NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PRECISION CONTROL AND MANEUVERING OF THE PHOENIX AUTONOMOUS UNDERWATER VEHICLE FOR ENTERING A RECOVERY TUBE

by

Duane T. Davis

September 1996

Thesis Advisors: Robert McGhee
Don Brutzman

Approved for public release; distribution is unlimited.

ABSTRACT

Because of range limitations imposed by speed and power supplies, covert launch and recovery of Autonomous Underwater Vehicles (AUVs) near the operating area will be required for their use in many military applications. This thesis documents the implementation of precision control and planning facilities on the *Phoenix* AUV that will be required to support recovery in a small tube and provides a preliminary study of issues involved with AUV recovery by submarines.

Implementation involves the development of low-level behaviors for sonar and vehicle control, mid-level tactics for recovery planning, and a mission-planning system for translating high-level goals into an executable mission. Sonar behaviors consist of modes for locating and tracking objects, while vehicle control behaviors include the ability to drive to and maintain a position relative to a tracked object. Finally, a mission-planning system allowing graphical specification of mission objectives and recovery parameters is implemented.

Results of underwater virtual world and in-water testing show that precise AUV control based on sonar data can be implemented to an accuracy of less than six inches and that this degree of precision is sufficient for use by higher-level tactics to plan and control recovery. Additionally, the mission-planning expert system has been shown to reduce mission planning time by approximately two thirds and results in missions with fewer logical and programming errors than manually generated missions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

A. NPS CENTER FOR AUV RESEARCH AND THE *PHOENIX* AUV

This thesis is concerned with the mission planning, mission control, and precision maneuvering required to support recovery of the *Phoenix* autonomous underwater vehicle (AUV) in a simulated torpedo tube. Specific issues covered include automated mission planning, finite state mission control, recovery path planning, recovery tube detection and localization, and precise maneuvering control for docking.

The Naval Postgraduate School (NPS) has been actively involved in autonomous underwater vehicle research for a number of years. Recently the NPS Center for AUV Research was established to explore concepts in the design and control of AUVs. As they are developed, concepts are implemented on the *Phoenix* AUV, a 236 centimeter long, neutrally buoyant vehicle weighing approximately 200 kilograms. Research goals include proving the feasibility of AUV use in shallow water mine countermeasure (MCM) operations by implementation of a working proof-of-concept system and furthering the state of the art in the field of AUVs in general. Specific research areas have included AUV control, navigation, software architecture and mission planning.

The *Phoenix* AUV (Figure 1) is controlled by two on-board computers connected via a local-area network (LAN). This LAN can be operated independently or can be connected to other networks for real-time monitoring of mission progress. Vehicle physical control is implemented using two lateral thrusters, two vertical thrusters, two aft propellers, and eight control planes.

Until recently in-water testing of *Phoenix* had been limited to the Center's 7.5 meter by 7.5 meter by 2.5 meter test tank and the sub-Olympic size NPS pool. Salt water testing began in January 1996 at Moss Landing, California. Future testing will be conducted at all three sites and preparations are in progress for open-water testing in Monterey Bay.

Figure 1: The *Phoenix* Autonomous Underwater Vehicle [Brutzman 96].

B. MOTIVATION

Counter-mine warfare has recently become an important issue in the eyes of the Navy's senior leadership [Boorda 95]. Joint doctrinal changes, especially the introduction of littoral warfare as a primary mission area, have pushed MCM operations to the forefront. Mines have many characteristics that make them attractive to coastal nations that might be the focus of littoral warfare. Mines are inexpensive, readily available, easy to use, difficult to detect and disable, and have proven very effective against naval and amphibious operations. The inadequacy of current United States MCM capabilities is amply documented [Cheney 92].

The inherently covert nature of AUVs makes them an appealing platform for shallow-water MCM operations. A small AUV launched and recovered covertly might be capable of mapping or neutralizing a mine field without being detected. This ought to be true even if the mine field is actively monitored by hostile forces.

C. PROBLEM DESCRIPTION

Since a small AUV will inevitably have a limited power supply, it will need to be launched and recovered close to its operating area. While this constraint does not pose a significant problem in civilian AUV applications, the need for covertness may preclude launching the AUV from aircraft or ships for military missions such as MCM operations. The obvious solution is to use submarines to launch and recover AUVs. Of specific interest therefore is the launch and recovery of AUVs using a submarine's torpedo tubes.

Launch of an AUV from a torpedo tube is a simple matter since launching is what torpedo tubes are designed for. Recovery is much more complex and is not a declared capability of any submarine. Recovery of an AUV via submarine torpedo tube can be

broken down into three subproblems: torpedo tube location and classification, recovery path planning, and physical control of the AUV maneuvering along the recovery path.

Torpedo tube localization and classification involves using the AUV position, the tube's expected position, and active sonar (or some other means) to precisely locate the AUV relative to the torpedo tube. [Murphy 96] uses the term *extoprioception* to describe this type of localization which involves the position of the vehicle relative to objects in the operating environment. This is in contrast to *exteroception*, which is the localization of objects in the environment relative to the AUV. The precise nature of the motion required for torpedo tube recovery dictates that estimates of AUV/tube relative positioning be continually refined while the recovery is in progress in order to ensure the AUV is safely maneuvering using the most accurate information possible.

Once the tube has been located and classified, a safe path into the tube must be determined. The AUV will attempt to travel along this path during the recovery. A smooth path must therefore be planned from the location of the AUV at the beginning of the evolution to its desired location at the end. This path may need to be periodically replanned as the position of the tube relative to the AUV is refined and updated.

The final aspect of torpedo tube recovery involves accurate movement of the AUV to a series of desired positions and orientations relative to the torpedo tube. Once the tube has been identified and a path planned, the AUV must be capable of accurately following the commanded path. Motion control must be robust, even in the presence of uniform or variable ocean currents.

D. THESIS GOALS

A large amount of research has been directed at executing MCM missions with the *Phoenix* AUV, but recovery problems have not yet been addressed in any depth. The primary goal of this thesis is to begin adapting the software architecture of the *Phoenix*

AUV to enable torpedo tube recovery. Specifically, developments to the *Phoenix* software will enable reliable recovery in a simulated torpedo tube in the Underwater Virtual World (UVW) [Brutzman 94]. UVW results are verified by in-water experiments to the greatest extent possible. Issues to be dealt with include global positioning of the recovery torpedo tube, recovery path planning, and local AUV positioning using active sonar and a mathematical model during recovery.

E. THESIS ORGANIZATION

The Rational Behavior Model (RBM) is a three layer software architecture designed to emulate the command structure of a manned submarine [Byrnes 96]. It is within the context of this architecture that this thesis is organized. This chapter is devoted to the motivation, problem discussion and goals for this project. Chapter II discusses previous work in the area of AUV recovery and related work conducted on the *Phoenix* AUV in particular. Chapter III discusses the core problems addressed by this work, the general research technique used in this project and the design of experiments. Chapters IV, V, and VI discuss implementation of features of this project at the three layers of the RBM. Specifically, Chapter IV discusses implementation at the lowest layer (execution level). Chapter V discusses implementation at the middle layer (tactical level). Chapter VI discusses implementation of the top layer (strategic level) and the off-line automatic mission generation expert system. Chapter VII focuses on the conduct and results of experiments. Conclusions and recommendations for future work are presented in Chapter VIII. The appendices contain source code, directions on how to obtain current versions of the software, and directions on how to install and use the software.

II. RELATED WORK

A. INTRODUCTION

There are several potential AUV applications in addition to MCM that are being explored by various organizations around the world. Environmental monitoring, oceanographic research and maintenance/monitoring of underwater structures are just a few examples. AUV's are attractive in these areas for a number of reasons. Because of their size and their nonreliance on human operators, they are potentially less expensive to purchase and operate than manned or remotely operated underwater vehicles. AUV's might be deployed in larger numbers, for longer periods and on shorter notice [Smith 94, Bellingham 94]. While remotely operated vehicles (ROV's) partially share these advantages, the requirement of a physical connection between the ROV and a host platform or ship limits the ROV's operating range and the required tether can be easily fouled. The latter problem can be particularly limiting in restricted environments such as kelp forests or under ice [Bellingham 94]. Given the potential applications and advantages of AUV's, it is no wonder that military, academic and commercial organizations around the world are conducting research using these vehicles.

This chapter is divided into two major parts. The first covers research efforts of other organizations that have been directed towards the recovery of AUVs. This section is by no means a complete survey of world-wide AUV research. For a broader overview of this subject, the reader is advised refer to [UUST 95, AUV 96]. The second section of this chapter describes related research conducted on *Phoenix*. In this latter section emphasis is given to the overall control architecture of *Phoenix* and the use of sonar for local-area navigation.

B. RECOVERY OF AUTONOMOUS UNDERWATER VEHICLES

1. Massachusetts Institute of Technology (MIT)

Odyssey II (Figure 2) is a robot developed by the Massachusetts Institute of Technology (MIT) Sea Grant College Program. *Odyssey II* was built for the conduct of two specific scientific missions: under-ice mapping and rapid response to volcanic events at mid-ocean ridges. *Odyssey II* is 215 centimeters in length, 59 centimeters diameter and displaces 140 kilograms. Major design goals were to minimize drag, power requirements and size while maximizing hull strength and endurance. These sometimes contradictory goals were necessary to support long missions under extreme environmental conditions. [Bellingham 94]

Figure 2: The *Odyssey II* AUV [MIT Home Page 96].

Physical control of *Odyssey II* is via a single aft-mounted thruster and four control planes mounted on the aft portion of the fuselage. The absence of lateral and vertical

thrusters means that Odyssey *II* must maintain forward motion in order to maneuver. Minimum maneuvering speed is approximately 0.5 meters per second and turn radius is approximately five meters [Bellingham 94]. Programmed vehicle behaviors must take these maneuvering characteristics into account.

Odyssey II uses three fixed sonars for obstacle detection/avoidance and an altitude sonar that can be oriented vertically to maintain altitude from the sea floor or overhead ice. A low-frequency hyperbolic long-baseline acoustic system is used for vehicle navigation during the conduct of a mission [Bellingham 92]. Mission sensors include various oceanographic instruments, a still camera and a video recorder. The primary on-board computer is a 40MHz 68030 operating under the OS-9 real-time operating system. This computer is connected to several microcontrollers that are responsible for control of some of the vehicle's subsystems. [Bellingham 94]

Logical control of *Odyssey II* uses a *layered* software system. The primary building block of the system is referred to as a *behavior*. An individual behavior is responsible for a specific type of action. Examples of behavior types include homing, collision detection, survey with navigation, and race track. The current values and priorities of all active behaviors as well as the sensor data is maintained in a vehicle state structure. This structure is evaluated by the dynamic controller which actually commands the vehicle's physical actuators. [Bellingham 94]

Recovery of *Odyssey II* relies on homing and uses a commercially available ultra-short baseline (USBL) acoustic system as a beacon. The homing behavior uses range and bearing updates from the UBSL system to guide *Odyssey II* into a capture net. The system has been successfully tested in under-ice operations with the vehicle typically returning to within 30 cm of the homing beacon [Bellingham 94]. While navigational accuracy of 30 cm is not sufficient to control an entire torpedo-tube recovery, a system such as this may be ideal for the near-field or close-proximity navigation portion of the recovery. An

acoustic navigation system providing accuracy to less than one meter might be used to position the AUV relative to the recovery tube, so that on-board AUV sensors can acquire/ classify the recovery tube and control the final portions of the recovery.

2. Florida Atlantic University

a. Ocean Voyager II

Ocean Voyager II, shown in Figure 3, is the result of a joint research effort conducted by the Ocean Engineering Department of Florida Atlantic University (FAU) and the Marine Science Department of the University of South Florida. *Ocean Voyager II* is an AUV similar in size and structure to *Odyssey II* and is intended for coastal oceanographic research. The vehicle is 240 centimeters long and displaces approximately 250 kilograms. Maximum speed is 1.54 meters per second and endurance is approximately eight hours. [Smith 94]

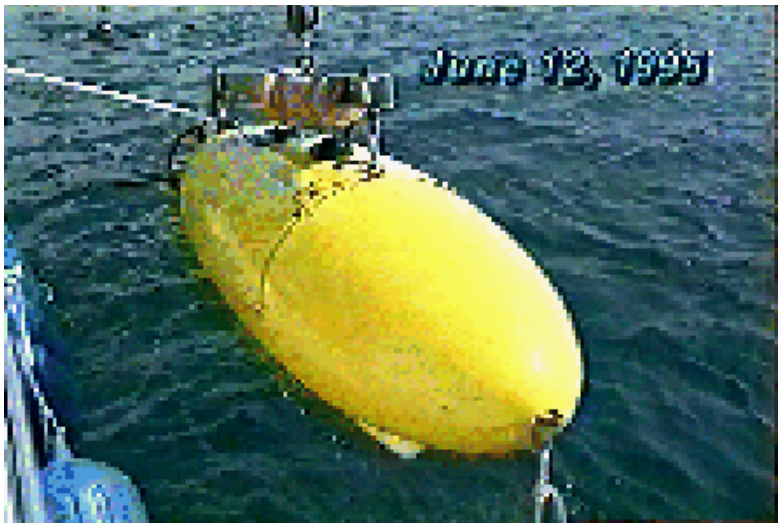


Figure 3: The *Ocean Voyager II* AUV [FAU 96].

Like *Odyssey II*, *Ocean Voyager II* uses a single aft mounted thruster and four control planes mounted on the aft portion of the fuselage. Again, this control arrangement requires *Ocean Voyager II* to maintain forward speed to maintain posture

control. This constraint is not a problem given the type of mission for which the vehicle is intended. While *Odyssey II* is designed for deep-water operations, the missions for which *Ocean Voyager II* is intended require the vehicle to cruise in a regular pattern at a fixed altitude above the bottom [White 96, Smith 94]. Specific missions include monitoring sea grass, monitoring macro-algae beds and evaluating the effects of storm-front passage [Smith 94].

While the missions for which *Ocean Voyager II* is designed are fairly specific in nature, each mission requires different sensor packages. The sensor payload is contained immediately aft of the AUV's nose cone and is designed to be modular in nature. This modularity allows for fairly simple but specialized sensor packages installed for each mission [White 96].

Logical control of *Ocean Voyager II* is implemented by a fuzzy rule-based algorithm. The control algorithm is similar to that of *Odyssey II* except that instead of behaviors, *Ocean Voyager II* control modes use the results of fuzzy rules to compute control commands and confidence levels. The output of each mode is evaluated by the fuzzy weighted decision arbiter which determines the actual control outputs. Abort and avoid modes provide for vehicle safety. Track, stable and no-operation modes provide for data collection during normal operation. Additional modes for waypoint navigation, docking or other behaviors are possible but have not yet been tested. [Smith 96]

b. Recovery of Ocean Voyager II

Significant simulation-based research in the area of AUV recovery has been conducted using *Ocean Voyager II*. In [Rae 92, Rae 93] the possibility of using fuzzy logic to control recovery by a submarine was explored, while [White 96] documents more recent research into using the same general procedure to control docking with a fixed structure.

The fuzzy docking algorithm uses a “virtual funnel” to control the AUV towards the goal. The virtual funnel is represented by fuzzy rules that define desired motion for the AUV given its current position. As long as the vehicle remains inside the region defined by the virtual funnel, it will proceed towards the docking target. If the AUV wanders outside the funnel, it will be vectored towards a new starting position and will begin again. The size and shape of the docking funnel are determined by vehicle characteristics and the external environment. A strength of the fuzzy docking algorithm is that obstacle avoidance is an integral consideration. A flow chart representation of the fuzzy docking algorithm is depicted in Figure 4. [Rae 92, Rae 93, White 96]

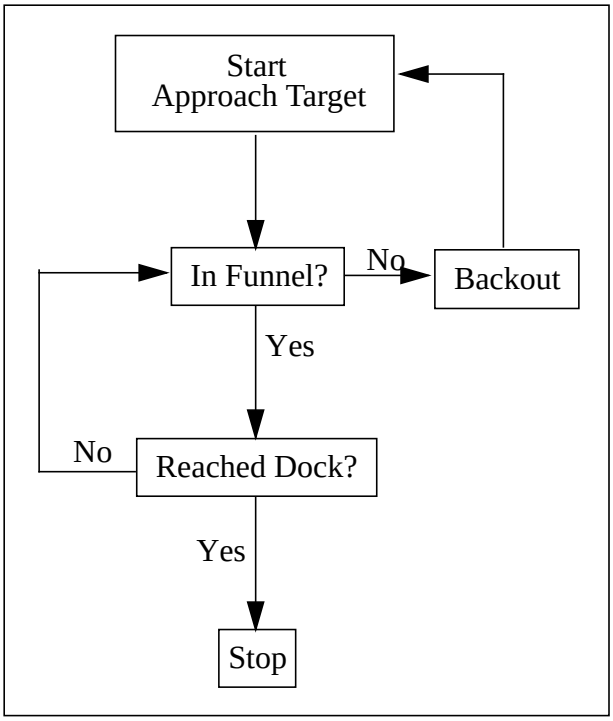


Figure 4: Fuzzy Docking Algorithm [Smith 96].

[Rae 92, Rae 93] assume that the AUV has accurate relative position information for itself and the dock throughout the docking procedure. While this assumption precluded immediate implementation in the vehicle, tests documented in these

papers indicated that the algorithm was valid so long as accurate navigational data was obtained. Specifically [Rae 92] documents simulation results of fuzzy docking algorithm use to dock with a stationary submarine. [Rae 93] expands on this work by simulating the use of the algorithm to dock with a moving submarine and also attempts to model suction forces and wake turbulence created by a moving submarine. [White 96] documents simulation results that indicate that if a boundary area is included in the virtual funnel to account for navigational inaccuracy as depicted in Figure 5, the algorithm is still valid. Simulation results documented in [White 96] were based on expected navigational accuracy using the DivetrackerTM system. An interesting additional result was that the navigational accuracy of the DivetrackerTM system may be sufficient to control the entire docking maneuver.

3. Shenyang Research and Development Centre of Robotics

The Shenyang Research and Development Centre of Robotics is a research organization in the People's Republic of China. The AUV being developed by this group is named *Explorer*. While *Explorer* is similar to the *Odyssey II* and *Ocean Voyager II* in operating capabilities and characteristics, it is interesting and relevant in this context because of its recovery device. Although *Explorer* is operated from a surface ship, launch and recovery takes place underwater.

Four options were considered for *Explorer*'s launch and recovery system: recovery in the center well of a support ship, recovery using a submarine, recovery using a semi-submersible platform, and recovery using a submersible platform. The final system uses a submersible cage that is lowered by a crane on the support ship. The decision to use an underwater launch and recovery procedure was based on two factors: the difficulty of a surface recovery in high sea states and *Explorer*'s relatively poor navigational capabilities on the surface. [Ditang 92]

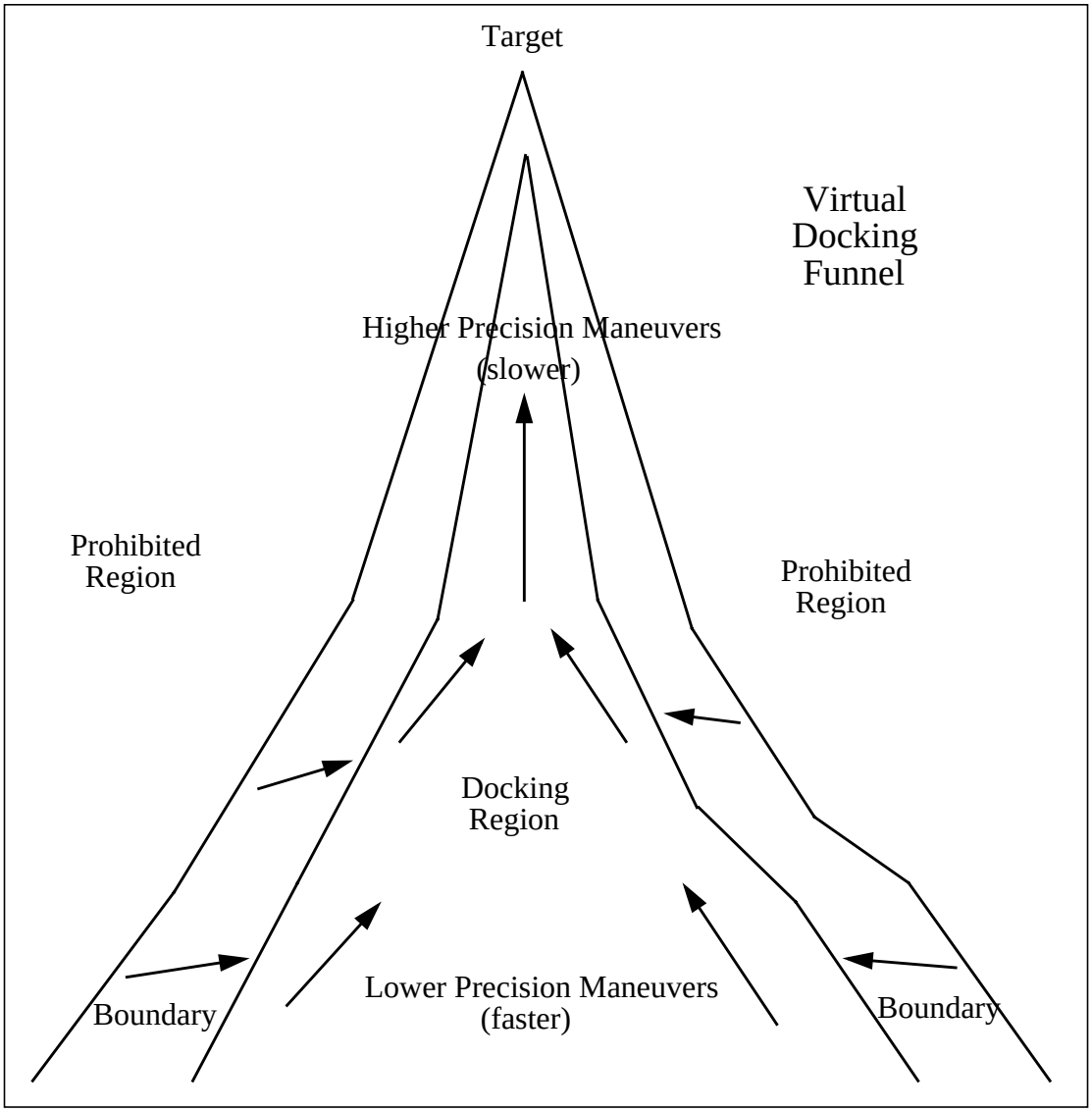


Figure 5: Virtual Docking Funnel for the Fuzzy Docking Algorithm [Smith 96].

The launcher itself is a cage-like structure that is lowered by crane to a depth of 30 to 50 meters. The launcher has two locking arms for securing the AUV when it is in the launcher, a television camera for monitoring of the recovery and two vertical thrusters which are used to maintain the launcher at the specified depth. *Explorer* uses an ultrashort baseline (USBL) navigation system and an on-board video camera to navigate during the recovery process. A drawing of the launcher configuration is depicted in Figure 6. [Ditang 92]

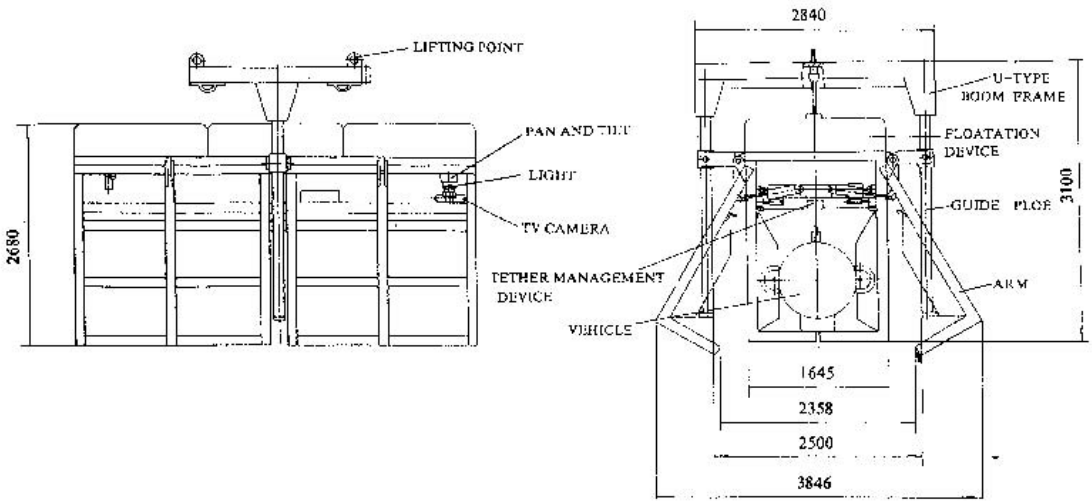


Figure 6: The *Explorer* AUV Launcher (units are mm) [Ditang 92].

The *Explorer* recovery process consists of five steps. First the launcher is lowered to the appropriate depth. Once lowered to the specified depth, the launcher thrusters automatically maintain the launcher's depth so that motion control by the ship-board operator is not required. Next *Explorer* uses the USBL system to navigate to a position in front of the launcher. Once within visual range, the on-board video camera is used to identify reference points on the launcher and provide precise relative position information during the final phase of the recovery. The launcher operator uses the launcher's camera to determine when the AUV is in the final recovery position. Once the AUV is in place the

locking arms are closed, and both launcher and AUV can be raised into the ship. A depiction of an *Explorer* recovery using the launcher can be seen in Figure 7. [Ditang 92]

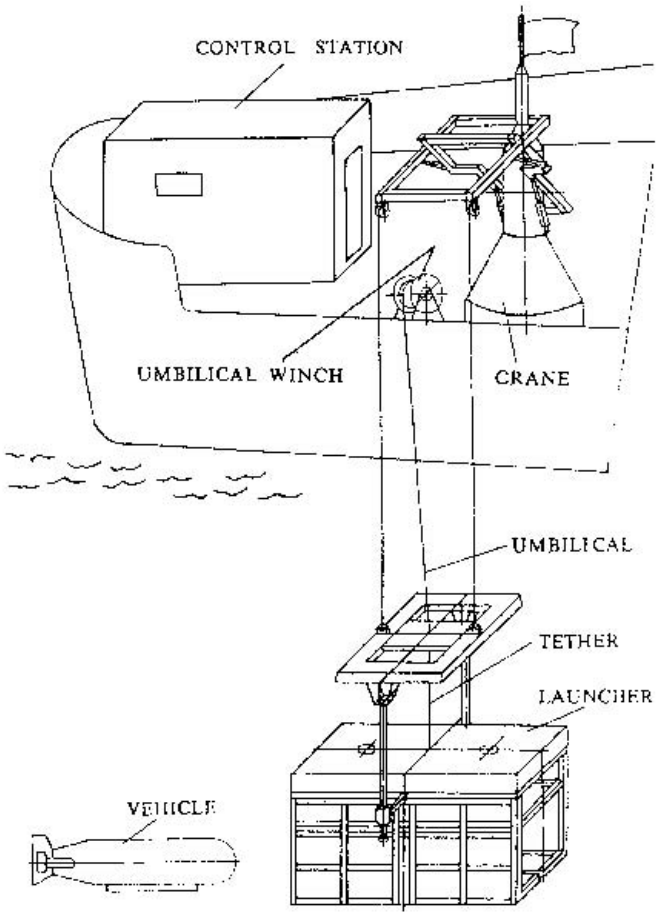


Figure 7: An *Explorer* AUV Recovery [Ditang 92].

While recovery of an AUV in this fashion is a far cry from recovery within a torpedo tube, it is noteworthy in two respects. First, this system requires close coordination between the recovering ship and the AUV. If the launcher is not at the appropriate depth or location, or if the locking arms are not operated properly, the recovery will not be successful. This coordination between the AUV and its recovery vehicle is a basic assumption upon which successful recovery is based and is discussed in more detail in [Gwin 92] and [Chapuis 96]. Second, *Explorer* uses multiple navigation techniques during

different phases of the recovery. The on-board television camera provides accurate local navigation during the final phase of the recovery, but video is of no use in locating the launcher from a distance of more than a few feet. The USBL navigation system allows *Explorer* to get close to the launcher, but does not provide enough precision to actually enter the launcher.

The recovery procedure being considered for *Phoenix* is similar, using the DivetrackerTM system to navigate to a position from which the recovery tube can be acquired and identified using the two on-board sonar systems. The *Phoenix* sonars are then used for precision maneuvering into the tube using techniques described in [Healey 94] and [Marco 96a].

4. Centre Technique Des Systemes Navals (CTSN)

As part of a larger feasibility study on the design of recoverable unmanned underwater vehicles (UUV's), the Centre Technique Des Systemes Navals (CTSN), located in Toulon France, has attempted to identify functions upon which UUV launch and recovery from submarines rely and the environmental factors affecting each of these functions [Chapuis 96]. For the most part the functions and environmental factors identified are relevant to the launch and recovery of both AUV's and ROV's. Functions are divided into two types: main functions and constraint functions. Main functions are those functions that directly accomplish high level goals. Constraint functions are those that facilitate the successful completion of main functions or are inherent subfunctions of a main function.

For UUV recovery, one main function and six constraint functions were identified. These functions and the factors effecting them are shown in Table 1. The main function is simply the transition of the UUV from the open sea into the submarine. Constraint functions include communication with the submarine, entry into the recovery system,

straight navigation despite swell and waves, adapting to depth effects such as pressure and light level, obstacle avoidance, and resistance to the marine environment. By performing this assessment process repeatedly, the problem requirements of torpedo-tube docking with a submarine are fully specified. [Chapuis 96]

Function	**Criteria**
Transition from open sea into submarine	vehicle speed vehicle path
Communicate with submarine	Communication system type attenuation intensity frequency band range
Enter recovery system	vehicle path vehicle speed vehicle size recovery device size recovery device sensors
Navigate straight in the presence of waves and current	wave significant height wave period wave direction current speed current direction
Adapt to depth effects	depth temperature
Avoid obstacles	obstacle density obstacle speed obstacle direction distance obstacle/vehicle
Resist the marine environment	acidity

Table 1. UUV Recovery Functions. Underlined criteria are considered dominant. After [Chapuis 96]

5. Institute for Systems and Robotics, Instituto Superior Tecnico

Still another AUV research project is being conducted by the Institute for Systems and Robotics Instituto Superior Tecnico (IST) of Lisbon, Portugal. The research vehicle of this organization is named *Marius*. However this research is relevant in the context of this thesis (specifically Chapter VI) because of the mission control/planning features rather than the vehicle itself.

High level mission control of Marius uses a mathematical structure called a *Petri net* [Cassandras 93, Peterson 81]. A Petri net is a type of graph consisting of *transitions*, *places* and *arcs*. When used for AUV mission control, transitions correspond to actions to be undertaken by the vehicle, places correspond to preconditions for execution of a transition or results of transition execution, and arcs are used to connect transitions to the appropriate precondition and postcondition places. A *token* is used to mark all places whose conditions are satisfied. When all of a transition's precondition places contain tokens, the transition is enabled. Since multiple transitions may be enabled at the same time, Petri nets are well suited to representing parallelism in a system.

The CORAL development environment has been developed by IST as the interface for generating missions. This system uses a graphical interface to define the Petri net representing a mission, and assign specific tasks to the transitions. A CORAL Engine has also been developed to accept and execute Petri net descriptions. Details of the CORAL system can be found in [Oliveira 96].

Recent research has been conducted to use the CORAL system for defining and executing missions with other AUVs. Towards this end, a mission was successfully executed by the *Phoenix* AUV using CORAL without making any modification to *Phoenix* software [Healey 96]. The results of this research are an indication of the general equivalence of several AUV multiple-level mission-control strategies.

C. THE *PHOENIX* AUTONOMOUS UNDERWATER VEHICLE

1. Hardware Configuration

The *Phoenix* AUV is 235 centimeters in length, 41 centimeters in width, 25 centimeters in height and displaces 198 kilograms [Leonhardt 96]. The main body, which houses *Phoenix*' electronic and power equipment, is constructed of aluminum and is designed to be water tight to eight meter depth. The free-flood nose cone is constructed of fiberglass and houses the vehicle's sonars, depth sensor and waterspeed probe. Physical control of *Phoenix* is via two aft thrusters, two lateral cross-body thrusters, two vertical cross-body thrusters, and eight control planes. The rectangular hull form and large number of propulsion effectors are intended to facilitate precise position and orientation control whether the vehicle is hovering or transiting. The external layout of *Phoenix* is depicted in Figure 8.

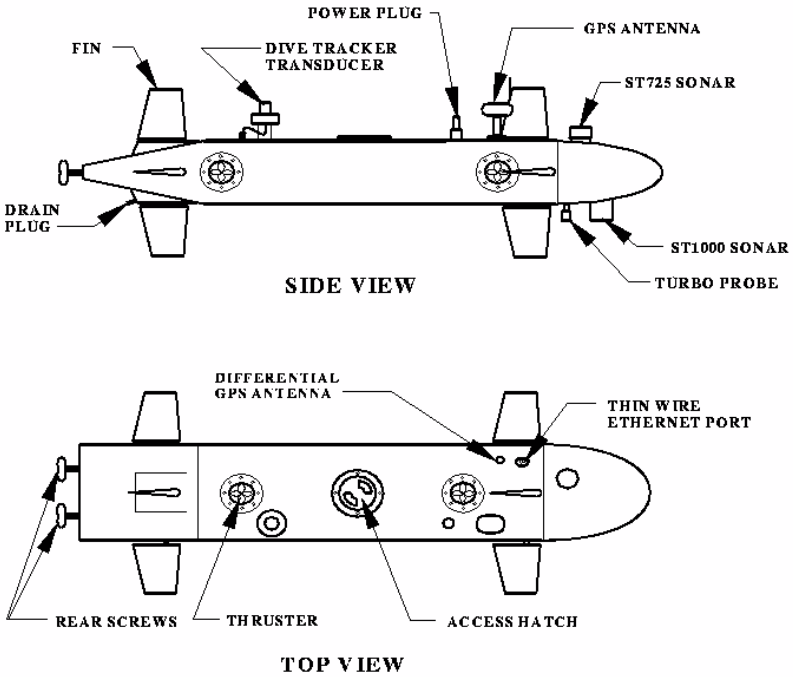


Figure 8: *Phoenix* External Configuration [Leonhardt 96].

Phoenix is controlled by two on-board computers. The vehicle's actuators and sensors are monitored and controlled by processes running on a 30 MHz Gespac 68030 computer under the OS-9 real-time operating system. Higher-level mission control, data collection and planning are handled by processes running on a Sun Voyager workstation under the Unix operating system. The two computers are connected by an on-board Ethernet local-area network (LAN). The vehicle also has an external Ethernet connector which can be used to communicate with the on-board computers from an external network. This external connection is primarily used for mission loading and data retrieval and is simply terminated during untethered missions.

Phoenix' primary navigational equipment consists of a differential Global Positioning System (GPS) receiver and a Divetracker™ short baseline acoustic tracking system. *Phoenix*' use of these systems is covered in detail in [McClarin 96] and [Scrivener96]. In addition, *Phoenix* has a turbine flow-meter probe for water speed measurement, a depth cell, pitch, roll and yaw rate gyros, and heading and vertical gyros.

Phoenix has three sonars: a PSA900 altimeter sonar, an ST1000 mechanically steered sonar and an ST725 mechanically steered sonar. The PSA900 and ST1000 sonars are controlled from the GESPAC computer while the ST725 is controlled by the Sun Voyager. The ST1000 has a one-degree conical beam and a 360-degree sweep [Tritech International Ltd. 92a]. The ST725 also has a 360-degree sweep, with a horizontal width of 2.5 degrees and a vertical width of 28 degrees [Tritech International Ltd. 92b].

Other on-board equipment includes two leak detectors, two lead-acid-gel batteries capable of providing approximately two hours of power for the vehicle's computers and motors, and hydrogen absorbers located throughout the vehicle. The internal layout of *Phoenix* is depicted in Figure 9.

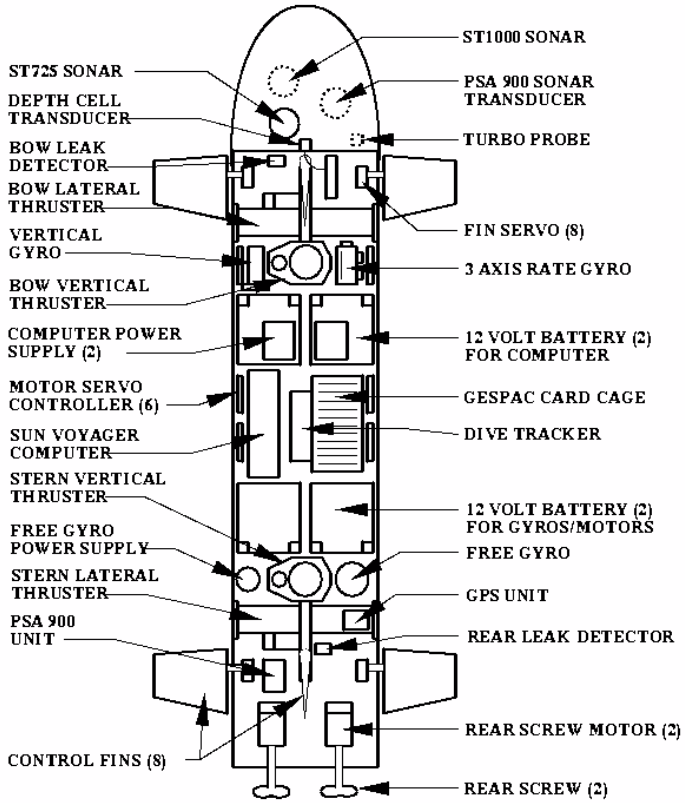


Figure 9: *Phoenix* Internal Hardware Configuration [Leonhardt 96].

2. The Rational Behavior Model (RBM)

a. Overview

The Rational Behavior Model (RBM) is a three-layer software architecture for the control of autonomous vehicles [Byrnes 93, Byrnes 96]. RBM attempts to closely model the command structure of manned ships as depicted in Figure 10. The top layer (strategic level) is responsible for defining high-level goals and controlling overall mission sequencing. The strategic level of RBM roughly corresponds to the commanding officer of a manned ship. The middle layer (tactical level) is responsible for interpreting the high-level guidance from the strategic level and issuing control commands to the lowest layer

(execution level) [Marco 96b]. In addition to direction of the execution level, the tactical level is responsible for navigation, obstacle detection/classification, obstacle avoidance, path planning, and system monitoring [Leonhardt 96]. The responsibilities of the tactical level are analogous to those of the officer watch team on a manned ship. The execution level is responsible for interfacing with the vehicle's hardware to produce desired physical responses. This layer corresponds to the watch-standers on a manned ship. In *Phoenix* implementation of RBM the strategic and tactical levels run on the Sun Voyager while the execution level runs on the Gespac computer. Communication between the tactical and execution levels is via BSD Unix sockets while communication between processes at the tactical level is via Unix pipes [Leonhardt 96].

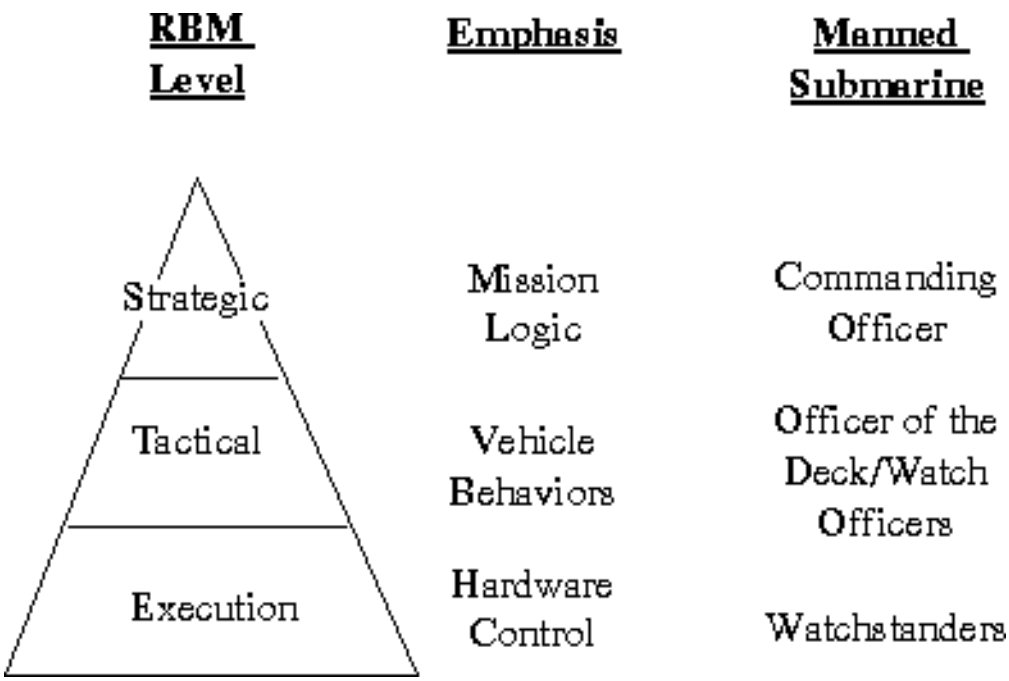


Figure 10: The Rational Behavior Model Software Architecture [Holden 95].

b. Strategic Level

An RBM strategic-level mission is structured as a *deterministic finite automata* (DFA), sometimes referred to as a *finite state machine* [Hopcroft 79]. Each high-

level mission goal (or phase) represents a node (or state) in the DFA. Transitions within the DFA occur whenever a phase succeeds or fails. Upon phase completion or failure, subsequent phases to execute are specified by the transitions of the DFA. Thus each node has two exit transitions: one for successful phase completion and one for phase failure. On first consideration, limiting each node of the DFA to exactly two exit transitions might seem to restrict the versatility of the RBM strategic level. However any DFA of arbitrary complexity can be restructured as a logically equivalent binary DFA because any decision tree can be restructured into an equivalent binary decision tree [Rowe 88]. Thus this restriction on the DFA structure in no way limits the versatility of the strategic level. A graphical representation of an RBM strategic-level DFA for a simple search mission is shown in Figure 11. Implementation of the strategic level as a structured DFA provides a flexible means of describing and sequencing sophisticated missions.

In order to execute a mission, the strategic level requires three software components. The first part is a DFA specification of the mission. The second part is a mission controller that will control transitions through the DFA and initiate the appropriate phases at the appropriate times. The final part is a set of primitive strategic-level goals that provide the syntax and semantics of a command language from the strategic to the tactical level. These goals are implemented as messages to the tactical level.

The set of available messages to the tactical level constitute what amounts to a tactical-level command language. Commands are used to tell the tactical level to start timers, specify hover points and waypoints, conduct searches and to perform other high-level operations that make up the strategic level's primitive goal set.

