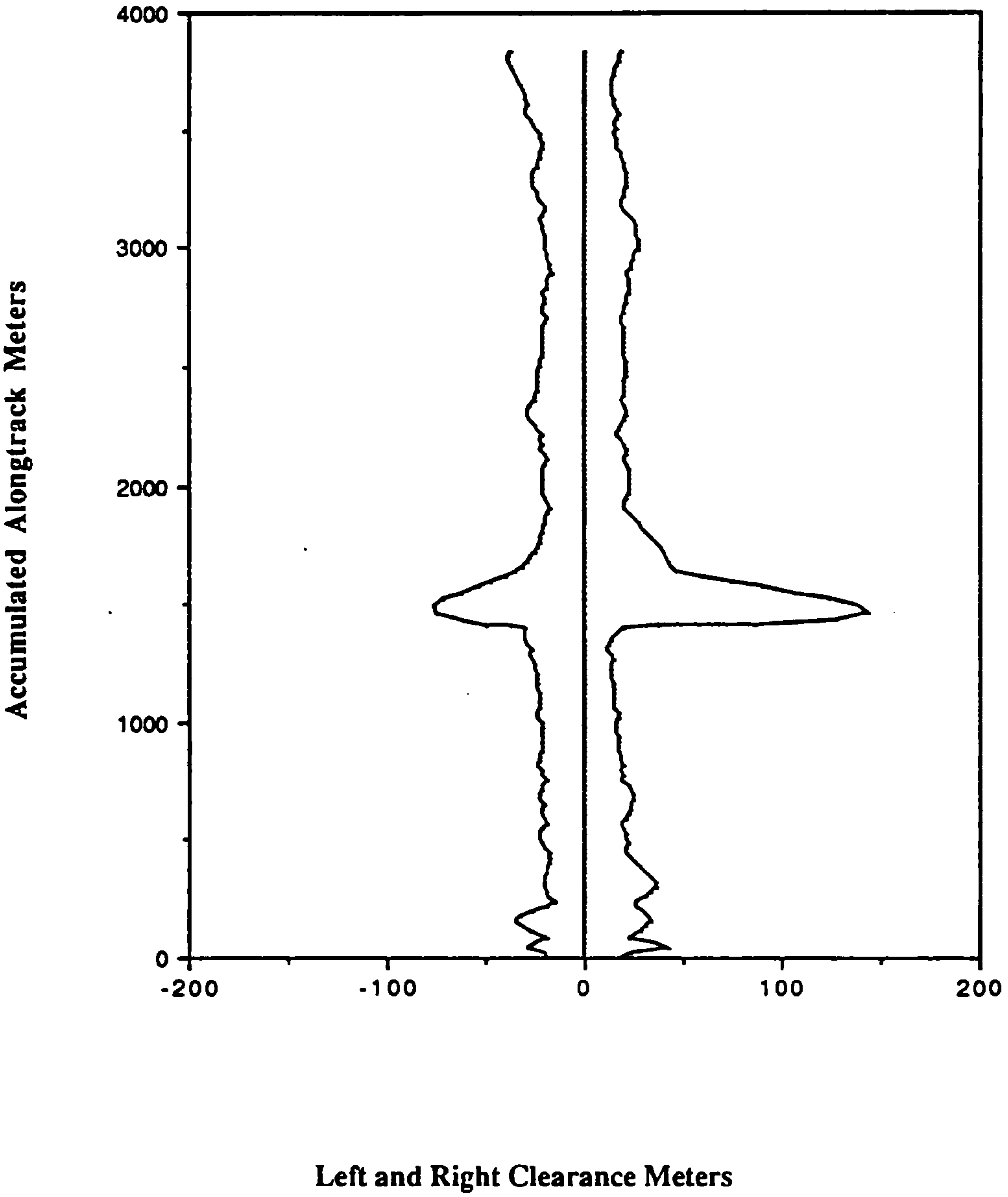
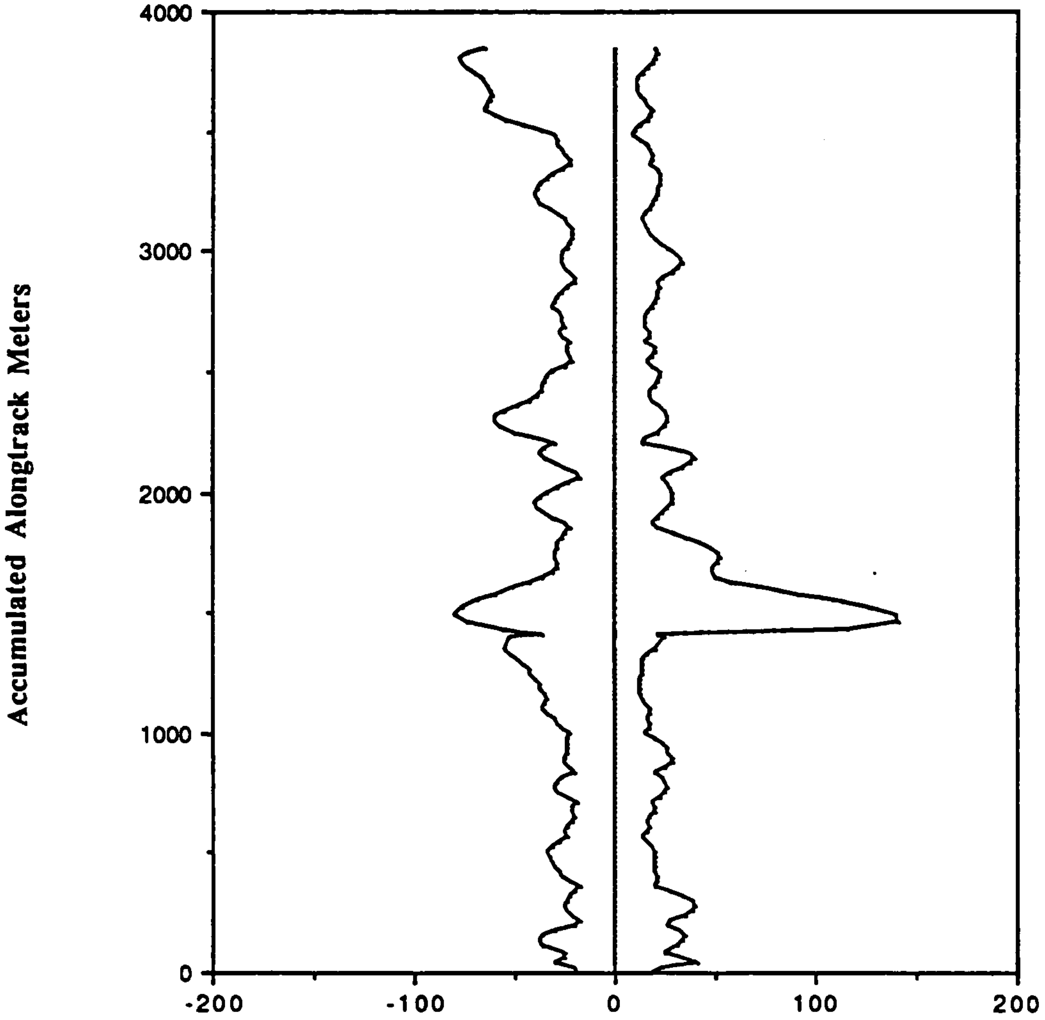
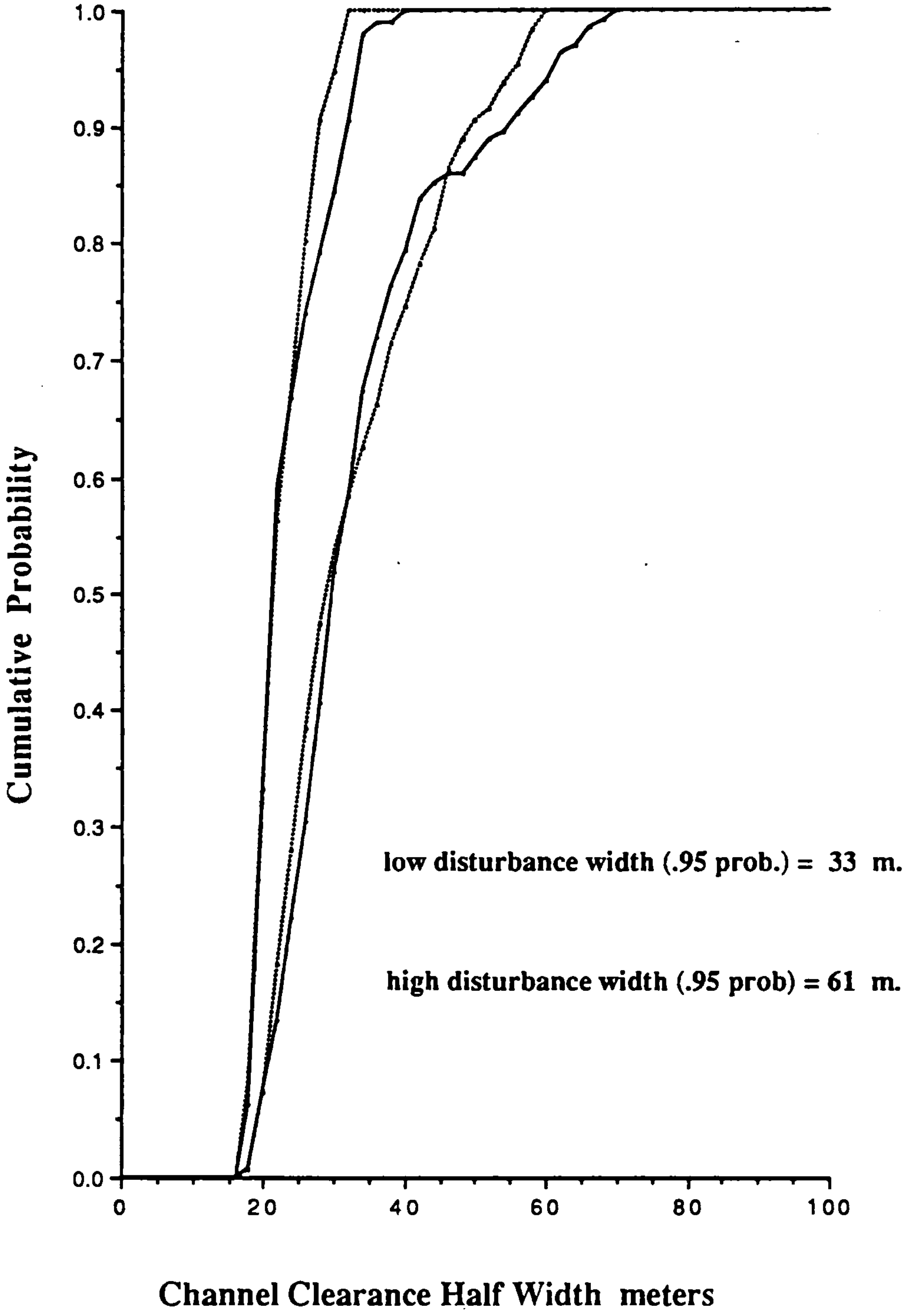


Figure 6-7 CCW, DGPS Sensor, Low Disturbance



Left and Right Clearance Meters

Figure 6-8 CCW, DGPS Sensor, High Disturbance



low disturbance width (.95 prob.) = 33 m.

high disturbance width (.95 prob) = 61 m.

Figure 6-9 Channel Clearance Width Distribution Before Turn, DGPS

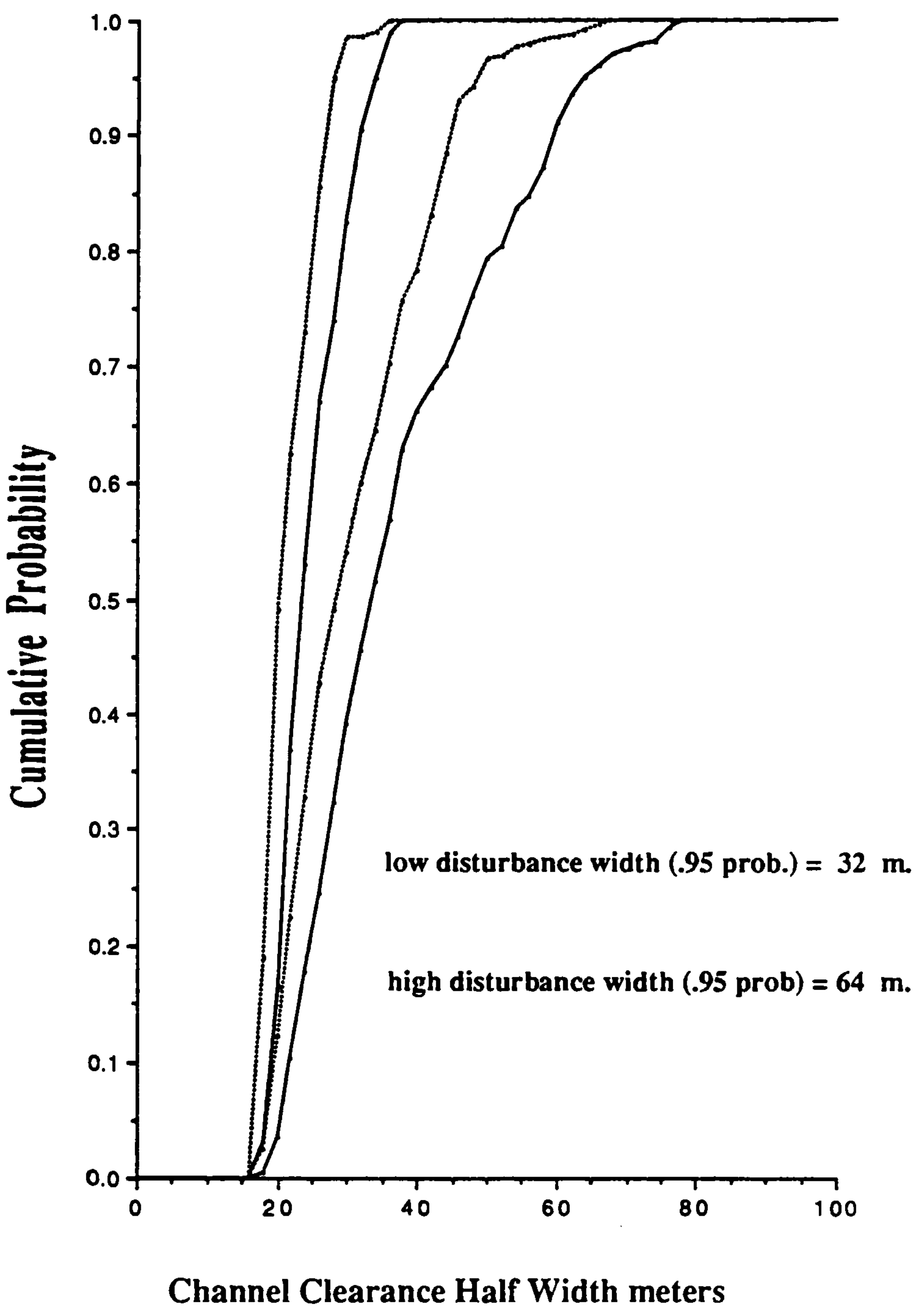


Figure 6-10 Channel Clearance Width Distributions After Turn, DGPS

warp conditions have proven difficult to calibrate in river and harbor areas. The magnitude of these errors is much larger than such GPS site dependent errors as multipath.

Operating in the "rho-rho" mode, the navigation filter of the simulator extracts clock bias and clock rate errors from the simulated LORAN-C sensor data. To facilitate certain comparisons between GPS and LORAN-C, the simulated tacking loop jitter was set to the GPS value of 5 meters. For an assumed 20 kHz SNR of +10 dB this corresponds to a tracking loop time constant of about 14 seconds. As previously discussed in **Section 5**, the fourteen state elements of the filter include LORAN-C signal tracker dynamics and, as for GPS, heading reference data and rudder knowledge is available to the filter at a one Hz sample rate.

Because of the substantial time constant of the simulated LORAN-C tracking loops, special attention was given to the overall stability of the closed loop ship steering control. It was found that some degree of both rudder knowledge and tracking loop dynamics, within the navigation filter, was essential for stabilization of the ship steering control loop. The system as a whole was unstable in the absence of rudder position knowledge. This may be an important consideration in the design of the human pilot interface. Although in practice the human pilot knows the rudder position at all times and may tend to compensate for sensor lags with this knowledge, ship navigation data presented to the pilot could be pre-compensated, as is done in the present simulation.

To add realism to the simulations, in the final LORAN-C runs a mis-match between the environment and navigation filter LORAN-C tracking loop time constants was utilized, 14 seconds in the environment and 3.5 seconds in the navigation filter. This mismatch resulted in a small loss in performance over the matched case. However, in all runs perfect rudder knowledge was employed by the navigation filter. **Figures 6-11** and **6-12** show typical results for CCW, for low and high disturbance conditions respectively. Both plots exhibit a good deal of drift, indicative of poorer velocity estimates within the navigation filter and a consequent loosening of the ship control loop.

Turning to the cumulative distributions for channel clearance width prior to the turn, **Figure 6-13**, a substantial loss in performance over the perfect-nav case is observed. Under the low disturbance condition the 95% half width has increased from 28 to 43 meters, and with high ship disturbances the increase is from 55 to 75 meters. As shown in

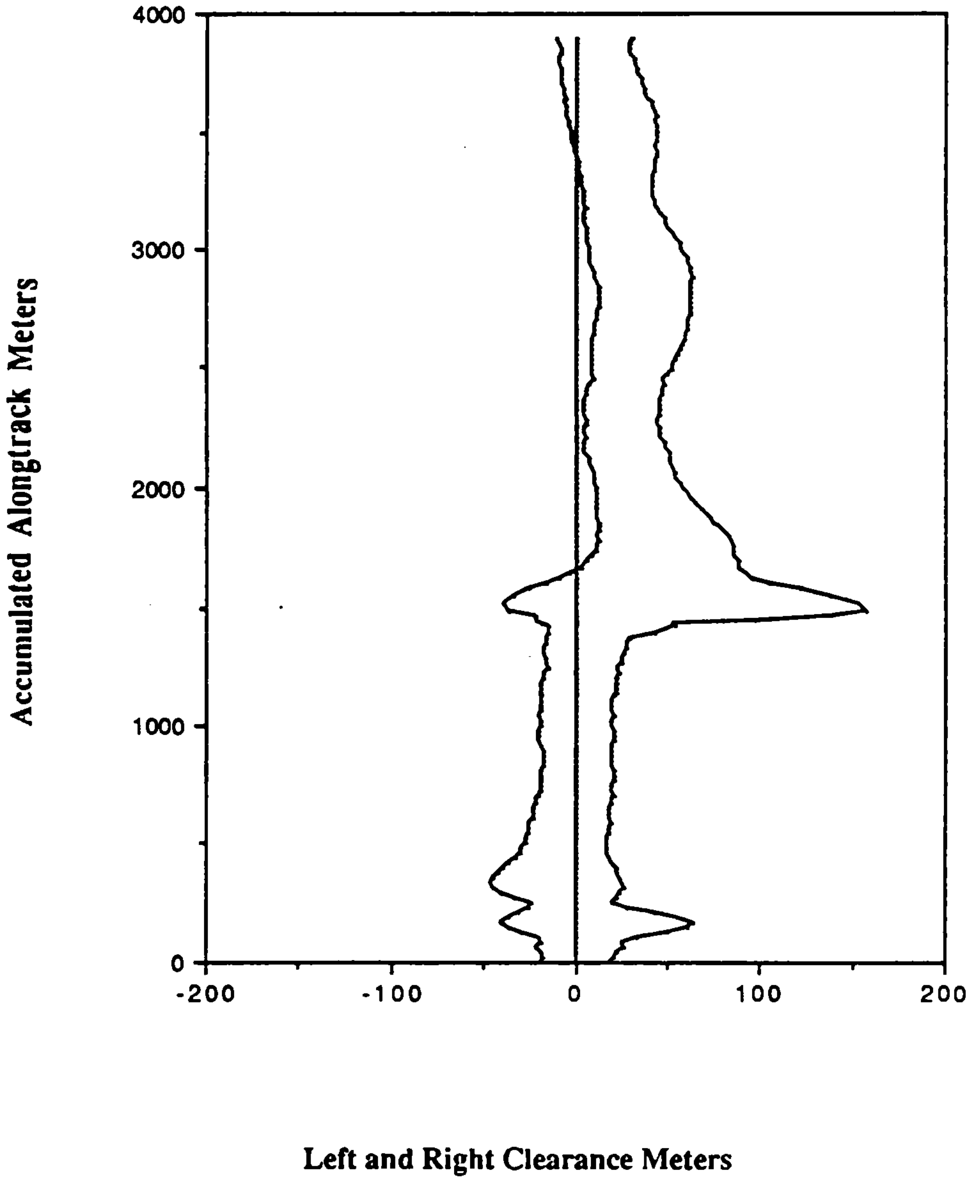


Figure 6-11 CCW, LORAN-C Sensor, Low Disturbance

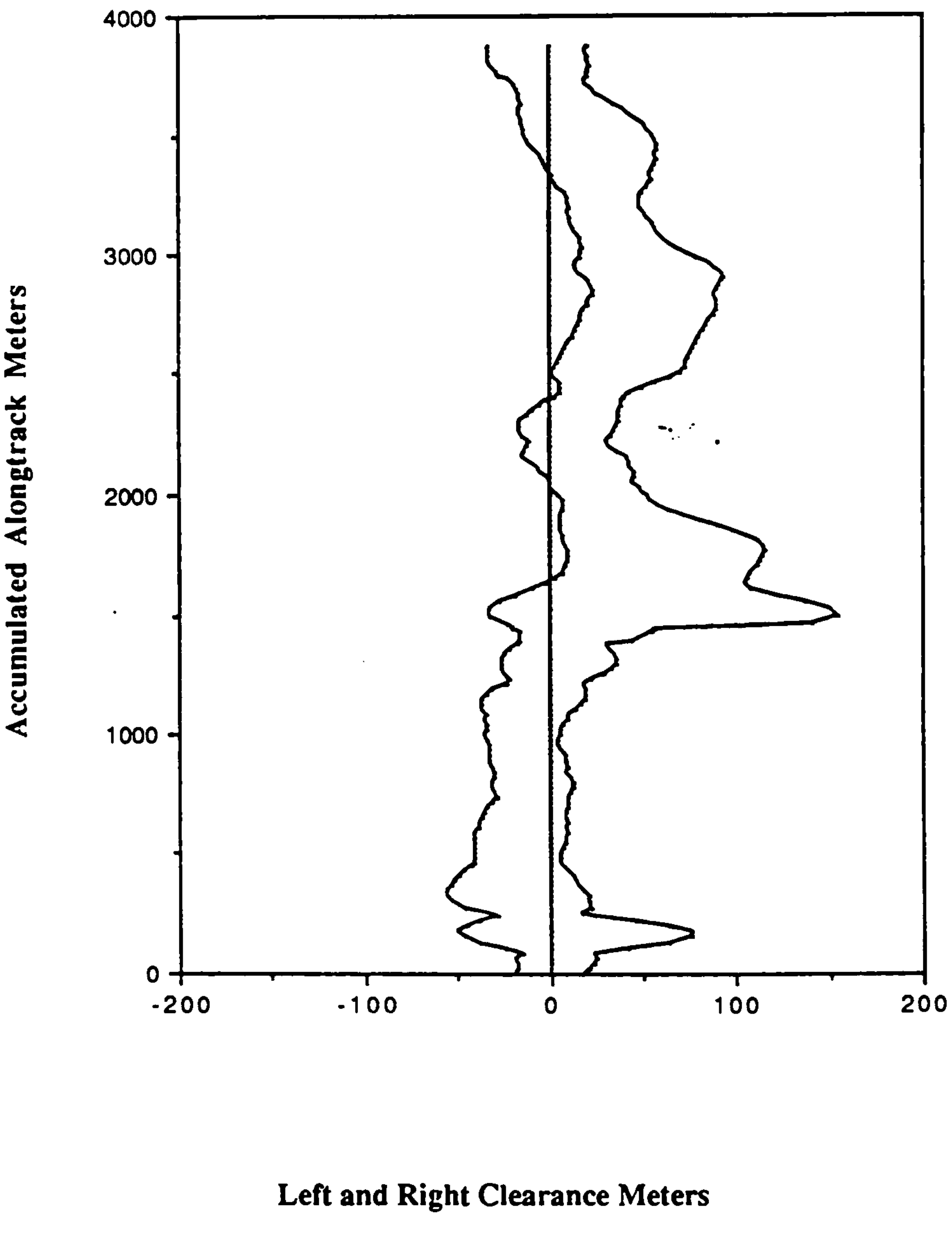


Figure 6-12 CCW, LORAN-C Sensor, High Disturbance

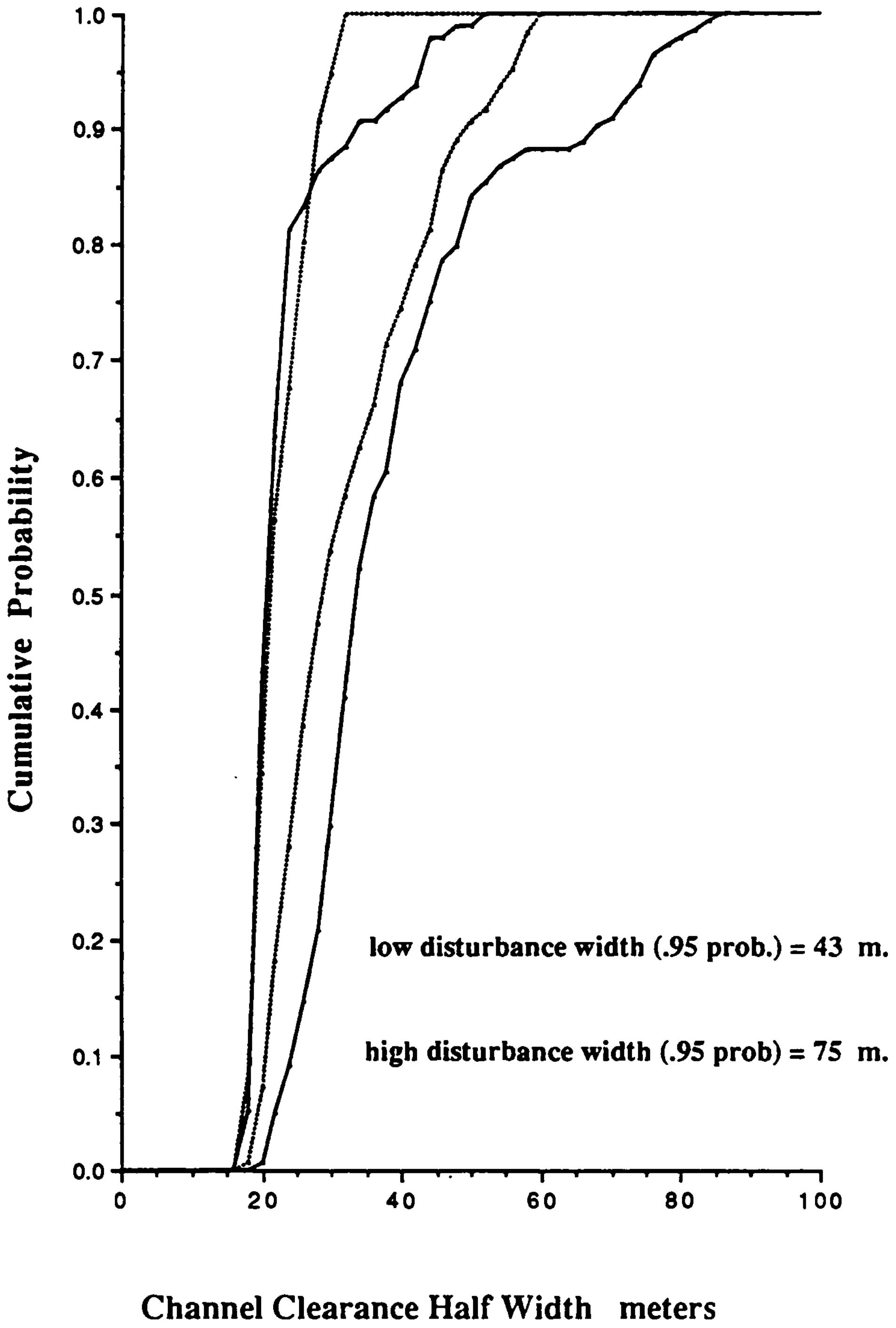


Figure 6-13 Channel Clearance Width Distributions Before Turn, LORAN-C

Figure 6-14 the performance is still poorer in the turn recovery region, where the 95% half width has increased to 85 meters in the high disturbance case.