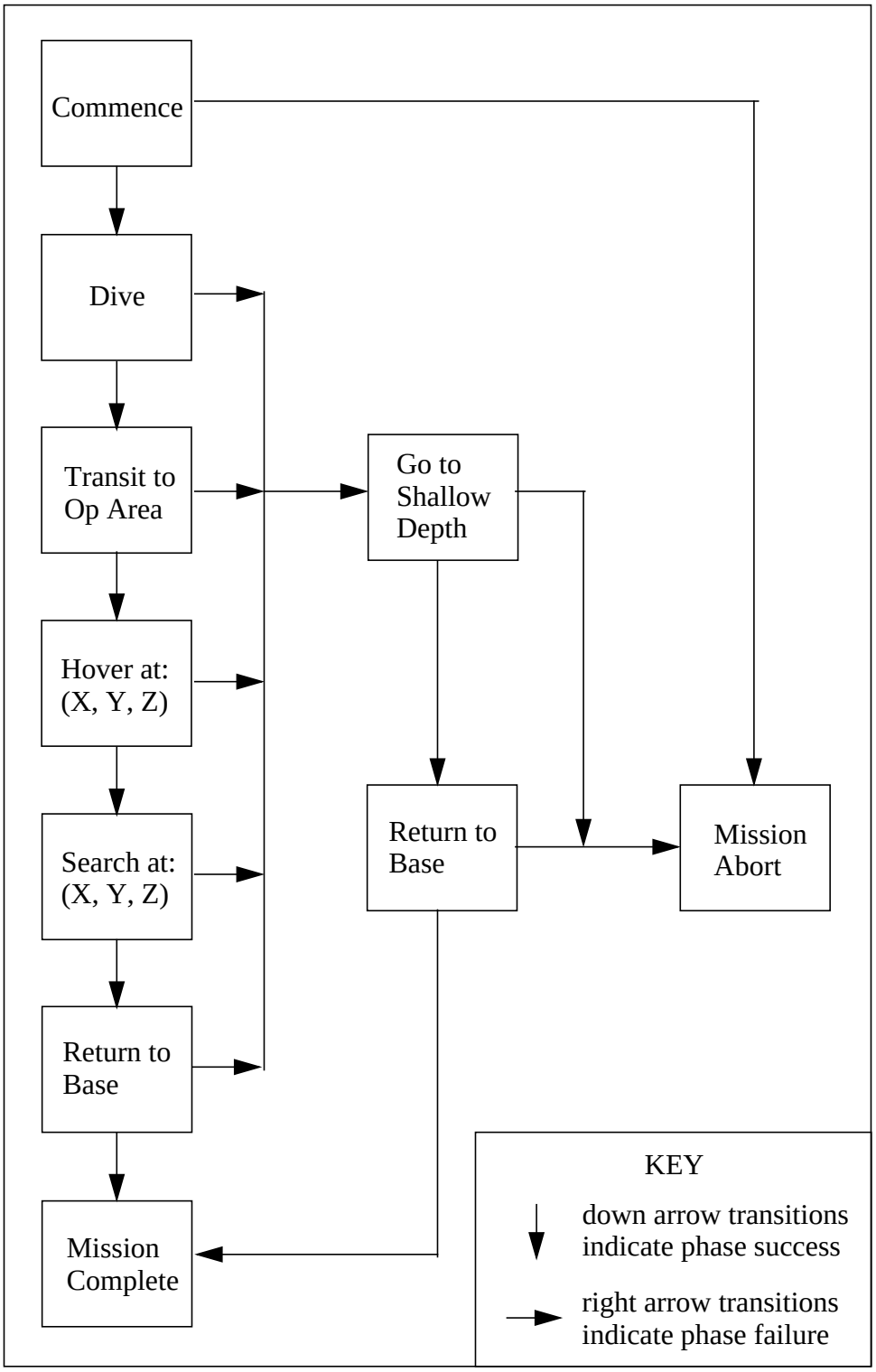


Figure 11: A Simple RBM Strategic Level Search Mission.

A final noteworthy aspect of the RBM strategic level is the absence of mathematical computation. The RBM architectural structure permits arithmetic computations to be performed only in the lower two RBM layers. In fact the strategic level as initially proposed in [Byrnes 93] further required that numerical data be confined entirely to the lower levels of the RBM. It has subsequently been found desirable to permit numerical data at the strategic level as part of the high-level goal specifications [Leonhardt 96], although this level remains concerned only with initiating phases and waiting for successful completion or failure. This example alludes to a larger issue that must constantly be dealt with: what facilities need to be placed at what layers of a multi-layer software architecture? In this instance, the final decision was based more on how the data was being used than what type of data it was.

c. Tactical Level

On *Phoenix* several concurrent processes are used to implement the tactical level. These processes consist of the officer of the deck (OOD) module, the sonar module, the navigator module and the replanner module [Leonhardt 96]. Future plans include the implementation of an engineer module that will be responsible for monitoring and troubleshooting vehicle systems and detecting system failures and degradations.

The OOD module receives commands from the strategic level and state information from the execution level. The OOD uses this information to direct the other tactical level modules and the execution level [Leonhardt 96]. Additionally, the OOD module determines when individual phases have completed or failed and responds accordingly to strategic level queries concerning the status of the current phase. OOD responses to strategic level queries are always binary in nature and indicate a yes or no response [Byrnes 96]. More detailed information concerning the implementation of the OOD module can be found in [Leonhardt 96].

The sonar module is responsible for controlling the ST725 sonar and interpreting the sonar's data. During most operations the sonar is swept back and forth directly in front of the AUV in order to find and classify objects in *Phoenix*' path, but it can also be used to conduct a 360 degree search from a hover [Campbell 96]. The sonar module uses parametric linear regression to construct line segments from sonar returns and a rule based expert system to connect line segments into polygons. This sonar return classification process is described in detail in [Brutzman 92] and [Campbell 96].

The navigator module is responsible for maintaining accurate current position information. A Kalman filter is used to combine GPS, differential GPS, Divetracker™ and dead reckoning data to compute *Phoenix*' position. Implementation details of the navigator module can be found in [McClarin 96].

The replanner is responsible for planning safe paths around obstacles detected by the sonar module. Replanner implementation is covered in detail in [Leonhardt 96].

d. Execution Level

The execution level is implemented as a single closed-loop process. Each loop iteration consists of three phases: sense, decide and act. The execution level process reads sensors and computes values for parameters that do not have a dedicated sensor during the sense portion of the loop. The execution level process then uses this information to determine what control inputs are necessary to achieve the most recent tactical level command. Finally appropriate commands are sent to each control actuator. [Burns 96]

In addition the execution level forwards a copy of the updated state vector to the tactical level and checks for a new command from the tactical level each time through the closed loop. The complete set of tactical level commands also constitutes a command language [Brutzman 96]. Each command consists of a keyword followed by a number of

parameters. Execution-level commands are available for explicitly setting control actuators, setting control modes and updating state information such as position and ocean current that is maintained at the execution level. The most recent command determines what control mode the AUV will use. Available control modes include hover control, waypoint control, lateral control, rotate control and a few others. A subset of the available commands is shown in Figure 12, with a complete listing included in Appendix B.

A final responsibility of the execution level is the initiation of a reflexive mission abort under certain circumstances [Burns 96]. A mission will be aborted if any of the following occurs: leak detected, low battery, imminent collision or loss of primary navigation system. In the event of an automatic abort, the AUV will surface as quickly as possible using thrusters and planes. Upon reaching the surface, the mission will terminate.

3. Precision Maneuvering using Sonar

Recognizing that accurate positioning relative to objects in the AUV's environment is at times more important than accurate global navigation, research into using *Phoenix*' sonars for navigational purposes was begun shortly after the project's inception. Early efforts focused on tactical and execution level coordination and command sequencing in order to facilitate navigational use of the sonar and on implementing primitive behaviors for control and use of vehicle sonars.

WAIT	#	Wait/run for # seconds
RPM	# [##]	Prop ordered rpm values
COURSE	#	Set new ordered course
TURN	#	Change ordered course #
RUDDER	#	Force rudder to # degrees
DEPTH	#	Set new ordered depth
PLANES	#	Force planes to # degrees
ROTATE	#	Open loop rotate control
NOROTATE		Disable open loop rotate
LATERAL	#	Open loop lateral control
POSTURE	#a #b #c #d #e #f (x, y, z, phi, theta, psi)	
POSITION	# ## [###]	Reset dead reckon i.e. navigation fix
ORIENTATION	# ## ###	(phi, theta, psi)
WAYPOINT	#X #Y [#Z]	
HOVER	[#X #Y] [#Z] [#orientation] [#standoff-distance]	
GPS-FIX		Proceed to shallow depth take GPS fix
GPS-FIX-COMPLETE		Surface GPS fix complete
TRACE		Verbose print statements

Figure 12: Sample Execution Level Commands [Brutzman 94].

Early results were published in [Healey 94]. The first significant result of this research was the implementation of vehicle behaviors that used the newly installed lateral and vertical thrusters to obtain hover-like control. These behaviors included heading control, depth and pitch control, lateral speed control and lateral position control. Behaviors were also implemented for use of the sonars and included center sonar, ping and get sonar range, step sonar (without pinging) and initiate or reset the sonar data filters. The philosophy used during this research was to accurately implement functionality at the execution level before attempting to use these behaviors at the higher RBM levels [Healey 94].

Once accurately implemented, these behaviors were used to achieve bottom-following and wall-following behaviors. These behaviors were implemented using simple proportional derivative (PD) control laws for thruster values. Command sequencing and timing were also addressed at this stage. For example, it is futile to command the AUV to maintain a distance from a wall if the sonar is not directed towards the wall. It is therefore the responsibility of the tactical level to sequence commands to the execution level appropriately. [Healey 94]

Recently a more robust method of AUV positioning relative to an object has been developed. This method, documented in [Marco 96a], uses the ST1000 to locate the target and uses a mathematical model to navigate to the commanded location relative to the object.

The position of the object, in this case a 0.5 meter diameter cylinder, is determined by continually sweeping the ST1000 through a sector centered on the expected bearing of the object. The sector size was 70 degrees and angular resolution of the sonar was 1.8 degrees. Sonar returns are connected into segments which are examined to determine which segments represent the cylinder. Simple rules based on the cylinder's size, shape and location are used to determine which segments comprise the cylinder. Once the

cylinder is identified, the location of the vehicle in a navigation frame attached to the cylinder with axes aligned North (x) and East (y) can be computed.

Since the target position update is much slower than the ten hertz control loop, a simplified mathematical model for hydrodynamic response is used to navigate towards the desired relative position between updates. The model includes drag, added mass and steady state surge. It is assumed that the estimated position of the target based on sonar returns is accurate while the mathematical model is inaccurate. Therefore the current model estimate is reset whenever the sonar updates the target position. Results reported in [Marco 96a] indicate that this methodology works well despite known inaccuracies in the mathematical model.

D. SUMMARY

Given the wide array of potential uses and advantages for AUVs, it is no surprise that research is being conducted by numerous organizations worldwide. There are however many issues that remain to be resolved. One of these is AUV recovery. Several organizations have begun work on different recovery techniques, and there are a number of systems in various research stages. Various aspects of these systems may prove helpful in solving the problem of covert launch and recovery of AUVs from submarines.

Research conducted using *Phoenix* in the area of precision maneuvering using sonar may prove helpful as well. The technique of combining sonar feature extraction and model-based control in particular forms the basis of a significant portion of the research detailed in the following chapters.

III. RESEARCH METHODOLOGY

A. INTRODUCTION

This chapter is intended as an overview of the tools and methodology used during this research. This discussion is broken into three sections. Section B covers the Underwater Virtual World (UVW), a three-dimensional (3D) graphical simulation that supports realistic and comprehensive testing of an AUV in the laboratory. Section B also examines specific of the UVW and the enhancements that were made to support this research. Section C covers implementation and testing using the UVW. Section D covers validation of vehicle software in the real world.

B. UNDERWATER VIRTUAL WORLD (UVW)

1. Overview

Implementation and testing of AUV software in the real world is inherently difficult for a number of reasons. Logistical requirements, vehicle maintenance and limited power supplies all limit the amount of in-water testing that is possible even under optimal circumstances. Additionally, the remote environment in which AUVs operate precludes run-time monitoring and can make data evaluation after the fact difficult at best. Finally the unpredictability of the marine environment may make it difficult or impossible to conduct tests within desired environmental parameters. The UVW is meant to address all of these issues. By providing a means of comprehensively and accurately testing the AUV in the laboratory, the UVW allows the implementation and testing of vehicle software under conditions such as ocean current, restricted-area maneuvering and depths that are impractical or impossible to duplicate in real-world testing. [Brutzman 94, Brutzman 95]

The UVW is organized in two fairly distinct pieces: the dynamics module and the viewer. The dynamics module represents the virtual world in which the AUV is operating.

Included in the dynamics module are vehicle hydrodynamics and simulated sensor response. During the sense portion of the control loop, the vehicle's execution-level software relays a copy of the state vector from the previous loop to the dynamics module. The state vector includes values for all salient vehicle characteristics including posture, velocities, accelerations, and control and sensor settings. The dynamics module applies the vehicle's hydrodynamics formulas, calculates the sensor readings, and returns an updated state vector to the execution level. This relay of state vectors between the dynamics module and the execution level takes the place of physical sensor readings and actuator response by the execution level in the real world. [Brutzman 94]

The second portion of the UVW, the viewer, provides real-time interactive 3D graphics visualization of the AUV during test runs in the UVW. Control settings (planes, propellers and thrusters) and sonar are represented graphically allowing intuitive qualitative analysis of vehicle performance. Since the AUV relies only upon its sensors, visualization is of little importance to the vehicle itself. It is, however, extremely useful to human operators to be able to see how the AUV is performing without having to analyze large amounts of data. The diagnostic value of this tool has been proven on an almost daily basis. [Brutzman 94, Brutzman 95]

The viewer is written using the *Open Inventor* graphics package [Wernecke 94]. Based on the *Open GL* graphics library, *Open Inventor* provides an object-oriented extension to the C++ programming language for scene description and manipulation. A scene is represented as a graph. A node in the graph represents some piece of information about the scene such as an object, a location, a material or a scaling factor. When an action (such as render) is applied to the scene graph, the graph is traversed in a depth-first fashion described in [Wernecke 94] and the action is applied to each node in turn. Figure 13 shows the *Open Inventor* scene graph used to represent *Phoenix*. A rendered depiction of a scene graph from the UVW is shown in Figure 14.

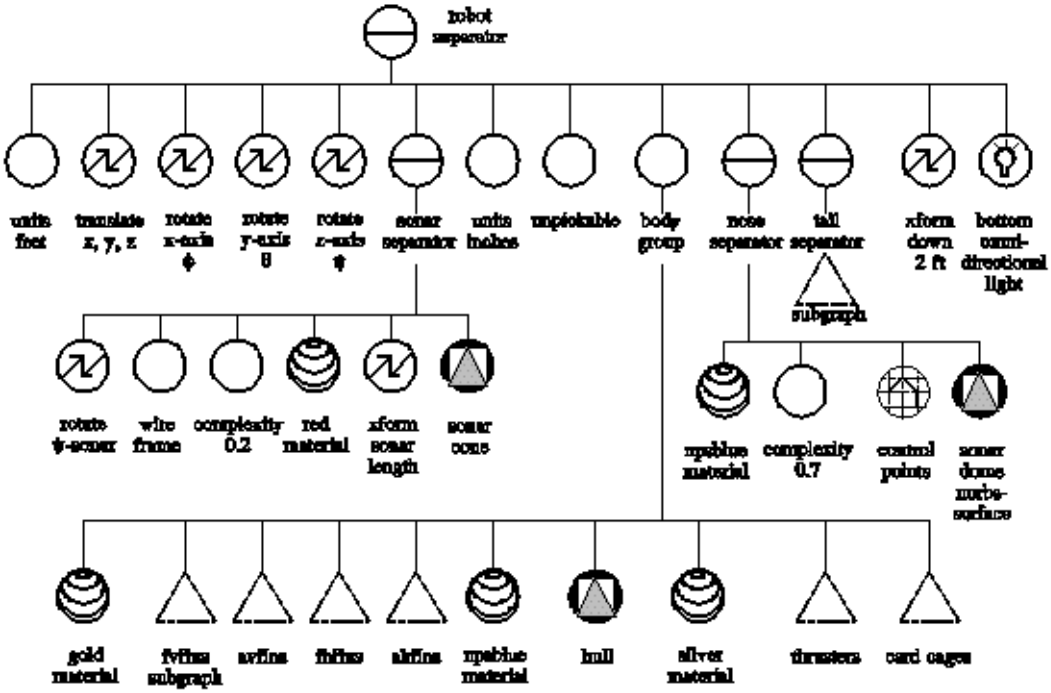


Figure 13: UVW Viewer Scene Graph Representation of *Phoenix* [Brutzman 94].

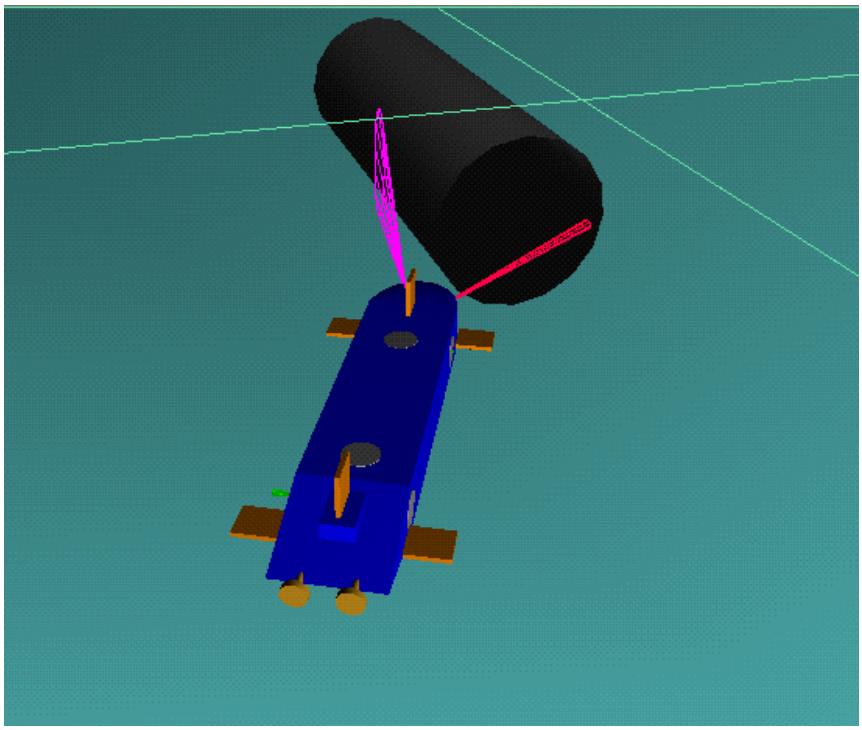


Figure 14: Visualization in the UVW.

A second feature of *Open Inventor* is its scene description language [Wernecke 94]. The scene description provides a means of representing scene graphs using readable text. Objects defined using the scene description language and stored in files can be loaded into the scene graph at run-time. Similarly, any portion of the scene graph can be written to a file at run-time for later use or analysis. The ability to read and write portions of the scene graph at run time is especially useful in the UVW since it allows arbitrary objects to be loaded into the scene graph for different missions.

2. Sonar Simulation and Visualization

a. General

The most significant limitation of the initial version of the UVW used during this thesis was the sonar model. Until recently UVW sonar representation was limited to the ST1000 sonar and only to the 25 ft by 25 ft CAUVR test tank. This representation used a simplified planar two-dimensional trigonometric model described in [Brutzman 94] to calculate sonar returns based on a known AUV position within the tank. Other objects present in the scene graph were not represented in the sonar model. In order to support this and other research, a more general sonar model representing arbitrary targets and both the ST1000 and the ST725 sonars was needed.

The solution produced for this thesis is to use facilities present in the *Open Inventor* package to simulate both sonars. One of the actions available in *Open Inventor* is a ray-pick action (SoRayPickAction) [Wernecke 94]. To use the ray-pick action, the starting point of the ray and its orientation are specified and the action is applied to the scene graph. After application, the ray-pick action returns the point (if any) where it first intersected an object in the scene graph to which it was applied. If the origin of the ray corresponds to the location of a sonar, and the orientation of the ray corresponds to the orientation of the sonar, then the distance from the origin of the ray to the first intersection

with an object in the scene graph is analogous to the sonar range. Because of the short ranges involved (less than thirty meters), bending of the sonar beam is assumed to be negligible [Brutzman 94]. Such an approximation usually remains valid at longer sonar ranges (hundreds of meters) but depends on the sound speed profile of the environment [Urick 83].

Since sensor modeling is handled in the dynamics module, a copy of the scene graph must loaded into this module in order to use the ray-pick action to compute sonar ranges. While the dynamics module uses a copy of the scene graph, there is no need for the dynamics module to render it. By maintaining a copy of the scene graph in the dynamics module without rendering it, a general geometric sonar model has been implemented without sacrificing real-time performance [Brutzman 96].

Because of the imperfect nature of sonar data an error model must also be implemented in order to accurately represent a sonar. In the absence of empirical sonar error data on the ST1000 and ST725 sonars, a uniform error distribution has been implemented where the user can specify the maximum amount of error as a percent of the range. The sonar range including error is computed for either sonar by the dynamics module using the formula

$$R_{Ray-error} = e \cdot (rand(2) - 1)R_{Ray} + R_{Ray} \qquad \text{(Eq. 1)}$$

where e is the maximum error percentage, $rand(2)$ is a random number between zero and two and R_{Ray} is the error-free sonar range returned by the geometric sonar model. As more empirical error data becomes available, the sonar error distribution will be modified to more accurately represent the performance of both sonars. A uniform error distribution can be modified to provide an arbitrary empirical probability distribution in a straightforward manner as explained in [Fishwick 95].

b. ST1000 Sonar

Because the ST1000 sonar is a one-degree conical (pencil-beam) sonar, its representation using the ray-pick action is fairly straightforward and uses a single ray. The location of the sonar head in world coordinates is computed using the position and orientation of the AUV in world coordinates (data that is encapsulated in the AUV's homogeneous transformation matrix) and the position of the ST1000 in AUV body coordinates. The homogeneous transformation matrix is defined as [Craig 89]

$$H = \begin{bmatrix} c(\psi)c(\theta) & c(\psi)s(\theta)s(\phi) - s(\psi)c(\phi) & c(\psi)s(\theta)c(\phi) + s(\theta)s(\phi) & x \\ s(\psi)c(\theta) & s(\psi)s(\theta)s(\phi) + c(\psi)c(\phi) & s(\psi)s(\theta)c(\phi) - c(\theta)s(\phi) & y \\ -s(\theta) & c(\theta)s(\phi) & c(\theta)c(\phi) & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{(Eq. 2)}$$

where ψ, θ and ϕ are the AUV azimuth, elevation and roll respectively, (x, y, z) is the AUV position in world coordinates, and $c(X)$ and $s(X)$ are cosine and sine functions respectively. Using the homogeneous transformation matrix the position of the ST1000 sonar in world coordinates is computed as

$$\begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix} = H \begin{pmatrix} x_b \\ y_b \\ z_b \\ 1 \end{pmatrix} \qquad \text{(Eq. 3)}$$

where (x_b, y_b, z_b) is the position of the ST1000 sonar head in AUV body coordinates.

The orientation of the ray representing the ST1000 is found in a similar fashion using the orientation of the sonar beam relative to the AUV and the rotation matrix corresponding to the orientation of the AUV. Since the ST1000 sonar only has one degree of freedom (DOF) (rotation about the z-axis), the unit vector representing ST1000 beam orientation relative to the AUV can be computed using

$$V_b = \begin{pmatrix} \mathbf{cos}(\psi_b) \\ \mathbf{sin}(\psi_b) \\ 0 \end{pmatrix} \qquad \text{(Eq. 4)}$$

where ψ_b is the bearing of the ST1000 sonar. The vector representing the orientation of the beam unit vector in world coordinates is computed using the formula

$$V_e = RV_b \qquad \text{(Eq. 5)}$$

where R is the rotation matrix of the AUV given by [Craig 89]

$$R = \begin{bmatrix} c(\psi)c(\theta) & c(\psi)s(\theta)s(\phi) - s(\psi)c(\phi) & c(\psi)s(\theta)c(\phi) + s(\theta)s(\phi) \\ s(\psi)c(\theta) & s(\psi)s(\theta)s(\phi) + c(\psi)c(\phi) & s(\psi)s(\theta)c(\phi) - c(\theta)s(\phi) \\ -s(\theta) & c(\theta)s(\phi) & c(\theta)c(\phi) \end{bmatrix} \qquad \text{(Eq. 6)}$$

This equation corresponds to the top left portion of the matrix of Equation 3.

Once the location of the sonar head and the orientation of the sonar beam have been calculated in world coordinates, the ray-pick action is applied to the scene graph. The distance from the origin of the beam to the point returned by the ray-pick action is then calculated and error is added to the result using Equation 1.

c. ST725 Sonar

The ST725 sonar differs from the ST1000 sonar in two significant respects that complicate its representation in the UVW. First, the sonar beam of the ST725 is not a pencil-beam and cannot be adequately represented by a single ray like the ST1000. Second, the data returned by the ST725 is not simply a range to the nearest target but rather a data structure representing the strength of the return at regular intervals out to the maximum range. These issues are both dealt with by fusing the results of multiple ray-pick actions.

Before describing the actual implementation of the ST725 sonar in the UVW, it is important to understand the data structure returned by the ST725 and how it is interpreted by the sonar manager. The data structure returned by the ST725 is a 32-byte

sequence that is divided into 64 bins of four bits each. A bin represents the strength on a scale from zero to 15 of the sonar return at a certain range. The range represented by a bin is proportional to the maximum sonar range and can be approximated linearly using the formula

$$R_i = \frac{R_{Max}}{64}\left(i + \frac{1}{2}\right) \qquad \text{(Eq. 7)}$$

where R_{Max} is the maximum range setting of the sonar and bins are numbered zero to 63.

The tactical-level sonar manager uses this data structure to compute a single range for the ST725. The range used is the shortest range whose bin value is above a predefined minimum unless the value of a bin representing a longer distance is significantly larger (strength difference greater than two). If this is the case the longer range is used. This algorithm is discussed in more detail in [Campbell 96].

The UVW implementation of the ST725 uses an array of 64 integers to represent the returned data structure. The values contained in this array are determined by the results of 13 ray-pick actions applied to the scene graph. The rays for all 13 ray-pick actions originate at the position of the ST725 sonar head which is computed using Equation 2 with (x_b, y_b, z_b) representing the location of the ST725 sonar head in AUV body coordinates (the positions of the ST725 and ST1000 sonars in AUV body coordinates is shown in Table 2). The vector representing the orientation of each of the 13 rays in AUV body coordinates varies above and below the horizontal plane of the sonar. Ray orientations are computed using

$$V_{bi} = \begin{pmatrix} \mathbf{cos}(\psi_b) \\ \mathbf{sin}(\psi_b) \\ \mathbf{tan}(2i^\circ - 12^\circ) \end{pmatrix} \qquad \text{(Eq. 8)}$$

where ψ_b is the bearing of the ST725 sonar and rays are numbered from zero to 12. This equation differs from Equation 4 only in the third term of the vector which allows the entire

vertical sonar beam to be represented. It should be noted that V_{bi} is not normally a unit vector. While conversion to a unit vector is a simple matter, the ray-pick function does not require orientation specified by a unit vector, so the conversion is not performed in the interest of computational efficiency. Once the orientation of the rays in AUV body coordinates has been calculated, the orientation in world coordinates can be computed using Equation 5. A ray-pick action is applied to the scene graph for each of the 13 rays. The range to the point returned by the ray-pick action is calculated, and the value stored in the array of integers corresponding to the appropriate range bin is incremented by one.

Sonar	x_b	y_b	z_b
ST1000	2.875	-0.167	0.3333
ST725	2.625	-0.167	-0.3333

Table 2. ST1000 and ST725 Positions (ft) in AUV Body Coordinates.

After all 13 ray-pick actions the error free sonar range is computed as the range corresponding to the element of the array of integers with the highest value. If no element in the array is greater than one, the error free sonar range is set to zero. Sonar error is then added to the error free range using Equation 1. Although no profiling measurements were performed on the source code, this operation appears highly efficient. The sonar module (operating in series with the network communications and hydrodynamics model) has no difficulty executing 14 ray-picks into complex scene graphs within the bounds of a ten Hertz update rate. Thus computational performance of the arbitrary geometric sonar model is excellent.

d. Visualization

Once ranges have been computed for the ST725 and ST1000 sonars, visualization using the viewer is straightforward. The goal of sonar visualization in the UVW is to enable the human operator to see the operation of both sonars. Visualization has proven particularly useful for detecting and troubleshooting sonar control algorithms since it provides the only intuitive verification that the sonars are being controlled as intended. Numerous experiments conducted in the course of this research have shown that sonar visualization is crucial to tactic diagnosis and mission rehearsal.

Sonar beams are represented in the UVW viewer using wireframe cones. Nodes representing the sonar cones are placed in the portion of the scene graph representing the AUV. Additional nodes are inserted into the graph to represent the positions and orientations of the sonars relative to the AUV. In order to accurately depict the pie-slice shape of the ST725 sonar beam, the vertical scale of the cone representing it is increased by a factor of 12. They are depicted as wire frames rather than solid objects in order to preclude the sonar cones from obscuring other portions of the scene.

In addition to positions and orientations of the sonars, target range information is depicted. This is accomplished quite simply by scaling the length of the cones representing the sonar beams to the range of the appropriate sonar return. If the ray-pick sonar range is zero (no scene graph object was within range), it is important to visually depict lack of contact as well. In this instance the sonar cone length is scaled out to the maximum range, and for visual contrast the color is changed and the wireframe complexity is decreased. The ST1000 sonar cone is red if a valid return is received and yellow if no return is obtained. The ST725 sonar is correspondingly rendered in magenta or white. The portion of the viewer scene graph representing the ST725 sonar is shown in Figure 15.

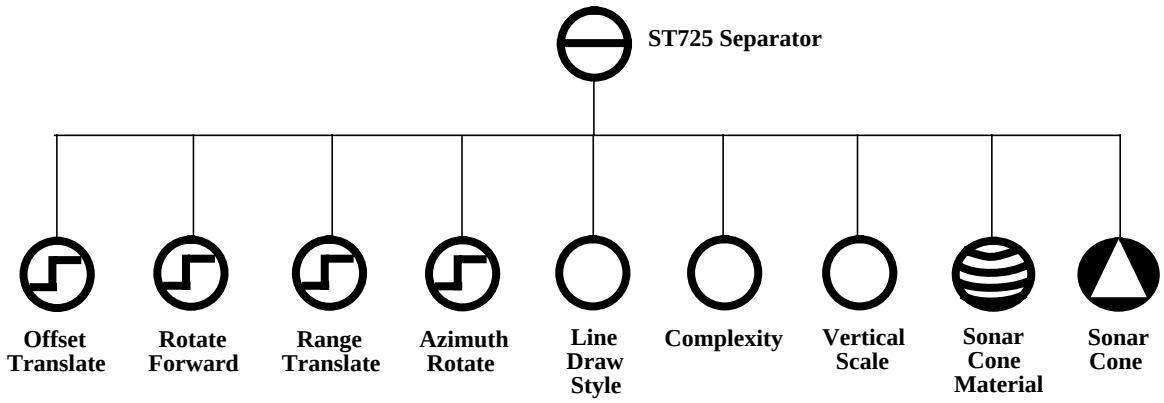


Figure 15: *Open Inventor* Scene Graph Representing the ST725 Sonar.

C. IMPLEMENTATION AND TESTING IN THE VIRTUAL WORLD

The UVW was the primary tool for *Phoenix* software implementation and initial testing during the conduct of this research. While it was often the case that in-water testing was conducted concurrently with UVW testing, for safety and reliability no algorithms were tested in the water prior to being tested in the UVW.

The general philosophy used during the conduct of this research was similar to that of the research documented in [Marco 96]. Primitive functionality was implemented and tested before attempting to implement higher level behaviors. While a variety of low-level issues were not identified until higher-level behaviors were implemented, these were the exception rather than the norm.

The first issues dealt with were sonar control, target acquisition and tracking using the ST1000 sonar. Once these behaviors were implemented, control modes were implemented to allow *Phoenix* to maintain a commanded relative range and bearing from a sonar target. These behaviors form the base upon which a great deal of this research rests. The next step was to implement higher-level routines that used these sonar and control modes to execute a torpedo-tube approach. These are primarily tactical-level issues and involved path planning and command generation.

Most strategic-level research relevant to this thesis was far enough removed even from the tactical level issues that it could be conducted in parallel almost from the beginning. The major goal of strategic level research was to simplify the mission-generation process to such a degree that a user did not have to be a *Phoenix* expert to be able to program a complex mission. Specification of location and type of recovery is one aspect of this area of research. The most significant result of this research was a mission-planning expert system for automatic generation of *Phoenix* missions. Much of this joint research is documented in [Leonhardt 96] with more detailed coverage later in this thesis.

D. IMPLEMENTATION AND TESTING IN THE REAL WORLD

Real-world implementation and testing occurred in two parts: implementation and testing on the vehicle's hardware and verification of virtual world results. Because *Phoenix* does not actually use physical sensors and controls when missions are conducted in the virtual world, it is necessary to verify the software's interface with the actual vehicle hardware before conducting in-water tests. Physical control of the sonars, reading and filtering of sensor data, and polarity and response of control actuators all must be verified by bench tests and (to a lesser degree) by in-water tests. A more detailed discussion of this topic can be found in [Burns 96].

Real-world verification of UVW results is conducted in much the same manner as the initial implementation. Initial tests were intended to confirm the sonar control and tracking behaviors, with subsequent tests verifying the station-keeping behaviors. Testing of higher-level behaviors (including the full torpedo-tube recovery) were contingent upon successful low-level behavior tests. A detailed discussion of real-world and virtual-world test results is contained in Chapter VII of this thesis.

E. SUMMARY

This chapter provides an overview on how this research was conducted. The UVW was of key importance to the conduct of this research. In order to facilitate its use, a general sonar model was implemented to simulate the response of the ST725 and ST1000 sonars. The sonar model was implemented by importing a copy of the scene into the UVW's dynamics module using the *Open Inventor* ray-pick function to simulate the sonar beam. Visualization was also implemented for both sonars in the viewer portion of the UVW.

Subsequent to implementation of a general sonar model for the ST725 and ST1000 sonars, the UVW was used as the primary implementation and testing tool. With the exception of hardware interfacing, all aspects of this research were implemented and tested in the UVW prior to attempting real-world tests. Implementation of *Phoenix* software was conducted primarily in a bottom-up fashion with low level functionality being implemented and tested prior to implementing higher level behaviors. Once functionality was tested in the UVW, real-world tests were conducted to ensure proper hardware utilization and response and verify UVW results.

The following chapter describes behaviors implemented at the execution level of *Phoenix* software architecture to support recovery. Implemented behaviors include various sonar control modes that can be used to locate and track objects in *Phoenix* environment, a vehicle control mode for stationkeeping relative to an object being tracked, and a vehicle control mode for physical entry into a recovery tube.

IV. EXECUTION LEVEL IMPLEMENTATION

A. INTRODUCTION

This chapter discusses implementation of behaviors at the execution level that are required during recovery. Since the execution level is primarily responsible for low-level physical control and interfacing with the vehicle's hardware, behaviors implemented at this level must be fairly simple but robust. It is the responsibility of the tactical level to invoke execution-level behaviors to carry out tactics that will (in turn) accomplish still higher-level goals specified by the strategic level.

The next section in this chapter details implementation of ST1000 sonar control which is built upon the primitive behaviors described in [Healey 94]. Specific sonar-control modes implemented include a manual control mode, a forward-looking-scan mode for collision avoidance, a target-search mode for locating targets specified by the tactical level, and two target-tracking modes for use during station keeping. The third section covers implementation of vehicle control modes for station keeping relative to a target. Finally, implementation of a vehicle control mode for entry into the recovery tube is presented in detail.

B. SONAR BEHAVIOR

1. Manual Control

The simplest and most obvious ST1000 behavior is "manual" control. This control mode responds to commands from the tactical level by positioning the sonar at specified relative bearings. Manual control provides a means for the tactical level to completely control the operation of the ST1000 sonar for target classification, obstacle detection or other operations that may be more suited to the ST1000 than the ST725. In addition manual

control is used during the final phase of the recovery to position the ST1000 for distance keeping from the side of the tube.

The current ST1000 bearing is maintained at the execution level. When a bearing is commanded, the ST1000 is stepped towards the commanded bearing at a rate of one step per closed loop cycle. Step size for the ST1000 can be set to 0.9, 1.8 or 3.6 degrees (a step size of 0.9 degrees was used during this research). Once the commanded bearing is reached, the sonar will remain at this relative bearing until a new command is received or until the control mode is changed.

When under manual control the sonar will ping once per closed loop cycle (six or 10 hertz) whether it is being stepped towards the commanded bearing or has already reached it. This behavior makes it possible for the tactical level to control a manual sector scan simply by alternating bearing commands between the edges of the scan sector. Other fairly robust behaviors can be similarly controlled by the tactical level.

Commanded sonar bearing is converted to an achievable bearing and normalized to a range of [0 .. 360) degrees before the sonar is actually scanned. This prevents the sonar from stepping back and forth across a commanded bearing and simplifies determination of scan direction. As an example suppose a bearing command of -10.0 degrees is received by the execution level. Using a step size of 0.9 degrees and starting from 0.0 degrees, the sonar is capable of being scanned to -9.9 degrees or -10.8 degrees, but not -10 degrees exactly. The commanded bearing is therefore converted to -9.9 degrees (since that legal value is closest to the actual commanded bearing). The roundoff function is defined as

$$\psi_{command-rounded} = \frac{(double)\left(\frac{(integer)(\psi_{command} \times 10.0))}{9} \times 9\right)}{10.0} \qquad \text{(Eq. 9)}$$

Since the current ST1000 bearing is maintained in the range of [0 .. 360) degrees, the commanded bearing is normalized to 350.1 degrees so that the commanded and current

bearings can be compared. The difference between the commanded bearing and current bearing is then normalized to a range of -180 degrees to 180 degrees. If this difference is greater than zero, the sonar is scanned to the right; if it is less than zero, the sonar is scanned to the left.

2. Forward Scan

For many *Phoenix* evolutions, particularly transits in flight mode, it is desirable to use the ST1000 in a forward-looking scan pattern. This sonar operating mode has been implemented and is automatically initiated whenever the execution level receives a command that will require any of the following vehicle control modes: hover control, waypoint control, open-loop lateral control, open-loop rotate control or any other kind of thruster control.

This forward scan pattern is primarily used for used for imminent collision detection and will trigger a reflexive mission abort as described in [Burns 96] if an obstacle is detected within a certain range. The scan sector is of constant size and is centered about zero degrees relative to the heading of the AUV. The default sector size is 30 degrees but can be arbitrarily changed using mission-script commands which are listed in Appendix B.

3. Target Search

Since the ST1000 is to be used for precision control relative to objects near *Phoenix*, sonar-control modes are necessary for locating and tracking those objects. The implementation of the ST1000 target-search mode makes two significant assumptions: the target has been identified by the tactical level, and the target can be discriminated from background objects based on range. The first assumption relates to the tasks assigned to the different levels of RBM, while the second relates to the type of environment expected during station keeping relative to a sonar target.

The first assumption (regarding target identification) relates to successful implementation of tactical level responsibilities including interpretation of sonar data and classification of objects. Initial location of objects by the tactical level can rely on data from the ST725, the ST1000 (probably using manual control by the tactical level) or both. Real-time object classification using sonar has been the subject of previous *Phoenix* research and continues to be an area of significant interest [Brutzman 92, Campbell 96]. Once an object has been identified by the tactical level, the ST1000 target-search mode uses the approximate range and bearing information to find it.

The second assumption (regarding target discrimination) is that the target is in a relatively open area. This assumption allows the target search to rely only on the expected range and bearing to the target rather than heuristics concerning the type of target. The advantage of this approach is its generality. Use of heuristics for target identification assumes that the vehicle has a certain amount of knowledge concerning the characteristics of the target [Marco 96]. This knowledge must be present for every type of object that is to be identified. Basing target identification strictly on range and bearing information does not require knowledge about the characteristics of the target and can therefore be used to locate any type of object. The disadvantage is that it is possible to incorrectly identify a target when operating in a cluttered environment. An uncluttered environment is a good assumption for an at-sea docking station. On the other hand torpedo tubes are a themselves a highly cluttered environment. Even in this case, however, successful maneuvering in an uncluttered environment is an essential prerequisite to attempting more difficult environments.

The method used to determine scan direction for a target search is the same as that used for manual control. Each sonar return during the search is examined to determine if the desired object has been detected. Sonar range and bearing information is used to determine earth-fixed coordinates. The bearing and range from the AUV to the object can

then be computed and compared to the expected range and bearing to the target. In order to simplify calculations, AUV pitch is assumed to be negligible. The position of the ST1000 sonar head in world coordinates is then given by

$$\begin{pmatrix} x_e \\ y_e \\ 1 \end{pmatrix}_{sonar} = H_2 \begin{pmatrix} x_b \\ y_b \\ 1 \end{pmatrix}_{sonar} \qquad \text{(Eq. 10)}$$

where (x_b, y_b) is the position of the ST1000 sonar head in AUV body coordinates and H_2 is the two-dimensional version of Equation 4 and is given by [Kanayama 96]

$$H_2 = \begin{bmatrix} \mathbf{cos}(\psi) & -\mathbf{sin}(\psi) & x \\ \mathbf{sin}(\psi) & \mathbf{cos}(\psi) & y \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{(Eq. 11)}$$

Once the global position of the sonar head has been determined, the range and bearing are converted to world coordinates using

$$\begin{pmatrix} x_e \\ y_e \end{pmatrix}_{return} = \begin{pmatrix} x_{e-sonar} + R\mathbf{cos}(\psi + \psi_{sonar}) \\ y_{e-sonar} + R\mathbf{sin}(\psi + \psi_{sonar}) \end{pmatrix} \qquad \text{(Eq. 12)}$$

where ψ is the AUV heading, ψ_{sonar} is the ST1000 relative bearing and R is the ST1000 range. The range from the AUV centroid to the target is then computed as

$$R = \sqrt{(x - x_{e-return})^2 + (y - y_{e-return})^2} \qquad \text{(Eq. 13)}$$

and bearing from the AUV to the target is computed as

$$\beta = \mathbf{atan}(y_{e-return} - y, x_{e-return} - x) \qquad \text{(Eq. 14)}$$

where $\mathbf{atan}(y,x)$ is a function returning an angle in the range of [0 .. 360) degrees. Equations 10 through 15 are equivalent to the equations defined in [Marco 96a] to calculate the location of *Phoenix* relative to a cylinder. This relationship between sonar range and bearing and AUV range and bearing is shown in Figure 16.

Once the range and bearing from the AUV to the sonar target have been computed, they are compared to the expected range and bearing to the desired sonar target as a

discriminator. If the measured range is within five feet of the expected range and the measured bearing is within 15 degrees of the expected bearing, the return is assumed to be part of the desired target. Once these conditions are met, the sonar-control mode is automatically switched to target track or target-edge track. Which mode to select is explicitly specified by the mission-script command that initiated the target search.

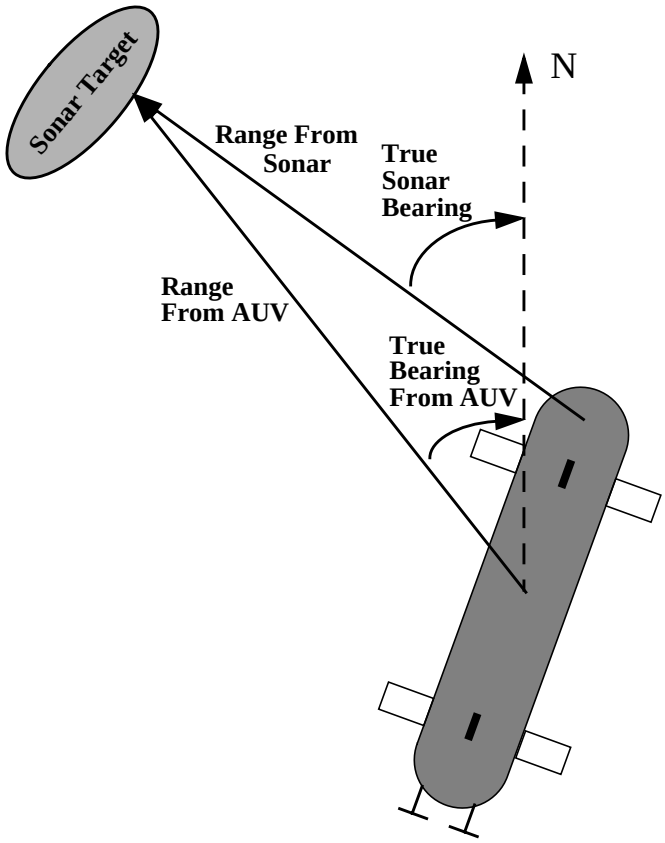


Figure 16: Sonar and AUV Range and Bearing.

4. Target Tracking

Once the desired sonar target has been located, a sonar mode is required to maintain contact with that target. Two such modes have been implemented: a full target-track mode and a target-edge-track mode. When using the full target-track mode the sonar continually

sweeps back and forth across the entire sonar target, updating target range and bearing only after the sonar has scanned off the edge of the target.

As the sonar tracks across the target, each range is compared with the previous range. If the range is within five feet of the previous range, it is assumed to be part of the same target. Because of the somewhat unreliable nature of sonar data, a return that does not meet the range criteria does not necessarily mean that the sonar has scanned off the edge of the target. To account for anomalous sonar returns, three consecutive off-target returns are required to initiate a sonar-scan reversal along with target range and bearing update.

When the sonar controller determines that the sonar has been scanned past the edge of the target, range and bearing estimates are updated using averaging. As the sonar tracks the target a range accumulator is maintained. Anomalous returns that cannot be included in the target are not included in the range accumulator. Sonar range to the target is simply the average of the valid returns from the previous sweep. In addition to the range accumulator, the sonar controller maintains the initial bearing of the current scan. The bearing to the target is then computed as the bisector of the starting and ending bearings of the current scan. Once the range and bearing of the target from the sonar have been determined, target range and bearing from the AUV are computed using Equations 10 through 14.

An illustration of the target-track geometry is shown in Figure 17. The target-track control mode implementation is similar to the sonar control described in [Marco 96a] and differs significantly only in two regards. First, sector width is not fixed but is determined by the size of the target. Second, as with initial target detection, target identification is based on range and bearing rather than target characteristics.

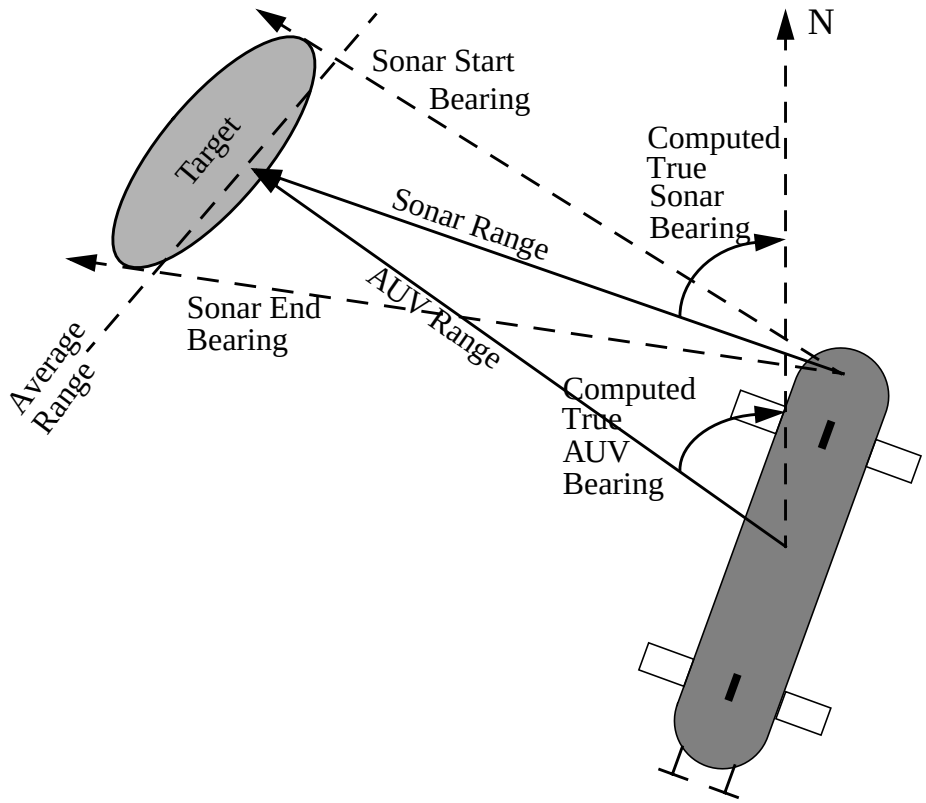


Figure 17: Sonar Full Target-Track Mode Geometry.

In addition to providing target range and bearing information, the target-track control mode has one more benefit. The range and bearing data obtained during the full target track contains a large amount of information about a single object in the world. Since ST1000 range and bearing are part of the state-vector, this data can be concurrently analyzed by the tactical-level sonar module to aid in object classification.

5. Target Edge Tracking

Maintaining sonar contact with the target by scanning across the entire target has one significant disadvantage: the time period between successive range and bearing updates can be as much as ten seconds [Marco 96a]. This slow update rate can lead to sluggish AUV response and navigational inaccuracy because of errors in the onboard hydrodynamics mathematical model described in the following section. In order to

increase the target data update rate, a second target-tracking sonar mode has been implemented. Rather than scanning across an entire object, this control mode attempts to track only the edge of the target.

Once a target has been located using the target-search control mode, the sonar is scanned left or right until it scans past the edge of the object. The target's edge is identified using the same algorithm as full target tracking. Again, once the edge is found the scan direction is reversed. Rather than tracking across the target all the way to the opposing edge, however, the sonar is scanned only until three returns that can be identified as part of the target have been received. Returns are identified as part of the target in the same manner as the full target-track mode. Once three target returns have been received, scan direction is reversed. The edge-track algorithm can be summarized as a loop consisting of four steps: scan off of the target, reverse scan direction, scan onto the target and reverse scan direction. The smaller scan sector width results in target range and bearing update rates that are much faster than those for full target tracking.

Since the sonar does not track across the entire target, average sonar range and bearing to the target's center cannot be computed. Instead, range and bearing are computed to the edge being tracked. Range computation is accomplished in the same manner as with the full target track. Normally the range computed will be the average of three individual sonar ranges. Depending on AUV motion during the scan, however, the actual number of returns included in the average may vary. The sonar bearing of the edge is simply the first bearing from which a valid return was received if the sonar is being scanned onto the target, or the last bearing from which a valid return was received if the sonar is being scanned off of the target. Again, once the range and bearing from the sonar have been determined, range and bearing from the AUV are determined using Equations 10 through 14. Geometry of the target edge-track mode is shown in Figure 18.

The target edge used for tracking is determined by the direction that the sonar is scanned immediately after target detection. If the sonar is scanned to the right, the right edge is used; if sonar is scanned left, the left edge is used. The sonar scan direction is based upon the direction that the AUV will need to move to reach the commanded range and bearing. If the current bearing to the target is less than the commanded bearing, the AUV will need to move left. In this case the sonar is scanned to the left following target detection. Choosing the tracking edge in this manner is intended to prevent the AUV from colliding with large targets because the wrong (i.e. far) edge was used.

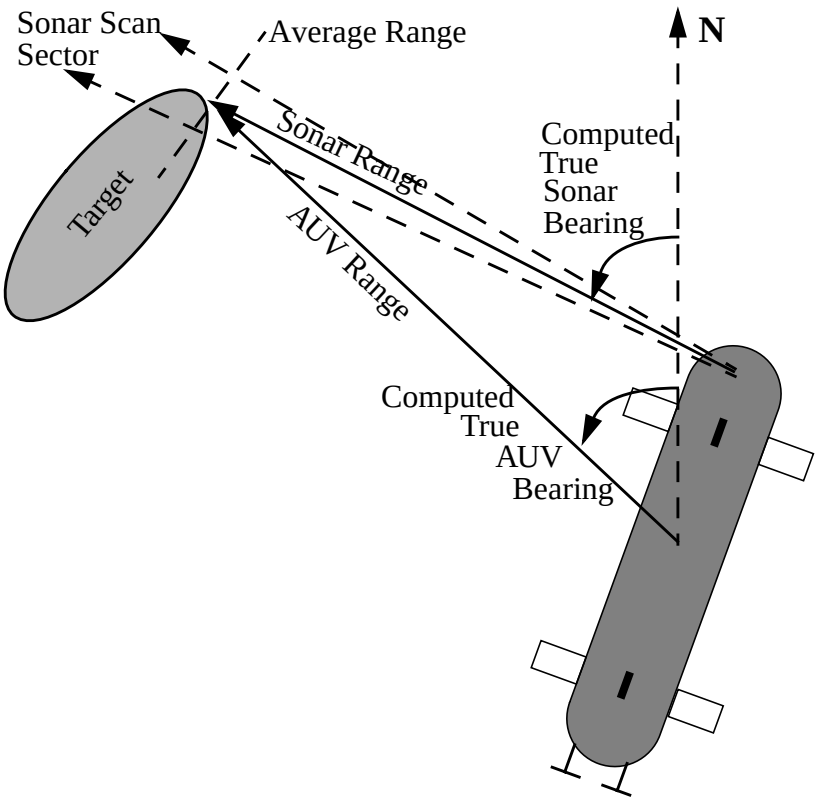


Figure 18: Sonar Target-Edge-Track Mode Geometry.

As stated previously the major advantage of the target-edge-track control mode is an increase in the range/bearing update rate over that of the target-track mode. Target-edge-track has the disadvantage of not obtaining as much information about the target as

the full target-track. An inability to compute the center point of the target is one example of this. Conceivably this disadvantage might be eliminated by simultaneous target-edge-tracking and target-tracking using both the ST725 and ST1000 sonars.

C. STATION KEEPING

1. Station Keeping Commands

Two commands were added to the execution level command language defined in [Brutzman 94] to control station keeping relative to a sonar target: target-station and target-edge-station. These two commands correspond to the two target-tracking sonar modes: target-track and target-edge-track respectively. Both commands have the same parameters and are interpreted in the same way by the execution level. The sole difference between these two commands is the sonar control mode that will be initiated. Format and parameter syntax details for both of these commands can be found in Appendix B.

Station keeping commands can have two, three, four, or five parameters. The four- and five-parameter versions are used to initiate a target search prior to station keeping, while the two- and three-parameter versions are used to change commanded range and bearing from a target already being tracked for station keeping. The two-parameter command specifies a commanded range and bearing, while the optional third parameter can be added to specify a commanded vehicle heading. If no third parameter is present, AUV heading will continuously point directly at the target. The four-parameter command specifies an estimated range and bearing to the target for use during the target search and a commanded range and bearing for station keeping. The fifth parameter specifies a commanded vehicle heading. The difference between the commanded bearing and the estimated current bearing specified in the command is used to determine which edge will be used if target-edge-tracking is called for.

Since execution-level target-track and target-edge-track sonar modes are initiated automatically by the target-search sonar mode, it is impossible to switch between target-track and target-edge-track without initiating a new target search (i.e. by using the target-station or target-edge-station command with four or five parameters). If the AUV is maintaining station relative to a target's edge and a target-station command is received, the target-station command will be interpreted as an edge-station command. Similarly, an edge-station command will be interpreted as a target-station command as appropriate. In addition, if a two- or three-parameter station keeping command is received while the sonar is not in target-track or target-edge-track mode, the station keeping command will be ignored.

2. Commanded AUV Position and Control

The implementation of *Phoenix*' target control involves the translation of the commanded range and bearing to global (x, y) coordinates. Basing station-keeping control laws on global coordinates allows the use of control laws similar to those used for hover control as described in [Burns 96]. Each time the sonar control updates the range and bearing to the target, the global position of the commanded station is computed as

$$\begin{pmatrix} x_{command} \\ y_{command} \end{pmatrix} = \begin{pmatrix} x + \mathbf{cos}(\beta_{current})R_{current} + \mathbf{cos}(\beta_{command} + 180°)R_{command} \\ y + \mathbf{sin}(\beta_{current})R_{current} + \mathbf{sin}(\beta_{command} + 180°)R_{command} \end{pmatrix} \qquad \text{(Eq. 15)}$$

where $\beta_{current}$ and $\beta_{command}$ are the current and command bearings from the AUV to the target and $R_{current}$ and $R_{command}$ are the current and commanded ranges from the AUV to the target. It is important to note that control is based on relative range and bearing between the AUV and target, despite the conversion to world coordinates.

The direction in which the AUV must move to achieve the commanded position is computed as

$$\Gamma = \mathbf{atan}(y - y_{command}, x - x_{command}) \qquad \text{(Eq. 16)}$$

while the distance that the AUV needs to travel is computed as

$$d = \sqrt{(x - x_{command})^2 + (y - y_{command})^2} \qquad \text{(Eq. 17)}$$

The forward and lateral distances relative to the AUV are computed as

$$d_{on-track} = d \cdot \mathbf{cos}(\Gamma - \psi) \qquad \text{(Eq. 18)}$$

and

$$d_{cross-track} = d \cdot \mathbf{sin}(\Gamma - \psi) \qquad \text{(Eq. 19)}$$

respectively. The computed values for $d_{on-track}$ and $d_{cross-track}$ are used with ψ, u and an estimate of ocean current in the form $(x_{current}, y_{current})$ in PD control laws for stern propeller rpm, and bow and stern lateral thruster voltage. The stern propeller rpm control law is

$$rpm = Prop_{range} - Prop_{current} - Prop_{surge} \qquad \text{(Eq. 20)}$$

where

$$Prop_{range} = k_{prop-hover} d_{on-track} \qquad \text{(Eq. 21)}$$

$$Prop_{current} = k_{prop-current}(x_{current} \cdot \mathbf{cos}(\psi) + y_{current} \cdot \mathbf{sin}(\psi)) \qquad \text{(Eq. 22)}$$

and

$$Prop_{surge} = k_{surge} u \qquad \text{(Eq. 23)}$$

The bow and stern lateral thruster voltage control laws are

$$V_{bow} = -Thruster_{yaw} + Thruster_{range} + Thruster_{sway} - Thruster_{current} \quad \text{(Eq. 24)}$$

and

$$V_{stern} = Thruster_{yaw} + Thruster_{range} + Thruster_{sway} - Thruster_{current} \quad \text{(Eq. 25)}$$

where

$$Thruster_{yaw} = -k_{thruster-\psi}(\psi - \psi_{command}) - k_{thruster-r}r \quad \text{(Eq. 26)}$$

$$Thruster_{range} = k_{thruster-hover}d_{cross-track} \quad \text{(Eq. 27)}$$

$$Thruster_{sway} = k_{thruster-sway}v \quad \text{(Eq. 28)}$$

and

$$Thruster_{current} = k_{thruster-current}(x_{current} \cdot \mathbf{sin}(\psi) - y_{current} \cdot \mathbf{cos}\,\psi) \quad \text{(Eq. 29)}$$

Values and units for PD control constants are listed in Table 3.

Constant	Value	Units
$k_{prop-hover}$	200.0	rpm / ft
$k_{prop-current}$	6600.0	rpm-secs / ft
k_{surge}	2400.0	rpm-secs / ft
$k_{thruster-\psi}$	0.200	Volts / degrees
$k_{thruster-r}$	2.0	Volts-secs / degrees
$k_{thruster-hover}$	5.3333	Volts / ft
k_{sway}	20.0	Volts-secs / ft
$k_{thruster-current}$	40.0	Volts-secs / ft

Table 3. Station Keeping PD Control Law Constants.

3. AUV Tracking

Because of the speed and asynchronous nature of sonar-based target-position update rate, a method is required for computing *Phoenix* position and velocity between

updates. Over the long term the best solution is probably the incorporation of an inertial measurement unit (IMU) capable of providing position updates in real time [McGhee 95, Bachmann 96]. For the present however, *Phoenix* does not have installed hardware that can provide real-time navigational information. As a short-term solution, a simple mathematical model based on control inputs has been developed and incorporated to estimate *Phoenix* position and velocity between target updates [Marco 96a].

The mathematical model is a three DOF dead reckoning model that includes drag, added mass and steady state surge. Because *Phoenix* hardware includes a directional gyro that directly provides yaw and indirectly provides yaw rate (by differentiation of yaw), only the surge and sway equations of motion from [Marco 96a] are used. The surge and sway equations of motion are [Marco 96a]

$$M_x\dot{u}(t) + b_x u(t)|u(t)| = 2\alpha_x v_x(t)|v_x(t)| \qquad \text{(Eq. 30)}$$

$$M_y\dot{v}(t) + b_y v(t)|v(t)| = \alpha_y v_{blt}(t)|v_{blt}(t)| + \alpha_y v_{slt}(t)|v_{slt}(t)| \qquad \text{(Eq. 31)}$$

where M_x and M_y are the sum of mass and added mass in the x and y body axes, b_x and b_y are square-law damping coefficients, α_x and α_y are voltage-to-force coefficients and $v_x(t)$, $v_{blt}(t)$ and $v_{slt}(t)$ are terms for the voltage applied to the propellers, bow lateral thruster and stern lateral thruster. More specifically asymmetric voltage to the aft propellers is accounted for using [Marco 96a]

$$v_x(t)|v_x(t)| = \frac{v_{ls}(t)|v_{ls}(t)| + v_{rs}(t)|v_{rs}(t)|}{2} \qquad \text{(Eq. 32)}$$

where $v_{ls}(t)$ and $v_{rs}(t)$ are the voltages applied to the left and right propellers. Known ocean current is accounted for when converting body fixed rates to world rates:

$$\begin{pmatrix} \dot{x}(t) - x_{current} \\ \dot{y}(t) - y_{current} \end{pmatrix} = \begin{bmatrix} \mathbf{cos}(\psi) & -\mathbf{sin}(\psi) \\ \mathbf{sin}(\psi) & \mathbf{cos}(\psi) \end{bmatrix} \begin{pmatrix} u(t) \\ v(t) \end{pmatrix} \qquad \text{(Eq. 33)}$$

Values and units of constants used in the mathematical model are shown in Table 4.

Constant	Value	Units
M_x	214.29	Kg
M_y	350.70	Kg
b_x	63.80	Kg / m
b_y	815.40	Kg / m
α_x	0.056	N / Volts^2
α_y	0.018	N / Volts^2

Table 4. Mathematical Model Constants [Marco 96a].

While this mathematical model is simple enough to calculate in real time and accurate enough to compute reasonable navigational values, it is not perfect [Marco 96a]. Target position updates based on sonar data remain the most accurate means of calculating the location of the AUV relative to a target. Since the purpose of the mathematical model is to maintain AUV position and velocity information between sonar-based position updates, it is reset each time a position update is received. In this way incremental errors in the mathematical model are not permitted to build up to unacceptable values over time.

An important area for future work remains validation of AUV hydrodynamics coefficients. Since a general six-DOF hydrodynamics virtual world model for *Phoenix* can run in real time, more accurate on-board dead reckoning is possible [Brutzman 94].

D. FINAL RECOVERY CONTROL

The final addition to the execution level in support of this research was the implementation of a control mode to drive *Phoenix* into the recovery tube. As with the target-tracking sonar modes and station keeping control mode, the recovery control mode assumes that the position, orientation and size of the recovery tube has been determined by

the tactical level. The goal of the recovery control mode is to drive *Phoenix* a specified distance into a tube while maintaining adequate clearance from both sides.

Recovery control is initiated by the tactical level once *Phoenix* is directly in front of the recovery tube with its nose just inside. Upon recovery initiation the ST1000 sonar is switched to manual control and slews relative 75 degrees left. At the same time the tactical level sonar manager slews the ST725 sonar relative 75 degrees to the right. *Phoenix* positioning relative to the tube at this point is shown in Figure 19.

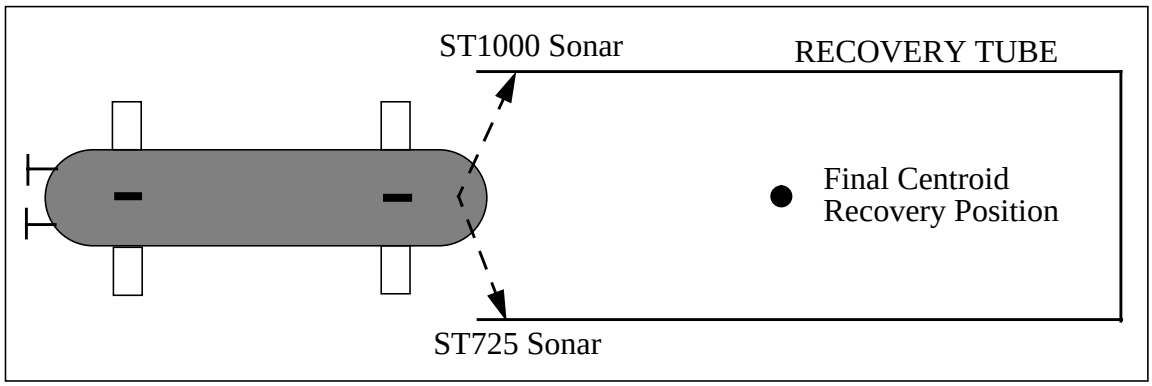


Figure 19: AUV and Recovery Tube Layout at Recovery Control Initiation.

PD control laws are then used to drive *Phoenix* into the tube. The mathematical model described in the previous section is used to estimate the distance travelled into the tube while the ST1000 and ST725 sonars are used to keep *Phoenix* in the center of the tube. The control law for the aft propellers is

$$rpm = k_{prop-range}d - k_{prop-current}(x_{current} \cdot \mathbf{cos}(\psi) + y_{current} \cdot \mathbf{sin}(\psi)) - k_{prop-surge}u \qquad \text{(Eq. 34)}$$

where d is the remaining distance into the tube as computed by the mathematical model. The control laws for the bow and stern lateral thrusters are

$$V_{bow} = -Thruster_{yaw} + Thruster_{range} - Thruster_{speed} - Thruster_{current} \qquad \text{(Eq. 35)}$$

and

$$V_{stern} = Thruster_{yaw} + Thruster_{range} - Thruster_{speed} - Thruster_{current} \qquad \text{(Eq. 36)}$$

where $Thruster_{yaw}$ and $Thruster_{current}$ are computed using Equations 26 and 29 respectively, and

$$Thruster_{range} = k_{thruster-range}(R_{ST725}\mathbf{sin}(75°) - R_{ST1000}\mathbf{sin}(75°)) \quad \text{(Eq. 37)}$$

and

$$Thruster_{speed} = k_{thruster-speed}\dot{R}_{ST1000}\mathbf{sin}(75°) \quad \text{(Eq. 38)}$$

where R_{ST1000} and R_{ST725} are the ST1000 and ST725 sonar ranges. These control laws are very similar to Equations 20, 24 and 25, differing primarily in the values of the control constants and how the individual terms are computed. Values of control constants are listed in Table 5.

Constant	Value	Units
$k_{prop-range}$	200.0	rpm / ft
$k_{prop-surge}$	6000.0	rpm-secs / ft
$k_{prop-current}$	6600.0	rpm-secs / ft
$k_{thruster-\psi}$	0.60	Volts / degrees
$k_{thruster-r}$	8.0	Volts-secs / degrees
$k_{thruster-range}$	8.0	Volts / ft
$k_{thruster-speed}$	40.0	Volts-secs / ft
$k_{thruster-current}$	40.0	Volts-secs / ft

Table 5. Recovery Control PD Control Constants.

E. SUMMARY

This chapter covers implementation of features at the execution level of *Phoenix* software architecture to support recovery operations. Robust sonar behaviors are implemented including modes to support manual control, forward scanning, target search, target tracking and target-edge tracking. These behaviors are used to support a *Phoenix*

control mode capable of transiting to and maintaining a commanded range and bearing from a sonar target. PD control laws are used to control motion relative to the target. Additionally, because of the asynchronous target-position update rate, a mathematical model was developed to estimate *Phoenix* motion between sonar-based target updates. Finally, a control mode was implemented to actually drive *Phoenix* into the recovery tube once the vehicle obtains a position immediately in front of the tube. This control mode uses PD control laws very similar to those used for station keeping. The ST1000 and ST725 sonars are used to ensure clearance from the sides of the recovery tube throughout the recovery evolution while the mathematical model is used to estimate forward travel into the tube.

The following chapter of this thesis covers implementation of features at *Phoenix*' tactical level that use the behaviors described in this chapter to control recovery. Significant issues in that chapter are recovery path planning and command generation. In addition, the mathematical structures used to implement path planning are discussed.

V. TACTICAL LEVEL IMPLEMENTATION

A. INTRODUCTION

One of the primary responsibilities of the tactical level software is to use the low-level functionality of the execution level in such a way as to accomplish the high-level goals of the strategic level. Specifically, this chapter will cover how the tactical level uses the edge-tracking sonar behavior and the station-keeping control available at the execution level to support vehicle recovery in a tube.

The second responsibility of the tactical level that directly relates to recovery is the identification and localization of the recovery tube. The ST725 and ST1000 sonars are the primary on-board sensors upon which this task depends. Real-time sonar classification using both of these sonars has been the subject of other research and is not directly addressed here. For more information concerning research in this area involving *Phoenix* refer to [Brutzman 92], [Campbell 96] and [Marco 96a]. A major assumption of the research of this thesis is that the recovery tube is at a known position and orientation.

The first section of this chapter discusses tactical-level planning of the recovery path. The mathematical structures used to implement recovery path planning are covered as well as the planning algorithm. The other major topic of this chapter is the generation of execution-level commands necessary for following the planned path.

B. RECOVERY PATH PLANNING

1. Transformations

a. Description

The mathematical structure used for recovery path planning is called a *transformation*. A transformation is a means of representing an object's position and orientation in two dimensions (2D) and takes the form of a state vector consisting of x

position, y position and orientation. The coordinate system used for a transformation is arbitrary and can represent an object's global position or its position relative to another object. [Kanayama 96]

In addition to representing an object's position and orientation, a transformation can be used to represent discrete motions with 2D translations and rotations specified in body-fixed coordinates. Finally, transformations are useful for defining lines and circles. Lines are specified by a position (which can be any point on the line) and an orientation. This representation of a line is convenient for path planning because it includes a direction which will generally represent the direction of motion along the line. The transformation portion of a circle representation consists of a point on the circle and the tangential direction of the circle at that point. A circle requires a fourth term representing the curvature of the circle. [Kanayama 96] Representation of lines and circles using transformations is covered in more detail in the following section.

b. Operations

There are two operations defined for transformations: *composition* and *inversion* [Kanayama 96]. Composition is a means of combining two transformations. Typically the first transformation represents a position and orientation, and the second represents a motion or a relative position. The result of a composition is the final global position of an object moved from a position (represented by the first transformation) by a change in position and orientation (represented by the second transformation). Composition is defined as [Kanayama 96]

$$\begin{pmatrix} x_1 \\ y_1 \\ \theta_1 \end{pmatrix} \bullet \begin{pmatrix} x_2 \\ y_2 \\ \theta_2 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \cdot \mathbf{cos}(\theta_1) - y_2 \cdot \mathbf{sin}(\theta_1) \\ y_1 + x_2 \cdot \mathbf{sin}(\theta_1) + y_2 \cdot \mathbf{cos}(\theta_1) \\ \theta_1 + \theta_2 \end{pmatrix} \qquad \text{(Eq. 39)}$$

This definition leads to the definition of the identity transformation (e). The identity transformation is defined as $(0, 0, 0)^T$ and has the following result when used in compositions [Kanayama 96]:

$$q \bullet e = e \bullet q = q \qquad \text{(Eq. 40)}$$

The definition of e leads to the definition of the inverse function. The inverse function for transformations is defined by [Kanayama 96]

$$q \bullet q^{-1} = q^{-1} \bullet q = e \qquad \text{(Eq. 41)}$$

and for $q = (x, y, \theta)^T$ the inverse, q^{-1}, is given by the equation [Kanayama 96]

$$q^{-1} = \begin{pmatrix} -x \cdot \mathbf{cos}(\theta) - y \cdot \mathbf{sin}(\theta) \\ x \cdot \mathbf{sin}(\theta) - y \cdot \mathbf{cos}\theta \\ -\theta \end{pmatrix} \qquad \text{(Eq. 42)}$$

Transformation composition can also be used to generate smooth trajectories in a plane by composition of a position and orientation with transformations representing small discrete motions. The transformation representing the motion is referred to as a *circular transformation*. A circular transformation is derived using the length of the motion (l) and the amount of change in orientation over that length (α). The circular transformation is computed as [Kanayama 96]

$$\Delta q(l,\alpha) = \begin{pmatrix} \frac{\mathbf{sin}(\alpha)}{\alpha} l \\ \frac{1 - \mathbf{cos}(\alpha)}{\alpha} l \\ \alpha \end{pmatrix} \qquad \text{(Eq. 43)}$$

For linear motions ($\alpha = 0$) this equation is undefined but can be approximated using a Taylor expansion resulting in [Kanayama 96]

$$\Delta q(l,\alpha) = \begin{pmatrix} (1 - \alpha^2/3! + \alpha^4/5! - \ldots) l \\ (1/2! - \alpha^2/4! + \alpha^4/6! - \ldots)\alpha l \\ \alpha \end{pmatrix} \qquad \text{(Eq. 44)}$$

A series of small discrete motions in the form of circular transformations is capable of approximating a continuous smooth path. As with other discrete approximations of continuous functions, smaller circular transformations will result in more accurate path approximation.

2. Line and Circle Tracking

a. Lines and Circles

As stated in the previous section, lines and circles can be specified using transformations. Representation of a line takes the form $(x, y, \theta)^T$ where (x, y) is any point on the line and θ is the orientation of the line. Since any point on the line can be used in the transformation representing a line, an infinite number of transformations are possible for representation of a single line. For this reason representation of lines using transformations is probably inappropriate if lines are to be compared. Since the recovery path planning involved in this research does not involve comparison of different paths, the inability to compare lines for equivalence does not pose a problem.

Representation of circles using transformations is only slightly more complex than representation of lines. Circle representation takes the form $(x, y, \theta, \kappa)^T$ where (x, y) is any point on the edge of the circle, θ is the tangential orientation of the circle at (x, y), and κ is the curvature of the circle defined as [Kanayama 96]

$$\kappa = \frac{d\theta}{ds} \qquad \text{(Eq. 45)}$$

where s is the distance along the edge of the circle. Representation of circles using transformations has similar advantages and disadvantages as representation of lines. The most significant advantage is that by specifying a tangential orientation it is possible to implicitly represent the direction of desired motion when traveling along a circular path (e.g., using θ). The disadvantage is that there are an infinite number of transformation

representations for a single circle. Again, since recovery path planning does not involve the comparison of circles, this potential disadvantage is not relevant here.

b. The Steering Function and Smooth Path Planning

While line and circle segments can be used to represent a desired path, representation of the path is only half the problem. The second problem is actually steering a vehicle (real or simulated) towards and along the desired path. This is the role of the steering function. The steering function is a continuous function based on the vehicle's current state and the desired path [Kanayama 96]. Vehicle state includes a transformation to represent vehicle position and orientation and a fourth term to represent the curvature of the vehicle's path. The steering function is used to adjust the derivative of this fourth term to move the vehicle towards and along the desired path. The steering function is given by [Kanayama 96]

$$\frac{d\kappa}{ds} = -(a(\kappa - \kappa_d) + b(\theta - \theta_d) + c\Delta d) \qquad \text{(Eq. 46)}$$

where κ and θ are the vehicle's current path curvature and orientation, κ_d and θ_d are the vehicle's desired path curvature and orientation, Δd is the signed distance of the vehicle from the desired path and a, b and c are constants. Critically damped values for a, b and c (values that will result in at most one overshoot) are computed as [Kanayama 96]

$$a = \frac{3}{\sigma} \qquad \text{(Eq. 47)}$$

$$b = \frac{3}{\sigma^2} \qquad \text{(Eq. 48)}$$

$$c = \frac{1}{\sigma^3} \qquad \text{(Eq. 49)}$$

where σ is an arbitrary positive constant corresponding to the vehicle's desired responsiveness. Lower values of σ will cause the vehicle to steer more sharply towards the desired path while larger values will cause a smoother path but a slower convergence with

the desired path. Figure 20 shows an illustration of a path tracking problem. As can be seen in the illustration, the steering function must be able to not only maintain the vehicle on the desired path but steer the vehicle towards the path if necessary.

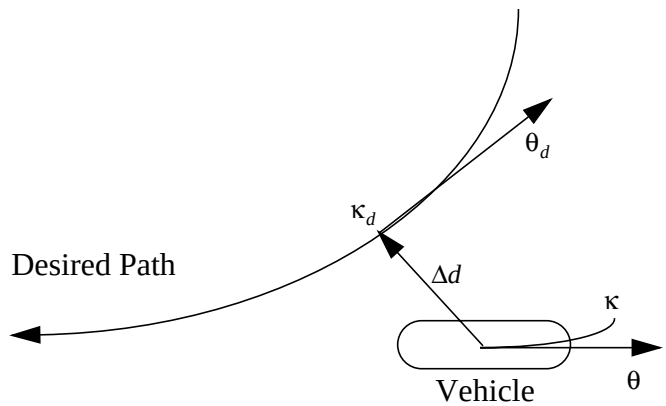


Figure 20: Steering Function Terms [Kanayama 96].

When a vehicle defined by $(x, y, \theta, \kappa)^T$ is tracking a line defined by $(x_0, y_0, \theta_0)^T$, κ_d is zero, θ_d is θ_0 and Δd is computed as [Kanayama 96]

$$-(x - x_0)\,\mathbf{sin}(\theta_0) + (y - y_0)\,\mathbf{cos}(\theta_0) \qquad \text{(Eq. 50)}$$

For the same vehicle tracking a circle defined by $(x_0, y_0, \theta_0, \kappa_0)^T$, κ_d is κ_0. θ_d and Δd are computed as [Kanayama 96]

$$\theta_d = \mathbf{atan}(\mathbf{sin}(\theta_0) + \kappa_0 \cdot (x - x_0), (\mathbf{cos}(\theta_0) - \kappa_0 \cdot (y - y_0))) \qquad \text{(Eq. 51)}$$

and

$$\Delta d = \frac{-(x - x_0)(\kappa_0 \cdot (x - x_0) + 2\,\mathbf{sin}(\theta_0)) - (y - y_0)(\kappa_0 \cdot (y - y_0) - 2\,\mathbf{cos}(\theta_0))}{1 + \sqrt{(\kappa_0 \cdot (x - x_0) + \mathbf{sin}(\theta_0))^2 + (\kappa_0 \cdot (y - y_0) - \mathbf{cos}(\theta_0))^2}} \qquad \text{(Eq. 52)}$$

respectively.

Circular transformations are used along with the steering function to incrementally steer the vehicle along the desired path. At each iteration the steering

function is used to compute $\frac{d\kappa}{ds}$. The vehicle's new position and orientation is then computed as

$$q_{new} = q \bullet \Delta q\left(\Delta s, \frac{d\kappa}{ds} \cdot \Delta s\right) \qquad \text{(Eq. 53)}$$

where q is the transformation representing the vehicle position and orientation at the beginning of the iteration and Δs is the circular distance traveled in each iteration. The updated value for κ is computed as

$$\kappa_{new} = \kappa + \frac{d\kappa}{ds} \cdot \Delta s \qquad \text{(Eq. 54)}$$

Figure 21 shows the track of a simulated vehicle steered using this method. The desired path of the figure consists of two line segments and a circle segment.

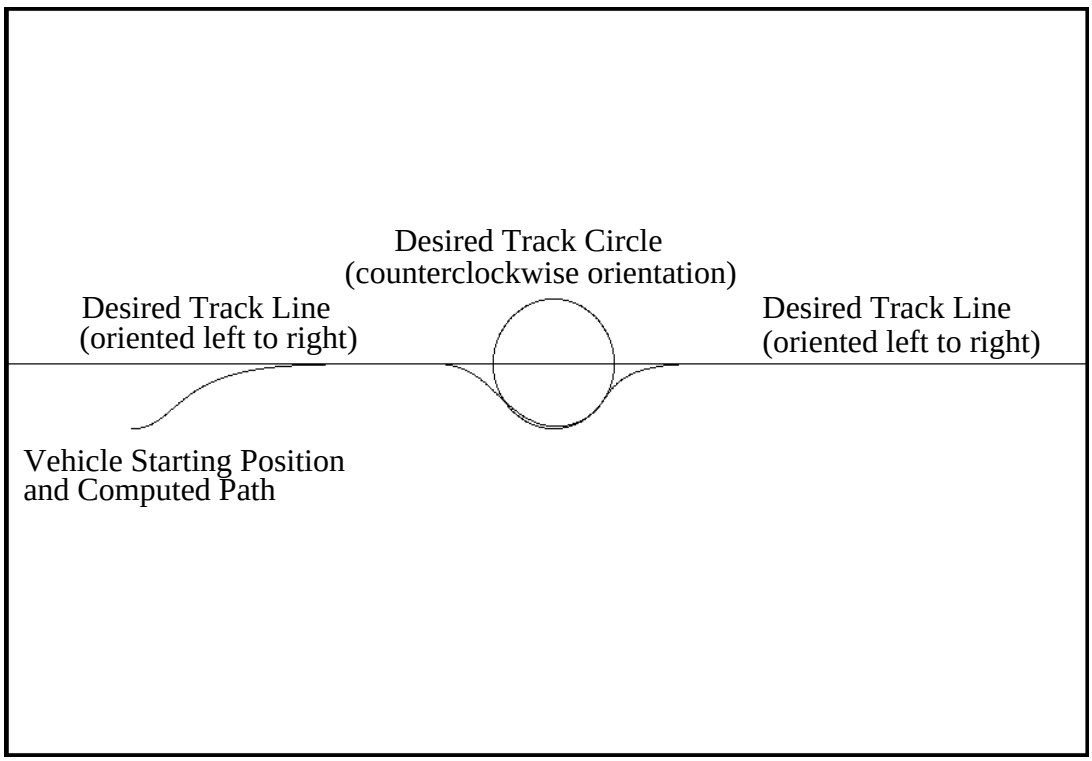


Figure 21: Tracking to a Desired Path Using the Steering Function.

3. Recovery Planning

a. Overview

While *Phoenix* is capable of six-DOF motions, recovery path planning is conducted in two dimensions in order to allow use of the methodology described above. The use of only two dimensions places two limitations upon recovery: vehicle depth and pitch control must be handled independently, and recovery is only possible in a horizontally level recovery tube. Presently these limitations are not considered significant, however future work may include the expansion of these algorithms to take advantage of *Phoenix*' six-DOF capability to support recovery in tubes of arbitrary orientation.

The steering function derived above is intended primarily for vehicles restricted to arbitrary tangential motions [Kanayama 96]. Such vehicles are typically incapable of lateral motion but are assumed to be capable of following a path of unlimited curvature. Since the steering function is being used only for motion planning and not for motion control, the steering function remains appropriate for recovery path planning even though *Phoenix* is capable of nontangential motions. In this implementation a planning vehicle that is restricted to tangential motions is used to generate a smooth path. The initial position of the virtual vehicle is set to *Phoenix*' position at the start of the recovery evolution while the initial orientation of the virtual vehicle points directly at the center of the recovery tube (unless *Phoenix* is too close to the tube in which case it points directly away from the center). The steering function is then used to drive the virtual vehicle around and into the recovery tube to generate the recovery path. During the actual recovery *Phoenix* must attempt to stay on the planned path but is not limited solely to tangential motions.

Another issue concerning the use of this methodology for AUV path planning is dealing with unintentional sideslip. While it is reasonable to assume that the

velocity vector of a wheeled vehicle will be aligned with the longitudinal axis, the same cannot be said for vehicle's such as *Phoenix*. Not only are lateral velocity components possible, they are in large part unavoidable. Figure 22 shows the geometry involved in this type of *holonomic* system. In the figure, ψ represents vehicle heading, β represents vehicle sideslip angle, and ϑ is the velocity vector orientation, while u and v are components of the velocity vector in vehicle coordinates.

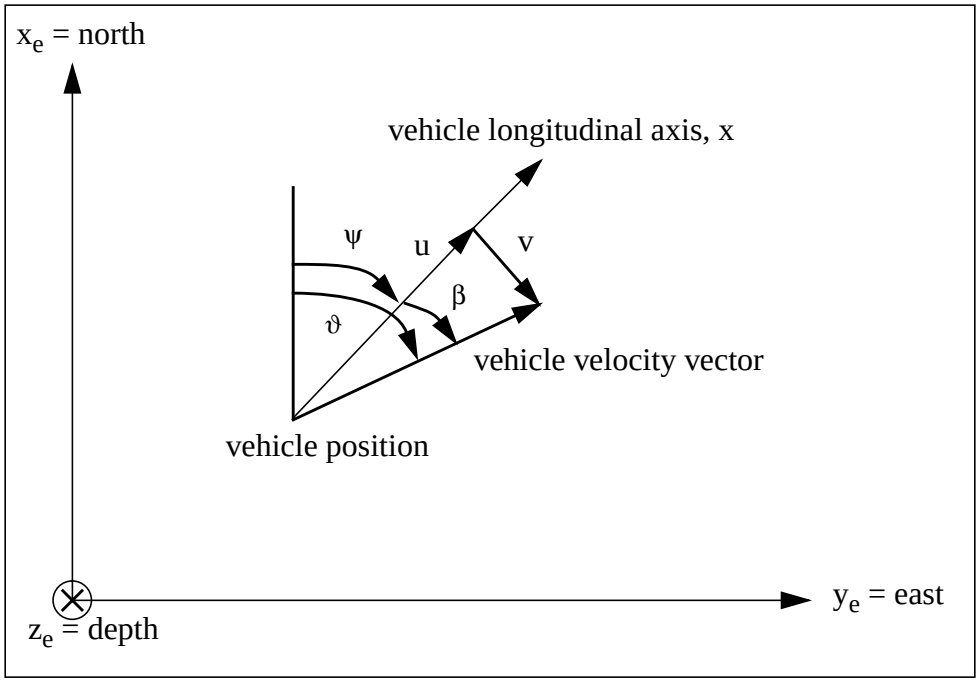


Figure 22: Holonomic System Geometry [McGhee 91].

The lateral component of *Phoenix* velocity vector can be partially controlled using lateral thrusters. A portion of the lateral velocity, however is dependent on the longitudinal velocity and the turn rate. While rigorous modeling of this phenomenon can become extremely complex, a fairly simple model can be used to predict sideslip angle. This model results in the equation [McGhee 91]

$$\dot{\vartheta} + \frac{2m\vartheta}{\rho AVC_\beta} = \psi \qquad \text{(Eq. 55)}$$

where m is the vehicle mass, ρ is the density of the medium, A is the lateral surface area of the vehicle, V is the magnitude of the vehicle velocity (including sideslip), and C_β is a constant relating to lift forces generated as a result of sideslip. Using this equation, an estimate of sideslip angle can be maintained as part of the vehicle state. Computer simulation indicates that mathematical modeling of sideslip in this manner is particularly useful during waypoint navigation [Davis 95].

Because of the low velocities and turn rates involved during hover and recovery operations, the uncommanded sideslip angle is small when compared to commanded sideslip induced by the lateral thrusters. Since the larger term dominates the smaller, it is safe in the tube recovery scenario to ignore uncommanded sideslip. Additionally, errors due to miscalculation of sideslip of other six-DOF holonomic effects due to added mass and other cross-coupled hydrodynamic drag forces are not allowed to accumulate during execution because of the frequent recalculation of the AUV position relative to the recovery tube.

b. Desired Path Planning

For overall recovery planning purposes, the area surrounding the recovery tube is divided into nine regions. Each region corresponds to the Voronoi region of a segment or corner of the tube [Kanayama 96]. A line or circle representing a desired path is defined for each region. With the exception of the line representing the final tube entry path, the desired path circles and lines maintain a constant safe standoff distance of six feet from the tube. Additionally, all lines and circles are directed towards the opening of the recovery tube. The transformation representations of the desired path lines and circles are computed as soon as the position and orientation of the recovery tube are known. An example of tube regions and desired paths is shown in Figure 23.