As these tests clearly suggest, a specification of receiver navigation solution jitter and update rates is insufficient for predicting ultimate closed loop performance. In particular, although sensor geometries, user clock models, tracking loop output standard deviation characteristics and sensor sample rates were identical in the above DGPS and LORAN-C examples, achieved control performances differed greatly. A complete sensor specification must therefore include a tracking loop equivalent model portraying actual raw sensor dynamics.

6.5 RACON System Performance

As discussed in **Section 2.1**, two variants of the RACON system are of interest, one using the standard ship radar and the other requiring a more precise ship interrogator and monopulse receiving antenna. Because of its wider applicability, emphasis was placed on the less accurate of these RACON systems.

The basic RACON system is assumed to provide an azimuth angle accuracy of 1.5 degrees, and a ranging accuracy, including the effects of RACON jitter, of 20 meters. Tracking loop time constants for both azimuth and ranging channels was set to one second. The sample rate at azimuth and range loop outputs was set at 1 Hz. The reader should refer to **Figure 6-1** for the RACON site geometry. Note that the ship navigation filter processes data from both RACON's throughout the passage. Additional inputs are heading sensor and rudder position.

Figures 6-15 and **6-16** show typical CCW results for low and high disturbance conditions, respectively. Generally there was evidence of some low-frequency drift in all the runs, particularly early on where the GDOP was poor, and in the early portion of the turn as well. **Figures 6-17** and **6-18** show the cumulative distributions for all runs, for both disturbance conditions. The loss in performance from the perfect-state case is most significant in the pre-turn distribution, **Figure 6-17**. The distributions in the turn recovery region are much tighter, reflecting improved geometry and the fast dynamics of the RACON system.

In conclusion, the RACON system with standard ship radar, when properly interfaced to the navigation computer, and operating with good geometry, was similar to the least

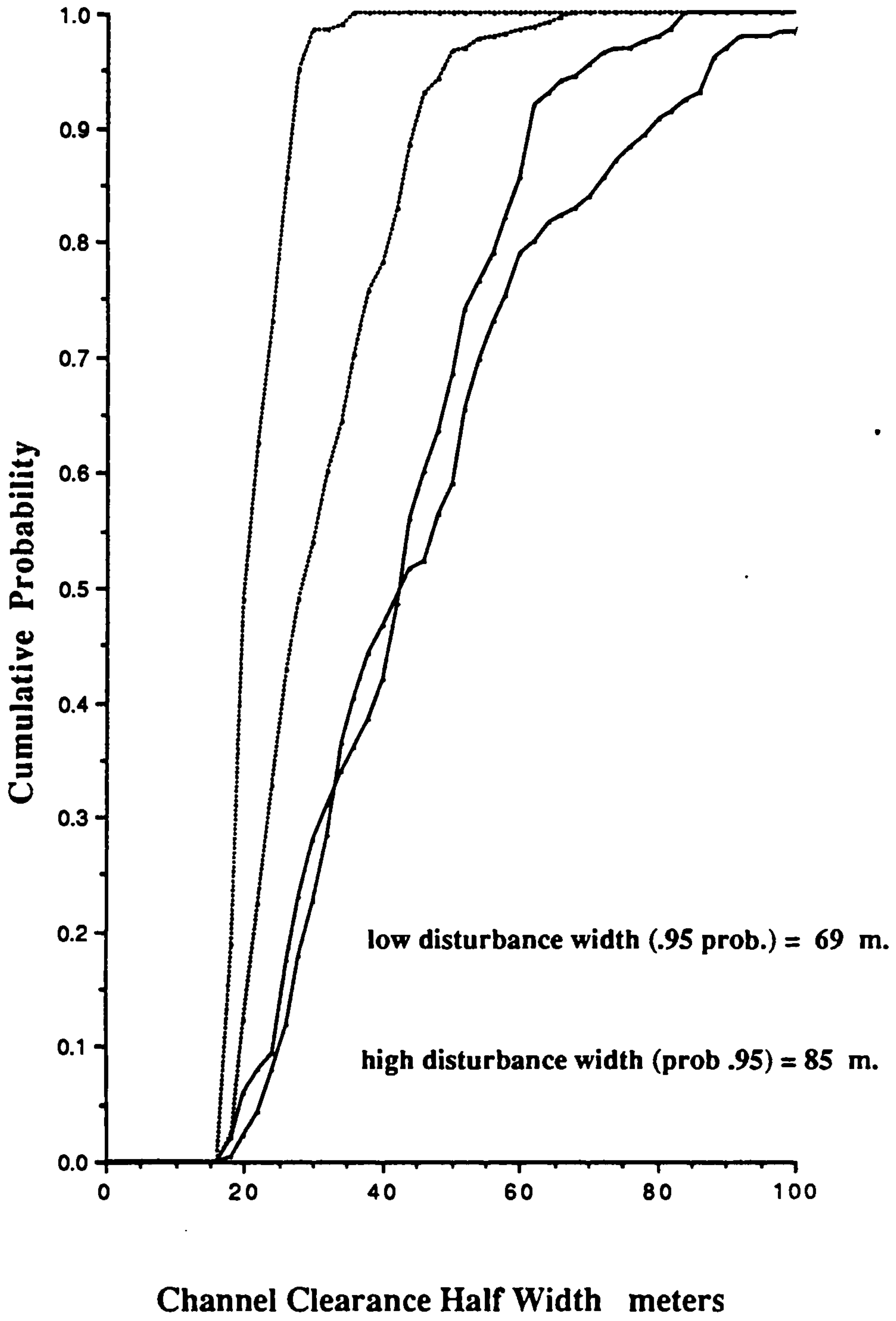


Figure 6-14 Channel Clearance Width Distributions After Turn, LORAN-C

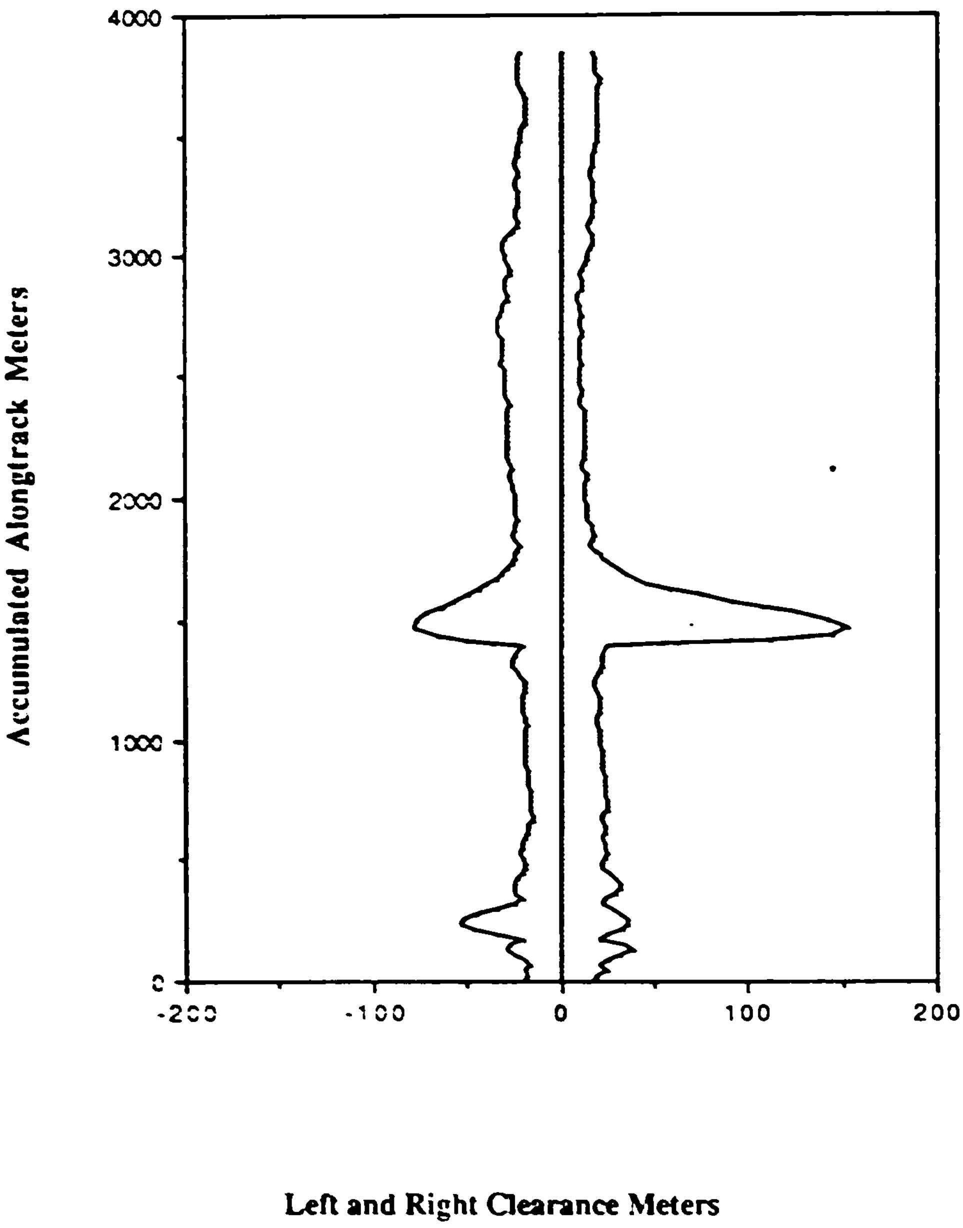


Figure 6-15 CCW, RACON Sensor, Low Disturbance

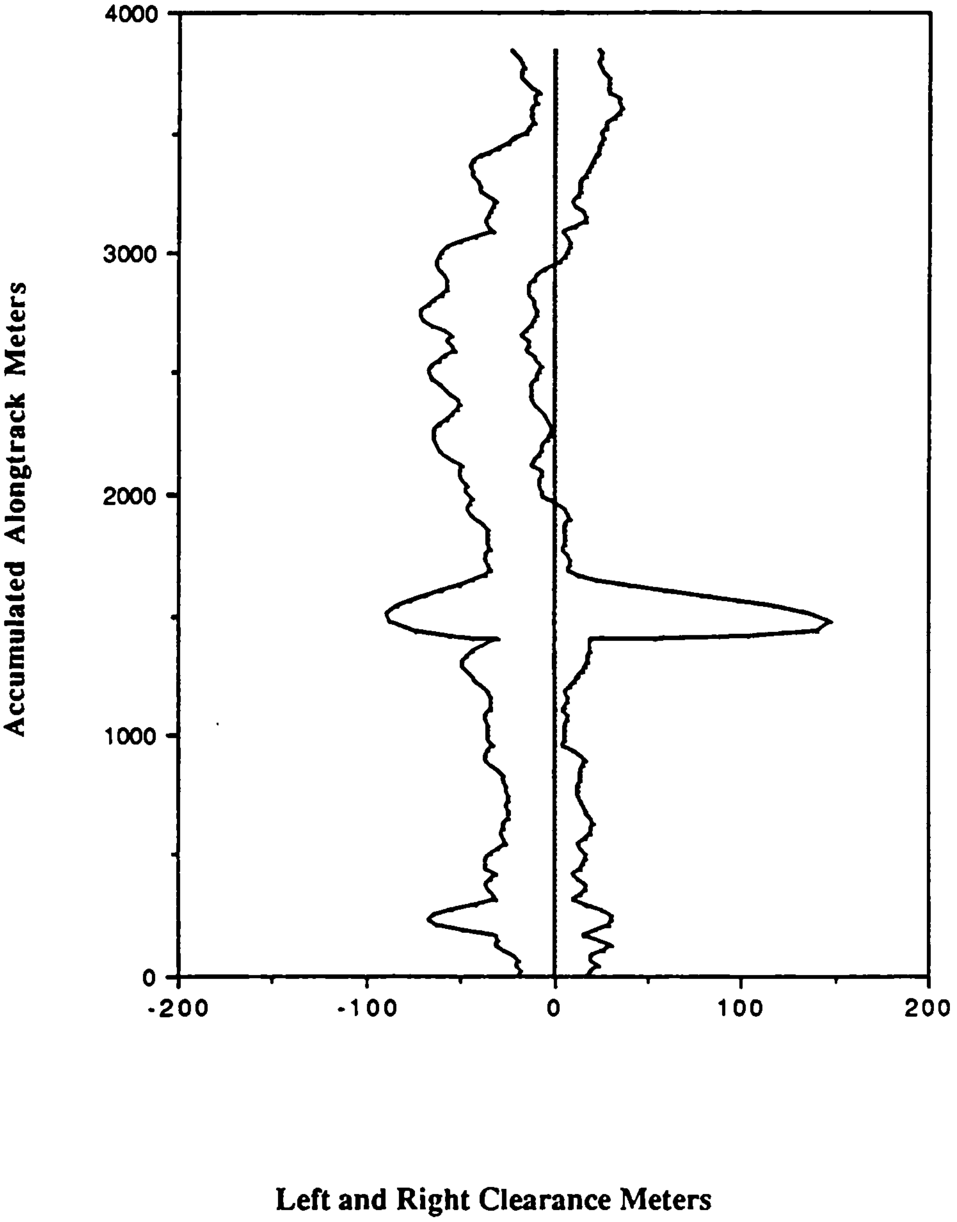


Figure 6-16 CCW, RACON Sensor, High Disturbance

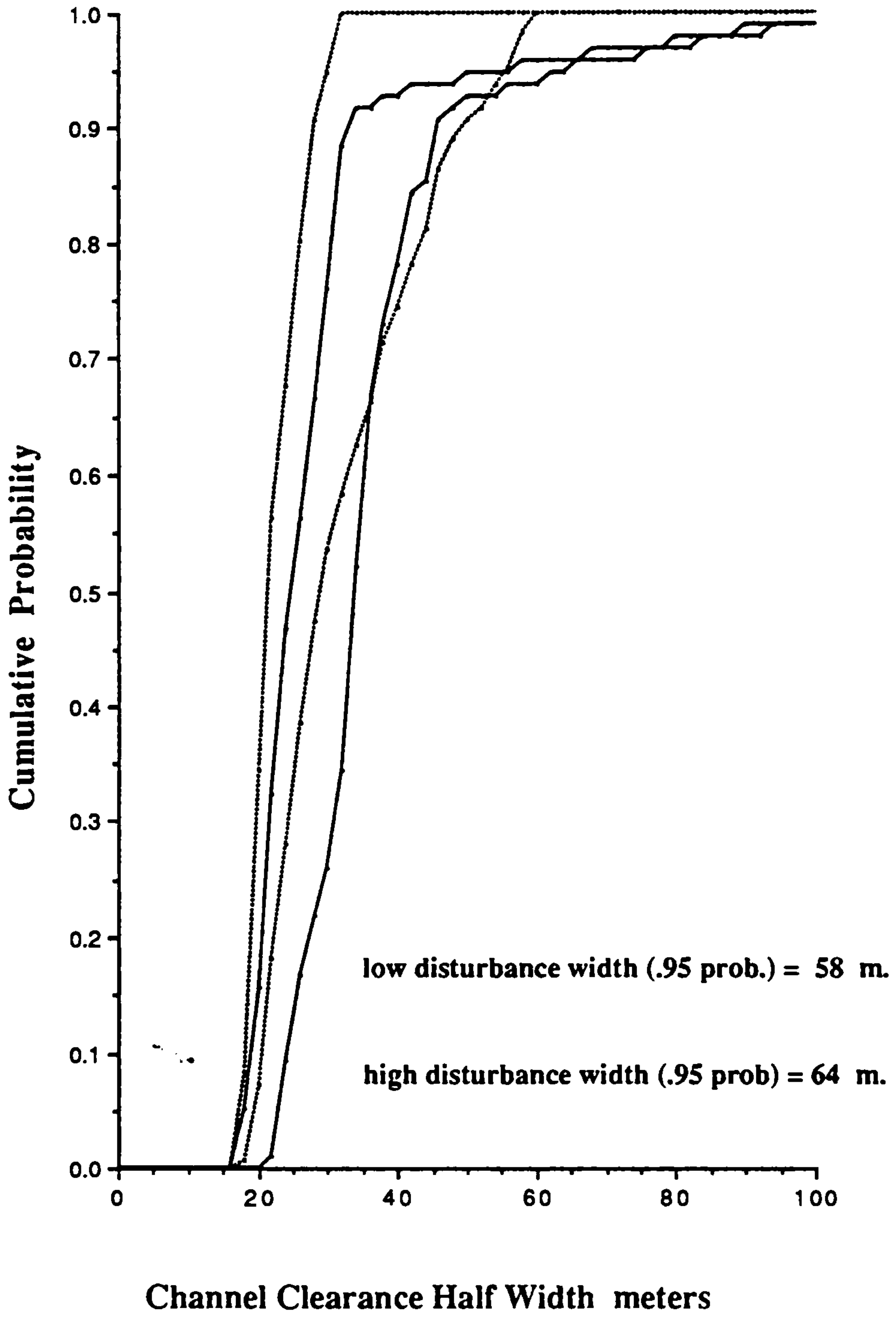


Figure 6-17 Channel Clearance Width Distributions Before Turn, RACON

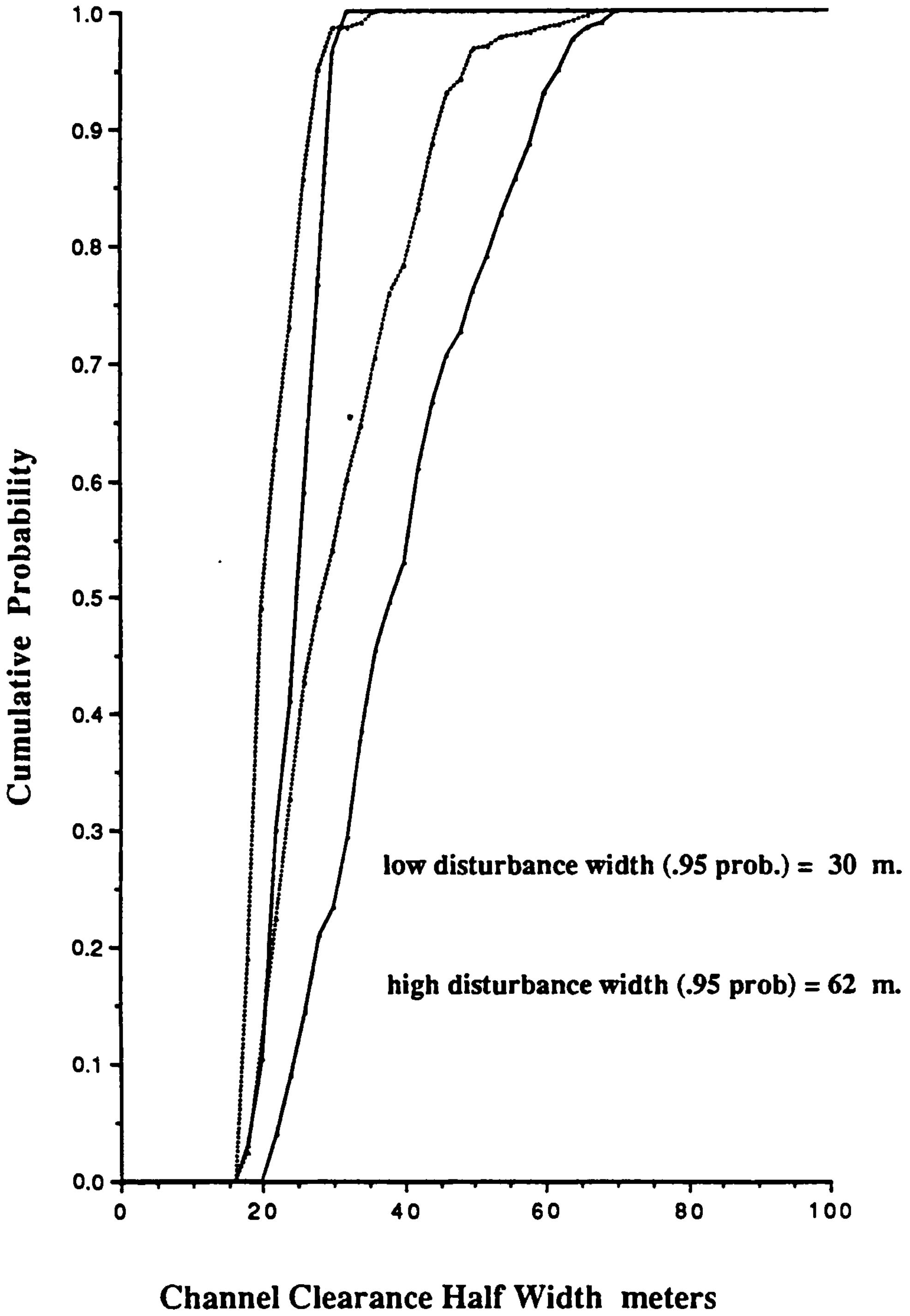


Figure 6-18 Channel Clearance Width Distributions After Turn, RACON

accurate DGPS system. However, given the angular accuracies delivered by the standard ship radar, siting geometry is a critical factor. A more precise monopulse ship interrogator would greatly ease the geometric problems of the RACON system.

6.6 Microwave Performance

As discussed in Section 2.1.1, ground based ranging sensors are commonly available in UHF and microwave variants. The microwave versions have a more limited range but are more accurate. All have quick response time and good dynamical performance. The range tracking parameters chosen for simulation are representative of the microwave systems. Tracking loop time constants were set to one second, and loop output jitter was set to 1 meter. Range loops were sampled by the navigation filter at a 1 Hz. rate. Three transponders were used in an overdetermined solution and, as shown in **Figure 6-1**, the layout of the three microwave transponder sites was nearly ideal. In practice, siting difficulties may preclude such good geometry.

Not surprisingly, the overall performance of the simulated microwave system was excellent. **Figures 6-19** and **6-20** show the low and high disturbance CCW results for two typical runs. **Figures 6-21** and **6-22** show the tight distributions obtained before and after the turn, under conditions of low disturbance. Insufficient runs were made under the high disturbance condition to obtain reliable cumulative distributions, but the quick response of the microwave sensor gave very good results in this case.