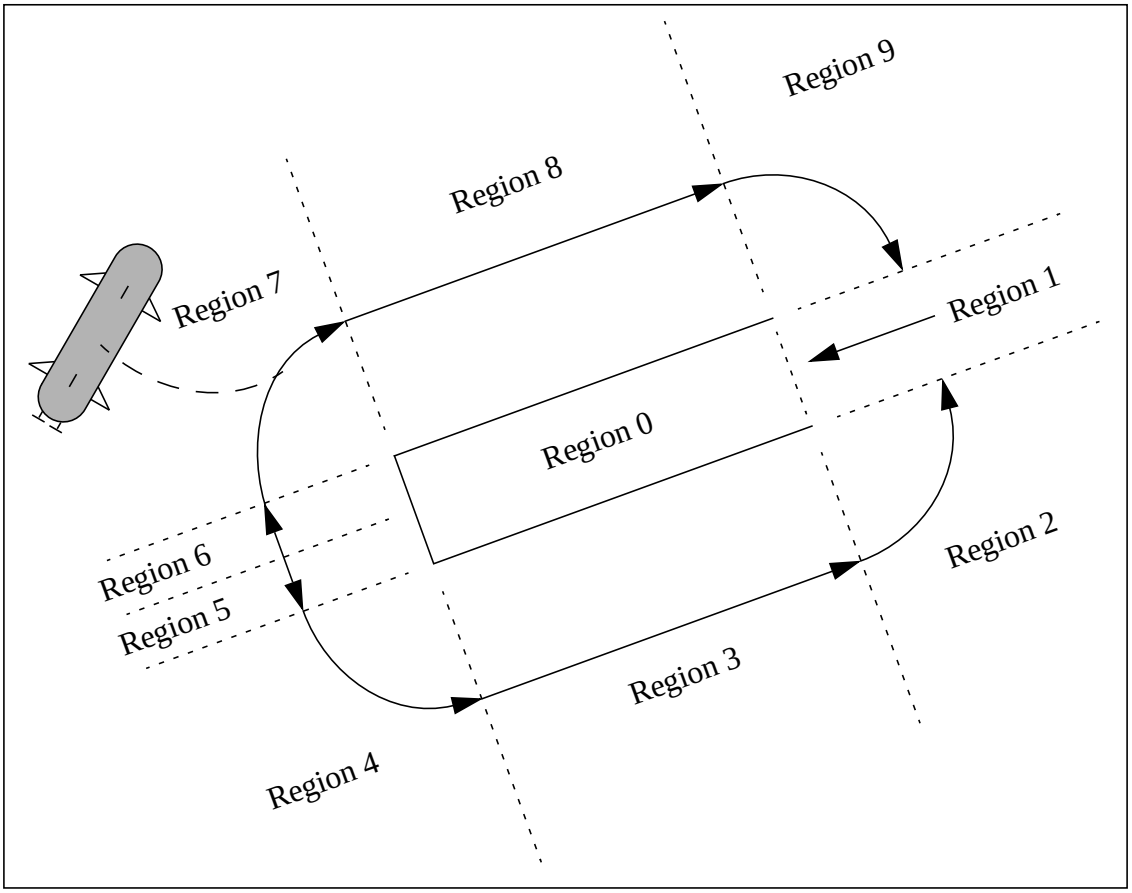


Figure 23: Voronoi-Based Recovery Regions and Path Planning Segments.

The next step is determining which region *Phoenix* is in at the beginning of the recovery evolution. Since the size, shape, position and orientation of the tube are known, this is simply a matter of computing the ranges from *Phoenix* to the different segments and corners of the tube and determining which is closest. After deciding which region the AUV is starting the recovery from, the planning vehicle is instantiated and incrementally moved towards the desired path for the region using the steering function. As the planning vehicle leaves one region and enters another, the desired path for the new region is used. The planning vehicle has left one region and entered another when the distance from the vehicle to the corner or segment defining the current region is greater than the distance of the vehicle to the corner or segment defining the new region. This process

continues until the planning vehicle has entered the tube. The path that the planning vehicle travelled represents the planned recovery path for the actual AUV.

Again, since *Phoenix* is capable of nontangential motion, neither θ_d in the steering function nor θ of the planning vehicle necessarily correspond to the desired orientation of *Phoenix* during the recovery. In fact, in order to facilitate continuous sonar contact, *Phoenix* will normally point directly at the portion of the recovery tube upon which it is taking station. This vehicle orientation policy has an exception in the final recovery phase when the AUV will be aligned with the recovery tube (although θ_d and θ still bear no correlation to desired AUV orientation). Thus θ and θ_d pertain to the tangential orientation of the track the AUV is to follow, while actual vehicle heading is determined by the relative bearing to the sonar tracking landmark. Precise six-DOF maneuverability and control of posture using the nontangential motion capabilities of *Phoenix* permit such a decoupling between vehicle track and vehicle orientation.

C. EXECUTION COMMAND GENERATION

Which corner to use for generated station-keeping commands depends on the recovery region that the planning vehicle is in when the command is generated. The corner must be visible from anywhere within the region and the AUV sonar routines must be able to recognize the edge. Since the target-search and edge-tracking sonar modes use range information to recognize targets, there must be a significant increase in range as the sonar scans past the corner. Figure 24 shows which corners are used for station-keeping command generation for the different regions.

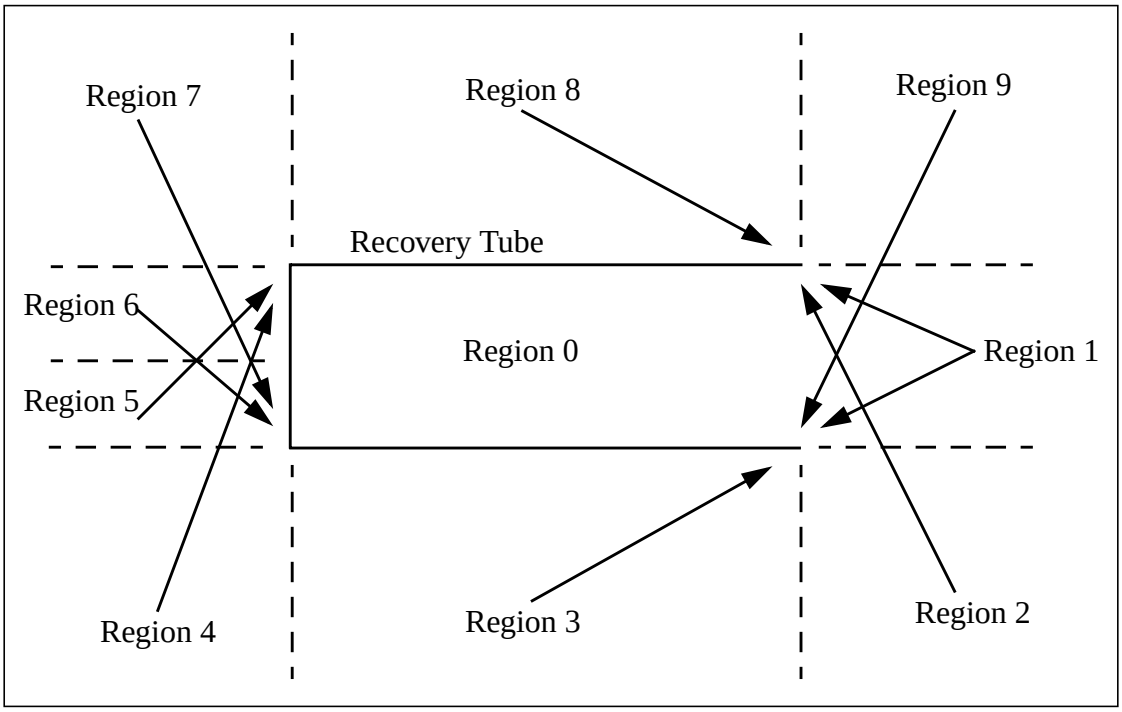


Figure 24: Recovery Regions and Station-Keeping Corner Assignments.

At predetermined intervals along the planning vehicle's path, execution-level commands are generated and stored in a file. Generated commands invoke the execution level's edge-tracking sonar mode and station-keeping control mode. Commanded stations in each Voronoi region are in the form of range and bearing from the planning vehicle's current location to the appropriate tube corner that *Phoenix*' ST1000 sonar is most likely to be able to discriminate. Interestingly, it must be noted that commands for the entire recovery are generated before any command is issued to the execution level. Upon completion of the recovery through the appropriate Voronoi regions plan the OOD module will dequeue and issue the generated commands one at a time.

The final command that is generated is the recovery command. When issued to the execution level, this command will invoke *Phoenix* recovery control mode. The recovery

command will be generated immediately after the planning vehicle has entered the recovery tube opening.

An example of recovery planning and virtual world results are shown in Figures 25 and 26. Figure 25 shows the execution-level commands generated for use during the recovery while Figure 26 shows an x-y plot of the recovery tube, the planned path, and the actual path followed by *Phoenix* in a virtual world test. The running of this and other test missions is discussed in Chapter VII.

```
#RECOVERY REGION 7
EDGE-STATION 6.231679 109.801091 6.231679 104.801091
EDGE-STATION 8.541297 115.850068
EDGE-STATION 9.411731 133.894357

#RECOVERY REGION 8
EDGE-STATION 11.636344 70.631182 11.636344 75.631182
EDGE-STATION 7.209702 101.266115
EDGE-STATION 5.999095 134.868091

#RECOVERY REGION 9
EDGE-STATION 9.332900 148.086638 9.332900 153.086638
EDGE-STATION 8.587834 171.619319
EDGE-STATION 7.367044 -168.706232 -135.000000

#RECOVERY REGION 1
EDGE-STATION 4.239524 -166.878360 -135.000000
EDGE-STATION 4.239524 -166.878360 -135.000000
EDGE-STATION 3.240133 -174.693612 -135.000000

#FINAL TUBE ENTRY
ENTER-TUBE 7.499992 -135.000000
```

Figure 25: Generated Commands Based on a Recovery Plan.

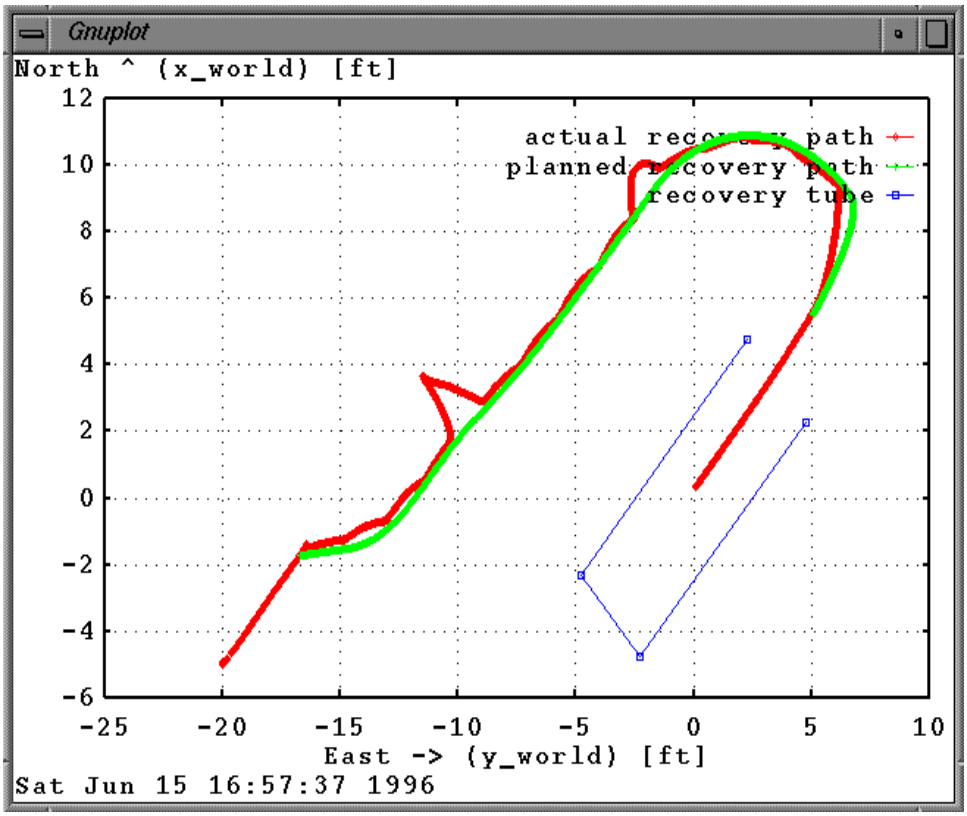


Figure 26: Planned and Actual Recovery Path Results from a UVW Mission.

D. SUMMARY

Implementation of features at the tactical level in support of recovery include recovery path planning and command generation. Recovery path planning utilizes a mathematical structure called a transformation which is used to represent vehicle position and orientation and discrete motions in two dimensions. The planned recovery path is generated by a planning vehicle which is driven by a steering function from *Phoenix* position at the start of the tube recovery evolution.

The area surrounding the recovery tube is divided into nine Voronoi regions, each of which has an associated desired path. As the planning vehicle passes through each region, the steering function drives the vehicle towards the desired path for that region. A

corresponding tube corner is chosen for optimal sonar discrimination while tracking. The path traveled by the planning vehicle becomes the planned path for *Phoenix* during the actual recovery maneuver.

At predetermined intervals along the planning vehicle's path, execution-level commands are generated and stored in a file. These commands are later issued to the execution-level one at a time by the OOD module. The commands use the execution level's station-keeping behavior to follow the planned path. When the planning vehicle has entered the recovery tube's opening, path planning is complete. A recovery command is then generated that will invoke the execution level's recovery control mode for actual entry into the recovery tube.

The following chapter discusses strategic level issues dealt with in the conduct of this research. Research at this level focuses primarily on mission specification, planning and generation. Specific issues include evolution of the strategic level and the development of a mission planning expert system.

VI. STRATEGIC LEVEL IMPLEMENTATION

A. INTRODUCTION

Since the strategic level of RBM is responsible only for high-level mission control, its responsibilities regarding recovery are few. Essentially, the strategic level is responsible only for deciding where and when the recovery is to take place, and what type of recovery is required. This role is analogous to that of a ship's commanding officer who specifies what port to go to, when to go there, and whether to anchor or dock, but is not physically involved in the actual anchoring or docking evolution.

Because of the limited role of the strategic level in recovery, research conducted at this RBM level has been more general in nature. The most significant result has been to simplify the process of strategic level mission planning and generation. The following section of this chapter describes implementation of features at the strategic level that facilitate this goal. The subsequent section describes the implementation of a graphical expert system for mission planning and automatic strategic-level code generation.

B. EVOLUTION OF THE STRATEGIC LEVEL

1. Mission Control

As stated previously the strategic level is structured as a DFA and consists of three software pieces: the DFA, the mission controller and a set of primitive goals. The mission controller is shown in Figure 27 implemented equivalently in Prolog and C++. Looping in the Prolog implementation is conducted using the basic Prolog backtracking control algorithm which tries to "prove" predicates [Rowe 88]. When a mission is initiated, Prolog tries to find a way to make the execute_phase predicate "true" by proving the execute_phase and mission_done predicates. If the execute_phase predicate is false, the phase has not yet completed. In this situation Prolog will backtrack into the repeat

predicate (which is always considered true). It then retries the execute_phase predicate. This looping pattern will continue until the execute_phase predicate becomes true, at which point the same process is executed for the mission_done predicate. When the mission_done predicate is proven, execute_mission is proven, and the mission completes. Otherwise, Prolog tries to prove the next_phase predicate. Which version of this predicate can be proven is determined by success or failure of the current phase. In contrast to the Prolog mission controller, the C++ mission controller uses a typical imperative programming language loop to obtain behavior equivalent to that of the Prolog version.

```
execute_mission :- asserta(current_phase(initialize)),
                   repeat, execute_phase, mission_done.
execute_phase   :- current_phase(X), execute_phase(X),
                    next_phase(X), !.
mission_done    :- current_phase(mission_abort).
mission_done    :- current_phase(mission_complete).
```

(a)

```
currentPhase = initialize ();
do {
   if (currentPhase->complete ())
   {
      currentPhase = currentPhase->completeSuccessor;
      currentPhase->initiate ();
   }
   else if (currentPhase->abort())
   {
      currentPhase = currentPhase->abortSuccessor;
      currentPhase->initiate ();
   }
} while ((currentPhase != missionAbort) &&
         (currentPhase != missionComplete));
```

(b)

Figure 27: Strategic Level Mission Controller in (a) Prolog and (b) C++.

2. Abstract Mission Control

Since initial *Phoenix* research was focused primarily on the strategic and execution levels of RBM, early versions of the tactical level were greatly simplified and mainly responsible for simply relaying commands from the strategic level to the execution level [Marco 96b]. Consequently many tasks appropriate for the tactical level were first implemented at the execution and strategic levels. Recent improvements in the tactical level now handle many of the tasks previously divided between the strategic and execution levels [Leonhardt 96, Campbell 96, McClarin 96, Scrivener 96]. This redistribution of responsibility among the levels of *Phoenix* RBM implementation has allowed strategic-level functionality to concentrate solely on the high-level mission control for which it was originally intended.

With the reassignment of many tasks to the tactical level, it became apparent that further strategic-level simplification was possible by limiting the allowable phase types to a few generic types. In fact this limitation was necessary since the tactical level is only capable of interpreting strategic-level commands from a predetermined set of primitive goals [Marco 96b, Leonhardt 96]. As the AUV's functionality evolves, new commands can be implemented in tandem at the strategic and tactical levels by adding to the vehicle's primitive goal set. Present strategic-level primitive goals include transits, searches, global positioning system (GPS) fixes, dives and hovers. Because of the explicit definition of all possible strategic-level primitive goals and the implementation of a robust tactical level, the RBM implementation of *Phoenix* is now versatile and simple enough to correctly perform a wide array of missions [Brutzman 96].

As stated in Chapter II, the strategic level does not perform any numerical computation [Byrnes 96], but the exclusive maintenance of numerical data at the lower RBM layers proved impractical in implementation. This was due to the high likelihood of

mismatches between the strategic level DFA and the numerical data file used by the tactical level [Leonhardt 96]. The solution was to include numbers upon which a phase is dependent (such as the location of a search) in the command sent to the tactical level. The tactical level interprets the parameters as numerical values, but at the strategic level they are just place holders within the command. The implementation of phase parameters eliminates the possibility of data file/DFA mismatch errors while maintaining the overall non-numerical nature of the strategic level. Figure 28 shows a strategic-level search phase with phase parameters defined in Prolog. Tactical-level replies to strategic-level commands are not tested so a sequence of strategic-level commands can initiate parallel tasks at the tactical level. Replies to strategic-level queries on the other hand, are tested so that execution does not proceed to the next query or command until an appropriate reply to the current query is received. As with the Prolog version of the mission controller, backtracking is used to implement looping behavior so that a phase will continue to execute until the execute_phase predicate is true. The phase depicted corresponds to the search phase of the mission depicted in Figure 11.

```
execute_phase(search_1) :-   ood("sonar_search 20 45 3",Reply),
                             ood("start_timer 250",Reply),
                             repeat, phase_completed(search_1).
phase_completed(search_1) :- ood("ask_search_complete",Reply),
                             Reply==1, asserta(succeed(search_1)).
phase_completed(search_1) :- ood("ask_time_out",Reply), Reply==1,
                             asserta(abort(search_1)).
next_phase(search_1) :-      succeed(search_1),
                             retract(current_phase(search_1)),
                             asserta(current_phase(return_to_base)).
next_phase(search_1) :-      abort(search_1),
                             retract(current_phase(search_1)),
                             asserta(current_phase(go_shallow)).
```

Figure 28: Strategic Level Phase Specified in Prolog.

While the primary goal of recent strategic-level research efforts has been the effective implementation of RBM in the real world [Brutzman 96], a collateral result has been the standardization of the strategic level. The strategic level code now has a standard form for a given type of phase. The only difference between two distinct phases of the same type is the parameters. A template can therefore be created for each type of phase. Strategic level code for a phase can be easily generated by inserting a label and phase parameters into a copy of the appropriate phase template. The boldface portions of the code fragment of Figure 28 indicate the phase label and parameters inserted into a sonar-search template. Using templates to code the strategic level has a number of advantages. For instance, the potential for syntactic programming errors is great when manually programming even a simple mission. Such errors can be virtually eliminated by utilizing templates. Furthermore phase templates make it possible to automate strategic level code generation and eliminate manual programming at the strategic level altogether [Leonhardt 96, Brutzman 96].

3. Programming Language Issues

A decision was made early in the development of mission-control software for the *Phoenix* AUV to implement the strategic level using the Prolog programming language. Because of its roots in predicate calculus, one advantage of Prolog is that it is relatively easy to use for specifying mission logic when compared to more common imperative languages. As a result, programs written in Prolog are typically shorter than equivalent programs written in functional or imperative languages. Additionally, programming of the strategic level of the RBM is primarily a symbolic programming problem which is well suited to expression in Prolog [Byrnes 96]. Finally, use of the Prolog inference engine is powerful since the current state of the DFA can be represented implicitly by the current rule that is being resolved [Byrnes 96]. However, in the current strategic level implementation,

the DFA state is maintained explicitly (using dynamically asserted facts) rather than implicitly in order to improve code readability and ease of use. This approach amounts to specializing the Prolog inference engine to a mission control engine or "mission controller" [Marco 96b].

A disadvantage of using Prolog at the strategic level is that it must interface with the tactical level which is currently written in C. At present there is no standard Prolog foreign language interface, so communication between the strategic and tactical levels is dependent on the vendor and version of the Prolog compiler used [Quintus Corporation 95]. Portability of the software system to new platforms is therefore a problem. Another disadvantage as missions become more complex is that the size of the Prolog program grows rapidly since each phase is programmed independently of all other phases. Finally, because of its reliance on backtracking for control of execution, Prolog tends to run more slowly than imperative languages [Rowe 88]. To date this has not been a problem since the speed at which the whole RBM system runs has been limited not by the speed of the strategic and tactical levels, but rather by the speed of the execution level [Leonhardt 96, Burns 96].

The advantages of using Prolog for *Phoenix* currently outweigh the disadvantages, particularly given the mission planning expert system described in the following section. However other programming languages have advantages which may make them attractive for use in the future. Two strategic levels equivalent to the one described above have been implemented using the Lisp and CLIPS programming languages [Byrnes 96]. However these implementations have proved to be much harder to write and understand.

More recently, research has been conducted into implementation of the RBM strategic level in C++ using object-oriented programming techniques. The polymorphism and inheritance characteristics of C++ classes allow the definition of a generic phase class from which more specific phase classes representing all allowable types of phases can

inherit. All phase class definitions together determine the vehicle's operational capabilities. A specific mission can be generated by instantiating instances of the appropriate phase classes and using pointers to connect them into a graph representing the strategic level DFA. As shown in Figure 27 this mission controller portion of the strategic level is implemented using a loop that queries the tactical level about the status of the current phase. If the current phase has either completed or aborted, the appropriate transition is executed by following a pointer and initiating the next phase. If the current phase has neither completed nor aborted, the loop repeats without initiating a new phase. Implementation of the strategic level in C++ directly addresses all three of the previously mentioned disadvantages of the Prolog strategic level (foreign language interface, size and speed). Since C++ can be directly linked with C functions, the system is inherently more portable than the Prolog version. Additionally, with the exception of individual phase instantiation and DFA construction, all code is contained within the phase class definitions. Therefore, as mission complexity increases, the size of a strategic level source program does not increase as rapidly as an equivalent mission written using Prolog. In the current C++ implementation, the size of the source program will typically increase by two lines for each additional phase (one line to instantiate the phase object, one line to link the object into the DFA graph). On the other hand, the very conciseness of this approach tends to present a barrier to easy understanding of the meaning and behavior of the vehicle in executing a mission so encoded. This difficulty is resolved by the development of a mission-generation expert system as explained in the following section.

C. A MISSION-GENERATION EXPERT SYSTEM

1. Introduction

In most scenarios involving the use of *Phoenix* class AUVs for mine countermeasure missions, operational naval personnel would be responsible for generation

of mission control software. While these individuals can be expected to be experts in anti-mine warfare, they are unlikely to have a high skill level in computer programming. Instead, they will probably require an easy-to-use mission programming interface in order to effectively and reliably specify an AUV mission. In this regard such individuals are probably typical of end users of autonomous vehicles in general. [Brutzman 96]

In order to facilitate ease of use, an expert system for programming *Phoenix* AUV missions has been developed. This expert system consists of three distinct subsystems and a graphical user interface (GUI). The first subsystem is used to automatically generate missions by specification of overall mission goals. The second subsystem is a mission-specification facility that can generate arbitrarily complex missions phase-by-phase. The last subsystem is an automatic strategic level code generator that creates Prolog or C++ programs using results from either of the other two subsystems. The GUI, automatic mission-generation facility, and mission-specification facility have been implemented using Quintus Prolog version 3.2 [Quintus Corporation 95] and XPCE version 4 for X-windows (Prowindows) [Wielemaker 94]. The strategic level code generator is written using C and can either be invoked explicitly as a standalone application or automatically from within the expert system itself.

2. The Automatic Mission Generator

a. Means-Ends Analysis

The intent of the automatic mission generator is to allow the user to generate a mission simply by specifying the AUV launch position, recovery position and mission objectives. A means-ends analysis algorithm is used to implement the automatic mission generator. In general, means-ends analysis uses a set of start conditions, a set of desired end conditions, and a set of operators to derive a sequence of operations that will eventually transform the system from the start state to the desired end state [Rowe 88, Winston 92].

In the automatic mission generator implementation of means-ends analysis, start conditions are the vehicle launch position, end conditions are the mission objectives and vehicle recovery position, and the operators represent all possible phase types. The automatic mission generator applies the means-ends analysis algorithm to produce results similar to those of Figure 29 which depicts a search mission.

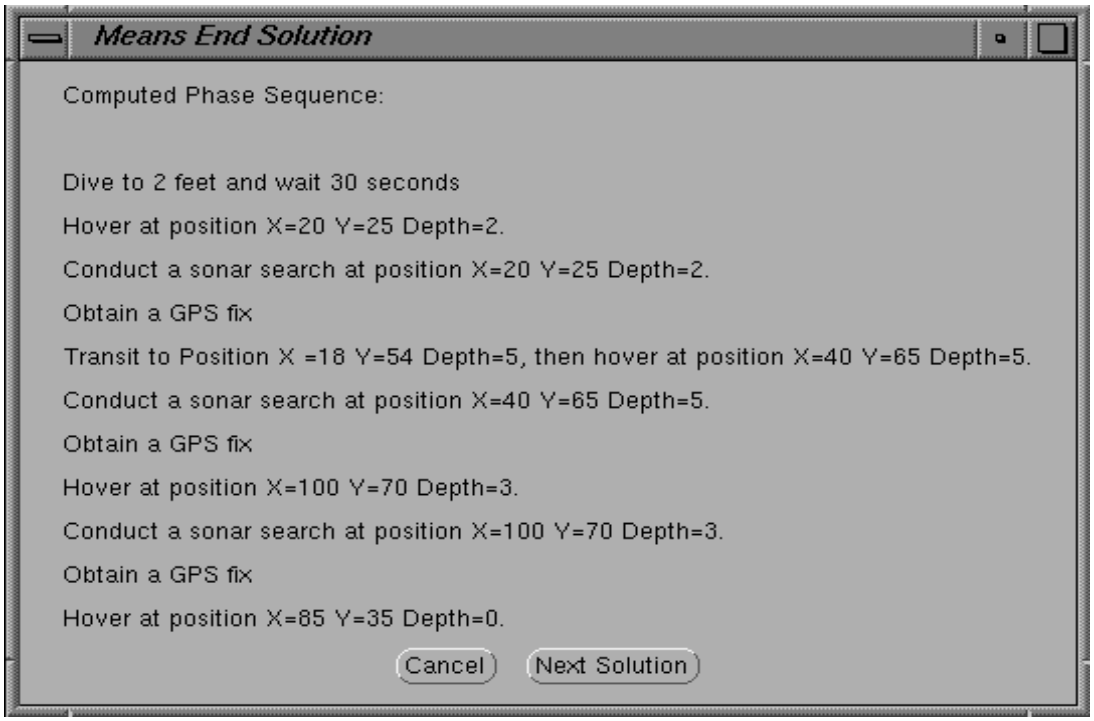


Figure 29: Search Mission Automatically Generated with Means-Ends Analysis.

In means-ends analysis, two mechanisms insure that a valid sequence of operations is generated. First, each condition in the desired end state has one or more recommended operators. for example if the desired end state is that a location has been searched, the recommended operator is to conduct a sonar search from the required location. Second, each operator has a set of required preconditions that must be satisfied before the operator can be applied as well as a set of postconditions that result from the application of the operator [Rowe 88]. The preconditions for the sonar search of position

P, for instance, might be that the vehicle is near position P and that the position must be verified by a GPS fix. An obvious postcondition of a sonar search is that position P has been searched. The means-ends algorithm uses these two mechanisms by choosing one of the desired end-state conditions and attempting to apply a recommended operator. If the preconditions of the recommended operator have not been satisfied, the algorithm attempts to satisfy the preconditions by recursively applying means-ends analysis. If the preconditions are satisfied in this way, the operator is applied. If the preconditions cannot be satisfied, the next recommended operator is attempted. The algorithm proceeds in this manner until all of the top level goals have been satisfied or until all recommended operators have been exhausted. If the operators, preconditions and postconditions are correct, means-ends analysis is guaranteed to compute a valid sequence of operations for accomplishing the desired goals [Rowe 88]. Since means-ends analysis is used to generate a sequence of phases, any sequence of phases generated can be logically executed by the *Phoenix* AUV and will accomplish all of the goals specified.

b. Adaptation of Means-Ends Analysis for Phoenix

The means-ends analysis implementation of the mission-planning system divides goals into two types. Top-level goals are those that are specified by the user while intermediate goals are used during recursive applications of the means ends analysis to accomplish top level goals. Intermediate-level goals appear as preconditions and postconditions of top-level goals and other intermediate-level goals. At present the top level goals implemented for *Phoenix* are position searches, position searches with specific routing, and entry into a recovery tube. As the functionality of lower layers of *Phoenix* software architecture evolves, high level goals will be implemented to take advantage of new capabilities. Future high level goals may include planting explosive charges, communicating with the controlling platform and taking still photographs.

There are a number of characteristics of the solutions obtained using means-ends analysis as presented in [Rowe 88] and [Winston 92] that are not well suited to planning for autonomous vehicles. The first is that solutions obtained through means-ends analysis are linear in nature. A basic assumption of the algorithm is that operations always succeed so there is no attempt to account for phase failure. This means that when the mission DFA is constructed, another algorithm or heuristic must be used for failure handling. The simplest and most obvious solution is to make the arbitrary decision that if a phase fails, the mission aborts. However, if this simple heuristic is used, the resulting DFA amounts to no more than a simple script that goes from one operation to the next and stops whenever an operation fails. Similar solutions such as having the vehicle proceed to its launch point or recovery point share this failing. Another possible solution might be reattempt any failed phase. The obvious disadvantage here is that if a phase cannot be successfully completed, the mission may not end until the vehicle exhausts its power supply. The solution that was opted for is to always attempt to proceed forward with the mission in the event of individual phase failure. If any phase fails, the succeeding phase will be the next transit or hover phase to be executed had the phase succeeded (the exception is the initial dive to operating depth). In this way if one or more phases fail, the vehicle will still attempt to accomplish as much of the mission as possible. Transit and hover phases were chosen as the phase failure successor type because, unlike other types of phases (such as searches and GPS fixes), transit and hover phases never directly rely on their predecessor phase. A graphical representation of the DFA resulting from the means-ends analysis solution of Figure 29 is shown in Figure 30.

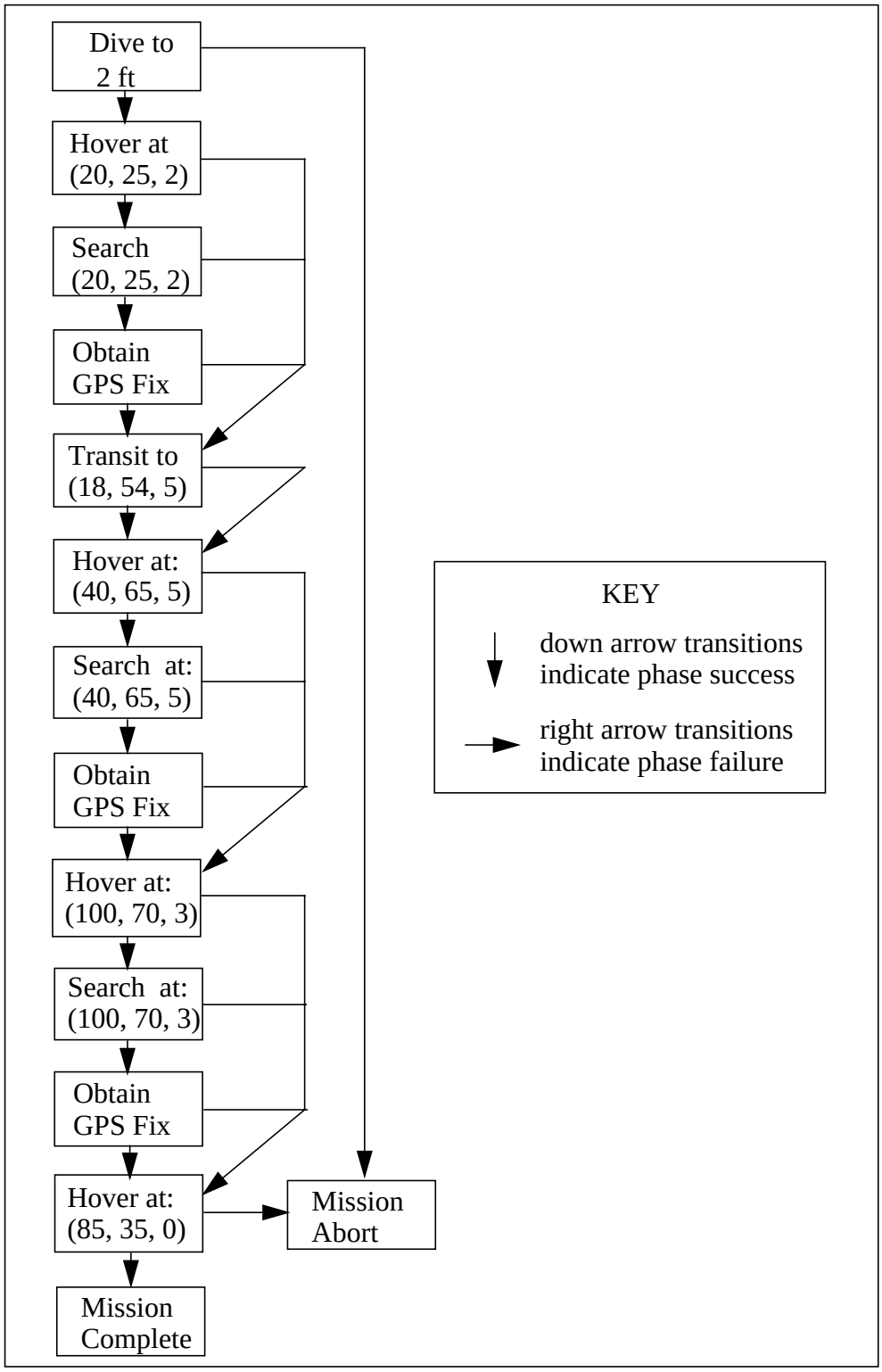


Figure 30: Graphical Representation of an Automatically Generated Mission.

The second disadvantage of means-ends analysis is that operators are implicitly prioritized by their order of appearance in the means-ends analysis specification. For instance if the operators search position P and plant an explosive charge at position P are specified in that order, the solutions obtained by means-ends analysis will always apply the search position P operator as many times as it can before it attempts to apply the plant an explosive charge at position P operator. If mission goals included a search of position (x_1, y_1, z_1), an explosive charge plant at position (x_1, y_1, z_1), a search of position (x_2, y_2, z_2), and an explosive charge plant at position (x_2, y_2, z_2), the means-ends analysis solution will conduct both searches, then plant both explosive charges. While this may be the type of behavior desired, if the transits between (x_1, y_1, z_1) and (x_2, y_2, z_2) cover a significant distance, missions of this sort become highly inefficient. The problem of operator prioritization is only a problem for operators intended to accomplish top-level goals since the ordering of intermediate-level goals do not significantly effect a mission's efficiency. The solution to this problem is the implementation of a single operator that accomplishes all top-level goals. What type of top level goal to accomplish is specified by the parameters of the operator. Different sets of preconditions and postconditions are then defined for each form of the single operator. Since only a single operator is involved, prioritization is no longer an issue. Figure 31 shows this operator definition for the accomplishment of searches and explosive placements (to date, the search operator has been fully implemented in the vehicle, but not the explosive placement operator).

```
%Recommended operators for goals.  Format is
%goal, operator

recommended(top_level_done(X), handle_top_level(X)).

%Preconditions for the application of operators
%Format is type of operator, preconditions that
%must be true list, and preconditions that
%must not be true list

precondition(handle_top_level([searched,X,Y,Z]),
     [position(X,Y,Z)],
     [explosive_ready,gps_fix_required]).
precondition(handle_top_level([charged,X,Y,Z]),
    [position(X,Y,Z),explosive_ready],
     [gps_fix_required]).

%Postconditions for the application of operators
%Add postconditions are true after application
%Delete postconditions are false after application

addpostcondition(handle_top_level([searched,X,Y,Z]),
   [top_level_done([searched,X,Y,Z]),
    gps_fix_required]).
deletepostcondition(handle_top_level([searched,X,Y,Z]),[]).
addpostcondition(handle_top_level([charged,X,Y,Z]),
   [top_level_done([searched,X,Y,Z])]).
deletepostcondition(handle_top_level([charged,X,Y,Z]),
   [explosive_ready]).
```

Figure 31: Top-Level Operator Definitions for Search and Explosive Planting Goals.

One final potential shortcoming of means-ends analysis is that while the initial solution obtained is guaranteed to be valid and complete, more optimal solutions may exist that can only be produced through repeated applications of the means-ends analysis algorithm. A possible solution to this shortcoming might be to obtain all solutions possible using means-ends analysis, compare them for efficiency, and use the most efficient one as the final solution. Another solution is to again obtain all possible solutions but allow the user to choose the one to be used. A slight modification of the second solution is currently used in the system. After a user specifies the vehicle's launch position, mission

goals and recovery position, the means-ends analysis algorithm is applied to obtain a solution. The solution is displayed textually in a window similar to the one in Figure 29 and a geographic plot of the mission path is displayed on an area map. The user is then given the option of accepting or refusing the mission. If the mission is refused, the means-ends analysis algorithm is applied to generate another solution. Using this approach the user can cycle through all obtainable solutions one at a time prior to selecting one.

3. Phase-by-Phase Mission Specification

While means-ends analysis provides a simple method for generating fairly complex missions, it is incapable of generating missions that take full advantage of the DFA structure of the strategic level. Therefore a facility has been developed for explicit specification of individual phases that can be linked together more or less arbitrarily into an executable mission. The mission-specification facility queries the user for information for each phase and uses information for all input phases to construct a mission. Information for each phase includes a phase label, the type of phase, phase parameters, the label of the follow-on phase upon successful completion, and the label of the follow-on phase upon phase failure. The expert system GUI insures that the user enters the appropriate information at the appropriate time. For instance, if a transit phase is being entered, the system will not ask for search-related information. The GUI also eliminates many data entry errors by the use of clickable maps, push buttons, clickable menus, and sliding scales. Sample GUI data entry windows are shown in Figures 32 and 33. Figure 32 shows the main window which is used for launching system facilities and visual entry and display of geographic information. Figure 33 shows windows for specifying the type of phase to be entered and phase related data for a transit phase. Data entry windows for other types of phases are similar to the one shown in Figure 33 but differ in the specific data entered.

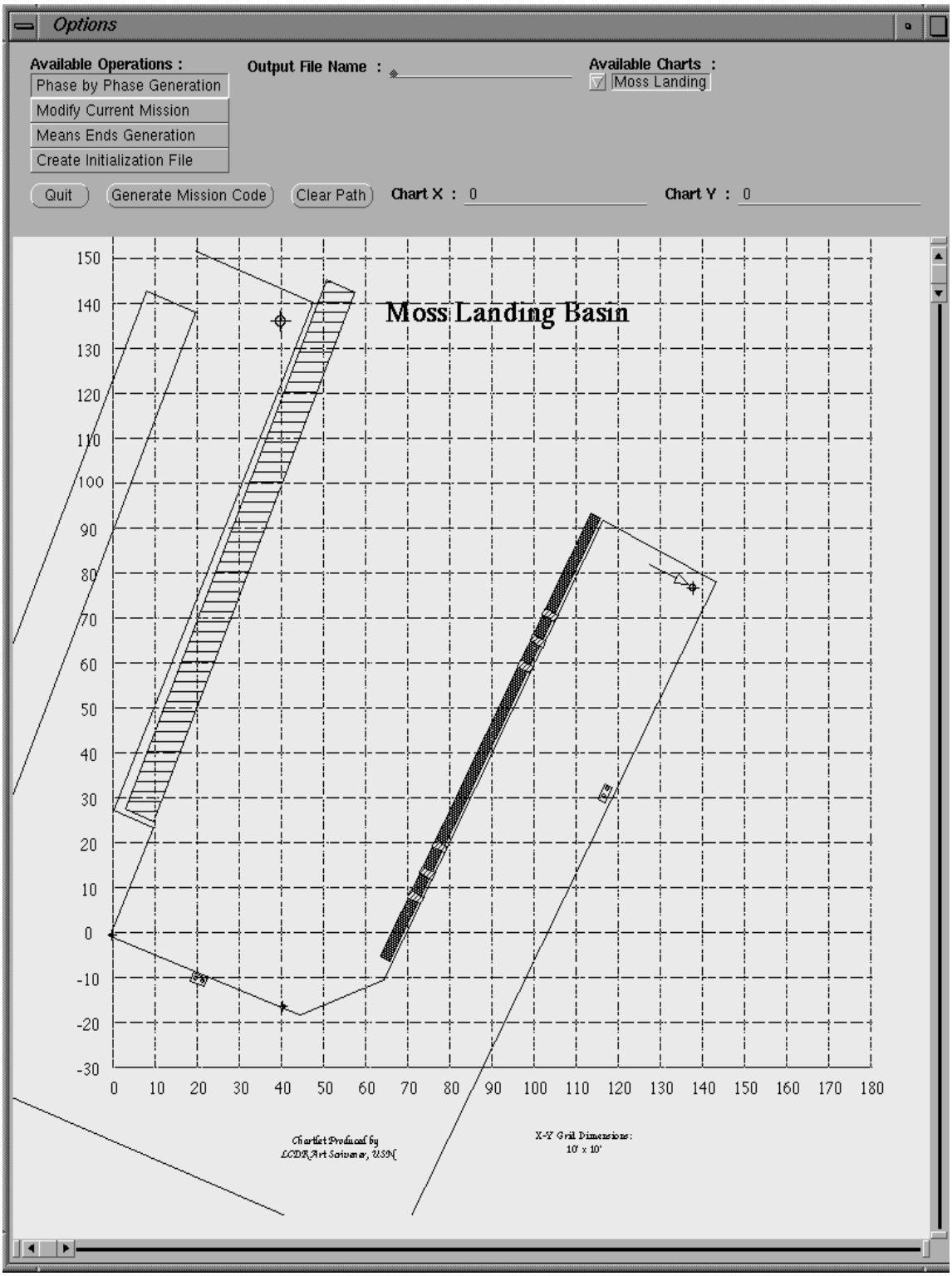
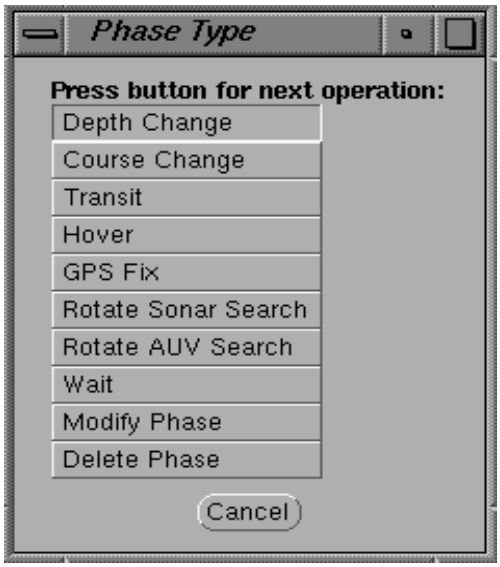
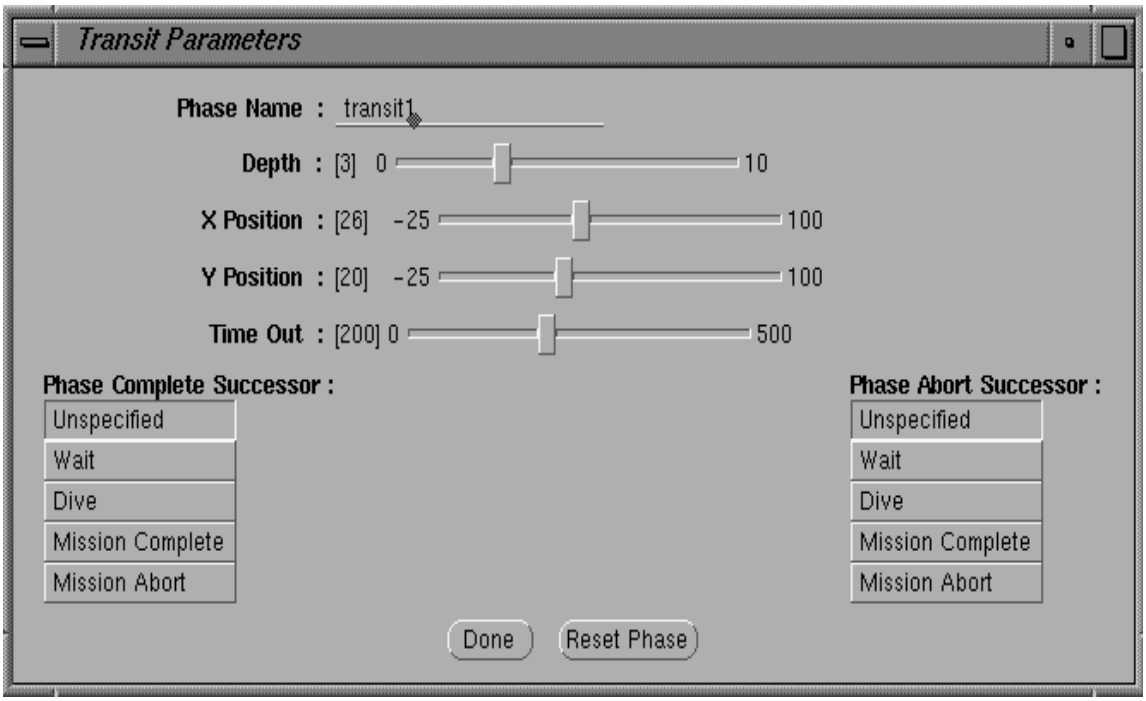


Figure 32: Mission Planning Expert System Main Window.



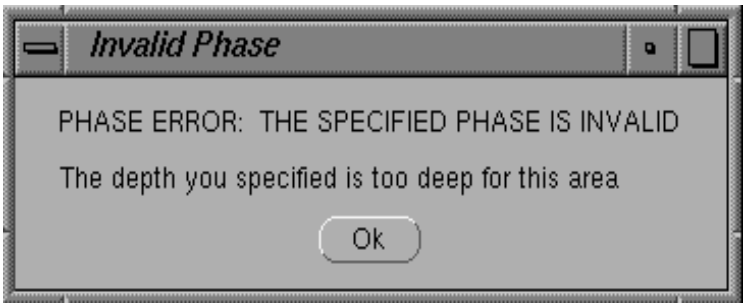
(a)



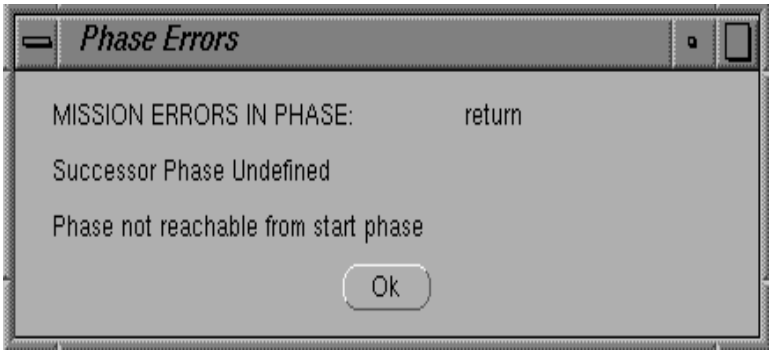
(b)

Figure 33: Data Input Windows for Phase-by-Phase Mission Specification.

Since manual phase-by-phase specification of a mission can be much more complicated than specifying a mission using means-ends analysis, a rule-based system has been implemented to insure that only valid missions are generated by the system. This requires checking each entered phase for validity in several ways. The phase must not be missing any parameters, vehicle physical limitations must be observed, and the specified locations must be within the designated operating area. This check is conducted as each phase is entered. Detected errors are immediately reported and the user is given the opportunity to modify the phase. If no errors are detected, the phase is accepted. Later, a second check is required to insure that all of the phases together make a valid mission. In general, many phases are inherently dependent upon their predecessors, so it is possible for a set of individually valid phases to constitute an invalid mission. For instance, a location cannot be searched until the vehicle has transited to the location. Errors of this type include incomplete missions, loops in the DFA, invalid phase sequences etc., and are detected by parsing with a second rule base immediately prior to mission code generation. Again detected errors are reported, and the user is given the opportunity to modify, delete or specify phases. Sample error reports for individual phase errors and mission errors are shown in Figure 34. If no errors are detected the mission is accepted and executable code is generated. By error checking both individual phases and the mission as a whole, the phase-by-phase mission-specification facility can insure that any specified mission is valid and achievable by the vehicle.



(a)



(b)

Figure 34: Error Reports for (a) Individual Phase Errors and (b) Mission Errors.

The phase-by-phase mission-specification facility is intended for users who are familiar with the structure of the RBM strategic level. Although the GUI and two rule-based systems prevent invalid missions from being specified, they do not insure that the specified mission will accomplish its intended goals. While the means-ends mission generation facility is goal driven, the phase-by-phase mission-specification facility is not. Since validity of a mission depends only on whether or not the mission is possible, it is not difficult to specify a valid mission that searches the wrong location, transits to the wrong end point, or generally does not do what it is supposed to. For this reason, it is important that a user know exactly what the intended mission is supposed to accomplish before using the mission-specification facility.

To assist in phase-by-phase mission development, a tabular representation of the mission is displayed as it is entered (Figure 35). The mission is represented as a state table listing each phase with the label of its follow-on phase upon successful completion and the label of its follow-on phase upon failure. While it might be argued that a graphical representation of the DFA (such as in Figures 11 and 30) is more intuitive, graph complexity increases far more rapidly than that of a state table as mission size increases. For arbitrarily complex missions, a state table is more concise and conveys the same information as a graph.

Phase Summary

SPECIFIED PHASES	COMPLETE SUCCESSORS	ABORT SUCCESSORS
dive	wait	mission_abort
wait	transit1	transit1
transit1	hover1	hover1
hover1	search1	transit2
search1	gps_fix1	transit2
transit2	hover2	hover2
gps_fix1	transit2	transit2
hover2	search2	gps_fix2
search2	gps_fix2	gps_fix2
gps_fix2	return_transit	return_transit
return_transit	surface	surface
surface	mission_complete	mission_abort

Figure 35: State Table Summary of a Mission Specified Phase-by-Phase.

4. Automatic Code Generation

Both the means-ends mission generator and the phase-by-phase mission-specification facility produce output in the form of a data file. This intermediate data file is not an executable strategic level mission but rather is an annotated state table description of a strategic level mission. Each line in the data file describes exactly one phase by specifying (in order) the phase type, the phase label, the label of the follow-on phase upon success, the label of the follow-on phase upon failure, the amount of time that the phase has to succeed, and the phase parameters. This output file format actually constitutes yet another RBM mission-specification language. Because of its high level of abstraction, the mission-specification language is programming-language independent and (together with the previously discussed phase type templates) enables automated executable code generation in any language for which phase templates have been created.

To date, phase type templates for the have been created for Prolog and C++. Concurrent with template development has been the construction of programs to generate executable code for missions specified using the mission-specification language. Figure 36 shows an example of a mission specified with the mission-specification language and the automatically generated executable code. Code generation programs are written using the C programming language and can be run as standalone programs or invoked from within the mission planning expert system. Standalone execution can be used to generate a mission based on a user-specified data file which can be automatically or manually created. From within the mission planning expert system, executable code is generated for a mission specified phase-by-phase or by means-ends analysis. While the mission planning system as a whole is dependent upon availability of Quintus Prolog and Prowindows, the programming language independence of the mission-specification language allows this portion of the system to be ported to virtually any platform.

```
#Mission Specification File Format
#phase_type          phase_label          completion_successor  abort_successor    [parameters]
depth_change         dive                 transit_to_op_area    mission_abort      30  3
waypoint             transit_to_op_area   hover_1               go_shallow         500 15 30 3
hoverpoint           hover_1              search_1              go_shallow         500 20 45 3 134
rotate_sonar_search  search_1             return_to_base        go_shallow         200 20 45 3
hoverpoint           return_to_base       surface               mission_abort      250 64 30 2 222
depth_change         surface              mission_complete      mission_abort      150 0
depth_change         go_shallow           return_to_base        mission_abort      100 1
```

(a) Mission-specification Language Example

```
execute_phase(hover_1)        :-  nl, printsc('PHASE hover_1 STARTED.'),
                                  ood('hover 20 45 3 135',Reply),
                                  printsc('hover 20 45 3 135!'),
                                  ood('start_timer 500',Reply),
                                  repeat,phase_completed(hover_1).

phase_completed(hover_1)      :-  ood('ask_hoverpoint_reached',Reply), Reply==1,
                                  printsc('HOVER COMPLETE.'),
                                  asserta(succeed(hover_1)).

phase_completed(hover_1)      :-  ood('ask_time_out',Reply), Reply==1,
                                  printsc('PHASE hover_1 ABORTED DUE TO TIME OUT.'),
                                  asserta(abort(hover_1)).

next_phase(hover_1)           :-  succeed(hover_1),
                                  retract(current_phase(hover_1)),
                                  asserta(current_phase(search_1)).

next_phase(hover_1)           :-  abort(hover_1),
                                  retract(current_phase(hover_1)),
                                  asserta(current_phase(go_shallow)).
```

(b) Automatically Generated Prolog Code for One Phase

```
Phase *
buildMissionGraph ()
{

    . . .

    Hover *phhover_1 = new Hover (20,45,3,134,500);

    . . .

    phHover_1->specifySuccessors(phsearch_1,phgo_shallow);

    . . .


} // buildMissionGraph
```

(c) Automatically Generated C++ Code for One Phase

Figure 36: Sample Mission Defined with (a) the Mission-Specification Language, (b) Automatically Generated Code in Prolog and (c) C++.

D. SUMMARY

This chapter describes recent modifications of the *Phoenix* strategic level and the implementation of a mission planning expert system. Recent developments in the execution and tactical levels of the RBM implementation on the *Phoenix* AUV have facilitated significant improvement at the strategic level as well. These improvements include the simplification of the strategic level through redistribution of responsibilities among the three RBM layers, definition of a finite number of phase types, the incorporation of phase parameters, and the development of phase templates. Additionally, the strategic level has been equivalently implemented in C++ and Prolog.

These improvements in the strategic level have in turn facilitated the development of the mission planning expert system. The system uses means-ends analysis to generate missions based on goals specified by a user. The system also has a facility for specifying missions one phase at a time. This facility incorporates a rule-based system to insure only valid missions are generated. Finally, automatic code generation programs were developed that use the phase templates to translate the output of the other two facilities into executable Prolog or C++ code.

The following chapter describes experimentation in support of this research. Attention is paid both to experimentation using the UVW and the physical vehicle.

VII. EXPERIMENTAL RESULTS

A. INTRODUCTION

This chapter discusses the experimental results of this research. The two major topics are virtual world results and real world results. While features were implemented in the virtual world one at a time (primarily in a bottom-up fashion), the focus of this section is on final results once all of the individual features were successfully implemented and integrated. This section consists therefore of recovery control results and mission planning system results. Mission planning expert system results are treated separately because of the broader nature of that research.

Since not all aspects of this research have been verified through in-water testing, simulation results are covered in more detail. As stated in Chapter III, the first area of in-water testing was hardware control verification. While all other in-water testing relied upon proper software/hardware interaction, this aspect of testing was not directly relevant to the research itself. Success of this aspect of testing is however shown implicitly by other test results. The primary focus of the in-water test results portion of this chapter is the execution-level behaviors upon which recovery relies. In addition in-water results of missions generated using the mission-planning expert system are covered.

B. VIRTUAL WORLD RESULTS

1. Recovery Control Results

UVW tests indicate that the low-level behaviors described in Chapter IV are capable of controlling *Phoenix* with sufficient precision to conduct recovery in a small tube. Further, the path planning routines described in Chapter V proved capable of planning an acceptable recovery path from virtually any location into a tube of known position and orientation. Figures 37 through 42 show the planned path and the actual path

followed by the vehicle during UVW test recoveries conducted using tubes of various orientations. Missions for these tests were generated using the mission-planning expert system and use the C++ version of the strategic level. The generated mission consists of three phases: dive to depth (three feet), transit to a point near the tube (-2, -15), and recover in the tube (located at (0, 0) at the orientations specified in the figures). Running the missions in the UVW requires loading the *Open Inventor* description of the desired tube (located in the viewer directory and named tube[angle][neg].iv for these tests) into the dynamics and viewer modules of the UVW. Occasional deviations between the planned and performed paths are attributable to anomalies in the edge-tracking behavior when simultaneously transitioning between voronoi regions and sonar edge-tracking targets. These anomalies are discussed in more detail later in this chapter.

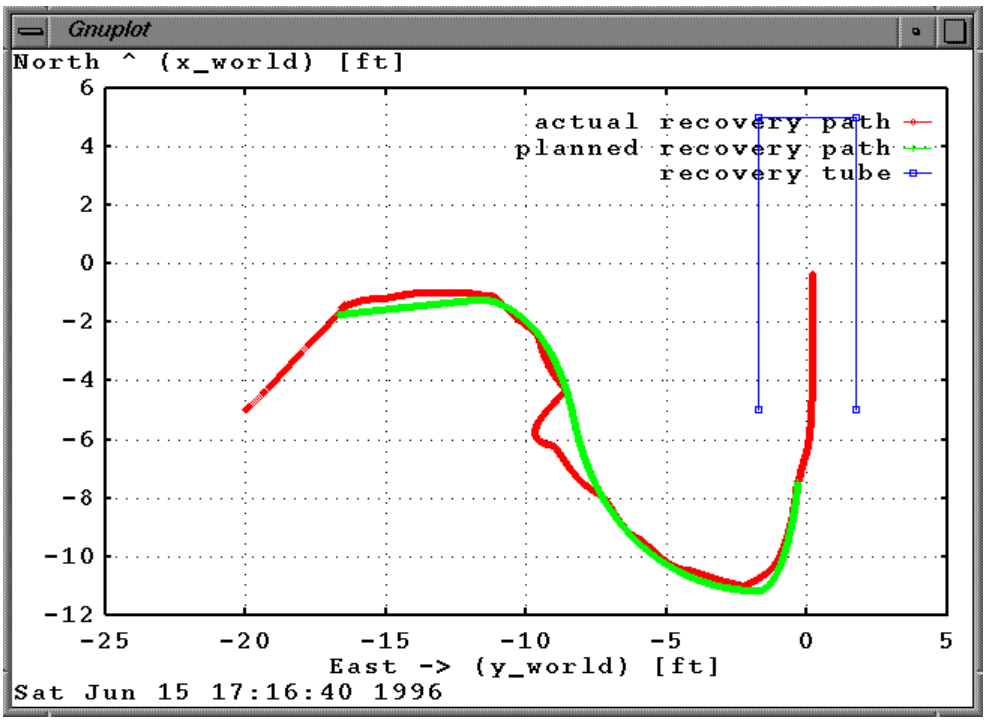


Figure 37: Planned vs. Actual Virtual World Recovery in a Tube Oriented North.

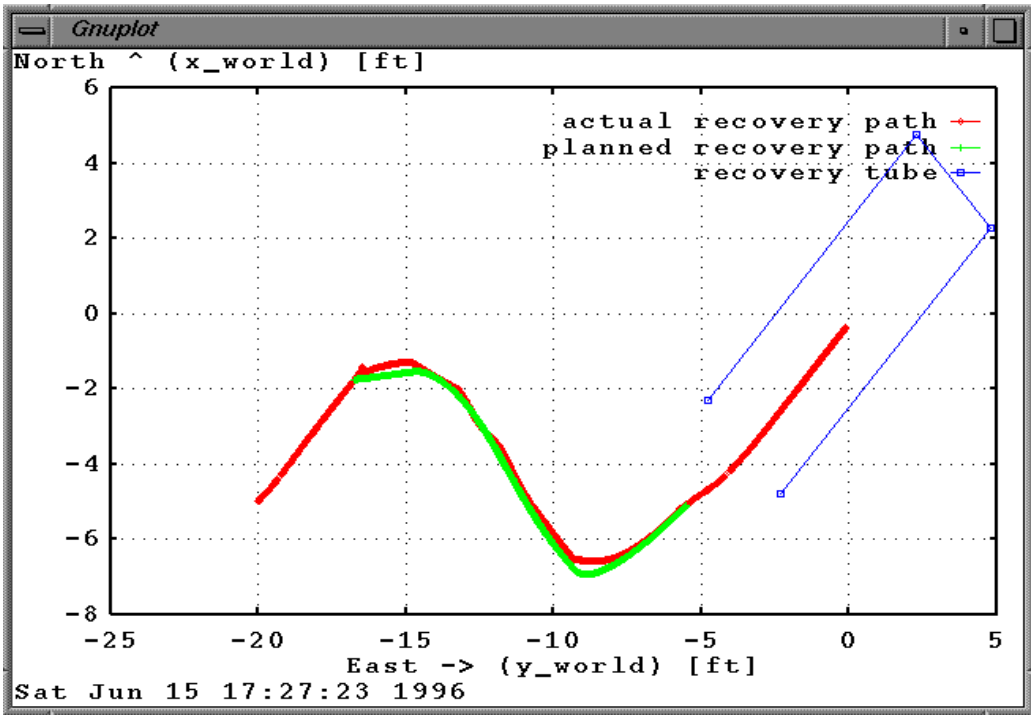


Figure 38: Planned vs. Actual Virtual World Recovery in a Tube Oriented Northeast.

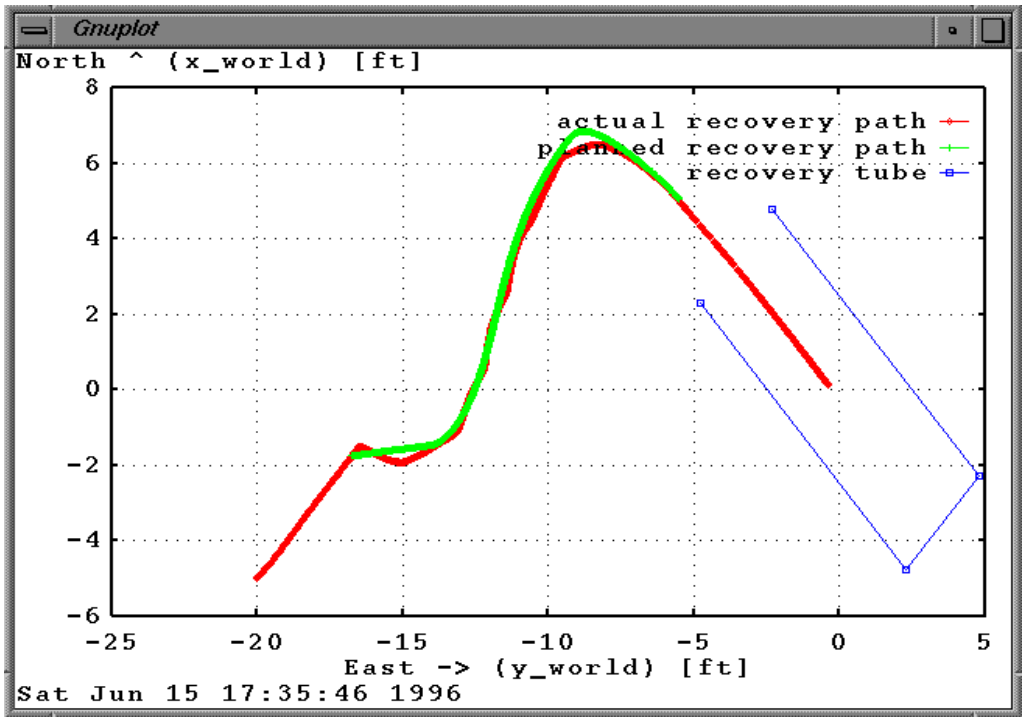


Figure 39: Planned vs. Actual Virtual World Recovery in a Tube Oriented Southeast.

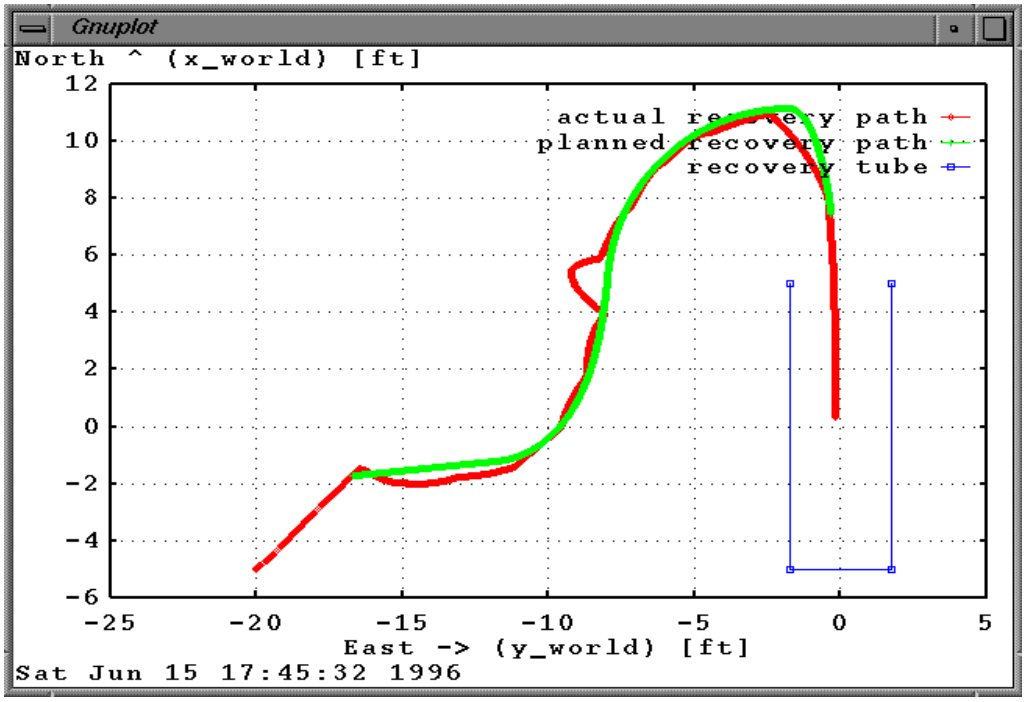


Figure 40: Planned vs. Actual Virtual World Recovery in a Tube Oriented South.

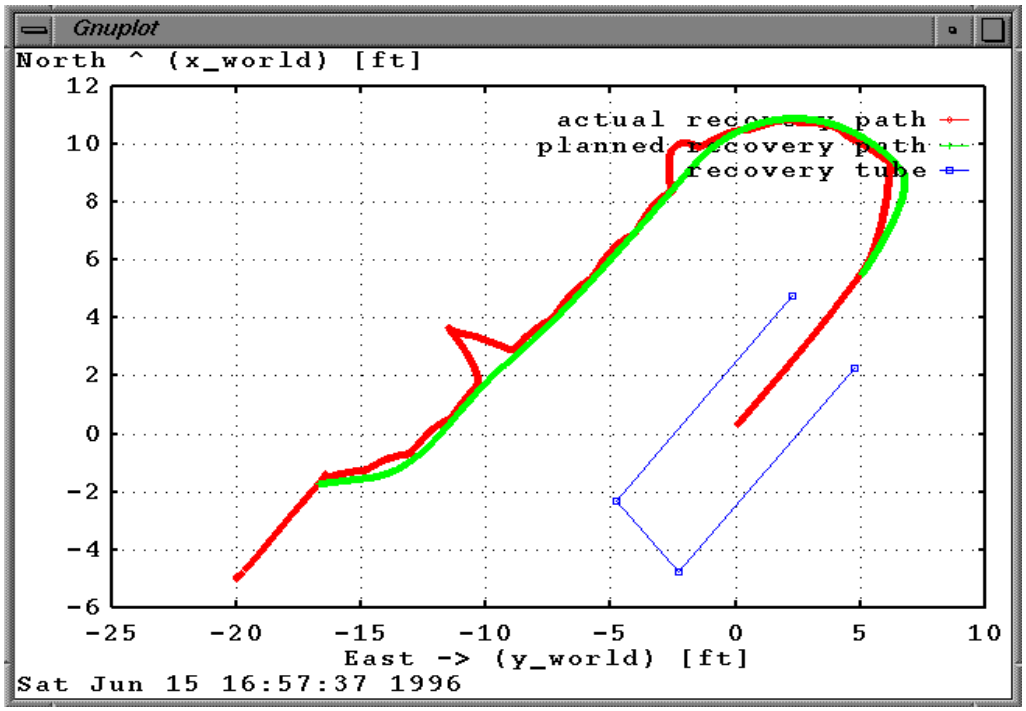


Figure 41: Planned vs. Actual Virtual World Recovery in a Tube Oriented Southwest.

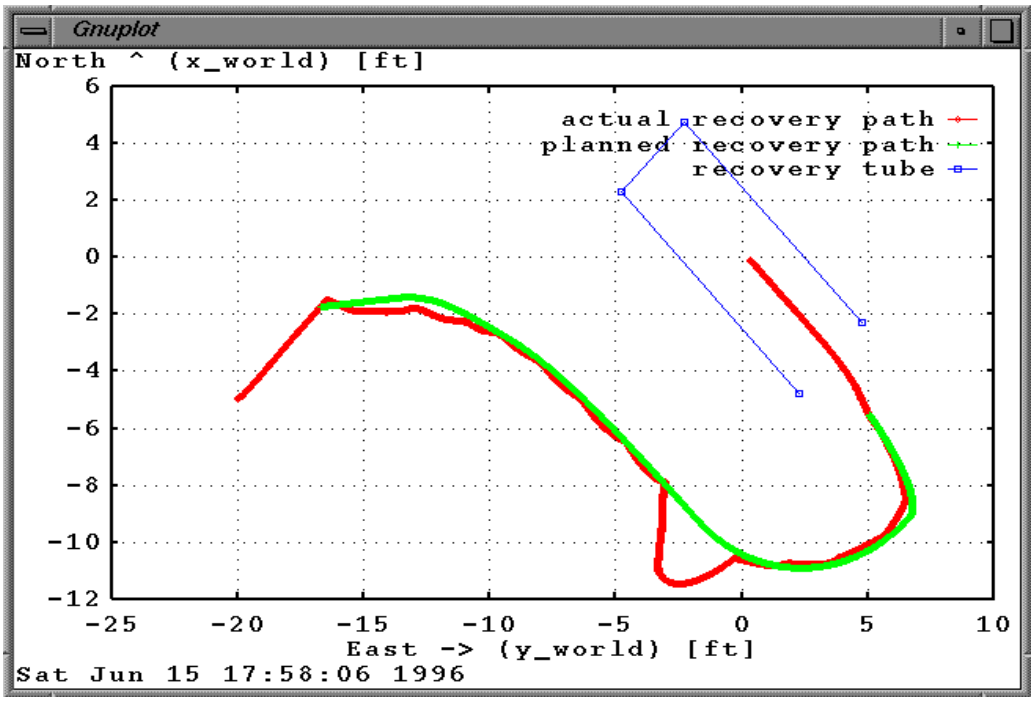


Figure 42: Planned vs. Actual Virtual World Recovery in a Tube Oriented Northwest.

The most significant aspect of these figures is that *Phoenix* transited to a point, planned a recovery path into a tube of arbitrary but known posture, and used automatically generated commands that relied on low-level sonar tracking and station-keeping behaviors to follow the planned path into the tube. There are however anomalies that require explanation. Most notable are the excursions from the planned path when rounding corners in Figures 40, 41 and 42. These excursions result from *Phoenix* being unable to distinguish the appropriate corner with the ST1000 sonar and therefore taking station on the wrong one. When transitioning from the back of the tube to the side, the corner upon which *Phoenix* is to take station is the opposing back corner. Depending on the vehicle's orientation, this corner may be masked by the near corner when *Phoenix* nears the corner. When this occurs, the near corner will be mistaken for the opposing corner resulting in an excursion from the planned path similar to the one in Figure 41. When rounding the tube's front corner, a similar phenomenon can result where *Phoenix* mistakes the near corner for the

opposing corner upon which it intends to take station. This will result in planned path excursions as depicted in all three figures. It is also possible for *Phoenix* to mistake the far corner for the near resulting in a planned path excursion that will bring the vehicle closer to the recovery tube. This type of planned path excursion is significantly more dangerous than excursions depicted in Figures 40, 41 and 42 since the vehicle may actually strike the tube. Excursions of this sort were encountered only during tests in which improperly tuned control constants resulted in underdamped vehicle response. In these tests it was not uncommon for *Phoenix* to overshoot an intended station exposing a corner that was supposed to be masked to the sonar. A potential solution to this problem might be to generate a desired range and bearing to the center of the tube rather than to a corner of the tube, thereby allowing OOD module to choose the most appropriate corner for station-keeping (based on *Phoenix* current position relative to the tube) and generating the appropriate execution-level command on the fly. Further testing may reveal whether such additional precautions are necessary.

A second anomaly is the unreliability of recovery from starting points directly in front of (or behind) the recovery tube. When directly facing the opening of either end of the recovery tube, there is simply not enough cross section for the ST1000 sonar to consistently locate and track a corner of the tube. UVW tests indicate a repeating pattern of locating a corner (sometimes but not always the correct one) and losing track of it almost immediately. This problem does not exist if the back portion of the tube is enclosed (as must be the case for an actual recovery tube). For the front portion of the tube, the simplest solution may be to increase the sonar cross section by adding a lip. Further testing is required to determine the size of the lip required and to make sure the lip does not interfere with tracking of the corner from other directions.

Finally, the figures indicate that *Phoenix* is slightly to the right of the tube's center when entering (although clearance was maintained on both sides throughout the recovery

evolution). This is a consistent aspect of all test runs. The apparent reason for this result is that neither the ST1000 nor the ST725 is placed exactly on the vehicle's centerline. This is not accounted for in Equation 37. Slight modification to Equation this equations to the following is the most likely solution.

$$Thruster_{range} = k_{thruster-range}((R_{ST725}\mathbf{sin}(75°) + |y_{ST725}|) - (R_{ST1000}\mathbf{sin}(75°) + |y_{ST1000}|)) \quad \text{(Eq. 56)}$$

where y_{ST725} and y_{ST1000} are the Y coordinates of the ST725 and ST1000 sonars in AUV body coordinates respectively.

Another issue that should be noted concerning UVW testing and the results shown in Figures 37 through 42 is the issue of control constants for the station-keeping PD control laws. UVW testing has shown that improper PD control constants will result in a failure to accurately follow the planned recovery path. Thruster and propeller PD constants must be tuned in such a way that lateral and longitudinal responsiveness are the same. Failure to properly tune control constants will not preclude reliable recovery, but will result in mediocre planned-path following such as that which occurred in the test depicted in Figure 43. It should also be noted that control constants used in the tests depicted in Figures 37 through 42 were tuned in the UVW. In-water testing described in the following section required retuning of the control constants. Control constants shown in Table 3 are real-world constants. This disparity between the virtual and real worlds highlights possibly the most important area of near-term future work: real-world validation of the UVW. Adjustment of coefficients of the UVW hydrodynamic model to accurately reflect the actual hydrodynamic characteristics of *Phoenix* is essential to long term software development using the UVW.

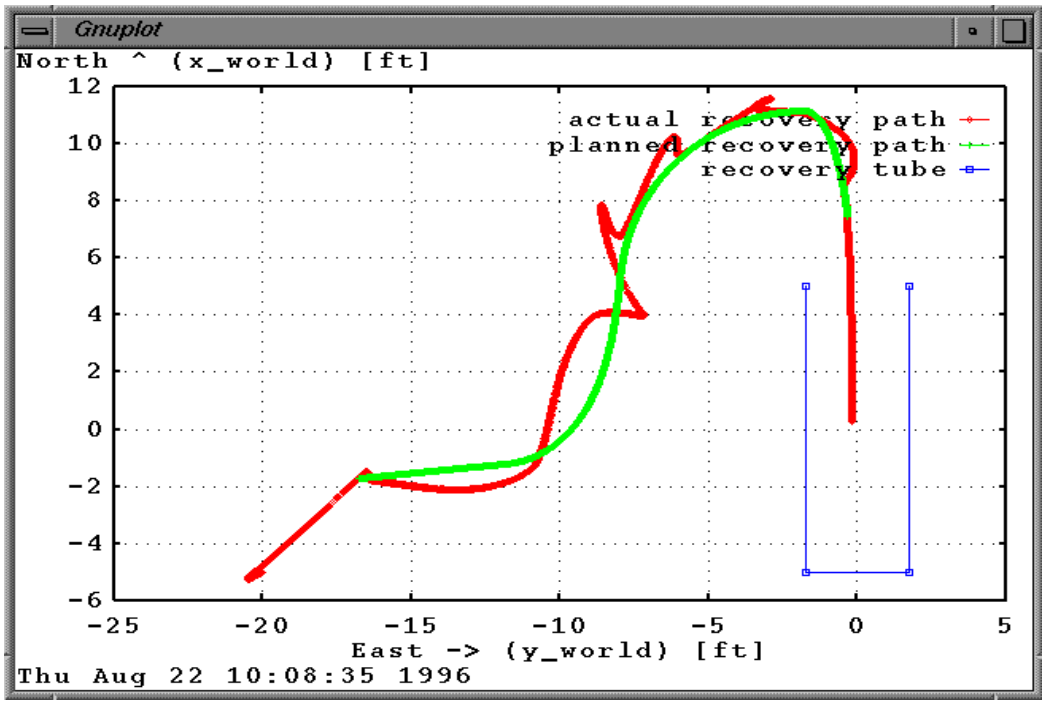


Figure 43: Recovery with Poorly Tuned PD Control Constants.

2. Strategic Level and Mission Planning Expert System Results

In addition to the real world and UVW, an interactive standalone ood_test program has been developed for strategic level testing. This program allows a human acting as the tactical level OOD to manually respond to strategic level queries by querying a human rather than the tactical level. Logic and sequencing of the strategic level (the structure of the DFA) can therefore be evaluated without the AUV or UVW. In Figure 44, the ood_test program is used to debug an automatically generated Prolog mission corresponding to the mission of Figure 11. By utilizing the standalone strategic level and UVW for code development and initial testing, and in-water testing for final testing and validation, it has been possible to rapidly and simultaneously develop and implement new features at all three layers of the RBM architecture [Brutzman 96].

```
Begin Initialization
OOD Received Command: initialize
OOD Received Command: start_timer 120
OOD Received Command: ask_initialized

?- yes

Initialization Complete
Phase Completed
Change Depth to 3 feet

. . .

Depth Reached
Phase Completed
Transit to 40 70 3

. . .

Waypoint Reached
Phase Completed
Hover at 20 60 3 270
OOD Received Command: hover 20.000000 60.000000 3.000000 270.000000
OOD Received Command: start_timer 300
OOD Received Command: ask_hoverpt_reached

?- yes

Hover Point Reached
Phase Completed
Conduct Sonar Search at 20 60 3 270
OOD Received Command: sonar_search 20.000000 60.000000 3.000000 270.000000
OOD Received Command: start_timer 180
OOD Received Command: ask_sonar_search_complete

?- no

OOD Received Command: ask_time_out

?- yes

Timer Expired
Phase Aborted
Change Depth to 0 feet

. . .

Depth Reached
Phase Completed
Transit to 50 20 0
OOD Received Command: waypoint 50.000000 20.000000 0.000000
OOD Received Command: start_timer 300
OOD Received Command: ask_waypt_reached

?- yes

Waypoint Reached
Phase Completed
Mission Complete
```

Figure 44: Standalone Testing of a Mission Using the `ood_test` Program.

Strategic-level test missions for *Phoenix* have been generated in both Prolog and C++ using manual programming, the mission-specification language and automatic code-generation programs, and using the entire mission planning expert system. The results of these tests have been predictable and correct.

Figure 45 shows a graphical plot of a C++ mission created using the means-ends analysis mission generation facility. Goals for the mission were to conduct sonar searches

from two locations (one with specific routing to the search point). When executed, the mission conducted both sonar searches, obtained GPS fixes to verify the search positions, classified detected objects, planned a safe path around detected obstacles (object classification and path planning were conducted at the tactical level), and proceeded to the designated recovery position. The need for GPS fixes to verify search positions (as well as the initial dive to operating depth) were not specified by the user, but were executed because of the means-ends analysis preconditions and postconditions for the sonar search operation.

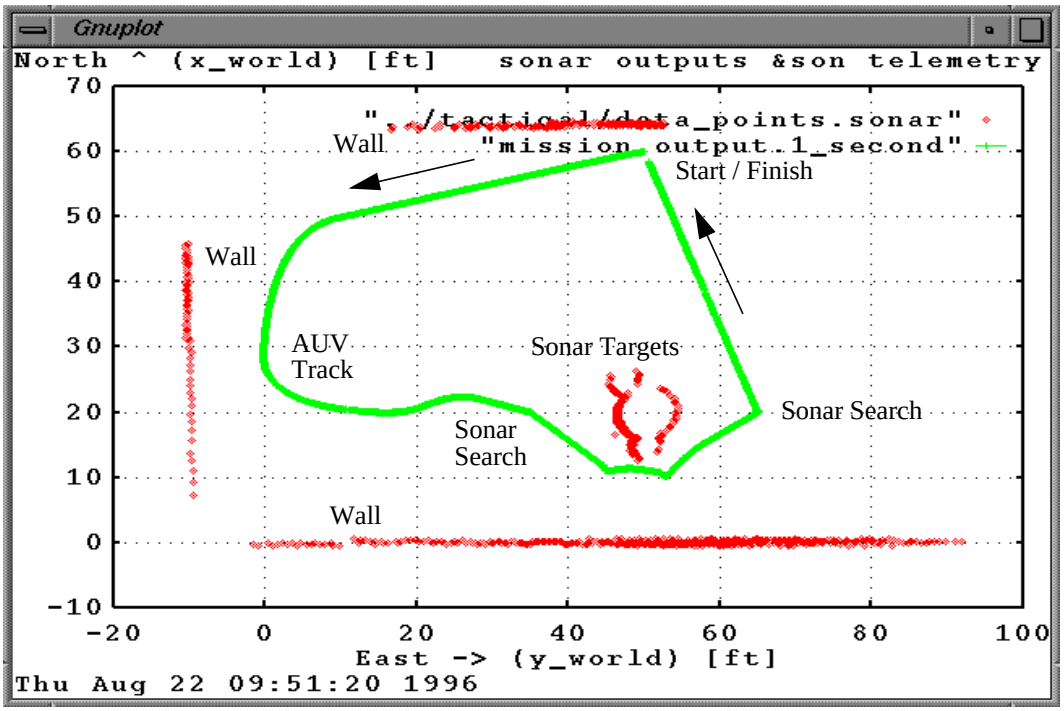


Figure 45: UVW Results of a Mission Generated Through Means-Ends Analysis.

Simplifications made in the structure of the strategic level proved useful. A Prolog mission using the new strategic level format is approximately one third as long as an equivalent program manually prepared using the format in place prior to the simplifications described in this paper [Leonhardt 96]. Templates and automatic code generation proved reliable and versatile and resulted in successful testing described here and in [Brutzman

96]. Testing also showed that it is a fairly simple matter to accurately implement the RBM strategic level in C++ using objects to represent the nodes of the DFA. Specifically, UVW tests showed that C++ missions produced by the mission planning expert system were indistinguishable in behavior from equivalent Prolog missions created by the same system.

In general, manually coded missions have been found to be error prone. Even missions produced manually using templates or by manually modifying working code are still subject to typographical errors, errors of syntax, and logical errors in individual phases or sequences of phases, any of which result in invalid or incorrect missions. Moreover the increased magnitude of the Prolog code as mission complexity increases makes manual programming of complex missions infeasible. The slower growth of C++ program size alleviates this problem only slightly. This result amounts to no more than a confirmation of previous results that were a primary motivator of this research work.

Missions produced using manually edited mission-specification language files and the automatic code-generation programs offer a substantial improvement over manual editing but are still prone to errors. This is because the mission planning expert system checks missions for validity prior to generating the intermediate mission-specification file. Mission specification files are not checked for errors by the automatic code generation programs. Therefore, logical errors that are otherwise caught by the mission planning expert system can be inserted by the human editor and processed by the automatic code generation program without complaint, resulting in incorrect and unpredictable mission code. Additionally, because the mission-specification language is significantly more abstract than programming languages, it is somewhat terse and cryptic. Manually edited mission-specification language files are therefore prone to typographical errors and misordered data as well as logical errors.

Missions produced using the entire mission planning system are easier to create than those coded manually or using mission-specification files. They have also proved

more reliable. The “Florida mission” [Marco 96b], a complex mine search and classification mission consisting of roughly 25 phases, was produced using the mission planning expert system in approximately ten minutes. A manually coded version of this mission was originally generated and debugged over a period of approximately two weeks.

C. REAL WORLD RESULTS

1. Sonar Tracking Behaviors

The primary goal of in-water testing to date has been the verification of execution-level sonar tracking and vehicle-control behaviors. Testing was conducted in the Center for AUV Research test tank. Sonar tracking behaviors were first tested with the sonar at a fixed position. Both the target-tracking and edge-tracking modes were successfully used to track a 0.5 meter diameter cylinder. In this series of tests, the cylinder was placed in various locations relative to the stationary sonar. The target search, target-tracking and edge-tracking modes were then used to locate and track the target for approximately 60 seconds. Figures 46 and 47 show plots of bearing versus time and range versus time for a test during which target-tracking mode was used to track the cylinder located on a bearing of approximately 70 degrees at a range of approximately 13 feet relative to the sonar. As can be seen in Figure 46, the sonar scanned to the right until reaching the target. At this point it scanned back and forth across the target (a sector width of approximately ten degrees). Figure 47 shows the range differential as the sonar scanned across and off the target during its sweeps. In this plot, zero ranges actually indicate that no sonar return was received.