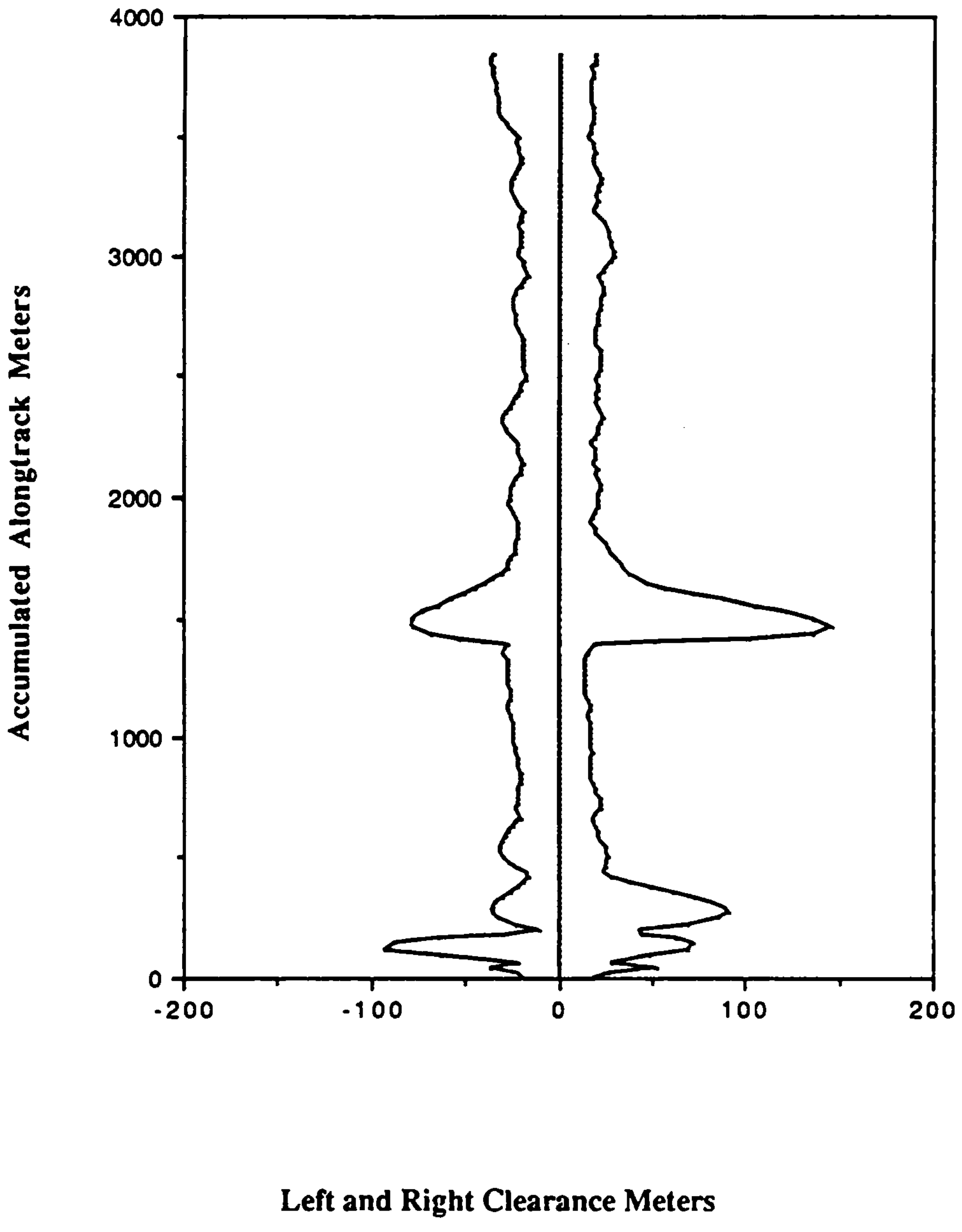
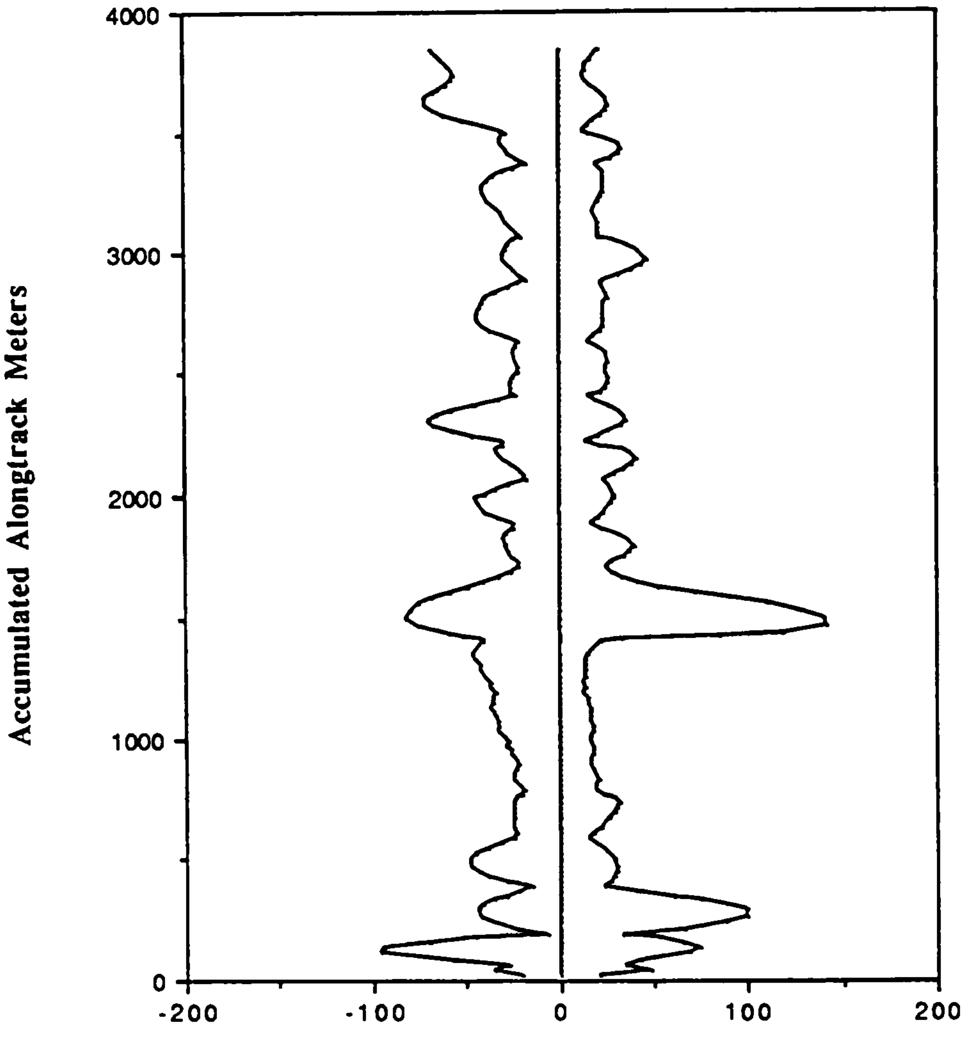


Figure 6-19 CCW, Microwave Sensor, Low Disturbance



Left and Right Clearance Meters

Figure 6-20 CCW, Microwave Sensor, High Disturbance

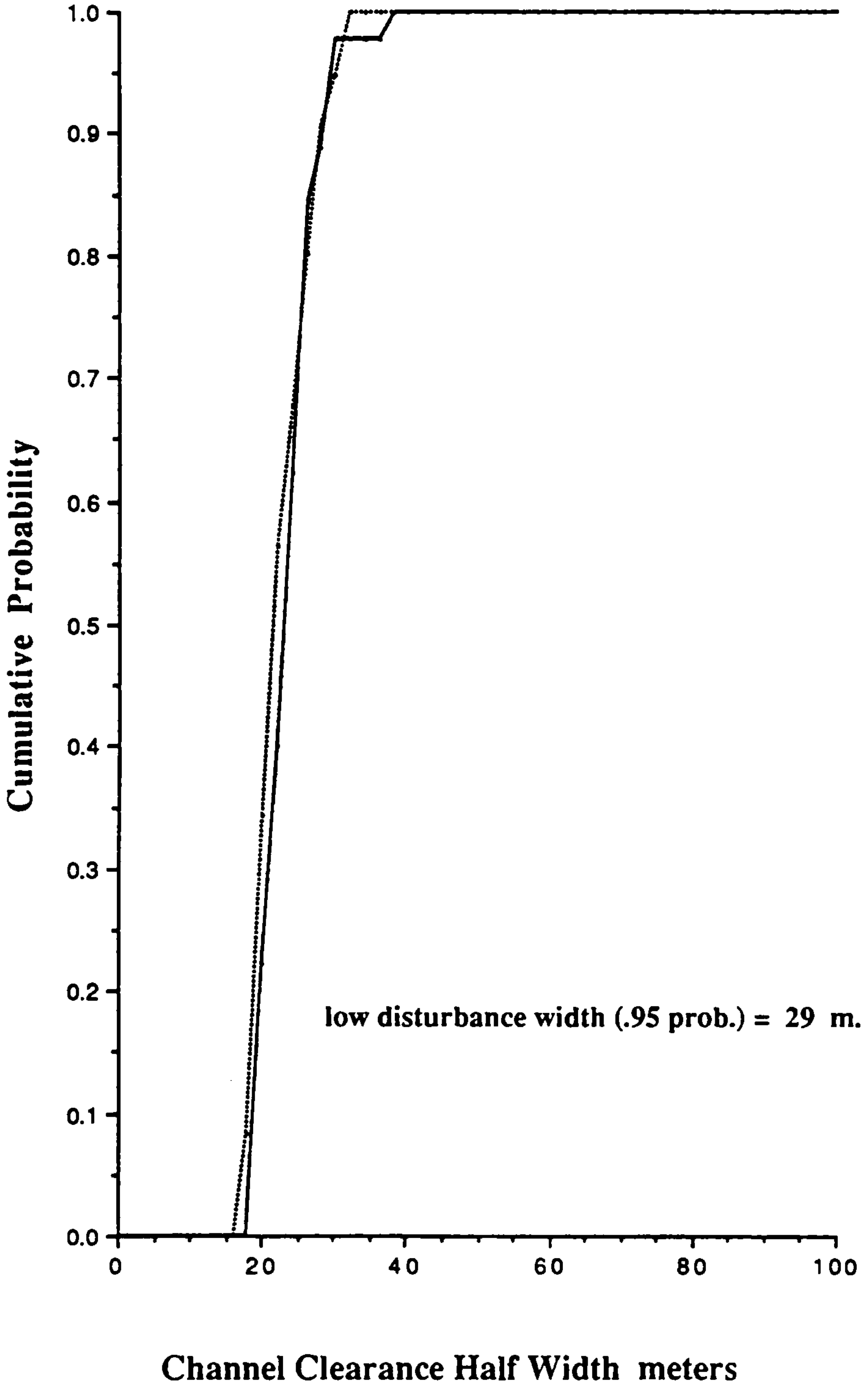


Figure 6-21 Channel Clearance Width Distributions Before Turn, Microwave

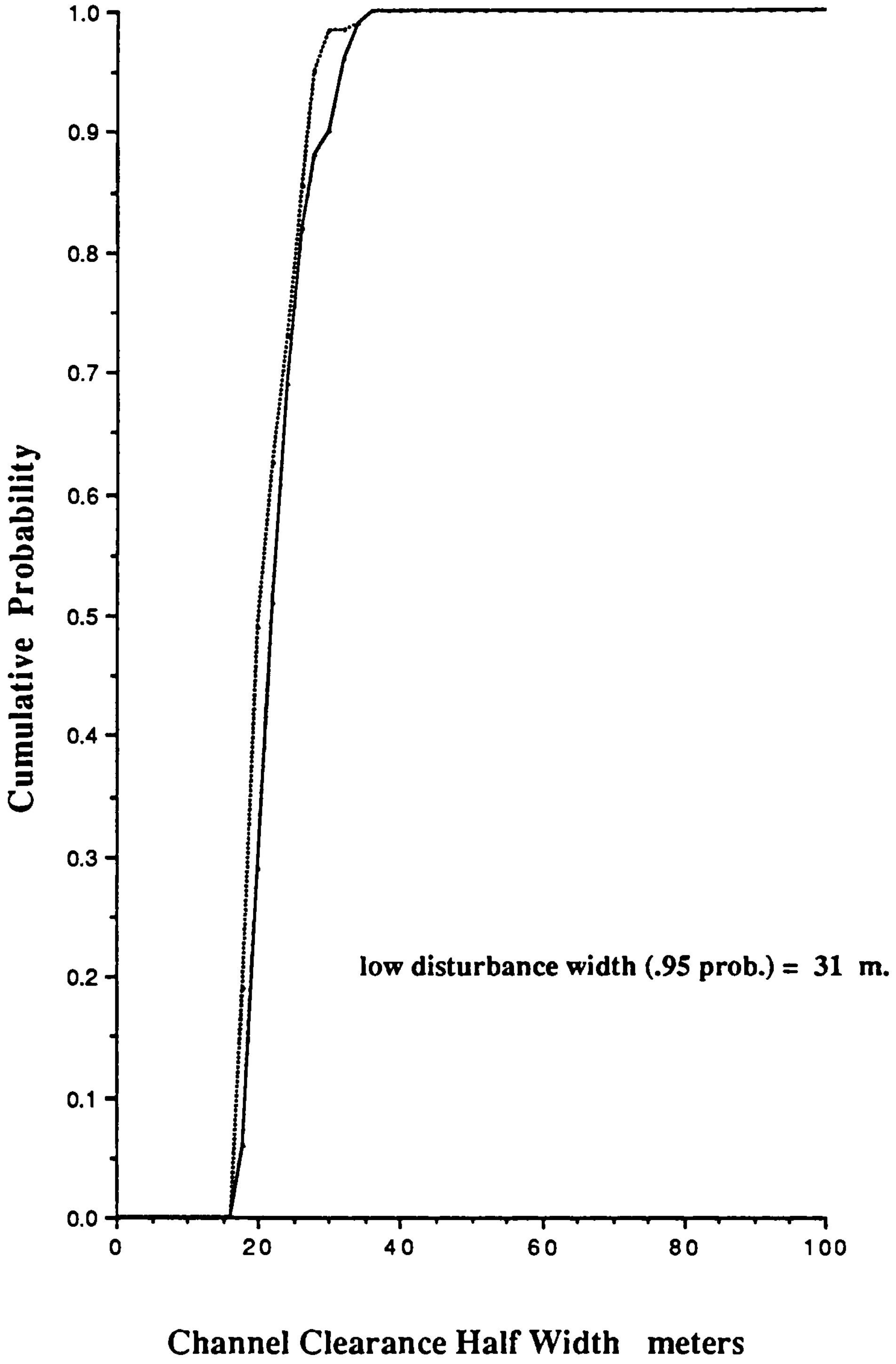


Figure 6-22 Channel Clearance Width Distributions After Turn, Microwave

7. Conclusions and Directions for the Future

As demonstrated in this report, precise control of the ship footprint is ultimately limited by ship controllability factors, which are independent of the sensor system. Channel clearance width distributions were especially good for the DGPS and microwave candidates, and when operated with good geometry, the RACON system provided adequate steering performance. The inadequacy of simple position fixing specifications was seen in the performance of the LORAN-C system. Tracking loop sensor dynamical models and a closed loop steering simulation provide a more complete picture of what can be expected from each sensor combination.

Reliability and integrity factors must be a part of the sensor selection process. A local monitor/calibration receiver can provide a warning to the user of an out-of-tolerance condition, thus enhancing system integrity. For two of the candidates, DPGS and differential LORAN-C, this is a natural by-product of the differential calibration process. Sensor reliability can be achieved by a combination of techniques, beyond the obvious measure of increasing the number of transponder/transmitter sites. Tolerance to unanticipated signal outages can be improved by utilizing secondary navigation sources, or by providing a more stable receiver clock. Although signal outage performance has been briefly explored for DGPS, a comprehensive range of outage scenarios across the candidate mix should be investigated. Further tests, under less ideal conditions, can be explored with the present simulation system.

The integration of precision navigation data into the ship radar and map displays remains a very important avenue for further exploration. To approach the closed loop ship footprint control distributions predicted in the simulation will require very careful design of the man-machine interface. An "aided-pilot" feature could be a useful adjunct to this interface. Such a system could monitor present ship state and helm commands, and warn of a potentially hazardous condition. Much of this system integration must be done on a full-fledged ship simulator. However, simulators of the type employed in this report provide an economical means for studying the efficacy of simple pilot displays, and their interaction with rudder control and sensor functions. Although VAX computers were employed in the present work, a high performance 32-bit workstation is capable of supporting the ship and sensor dynamics environment, navigation filtering software , and a simplified pilot display.

Advancing our knowledge of confined waterway visual pilotage performance and vessel steering response should also be pursued. Operating in its integrated Doppler mode, the DGPS system provides a low-cost means of gathering this fine-grained data. A DGPS experiment aimed at several vessel types and pilotage situations, is recommended. A secondary objective of this activity would be the gathering of DGPS signal reliability/blockage data.

Finally the specification of required navigation accuracies in the Federal Radionavigation Plan (FRP) should be examined in the context of closed loop steering control and precision pilotage, and new standards for the characterization of sensor dynamical performance should be established.

Appendix A

A.1 Hydrodynamic coefficients

For more information, refer to [B-9}.

$C_{v\omega}$ derivative of transverse specific force coefficient with respect to ship's space angular rate

$C_{vv'}$ derivative of transverse specific force coefficient with respect to yaw angle

$C_{v\delta}$ derivative of transverse specific force coefficient with respect to rudder angle

$C_{\omega\omega}$ derivative of specific torque coefficient with respect to ship's space angular rate

$C_{\omega v'}$ derivative of specific torque coefficient with respect to yaw angle

$C_{\omega\delta}$ derivative of specific torque coefficient with respect to rudder angle

Appendix B

B.1 Riccati Equation

The Riccati equation can be solved by many methods - recursive method, eigenstructure assignment, and numerical computation. Backward and forward recursive method provides an optimal gain sequence but requires rather extensive off-line computation and additional coding complexity which must generate a predicted destination in advance and go backward in time from a destination point to a current location for gain computation. Particularly, during a turning predicting an intermediate destination point is difficult. Since the turning involves rotation of the coordinate frame. On the contrary, the eigenstructure method does not involve off-line backward iteration for a sequence of gain matrix. It is a purely algebraic method and furthermore provides a stable solution. However, it only yields a steady state gain which is suboptimal. The numerical computation, as its name implies, cranks the Riccati by directly substituting an initial guess into the equation and repeats iterations until desired error accuracy is achieved. It also yields the same suboptimal gain as the solution of the eigenstructure method.

In practice, the suboptimal gain can be used without much degradation of the system performance. As we assumed in **Section 5.5**, the destination is not clearly set up in advance. Instead, our emphasis is to regulate crosstrack and heading angle errors as small as possible with respect to a linear segment of the trajectories. To achieve the regulation, we mostly need the suboptimal gain sequence, not the off-line optimal solution. In fact, in our study the system has responded well to the suboptimal gain sequence.

Consider the Riccati solution using the eigenstructure method which provides the theoretical suboptimal solution. We will discuss this method and compare it to the direct numerical computation in **Section B.2**.

A solution sequence S(k) of the discrete-time Riccati equation is given by

$$S(k) = \Phi^T[S(k+1) - S(k+1)\Gamma\{\Gamma^T S(k+1)\Gamma + R\}^{-1}\Gamma^T S(k+1)]\Phi + Q$$

where Φ is the state transition matrix, Γ the input matrix, Q the weighting matrix for states, and R the weighting matrix for the inputs as shown in **Sections 5.4** and **5.5**.

Then the gain sequence G(k) is obtained from the Riccati solution sequence S(k) as follows:

$$G(k) = \{\Gamma^T S(k+1)\Gamma + R\}^{-1}\,\Gamma^T S(k+1)\Phi$$

The steady state solution of S(k) is obtained as k approaches to infinity. Hence,

$$S(\infty) = \lim_{k\to\infty} S(k)$$

Then the steady state gain G = G(∞) becomes

$$G = \{\Gamma^T S(\infty)\Gamma + R\}^{-1}\,\Gamma^T S(\infty)\Phi$$

Then the suboptimal control input u(k) to the controller is given by

$$\underline{u}(k) = -\,G\hat{\underline{x}}(k) \qquad , \quad \text{where } \hat{\underline{x}}(k) \text{ is an estimate of } \underline{x}(k).$$

B.2 Coding of Riccati solution

Generating the Riccati gain sequence requires several subroutines, especially, subroutines from EISPACK, in the eigenstructure method. Initially the eigenstructure method for the Riccati solution was implemented using FORTRAN in a CYBER main frame computer. Main reason for using the CYBER was lack of proper softwares for eigenvalue and eigenvector matrix computation in mini- or micro-computer system. Because this method requires the formulation of the Hamiltonian matrix H and computation of eigenvalues and modal matrix for the H. However, main problem we have observed was numerical instability due to unstable eigenvalues in the ship state variables x_1, x_2, and x_3. Since the ship model contains an unstable state and a pure integrator, the eigenvalues of the H for the discretized linear system are clustered near the unit circle in the complex Z-plane. Furthermore, because of round-off errors accumulated during computation, it has been detected that eigenvalues of the H cross the unit circle and thus cause instability in computation itself as well as generate unstable system gains. Careless coding will yield an unreliable gain sequence. To correct the problem, sorting in descending order in absolute values of eigenvalues is needed and based on the ordering, swapping of columns of the eigenvector matrix is required. EISPACK subroutines generally provide a modal matrix which stores the real and imaginary parts of eigenvectors in a special form, the column swapping is very important. If the swapping is not properly done, then the result is not reliable.

Because of the above reasons, the eigenstructure method was implemented but later replaced by direct numerical computation of the Riccati solution. The direct computation does not require the formulation of the Hamiltonian and, hence, computing and sorting eigenvalues. Nowadays, use of microcomputers is increasingly popular and economical. But not many numerical analysis software packages are available in the microcomputer compared to the main frame computer system. Clearly, the direct numerical computation is a better approach than the eigenstructure method which requires many sophisticated subroutines for computing eigenvalues and eigenvectors. In our study the controller has been implemented based on the direct computation approach using C language.

If we carefully examine the Riccati equation in **Section B.1**, there is a steady state solution $S(\infty)$ as k approaches to ∞. Since what we need is the steady state solution, we start solving the equation recursively with an arbitrary positive definite matrix $S(0)$. The numerical computation method is slow without employing any acceleration scheme for convergence. However, the coding is simple and straight forward. It yields a reliable result compared to the frequently unstable eigenstructure method. To preserve the symmetry in $S(k)$ at each iteration step, $S(k)$ and $S(k)^T$ are summed together and averaged.

B.3 Reduction of computation time

Computation of the Riccati gain sequence is the most time consuming part in the simulation. From the simulation study, it has been learned that the ship dynamics do not change significantly if there are no abrupt changes in the environment or no turning in a foreseeable period. Hence, frequent gain updates are unnecessary. To alleviate the computational burden of updates, changes in heading angle and crosstrack errors are constantly checked. If changes exceed a certain threshold, then evaluate a new dynamics and corresponding gains.

Appendix C

Ship Simulation Source Code

1) Brief Listing of Functions and Resident Functions
2) Source Code
3) Expanded File and Function Listing
4) Control File Listing
5) Control File Description

Brief Listing of Files and Resident Functions

File	Functions	Comments
array.h	--	array structure
con.c	controller environment SensorSimulator EnvironmentSTM UpdateSTM riccati CompressPhiGamma ExpandGain	environment routines
est.c	estimator extrapolate update NewObservation ComputeResidual EstimatorSTM NominalMeasurements CheckFilterStatus RestartFilter UpdateQ ComputeDops	estimator routines
init.c	InitConstants ReadControlFile Sizes CreateMemory InitArrays	initialization functions
output.c	output PrintResidual	send results to output
ship.c	main InitAll	start of program
ship.h	--	ship defines and structure templates
store.c	--	external variable storage
store.x	--	external declarations
utils.c	mondecode MType cmdln help finish	utilities

	ACtoEN	
	AngleSense	
	NormalizeAngle	
	noise	
	fe	
	decode	
way.c	CheckWaypointStatus	waypoint handling routines
	ReadWaypoint	
	ProcessMap	
	RotateShip	

Array.h

```
struct array {

      char   *name;              /* array name */
      int    p;                  /* number of rows */
      int    n;                  /* number of columns */
      double *_a;                /* type double pointer to linear array */

};


double get(),put();
struct array *add(), *sub(), *copy(), *dim(), *ident(), *inv();
struct array *mul(), *neg(), *scal(), *tpos(), *zero();

extern FILE *inpfp, *outfp;
# define MULTMP 256
# define matrix(p,n) (*(p->_a+n))
# define vector(p,n) matrix(p,n)
# define scalar(p) (*(p->_a))
# define row(r) (r->p)
# define col(c) (c->n)
# define base(a) (a->_a)
```