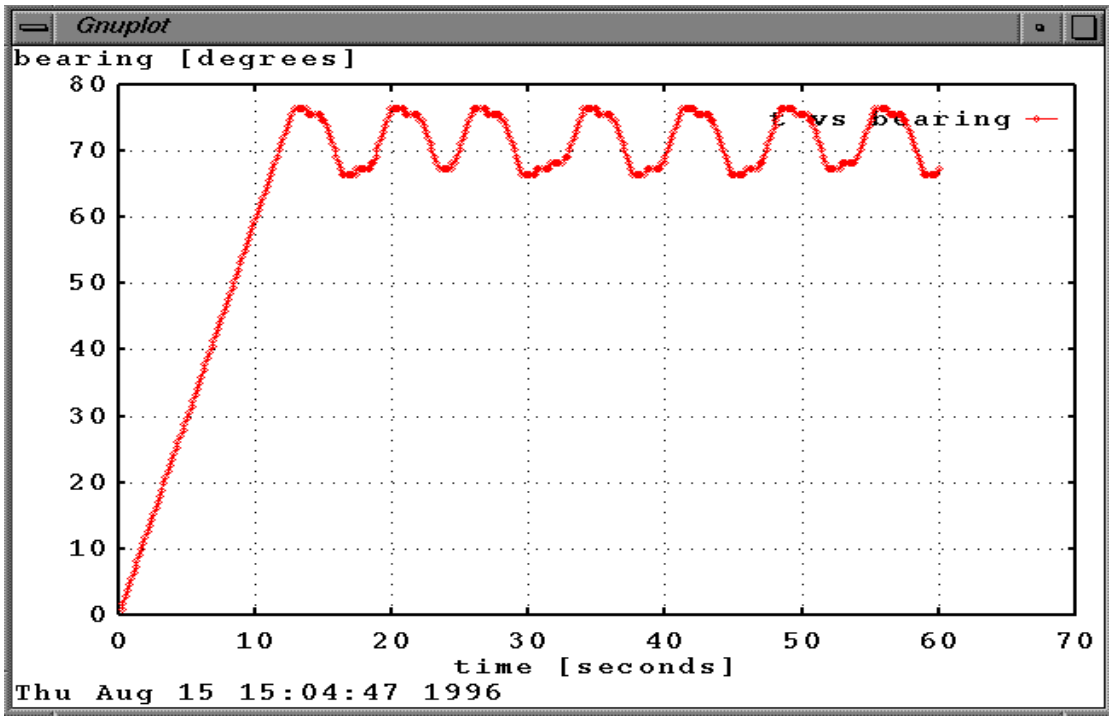


Figure 46: Stationary Sonar Full Target Track Bearing vs. Time.

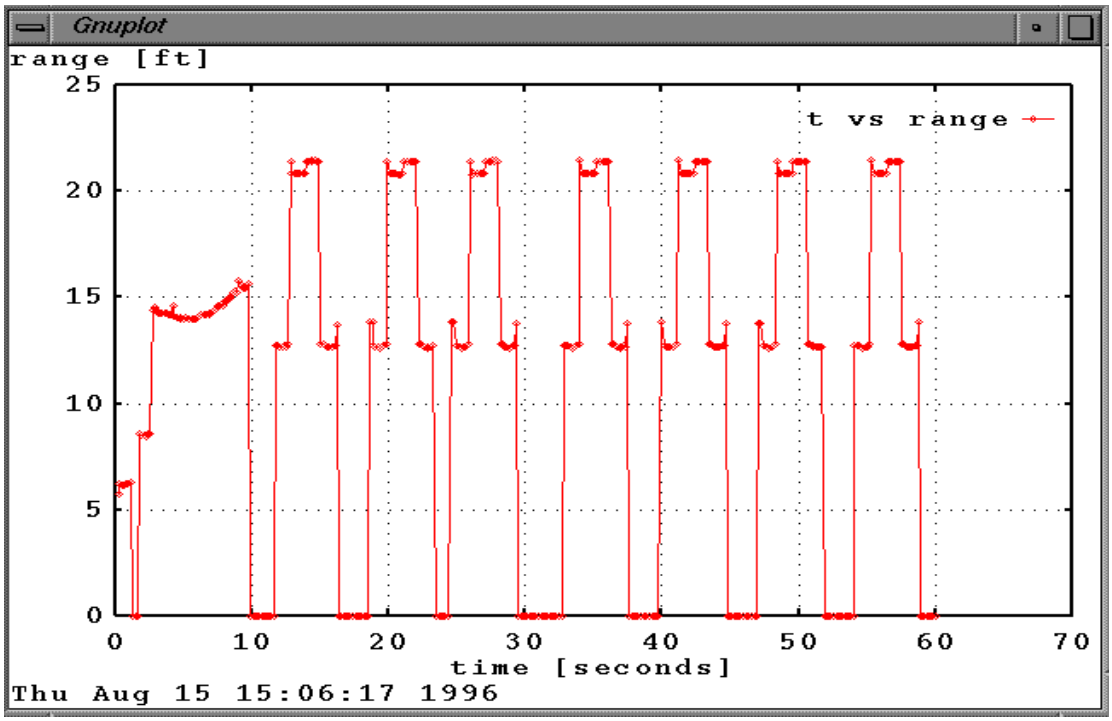


Figure 47: Stationary Sonar Full Target Track Range vs. Time.

Figures 48 and 49 show test results from static sonar tracking of the edge of the cylinder located at a range of approximately nine feet bearing approximately 337 degrees relative to the sonar. Figure 48 shows the sonar sweeping to the left until locating the target. At this point, it begins sweeping back and forth across the cylinder's right edge. The sweep width during tracking is approximately eight degrees. As target size decreases or range increases, the sweep width for edge tracking will be very close to the sweep width for full target tracking. Figure 49 shows the range vs time plot. Again, zero ranges indicate no sonar return was received. In this tests, the sonar located the target and successfully tracked the edge for a period of 60 seconds. In this series of static sonar tests, both tracking modes proved reliable so long as sufficient separation between the intended target and the test tank wall existed to ensure adequate range differential between the target and the background.

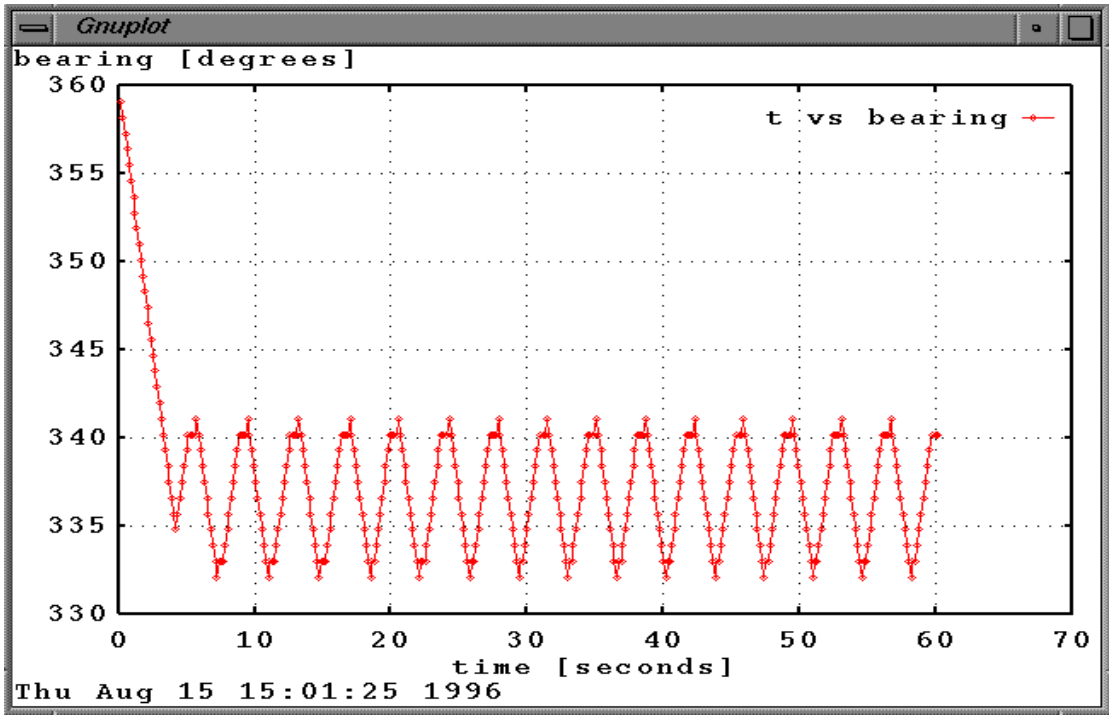


Figure 48: Stationary Sonar Target Edge Track Bearing vs. Time.

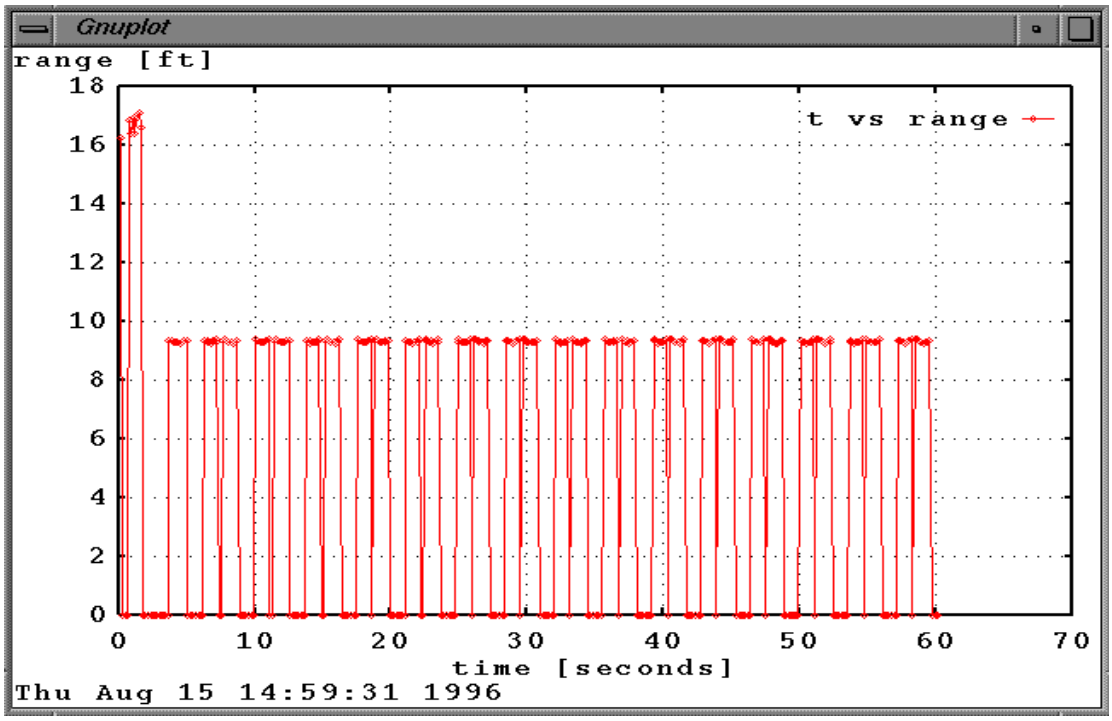


Figure 49: Stationary Sonar Target Edge Track Range vs. Time.

Because of the sometimes unreliable nature of sonar data it was occasionally possible to lose a target that was being tracked. Figures 50 and 51 show a portion of a test where a pair of spurious ranges caused the sonar to lose the edge of the recovery tube after it had been tracking for over 90 seconds. Figure 50 shows that between 96 and 97 seconds into the test, two sonar ranges at approximately nine feet were obtained. The previous on-target return was at a range of approximately six feet, so the nine foot ranges were assumed to be part of the target. The subsequent ranges were accurate and represented the test tank wall at approximately 12 feet, but since the previous on-target range was nine feet, the 12 foot range was also assumed to be part of the target. Figure 51 shows the sonar bearing as it continues to sweep to the left across the wall which it believes to be part of the target. At present, the only solution to this problem is to avoid situations where a spurious return will cause loss of track. This means that targets must be at least ten feet from background objects (or the range differential for target discrimination must be reduced from five feet).

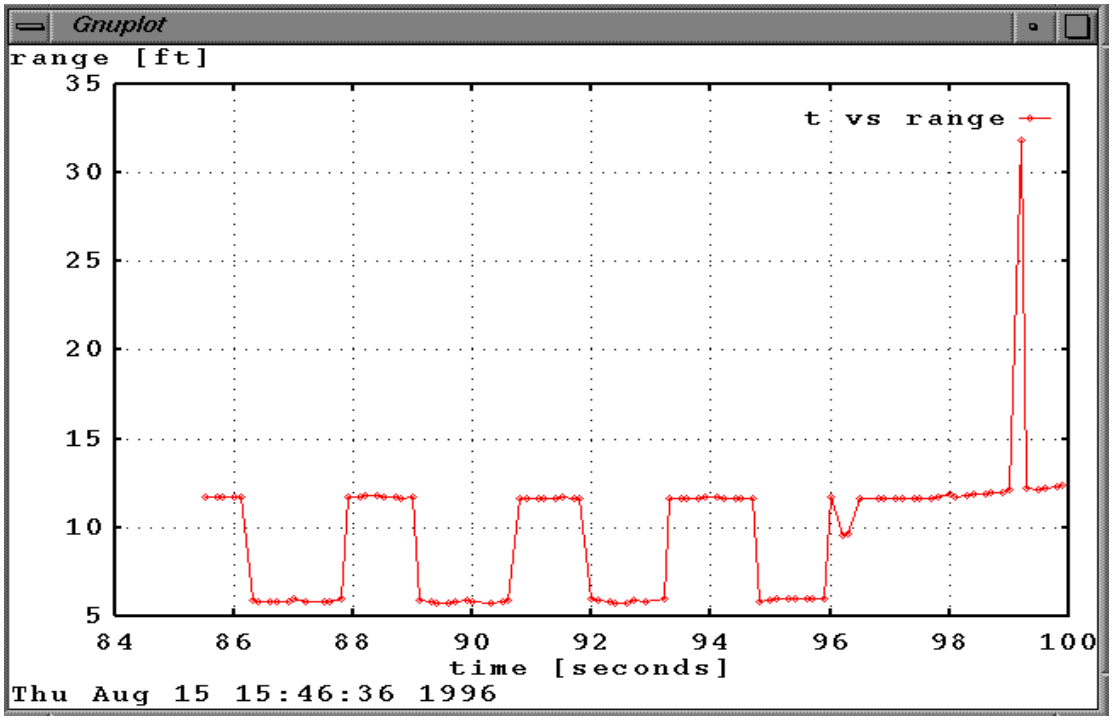


Figure 50: Range vs. Time Plot Showing Loss of Track in a Confined Area.

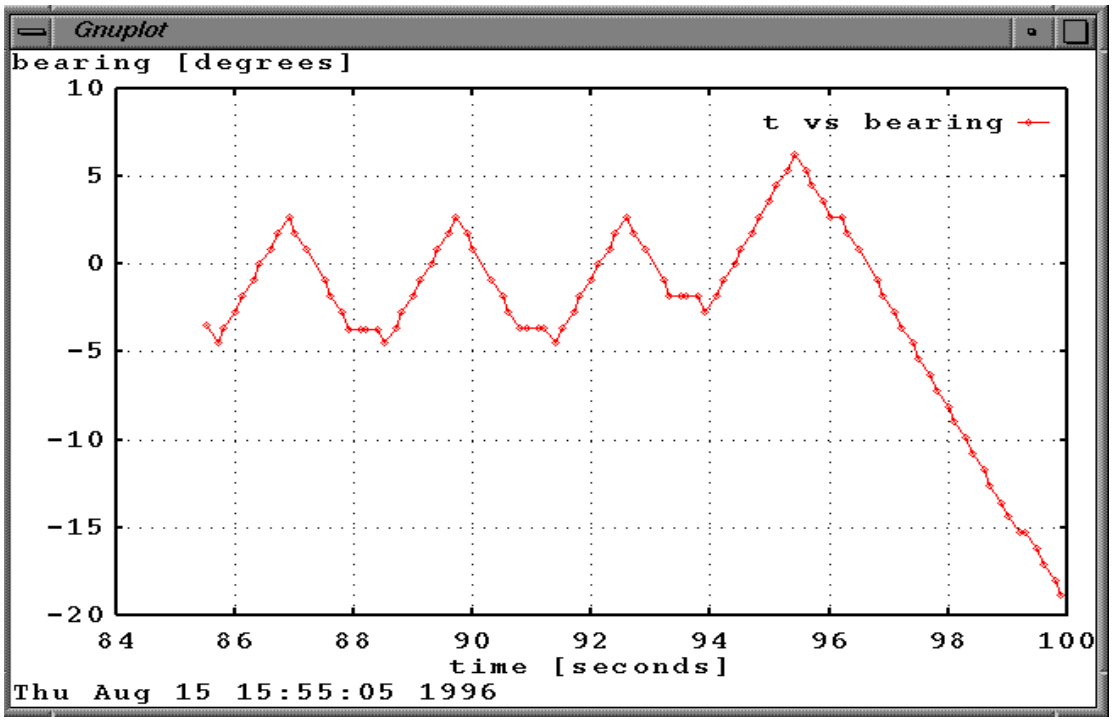


Figure 51: Bearing vs. Time Plot Showing Loss of Track in a Confined Area.

2. Station-Keeping Results

The next series of in-water tests were intended to verify execution-level station-keeping behaviors. Tests were first conducted using full target tracking and edge tracking to maintain a series of stations relative to the 0.5 meter diameter cylinder. As UVW results had indicated, both sonar control modes can be used to navigate to and maintain stations to within six inches. As expected, the higher target update rates of the edge-tracking sonar mode allowed more responsive control than the full target-tracking sonar mode and resulted in achievement of commanded stations in less than half the time. Figures 52 through 54 show the results of a test requiring *Phoenix* to proceed through a series of three stations relative to the cylinder using a full target sonar scan. In addition, the vehicle maintained the final station for a period of 30 seconds. Vehicle heading pointed directly at the target for the first two stations and north for the final station.

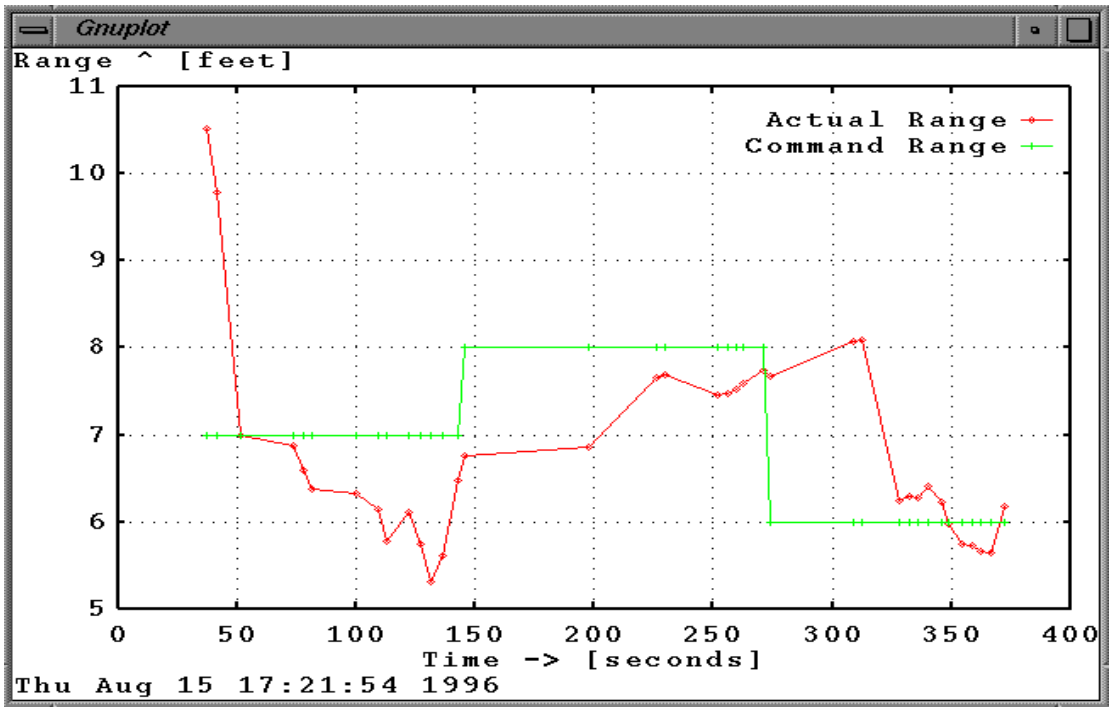


Figure 52: Commanded and Actual Range to a Cylinder with Target Tracking

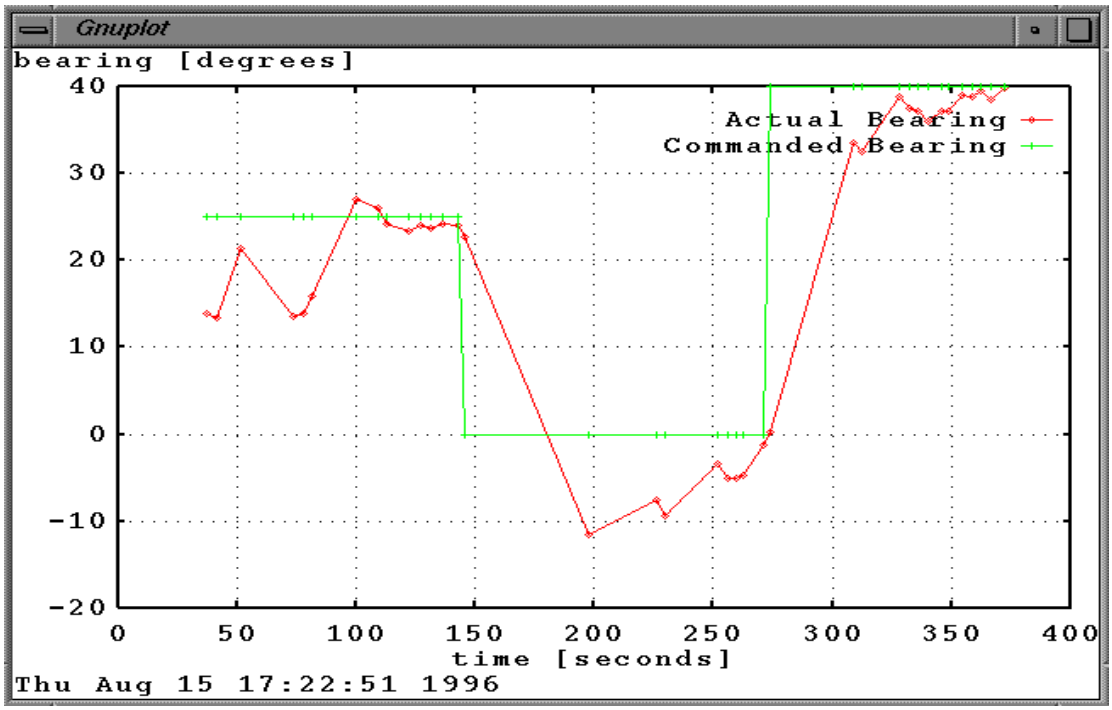


Figure 53: Commanded and Actual Bearing to a Cylinder with Target Tracking.

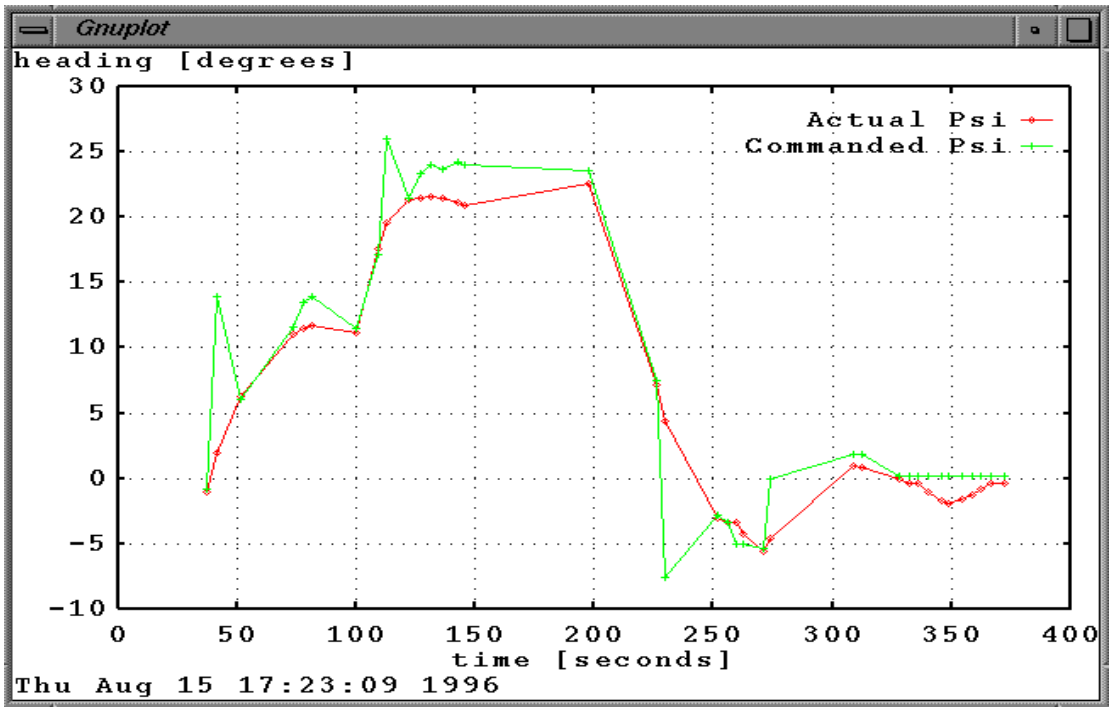


Figure 54: Commanded and Actual Heading while using Target Tracking.

As can be seen in the previous figures, commanded stations were achieved and maintained. However time between target updates was normally five to ten seconds and occasionally as long as 20 seconds. This slow update rate resulted in a slow convergence with commanded range, bearing and heading and an occasional tendency to overshoot. While part of this is likely due to improperly tuned control constants, the fact that station keeping using the edge-scanning sonar mode converges upon the commanded station much more quickly indicates that the slow target update rate significantly reduces the vehicle's ability to accurately control relative to the target.

Figures 55, 56 and 57 show the results of using edge tracking as the basis for station keeping. Stations were the same as those used during testing of the full target scan based station-keeping behavior. Again, the vehicle achieves all three stations and maintains the third for 30 seconds. The roughness of the range versus time and bearing versus time plots indicates that further tuning of control constants is required. The increased update rate of edge tracking when compared to target tracking enables *Phoenix* to achieve each station in approximately half the time and significantly improves the accuracy of vehicle control.

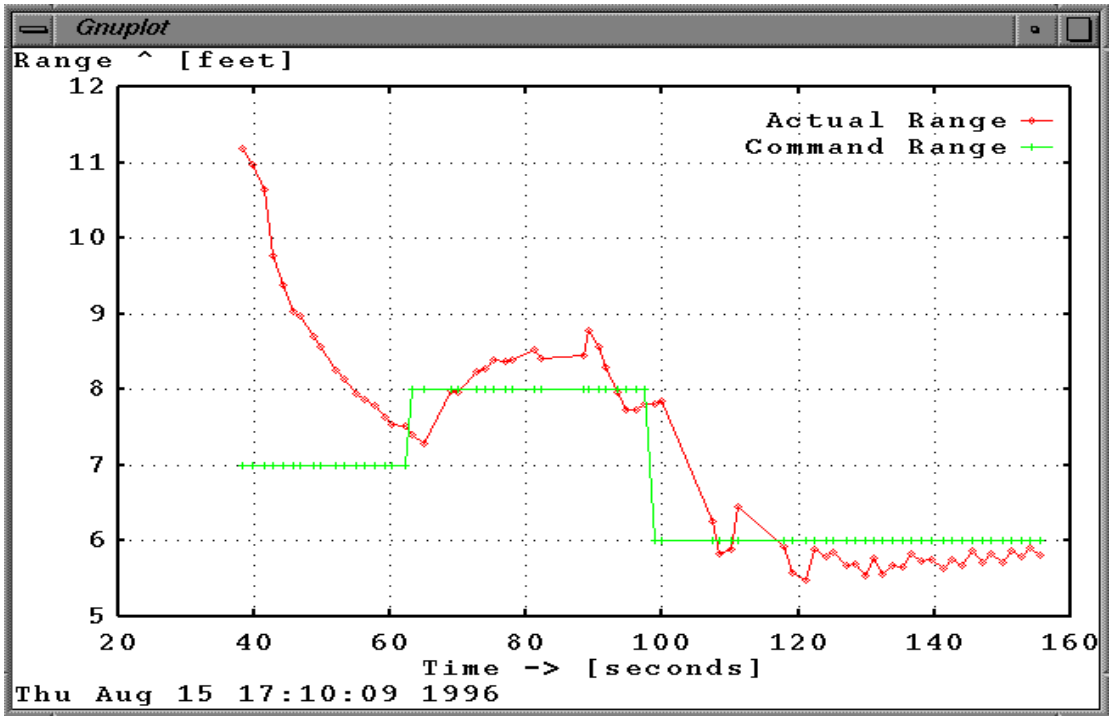


Figure 55: Commanded and Actual Range to a Cylinder with Edge Tracking.

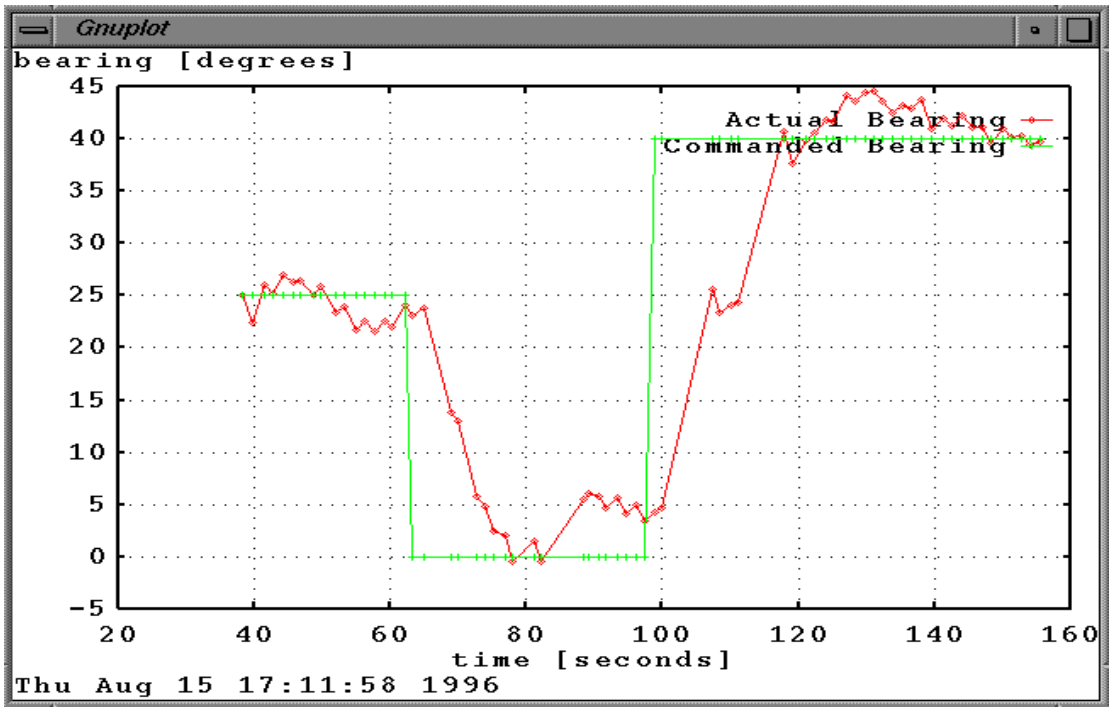


Figure 56: Commanded and Actual Bearing to a Cylinder with Edge Tracking.

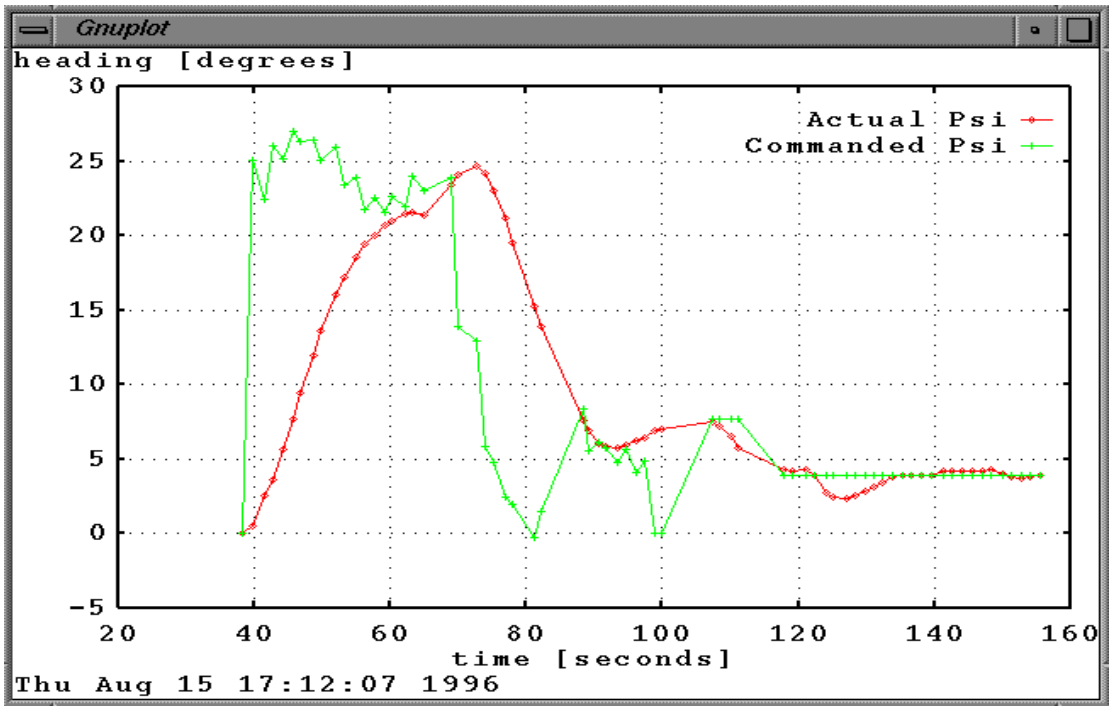


Figure 57: Commanded and Actual Heading while using Edge Tracking.

The final series of in-water tests were intended to test the vehicle's ability to maneuver around the recovery tube in order to position for final recovery. Because of the uneven shape of the recovery tube, this was a much more difficult task than maneuvering about the cylinder. This test required *Phoenix* to travel through a series of four stations using the edge tracking sonar mode and maintain the final station for a period of 60 seconds. Heading was directed at the corner of the tube being tracked for the first two stations, and was aligned with the recovery tube for the final two stations. The AUV starting position was approximately 11 feet from the front left corner of the recovery tube. The final station placed the nose of the vehicle just inside the recovery tube. From this position, recovery is possible using the final recovery control mode described in Chapter IV. Figures 58, 59 and 60 show the results of one of these tests.

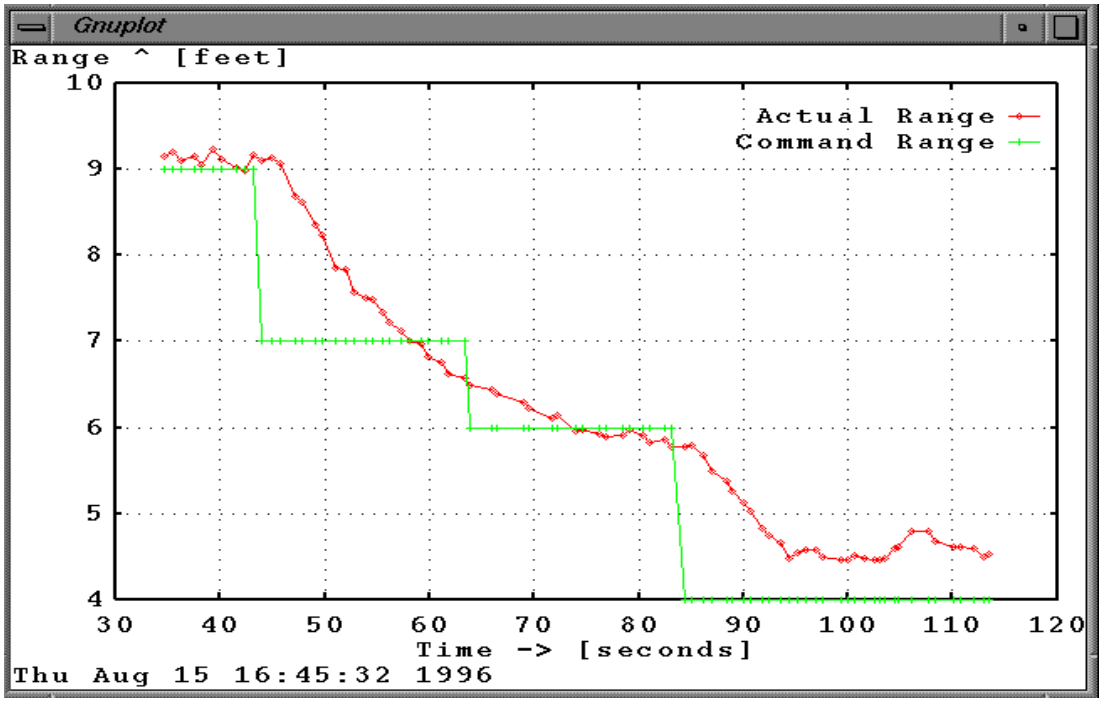


Figure 58: Commanded and Actual Range during Tube Station Keeping.

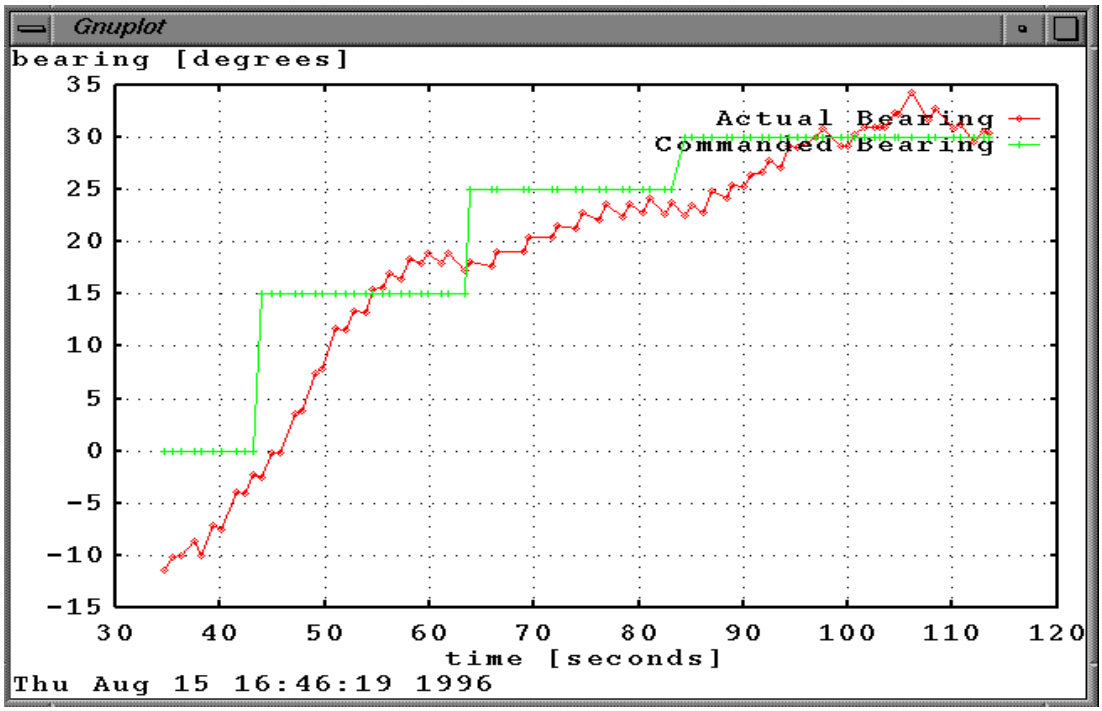


Figure 59: Commanded and Actual Bearing during Tube Station Keeping.

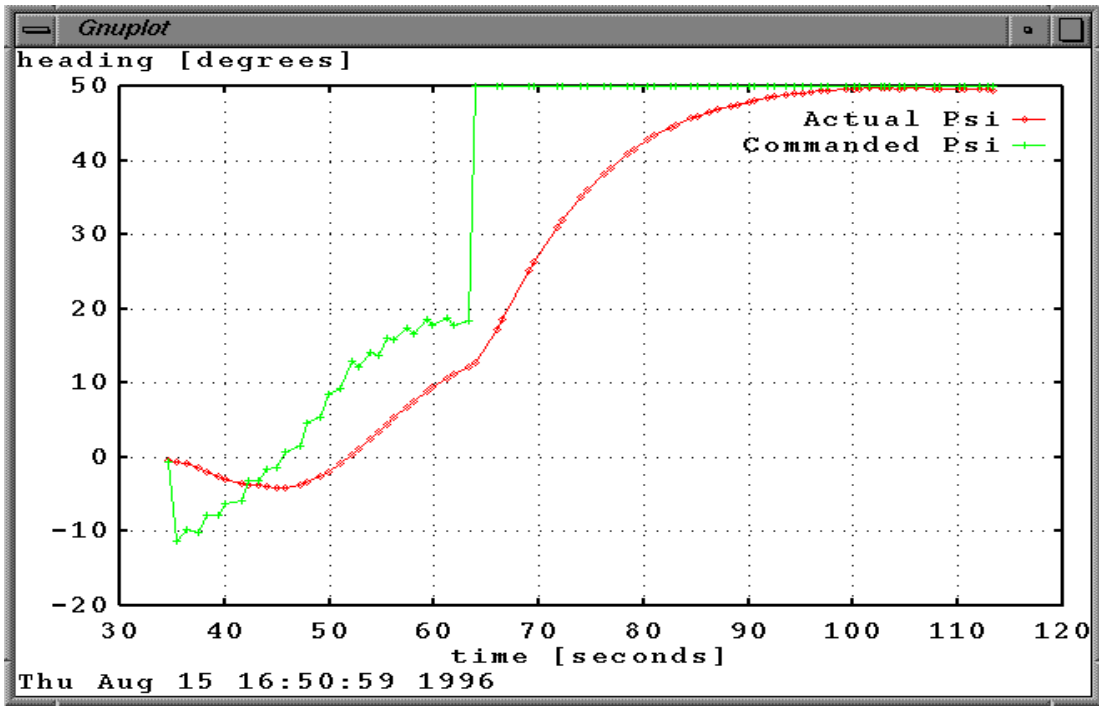


Figure 60: Commanded and Actual Heading during Tube Station Keeping.

The ability of *Phoenix* to maintain station on different types of objects using the same sonar tracking and control modes is clearly demonstrated by these results. This capability has the potential to prove valuable not only during recovery operations, but during execution of various other types of missions as well.