Con.c

```
# include <stdio.h>
# include <math.h>
# include "array.h"
# include "ship.h"

# include "store.x"

/*********************************************************************
 *  Functions contained in this file are:
 *
 *    controller()             -     compute control input U
 *    environment()            -     update plant
 *    SensorSimulator()        -     create sensor readings for estimator
 *    EnvironmentSTM()         -     update regulator phi
 *    UpdateSTM()              -     update given state transition matrix
 *    riccati()                -     compute Riccati gain
 *    CompressPhiGamma()       -     convert phi and gamma to 7x7 and 7x2
 *    ExpandGain()             -     expand gain to size needed
 *    ninv()                   -     inverse with pivoting
 *
**********************************************************************/

/*********controller**************************************************/

double LastClock=0.0;

/*
 *  Compute controlling 'optimal' input U = -K*X.
 *  For these runs ship velocity is held constant.
 *  NOTE: GAIN IS ALREADY NEGATED
 */
controller()
{
      if(TrueFlag) {
            mul(true.ship.gain, true.ship.x, true.ship.u);
            vector(true.ship.u,1) = true.ship.vel;
      } else {
            mul(nav.ship.gain, nav.fil.x, true.ship.u);
            vector(true.ship.u,1) = nav.ship.vel;
      }

}


/********* Environment ***********************************************/

/*
 *  Update the plant.
 */
environment()
{
      int i;                  /* loop incrementor */
      double p;               /* phi element in noise calculation */

      /*
```

```
       *  Check need for plant state transition matrix update
       *
      if( fabs(HeadingError(true.ship.x)-LastPHE)>HeadingThreshold
        || fabs(CrossTrack(true.ship.x)-LastPCT)>CrossTrackThreshold) {

                  EnvironmentSTM();
      }

      /*
       *  Update plant: X(n+1) = PHI*X(n) + GAMMA*U(n) + noise
       */
      mul(true.ship.gamma, true.ship.u, tmp2);
      mul(true.ship.phi, true.ship.x, tmp1);
      add(tmp1, tmp2, true.ship.x);

      /*
       *  Add disturbances to:
       *    Sway Velocity
       *    Yaw Rate
       *    East Current
       *    North Current
       *    Clock Drift
       */
      p = get(true.ship.phi,1,1);
      SwayVelocity(true.ship.x)
                  += noise((one-p*p)*true.ship.variance[0]);

      p = get(true.ship.phi,2,2);
      YawRate(true.ship.x)
                  += noise((one-p*p)*true.ship.variance[1]);

      p = get(true.ship.phi,6,6);
      vector(true.ship.x,5)
                  += noise((one-p*p)*true.ship.variance[2]);

      p = get(true.ship.phi,7,7);
      vector(true.ship.x,6)
                  += noise((one-p*p)*true.ship.variance[3]);

      p = get(true.ship.phi,9,9);
      ClockDrift(true.ship.x)
                  += noise((one-p*p)*true.ship.variance[4]);
}


/********* SensorSimulator ********************************************/

/*
 *  Simulate sensor measurements for use in estimator.  Sensor lags
 *  and noise are included, if desired.
 */
SensorSimulator()
{
      int i;                  /* loop increment */

      double
            dc,               /* cross track position difference */
            da,               /* along track position difference */
      ClockBias,        /* current clock bias */
      RawMeasurement,   /* measurement without lag or noise */
      NoiseValue        /* current noise value */
;

/*
 *  Compute raw range measurements
 */
for(i=0; i<SensorChannels; ++i) {

      switch(sensor[i].type) {

      case RANGE:
            dc = CrossTrack(true.ship.x) - sensor[i].cross;
            da = AlongTrack(true.ship.x) - sensor[i].along;
            RawMeasurement = sqrt(dc*dc + da*da);
            ClockBias = LastClock;
            break;

      case HEADING:     /* heading error for now */
            RawMeasurement = HeadingError(true.ship.x);
            ClockBias = 0.0;
            break;

      case BEARING:
            dc = sensor[i].cross - CrossTrack(true.ship.x);
            da = sensor[i].along - AlongTrack(true.ship.x);
            RawMeasurement = atan2(dc,da)
                        - HeadingError(true.ship.x);
            ClockBias = 0.0;
            break;

      default:
            fprintf(ErrorFile,
"SensorSimulator: unknown measurement type %d for element = %d\n",
                  sensor[i].type,i);

      }

/*
 *  Simulate sensor lags and add noise:
 *
 *  X(k+1) = PHI*X(k) + GAMMA*RawMeasurement + NOISE
 */
      if(LagFlag) {

            NoiseValue = noise(
       (one - sensor[i].phi * sensor[i].phi) * sensor[i].EnvVar
                              );

            Measurement(i)
                  = sensor[i].phi * Measurement(i)
            + sensor[i].gamma*(LastRawMeasurement[i]-ClockBias)
                        + NoiseValue;

      } else {
            Measurement(i) = RawMeasurement;
```

```
                }
                LastRawMeasurement[i] = RawMeasurement;
        }

        LastClock = Clock(true.ship.x)*c;
}


/********* EnvironmentSTM ***********************************************/

/*
 *  Update environment's state transition matrix
 */
EnvironmentSTM()
{
        double
                SinX3,                          /* sin(HeadingError) */
                CosX3;                          /* cos(HeadingError) */

        /*
         *  update A & B matrices and call RICATTI
         */
        SinX3 = sin(HeadingError(true.ship.x));
        CosX3 = cos(HeadingError(true.ship.x));
        put(true.ship.a, CosX3, 4,1);
        put(true.ship.a,
                -SwayVelocity(true.ship.x)*SinX3+true.ship.vel*CosX3, 4,3);
        put(true.ship.a, -SinX3, 5,1);
        put(true.ship.a, -SwayVelocity(true.ship.x)*CosX3, 5,3);

        put(true.ship.b, CosX3, 5,2);

        UpdateSTM(&true.ship);

        LastPHE = HeadingError(true.ship.x);
        LastPCT = CrossTrack(true.ship.x);

        STMStatus |= 1;
}


/********* UpdateSTM ****************************************************/

/*
 *  Update given state transition matrix
 */
UpdateSTM(s)
struct _ship *s;
{
        int i;

        /*
         *  Compute state transition matrix and gamma
         *  NOTE:  tmp3 = psi
         */
        scal(s->a, TimeStep, tmp1);
        copy(s->e, tmp3);
        for(i=60; i>=2; i--) {
            mul(tmp1, tmp3, tmp2);
            scal(tmp2, 1.0/(double)i, tmp2);
            add(s->e, tmp2, tmp3);
      }
      mul(tmp1, tmp3, tmp2);
      add(s->e, tmp2, s->phi);
      mul(tmp3, s->b, tmp1);
      scal(tmp1, TimeStep, s->gamma);

      if(TrueFlag){
            if(fabs(HeadingError(true.ship.x)) >= RicHEThreshold
                  || fabs(CrossTrack(true.ship.x)) >= RicCTThreshold) {

                  riccati(&true.ship);
            }

      } else {
            if(fabs(HeadingError(nav.fil.x)) >= RicHEThreshold
                  || fabs(CrossTrack(nav.fil.x)) >= RicCTThreshold) {
                  riccati(&nav.ship);

            }
      }
}


/********* riccati ******************************************************/

/*
 *  Compute optimal gain via Riccati equation
 */
riccati(s)
struct _ship *s;
{
      int i;                    /* loop incrementor */
      struct _control *c;
      struct array *ninv();

      c = &control;

      CompressPhiGamma(s);      /* convert phi and gamma to 7x7 and 7x2 */

      /*
       *  The following code implements
       *
       *    P(k+1) = Q + PHI'*P(k)
       *          * [PHI - GAMMA * inv[R + GAMMA'*P(k)*GAMMA]
       *                * GAMMA'*P(k)*PHI
       *
       *  where P is the performance index, and Q and R are performance
       *  indices.  To maintain symmetry P is recomputed as:
       *
       *          P = (P + P')/2
       *
       */

      ident(c->p);
      for(i=1; i<=100; i++)  {
```

```
                mul(c->gammat, c->p, tmp1);   /* TMP1 = GAMMA'*P */
                mul(tmp1, c->gamma, tmp2);    /* TMP2 = GAMMA'*P*GAMMA */
                add(tmp2, c->r, tmp2);        /* TMP2 = R+GAMMA'*P*GAMMA */
                ninv(tmp2, tmp3);             /* TMP3 = inv(TMP2) */
                mul(tmp1, c->phi, tmp2);      /* TMP2 = GAMMA'*P*PHI */
                mul(tmp3, tmp2, tmp1);        /* TMP1 = inv()*GAMMA'*P*PHI*/
                mul(c->gamma, tmp1, tmp2);    /* TMP2 = GAMMA*TMP1 */
                sub(c->phi, tmp2, tmp1);      /* TMP1 = PHI-GAMMA*inv() */
                mul(c->phit, c->p, tmp2);     /* TMP2 = PHI'*P */
                mul(tmp2, tmp1, tmp3);        /* TMP3 = PHI'*P*TMP1 */
                add(c->q, tmp3, c->p);        /*    P = Q + TMP3 */
                tpos(c->p, tmp1);             /* TMP1 = P' */
                add(c->p, tmp1, c->p);        /*    P = P + P' */
                scal(c->p, 0.5, c->p);        /*    P = P/2 */
        }

        /*
         *  Finally, we compute the gain:
         *
         *  GAIN = inv(GAMMA'*P*GAMMA + R) * GAMMA'*P*PHI
         */
        mul(c->gammat, c->p, tmp1);   /* TMP1 = GAMMA'*P */
        mul(tmp1, c->gamma, tmp2);    /* TMP2 = GAMMA'*P*GAMMA */
        add(tmp2, c->r, tmp2);        /* TMP2 = R + GAMMA'*P*GAMMA */
        ninv(tmp2, tmp3);             /* TMP3 = inv(TMP2) */
        mul(tmp1, c->phi, tmp2);      /* TMP2 = GAMMA'*P*PHI */
        mul(tmp3, tmp2, c->gain);     /* GAIN = inv()*GAMMA'*P*PHI */

        put(c->gain, 0.0, 2,5);
        neg(c->gain, c->gain);        /* for U = -KX later */
        ExpandGain(s);                /* expand gain to size needed */
        STMStatus |= 4;               /* indicate a Ricatti call */
}


/********* CompressPhiGamma ******************************************/

/*
 *  Compress phi and gamma to basic 7x7 and 7x2 size for ricatti().
 */
CompressPhiGamma(s)
struct _ship *s;
{
        int
                i,          /* row loop incrementor */
                j           /* column loop incrementor */
        ;

        for(i=1; i<=ShipStates; ++i) {
                for(j=1; j<=ShipStates; ++j) {
                        put(control.phi, get(s->phi, i,j), i,j);
                }
        }

        tpos(control.phi, control.phit);

        for(i=1; i<=ShipStates; ++i) {
                put(control.gamma, get(s->gamma, i,1), i,1);
```

```
            put(control.gamma, get(s->gamma, i,2), i,2);
      }

      tpos(control.gamma, control.gammat);
}


/********* ExpandGain *************************************************/

/*
 *  Expand gain to 2x9 (true state feedback) or 2x14
 *  (estimated state feedback), as required.
 */
ExpandGain(s)
struct _ship *s;
{
      int   j;    /* column incrementor */

      for(j=1; j<=ShipStates; ++j) {
            put(s->gain, get(control.gain, 1,j), 1,j);
            put(s->gain, get(control.gain, 2,j), 2,j);
      }

}


/********* ninv *******************************************************/

/*
 *  New matrix inverse.  Find square matrix inverse using the
 *  Gaussian row reduction with column pivoting.
 */
struct array *ninv(a,b)
register struct array *a,*b;
{
      register int i,j,n,order;
      register double
            z,                /* diagonal value */
            *adp,             /* a diagonal pointer */
            *acp,             /* a column pointer */
            *anp, *bnp,       /* [ab] n-row place-holder */
            *arm, *brm,       /* [ab] current max row */
            *aro, *bro,       /* [ab] current origin row   */
            *arp;             /* a row pointer  */

#ifndef FAST
      if(row(a) != col(a)){
            fprintf(stderr,"ninv: non-square matrix %s.\n",a->name);
            exit(0);
      }
      if(a==0 || b==0){
            fprintf(stderr,"ninv: null array pointer.\n");
            exit(0);
      }
#endif

      order = col(b) = row(b) = col(a);
      ident(b);
      adp = acp = anp = base(a);
      bnp = base(b);

      for(n=1; n<=order; ++n,adp+=order+1,acp++,anp+=order,bnp+=order){
            aro = arm = arp = adp;
            arp += order;
            bro = bnp;
                                    /* pick up pivot element */
            for(i=n; i<order; i++,arp+=order)
                  if(fabs(*arp) > fabs(*arm))
                        arm = arp;

            if((*arm)==0.0){
                  fprintf(stderr,
                        "ninv: zero elements in one column.\n");
                        exit(0);
            }
                        /* if i row not max, then interchange  */
            if((*arm) != (*aro)) {
                  i=n;
                  for(; i<=order; i++) {
                        z = *arm;
                        *arm++ = *aro;
                        *aro++ = z;
                  }
                  j = arm - aro;
                  brm = bro + j;
                  for(i=1; i<=order; i++) {
                        z = *brm;
                        *brm++ = *bro;
                        *bro++ = z;
                  }
            }
      /*
       *  Reduce the remaining n-1 rows with in [AB] row n.
       */
           for(i=1,arm=acp,arp=base(a),brm=base(b); i<=order;
            ++i,arm+=order){
                  if(i==n){ arp += order; brm += order; continue; }
                  z = (*arm)/(*adp);
                  for(j=1,aro=anp,bro=bnp; j<=order; ++j){
                        *arp++ -= (*aro++)*z;
                        *brm++ -= (*bro++)*z;
                  }
            }
      }

      /*
       *  Normalize n_th row in [AB] matrix with element A(n,n).
       */
      for(n=1,adp=base(a),bro=base(b); n<=order; n++,adp+=order+1) {
            z = *adp;
            for(i=1; i<=order; i++)
                  *bro++ /= z;
      }
      return(b);
}
```

Est.c

```
# include <stdio.h>
# include <math.h>
# include "array.h"
# include "ship.h"

# include "store.x"

# define RUDDER
# undef FAILURE
# undef RESETP

/**********************************************************************
 *  Functions contained in this file are:
 *
 *    estimator()              -     executes ship Kalman filter
 *    extrapolate()            -     extrapolates estimated ship state
 *    update()                 -     update estimated ship state
 *    NewObservation()         -     observation for i'th measurement
 *    ComputeResidual()        -     residual for i'th measurement
 *    EstimatorSTM()           -     updates filter phi
 *    NominalMeasurements()    -     measurements based on nominal state
 *    CheckFilterStatus()      -     check threshold levels, etc.
 *    RestartFilter()          -     prepare filter for transient, etc.
 *    UpdateQ()                -     recompute Q
 *    ComputeDops()            -     compute some DOP values
 *
 *********************************************************************/


/********* estimator **************************************************/

/*
 *  Executes the ship's Kalman filter.
 */
estimator()
{
      CheckFilterStatus();
      extrapolate();
      update();
}


/********* extrapolate ************************************************/

/*
 *  Extrapolate state and covariance TimeStep seconds.
 */
extrapolate()
{
      int i;

      /*
       *  Form proper control input for the filter
       */
# ifdef RUDDER
      vector(nav.ship.u,0) = vector(true.ship.u,0);
```

```
# else
      vector(nav.ship.u,0) = 0.0;
# endif
      vector(nav.ship.u,1) = vector(true.ship.u,1);
      if(SensorFlag) {
            NominalMeasurements();
      }

      /*
       *  Extrapolate nominal state:  XNOM = PHI*XNOM
       *  NOTE: sensors have no nominal values!
       */
      mul(nav.ship.phi, nav.fil.xnom, nav.fil.xnom);
      for(i=0; i<SensorChannels; ++i) {
            SensorState(nav.fil.xnom,i) = 0.0;
      }

      /*
       *  Extrapolate incremental state:  DX = PHI*DX + GAMMA*U
       */
      mul(nav.ship.phi, nav.fil.dx, nav.fil.xdx);
      mul(nav.ship.gamma, nav.ship.u, tmp1);
      add(nav.fil.xdx, tmp1, nav.fil.dx);

      /*
       *  Extrapolate covariance:  P = PHI*P*PHIT+Q
       */
      mul(nav.ship.phi, nav.fil.p, tmp1);
      mul(tmp1, nav.ship.phit, tmp2);
      add(tmp2, nav.fil.q, nav.fil.p);

      if(MonitorFlag){
            if(MonitorStatus&X)
                  out("nav.fil.dx: after ext.\n",nav.fil.dx);
            if(MonitorStatus&P)
                  out("nav.fil.p: after ext.\n",nav.fil.p);
      }
}


/********* update ****************************************************/

/*
 *  Update Kalman gain, covariance, and incremental state.
 */
update()
{
      int
            i,                  /* loop increment */
            BadResidualCount    /* number of bad residuals this time */
      ;

      double
            ComputeResidual(),       /* compute residual for i'th meas.
*/
            residual,                /* filter measurement */
            rw                       /* weight for current meas. */
      ;
      BadResidualCount = 0;         /* set bad residual count to zero */

      /*
       *  Compute residual and update Kalman gain, covariance, and
       *  state for each sensor's data.
       */
      for(i=0; i<SensorChannels; ++i){

            /*
             *  Compute observation and residual for i'th measurement.
             */
            NewObservation(i);
            residual = ComputeResidual(i);

            /*
             *  Check residual against desired thresholds value.
             *  If threshold is exceeded deweight current measurement;
             *  else, use normal noise value.
             */
            if(fabs(residual) > sensor[i].threshold[ThresholdLevel]){
                  BadResidualCount++;
                  rw = sensor[i].MNoise[HIGH];  /* de-weighting */
                  PrintResidual(residual,i,"BAD");
                  residual = 0.0;               /* replacement  */
            } else {
                  PrintResidual(residual,i,"   ");
                  rw = sensor[i].MNoise[LOW];   /* normal weighting */
            }

# ifdef FAILURE
            /*
             *  Simulate sensor outage.  Sensor 0 was
             *  chosen arbitrarily.
             */
            if(ShipFrompointIndex>0 && i==0) {
                  rw = sensor[i].MNoise[HIGH];
            }
# endif

            if(UpdateFlag) {

                  /*
                   * GAIN UPDATE:   K = P*HT*inv[H*P*HT+R]
                   */
                  mul(nav.fil.p, nav.fil.ht, nav.fil.k);
                  mul(nav.fil.h, nav.fil.k, tmp2);
                  scal(nav.fil.k, one/(scalar(tmp2)+rw), nav.fil.k);

                  /*
                   *  COVARAINCE UPDATE: P = (I-K*H)*P
                   */
                  mul(nav.fil.k, nav.fil.h, tmp1);
                  mul(tmp1, nav.fil.p, tmp2);
                  sub(nav.fil.p, tmp2, nav.fil.p);

                  /*
                   *  INCREMENTAL STATE UPDATE: DX = DX+K*(Z-H*DX)
```

```
                         */
                        scal(nav.fil.k, residual, tmp1);
                        add(nav.fil.dx, tmp1, nav.fil.dx);
                }

                /*
                 *  Update true state estimate:  X = XNOM + DX
                 */
                add(nav.fil.xnom, nav.fil.dx, nav.fil.x);

                if(MonitorFlag) {
                        if(MonitorStatus & H) out("nav.fil.h:\n",nav.fil.h);
                        if(MonitorStatus & K) out("nav.fil.k:\n",nav.fil.k);
                        if(MonitorStatus & X) out("nav.fil.dx:\n",nav.fil.dx);
                        if(MonitorStatus & P) out("nav.fil.p:\n",nav.fil.p);
                        if(MonitorStatus & RES)
                                fprintf(OutputFile,"res: %lf\n",residual);
                }
        }

        /*
         *  If any residuals were bad, update the bad update count
         */
        if(BadResidualCount > 0) {
                ++BadUpdateCount;
        }
}


/********* NewObservation ********************************************/

/*
 *  Compute direction cosines from sensor to ship; also compute
 *  estimated range and bearing angle from ship to sensor, and
 *  estimated heading.
 */
NewObservation(n)
int n;                  /* channel type */
{
        double
                dc,     /* cross track distance from ship to sensor */
                da      /* along track distance from ship to sensor */
        ;

        zero(nav.fil.h);
        zero(nav.fil.ht);

        switch(sensor[n].type){

                /*
                 *  Range sensor
                 */
                case RANGE:
                        dc = CrossTrack(nav.fil.x) - sensor[n].cross;
                        da = AlongTrack(nav.fil.x) - sensor[n].along;
                        EstimatedRange = sqrt(dc*dc + da*da);
                        vector(nav.fil.h,3) = dc/EstimatedRange;
                        vector(nav.fil.h,4) = da/EstimatedRange;
                Clock(nav.fil.h) = -c;
                Cosine[2*n] = vector(nav.fil.h,3);
                Cosine[2*n+1] = vector(nav.fil.h,4);
                break;

        /*
         *  Heading sensor (actually heading error)
         */
        case HEADING:
                vector(nav.fil.h,2) = one;
                EstimatedHeading = HeadingError(nav.fil.x);
                break;

        /*
         *  Bearing sensor
         */
        case BEARING:
                dc = CrossTrack(nav.fil.x) - sensor[n].cross;
                da = AlongTrack(nav.fil.x) - sensor[n].along;
                EstimatedRange = dc*dc + da*da;
                vector(nav.fil.h,2) = -one;
                vector(nav.fil.h,3) = da/EstimatedRange;
                vector(nav.fil.h,4) = -dc/EstimatedRange;
                EstimatedRange = sqrt(EstimatedRange);
                Cosine[2*n] = vector(nav.fil.h,3);
                Cosine[2*n+1] = vector(nav.fil.h,4);
                EstimatedBearing = atan2(-dc,-da)
                                - HeadingError(nav.fil.x);
                break;

        /*
         *  This sensor was not recognized
         */
        default:
                fprintf(ErrorFile,
"NewObservation: element %d has unknown measurement type = %d\n",
                        n,sensor[n].type);
                return;
}

/*
 *    Select current sensor in H vector and update ship
 *    continuous system matrix
 */
if(SensorFlag) {
        vector(nav.fil.h, EnvStates+n) = one;

        put(nav.ship.a,
                vector(nav.fil.h,2)*sensor[n].EstAlpha,
                        n+EnvStates+1,3);
        put(nav.ship.a,
                vector(nav.fil.h,3)*sensor[n].EstAlpha,
                        n+EnvStates+1,4);
        put(nav.ship.a,
                vector(nav.fil.h,4)*sensor[n].EstAlpha,
                        n+EnvStates+1,5);
        put(nav.ship.a,
                Clock(nav.fil.h)*sensor[n].EstAlpha,
                        n+EnvStates+1,8);
      }

      /*
       *  H transpose
       */
      tpos(nav.fil.h, nav.fil.ht);
}

/********* ComputeResidual ******************************************/

/*
 *  Compute residual based on measurement type
 */
double ComputeResidual(n)
int n;            /* measurement n */
{
      double AngleSense();

      switch(sensor[n].type){

            case RANGE:
                  return( Measurement(n) -
            (SensorFlag ? SensorState(nav.fil.dx,n) : EstimatedRange)
                        );

            case HEADING:
                  return( Measurement(n)
            - (SensorFlag ? SensorState(nav.fil.dx,n) :
                                          EstimatedHeading)
                        );

            case BEARING:
                  return( Measurement(n)
            - (SensorFlag ? SensorState(nav.fil.dx,n) :
                                          EstimatedBearing)
                        );

            default:
                  fprintf(ErrorFile,
      "ComputeResidual: element %d has unknown measurement type = %d\n",
                        n,sensor[n].type);
                  return(0.0);
      }
}

/********* EstimatorSTM *********************************************/

/*
 *  Update estimator state transition matrix (phi and gamma)
 */
EstimatorSTM()
{
      int i;                  /* loop incrementors */
      double
            SinX3,            /* sin(HeadingError) */
```

```
                CosX3               /* cos(HeadingError) */
        ;

        /*
         *  Update A & B matrices.
         */
        SinX3 = sin(HeadingError(nav.fil.x));
        CosX3 = cos(HeadingError(nav.fil.x));
        put(nav.ship.a, CosX3, 4,1);
        put(nav.ship.a,
                SwayVelocity(nav.fil.x)*SinX3+nav.ship.vel*CosX3,4,3);
        put(nav.ship.a, -SinX3, 5,1);
        put(nav.ship.a, -SwayVelocity(nav.fil.x)*CosX3, 5,3);

        put(nav.ship.b, CosX3, 5,2);

        /*
         *  Update state transition matrix and conditionally call ricatti.
         */
        UpdateSTM(&nav.ship);

        tpos(nav.ship.phi, nav.ship.phit);

        /*
         *  Update nomimal ship state
         */
        copy(nav.fil.x, nav.fil.xnom);
        for(i=0; i<EnvStates; ++i) {
                vector(nav.fil.dx,i) = 0.0;
        }

        /*
         *  Save heading error, cross track position, and direction
         *  cosines indicate ship position at last STM update
         */
        LastEHE = HeadingError(nav.fil.x);
        LastECT = CrossTrack(nav.fil.x);
        for(i=0; i<8; ++i) {
                LastCosine[i] = Cosine[i];
        }

        /*
         *  Update the estimators Q matrix based on new phi
         */
        UpdateQ();
        STMStatus |= 2;
}


/********* NominalMeasurements *****************************************/

/*
 *  Compute nominal measurements for use in control input
 */
NominalMeasurements()
{
        int i;          /* loop increment */
```

```
      double
            dc,   /* cross track distance */
            da;   /* along track distance */

      for(i=0; i<SensorStates; ++i){

            switch(sensor[i].type) {

            case RANGE:
                  dc = CrossTrack(nav.fil.xnom) - sensor[i].cross;
                  da = AlongTrack(nav.fil.xnom) - sensor[i].along;
                  vector(nav.ship.u,i+2) = sqrt(dc*dc + da*da)
                                    - Clock(nav.fil.xnom)*c;
                  break;

            case HEADING:
                  vector(nav.ship.u,i+2) = HeadingError(nav.fil.xnom);
                  break;

            case BEARING:
                  dc = sensor[i].cross - CrossTrack(nav.fil.xnom);
                  da = sensor[i].along - AlongTrack(nav.fil.xnom);
                  vector(nav.ship.u,i+2) = atan2(dc,da)
                              - HeadingError(nav.fil.xnom);
                  break;

            default:
                  fprintf(ErrorFile,
  "NominalMeasurement: element %d has unknown measurement type = %d\n",
                        i,sensor[i].type);
                  return;
            }
      }
}


/********* CheckFilterStatus *****************************************/

/*
 *  Perform miscellaneous checks on times, thresholds, etc.
 */
CheckFilterStatus()
{
      int i;            /* loop incrementor */

      /*
       *  Check time step. If different, recompute phi (and q).
       *  If heading error or cross track error exceed assigned
       *  thresholds, update phi (and q).
       */
      if(fabs(DataTime - LastTime - TimeStep) > .001){
            TimeStep = DataTime - LastTime;
            EstimatorSTM();
            if(TimeStep > TimeGap)
                  RestartFilter();

      } else if(
            fabs(HeadingError(nav.fil.x) - LastEHE) >
            HeadingThreshold
        || fabs(CrossTrack(nav.fil.x) - LastECT) >
            CrossTrackThreshold) {

                EstimatorSTM();

    } else if(SensorFlag) {
        for(i=0; i<8; ++i) {
            if(fabs(Cosine[i] - LastCosine[i]) >
                CosineThreshold) {

                    EstimatorSTM();
            }
        }
    }

    /*
     *  Check if specified filter settling time has expired.
     *  If so, set threshold levels lower.
     */
    if(DataTime > SettlingTime) {
        ThresholdLevel = LOW;
    }

    /*
     *  Current bad update count is compared against limit
     *  as set in the control file. If test succeeds
     *  filter thresholds are increased.
     */
    if(BadUpdateCount >= BadUpdateLimit){
        ThresholdLevel = MIDDLE;
        BadUpdateCount = 0;
        SettlingTime = DataTime + SettlingPeriod;
    }

    if(MonitorFlag){
        if(MonitorStatus & PHI) out("nav/ship/phi:\n",nav.ship.phi);
        if(MonitorStatus & Q) out("nav/fil/q:\n",nav.fil.q);
        if(MonitorStatus & Z) out("nav/fil/z:\n",nav.fil.z);
    }

    LastTime = DataTime;
}


/********* RestartFilter ***********************************************/

/*
 *  Called when filter needs restarting due to excessive time gaps, etc.
 *  Purpose is to reinitialize critical filter elements.
 */
RestartFilter()
{
    ThresholdLevel = HIGH;
    SettlingTime = DataTime + SettlingPeriod; /* thresh switching */
                                              /* times */
    LastTime = DataTime;
```

```
# ifdef RESETP
{
      int i;

      for(i=1; i<=FilterStates; ++i)            /* reset uncertainty */
            put(nav.fil.p, pe[i-1], i,i);       /* to initial values */
}
# endif

}


/********* UpdateQ ***************************************************/

/*
 *  Recompute Q matrix.
 */
UpdateQ()
{
      int i, j;   /* loop incrementors */
      double p;   /* temporary phi */

      /*
       *  Q for ship states.
       */
      p = get(nav.ship.phi, 1,1);
      put(nav.fil.q,
            nav.ship.variance[0]/(two*nav.ship.alpha[0])*(one-p*p),
                  1,1);

      p = get(nav.ship.phi, 2,2);
      put(nav.fil.q,
            nav.ship.variance[1]/(two*nav.ship.alpha[1])*(one-p*p),
                  2,2);

      p = get(nav.ship.phi, 6,6);
      put(nav.fil.q,
            nav.ship.variance[2]/(two*nav.ship.alpha[2])*(one-p*p),
                  6,6);

      p = get(nav.ship.phi, 7,7);
      put(nav.fil.q,
            nav.ship.variance[3]/(two*nav.ship.alpha[3])*(one-p*p),
                  7,7);

      /*
       *  Clock Q
       */
      if(ClockStates) {
            p = get(nav.ship.phi, 9,9);
            put(nav.fil.q,
                nav.ship.variance[4]/(two*nav.ship.alpha[4])*(one-p*p),
                        9,9);
      }

      /*
       *  Q for sensor states
       */
      for(i=EnvStates+1,j=0; j<SensorStates; ++i,++j) {
            p = get(nav.ship.phi, i,i);
            put(nav.fil.q,
                  sensor[j].EstVar/(two*sensor[j].EstAlpha)*(one-p*p),
                        i,i);
      }
}


/********* ComputeDops **************************************************/

/*
 *  Compute VDOP and GDOP.
 */
ComputeDops()
{
      tpos(nav.fil.hdop, tmp1);
      mul(tmp1, nav.fil.hdop, tmp2);
      inv(tmp2, tmp1);
      gdop = sqrt( get(tmp1,1,1)+get(tmp1,2,2)+get(tmp1,3,3));
      vdop = sqrt(get(tmp1,4,4));
}
```