3. Strategic Level and Mission Planning Expert System Results

While the primary purpose of in-water testing in support of this research was to verify execution-level behaviors upon which further testing would depend, an effort was also made to test improvements to the strategic level software and missions generated with the mission planning expert system. In-water testing of Prolog missions generated with the mission-planning expert system were conducted in the NPS sub-Olympic swimming pool in March 1996. Many of the results of these tests can be found in [Leonhardt 96]. Missions consisted primarily of search missions and missions that transited through various

locations. Figure 61 shows a geographic plot of a search mission similar to the one depicted graphically in Figure 11. This mission was generated with the phase-by-phase mission specification facility. The mission includes two waypoints, a hoverpoint, a sonar search, and a GPS fix followed by a waypoint and a hoverpoint enroute to the recovery position.

Similar missions were generated for in-water tests using the means-ends analysis portion of the system and also by manually editing mission specification language files. Results obtained during in-water testing were similar to results obtained in the UVW.

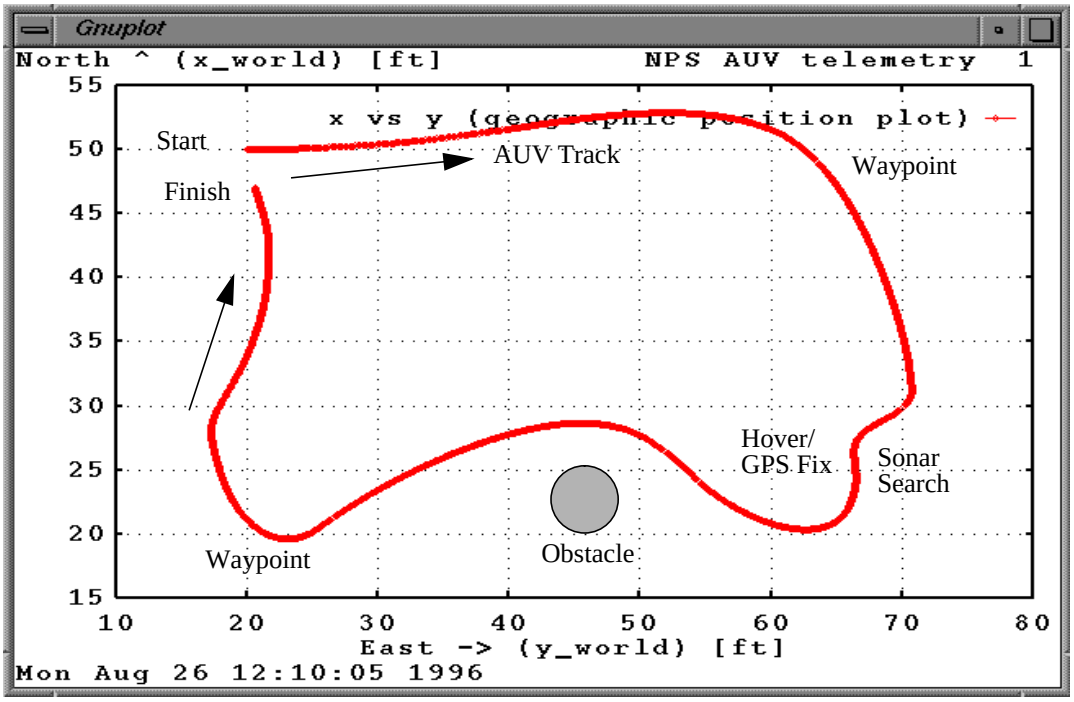


Figure 61: In-Water Results of an Automatically Generated Mission.

D. SUMMARY

Tests of features implemented during the conduct of this research were conducted in two distinct but complementary environments: the UVW and the real world. UVW tests were conducted to test features at all three layers of *Phoenix*' RBM implementation. These tests using methods described in the previous chapters resulted in successful recovery in

the tube in all but a few specific instances. The only failures occurred when *Phoenix* was initially positioned directly in front of or behind the recovery tube and was unable to acquire and track a tube edge because of the narrow sonar cross section.

In addition UVW tests were conducted to test modifications to the strategic level software and the mission planning expert system. These tests were highly successful and show that an expert system for AUV mission planning/generation is an extremely useful tool that greatly reduces mission generation time while improving mission reliability.

In water tests were conducted to verify low level sonar and vehicle control behaviors. These tests indicate that the sonar control modes described in Chapter IV are capable of reliably locating and tracking targets in the AUV environment. Also, target data obtained during sonar tracking can be used as the basis for maneuvering relative to objects being tracked. Maneuvering based on target edge tracking proved to be more responsive, but both sonar tracking modes were successfully used for station-keeping operations. station keeping was demonstrated relative to a 0.5 meter cylinder and also a recovery tube. Station keeping relative to the recovery tube using target edge-tracking proved precise enough to position *Phoenix'* nose in a position from which final recovery as described in Chapter IV was possible.

Finally, missions were generated using the mission planning expert system and successfully executed in the real world. Successful in-water tests of expert system generated missions verify the utility of the system.

In the following chapter, the conclusions of this research are discussed. Additionally, possible areas of future work are outlined. The conduct of this research has indicated numerous possible future projects, not only relating to the goals of this thesis, but also to broader research goals of the Center for AUV Research and other organizations.

VIII. CONCLUSIONS AND RECOMMENDATIONS

A. INTRODUCTION

Previous chapters of this thesis document the implementation and testing of features intended to support AUV recovery in a small tube. The purpose of this chapter is to draw conclusions based on the results of this research and to propose possible areas for future work. The following section discusses conclusions. The section concerning recommendations for future work is divided into twelve subsections. Each subsection discusses an area for possible future work that was directly or indirectly relevant to the conduct and results of this research.

B. RESEARCH CONCLUSIONS

The most obvious conclusion of this research is that sonar target tracking can be used as the basis for precision autonomous underwater maneuvering. UVW and in-water testing indicate that the precision of this maneuvering is sufficient for use throughout a recovery evolution. Further, UVW testing indicates that path planning and command generation can be implemented at higher levels of the RBM to use lower-level sonar-based maneuvering to plan and control recovery in a small tube.

A more general conclusion concerning the station-keeping behaviors is the applicability of sonar-based maneuvering to broader mission areas. The ability to take station relative to arbitrary objects will enable an AUV to become an active participant in the environment rather than merely an observer. This ability has potential applications in many types of missions that require interaction with objects in the environment. Underwater filming, sampling, repair and construction are just a few examples of potential AUV tasks that will require this capability.

The most significant conclusion concerning the mission planning expert system is that the use of a planning system such as this can greatly reduce mission generation effort and improve reliability. Additionally, artificial intelligence planning techniques can be used to create error-free missions that are guaranteed to accomplish the mission's high-level goals (assuming the goals are in fact possible and no catastrophic vehicle failures occur). It is this research aspect that may prove most beneficial to the field of AUV research in general. Only through the successful implementation of easy-to-use mission planning tools will AUVs evolve beyond their current role of academic and industrial research projects.

Another interesting conclusion that resulted indirectly from this research is that it is possible to satisfactorily control a real-time system using an unmodified Unix operating system. In *Phoenix* RBM implementation, hard-real-time (synchronous) tasks are executed on the GESPAC computer under the OS-9 real-time operating system, while soft-real-time (asynchronous) tasks are executed on a separate onboard computer running under the Sun Unix operating system. Hard-real-time tasks consist primarily of physical control of vehicle hardware. Soft-real-time tasks on the other hand, consist of high-level and medium-level mission control, planning, object classification and navigation. Most of what might be considered "intelligent" behaviors fall into the category of these soft-real-time tasks. By dividing tasks into hard and soft real time categories in this manner, *Phoenix* control software is implemented primarily on a system that many might consider unsuitable for control of a real-time system. The only drawback of this system is that it requires two separate onboard computers connected via LAN and relies on BSD socket communication.

Perhaps the broadest and most significant conclusion of this research results from the successful use of the UVW for the implementation of vehicle software. The robustness of the UVW allowed for deterministic testing of vehicle software in a benign environment. Features were implemented and comprehensively tested one at a time over a period of

approximately one year prior to in-water verification. By exhaustively testing software features in the UVW prior to attempting in-water tests, it was possible to conduct the bulk of in-water testing documented in this thesis over a two-week period. The UVW is a virtually unlimited resource, whereas power supply, hardware limitations, and logistics requirements limit the availability of the physical vehicle for in-water testing significantly. The in-water results of this research area depended heavily upon virtual world testing and would have not have been possible were it not for the UVW.

While the preceding conclusions are significant, they amount to little more than a first step towards recovery of AUVs using docking stations and submarines. A great deal of work remains. The research detailed in this thesis is preliminary in nature and is intended to begin dealing with issues involved with self recovery of an AUV in a confined space. The following section of this thesis details some of the work that remains.

C. RECOMMENDATIONS

1. General Tactical Level Tests and Enhancements

The most obvious area for future work is the verification of tactical level features through in-water testing. While the in-water tests described in Chapter VII verify the reliability and correctness of the low-level sonar and vehicle-control behaviors, these tests did not verify their use by the tactical level for successful recovery control. Along the same lines, the development of tactics that use these low-level behaviors to accomplish more general goals remains a topic for future work. In particular, these behaviors might be used to implement many of the advanced capabilities outlined in [Brutzman 96].

2. Sonar Tracking Behaviors

The next area of future work involves improvement of the sonar tracking behaviors. As depicted in Figures 50 and 51, under some circumstances it is possible for the sonar to

lose contact with the target being tracked. Even under ideal circumstances (error-free sonar data and a clutter free environment), it is possible for AUV motion to cause the sonar to lose track of the intended target.

Improvements in this area fall into two categories: improvements intended to prevent the sonar from losing contact with its intended target and improvements intended to enable to sonar to regain the target in the event of loss (which also involves recognition of target loss). If a full target scan is used, features can be extracted as the sonar sweeps across the target as documented in [Marco 96a]. This work might be augmented by the implementation of a simple learning algorithm to "memorize" target features on the first sweep to allow tracking of arbitrary objects without maintaining a target feature database. Successful implementation of this type of system will also need to deal with issues such as asymmetric objects which have different features when viewed from different angles. Successful implementation of this type of system to support the faster edge tracking sonar mode might involve periodic sweeps across the entire target to ensure that the proper target was being tracked (by recognizing and verifying the features recorded in the previous full target sweep).

The computational and storage requirements of this type of system will doubless necessitate its implementation at the tactical rather than the execution level. Since learning and object classification are involved, placement at the tactical level is a good match with ideal RBM tasking. Tactical-level implementation will allow implementation with minimal changes to the current execution level sonar modes. The addition of commands to the execution level command language to enable switching from edge tracking to target tracking for one sweep is probably the only change that is required at the execution level.

3. Sonar Classification

A third area for future work exists in improving *Phoenix*' sonar classification capabilities. While a significant amount of research effort has already been directed at this topic, current *Phoenix* sonar classification capabilities were not specifically intended to support recovery operations. Sonar classification research to date has been directed at the general case of translating sonar data into line segments and polygons representing generic objects [Brutzman 92, Campbell 96] and the specific case of classifying mine-like objects [Campbell 96, Marco 96a]. Complete implementation of recovery capabilities must include sonar classification of the intended recovery device. The most straightforward implementation of this capability is probably best performed by augmenting currently existing sonar classification algorithms.

4. AUV Tracking and Control

While the PD control laws discussed in this thesis are fairly effective, the experimental results of the previous chapter clearly show that they are far from ideal. In practice, sliding mode control laws such as those derived in [Marco 96a] have proven more accurate and responsive (albeit more computationally expensive) than PD control laws. Because of the current *Phoenix* execution level implementation demands and a weak 60830 CPU, the incorporation of sliding mode control laws (for station keeping and other control modes) may be possible only after optimization of the execution level as described later in this section. A more thorough discussion of various control modes and their suitability for AUV control during recovery can be found in [Chapuis 96].

In the interim an effort to tune PD control constants is necessary. The present *Phoenix* execution level uses PD control laws for all closed loop control modes (hover control, waypoint control, etc.) and will require tuning of constant terms of these control laws as well. UVW tests documented in the previous chapter demonstrate that PD control

laws can be used to obtain smooth motion along a planned path. Tuning of constants based on accurate *Phoenix* hydrodynamics to duplicate UVW results during in-water testing is therefore possible. Ideally, this work will proceed in parallel with UVW validation discussed later in this section, since accurate UVW hydrodynamics will enable tuning of control constants without requiring in-water tests.

Similarly, an effort to ensure standardization of control law nomenclature among the various Phoenix control modes is needed. This will facilitate the modification and tuning of existing control laws and the implementation of new control laws as well.

Finally, improvement of the mathematical model used for dead reckoning between sonar target updates is required. Errors introduced because of an inaccurate mathematical model lead to improper control response that can be counterintuitive to diagnose. This phenomenon was encountered on numerous occasions during the conduct of this research. The six DOF model of the UVW world demonstrates that accurate mathematical modeling of AUV hydrodynamics is possible and can run in real time. Improvement of the mathematical model represented by Equations 30, 31 and 32 will improve many aspects of the system that depend on accurate vehicle response in addition to those documented in this thesis. Hover behavior, navigation and sonar classification are three examples. Ideally, the need for a dead reckoning mathematical model can be eliminated entirely by the incorporation of an IMU as described later.

5. Ocean Current and a Moving Submarine

Because of the preliminary nature of this research, no real-world experimentation was conducted into the effects of current during recovery. UVW testing in the presence of a uniformly constant ocean current pointing in an arbitrary direction demonstrated that these control algorithms are robust, but before this research can be applied in an uncontrolled environment such as the open ocean, it will be necessary to research the

effects of time-varying current during station-keeping operations. Flow field issues (including their effects on control laws) need to be addressed in detail. Recovery in the torpedo tube of a moving submarine will require the resolution of numerous similar issues.

Research documented in this thesis dealt only with station keeping relative to stationary objects in a current-free environment. Nontrivial steady-state and varying currents (as well as target motion) are issues that will require significant research efforts.

6. Obstacle Avoidance During Recovery

The tactical level of *Phoenix* current software implementation conducts path planning and obstacle avoidance for waypoints and hoverpoints [Leonhardt 96]. Obstacle avoidance is not currently included in the recovery path planning discussed in Chapter V of this thesis. Vehicle safety during recovery operations requires that this issue be resolved. Features already existing at the tactical level will adapt fairly easily to this role. Recovery path planning can then use these enhanced features to plan an obstacle-free path.

7. Sensor and Hardware Issues

The most significant hardware issues encountered during the conduct of this research involved *Phoenix* navigation systems. While the Divetracker, GPS and differential GPS systems make up a robust navigational suite by most standards, they are all asynchronous in nature and provide navigational updates (fixes) every few seconds at best [McClarin 96]. The asynchronous nature of hardware-derived navigational data necessitates the use of the dead reckoning mathematical model described Chapter IV for real time navigation between fixes. The turbo-probe speed wheel mitigates this problem somewhat when the longitudinal motion of the vehicle exceeds approximately 0.25 feet per second, but even in this instance, the lateral motion of the vehicle must be accounted for (e.g., using the sideslip model discussed in Chapter VI).

As previously mentioned, the mathematical model currently in use contains inherent errors. Even a far more robust mathematical model is unlikely to account for external disturbances such as wave motion or uneven current effects. Navigational errors introduced by the mathematical model have a significant negative impact on all vehicle functions that rely on accurate navigational data. Among these are hover control, waypoint control, path planning, obstacle avoidance and object classification.

Research is currently ongoing into incorporating an IMU into *Phoenix* [Bachman 96, McGhee 95]. The successful implementation of accurate real-time navigation using an IMU is critical to many Center for AUV Research goals.

8. Strategic Level Enhancement

A possibility for future work at the RBM strategic level includes porting to other languages such as Ada95 (including the development of phase templates and automatic code generation programs) [Holden 95]. Additionally, work to be conducted at all three levels of the RBM will involve the expansion of the strategic level's primitive goal set and execution level command language [Brutzman 96]. Present primitive goals primarily support search missions. Future improvements might support run-time communication between *Phoenix* and its support platform, dynamic missions that can be modified as directed by the support platform, and more versatile interaction with located underwater objects (e.g., mine neutralization).

It may also be possible in the near future to implement a more dynamic strategic level that is capable of constructing portions of the DFA at run time. This might prove to be a very useful feature, particularly given the unpredictable nature of the marine environment. The previously addressed shortcomings of the means-ends analysis algorithm (particularly those concerning non-optimal solutions) may preclude its use in this manner. However, another planning algorithm such as search reduction through least

commitment, dependency-directed search, or meta-level planning [Tate 90a] may prove more applicable in this area. Planning systems such as Tweak [Chapman 90], MOLGEN [Stefik 90], NONLIN [Tate 90b], DEVISER [Vere 90], and FORBIN [Dean 90] for example, address both the efficiency and the temporal aspects of their solutions. Other issues to be dealt with before this type of self-modifying system is possible include missions of nondeterministic length and goal prioritization.

9. The Mission Planning Expert System

Possible future work might also be directed at improvements to the mission planning expert system. Obviously, as the functionality of *Phoenix* evolves, the mission planning expert system will need to evolve to take advantage of new vehicle capabilities. Additionally, work is ongoing to simplify the phase-by-phase mission-specification portion of the system and to improve the automatic mission-generation portion of the system. Ideally, the means-ends analysis algorithm will evolve to allow automatic generation of missions that take full advantage of the DFA structure of the strategic level. If the automatic mission planning is enhanced substantially, it may be possible to completely eliminate the phase-by-phase specification portion of the system without sacrificing flexibility. Other planning systems including those mentioned as possible run-time mission planners, may prove useful in this area as well. Modifications that may be applicable in the short term include modifying the phase-by-phase specification facility to incorporate error correction rather than simple error detection.

One potential mission planner implementation might involve a combinion of means-ends analysis with another search technique. For instance by assigning costs to the application of each operator, it is possible to determine the total cost of a solution. If a hybrid means-ends/search algorithm is applied in parallel to find operation sequences satisfying each top level goal, the costs of each partial solution can be compared. By

choosing the lowest cost option and reapplying the algorithm to the remaining goals (starting from the end state after application of the low-cost partial solution), a sequence of operations can be generated to accomplish all of the goals. This strategy amounts to a combination of means-ends analysis with best-first search [Winston 92]. Other search strategies may be useful in this sort of implementation as well.

Finally, the current version of the mission planning expert system is dependent upon the availability of Quintus Prolog and Prowindows. Possible future work to permit cross-platform independence includes porting the expert system to an HTML interface that can be run across a computer network. Such a system would probably involve a server-based script that executes queries against the rule base. Since such a system can be run from any platform using any web browser, such an approach provides complete platform and window system independence.

10. Operating System Issues

One of the conclusions drawn earlier in this chapter is that it is possible to control a real-time system using a standard Unix operating system. This does not however mean that it is necessarily desirable. The requirement of two computers and the reliance upon network communications may justify the transition to a single computer running a real-time operating system such as VxWorks [Wind River Systems 95]. On the other hand, since the execution level is not multi-threaded, it may be possible to control even the hard-real-time tasks by using a dedicated processor running a standard operating system such as Unix. This implementation still requires at least two onboard computers. As both of these alternatives are worth looking into, a comparative study may lead to interesting and insightful conclusions that will be relevant to a number of areas in addition to AUV research.

Another operating system issue involves the concurrent process implementation of the tactical and strategic levels. The Unix version under which the current system runs does not support shared memory between separate processes even when they are forked by a common process [Stevens 92]. This shortcoming necessitates the use of Unix pipes for interprocess communication. Newer versions of the Unix operating system now support shared memory [McKusick 96]. It may be worthwhile to rewrite the communications portions of the tactical level to use shared memory for some communications. This might improve the efficiency of the tactical level and may prove more readable as well. While it is probably impractical to replace all interprocess communication with shared memory, maintaining a single copy of the vehicle state vector as opposed to one copy for each tactical level module might prove very beneficial. A similar upgrade of the strategic and tactical levels that addresses the same issues might be their implementation in an inherently multi-threaded computer language such as Ada95 [Holden 95].

11. Code Optimization

Numerous features have been added to all three layers *Phoenix* RBM implementation during the conduct of this and other research. While these new features are quite robust, little effort has been expended to ensure efficiency of the overall system. As a result, the execution level in particular is only capable of maintaining a synchronous speed of just over five Hertz [Burns 96]. While this speed appears adequate, it leaves little room for future enhancements. The two possible solutions are the methodical optimization of source programs or an upgrade of execution level computer hardware (which likely will require a software rewrite anyway).

While the tactical level does not currently suffer from inefficiency to as great a degree as the execution level, optimization is still possible. At the strategic level, however, readability is considered more important than efficiency. This coupled with the relatively

small size of strategic level programs makes optimization of this software layer unnecessary.

12. Underwater Virtual World Improvement

A final area for possible future work is improvement of the UVW. As discussed in the previous section, the UVW proved to be an invaluable tool during the conduct of this research. Validation of the UVW by using real-world data to tune hydrodynamic model coefficients will make it even more valuable. Tuning of control constants based on an invalid vehicle response model inevitably leads to control problems that are extremely difficult to diagnose due to the large number of coefficient and variable terms in most control laws. Problems of this sort encountered during this research included both overdamped and underdamped control law coefficients. Complete validation and verification of all vehicle hydrodynamic response parameters is essential to the accurate modeling required for development of reliable control response. This is easily the most important area of research concerning the UVW since improving the accuracy of vehicle response in the UVW will have a dramatic effect on the reliability of developed software.

Another possible area of work concerning the UVW is the translation of the viewer from C++ and Open Inventor to Java [Gosling 96] and the Virtual Reality Modeling Language version 2.0 (VRML) [VRML Repository 96]. Translation ov network code to Java and graphics code to VRML will allow use of the UVW on any platform using a VRML-compatible web browser. It will also facilitate the sharing of world models by allowing objects to be imported into the UVW from anywhere on the Internet.

D. SUMMARY

This chapter discusses conclusions drawn based on this research and possible areas for future work. The first major conclusions of this work are that it is possible to use sonar information as the basis for precision maneuvering of an AUV and that higher level

behaviors can use this capability to control recovery operations. Additionally, precision maneuvering based on sonar data can be implemented in a general enough way to facilitate its use during various portions of a mission. It was further concluded that a mission planning expert system is an invaluable tool for the rapid development of complex missions that are free from errors and accomplish the mission's goals. Possibly the most important conclusion drawn from this research is the value of the UVW for rapid development and testing of vehicle software.

During the conduct of this research, numerous areas for potential future work were encountered. First is the development and in-water verification of tactical level behaviors that rely on the sonar and vehicle control behaviors described in this thesis. Other possibilities directly related to furthering this research area are the enhancement of the sonar tracking behaviors, sonar classification directed at identification of the recovery device, improvements to the current PD control laws and dead-reckoning mathematical model, and dealing with ocean current, moving targets and unexpected obstacles during recovery operations. Concerning the strategic level and the mission planning expert system, possible future work includes implementation of a more dynamic strategic level capable of limited run-time planning, improvement of the automatic mission generation and phase-by-phase mission specification facilities, and porting of the expert system to a platform-independent server based architecture accessible from the internet. Finally, more general areas of possible future work include a comparison of real-time and standard operating systems for AUV control, optimization of the execution and tactical level programs, validating the UVW based on real world data, and ultimately conversion of the UVW viewer to Java and VRML.

APPENDIX A. OBTAINING ONLINE RESOURCES

One of the Naval Postgraduate School's primary missions is to conduct research of value to the military and public. The Center for Autonomous AUV Research makes all of its significant work available online. Via the Internet, copies of all current software which is used to run *Phoenix* or the underwater virtual world are available for downloading. Other items available include graphics images, photographs, master's theses, Ph.D. dissertations, briefings, personnel listings, and other information relating to AUV research at NPS. [Leonhardt 96]

An electronic mail (e-mail) group (*auvrg@cs.nps.navy.mil*) is used to distribute message traffic to all members involved in the research group. Interested individuals group can subscribe to the e-mail group by filling out a request form which is available on the Center for AUV Research World Wide Web site (*http://www.cs.nps.navy.mil/research/auv*). [Leonhardt 96]

Files for the software can be downloaded individually or as a complete compressed archive package. In addition, numerous sample missions written in Prolog, C++, and the execution level scripting language are included. The complete download and installation instructions are available at the Software Reference site (*http://www.stl.nps.navy.mil/~brutzman/dissertation/software_reference.htm*l). The size of the complete uncompressed archive is approximately 15 MB.

APPENDIX B. EXECUTION LEVEL COMMAND LANGUAGE

This appendix contains the mission.script.HELP file. This file describes the syntax of the execution level language. This language is used to construct mission script files that can be read by the execution level or tactical level process to execute a scripted mission. Additionally, this language is used by the tactical level to direct the execution level in order to accomplish strategic-level goals. Finally, this language can be used interactively and entered from a command line to control the AUV using one command at a time. The mission.script.HELP file also contains instructions on how to construct and use mission script files. This file is available online at

http://www.stl.nps.navy.mil/~brutzman/dissertation/software_reference.html

This file is available individually or as part of the .tar package containing all *Phoenix* and UVW software.

```
//-----------------------------------------------------------------------//

          mission.script.HELP                                12 August 96

          Mission script syntax for NPS AUV execution level and tactical
          control, in water and in the NPS AUV Underwater Virtual World.

          http://www.stl.nps.navy.mil/~auv/execution/mission.script.HELP

          Don Brutzman                             brutzman@nps.navy.mil

//-----------------------------------------------------------------------//

This file describes how to change and create NPS AUV mission script files.
        Example mission.script files and the 'execution' program are in the
        ~/execution subdirectory.

Script commands are received by the AUV execution level (execution.c) from
        the tactical level during a mission, the operator at the keyboard, or
        read from the "mission.script" file.  Both tactical and execution can
        carry out mission scripts.

To run a new mission, copy a different existing mission file over file
        'mission.script' or edit the mission.script file for a new mission.

        Example:

        unix> cd execution
        unix> cp  mission.script.siggraph  mission.script
        unix> execution  virtual fletch.cs.nps.navy.mil
   or
        unix> execution  virtual fletch  mission  mission.script.siggraph

Many of the following commands will also work when invoked from the command
        line upon execution.  Detailed command line guidance is also available
        interactively using the online NPS AUV process launcher form at
        http://blackand.stl.nps.navy.mil/~auv/launcher/launcher.cgi

Numerous script keywords (and synonyms) are currently recognized. We have been
        generous in the use of synonyms in order to reduce the possibility of
        catastrophic spelling errors.  This approach might be further extended
        to include synonyms in other languages (French, Portuguese etc.)
        Hint hint!

Sections in this syntax help file:

        - Helm commands:  open-loop and closed-loop control
        - Navigation commands
        - Mission timing commands
        - Mission setup and configuration commands
        - Sonar commands
        - Miscellaneous commands


         |            |
---------+------------+-------------------------------------------------------
Keywords | Parameters |Description
Synonyms | [optional] |(all units are feet, degrees or seconds as appropriate)
---------+------------+-------------------------------------------------------
         |            |
         |            |

// Helm commands:  open-loop and closed-loop control ----------------------//

RPM            # [##] Set ordered rpm values to # for both propellers
SPEED          # [##] [ or independently set left & right rpm values
PROPS          # [##] to # and ## respectively]
PROPELLORS     # [##] maximum propellor speed is +- 700 rpm => 2 ft/sec
```

```
THRUSTERS-ON         Enable vertical and lateral thruster control
THRUSTERS
THRUSTERON
THRUSTERSON

NOTHRUSTER           Disable vertical and lateral thruster control
NOTHRUSTERS
THRUSTERS-OFF
THRUSTERSOFF

RUDDER        #      Force rudder to remain at # degrees, thrusters-off.
                     Value is for after rudder, negative command turns left.

DEADSTICKRUDDER[#]   Force rudder to remain at 0 [or #] degrees,
                     thrusters-off.

COURSE        #      Set new ordered course (commanded yaw angle)
HEADING       #
YAW           #

TURN          #      Change ordered course by # degrees
CHANGE-COURSE #      (positive # to starboard, negative # to port)

PLANES        #      Force planes to remain at # degrees, thrusters-off.
                     Value is for after planes, negative command points down.

DEADSTICKPLANES[#]   Force planes to remain at 0 [or #] degrees,
                     thrusters-off.

DEPTH         #      Set new ordered depth (commanded z)

PITCH         #      Set new ordered pitch (commanded theta angle).
THETA         #      Only effective during HOVERCONTROL.

ROTATE        #      open loop lateral thruster rotation control
                     at # degrees/sec

NOROTATE      disable open loop lateral thruster rotation control
ROTATEOFF
ROTATE-OFF

LATERAL       #      open loop lateral thruster translation control
                     at # ft/sec
                     (positive is to starboard, maximum is 0.5 ft/sec)

NOLATERAL     disable open loop lateral thruster translation control
LATERALOFF
LATERAL-OFF

// Navigation commands ----------------------------------------------------//

DIVETRACKER1 # ## ###Position of DiveTracker transducer 1
DIVETRACKER2 # ## ###Position of DiveTracker transducer 2
                     Still need to incorporate bearing to DiveTrackers.

GPS                  Proceed to shallow depth, take Global Positioning
GPSFIX               System (GPS) fix, restore ordered depth when done.
GPS-FIX              Control (thrusters, propellers/planes, combined)
                     is not modified.  Maximum fix time is 30 seconds,
                     at which time execution returns to previously
                     ordered depth.

GPS-COMPLETE         GPS fix complete, resume previously ordered depth.
GPS-FIX-COMPLETE

GYRO-ERROR       #   Degrees of error measured for gyrocompass.
GYROERROR        #   [GYRO + ERROR = TRUE]
```

```
DEPTH-CELL-BIAS  #      Feet of bias error measured for depth cell.
DEPTHCELLBIAS    #      [DEPTH CELL Z + BIAS = TRUE Z]
DEPTH-CELL-ERROR #
DEPTHCELLERROR   #

POSITION     # ## [###]reset vehicle dead reckon position to (x, y) or
LOCATION     # ## [###](x, y, z) = (#, ##, ###) at current clock time
FIX          # ## [###]This is a navigational position fix.  Receipt of a
                       POSITION/LOCATION/FIX command resets the execution
                       level dead-reckon position.  Note that depth value z
                        will likely be reset by depth cell if operational.

ORIENTATION  # ## ###reset vehicle orientation to
ROTATION     # ## ###(phi, theta, psi) = (#, ##, ###)

POSTURE #a #b #c #d #e #f
                     reset vehicle dead reckon posture to
                     (x, y, z, phi, theta, psi) = (#a, #b, #c, #d, #e, #f)

OCEANCURRENT  #x #y [#z]  Ocean current rate along North-axis, East-axis and
OCEAN-CURRENT #x #y [#z] [optional] Depth-axis (feet/sec)
                         (this is cartesian version of set and drift)

WAYPOINT    #X #Y [#Z]  [#rpm]
WAYPOINT-ON #X #Y [#Z]  [#rpm]
                     Point towards waypoint with coordinates (#X, #Y)
                     (depth #Z optional) (speed  #rpm optional).  You can
                     leave waypoint control by ordering course, rudder,
                     sliding-mode, rotate or lateral thruster control.

                     If in TACTICAL mode, execution reports STABLE when
                     waypoint is achieved.

STANDOFF       #     Change standoff distance for WAYPOINT-FOLLOW and HOVER
STAND-OFF      #     control
STANDOFFDISTANCE   #
STANDOFF-DISTANCE  #
STAND-OFF-DISTANCE #

HOVER   [#X #Y] [#Z] Hover using thrusters and propellers for longitudinal
                     and lateral positioning at specified or previous
                     waypoint

HOVER   [#X #Y] [#Z][#orientation]  [#standoff-distance]
                     Uses WAYPOINT control until within #standoff-distance
                     of HOVER point (#X, #Y, #Z), then switches to
                     HOVER control with [optional] final #orientation

                     Full speed (700 RPM) port & starboard is used if
                     AUV distance to WAYPOINT is > #standoff-distance + 10',
                     then slows to 200 RPM until within #standoff-distance,
                     then HOVER control.

                     HOVER without parameters is the preferred method of
                     slowing since backing down with negative propellers may
                     result in large sternway and severe depth excursions.

                     If in TACTICAL mode, execution reports STABLE when done.

HOVEROFF             Turn off HOVER mode
HOVER-OFF
HOVER_OFF

TARGETSTATION  #R #B [#Psi]
TARGET-STATION #R #B [#Psi]
                     Hover relative to a sonar target at range = #R and
                     target bearing #B from the AUV.  Commanded AUV
```

```
                      heading is #Psi (default is point at target).
                      Stationkeeping will use full target tracking
                      sonar mode

TARGETSTATION  #R1 #B1 #R2 #B2 [#Psi]
TARGET-STATION #R1 #B1 #R2 #B2 [#Psi]
                      Hover relative to sonar target.  Target currently
                      at range = #R1, bearing #B1 from AUV.  Commanded
                      range = #R2, commanded bearing = #B2, commanded
                      heading = #Psi (default is point at target).
                      Stationkeeping will use full target tracking
                      sonar mode

EDGESTATION    #R #B [#Psi]
EDGE-STATION   #R #B [#Psi]
                      Hover relative to a sonar target at range = #R and
                      target bearing #B from the AUV.  Commanded AUV
                      heading is #Psi (default is point at target).
                      Stationkeeping will use full target tracking
                      sonar mode

EDGESTATION    #R1 #B1 #R2 #B2 [#Psi]
EDGE-STATION   #R1 #B1 #R2 #B2 [#Psi]
                      Hover relative to sonar target.  Target currently
                      at range = #R1, bearing #B1 from AUV.  Commanded
                      range = #R2, commanded bearing = #B2, commanded
                      heading = #Psi (default is point at target).
                      Stationkeeping will use target edge tracking
                      sonar mode

TARGET-OFF            Turn off stationkeeping control mode
TARGETOFF
NO-TARGET
NOTARGET

TARGET-POINT          Commanded #Psi during stationkeeping will point
TARGETPOINT           directly at target center

NO-TARGET-POINT       Commanded #Psi during stationkeeping can be
NOTARGETPOINT         manually controlled using HEADING commands
TARGET-POINT-OFF
TARGETPOINTOFF

ENTERTUBE   # ##      Experimental control mode.  This tells execution level
ENTER-TUBE  # ##      that nose has entered the tube, drive the rest of the
                      way in using dead reckon for forward motion and sonars
                      (pointing to opposite sides) to maintain tube side wall
                      standoff.  Parameters:
                      #   How far forward to travel to be fully inside tube
                      ##  Tube orientation in degrees

// Mission timing commands ---------------------------------------------------
//

WAIT           #      Wait (or run) for # seconds (letting the robot execute)
RUN            #      prior to reading from the script file again

                      If in TACTICAL mode, execution ignores WAIT commands.

TIME           #      Wait (or run) until robot clock time #
WAITUNTIL      #      (letting the robot execute its current orders)
PAUSEUNTIL     #      prior to reading from the script file again

                      If in TACTICAL mode, execution ignores TIME commands.

TIMESTEP       #      change default execution level time step interval
TIME-STEP      #      from default of 0.1 sec to # sec
```

```
STEP                  loop for another timestep prior to reading script again.
SINGLE-STEP           Only useful in execution keyboard mode.

PAUSE                 temporarily stop execution until <enter> is pressed

REALTIME              run execution level code in real-time
REAL-TIME             (busy wait at the end of each timestep if time remains)

NOREALTIME            run execution level code as quickly as possible
NO-REALTIME
NONREALTIME
NOWAIT
NO-WAIT
NOPAUSE
NO-PAUSE

// Mission setup and configuration commands --------------------------------//

HELP                  Provide a list of available keywords
?                     (as specified in this HELP file).
/?
-?

//                    comments follow on this line which are not executed
/*                    note comments will still be spoken if AUDIO-ON
#                     pound sign also indicates a comment if in first column

// Three startup modes:[LOCATIONLAB] | TETHERED | UNTETHERED

LOCATIONLAB           Vehicle is operating in lab using virtual world.
LOCATION-LAB          This is default mode.

TETHER                command line switch only, used for in-water runs
TETHERED              set DISPLAYSCREEN=TRUE  and LOCACTIONLAB=FALSE

UNTETHER              command line switch only, used for in-water runs
UNTETHERED            set DISPLAYSCREEN=FALSE and LOCACTIONLAB=FALSE
NOTETHER
NO-TETHER

VIRTUAL     hostname tells execution level to open socket to virtual world
VIRTUALHOST hostname which is already running and waiting on 'hostname'
REMOTE      hostname VIRTUALHOST is a command line switch.  Example:
REMOTEHOST  hostname unix> execution  virtualhost fletch.stl.nps.navy.mil
DYNAMICS    hostname

TACTICAL     hostname tells execution level to open socket to tactical level
TACTICALHOST hostname which is already running and waiting on 'hostname'
STRATEGIC    hostname TACTICAL/STRATEGIC is a command line switch.  Example:
STRATEGICHOST hostnameunix> execution  tacticalhost fletch.stl.nps.navy.mil

MISSION filename      Replace 'mission.script' with 'filename' and start
SCRIPT  filename      the new mission.  Read tactical commands for execution
FILE    filename      level from filename.

TELEMETRY filename    Playback prerecorded telemetry data from filename.
                      Consider using with NOSCRIPT if no script file present.
                      dynamics should be run with selection
                      E dEad_reckon_test_with_execution_level

NOSCRIPT              Ignore script command file.  Selectively used
                      in combination with TELEMETRY data file playback.

KEYBOARD              read script commands from keyboard
KEYBOARD-ON

KEYBOARD-OFF          read script commands from mission.script file
NO-KEYBOARD
```

```
TRACE                 enable  verbose print statements in execution level
TRACE-ON

TRACEOFF              disable verbose print statements in execution level
TRACE-OFF
NOTRACE
NO-TRACE

LOOPFOREVER           repeat current mission when done.
LOOP-FOREVER          each repetition is called a 'replication.'

LOOPONCE              do not LOOPFOREVER, stop when end of script is reached
LOOP-ONCE

LOOPFILEBACKUP        back up output files during each loop replication
LOOP-FILE-BACKUP      to permit inspection while new files are written
                      the backup files are in execution directory:
                      output.telemetry.previous & output.1_second.previous

ENTERCONTROLCONSTANTS       start a keyboard dialog to enter
ENTER-CONTROL-CONSTANTS     revised control algorithm coefficients

CONTROLCONSTANTSINPUTFILE   read revised control algorithm coefficients
CONTROL-CONSTANTS-INPUT-FILEfrom file "control.constants.input"

BENCH-TEST            Simplified initial command-line parameter for quick
BENCHTEST             switch setting during Russ's control and prop testing.
BENCH

NOTEXT                Eliminate text display in command window
NO-TEXT               (useful for verbose/long runs in virtual world)

TEXT                  Turn text display in command window back on
TEXT-ON

QUIT                  do not execute any more commands in this script, but
STOP                  repeat the mission again if LOOP-FOREVER is set
DONE
EXIT
REPEAT
RESTART
COMPLETE
<eof> marker

KILL                  same as QUIT but also shuts down socket to virtual world
SHUTDOWN              'dynamics' process.

// Sonar commands -------------------------------------------------------//

SONAR725  #b #r #p    Set the bearing (#b), range (#r), and power (#p) of the
SONAR-725 #b #r #p    ST-725 sonar. In virtual world, bearing is necessary for
SONAR_725 #b #r #p    sonar model.  In water, this stores data in the state
ST725     #b #r #p    vector for replay and examination.

SONAR1000  #b         Manually control the ST1000 sonar bearing to #b degrees
SONAR-1000 #b         relative to Phoenix heading
SONAR_1000 #b
ST1000     #b

SCAN-WIDTH #          Total degrees for default ST1000 sonar scan, centered
SCANWIDTH  #          about bow

SONARTRACE            Enable verbose print statements in execution sonar code

SONARTRACEOFF         Disable verbose print statements in execution sonar code
```

```
SONARINSTALLED       Sonar interface installed, use them
SONAR-INSTALLED

NOSONARINSTALLED     Sonar interface not installed, don't use
NO-SONAR-INSTALLED

// Miscellaneous commands ------------------------------------------------//

AUDIBLE              enable  text-to-speech audio output
AUDIO
AUDIO-ON
SOUND-ON
SOUNDON
SOUND

SILENT               disable text-to-speech audio output
SILENCE
NOSOUND
SOUNDOFF
SOUND-OFF
AUDIOOFF
AUDIO-OFF
QUIET

SOUNDSERIAL          tell virtual world to pause while playing back sound
SOUND-SERIAL         (default)

SOUNDPARALLEL        tell virtual world to play sounds as parallel processes
SOUND-PARALLEL       (this may cause garbles if speeches play simultaneously)

EMAIL                ask  user for electronic mail address at mission start,
EMAIL-ON             send user an  electronic mail report  at mission finish
E-MAIL
E-MAIL-ON
EMAILON

EMAILOFF             disable electronic mail address query feature
EMAIL-OFF
E-MAILOFF
E-MAIL-OFF
NO-E-MAIL
NO-EMAIL
NO-E-MAIL
NOEMAIL

SLIDINGMODECOURSE    Sliding mode course control algorithm (not yet working)
SLIDING-MODE-COURSE

SLIDINGMODEOFF       Disable sliding mode course control algorithm ( " " " )
SLIDING-MODE-OFF

PARALLELPORTTRACE    enable trace statements for parallel port communications

WAYPOINTFOLLOW       Set mode to arrive at each waypoint before reading the
WAYPOINT-FOLLOW      next mission script command, i.e. continue towards each
WAYPOINTFOLLOWON     waypoint for however long it takes to reach the standoff
WAYPOINT-FOLLOW-ON   distance before pausing to read the next command.
                     Probably not needed anymore.

WAYPOINTFOLLOWOFF    Disables WAYPOINTFOLLOW mode
WAYPOINT-FOLLOW-OFF

//----------------------------------------------------------------------//
```

APPENDIX C. MISSION GENERATION EXPERT SYSTEM USER GUIDE

A. INTRODUCTION

This appendix consists of operating instructions for the mission planning expert system. Included sections include startup and initial operations, tactical level initialization file generation, the means-ends mission planning facility operation, phase-by-phase mission specification facility operation, and finally, executable code generation, compilation and execution.

B. STARTUP AND INITIAL OPERATIONS

The mission planning expert system requires Quintus Prolog version 3.2 and Prowindows [Weilemaker 94]. At present, these are only installed on `ai4.cs.nps.navy.mil` located in the artificial intelligence (AI) lab, however the expert system can still be run from anywhere on campus by starting a remote xterm. Log onto any Unix workstation on campus, start X-Windows (if necessary) and type the following from any `xterm` shell:

```
> xhost ai4.cs.nps.navy.mil
> telnet ai4.cs.nps.navy.mil
```

Log onto the `auv` account on `ai4.cs.nps.navy.mil` and change to the `strategic` directory. Start Prowindows by typing

```
ai4> cd strategic
ai4> xterm -display localmachine:0
```

For this command, *`localmachine`* is the machine upon which you are working. After executing this command, a new `xterm` will pop up. In this window type:

```
ai4> newprowin
```

Once Prowindows has started, the expert system is loaded and started by typing (including the period):

```
?- [mission_expert].
```

This will cause the software to load and start automatically. To later restart the expert system from prowindows (if it has been exited using the quit button), simply type

```
?- go.
```

Once the windowed mission generation expert system has started, use the menu button labeled **Available Charts** to choose the operating area for the mission you wish to generate. Clicking the left mouse button over the menu will cause the available operating area information (and the area maps) to cycle one at a time. Depressing the right mouse button over the menu button will invoke a drop-down menu displaying all possible operating areas. Dragging the mouse to the desired operating area and releasing the mouse button will cause it to load. The currently displayed map will always correspond to the currently loaded operating area. The operating area can be changed at any time, however, changing the current operating area while editing a mission will automatically clear all current mission data.