Init.c

```
# include <stdio.h>
# include <math.h>
# include "array.h"
# include "ship.h"

# include "store.x"

/***********************************************************************
 *  Functions contained in this file are:
 *
 *    InitConstants()         -     initialize scalar variables
 *    ReadControlFile         -     read parameters from control file
 *    Sizes()                 -     set arrays sizes
 *    CreateMemory()          -     allot memory for array structures
 *    InitArrays()            -     assign initial conditions to arrays
 *
 ***********************************************************************/


/********* InitConstants ***********************************************/

/*
 * Initialize constant scalars
 */
InitConstants()
{
      ShipFrompointIndex = -1;      /* start from beginning w/offset */

      one = 1.0;
      two = 2.0;

      pi = 4.0*atan(one);
      c = .29979245898e+9;                /* speed of light */
      Angle90 =  pi/two;                  /* 90 degrees in radians */
      Angle180 = pi;                      /* 180 degrees in radians */
      Angle270 = 1.5*pi;                  /* 270 degrees in radians */
      Angle360 = two*pi;                  /* 360 degrees in radians */
}


/********* ReadControlFile *********************************************/

/*
 *  Read the environment and filter parameters.
 */
ReadControlFile()
{
      int i;
      char *p;

      /*
       *  Waypoint file name
       */

      if(strlen(p = fe(ControlFile)) > MAXFNAME){
            fprintf(stderr,
            "ReadControlFile: File name %s to long.\n",p);
      finish(0);
}


if((NULL==(WaypointFile=fopen(p,"r")))){
      fprintf(stderr,
      "ReadControlFile: Can't find waypoint data file %s\n",p);
      finish(0);
}

/*
 *  Set-up important array sizes
 */
SensorChannels = atoi(fe(ControlFile));

if(SensorChannels > 5) {
      fprintf(ErrorFile,
            "Cannot have more than 5 sensor channels\n");
      finish(0);
}

Sizes();

/*
 *  Read initial states
 */
for(i=0; i<FilterStates; ++i) {
      if(i<EnvStates){
            xtrue[i] = atof((p=fe(ControlFile)));

      } else {
            LastRawMeasurement[i-EnvStates]=atof(fe(ControlFile));
      }
      xest[i] = atof(fe(ControlFile));
      pe[i] = atof(fe(ControlFile));
}

/*
 *  Skip sensor states (if flagged).
 */
if(!SensorFlag){
      for(i=0; i<SensorChannels; ++i){
            LastRawMeasurement[iShipStates]
                  = atof(fe(ControlFile));
            fe(ControlFile);
            fe(ControlFile);
      }
}


/*
 *  Type, alpha, channel variance, thresholds and noise values
 *  for each sensor
 */
for(i=0; i<SensorChannels; ++i) {
      sensor[i].type = MType(fe(ControlFile));
      sensor[i].EnvAlpha = one/atof(fe(ControlFile));
      sensor[i].EstAlpha = one/atof(fe(ControlFile));
      sensor[i].EnvVar = atof(fe(ControlFile));
      sensor[i].EstVar = atof(fe(ControlFile));
      sensor[i].threshold[HIGH] = atof(fe(ControlFile));
      sensor[i].threshold[MIDDLE] = atof(fe(ControlFile));
      sensor[i].threshold[LOW]= atof(fe(ControlFile));
      sensor[i].MNoise[HIGH] = atof(fe(ControlFile));
      sensor[i].MNoise[LOW] = atof(fe(ControlFile));
}

/*
 *  Environment and estimator time constants and variances
 *  for heading error, cross track, along track, east current
 *  north current, and clock.
 */
true.ship.variance[0] = atof(fe(ControlFile));  /* Sway var. */
nav.ship.variance[0] = atof(fe(ControlFile));

true.ship.variance[1] = atof(fe(ControlFile));  /* Yaw variance */
nav.ship.variance[1] = atof(fe(ControlFile));

true.ship.alpha[2]  = one/atof(fe(ControlFile));  /* east cur. */
nav.ship.alpha[2] = one/atof(fe(ControlFile));
true.ship.variance[2]  = atof(fe(ControlFile));
nav.ship.variance[2] = atof(fe(ControlFile));

true.ship.alpha[3]  = one/atof(fe(ControlFile));  /* north cur. */
nav.ship.alpha[3] = one/atof(fe(ControlFile));
true.ship.variance[3]  = atof(fe(ControlFile));
nav.ship.variance[3] = atof(fe(ControlFile));

true.ship.alpha[4]  = one/atof(fe(ControlFile));  /* clock */
nav.ship.alpha[4] = one/atof(fe(ControlFile));
true.ship.variance[4]  = atof(fe(ControlFile));
nav.ship.variance[4] = atof(fe(ControlFile));

/*
 *  Miscellaneous times, thresholds, and limits
 */
TimeGap = atof(fe(ControlFile));
TimeStep = atof(fe(ControlFile));
HeadingThreshold = atof(fe(ControlFile));
CrossTrackThreshold = atof(fe(ControlFile));
RicHEThreshold = atof(fe(ControlFile));
RicCTThreshold = atof(fe(ControlFile));
CosineThreshold = atof(fe(ControlFile));
WaypointThreshold = atof(fe(ControlFile));
SettlingPeriod = atof(fe(ControlFile));
BadUpdateLimit = atoi(fe(ControlFile));


/*
 *  Read sensor coordinates.
 *  Note that LandStations count is not literal in the
 *  range/bearing case.
 */
LandStations = SensorChannels - 1;
for(i=0; i<LandStations; ++i) {
```

```
             sensor[i].east  = atof(fe(ControlFile));  /* East */
             sensor[i].north = atof(fe(ControlFile));  /* North */
      }

      /*
       *  Sway velocity variance, yaw rate variance, ship length and
       *  velocity, and ship hydrodynamic constants for environment
       *  and estimator.
       */
      true.ship.length = atof(fe(ControlFile));        /* length */
      nav.ship.length = atof(fe(ControlFile));

      true.ship.vel  = atof(fe(ControlFile));          /* velocity */
      nav.ship.vel  = atof(fe(ControlFile));

      true.ship.cvu  = atof(fe(ControlFile));          /* CVU */
      nav.ship.cvu  = atof(fe(ControlFile));

      true.ship.cwu  = atof(fe(ControlFile));          /* CWU */
      nav.ship.cwu  = atof(fe(ControlFile));

      true.ship.cvdv = atof(fe(ControlFile));          /* CVDV */
      nav.ship.cvdv = atof(fe(ControlFile));

      true.ship.cvw  = atof(fe(ControlFile));          /* CVW */
      nav.ship.cvw  = atof(fe(ControlFile));

      true.ship.cwdv = atof(fe(ControlFile));          /* CWDV */
      nav.ship.cwdv = atof(fe(ControlFile));

      true.ship.cww  = atof(fe(ControlFile));          /* CWW */
      nav.ship.cww  = atof(fe(ControlFile));

      /*
       *  Performance criterion
       */
      for(i=0; i<ShipStates; ++i) {
             ControlQ[i] = atof(fe(ControlFile));
      }
      ControlR[0] = atof(fe(ControlFile));
      ControlR[1] = atof(fe(ControlFile));

      /*
       *  Determine which arrays are to viewed.
       */
      do {
             MonitorStatus |= mondecode(fe(ControlFile));
      } while(token != END);         /* must have an END! */

}

/********* Sizes *******************************************************/

/*
 *  Initialization of important array dimensions.
 */
Sizes()
```

```
{
      /*
       * Number of sensor states
       */
      if(SensorFlag){
            SensorStates = SensorChannels;
      } else {
            SensorStates=0;
      }

      ClockStates = 2;              /* Number of clock states */
      ShipStates = 7;               /* Number of ship states */
      EnvStates = ShipStates + ClockStates;     /* Number of env. */
                                                /* states */

      FilterStates =    ShipStates  /* Number of filter states */
                  + ClockStates
                  + SensorStates;

}

/******** CreateMemory ***********************************************/

/*
 *  Assign pointers to memory and dimension arrays.
 */
CreateMemory()
{

      /*
       *  Scratch arrays
       */
      dim(&tmp1, FilterStates,FilterStates, "tmp1");
      dim(&tmp2, FilterStates,FilterStates, "tmp2");
      dim(&tmp3, FilterStates,FilterStates, "tmp3");

      /*
       *  Estimator memory
       */
      dim(&nav.fil.dx, FilterStates, 1, "nav.fil.dx");
      dim(&nav.fil.xdx, FilterStates, 1, "nav.fil.xdx");
      dim(&nav.fil.xnom, FilterStates, 1, "nav.fil.xnom");
      dim(&nav.fil.x, FilterStates, 1, "nav.fil.x");
      dim(&nav.fil.z, SensorChannels, 1, "nav.fil.z");
      dim(&nav.fil.p, FilterStates, FilterStates, "nav.fil.p");
      dim(&nav.fil.q, FilterStates, FilterStates, "nav.fil.q");
      dim(&nav.fil.h, 1, FilterStates, "nav.fil.h");
      dim(&nav.fil.ht, FilterStates, 1, "nav.fil.ht");
      dim(&nav.fil.hdop, SensorChannels, 4, "nav.fil.hdop");
      dim(&nav.fil.k, FilterStates, 1, "nav.fil.k");

      dim(&nav.ship.a, FilterStates, FilterStates, "nav.ship.a");
      dim(&nav.ship.b, FilterStates, SensorStates+2, "nav.ship.b");
      dim(&nav.ship.e, FilterStates, FilterStates, "nav.ship.e");
      dim(&nav.ship.u, SensorStates+2, 1, "nav.ship.u");
      dim(&nav.ship.phi, FilterStates, FilterStates, "nav.ship.phi");
      dim(&nav.ship.phit, FilterStates, FilterStates, "nav.ship.phit");
      dim(&nav.ship.gamma, FilterStates, SensorStates+2,
                  "nav.ship.gamma");
      dim(&nav.ship.gain, 2, FilterStates, "nav.ship.gain");

      /*
       *  Environment memory
       */
      dim(&true.ship.x, EnvStates, 1, "true.ship.x");
      dim(&true.ship.u, 2, 1, "true.ship.u");
      dim(&true.ship.phi, EnvStates, EnvStates, "true.ship.phi");
      dim(&true.ship.gamma, EnvStates, 2, "true.ship.gamma");
      dim(&true.ship.gain, 2, EnvStates, "true.ship.gain");
      dim(&true.ship.a, EnvStates, EnvStates, "true.ship.a");
      dim(&true.ship.b, EnvStates, 2, "true.ship.b");
      dim(&true.ship.e, EnvStates, EnvStates, "true.ship.e");

      dim(&control.q, ShipStates, ShipStates, "control.q");
      dim(&control.r, 2, 2, "control.r");
      dim(&control.p, ShipStates, ShipStates, "control.p");
      dim(&control.phi, ShipStates, ShipStates, "contol.phi");
      dim(&control.phit, ShipStates, ShipStates, "contol.phit");
      dim(&control.gamma, ShipStates, 2, "contol.gamma");
      dim(&control.gammat, 2, ShipStates, "contol.gammat");
      dim(&control.gain, 2, ShipStates, "control.gain");
}


/********* InitArrays ************************************************/

/*
 *  Perform all vital array initializations before running regulator.
 */
InitArrays()
{
      int i;            /* loop increment */
      double v,l; /* ship velocity and length */

      /*
       *  ESTIMATOR ARRAYS
       */

      zero(nav.fil.q);

      for(i=0; i<FilterStates; ++i) {
            vector(nav.fil.x,i) = xest[i];
      }

      for(i=0; i<SensorStates; ++i) {
            SensorState(nav.fil.dx,i) = SensorState(nav.fil.x,i);
      }

      /*
       *  Initial covariance diagonal.
       */
      zero(nav.fil.p);
      for(i=1; i<=FilterStates; ++i)
            put(nav.fil.p, pe[i-1], i,i);
```

```
/*
 *  Intialize constant terms in A and B and E for later
 *  phi and gamma computations
 */
v = nav.ship.vel;
l = nav.ship.length;

zero(nav.ship.a);
put(nav.ship.a, -nav.ship.cvdv*v/l, 1,1);
put(nav.ship.a, -nav.ship.cvdv*v/l, 1,1);
put(nav.ship.a, v*(nav.ship.cvw-one), 1,2);
put(nav.ship.a, -nav.ship.cwdv*v/(l*l), 2,1);
put(nav.ship.a, -nav.ship.cww*v/l, 2,2);
put(nav.ship.a, one, 3,2);
put(nav.ship.a, -nav.ship.alpha[2], 6,6);
put(nav.ship.a, -nav.ship.alpha[3], 7,7);
put(nav.ship.a, one, 8,9);
put(nav.ship.a, -nav.ship.alpha[4], 9,9);

zero(nav.ship.b);
put(nav.ship.b, -nav.ship.cvu*v*v/l, 1,1);
put(nav.ship.b, nav.ship.cwu*v*v/(l*l),2,1);

ident(nav.ship.e);
zero(nav.ship.gain);

nav.ship.alpha[0] = -one/get(nav.ship.a,1,1);
nav.ship.alpha[1] = -one/get(nav.ship.a,2,2);

/*
 *  If sensor states are included...
 */
for(i=0; i<SensorChannels; ++i) {
      if(SensorFlag) {
            put(nav.ship.a, -sensor[i].EstAlpha,
                  i+EnvStates+1, i+EnvStates+1);
            put(nav.ship.b, sensor[i].EstAlpha,
                  i+EnvStates+1, i+3);
      }
      sensor[i].phi = exp(-TimeStep*sensor[i].EnvAlpha);
      sensor[i].gamma = one - sensor[i].phi;
      Measurement(i) = LastRawMeasurement[i];
}


/*
 *  ENVIRONMENT ARRAYS
 */

for(i=0; i<EnvStates; ++i) {
      vector(true.ship.x,i) = xtrue[i];
}

v = true.ship.vel;
l = true.ship.length;

zero(true.ship.a);
put(true.ship.a, -true.ship.cvdv*v/l, 1,1);
```

```
    put(true.ship.a, -true.ship.cvdv*v/l, 1,1);
    put(true.ship.a, v*(true.ship.cvw-1.0), 1,2);
    put(true.ship.a, -true.ship.cwdv*v/(l*l), 2,1);
    put(true.ship.a, -true.ship.cww*v/l, 2,2);
    put(true.ship.a, one, 3,2);
    put(true.ship.a, -true.ship.alpha[2], 6,6);
    put(true.ship.a, -true.ship.alpha[3], 7,7);
    put(true.ship.a, one, 8,9);
    put(true.ship.a, -true.ship.alpha[4], 9,9);

    zero(true.ship.b);
    put(true.ship.b, -true.ship.cvu*v*v/l, 1,1);
    put(true.ship.b, true.ship.cwu*v*v/(l*l),2,1);

    zero(true.ship.u);

    ident(true.ship.e);
    zero(true.ship.gain);

    zero(control.q);
    for(i=1; i<=ShipStates; ++i) {
        put(control.q, ControlQ[i-1], i,i);
    }

    zero(control.r);
    put(control.r, ControlR[0], 1,1);
    put(control.r, ControlR[1], 2,2);
}
```

Output.c

```
# include <stdio.h>
# include <math.h>
# include "array.h"
# include "ship.h"

# include "store.x"

/***********************************************************************
 *   Functions contained in this file are:
 *
 *    output()                -     print results to screen or file
 *    PrintResidual()         -     print residual with status
 *
 **********************************************************************/


/********* output *****************************************************/

output()
{
      fprintf(OutputFile,

"%.2lf\t%.2lf\t%.2lf\t%.2lf\t%.2lf\t%.2lf\t%.2lf\t%.2lf\t%.2lf\t%d\t%d\n",
            DataTime, vector(true.ship.u,0),
            HeadingError(true.ship.x),
            CrossTrack(true.ship.x),
            AlongTrack(true.ship.x),
            HeadingError(nav.fil.x),
            CrossTrack(nav.fil.x),
            AlongTrack(nav.fil.x),
            sqrt(get(nav.fil.p,4,4)),
            STMStatus, waypoint[ShipFrompointIndex].id);
}


/********* PrintResiduals *********************************************/

/*
 *  Send bad residual message to screen or file.
 */
PrintResidual(r,n,s)
double r;         /* residual value */
int n;            /* channel */
char *s;          /* comment */
{
      if(*s != ' ') {
            fprintf(ErrorFile,
                  "Bad residual = %.3lf on channel %d @ time %.2lf\n",
                        r,n,DataTime);
      }
}
```

Ship.c

```
# include <stdio.h>
# include <signal.h>
# include "array.h"
# include "ship.h"

# include "store.x"

/***********************************************************************
 *
 *  This program implements a discete time regulator.  A Kalman filter
 *  is used as an observer to provide estimated state feedback to the
 *  the plant (a ship).  Sensor measurements are created from the
 *  ship's true state and are corrupted by noise and lag.
 *
 *  States for the plant (also known as the "environment") and
 *  observer are:
 *
 *                Plant                   Observer
 *                -----                   --------
 *           1)   Sway velocity           Sway velocity
 *           2)   Yaw rate                Yaw rate
 *           3)   Heading error           Heading Error
 *           4)   Cross track position    Cross track position
 *           5)   Along track position    Along track position
 *           6)   East current            East current
 *           7)   North current           North current
 *           8)   Clock bias              Clock bias
 *           9)   Clock drift             Clock drift
 *          10)                           Sensor 1
 *          11)                           Sensor 2
 *          12)                           Sensor 3
 *          13)                           Sensor 4
 *          14)                           Sensor 5
 *
 *  Note that the observer does not always use all 5 sensors.
 *  Control input to the plant (and observer) are:
 *
 *           1) rudder
 *           2) velocity
 *
 **********************************************************************/

/***********************************************************************
 *  Functions contained in this file are:
 *
 *    main()          -           startup routine
 *    InitAll()       -           take care of all initializaions
 *
 **********************************************************************/

/********* main *******************************************************/

main(argc,argv)
int argc;
char **argv;
```

```
{
      InitAll(argc,argv);

      while(1) {
            DataTime += TimeStep;
            STMStatus = 0;                /* clear stm status */

            controller();                 /* compute control input U */
            environment();                /* update plant */

            if(EstimatorFlag){
                  SensorSimulator();      /* simulate sensors */
                  estimator();            /* estimate current state */
                  }

            CheckWaypointStatus();        /* waypoint switch needed? */
            output();                     /* print results */
      }

}


/********* InitAll **************************************************/

/*
 *  ALL initializations done here
 */

InitAll(argc,argv)
int argc;
char **argv;
{
      int finish();

      signal(SIGINT, finish); /* set up signal trap */

      cmdln(argc,argv);       /* interpret command line argument(s) */
      InitConstants();        /* initialize constant scalars */
      ReadControlFile();      /* read control file */

      CreateMemory();         /* allocate memory for filter arrays */
      InitArrays();           /* initialize filter arrays */
      ProcessMap();           /* read total map and compute constants */
      RotateShip();           /* set-up first waypoint geometry */
      RestartFilter();        /* initialize filter covariance, etc. */
      EnvironmentSTM();       /* initial regulator PHI and GAMMA */
      if(EstimatorFlag){
            EstimatorSTM();   /* initial filter PHI & GAMMA */
      }
      if(TrueFlag){           /* initialize first gain */
            riccati(&true.ship);
      } else {

            riccati(&nav.ship);
      }
      LastTime = DataTime;
      output();               /* print first point */
}
```

Ship.h

```
/*
 *  All #define's and structure templates.
 *  NOTE: array.h must be included before this file
 */

# define MAXFNAME 30    /* Maximum length of any file name */

/*
 *  Threshold levels
 */
# define    LOW   0
# define    HIGH  1
# define    MIDDLE      2

/*
 *  Bit encoded matrix descriptors for monitor function.
 */
# define K        1
# define P        2
# define PHI      4
# define Q        8
# define X        16
# define Z        64
# define RES      128

# define OFF      0
# define ON       1
# define NUMBER   2
# define END      4

/*
 *  Measurement-type flags
 */
# define    RANGE            0
# define    HEADING          1
# define    BEARING          2

/*
 *  Macros intended for readability
 */
# define    SwayVelocity(s)         vector(s,0)
# define    YawRate(s)              vector(s,1)
# define    HeadingError(s)         vector(s,2)
# define    CrossTrack(s)           vector(s,3)
# define    AlongTrack(s)           vector(s,4)
# define    Clock(s)                vector(s,7)
# define    ClockDrift(s)           vector(s,8)
# define    SensorState(s,i)        vector(s,i+9)
# define    Measurement(i)          vector(nav.fil.z,i)


/*
 *  Templates for all structures
 */
```

```
/*
 *  Template for ship observer (Kalman filter)
 */
struct _filter {
      struct array
            *dx,         /* incremental state */
            *xnom,       /* nominal value of state */
            *xdx,        /* extrapolated dx */
            *x,          /* best estimate of total ship state */
            *p,          /* covariance */
            *q,          /* state noise */
            *h,          /* observation matrix (ENA) */
            *ht,         /* its transpose */
            *z,          /* measurement vector */
            *k,          /* kalman gain pointer */
            *hdop        /* hdop/vdop utility matrix */
      ;
};

/*
 *  Template for each sensor
 */
struct _sensor {

      int
            type;              /* type of sensor (heading, etc.) */

      double
            EnvAlpha,          /* environment alpha for this sensor */
            EstAlpha,          /* estimator alpha for this sensor */
            phi,
            gamma,

            east,              /* sensor position - east */
            north,             /* sensor position - north */
            cross,             /* sensor position - cross-track */
            along,             /* sensor position - along-track */

            EnvVar,            /* environment sensor output variance */
            EstVar,            /* estimator sensor output variance */
            MNoise[2],         /* measurement noise for estimator */
            threshold[3]       /* residual thresholds for estimator */
      ;
};

/*
 *  Template for ship related material
 */
struct _ship {

      struct array
            *phi,              /* state transition matrix */
            *phit,             /* stm transpose */
            *gamma,            /* discrete control input */
            *gain,             /* Riccati gain */
            *u,                /* control input */
            *x,                /* ship state (used only in controller) */
            *a,               /* continuous system matrix */
            *b,               /* continuous-control input matrix */
            *e                /* identity matrix */
      ;

      double
            alpha[5],         /* alpha */
            variance[5],      /* state variances */
            vel,              /* ship velocity */
            length,           /* ship length */
            cvw,              /* ship hydrodynamic constants... */
            cvdv,
            cvu,
            cww,
            cwdv,
            cwu
      ;
};

/*
 *  Template for each waypoint
 */
struct _waypoint {
      double
            east,             /* east axis position of waypoint */
            north,            /* north axis position of waypoint */
            TrajectoryLength, /* distance to next waypoint */
            TurnAngle,        /* New Heading - OldHeading */
            heading,          /* absolute heading of trajectory */
            CosTA,            /* cos(TurnAngle) */
            SinTA,            /* sin(TurnAngle) */
            TanTA2,           /* tan(TurnAngle/2) */
            CosHeading,       /* cos(Heading) */
            SinHeading        /* sin(Heading) */
      ;

      int   id;               /* waypoint identification number */
};

/*
 *  Template for control related variables
 */
struct _control {
      struct array
                  *phi,       /* 7x7 phi */
                  *phit,      /* 7x7 phi transpose */
                  *gamma,     /* 7x2 gamma */
                  *gammat,    /* 2x7 gamma transpose */
                  *q,         /* 7x7 performance indice */
                  *r,         /* 2x2 performance indice */
                  *p,         /* 7x7 performance index */
                  *gain       /* 2x7 Ricatti gain */
            ;
};

/*
 *  Estimated ship state
 */
```

```
struct _nav {
      struct _ship ship;      /* filter model of ship */
      struct _filter fil;     /* basic Kalman filter structure */
};

/*
 *  True ship state
 */
struct _true{
      struct _ship ship;
};


/*
 *  Structure for each given point on the ship
 */
struct _shippoint{
      int FrompointIndex;     /* waypoint index to corner's frompoint */

      double
            cross,            /* cross-track position of point */
            along,            /* along-track position of point */
            RefCross,         /* reference cross-track position of point */
            RefAlong,         /* reference along track position of point */
            RelCross,         /* relative cross track position of point */
            RelAlong,         /* relative along track position of point */
            MinClearance      /* smallest clearance thus far */
      ;

      struct _waypoint *from; /* current ship-point frompoint */
};
```

Store.c

```
# include <stdio.h>
# include "ship.h"

/*
 *  Storage for all external variables (except most struct arrays)
 */

FILE
      *WaypointFile,         /* waypoint data file pointer */
      *ControlFile,          /* controlfile pointer */
      *OutputFile,           /* output file pointer */
      *ErrorFile             /* error file pointer */
;


int
      token,                 /* code for string identified by fe() */

      DopFlag,               /* compute dilution-of-precision */
      EstimatorFlag,         /* enable/disable estimator */
      LagFlag,               /* enable sensor lag & noise modeling */
      MonitorStatus,         /* monitor status */
      MonitorFlag,           /* monitor enable flag */
      SensorFlag,            /* enable inclusion of sensor states */
      TrueFlag,              /* true state feedback only */
      UpdateFlag,            /* enable/disable Kalman update */

      ShipFrompointIndex,    /* filter/env. index in waypoint array */
      BadUpdateCount,        /* consecutive count of bad updates */
      BadUpdateLimit,        /* max # of allowable bad updates */
      ThresholdLevel,        /* current threshold level */
      STMStatus,             /* which STM & Riccati calls occured */

      ShipStates,            /* Number of ship states */
      ClockStates,           /* Number of clock states */
      SensorStates,          /* Number of sensor states */
      SensorChannels,        /* Number of sensor channels */
      EnvStates,             /* Number of environment states */
      FilterStates,          /* Number of filter states */
      LandStations,          /* Number of range stations */
      Waypoints              /* Number of waypoints */
;

double
      DataTime,              /* Sensor channel data time tag */
      TimeStep,              /* DataTime - LastTime */
      LastTime,              /* previous DataTime */

      WaypointThreshold,     /* criterion for switching to next
waypoint */
      HeadingThreshold,      /* HeadingError change > # causes stm */
                             /* update */
      CrossTrackThreshold,   /* C-T change > # causes stm update */
      RicHEThreshold,        /* call riccati if heading error > # */
      RicCTThreshold,        /* call riccati if cross track > # */
      CosineThreshold,       /* BAngle change > # causes stm update */
      SettlingPeriod,         /* time required for filter to settle */
      SettlingTime,           /* Starting time + SettlingPeriod */
      TimeGap,                /* time gap (sec) before filter reset */

      gdop,                   /* geometric dilution of precision */
      vdop,                   /* vertical dilutution of precision */

      xtrue[14],              /* initial environment state */
      xest[14],               /* initial estimated state */
      pe[14],                 /* initial uncertainty */

      ControlQ[7],            /* optimal control indice diagonal */
      ControlR[2],            /* optimal control indice diagonal */

      EstimatedRange,         /* Est. distance from sensor to ship */
      EstimatedHeading,       /* Est. heading of ship */
      EstimatedBearing,       /* Est. bearing from ship to sensor */
      LastRawMeasurement[5],  /* Last raw measurement in sensor */
                              /* simulator */

      LastEHE,                /* Previous Estimator Heading Error */
      LastECT,                /* Previous Estimator Cross Track */
      LastPHE,                /* Previous plant Heading Error */
      LastPCT,                /* Previous plant Cross Track */
      LastCosine[8],          /* Previous direction cosine */
      Cosine[8],              /* Current direction cosine */
      SinHeading,             /* sin(Heading) ( from NextWaypoint() ) */
      CosHeading,             /* cos(Heading) ( from NextWaypoint() ) */
      Angle90,                /* pi/2 radians */
      Angle180,               /* pi radians */
      Angle270,               /* 3*pi/2 radians */
      Angle360,               /* 2*pi radians */

      one,                    /* unity */
      two,                    /* unity + unity */
      pi,                     /* 3.1415... */
      c                       /* speed of light */
;

/*
 *  Templates for the following can be found in ship.h
 */

struct _nav nav;                     /* navigation filter */
struct _waypoint waypoint[20];       /* waypoint map */
struct _sensor sensor[5];            /* sensors */
struct _true true;                   /* controller */
struct _control control;             /* control variables */
struct array *tmp1, *tmp2, *tmp3;    /* temporary arrays */
```

Store.x

```
/*
 *    Extern's for global variables found in "store.c".
 *    Corresponding comments for most of the variables
 *    can be found in "store.c".
 *
 *    NOTE: stdio.h must be included before this file
 */

extern FILE
      *WaypointFile,
      *ControlFile,
      *OutputFile,
      *ErrorFile
;

extern int
      token,

      DopFlag,
      EstimatorFlag,
      LagFlag,
      MonitorStatus,
      MonitorFlag,
      SensorFlag,
      TrueFlag,
      UpdateFlag,

      ShipFrompointIndex,
      BadUpdateCount,
      BadUpdateLimit,
      ThresholdLevel,
      STMStatus,

      ShipStates,
      ClockStates,
      SensorStates,
      SensorChannels,
      EnvStates,
      FilterStates,
      LandStations,
      Waypoints
;

extern double
      DataTime,
      TimeStep,
      LastTime,

      WaypointThreshold,
      HeadingThreshold,
      CrossTrackThreshold,
      RicHEThreshold,
      RicCTThreshold,
      CosineThreshold,
      SettlingPeriod,
      SettlingTime,
    TimeGap,

    gdop,
    vdop,
    Count[],

    xtrue[],
    xest[],
    pe[],

    ControlQ[],
    ControlR[],

    EstimatedRange,
    EstimatedHeading,
    EstimatedBearing,
    LastRawMeasurement[],
    LastEHE,
    LastECT,
    LastPHE,
    LastPCT,
    LastCosine[],
    Cosine[],
    SinHeading,
    CosHeading,
    Angle90,
    Angle180,
    Angle270,
    Angle360,

    one,
    two,
    pi,
    c
;

/*
 *  Extern declarations for templates found in "ship.h".
 */

extern struct _nav nav;
extern struct _waypoint waypoint[];
extern struct _sensor sensor[];
extern struct _true true;
extern struct _control control;
extern struct array *tmp1, *tmp2, *tmp3;

/*
 * External function declaration
 */
double noise();
char *fe();
```

Utils.c

```
# include <stdio.h>
# include <math.h>
# include <ctype.h>
# include "array.h"
# include "ship.h"

# include "store.x"

/**********************************************************************
 *   Functions contained in this file are:
 *
 *    mondecode()             -     decode user's monitor selections
 *    MType()                 -     decode sensor types
 *    cmdln()                 -     general command line arg handler
 *    help()                  -     prints help info
 *    finish()                -     end window mode and exit
 *    ACtoEN()                -     waypoint system to east-north
 *    AngleSense()            -     remove 360<->0 degree discontinuity
 *    NormalizeAngle()        -     insure angle is between 0 & 360
 *    noise()                 -     simulate noise
 *    fe()                    -     extract uncommented text
 *    decode()                -     indicate fe string is recognized
 *
 **********************************************************************/


/********* mondecode *************************************************/

/*
 *  Decode monitor selections
 */
mondecode(p)
char *p;
{
      if(!strcmp(p,"END")) return 0;
      if(!strcmp(p,"K")) return K;
      if(!strcmp(p,"P")) return P;
      if(!strcmp(p,"PHI")) return PHI;
      if(!strcmp(p,"Q")) return Q;
      if(!strcmp(p,"X")) return X;
      if(!strcmp(p,"Z")) return Z;
      if(!strcmp(p,"H")) return H;
      if(!strcmp(p,"RES")) return RES;
      fprintf(stderr,"mondecode: illegal monitor request %s.\n",p);
      return(0);
}


/********* MType ****************************************************/

/*
 *  Determine type for each filter measurement
 */
int MType(p)
char *p;
{
      if(!strcmp(p,"range")) return RANGE;
      if(!strcmp(p,"bearing")) return BEARING;
      if(!strcmp(p,"heading")) return HEADING;
      fprintf(stderr,"MType: unknown measurement type %s.\n", p);
}


/********* cmdln *****************************************************/

/*
 *  General command line arguments handler.
 */
cmdln(argc,argv)
int argc;
char *argv[];
{
      register char *name = *argv;
      int ErrorFlag;

      /*
       *  Some default settings
       */
      ErrorFile = stderr;        /* error file is stderr */
      OutputFile = stdout;       /* output is stdout */

      ErrorFlag = OFF;           /* no error file creation */
      EstimatorFlag = ON;        /* enable estimator */
      LagFlag = ON;              /* enable sensor lag & noise modeling */
      SensorFlag = ON;           /* sensor states included */
      TrueFlag = OFF;            /* allow estimator feedback to controller
*/
      UpdateFlag = ON;           /* enable Kalman updates */

      argc--, argv++;

      if(argc == 0){
            help(name);
      }

      if(**argv == '-')
            for(;;){
                  switch(*++(*argv)){
                        /*
                         * Options go here.
                         */
                        case NULL: argv++; argc--;break;
                        case 'd': DopFlag = ON; continue;
                        case 'e': ErrorFlag = ON; continue;
                        case 'f': EstimatorFlag = OFF; continue;
                        case 'l'! LagFlag = OFF; continue;
                        case 'm': MonitorFlag = ON; continue;
                        case 's': SensorFlag = OFF; continue;
                        case 't': TrueFlag = ON; continue;
                        case 'u': UpdateFlag = OFF; continue;
                        default: fprintf(stderr,
                              "Unknown option %c.",**argv);
                              help(name);
                  }
                  break;
            }

      /*
       *  Open control file.
       */
      if(argc--){
            if((NULL == (ControlFile = fopen(*argv,"r")))) {
                  fprintf(stderr, "%s: Can't open %s.\n",name,*argv);
                  finish(0);
            }
      } else
            help(name); /* No control file named */

      /*
       *  Create output file, if named.
       */
      if(argc){
            if((NULL == (OutputFile = fopen(*++argv,"w")))){
                  fprintf(stderr,
                  "%s: Can't create %s.\n",name,*argv);
                  finish(0);
            }
            if(!ErrorFlag){
                { char tmp[100];
                  sprintf(tmp,"%s.err",*argv);
                  if((ErrorFile = fopen(tmp,"w"))==NULL){
                        fprintf(stderr,"%s: Can't create %s\n",
                              name,tmp);
                        finish(0);
                  }
                }
            }
      }

      /*
       *  If no observer, feedback must be true state.
       *  Sensor states would be irrelevant.
       */
      if(!EstimatorFlag){
            TrueFlag = ON;
            SensorFlag = OFF;
      }

      /*
       *  "out()" prints to same place as OutputFile
       */
      outfp = OutputFile;
}


/********* help ******************************************************/

help(name)
char *name;
{
      fprintf(stderr,
      "\nUsage: %s -[deflmstuw] ControlFile [OutFile]\n\
      \nList of options:\n\n\
      d - evaluate DOPS\n\
      e - suppress creation of error file\n\
      f - disable estimator\n\
      l - no sensor lag & noise in sensor simulator\n\
      m - monitor filter variables as specified in control file\n\
      s - ignore sensor states\n\
      t - true state feedback only\n\
      u - no Kalman update\n\
      Options may be combined.\n\n", name);

      finish(0);
}

/********* finish ******************************************************/

/*
 *  Take care of any last business and exit.
 */
finish(code)
int code;
{
      exit(code);
}

/********* ACtoEN ******************************************************/

/*
 *  Convert along/cross track position to east-north
 */
ACtoEN(along, cross, east, north)
double
      along,            /* along track position */
      cross,            /* cross track postion */
      *east,            /* returned east position */
      *north            /* returned north position */
;
{
      *east = waypoint[ShipFrompointIndex].east
                  + along*SinHeading + cross*CosHeading;
      *north = waypoint[ShipFrompointIndex].north
                  + along*CosHeading - cross*SinHeading;
}

/********* AngleSense *************************************************/

/*
 *  Adjust VarAngle sense to coincide with RefAngle
 *  to avoid the 360<->0 discontinuity.
 */
double AngleSense(RefAngle, VarAngle)
double
      RefAngle,   /* the angle with which to compare VarAngle */
      VarAngle;   /* the angle to be normalized (if needed) */
{
      if(RefAngle>Angle270 && VarAngle<Angle90) {
            VarAngle += Angle360;

      } else if(RefAngle<Angle90 && VarAngle>Angle270) {
            VarAngle -= Angle360;

      }

      return( VarAngle );
}


/********* NormalizeAngle ********************************************/

/*
 *  Insure given angle is within 0 to 360 degree range
 */
double NormalizeAngle(angle)
double angle;
{
      if( angle < 0.0 ) {
            angle += Angle360;

      } else if( angle > Angle360 ) {
            angle -= Angle360;

      }
      return(angle);
}


/********* noise *****************************************************/

#define NORM      2147483647  /* 2^31 - 1 */

/*
 *  Simulate noise with random number generator.  Random returns a
 *  long integer between 0 and 2^31 - 1, which is normalized by NORM
 *  to a double between 0.0 and 1.0.  Subtracting by 0.5 and multiplying
 *  by sqrt(12*variance) makes the number zero mean with variance v.
 */

double noise(v)
double v;
{
      return ((((double) rand()/NORM) - (double)0.5)*sqrt(12.0*v));
}


/********* fe ********************************************************/

/*
 *  "Front End."  Fe is used by ReadControlFile to extract uncommented
 *  text from the control file.
 */
# define MAXCHRS 100

char *fe(fp)
```

```
FILE *fp;
{
      register int c,i;
      static int lineno = 1;
      static char text[MAXCHRS];

      for(;;) switch((c=getc(fp))) {
            case ';': while((c=getc(fp)) != '\n'); lineno++; break;
            case '\n':lineno++;break;
            case '\t': case ' ': break;
            case EOF: token = EOF; return;

            default: if(isalpha(c) || c =='/'){
                        i=0;
                        text[i++] = c;
                        c=getc(fp);
                        while(isalnum(c) || c == '_'
                              || c == '.' || c =='7'){
                              text[i++] = c;
                              if(i >= MAXCHRS)
            fprintf(stderr,
                  "fe: too many characters on line %d\n",lineno);
                              c = getc(fp);
                        }
                        text[i] = NULL;
                        token = decode(text);
                        return text;
                  }
                  if(isdigit(c) || c == '.' || c== '-') {
                        i = 0;
                        text[i++] = c;
                        c=getc(fp);
                        while(isdigit(c) || c == '.'
                              || c == 'e' || c == '-'){
                              if(i >= MAXCHRS)
            fprintf(stderr,
                  "fe: too many characters on lineno %d\n",lineno);
                              text[i++] = c;
                              c = getc(fp);
                        }
                        text[i] = NULL;
                        token = NUMBER;
                        return text;
                  }
                  fprintf(stderr, "fe: unknown character %x line:%d\n",
                        c&0xff,lineno);
                  break;
      }
}

/********* decode ****************************************************/

/*
 *  Set token to identify string returned by fe.
 */
decode(text)
char *text;
{
        if(!strcmp(text,"END")) return END;
}
```

Way.c

```
# include <stdio.h>
# include <math.h>
# include "array.h"
# include "ship.h"

# include "store.x"


/***********************************************************************
 *   Functions contained in this file are:
 *
 *    CheckWaypointStatus()    -     check need for waypoint switch
 *    ReadWaypoint()           -     read waypoint map
 *    ProcessMap()             -     process map
 *    RotateShip()             -     project ship onto new system
 *
***********************************************************************/


/********* CheckWaypointStatus ****************************************/

/*
 *  Check need for waypoint switch
 */

CheckWaypointStatus()
{
      double
            atst        /* Along-Track Switching Threshold */
      ;

      struct _waypoint *w;

      w = &waypoint[ShipFrompointIndex];

      if(TrueFlag) {
            atst =    w->TrajectoryLength
                  - CrossTrack(true.ship.x) * w->TanTA2
                  - WaypointThreshold;

            if(AlongTrack(true.ship.x) > atst) {
                  RotateShip();
                  EnvironmentSTM();
            }

      } else {
            atst =    w->TrajectoryLength
                  - CrossTrack(nav.fil.x) * w->TanTA2
                  - WaypointThreshold;

            if(AlongTrack(nav.fil.x) > atst) {
                  RotateShip();
                  RestartFilter();
                  EstimatorSTM();
            }
      }
}


/********* ReadWaypoint *********************************************/

/*
 *  Read data for one waypoint
 */
ReadWaypoint(w)
struct _waypoint *w;
{
      return( (int)fscanf(WaypointFile, "%d%lf%lf",
            &w->id, &w->east, &w->north)
      );

}


/********* ProcessMap ***********************************************/

/*
 *  Read in waypoint map and compute turning angle and other constants
 *  for each waypoint.
 */
ProcessMap()
{
      double
            AngleSense(),     /* returns arg2 relative to arg1 */
            de,               /* east axis difference */
            dn                /* north axis difference */
      ;

      struct _waypoint
                  *to,        /* topoint waypoint pointer */
                  *from       /* frompoint waypoint pointer */
            ;


      /*
       *  Start from zero'th waypoint
       */
      Waypoints = 0;
      from = waypoint;
      to = waypoint;

      /*
       *  Read first waypoint
       */
      if( ReadWaypoint(from) == (int)EOF ) {
            fprintf(ErrorFile,
                  "ProcessMap: incomplete or empty map file.\n");
            finish(0);
      }


      for(++Waypoints,++to; Waypoints<20; ++Waypoints,++to) {
```

```
      along = AlongTrack(nav.fil.x);
      CrossTrack(nav.fil.x) = cross * w->CosTA - along * w->SinTA;
      AlongTrack(nav.fil.x) = cross * w->SinTA + along * w->CosTA;

      /*
       *  Project controller ship position onto new trajectory
       */
      cross = CrossTrack(true.ship.x);
      along = AlongTrack(true.ship.x);
      CrossTrack(true.ship.x) = cross * w->CosTA - along * w->SinTA;
      AlongTrack(true.ship.x) = cross * w->SinTA + along * w->CosTA;

      /*
       *  Update sensor cross track/along track coordinates
       */
      for(i=0; i<LandStations; ++i) {
            de = sensor[i].east - w->east;
            dn = sensor[i].north - w->north;
            sensor[i].cross = de * w->CosHeading - dn * w->SinHeading;
            sensor[i].along = de * w->SinHeading + dn * w->CosHeading;
      }

      /*
       *  Update estimator and environment A matrix water current
       *  coordinate transformation.
       */
      put(nav.ship.a, w->CosHeading, 4,6);
      put(nav.ship.a, -w->SinHeading, 4,7);
      put(nav.ship.a, w->SinHeading, 5,6);
      put(nav.ship.a, w->CosHeading, 5,7);

      put(true.ship.a, w->CosHeading, 4,6);
      put(true.ship.a, -w->SinHeading, 4,7);
      put(true.ship.a, w->SinHeading, 5,6);
      put(true.ship.a, w->CosHeading, 5,7);
}
```

Synopsis of Matrix Library Calls

Call	Operation
GET(A1, ROW, COLUMN)	GET accesses the element pointed to by (ROW,COLUMN) from the array A1. GET returns a double. ROW/COLUMN values are assumed to start at 1.
PUT(A1, VALUE, ROW, COLUMN)	PUT places VALUE at (ROW,COLUMN) in array A1. VALUE is of type double. ROW/COLUMN values are assumed to start at 1.
ADD(A1,A2,A3)	Performs A3 = A1 + A2
SUB(A1,A2,A3)	Performs A3 = A1 - A2
COPY(A1,A2)	Copies A1 to A2
DIM(A1, ROW,COLUMN, STRING)	DIM allocates memory required for A1. ROW and COLUMN refers to the dimensions of A1; STRING is the name of the array.
IDENT(A1)	Replace contents of A1 with identity matrix
INV(A1,A2)	Performs A2 = inv(A1)
MUL(A1,A2,A3)	Performs A3 = A1 * A2
NEG(A1,A2)	Performs A2 = -A1
SCAL(A1,VALUE,A2)	Performs A2 = VALUE*A1. VALUE is of type double.
TPOS(A1,A2)	Performs A2 = transpose(A1)
ZERO(A1)	Replace contents of A1 with zeros

Expanded File and Function Listing

In the list to come each file is named along with the functions contained therein. Each function entry has the following form:

Function: Name of this function.

Purpose: What this function does.

Called By: Who calls this function.

Calls: Who this function calls.

Passed Values: List of values passed via the function's argument.

Input Variables: List of external variables whose values are used in this function.
Example: nav.fil.q to extrapolate.

Intermediate Variables: List of external variables used to store important but intermediate results.
Example: nav.fil.k to update.

Output Variables: List of external variables that contain the function's results.
Example: nav.fil.x from update.

Returned Value(s): The quantity returned through the function's name and/or quantities returned by passed-addresses.

Parts that do not apply to a particular function will be left out. Standard C functions and matrix library calls are not listed.

For a better description of external variables please see the file "store.c." For information on local variables consult the comments in the function of interest.

Often macros appear in place of array names in the source code. These were created to increase readability. For this list the array names were used in favor of the macros to more clearly indicate what external arrays are at work.

Con.c

Function: controller()

Purpose: Compute control input U = -K*X. X may be either true or estimated state.

Called By: main

Input Variables:

nav.fil.x	-	estimated state (14x1)
nav.ship.gain	-	2x14 Riccati gain
nav.ship.velocity	-	ship velocity
TrueFlag	-	true state feedback if ON
true.ship.gain	-	2x9 Riccati gain
true.ship.velocity	-	ship velocity
true.ship.x	-	true plant state (9x1)

Output Variables:

true.ship.u	-	control input (2x1)

Function: environment()

Purpose: Update the plant's state transition matrix and state.

Called By: main

Calls: EnvironmentSTM, noise

Input Variables:

CrossTrackThreshold	-	STM update threshold
HeadingThreshold	-	STM update threshold
LastPCT	-	cross track at last STM call
LastPHE	-	heading error at last STM call
true.ship.gamma	-	discrete control input
true.ship.phi	-	state transition matrix
true.ship.u	-	control input
true.ship.variance[]	-	state noise scalars
true.ship.x	-	true plant state

Intermediate Variables:

tmp1, tmp2	-	temporary arrays

Output Variables:

true.ship.x	-	true plant state

Function: SensorSimulator()

Purpose: Simulate sensor measurements for the estimator.

Sensor lags and noise are optionally included.

Input Variables:

LagFlag	-	Sensor lag if ON
LastClock	-	previous clock bias (external only to con.c)
sensor[i].along	-	sensor i along rack position
sensor[i].cross	-	sensor i cross track position
sensor[i].gamma	-	for lag equation
sensor[i].phi	-	for lag equation
sensor[i].type	-	type for sensor i
true.ship.x	-	true ship state

Output Variables

LastClock	-	save clock bias for next time
LastRawMeasurement	-	for lag computation
nav.fil.z	-	filter measurement vector

Function: EnvironmentSTM(s)

Purpose: Update environment's A and B matrices, and recalculate phi and gamma.

Called By: environment

Calls: UpdateSTM

Passed Value:

s	-	true.ship or nav.ship pointer

Input Variables:

true.ship.x	-	true ship state

Intermediate Variables:

tmp1,tmp2,tmp3	-	temporary variables

Output Variables:

LastPCT	-	Last Plant Cross Track
LastPHE	-	Last Plant Heading Error
true.ship.a	-	continuous system matrix
true.ship.b	-	continuous-control input matrix
true.ship.vel	-	ship velocity
STMStatus	-	note that this call was made

Function: UpdateSTM(s)

Purpose: Update given phi and gamma.

Called By: EstimatorSTM, EnvironmentSTM

Calls: riccati

Passed Variables:

s	-	true.ship or nav.ship pointer

Input Variables:

nav.fil.x	-	estimated ship state
ship.a	-	continuous system matrix
ship.b	-	continuous-control input matrix
ship.e	-	identity matrix
RicCTThreshold	-	riccati call threshold
RicHEThreshold	-	riccati call threshold
TimeStep	-	time increment
TrueFlag	-	ON if true state feedback
true.ship.x	-	true ship state

Intermediate Variables:

tmp1, tmp2, tmp3	-	temporary arrays
nav.ship	-	address of nav.ship struct
true.ship	-	address of true.ship struct

Output Variables:

ship.gamma	-	discrete control input
ship.phi	-	state transition matrix

Function: riccati(s) .

Purpose: Compute optimal gain via the Riccati equation.

Called By: UpdateSTM

Calls: CompressPhiGamma, ExpandGain, ninv

Passed Variables:

s	-	true.ship or nav.ship pointer

Input Variables:

control.q	-	state weight
control.r	-	control weight
ship.gamma	-	discrete control input
ship.phi	-	state transition matrix

Intermediate Variables:

tmp1,tmp2,tmp3	-	temporary arrays
control.gain	-	2x7 gain
control.gamma	-	7x2 gamma
control.gammat	-	control.gamma transpose
control.p	-	performance index
control.phi	-	7x7 phi
control.phit	-	control.phi transpose

Output Variables:

ship.gain	-	2x9 or 2x14 gain (as needed)
STMStatus	-	indicate this call was made

Function: CompressPhiGamma(s)

Purpose: Reduce given phi and gamma to 7x7 and 7x2, respectively.

Called By: riccati

Passed Value:

s	-	true.ship or nav.ship pointer

Input Variables:

ship.gamma	-	9x2 or 14x2 gamma
ship.phi	-	9x9 or 14x14 phi

Output Variables:

control.gamma	-	7x2 gamma
control.gammat	-	control.gamma transpose
control.phi	-	7x7 phi
control.phit	-	control.phi transpose

Function: ExpandGain(s)

Purpose: Expand 2x7 gain to 2x9 or 2x14.

Called By: riccati

Passed Value:

s	-	true.ship or nav.ship pointer

Input Variables:

control.gain	-	2x7 gain

Output Variables:

ship.gain	-	2x9 or 2x14 gain

Function: ninv(a,b)

Purpose: New square matrix inverse implementing Gaussian row reduction with column pivoting.

Called By: riccati

Variable Passed:

a	-	array to be inverted

Returned Variable:

b	-	inv(a) through address and argument

Est.c

Function: estimator()

Purpose: Call estimator routines in order.

Called By: main

Calls: CheckFilterStatus, extrapolate, update

Function: extrapolate()

Purpose: Extrapolate state and covariance to present time.

Called By: estimator

Calls: NominalMeasurements

Input Variables:

nav.fil.dx	-	incremental state
nav.fil.p	-	covariance
nav.fil.q	-	noise
nav.fil.xnom	-	nominal state
nav.ship.gamma	-	discrete control
nav.ship.phi	-	state transition matrix
nav.ship.phit	-	stm transpose
true.ship.u	-	control input

Intermediate Variables:

nav.fil.xdx	-	PHI*DX
nav.ship.u	-	augmented true.ship.u
tmp1,tmp2	-	temporaries

Output Variables:
nav.fil.dx
nav.fil.xnom
nav.ship.p

Function: update()

Purpose: Perform Kalman update.

Called By: estimator

Calls: NewObservation, ComputeResidual

Input Variables:

nav.fil.dx	-	incremental state
nav.fil.p	-	covariance

nav.fil.xnom	-	nominal state
sensor[i].MNoise[]	-	sensor i measurement noise
SensorChannels	-	number of sensor channels

Intermediate Variables:

nav.fil.h	-	observation vector
nav.fil.ht	-	nav.fil.h transpose
nav.fil.k	-	Kalman gain
tmp1, tmp2	-	temporary arrays

Output Variables:

BadUpdateCount	-	increment if any residual
nav.fil.dx	-	incremental state
nav.fil.x	-	estimated state
		was bad

Function: NewObservation(n)

Purpose: Compute direction cosines from sensor to ship; also compute estimated range, heading, and bearing. Elements from the observation vector are appropriately installed into estimator's A matrix.

Called By: update

Passed Value:

n	-	sensor for which observation is computed

Input Variables:

nav.fil.x	-	estimated plant state
sensor[n].along	-	sensor n cross track position
sensor[n].cross	-	sensor n cross track position
sensor[n].EstAlpha	-	sensor n alpha
sensor[n].type	-	sensor n type (range, etc.)
SensorFlag	-	ON if sensors are employed

Output Variables:

Cosine[]	-	array of direction cosines for STM update check
EstimatedBearing	-	estimated bearing to sensor
EstimatedHeading	-	estimated heading error
EstimatedRange	-	estimated range to sensor
nav.ship.a	-	estimator's A matrix
nav.fil.h	-	observation vector
nav.fil.ht	-	nav.fil.h transpose

Function: ComputeResidual(n)

Purpose: Compute residual based on measurement type.

Called By: update

Calls: AngleSense

Passed Value:

n	-	sensor for which residual is computed

Input Variables:

EstimatedBearing	-	estimated bearing to sensor
EstimatedHeading	-	estimated heading error to sen.
EstimatedRange	-	estimated range to sensor
nav.fil.z	-	sensor measurement (range, etc)
nav.fil.dx	-	sensor states in incremental estimated ship state
sensor[n].type	-	sensor n type (range, etc.)
SensorFlag	-	ON if sensors are estimated

Returned Value:

residual	-	difference

Function: EstimatorSTM()

Purpose: Update estimator state transition matrix (phi and gamma).

Called By: InitAll, CheckFilterStatus, CheckWaypointStatus

Calls: UpdateSTM, UpdateQ

Input Variables:

Cosines[]	-	present direction cosines
nav.fil.x	-	ship's estimated state
EnvStates	-	number of environment states

Output Variables:

LastCosine[]	-	direction cosines at STM call
LastECT	-	cross track at STM call
LastEHE	-	heading error at STM call
nav.fil.dx	-	(environment states cleared)
nav.fil.xnom	-	equals nav.fil.x
nav.ship.a	-	estimator's A matrix
nav.ship.b	-	estimator's B matrix
STMStatus	-	indicate this call was made

Function: NominalMeasurements()

Purpose: Compute nominal measurements for use in control input to estimator.

Called By: extrapolate

Input Variables:

nav.fil.xnom	-	nominal ship state
sensor[i].along	-	sensor i along track position
sensor[i].cross	-	sensor i cross track position
sensor[i].type	-	sensor i type (range, etc.)
SensorStates		number of sensor states

Output Variables:

nav.ship.u	-	control input to estimator (true.ship.u augmented with nominal measurements)

Function: CheckFilterStatus()

Purpose: Perform miscellaneous checks on times, thresholds, etc.

Called By: estimator

Calls: EstimatorSTM, RestartFilter

Input Variables:

Cosine[]	-	current direction cosines
DataTime	-	current time
EnvStates	-	number of environment states
LastCosine[]	-	cosines at last STM call
LastTime	-	previous DataTime
nav.fil.x	-	estimated ship's state
SensorFlag	-	ON if sensors estimated
TimeGap	-	maximum TimeStep allowed
TimeStep	-	operating time increment

Output Variables:

BadUpdateCount	-	0 if > BadUpdateLimit
LastTime	-	present DataTime becomes old
SettlingTime	-	estimator's stable DataTime
ThresholdLevel	-	estimator's threshold level
TimeStep	-	(if different)

Function: RestartFilter()

Purpose: Called when estimator requires resetting due to an excessive time gap.

Called By: CheckFilterStatus

Input Variables:

DataTime	-	current time
FilterStates	-	number of filter states
SettlingPeriod	-	length of time estimator requires to stabilize

Output Variables:

LastTime	-	present DataTime becomes old

nav.fil.p	-	initial covariance if RESETP
SettlingTime	-	estimator's stable DataTime
ThresholdLevel	-	new estimator threshold level

Function: UpdateQ()

Purpose: Recompute Q.

Called By: EstimatorSTM

Input Variables:

EnvStates	-	number of environment states
nav.ship.alpha[]	-	alpha for est. ship state
nav.ship.phi	-	estimator's phi matrix
nav.ship.variance[]	-	variance for est. ship state
sensor[i].EstAlpha	-	alpha for est. sensor state
sensor[i].EstVar	-	variance for est. sensor state
SensorStates	-	number of sensor states
ShipStates	-	number of ship states

Output Variables:

nav.fil.q	-	estimator's q matrix

Function: ComputeDops

Purpose: Compute VDOP and HDOP (not used - needs revision).

Called By: (no one)

Input Variables:

nav.fil.hdop	-	compressed nav.fil.h

Intermediate Variables:

tmp1,tmp2	-	temporary arrays

Output Variables:

gdop	-	geometric dilution of precision
vdop	-	vertical dilution of precision

Init.c

Function: InitConstants()

Purpose: Initialize scalar values used elsewhere in the program.

Called By: InitAll

Output Variables:
See source listing.

Function: ReadControlFile()

Purpose: Read user specified parameters from the control file.

Called By: InitAll

Calls: fe, Sizes

Output Variables:
Many - see source listing.

Function: Sizes()

Purpose: Initialization of important array dimensions.

Called By: ReadControlFile

Input Variables:

SensorChannels	-	number of sensor channels

Output Variables:

ClockState	-	number of clock states
EnvStates	-	" " environment states
FilterStates	-	" " filter states
SensorStates	-	" " sensor states
ShipStates	-	" " ship states

Function: CreateMemory()

Purpose: Assign pointers to memory and dimension arrays.

Called By: InitAll

Output Variables:
ALL struct arrays. See source listing.

Function: InitArrays()

Purpose: Assign initial values to newly created array structures.

Called By: InitAll

Input Variables:

ControlQ[]	-	state weight diagonal
ControlR[]	-	control weight diagonal
EnvStates	-	number of environment states
FilterStates	-	number of filter states
LastRawMeasurement[]	-	initial sensor measurements
pe[]	-	initial covariance diagonal
TimeStep	-	expect time increment
xest[]	-	initial estimator state
xtrue[]	-	initial plant state
ship hydrodynamic coefficients		

Output Variables:

control.q	-	gain state weight
control.r	-	gain control weight
nav.fil.dx	-	incremental ship state
nav.fil.p	-	covariance
nav.fil.q	-.	estimator's q matrix
nav.fil.x	-	estimated ship state
nav.fil.z	-	initial filter measurements
nav.ship.a	-	estimator's A matrix
nav.ship.b	-	estimator's B matrix
nav.ship.e	-	identity matrix (used in STM)
nav.ship.gain	-	estimator's riccati gain
sensor[i].gamma	-	gamma for sensor i
sensor[i].phi	-	state transition for sensor i
true.ship.a	-	environment's A matrix
true.ship.b	-	environment's B matrix
true.ship.e	-	identity matrix (used in STM)
true.ship.gain	-	environment's riccati gain
true.ship.x	-	initial ship true state

Output.c

Function: output()

Purpose: Send results to screen or file.

Called By: main

Function: PrintResidual(r,n,s)

Purpose: Send bad residual message to screen or file.

Called By: Update

Passed Values:

n	-	channel number
r	-	residual value
s	-	string

Input Variable:

DataTime	-	current time

Ship.c

Function: main()

Purpose: Start the program.

Called By: UNIX

Calls: controller, environment, SensorSimulator, estimator,
CheckWaypointStatus, output

Function: InitAll()

Purpose: Perform ALL initialations.

Called By: main

Calls: cmdln, InitConstants, ReadControlFile, CreateMemory,
InitArrays, ProcessMap, RotateShip, RestartFilter,
EnvironmentSTM, EstimatorSTM, riccati, output

Store.c

No calls are made in this file. All external variable storage occurs in this file.

Utils.c

Function: mondecode(p)

Purpose: Decode monitor selections given in control file.

Called By: ReadControlFile

Passed Value:

p	-	selection string pointer

Returned Value:
Code for monitor selection

Function: MType(p)

Purpose: Determine sensor type as provided in the control file.

Called By: ReadControlFile

Passed Value:

p	-	sensor type string pointer

Returned Value:
Code for sensor type

Function: cmdln(argc,argv)

Purpose: Interpret command line arguments. Arguments consist of options the user may wish to select, control file name, and an optional output file name.

Called By: InitAll

Calls: help

Passed Values:

argc	-	number of command line arguments
argv[][]	-	command line strings array

Output Variables:

ControlFile	-	control file pointer
ErrorFlag		
EstimatorFlag		
LagFlag		
outfp	-	out() file pointer
OutputFile	-	output file pointer
SensorFlag		
TrueFlag		

UpdateFlag

Function: help(name)

Purpose: Print useful message to aid running the program.

Called By: cmdln

Value Passed:

name	-	name of executable file

Function: finish(code)

Purpose: Take care of closing business and exit the program.

Passed Variable:

code	-	value to pass to exit()

Function: ACtoEN(along, cross, east, north)

Purpose: Convert along track/cross track position to east/north.

Called By: (sometimes output)

Passed Values:

along	-	ship's along track position
cross	-	ship's cross track position

Input Variables:

ShipFrompointIndex	-	index in waypoint array to ship's frompoint
waypoint[ShipFrompointIndex].east	-	frompoint's east location
waypoint[ShipFrompointIndex].north	-	frompoint's north location

Returned Values:

east	-	ship's east position
north	-	ship's north position

Function: AngleSense(RefAngle, VarAngle)

Purpose: Adjust the variable angle to coincide with the reference angle. This is to avoid 360<->0 discontinuity.

Called By: ComputeResidual, ProcessMap

Passed Values:

RefAngle	-	reference angle
VarAngle	-	variable angle

Returned Value:
Normalized variable angle.

Function: NormalizeAngle(angle)

Purpose: Insure given angle is within 0 to 360 degree range.

Called By: (nowhere at present)

Passed Values:

angle	-	given angle

Returned Value:
Angle between 0 and 360.

Function: noise(v)

Purpose: Simulate noise with a random number generator.
Noise is zero mean with variance v.

Called By: environment, SensorSimulator

Passed Value:

v	-	desired noise variance

Returned Value:
A point of noise.

Function: fe(fp)

Purpose: Extract uncommented text from a file.

Called By: ReadControlFile

Calls: decode

Passed Value:

fp	-	pointer to file

Returned Value:
Pointer to uncommented text string.

Function: decode(text)

Purpose: Set integer token to identify string returned by fe.

Called By: fe

Passed Value:

text	-	text to be identified

Returned Value:

Code identifying string.

Way.c

Function: CheckWaypointStatus()

Purpose: Check need for waypoint switch.

Called By: main

Calls: EnvironmentSTM, RotateShip, RestartFilter, EstimatorSTM

Input Variables:

nav.fil.x	-	estimated ship state
ShipFrompointIndex	-	frompoint index in wp array
true.ship.x	-	true ship state

Function: ReadWaypoint(w)

Purpose: Read data for one waypoint.

Called By: ProcessMap

Passed Value:

w	-	pointer to current waypoint array being processed.

Output Variables:
Modified waypoint structure for current waypoint.

Function: ProcessMap()

Purpose: Read in waypoint map and compute turning angle and other values for each waypoint.

Called By: InitAll

Calls: AngleSense

Output Variables:

waypoint[]	-	waypoint array
Waypoints	-	number of waypoints

Function: RotateShip()

Purpose: Project ship onto new waypoint system.

Called By: InitAll, CheckWaypointStatus

Input Variables:

nav.fil.x	-	estimated ship state

sensor[i].along	-	sensor along track position
sensor[i].cross	-	sensor cross track position
ShipFrompointIndex	-	ship's frompoint index
true.ship.x	-	true ship state
waypoint[]	-	waypoint array

Output Variables:

nav.fil.x	-	rotated nav.fil.x
nav.ship.a	-	estimator's A matrix
sensor[i].along	-	rotated sensor i along position
sensor[i].cross	-	rotated sensor i cross position
true.ship.a	-	environment's A matrix
true.ship.x	-	rotated true.ship.x

Ship Control File

```
;     Ship Control File

WPMap.1           ; waypoints file

4                 ; number of sensor channels


;     Initial states          |     Initial uncertaities
;
;     control      est          p
;     --------     ---         ----
      0.0          0.0          1.0               ; sway velocity
      0.0          0.0          3.04e-4           ; yaw rate
      0.0          0.0          3.04e-2           ; heading error
      0.0          0.0          1.0e2             ; cross track
      0.0          0.0          1.0e2             ; along track
      0.5          0.0          1.0               ; east current
      0.5          0.0          1.0               ; north current

      0.0          0.0          1.0e-12           ; clock bias
      1.0e-8       1.0e-8       1.0e-18           ; clock drift

      1.0e7        1.0e7        1.0e6             ; sensor 1
      1.0e7        1.0e7        1.0e6             ; sensor 2
      1.0e7        1.0e7        1.0e6             ; sensor 3
      0.0          0.0          3.04e-2           ; sensor 4

;     Sensor types, with time constant, channel variance, threshold
;     levels, and measurement noises per sensor.  Sensor types are
;     listed in order of occurrence.
;
;     Sensor types are:
;     range              -     range in meters
;     bearing            -     bearing in radians
;     heading            -     heading in radians
;
;
;     Time Const.s       Sensor var.     Thresholds
;     -------------      ------------      --------------------
;type    env     est     env      est     hi      mid   lo     R-hi   R-lo
;----    -----   -----   ---      ----    ----    ----- ----   ----   ----
range    0.159   0.159   25.0     25.0    1.0e4   50.0  10.0   1.0e6  1.0e0
range    0.159   0.159   25.0     25.0    1.0e4   50.0  10.0   1.0e6  1.0e0
range    0.159   0.159   25.0     25.0    1.0e4   50.0  10.0   1.0e6  1.0e0
heading  1.0     1.0     7.6e-5   7.6e-5  6.28    3.0   2.0    1.0e6  1.0e-4


;     Time Const.  Variance
;     -----------  --------
;     env     est     env      est
;     ---     ---     ---      ---
                      0.1      0.1      ; sway velocity
                      4.0e-6   4.0e-6   ; yaw rate
      10.0    10.0    0.0      0.0      ; east current
      10.0    10.0    0.0      0.0      ; north current
      100.0  100.0  1.0e-18  1.0e-18  ; clock


50.0              ; max time gap
1.0               ; estimated data timestep
.01               ; heading error difference threshold
2.0               ; cross-track error threshold
0.1               ; Riccati heading error threshold
10.0              ; Riccati cross-track threshold
0.2               ; direction cosine change threshold (radians)
150.0             ; distance to topoint to switch waypoints

10.0              ; settling period (seconds)
5                 ; number of bad updates tolerated

;     Land sensor coordinates in East/North
;     For range/sensor case repeat land coordinates.
;
;     east           north
;     -----          ------
      5000000.0      8660254.0     ; land sensor 1
      5000000.0      -8660254.0    ; land sensor 2
      -10000000.0    0.0           ; land sensor 3


;     ship hydrodynamic coefficients
;
;     env         est
;     ---         ---
      305.0       305.0       ; ship length
      5.14        5.14        ; ship velocity
      0.175       0.175       ; CVU
      1.38        1.38        ; CWU
      0.863       0.863       ; CVDV
      0.518       0.518       ; CVW
      5.25        5.25        ; CWDV
      5.32        5.32        ; CWW


;     Performance criteria

1.0 1.0 1.0 1.0 0.0 0.0 0.0   ; q matrix
1.0 1.0                       ; r matrix


;     monitor selections (K,P,PHI,Q,X,Z,H,RES)

Q
END               ; MUST HAVE THIS!!
```

Control File Description

User definable parameters are read from a control file. Description of these parameters and what they are used for forms the purpose of this document.

Interleaved with parameters in the control file are informative comments which are intended to aid the user. Comments are begun with a semicolon (;) and terminated with a newline.

Waypoint File Name
Name of file containing complete waypoint map.

Number of Sensor Channels
Number of sensor channels available to the estimator.

Initial Environment State
Initial Estimator State
Initial Covariance
Initial environment and estimator states, and initial covariance diagonal. Units for distance, angle, and time are meters, radians, and seconds, respectively; this includes rate terms. Covariance elements are in terms of units-squared. Note that the environment has no sensor states - the values shown serve as the sensor lag simulation's initial state.

Parameters/Sensor
The first entry indicates what sensor the remaining parameters describe. Available sensor types are range, bearing, and heading. The ensuing simulation will use the list of sensors specified.

Environment and estimator sensor time constants (seconds) are next. Lag in the sensors will be created given the environment's sensor time constants, while the estimator will try to model this lag with the time constants provided to it.

Environment and estimator sensor noise variances (meters-squared radians-squared) follow. Environment variances will form the disturbances added to the simulated sensor measurements, while the estimator variances are used in the Q matrix sensor elements.

The remaining five columns belong exclusively to the estimator.

High, middle, and low threshold levels (meters or radians) set reasonableness limits on filter residuals. When starting, the threshold is set high for "Settling Period" (see below) length of time. After this time the normal, low, threshold level is active. If a "Bad Update Limit" (see below) has been exceeded, the middle threshold level is set until "Settling Period" length of time has expired, at which time the low level once again becomes active.

High and low measurement noises (meters-squared or radians-squared) are used to weight filter measurements. Residuals within the threshold limit are weighted by the low noise, while residuals exceeding the threshold are deweighted with the higher noise level.

Environment and Estimator State
Time Constants and Variances
The first two columns are environment and estimator state time constants (seconds). Values found here are used in the environment's and estimator's state transition matrices, as well as the estimator's Q matrix. Entries for sway velocity and yaw rate are not present, as the "Ship Hydrodynamic Coefficients" (see below) embodies these time constants.

The last two columns are state noise variances (units identical to covariance) for environment and estimator. Environment noise variances form the disturbances added to the true ship state, while the estimator variances are used in the formation of the corresponding Q matrix elements.

Miscellaneous Parameters

Maximum Time Gap (seconds)
Greatest time between filter updates allowed before "restarting" filter. Restarting the filter involves setting the threshold high for "Settling Period" (see below) length of time.

Estimated Data Timestep (seconds)
Assumed interval between filter updates.

Heading Error Difference Threshold (radians)
A heading error change from last State Transition Matrix (STM) update to present surpassing this value will cause another STM update.

Cross Track Threshold (meters)
A cross track change from last STM update to present surpassing this value will cause another STM update.

Riccati Heading Error Threshold (radians)
A heading error greater than this value present at the time of an STM update will also cause a control gain update (calculated by A.R.E.).

Riccati Cross Track Threshold (meters)
A cross track greater than this value present at the time of an STM update will also cause a control gain update.

Direction Cosine Change Threshold (no units)
A direction cosine change from the last STM call to present exceeding this value will prompt another STM update.

Distance to Topoint (meters)
When ship's along track position is within this distance from the topoint, a waypoint switch will occur.

Settling Period (seconds)

The length of time assumed for the filter to recover from a transient, due either from startup or a restart. Filter threshold is set high during this interval, and returned to low when complete.

Number of Bad Updates Tolerated (no units)

A filter update with any number of bad residuals is considered a bad update. An unbroken string of bad updates surpassing this value will cause the filter threshold to be set at the middle level for Settling Period length of time.

Sensor Locations

East/North coordinates in meters of each sensor's land station (except heading).

Ship Hydrodynamic Coefficients

True and assumed ship length (meters), ship velocity (meters/second), and hydrodynamic coefficients.

Performance Criteria

Diagonals for optimal gain-matrices. Q and R here should not be confused with variables with similar names used in the estimator.

Filter Monitor Selections

Estimator matrices selected here may be viewed by using the -m option. The key is:

Letter	Matrix
------	------
K	Kalman gain
P	Covariance
PHI	State transition matrix
Q	State noise
X	Incremental state
Z	measurement vector
H	Observation vector
RES	Filter residual

The string END must terminate the selection list.

Appendix D

Footprint Program

1) Source Code
2) Control File

```
# include <stdio.h>
# include <math.h>
# include "ship.h"

# define ENXY
# define RLCROSS

/**********************************************************************
 *
 *    Program "foot"
 *
 *    Object:  Determine footprint of ship navigating through
 *             a pre-specified waypoint system.
 *
 *    Input: (those used by foot)
 *           time, ship's cross track, along track, and heading error,
 *           and current frompoint index.
 *
 *    Output:
 *      if RLCROSS
 *          input, acculumated along track, left extreme, right extreme
 *      if ENXY
 *          input, EN position of origon, EN position of each point
 *
 **********************************************************************/


/**********************************************************************
 *   Functions contained in this file are:
 *
 *    main()                    -     UNIX calling program
 *    InitVariables()           -     set variables to initial values
 *    ReadControlFile()         -     read data from control file
 *    NextPosition()            -     read next position from file
 *    ReadWaypoint()            -     read waypoint map
 *    ProcessMap()              -     process waypoint map
 *    CheckShipPoints()         -     check each corner for need to rotate
 *    AngleSense()              -     adjust angle sense
 *    ACtoEN()                  -     along/cross to east/north conversion
 *    Bisector()                -     position of side on bisector
 *    LRExtremes()              -     left/right extremes
 *    output()                  -     print results to screen or file
 *
 **********************************************************************/


/*
 *  Global Declarations
 */

int
      Waypoints,                /* number of waypoints in map */
      ShipPoints,               /* number of given points on ship */
      ShipFrompointIndex,       /* current ship position frompoint */
      STMStatus                 /* (not used by foot, but in output) */
;
```

```
double
      time,                   /* current time */
      ShipCross,              /* ship cross track position */
      ShipAlong,              /* ship along track position */
      ShipHeadingError,       /* ship heading error */
      CosHE,                  /* cos(ShipHeadingError) */
      SinHE,                  /* sin(ShipHeadingError) */
      pi,                     /* 3.1415... */
      Angle90,                /* 90 degrees */
      Angle270,               /* 270 degrees */
      Angle360,               /* 360 degrees */

      /*
       *  Other variables, not necessarily used by this program,
       *  but included in an augmented output file.
       */
      rudder,           /* ship's rudder position */
      EnvHE,            /* environment's heading error */
      EnvCT,            /*    "      cross track position */
      EnvAT,            /*    "      along track position */
      EstHE,            /* estimator's heading error */
      EstCT,            /*    "      cross track position */
      EstAT,            /*    "      along track position */
      EstCTSD,          /*    "      cross track standard deviation */

      SkipInterval,
      NextENTime
;

FILE
      *ControlFile,           /* control file */
      *OutputFile,            /* output file */
      *ErrorFile,             /* error message file */
      *PositionFile,          /* ship position file */
      *WaypointFile           /* waypoint map file */
;

struct _shippoint shippoint[10];    /* shippoint structure storage */

struct _waypoint
            waypoint[20],           /* waypoint structure storage */
            *LastFrompoint,         /* pointer to last frompoint in map */
            *ShipFrompoint          /* pointer to ship's frompoint */
;


/********* main *****************************************************/

main(argc,argv)
int argc;
char **argv;
{
      InitVariables();              /* initialize pi, etc */
      ReadControlFile(argc,argv);   /* get run dependent information */
      ProcessMap();                 /* read in and process waypoint map */

      while( 1 ) {
            NextPosition();         /* read next position */
            CheckShipPoints();      /* update each ship point position */
            output();               /* display results */
      }

}


/********* InitVariables **********************************************/

InitVariables()
{
      pi = 4.0*atan(1.0);
      Angle90 = pi/2.0;
      Angle270 = 1.5*pi;
      Angle360 = 2.0*pi;

      NextENTime = 0.0;

      OutputFile = stdout;    /* default file */
      ErrorFile = stderr;     /* default file */
}


/********* ReadControlFile *******************************************/

/*
 *  Read run dependent information from file.
 *  Control file name is given on the command line.
 */
ReadControlFile(argc,argv)
int argc;
char **argv;
{
      char
            *fe(),      /* return pointer to uncommented text */
            *p          /* temporary text pointer */
      ;

      /*
       *  Open control file.
       *  If not enough arguments are provided, give usage message.
       */
      if(argc<2) {
            fprintf(ErrorFile,
                  "usage: %s controlfile [outfile]\n",argv[0]);
            exit(0);
      }
      if( (ControlFile=fopen(argv[1], "r")) == NULL ) {
            fprintf(ErrorFile, "%s: can't open control file '%s'\n",
                  argv[0], argv[1]);
            exit(0);
      }

      /*
       *  Open external output file, if one is given
       *  (stdout is default).
       */
      if(argc>2) {
        if( (OutputFile=fopen(argv[2], "w")) == NULL ) {
            fprintf(ErrorFile,
                "%s: can't open output file '%s'\n",
                argv[0], argv[1]);
            exit(0);
        }
    }

    /*
     *  Open ship position file
     */
    if( (PositionFile=fopen((p=fe(ControlFile)), "r")) == NULL ) {
        fprintf(ErrorFile, "%s: can't open position file '%s'\n",
            argv[0], p);
        exit(0);
    }
    if( (WaypointFile=fopen((p=fe(ControlFile)), "r")) == NULL ) {
        fprintf(ErrorFile, "%s: can't open waypoint file '%s'\n",
            argv[0], p);
        exit(0);
    }

    /*
     *  Points to check on ship
     */
    for(ShipPoints=0; ShipPoints<4; ++ShipPoints) {
        shippoint[ShipPoints].FrompointIndex = 0;
        shippoint[ShipPoints].RelCross = atof(fe(ControlFile));
        shippoint[ShipPoints].RelAlong = atof(fe(ControlFile));
    }

    SkipInterval = atof(fe(ControlFile));
}


/********* NextPosition ***********************************************/

/*
 *  Read next ship position point.
 *  Format:  time cross track along track headingerror frompointindex
 */
NextPosition()
{
    if(fscanf(PositionFile,"%lf%lf%lf%lf%lf%lf%lf%lf%lf%d%d",
      &time,&rudder,&EnvHE,&EnvCT,&EnvAT,&EstHE,&EstCT,&EstAT,
        &EstCTSD,&STMStatus,&ShipFrompointIndex)
           == EOF) {
        fprintf(ErrorFile, "End of input.\n");
        exit(0);

    }

    /*
     *  Check if ShipFrompointIndex is beyond waypoint map.
     */
    if(ShipFrompointIndex == Waypoints) {
        fprintf(ErrorFile, "NextPosition: No more waypoints.\n");
        exit(0);
      }

      ShipFrompoint = &waypoint[ShipFrompointIndex];
      /*
       *  These points are taken as the ship's reference points
       */
      ShipHeadingError = EnvHE;
      ShipCross = EnvCT;
      ShipAlong = EnvAT;

      /*
       *  Useful values in CheckShipPoints() heading error rotation
       */
      CosHE = cos(ShipHeadingError);
      SinHE = sin(ShipHeadingError);
}


/********* ReadWaypoint **********************************************/

/*
 *  Read data for one waypoint
 */
ReadWaypoint(w)
struct _waypoint *w;
{
      return( (int)fscanf(WaypointFile, "%d%lf%lf",
            &w->id, &w->east, &w->north)
      );

}


/********* ProcessMap ************************************************/

/*
 *  Read waypoint map and compute turning angle and other constants
 *  for each waypoint.
 */
ProcessMap()
{
      int i;                  /* loop incrementor */

      double
            AngleSense(),     /* returns arg2 in same quadrant as arg1 */
            de,               /* east axis difference */
            dn                /* north axis difference */
      ;

      struct _waypoint
                  *to,        /* topoint waypoint pointer */
                  *from       /* frompoint waypoint pointer */
            ;

      struct _shippoint *s;   /* ship point utility pointer */
```

```
/*
 *  Start with first waypoint
 */
Waypoints = 1;
from = waypoint;
to = waypoint;

/*
 *  Read first waypoint
 */
if( ReadWaypoint(from) == (int)EOF ) {
      fprintf(ErrorFile,
            "ProcessMap: incomplete or empty map file.\n");
      exit(0);
}

for(++to; Waypoints<20; ++Waypoints,++to) {

      /*
       *  Read next waypoint
       */
      if(ReadWaypoint(to) == (int)EOF) {
            if(Waypoints == 1) {
                  fprintf(ErrorFile,
"ProcessMap: At least 2 are waypoints need for navigation!\n");
                  exit(0);

            } else {
                  /*
                   *  End of map - stop processing.
                   */

                  /*
                   *  Set frompoint for each ship point to
                   *  first waypoint
                   */
                  for(i=0,s=shippoint; i<ShipPoints; ++i,++s) {
                        s->from = waypoint;
                  }

                  LastFrompoint = &waypoint[Waypoints-2];

                  return;

            }

      }

      /*
       *  Distance from last waypoint to current waypoint.
       */
      de = to->east - from->east;
      dn = to->north - from->north;
      from->TrajectoryLength = sqrt(de*de + dn*dn);

      /*
       *  Absolute heading to topoint from frompoint
       */
```

```
            from->heading = atan2(de,dn) + (de>=0.0 ? 0.0 : Angle360);

            /*
             *  Heading change (turning angle) from last trajectory
             *  to current.
             *  NOTE:  AngleSense() returns an adjusted second argument
             */
            if(Waypoints == 1) {
                  from->TurnAngle = 0.0;

            } else {
                  from->TurnAngle
                        = AngleSense((from-1)->heading, from->heading)
                              - (from-1)->heading;
            }

            /*
             *  Constants for later use in waypoint switching,
             *  coordinate transformations, etc.
             */
            from->SinTA = sin(from->TurnAngle);
            from->TanTA2 = tan(from->TurnAngle/2.0);
            from->CosTA = cos(from->TurnAngle);
            from->CosHeading = cos(from->heading);
            from->SinHeading = sin(from->heading);

            /*
             *  Get ready for the next round;
             *  the topoint becomes a frompoint.
             */
            from = to;
      }
}


/********* CheckShipPoints ******************************************/

/*
 *  Update positions of all given ship points.  Then check all ship
 *  points for waypoint proximity.  If check passes, transform point
 *  to next waypoint system.
 */
CheckShipPoints()
{
      int i;                  /* Loop incrementor */

      double
            xalong,           /* spare cross track variable */
            atst              /* Along Track Switching Threshold */
      ;

      struct _waypoint *sf;   /* ship's reference point frompoint */
      struct _shippoint *s;   /* current ship point */

      /*
       *  The following loop cycles through each given ship point.
       *  In the following comments, "ship's reference point"
       *  refers to the position of the ship as provided in
 *  the position file.  "This point" refers to the
 *  current ship point being updated.
 */
for(i=0,s=shippoint; i<ShipPoints; ++i,++s) {

        /*
         *  Check this point for waypoint proximity if
         *  frompoint is not last in waypoint map.
         *  If point is across channel bisector, switch
         *  to next waypoint system.
         */
        if(s->from != LastFrompoint) {
                atst = s->from->TrajectoryLength
                                - s->cross * (s->from+1)->TanTA2;

                if(s->along > atst) {
                        ++s->FrompointIndex;
                        ++s->from;
                }
        }


        /*
         *  Update the along track/cross track position of this point.
         *
         *  Rotate the relative location for this point by the
         *  ship's heading error.  Then add to the ship's position.
         *  to find the location of this point in the reference
         *  point's waypoint system.
         */
        s->RefCross
                = ShipCross + s->RelAlong*SinHE+s->RelCross*CosHE;
        s->RefAlong
                = ShipAlong + s->RelAlong*CosHE-s->RelCross*SinHE;

        if(ShipFrompointIndex == s->FrompointIndex) {
                /*
                 *  Ship's reference point is in same waypoint system
                 *  as this point; no coordinate rotations are needed.
                 */
                s->cross = s->RefCross;
                s->along = s->RefAlong;

        } else if(ShipFrompointIndex > s->FrompointIndex) {
                /*
                 *  Ship's reference point is in front of this point,
                 *  and is separated from this point by the channel
                 *  bisector.  Therefore, rotate this point by
                 *  -TurnAngle (minus TurnAngle).
                 */
                sf = ShipFrompoint;
                s->cross
                        = s->RefAlong*sf->SinTA+s->RefCross*sf->CosTA;
                s->along
                        = s->RefAlong*sf->CosTA-s->RefCross*sf->SinTA
                                + s->from->TrajectoryLength;

        } else if(ShipFrompointIndex < s->FrompointIndex) {
```

```
                  /*
                   *  This point is ahead of ship's reference point,
                   *  and is separated from the reference point by
                   *  the channel bisector.  Therefore, rotate this
                   *  point by TurnAngle.
                   */
                  xalong = s->RefAlong - ShipFrompoint->TrajectoryLength;
                  s->cross = s->RefCross * s->from->CosTA
                              - xalong * s->from->SinTA;
                  s->along = s->RefCross * s->from->SinTA
                              + xalong * s->from->CosTA;
            }
      }
}


/********* AngleSense **************************************************/

/*
 *  Adjust VarAngle sense to coincide with RefAngle
 *  to avoid the 360<->0 discontinuity.
 */
double AngleSense(RefAngle, VarAngle)
double
      RefAngle,   /* the angle with which to compare VarAngle */
      VarAngle;   /* the angle to be normalized (if needed) */
{
      if(RefAngle>Angle270 && VarAngle<Angle90) {
            VarAngle += Angle360;

      } else if(RefAngle<Angle90 && VarAngle>Angle270) {
            VarAngle -= Angle360;

      }

      return( VarAngle );
}


/********* ACtoEN *******************************************************/

/*
 *  Convert along/cross track position to east-north
 */
ACtoEN(f, cross, along, east, north)
struct _waypoint *f;    /* frompoint structure pointer */
double
      cross,            /* given cross track */
      along,            /* given along track */
      *east,            /* returned east position */
      *north            /* returned north position */
;
{
      *east  = f->east  + along*f->SinHeading + cross*f->CosHeading;
      *north = f->north + along*f->CosHeading - cross*f->SinHeading;
}
```

```
/********* Bisector **************************************************/

/*
 *  Position of ship's side on bisector
 */
double Bisector(p1,p2)
struct _shippoint *p1, *p2;   /* ship points */
{
      double
            ma,               /* ship's side slope */
            mb,               /* slope of bisector */
            TrajLen           /* along track position of bisector */
      ;

      struct _waypoint *to, *from;

      /*
       *  Determine which waypoint system reference
       *  point is in.
       */
      if(p2->FrompointIndex > p1->FrompointIndex) {
            from = p1->from;
            to = p2->from;

      } else {
            from = p2->from;
            to = p1->from;
      }

      ma = p2->RefCross - p1->RefCross;
      if(ma == 0.0){
            return( p2->RefCross );
      }
      ma = (p2->RefAlong - p1->RefAlong)/ma;

      if(from == ShipFrompoint) {
            mb = -to->TanTA2;
            TrajLen = from->TrajectoryLength;

      } else {
            mb = to->TanTA2;
            TrajLen = 0.0;
      }

      return( (TrajLen + ma*p2->RefCross - p2->RefAlong)/(ma - mb) );
}


/********* LRExtremes ************************************************/

LRExtremes(LeftCross,RightCross,LeftIndex,RightIndex)
double
      *LeftCross,       /* leftmost ship position */
      *RightCross       /* rightmost ship position */
;
int
      *LeftIndex,       /* leftmost point ID (if any) */
      *RightIndex       /* rightmost point ID (if any) */
;
{
      int i;            /* loop incrementor */

      double
            Bisector(), /* ship's side position on bisector */
            cross       /* cross track temp */
      ;

      struct _shippoint *s;

      s = shippoint;
      *LeftIndex = 0;
      *RightIndex = 0;
      *LeftCross = s->cross;
      *RightCross = s->cross;
      for(i=1,++s; i<ShipPoints; ++i,++s) {
            cross = s->cross;

            if(cross < *LeftCross) {
                  *LeftCross = cross;
                  *LeftIndex = i;

            } else if(cross > *RightCross) {
                  *RightCross = cross;
                  *RightIndex = i;
            }

            if( (s-1)->FrompointIndex != s->FrompointIndex) {
                  cross = Bisector(s-1,s);

                  if(cross < *LeftCross) {
                        *LeftCross = cross;
                        *LeftIndex = i + ShipPoints;

                  } else if(cross > *RightCross) {
                        *RightCross = cross;
                        *RightIndex = i + ShipPoints;
                  }
            }
      }

}

/********* output ****************************************************/

output()
{
      int
            i,                /* loop incrementor */
            LeftIndex,        /* index to leftmost point */
            RightIndex        /* index to rightmost point */
      ;
      double
            Bisector(),       /* cross track position on bisector */
            LeftCross,        /* cross track of leftmost point */
            RightCross,       /* cross track of rightmost point */
            east,                  /* east position of point */
            north                  /* north position of point */
      ;

      struct _shippoint *s;


# ifdef ENXY
      if(time >=NextENTime) {
            fprintf(OutputFile,"%.21f",time);
            ACtoEN(ShipFrompoint, 0.0, ShipAlong, &east, &north);
            fprintf(OutputFile, "\t%.21f\t%.21f",east,north);
                  for(i=0,s=shippoint; i<ShipPoints; ++i,++s) {
            ACtoEN(s->from,s->cross,s->along,&east,&north);
                  fprintf(OutputFile, "\t%.21f\t%.21f",east,north);
            }

            NextENTime = time + SkipInterval;
            putc('\n', OutputFile);
      }
# endif

# ifdef     RLCROSS
      fprintf(OutputFile,"%.21f",time);
      for(i=0; i<ShipFrompointIndex; ++i) {
            ShipAlong += waypoint[i].TrajectoryLength;
      }
      fprintf(OutputFile,
      "\t%.21f\t%.21f\t%.21f\t%.21f\t%.21f\t%.21f\t%.21f\t%.21f\t%d\t%d",
            rudder,EnvHE,EnvCT,EnvAT,
            EstHE,EstCT,EstAT,EstCTSD,
            STMStatus,ShipFrompointIndex);
      LRExtremes(&LeftCross, &RightCross, &LeftIndex, &RightIndex);
      fprintf(OutputFile, "\t%.21f\t%.21f\t%.21f",
            ShipAlong,LeftCross,RightCross);
      putc('\n', OutputFile);
# endif

}
```

Control File

```
;       Program "foot" control file

run4.raw                        ; input file
WPMap.1                         ; waypoint file

;       Ship corner points

;       cross   along
;       -----   -----
        -25.0   -152.0          ; rear left
        -25.0   152.0           ; foward left
        25.0    152.0           ; rear right
        25.0    -152.0          ; forward right

        120.0                   ; skip interval for EN box plot
```

REFERENCES & BIBLIOGRAPHY

[A] GENERAL

[A-1] Greenwood, J.O. , *Greenwoods Guide to Great Lakes Shipping*, 26th. Edition, Freshwater Press, April 1985.

[A-2] MacKenzie, F.D. "Maritime Navigation/Communications Program Vol I. Navigation and Communications Systems Study", DOT/TSC report MA-RD-760-84-32, October 1984.

[A-3] MacKenzie, F.D. "Maritime Navigation/Communications Program Vol II. Requirements Definition Statement", DOT/TSC report MA602-86-10, July 1986.

[B] VESSEL CONTROL and DYNAMICS

[B-1] Amerongen, J. *et. al.* "Model reference adaptive autopilots for ships", *Automatica*, Vol. 11, 1975.

[B-2] Astrom, KJ. and C.G. Kallstrom (1976) "Identification of ship steering dynamics", *Automatica* 12, p. 9

[B-3] Bertsche, W.R. , Atkins, D.A. and Smith M.W. "Aids to navigation principal findings report on the ship variables experiment: The effect of ship characteristics and related variables on piloting performance" USCG report CG-D-55-81, November 1981.

[B-4] Carpenter, G. and Falco, M. , "Application of Optimal Control Theory to Supership Maneuvering", Gruman Data Systems CAORF Program, USMA Kings Point, NY, Report RE-557, August 1978.

[B-5] Coe, T., "Side by side buoy tender evaluation seakeeping and maneuvering comparisons of the USCG Mallow and SSP Kaimalo", CG-D-34-84, February 1984.

[B-6] Eda, H. Debord, F., and Dane, K. , Captive model tests in ice with two great lakes bulk carriers", Marad report MA-RD-940-80-81060, August 1981.

[B-7] Fujino, N. "Experimental studies on ship maneuverability in restricted waters, Part I" *International Shipbuilding Progress*, Vol 15, nr. 168 (1968).

[B-8] Fung, p. and Grimble M.J. "Dynamic Ship Positioning Using a Self-Tuning Kalman Filter", *IEEE Transactions on Automatic Control*, Vol AC-28, No.3, March 1983, pp. 339-349.

[B-9] Goclowski, J. and Gelb, A. "Dynamics of Automatic Ship Steering Systems", IEEE Trans. Control Theory, AC-11, 1966

[B-10] Grimble MJ., *et. al.*, "Use of Kalman filtering techniques in dynamics ship positioning systems," *Proc. IEE*, Vol. 127, pt. D, May 1980.

[B-11] Haruzo E., DeBord, F. and K. Dane "Captive Model Tests in Ice with Two Great Lakes Bulk Carriers", MARAD Report MA-RD-940-80-81060, August 1981.

[B-12] Kendrick, A. "Ship maneuvering characteristics", Proceedings of the Precise Navigation Workshop, Transport Canada Report TP8407E, November 1986.

[B-13] Kwakernaak, H. and R. Sivan, *Linear Optimal Control Systems",* Wiley-Interscience, *1972*.

[B-14] Linkens, D.A., "Marine systems", in *Modeling Of Dynamic Systems*, Institute of Electrical Engineers, 1980.

[B-15] Ohtsu, K. *et.al.* "A new ship's autopilot design through a stochasatic model," *Automatica*, Vol. 15, 1979

[B-16] Prado, G. , "Optimal estimation of ship's attitude and attitude rates", *IEEE transactions on Ocean Engineering*, Vol OE-4, April 1979.

[B-17] Price, W. G. and Bishop, R. E. D. *Probabalistic Theory of Ship Dynamics. London*: Chapman and Hall, 1974.

[B-18] Puglisi, J. and Schryver, et.al. "Simulation techniques and methodology in channel design", Computer Aided Operations Research Facility Report, U.S. Maritime Administration.

[B-19] Puglisi, J.J., "Simulation techniques and methodologies in channel design", MARAD computer aided operations research facility paper.

[B-20] Reid , R.E. and Mears, B.C. "The Use of Wave Filter Design in Kalman Filter State Estimation of the Automatic Steering Problem of a Tanker in a Seaway", *IEEE Transactions on Automatic Control*, Vol. AC-29, No. 7 July 1984, pp. 577-584.

[B-21] Reid, E.R. and Tugcu, A.K. "Optimal Adaptive Filtering Using Finite Banks of Kalman Filters Applied to Automatic Steering of Ships", 23 rd Conference on Decision and Control, December 1984.

[B-22] Stengel, R.F., *Stochastic Optimal Control*, Wiley and sons, 1986.

[B-23] Tiano, A. "Identification and control of the ship steering process", *Ship operation automation 2nd international symposium.* Washington (1976).

[C] VISUAL PILOT PERFORMANCE

[C-1] Bertsche, W.R. , Atkins, D.A. and Smith M.W. "Aids to navigation principal findings report on the ship variables experiment: The effect of ship characteristics and related variables on piloting performance" USCG report CG-D-55-81, November 1981.

[C-2] Bertsche, W.R. and Smith M.W. "Aids to navigation principal findings on the CAORF experiment - the performance of visual aids to navigation as evaluated by simulation" USCG report CG-D-51-81, February 1981.

[C-3] Bertsche, W.R. Mirino, K.L. and Smith M.W. "Aids to navigation principal findings report: The effect of one-sided channel marking and related conditions on piloting performance" USCG report CG-D-56-81, December 1981.

[C-4] Cooper, R.B. , Cook R.C., and Marino K.L. "At-sea data collection for the validation of piloting simulation", USCG report CG-D-60-81, December 1981.

[C-5] Feldman, D., "An approach to the study of the effect of visual aid configuration on navigation in restricted waterways", Draft USCG memo, unknown date.

[C-6] Marino, K.L., Smith, M.W., and Moynehan, J.D., "Aids to navigation SRA supplemental experiment principal findings: performance of short range aids under varied shiphandling conditions", CG-D-03-84, September 1984.

[C-7] Moynehan, J.D., and Smith, M.W. , "Aids to navigation and meeting traffic", CG-D-19-85, June 1985.

[C-8] Moynehan, J.D., and Smith, M.W., "Aids to navigation and meeting traffic", CG-D-19-85, June 1985.

[C-9] Multer, J. and Smith, M.W. "Aids to Navigation Turn Lights Principal Findings: Effects of Turn Lighting Characteristics, Buoy Arrangement, and Ship Size on Nighttime Piloting", USCG report CG-D-49-82, February 1983.

[C-10] Smith, M.W., Marino, K.L., and Multer, J., "Short range aids to navigation systems design manual for restricted waterways", CG-D-18-85, March 1985.

[D] AIDED PILOT PERFORMANCE

[D-1] Bertsche, W. R. , Cooper R.B. , Feldman, D.A. and Schroeder, K.R. "An evaluation of display formats for use with marine radio navigation piloting systems", *Journal of Institute of Navigation*, Vol. 27 No.2 pp. 116-123.

[D-2] Cooper, R.S., Marino, K.L., and Bertsche, W.R., "Simulator evaluation of electronic aids to navigation displays, the RA-2 experiment", USCG CG-D-50-81, July 1981.

[D-3] Feldman, D., "An approach to the study of electronic Dispalys for use in restricted waterways", Draft USCG memo, unknown date.

[D-4] Feldman, D., "Using Alpha-beta trackers in marine piloting by radio navigation" USCG-D-46-80, July 1980.

[D-5] Olsen, D.L. and J.R. Stoltz, "Precision LORAN-C Navigation on the St. Marys River", *Journal of Institute of Navigation*, Vol. 25, No.3, pp. 290-297.

[D-6] Puglisi,J., Bertsche, W. and Marino, K. "The performance of electronic steering displays in restricted waterway" CAORF Report

[D-7] Williams, K. "Application of precision navigator system in Sandy Hook channel under conditions of degraded aids-to -navigation" CAORF report 24-7907-01, May 1981.

[E] SENSOR PERFORMANCE, GROUND BASED

[E-1] Austin, G. , "Radar image correlation", Proceedings of the Precise Navigation Workshop, Transport Canada Report TP8407E, November 1986.

[E-2] Ball, D. , "Comparitive analysis of systems", Proceedings of the Precise Navigation Workshop, Transport Canada Report TP8407E, November 1986.

[E-3] Del-Norte Trisponder System Data Sheet, Del-Norte Technology, Euless, Texas.

[E-4] Ewen, H.I. "LORAN-C ST. Lawrence Seaway Study", DOT Contract DTRS-57-83-C-00120, December 1983.

[E-5] Haykin, S., "Polarized retro-reflectors", Proceedings of the Precise Navigation Workshop, Transport Canada Report TP8407E, November 1986.

[E-6] Hope, A. "Evaluation of a Precise Radar Navigation System (PRANS)", Transport Canada Report TP 2980, August 1981.

[E-7] Kuhn, J. , "The Incorporation of a RACON System Into a Precison Automated Navigation System - Technical Issues", Technical Memo, U.S. Department of Transportation, Transportation System Center, Cambridge MA, November 1979.

[E-8] McGillem, C.D. "Final Report LORAN-C Project", DOT Report ATC-80-25, April 1981

[E-9] Micro-Fix, Data Sheet, RACAL Positioning Systems, Surrey England, 1984.

[E-10] Miller, Wesley J., "Manual for Evaluation and Planning of Ranges", NDE Technology Inc. report prepard for U.S. Coast Guard Headquaters Office of Navigation, December 1981.

[E-11] Mini-Ranger Falcon 484 Product Sheet, Motorola, March 1983.

[E-12] Morwing, B. , "AGA-ERICON A Marine Radar Beacon", Ericsson Review No. 4, 1981.

[E-13] RAYDIST Director System Specification Bulletin R7907, Teledyne Hastings-Raydist, August 1979.

[E-14] Svala, C. , "RACONS in Theory and Practice", National Marine Electronics Association News, July-August 1984.

[E-15] Syledis Receiver Data Sheet SR3, SERCEL Industries Corp., Redmond, WA.

[E-16] Uttam. B.J. and D'Appolito, "Direct-Ranging LORAN Model Identification and Performance Predictions" IEEE Trans. on Aerospace and Electronic Systems, Vol. AES-11, No. 3, May 1978, pp. 380-385

[F] SENSOR PERFORMANCE, SPACE BASED

[F-1] Ball, D. , "Comparitive analysis of systems", Proceedings of the Precise Navigation Workshop, Transport Canada Report TP8407E, November 1986.

[F-2] Kalafus, R.M. , "Recommendations of Special Committee 104, Differential NAVSTAR/GPS Service", available from Radio Technical Commission for Marine Service (RTCM), Washington, DC.

[F-3] Pietraszewski, D. , Spalding, J. , Viehweg, C. , and Luft, L. , "Status of United States Coast Guard Sponsored Differential GPS Demonstration System Development", *Proceedings of Institute of Navigation Space Meeting*, October 1987.

[F-4] Pietraszewski, D. and Sennott, J.W., Experimental Measurement and Characterization of Ionospheric and Multipath Errors in Differential GPS, *Journal Institute of Navigation*, Vol 34, Summer 1987.

[F-5] Sennott, J.W. and Fox, T.J., Experimental Measurement and Characterization of Ionospheric and Multipath Errors in Differential GPS, USCG R&D Center Report, Prepared by Bradley University, February 1988

[F-6] Sennott, J.W. "Dynamical Errors for GPS C/A Demodulation/Navigation Processors Subject to Signal Failure and Maneuver Induced fAttenuation", DOT report DOT-TSC-RS117-PM-81-19

[F-7] Sennott, J.W. "Real-Time GPS and LORAN-C Performance for Critical Marine Applications, *Proceedings Oceans '81 Conference*, Boston MA, September 1981.

[F-8] Sennott, J.W., and Ahn, I.S., Differential GPS User and Reference Station Filter Software, and Reduction of Bay Saint Louis Data, USCG R&D Center Report, Prepared by Bradley University, May 1987.

[F-9] Yakushenkov, A. "Developement of satellite navigation systems for USSR merchant marine", Marine Research Institute Report, Leningrad, USSR.