It is also a good idea at this point to enter the name of the desired output file on the **Output File Name** item (although the file name can be entered or changed any time prior to code generation). The file name must conform to standard Unix naming conventions. It is best to add the `.pl`, `.C`, `.cc`, or `.cpp` extension appropriate for the final output language, but this is not a requirement of the expert system itself. The naming convention used to date follows the form `mission.pl.myexample` for Prolog code and `mission_graph.C.myexample` for C++ code. Prior to compiling or executing the missions, the file should be copied into `mission_graph.C` or `mission.pl` as appropriate. Currently the autogenerated C++ code compiles and runs under the Silicon

Graphics (SGI) Irix operating system on any campus SGI workstation. The autogenerated Prolog runs on the Voyager laptop only (`tony.cs.nps.navy.mil`).

To start the different system facilities, the **Available Operations** menu buttons (from the main menu shown in Figure 62) are used. Starting any system by selecting the **Phase by Phase Generation** or **Means Ends Generation** button will automatically end any system facility currently being executed and will clear all data from that operation. Use of the **Create Initialization File** or **Modify Current Mission** will start the appropriate facility, but will not clear data from memory if another system facility was interrupted. Directions for use of each of the system facilities are provided in the following sections.

C. TACTICAL LEVEL INITIALIZATION FILE GENERATION

When running all three RBM layers, the tactical level requires an initialization file. This file contains information such as the initial vehicle posture, the locations of the Divetracker units, and the gyro error. To start generate this file, use the left mouse button to click the **Create Initialization File** button under **Available Operations**. At this point, the data is entered using the data input window shown in Figure 63. Data must be entered using the sliding bars (point and click using the map is not enabled). Locations are in X, Y coordinates corresponding to the grid overlaying the operating area map. When finished, use the left mouse button to press the **Done** button on the **Initialization Parameters** data entry window. The information will be saved in a file called `initialization.script`. To end the facility without creating the initialization file, use the left mouse button to press the **Cancel** button. NOTE: If more than one mission is to be generated, copy each `initialization.script` file to another file before creating the next one to avoid overwriting.

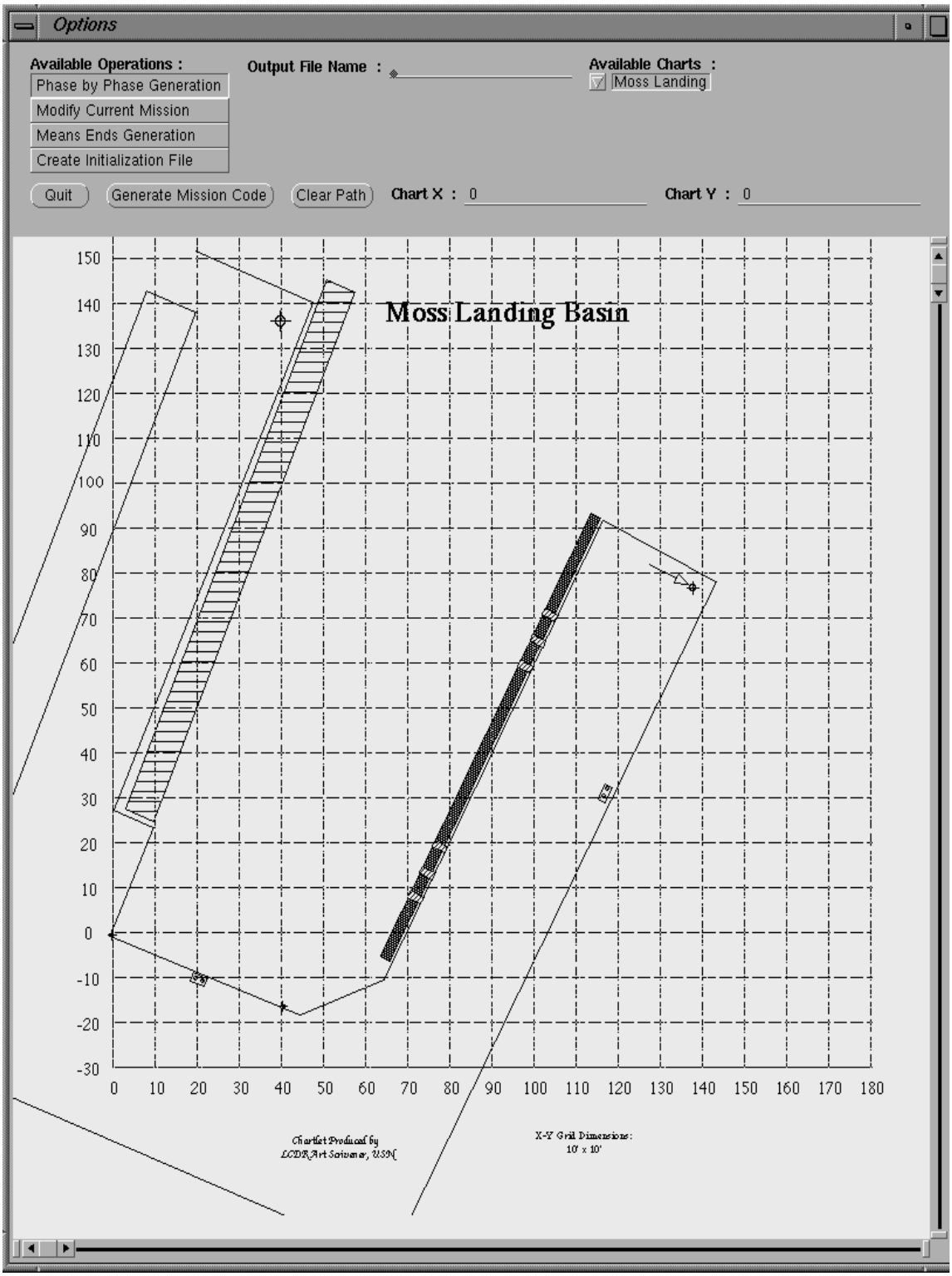


Figure 62: Mission Planning Expert System Main Window.

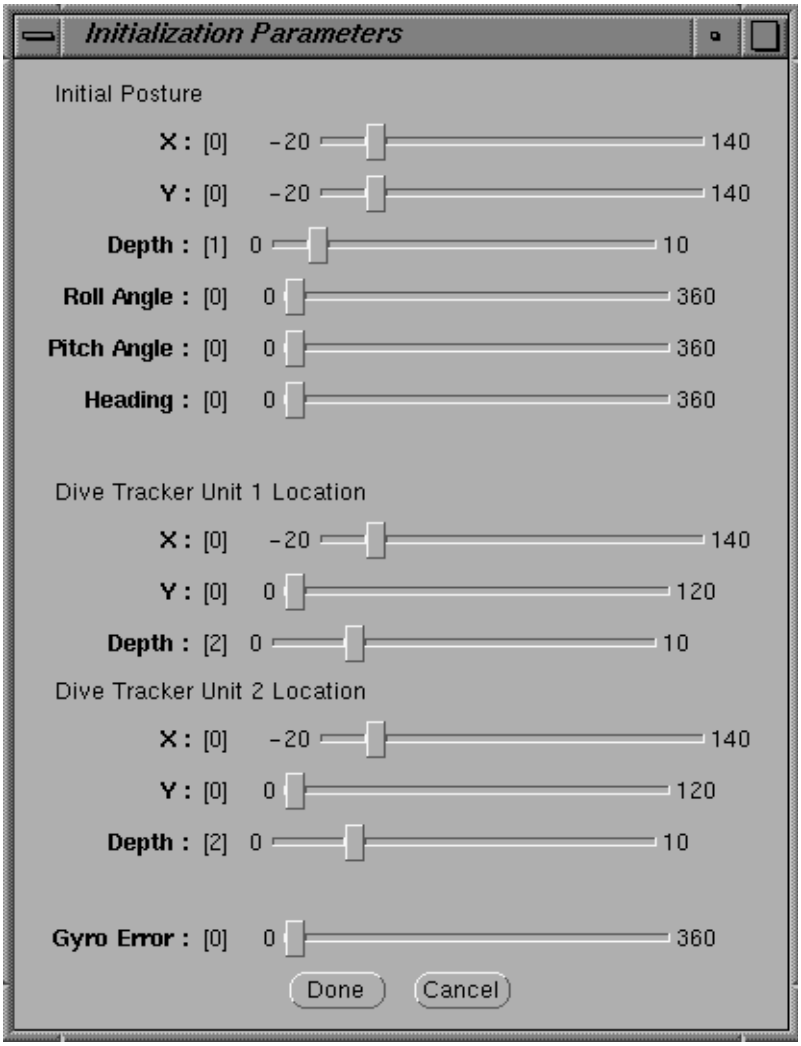


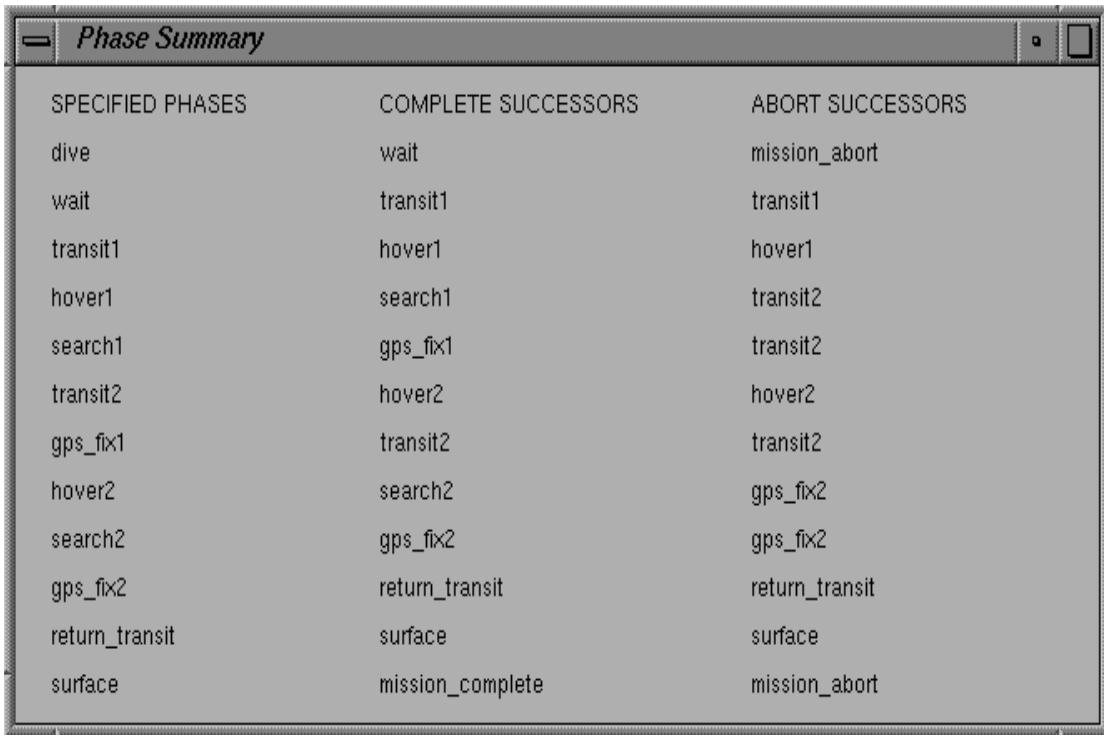
Figure 63: Initialization Parameters Data Input Window.

D. PHASE-BY-PHASE MISSION SPECIFICATION

1. Entering New Phases

To start the phase-by-phase mission specification facility, use the left mouse button to click the **Phase by Phase Generation** button under **Available Operations**. At this point, **Phase Type** and **Phase Summary** windows will be created. The **Phase Type** window is used to specify the type of phase that is to be entered, while the **Phase Summary** window will display a state table representation of the mission as it is entered in. The **Phase Type** and **Phase Summary** windows are depicted in Figures 64 and 65 respectively. To enter a phase, use the left mouse button to choose the appropriate type of phase on the **Phase Type** menu. Data entry for phase parameters for each type of phase is via windows that vary depending on the type of phase (all are similar to the data entry window depicted in Figure 66). The following section provides a brief summary of what each type of phase will accomplish and what parameters must be specified.

Figure 64: Phase Type Input Window.



Phase Summary

SPECIFIED PHASES	COMPLETE SUCCESSORS	ABORT SUCCESSORS
dive	wait	mission_abort
wait	transit1	transit1
transit1	hover1	hover1
hover1	search1	transit2
search1	gps_fix1	transit2
transit2	hover2	hover2
gps_fix1	transit2	transit2
hover2	search2	gps_fix2
search2	gps_fix2	gps_fix2
gps_fix2	return_transit	return_transit
return_transit	surface	surface
surface	mission_complete	mission_abort

Figure 65: State Table Summary of a Mission Specified Phase-by-Phase.

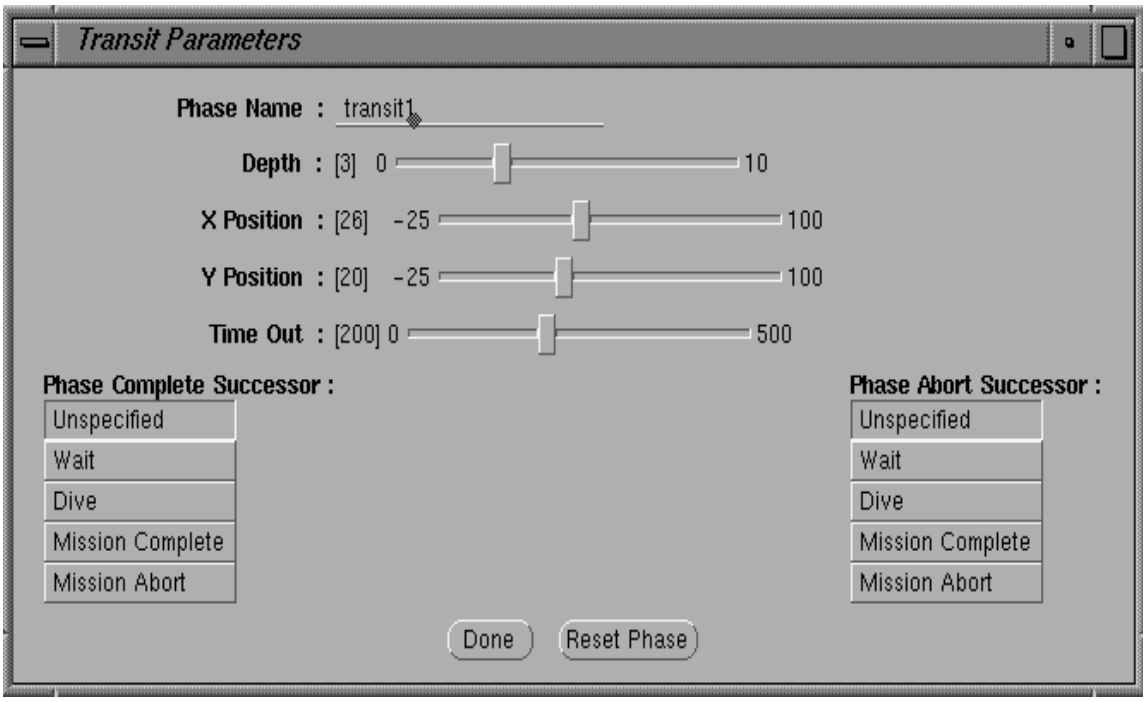


Figure 66: Data Input Window for Transit Phase Specification.

Depth Change: Change to new depth while hovering or after transiting. New depth is specified.

Course Change: Change to new course while hovering or after transiting. New course is specified.

Transit: Use maximum RPM to transit to a new location. Vehicle will not stop upon reaching this new location, but will drive through. Transit location is specified as an (X, Y) position and depth is specified as well.

Hover: Transit to a new location and establish a hover. Hover location is specified as an (X, Y) position. Hover depth and hover heading are also specified.

GPS Fix: Obtain a global positioning system fix. No special parameters are required.

Rotate Sonar Search: Conduct a sonar search from a specified location by rotating the sonar 360 degrees. Search location is specified as an (X, Y) position. Search depth is specified as well.

Rotate AUV Search: Conduct a sonar search from a specified location by rotating the AUV 360 degrees while maintaining a fixed forward sonar bearing relative to the AUV. Search location is specified as an (X, Y) position. Search depth is specified as well.

Wait: Continue with current operation (eg., hover) for a specified period of time. Time to wait is specified.

Recover in Tube: Perform a recovery in a recovery tube. Location of recovery tube is specified as an (X, Y) position. Recovery tube depth and heading are also specified.

In addition to the above data required by each individual type of phase, all types of phases require the following information:

Phase Name: This can be made up of numbers and letters and is typed in by the user. No blanks are allowed, and the first character cannot be a capital letter (if Prolog code is to be generated).

Time Out: This is the amount of time (in seconds) that the phase has to succeed.

Phase Complete Successor: The name of the phase to execute upon successful completion of the phase currently being entered.

Phase Abort Successor: The name of the phase to execute upon failure of the phase currently being entered.

The means of data entry varies depending on the type of data. Numerical data can be entered using the sliders. The slider ranges for X positions, Y positions, and depths are defined according to the current operating area. Positions for hovers, transits and searches can also be entered by clicking the desired position on the area map with the left mouse button. The name of the phase is typed in by the user at the **Phase Name** text entry location.

Push button menus are used to enter the **Phase Complete Successor** and **Phase Abort Successor**. The names of all phases that have been specified previously will have pushbuttons on both menus (in addition to selections for **Mission Abort** and **Mission Complete**). To set one of these phases as the complete or abort successor, simply use the left mouse button to select the desired phase. If the desired successor phase has not yet been defined, use the left mouse button to select the **Unspecified** menu item. A new data entry window for specifying the name of the successor phase will then be displayed. Enter the intended name of the successor phase on the **Name** blank and use the left mouse button to press the **Ok** button. This phase will need to be specified later (using the correct name) or an error will be generated when the system parses the mission prior to code generation.

Any phase information can be changed after being entered simply by re-entering it. When all required phase information has been entered, use the left mouse button to press the **Done** button on the data entry window. The phase will then be stored in memory and displayed in the state table of the **Phase Summary** window. To cancel entry of the current phase, use the left mouse button to press the **Reset Phase** button on the data entry window.

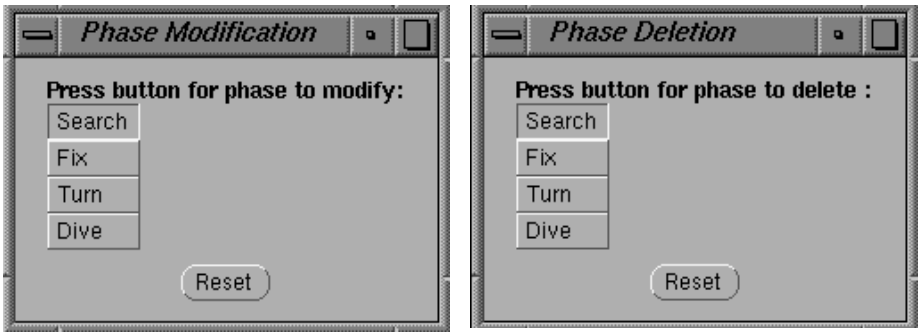
NOTE: Mission specification can be interrupted at any time to create the tactical level initiatialization file without losing phases that have been specified previously (restarting the phase-by-phase specification facility is discussed later). If the means-ends mission generation facility is invoked or the phase-by-phase mission specification facility is restarted improperly, however, all previously specified phases will be deleted from memory.

2. Modifying and Deleting Phases Specified Phases

To modify a phase that has been previously entered without deleting it, use the left mouse button to select the **Modify Phase** button in the **Phase Type** window. A **Phase Modification** window similar to the one shown in Figure 67 will be displayed. Use the left mouse button to select the name of the phase that you wish to modify from the menu in this window (or the **Reset** button to remove the window without modifying a phase). A data entry window for this phase (with the phase data as previously entered) will be displayed. Data can be entered using this window as if the phase were being initially specified. To remove the window without changing the phase, use the left mouse button to press the **Reset Phase** button, or press the **Done** button to store the changed phase definition.

To delete a phase that has been previously entered, use the left mouse button to select the **Delete Phase** button in the **Phase Type** window. A **Phase Deletion** window similar to the one shown in Figure 67 will be displayed. Use the left mouse button to select the button corresponding to the phase to be deleted (or the **Reset** button to remove the window without deleting a phase).

If no phases have been previously entered into the system and the **Modify Phase** or **Delete Phase** button is depressed, an error window will be displayed. Use the left mouse button to press the **Ok** button in the error window to clear the error message. When done specifying the mission, do not press the **Cancel** button. Rather follow the instructions for generation of executable code provided in Section F.



(a) (b)

Figure 67: (a) Phase Modification and (b) Phase Deletion Windows.

3. Phase Errors

If the data entered for a phase is invalid or incomplete when the **Done** button is pressed, an error message will be displayed describing the type of error in an **Invalid Phase** window similar to the one shown in Figure 68. Phases containing errors will not be accepted by the system. To clear the error message use the left mouse button to press the **Ok** button in the error window. Phase data can then be entered or changed as appropriate, or the **Reset Phase** button can be used to cancel phase entry.

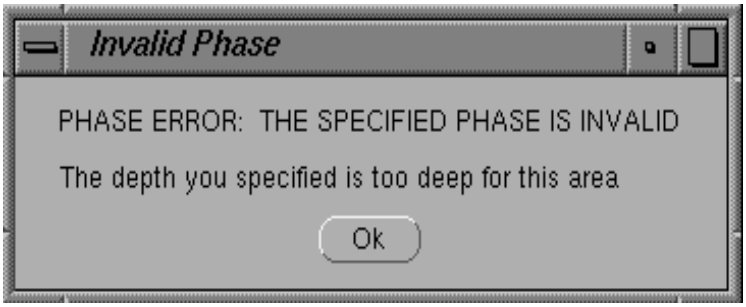


Figure 68: Invalid Phase Error Report Window.

4. Mission Modification

If the **Phase Type** window is not present and mission specification is not complete, mission specification can be continued without clearing previously specified phases from

memory by using the left mouse button to select the **Modify Current Mission** button on the **Available Operations** menu of the main system menu. Loss of the **Phase Type** window can have a number of causes. The most common is accidental (or intentional) use of the **Cancel** button on the **Phase Type** window. The **Phase Type** window will also be removed if the **Create Initialization File** facility is invoked. It may also be the case that the window is simply hidden by another window on the screen. Regardless, use of the **Modify Current Mission** button will generate a new **Phase Type** window. Mission specification can then proceed. If the **Modify Current Mission** button is pressed when no specified phases are contained in memory (either no phases have been specified or memory was cleared by the invocation of one of the other system facilities), an error message will be displayed. To clear the error message, use the left mouse button to press the **Ok** button on the error window.

5. Code Generation

Once mission specification is complete, code can be generated by following the instructions in Section F, Executable Code Generation, Compilation and Operation.

F. MISSION GENERATION WITH MEANS-ENDS ANALYSIS

1. Overview and Launch and Recovery Position Specification

The means-ends facility is used to automatically generate complete missions by specifying the AUV launch position, recovery mission, type of recovery and the goals of the mission. To invoke this facility, use the left mouse button to press the **Means Ends Generator** button on the **Available Operations** menu of the main window. The system will display the window shown in Figure 69. Use the sliders in this window to enter the AUV launch and recovery positions (point and click is not enabled). This window is also used to define the type of recovery to be executed at the end of the mission and enter locations to be searched during the mission.

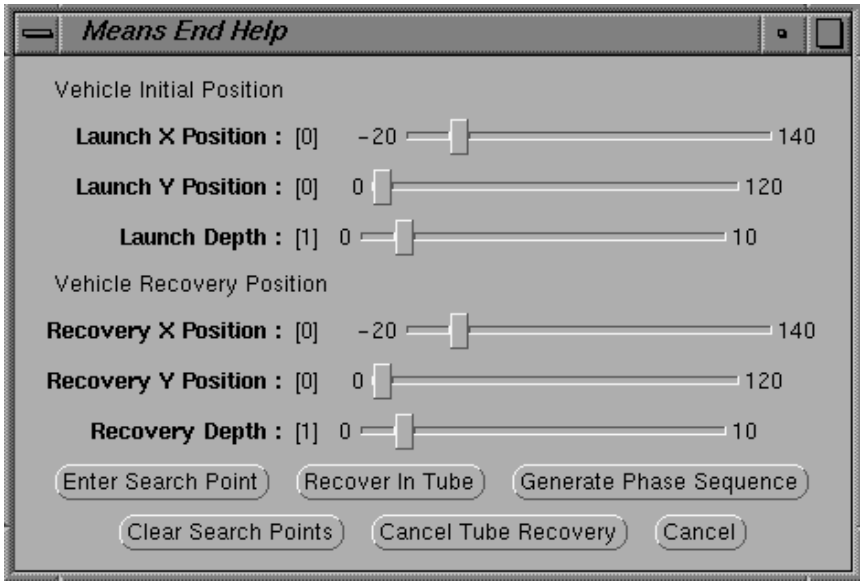


Figure 69: Means-Ends Mission Generator Facility Main Window.

2. Specifying the Recovery

The default AUV recovery is simply to surface once reaching the recovery location. Also available is a recovery in a tube. To specify this type of recovery, use the left mouse button to press the **Recover In Tube** button. The system will then request recovery tube data using the window shown in Figure 70. The (X, Y) position of the recovery tube can be entered using the sliders by using the left mouse button to select the appropriate position from the area map. **Tube Depth** and **Tube Entry Heading** (the heading of the AUV when entering the tube) must be entered using the provided sliders. Once the data has been entered, use the left mouse button to press the **Store Data** button to save the recovery tube information. To cancel entry of recovery tube data without saving, use the left mouse button to press the **Cancel** button on the **Recovery Tube Data** window. Once the recovery tube data has been entered and saved, it can be changed simply by re-entering it. To cancel a tube recovery that has been previously specified, simply use the left mouse button to press the **Cancel Tube Recovery** button on the **Means End Help** window.

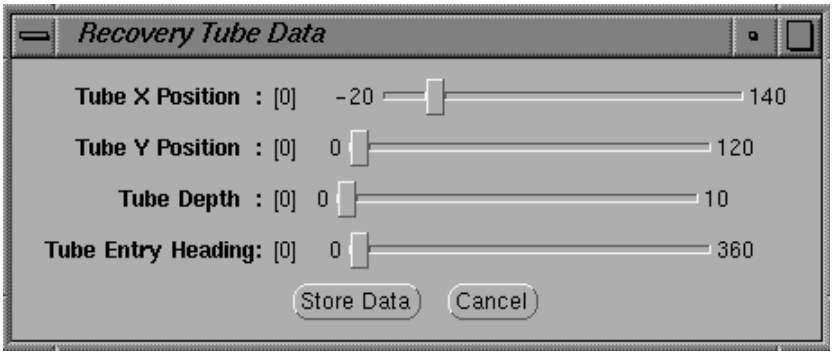


Figure 70: Recovery Tube Data Entry Window.

When the position and orientation of the recovery tube have been specified and saved, the system will automatically update the vehicle recovery position information in the **Means End Help** window. This information can be manually changed using the sliders if desired, however the automatically updated position is one from which recovery is possible. Manually changing this position may cause an error when the mission is parsed prior to code generation if the manually selected position is not within the system defined range limitations.

3. Specifying Search Points

At present the primary mission goal supported by the system is the conduct of searches from user-specified locations. Points at which to conduct searches are entered one at a time. An unlimited number of search points can be entered into the system in any order. Routing to a search point can also be specified.

To enter a search point, use the left mouse button to press the **Enter Search Point** button in the **Means End Help** window. The system will display the **Search Point Data** window shown in Figure 71. To enter the location to be searched, use the sliders in the window or use the left mouse button to indicate the search location on the area map. This will cause update of the search point location and the transit point location fields in the **Search Point Data** window. The search depth must be entered using the slider. To specify

routing to the search point, use the transit point location field. The generated mission will cause the AUV to transit through this point prior to establishing a hover at the search location (if the search point and transit point are collocated, the AUV will transit directly to the search point and establish a hover). This point can be specified using the sliders or by indicating the transit point location on the area map using the right mouse button (after the search location has been specified manually or with the left mouse button). To save the search data, use the left mouse button to press the **Store Point** button in the **Search Point Data** window. To cancel search point entry without saving the search point data, use the **Cancel** button in the **Search Point Data** window. Once a searchpoint has been entered into the system and saved, it cannot be saved. All search points can however be deleted from memory by using the left mouse button to press the **Clear Search Points** button in the **Means End Help** window.

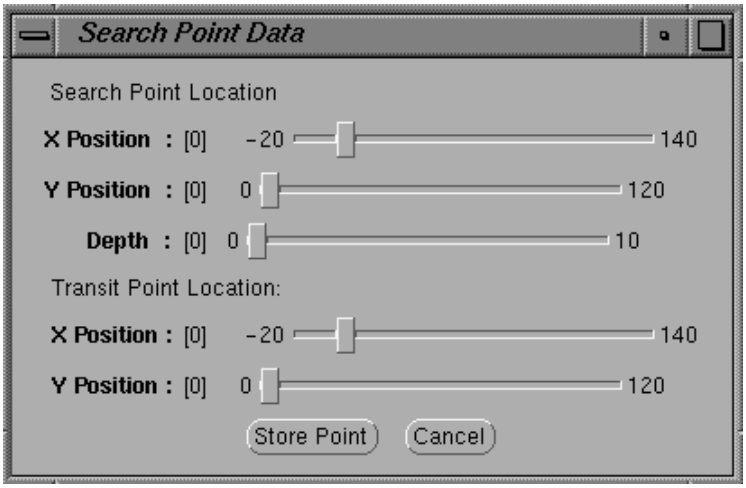


Figure 71: Search Point Data Entry Window.

4. Computing a Sequence of Phases

Once the launch position, recovery position, type of recovery, and searchpoints have been specified, the system can be used to generate a sequence of phases that will accomplish the mission. To invoke this feature, use the left mouse button to press the

Generate Phase Sequence button in the **Means End Help** window. The system will then generate the sequence of phases and display a textual description of the resulting mission in a window similar to the one shown in Figure 72. In addition the path of the mission will be depicted on the area map of the main system window.

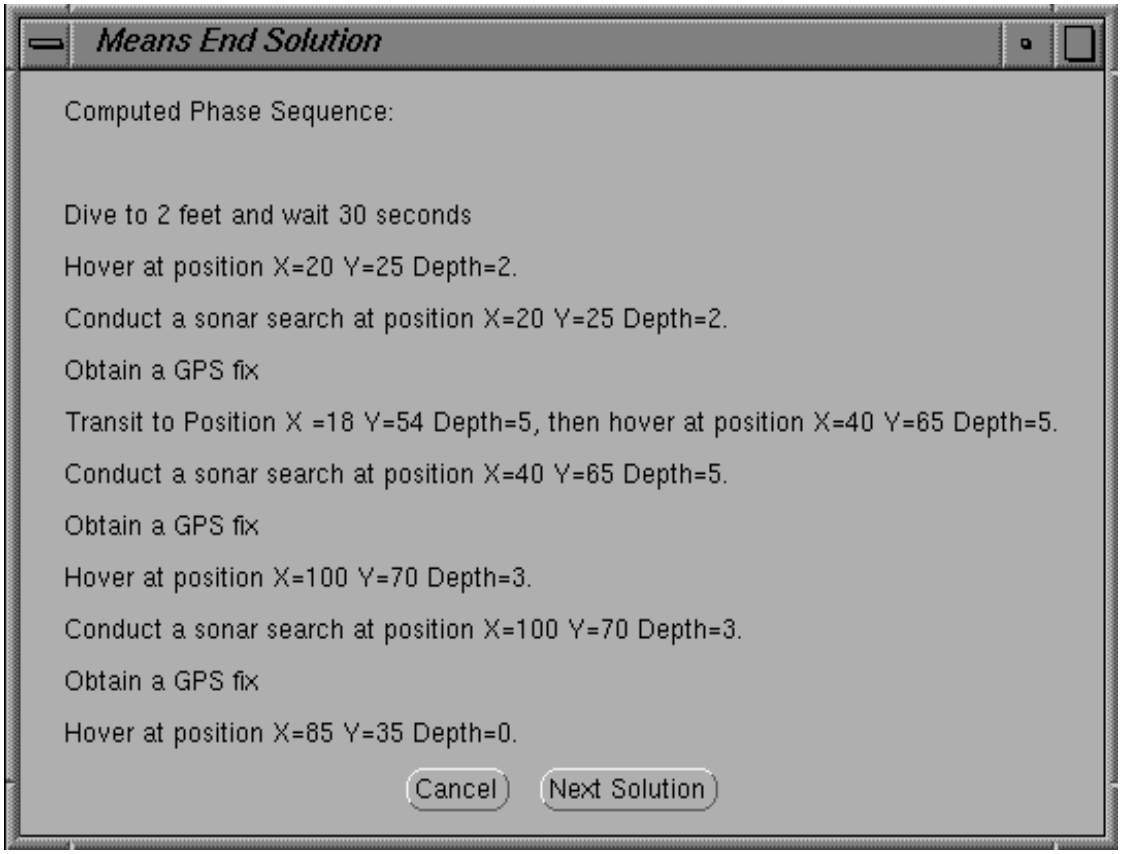


Figure 72: Sample Means-Ends Analysis Mission Solution Window.

Since means-ends analysis is not guaranteed to find the best solution first, the capability exists to cycle through all possible solutions one at a time. To generate another solution using means-ends analysis, use the left mouse button to press the **Next Solution** button in the **Means End Solution** window. A new solution will be computed and displayed textually in a **Means End Solution** window and geographically on the area map of the main window.

5. Code Generation

Once mission specification is complete and a satisfactory solution has been obtained, code can be generated by following the instructions in Section F, Executable Code Generation, Compilation and Operation.

F. EXECUTABLE CODE GENERATION, COMPILATION AND OPERATION

1. Code Generation

Executable code can be generated based on a mission specified using the phase-by-phase mission specification facility or the means-ends mission generation facility. If the phase-by-phase mission specification facility was used to specify the mission, the generated executable code will correspond to the mission described in the **Phase Summary** window. If the means-ends-analysis mission generation facility was used, the generated executable code will correspond to the mission described in the current **Means End Solution** window.

To generate executable code simply use the left mouse button to press the **Generate Mission Code** button in the main system window. If the phase-by-phase mission specification facility was used to specify the mission, the system will request the name of the first phase of the mission. Simply use the left mouse button to select the appropriate first phase from the menu displayed. If the means-ends mission generation facility was used, this step is not required. In either case, the system will parse the mission to check for errors (loops in the graph, no mission complete specified, unspecified phases, etc.) prior to generating code.

If errors are detected, a window similar to the one in Figure 73 will be displayed. To clear the error message, use the left mouse button to press the **Ok** button in the **Phase Error** window. The mission can be then edited using the phase-by-phase mission specification facility or means-ends mission generation facility as appropriate (the only

errors that can be introduced by the means-ends mission-generation facility are caused by manually changing the recovery position when a tube recovery is requested as described above). Editing of missions with errors is conducted as if no attempt to generate code had been made.

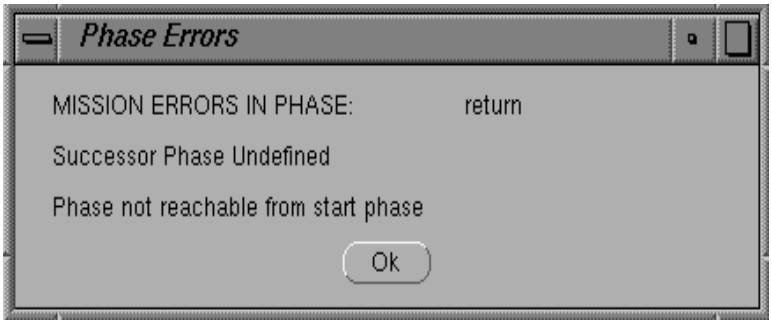


Figure 73: Error Window for Detected Mission Errors.

If no errors are detected during parsing, the window displayed in Figure 74 will be displayed. To select the desired language for output, use the left mouse button to press the **Prolog** or **C++** button as appropriate. The output file will be stored in the current directory (`~auv/strategic`) and will be named according to the Output File Name entry. If Prolog code is generated, an additional file will be created with a `standalone_` added to the beginning of the name. These two files are equivalent except that running the standalone file will make queries to the user rather than the tactical level. To clear this window without generating code, press the **Cancel** button.

Figure 74: Output Language Selection Window.

2. Compiling and Running the Mission

a. *The Tactical Level Initialization File*

To run a mission, generated files must be transferred from `ai4.cs.nps.navy.mil` to the appropriate directories on file system of the machine upon which the tactical level is to run. All files should be transferred as ascii files using the Unix ftp facility.

The `initialization.script` file should be transferred to the `~auv/tactical` directory (or `~auv/uvw/tactical` on the Sun Voyager) regardless of the language generated by the expert system. When running, the tactical level requires the file to be called `initialization.script`, but for storing multiple initialization files, it is ok to use different names (eg., .extensions for describing what mission the initialization file is for). An example ftp session is shown below. This session is conducted from the strategic directory on `ai4.cs.nps.navy.mil` (from the `xterm` window).

```
ai4> ftp gravy3.cs.nps.navy.mil
username:  auv
password:
ftp> cd strategic
ftp> put mission_graph.C.example
ftp> cd ../tactical
ftp> put mission.pl.example
ftp> put initialization.script initialization.script.example
ftp> put command_strings command_strings.example
ftp> quit
```

In this example, it is assumed that a Prolog and a C++ mission were generated. If only the C++ version was created, the `put mission.pl.example` command can be omitted. If only a Prolog version was created, the cd strategic and put

`mission_graph.C.example` can be omitted and the `cd ../tactical` command should be modified to

```
ftp> cd tactical
```

b. ***Prolog Execution***

If Prolog code was generated, the standalone file (standalone_ prefix) should be placed in the `~auv/strategic` directory on the target machine. The mission file itself should be placed in the `~auv/tactical` directory. Either of these files can have any name so long as they end in the Prolog `.pl` extension (and contain no other periods). To run the standalone strategic level, switch to the `~auv/strategic` directory on the appropriate machine and start Prolog (or Prowindows). Load the mission into memory by typing the file name (minus the `.pl` extension) in brackets followed by a period:

```
?- [mission].
```

To run the mission, type:

```
?- execute_mission.
```

Answer the strategic level queries by typing yes or no. To run the mission in the vehicle or the virtual world (tactical level attached), switch to the `~auv/tactical` directory. It is probably a good idea to make sure the proper version of the tactical level has been compiled. To do this type:

```
> make strategic
```

Start Prolog and load the mission file into memory by typing the file name (minus the `.pl` extension) in brackets followed by a period, just as for the standalone version of the strategic level. The mission is started in the same way as the standalone version as well:

```
?- execute_mission.
```

c. *C++ Compilation and Execution*

If the expert system generated a C++ strategic level mission, it must be compiled prior to running the mission. The generated C++ file should be transferred to the `~auv/strategic` directory on the appropriate file system. Before compiling, the C++ mission file must be named `mission_graph.C` and must be located in the `strategic` directory. To compile the mission, from the `auv` directory and type:

```
> cd strategic
> cp mission_graph.C.example mission_graph.C
> cd ../tactical
> make strategic_cpp
```

The executable file upon completion of the make will be located in the `tactical` directory and will be called `strategic`. To make a standalone version of the mission, type:

```
> make strategic_standalone
```

from the `tactical` directory. The executable standalone mission will be placed in the tactical directory and will be called `ood_test`. Either the standalone version of the strategic level or the full RBM version are invoked by typing the name of the executable file on the command line.

G. EXITING THE SYSTEM AND INDIVIDUAL FACILITIES

To exit the system at any time during execution, use the left mouse button to press the **Quit** button in the main window. The Prowindows interpreter can be exited by typing

```
?- halt.
```

Most system windows provide a **Cancel** button. Pressing this button using the left mouse button will cancel the operation without performing it. As a rule, use this button to

cancel operations and destroy windows rather than the minus button in the upper left window corner.

LIST OF REFERENCES

Bachmann, E. R., McGhee, R. B., Whalen, R. H., Steven, R., Walker, R. G., Clynch, J. R., Healey, A. J. and Yun, X. P., "Evaluation of an Integrated GPS/INS System for Shallow-Water AUV Navigation (SANS), *Proceedings of the1996 Symposium on Autonomous Underwater Vehicle Technology*, pp. 268-275, Monterey California, June 1996.

Bellingham, J. G., Consi, T. R., Tedrow, U. and DiMassa, D., "Hyperbolic Acoustic Navigation for Underwater Vehicles: Implementation and Demonstration," *Proceedings of the 1992 Symposium on Autonomous Underwater Vehicle Technology*, pp. 304-309, Washington D.C., June 1992.

Bellingham J. G., Goudey, C. A., Consi, T. R., Bales, J. W., Atwood, D. K., Leonard, J. J. and Chryssostomidis, C., "A Second Generation Survey AUV," *Proceedings of the 1994 Symposium on Autonomous Underwater Vehicle Technology*, pp. 148-156, Cambridge Massachusetts, July 1994.

Boorda, J. M., *Mine Countermeasures: An Integral Part of Our Strategy and Our Forces*, White Paper, U.S. Navy, Washington D.C., December 1995.

Brutzman, D. P., Compton, M. A. and Kanayama, Y., "Autonomous Sonar Classification Using Expert Systems," *Proceedings of the Oceans 92 Conference*, pp. 554-559, October 1992.

Brutzman, D. P., *A Virtual World for an Autonomous Undersea Vehicle*, Ph.D. Dissertation, Naval Postgraduate School, Monterey California, December 1994. Available at *http://www.stl.nps.navy.mil/~brutzman/dissertation*

Brutzman, D. P., "Virtual World Visualization for an Autonomous Underwater Vehicle," *Proceedings of the Oceans 95 Converence*, pp. 1592-1600, San Diego California, October 1995. Available at *ftp://taurus.cs.nps.navy.mil/pub/auv/oceans95.ps.Z*

Brutzman, D., Burns, M., Campbell, M., Davis, D., Healey, T., Holden, M., Leonhardt, B., Marco, D., Mcclarin, D., McGhee, R. and Whalen, R., "NPS Phoenix AUV Software Integration and In-Water Testing," *Proceedings of the1996 Symposium on Autonomous Underwater Vehicle Technology*, pp. 99-108, Monterey California, June 1996. Available at *ftp://taurus.cs.nps.navy.mil/pub/auv/auv96.ps*

Burns, M., *An Experimental Evaluation and Modification of Simulator-Based Vehicle Control Software for the Phoenix Autonomous Underwater Vehicle (AUV)*, Master's Thesis, Naval Postgraduate School, Monterey California, September 1996. Available at *http://www.cs.nps.navy.mil/research/auv/auv_thesisarchive.html*

Byrnes, R., *The Rational Behavior Model: A Multi-Paradigm, Tri-Level Software Architecture for the Control of Autonomous Vehicles*, Ph.D. Dissertation, Naval Postgraduate School, Monterey California, March 1993.

Byrnes, R., Healey, A., McGhee, M., Nelson, M., Kwak, S. and Brutzman, D., "The Rational Behavior Software Architecture for Intelligent Ships," *Naval Engineers Journal*, vol. 108 no. 2, pp. 43-55, March 1996.

Campbell, M., *Real-Time Sonar Classification for Autonomous Underwater Vehicles*, Master's Thesis, Naval Postgraduate School, Monterey California, March 1996.

Cassandras, C., *Discrete Event Systems: Modeling and Performance Analysis*, Aksen Associates Incorporated Publishers, Homewood Illinois, 1993.

Chapman, D., "Planning for Conjunctive Goals," *Readings in Planning*, Morgan Kaufman Publishers, San Mateo California, 1990, pp. 537-558.

Chapuis, D., Deltheil, C. and Leandre, D., "Determination and Influence of the Main Parameters for the Launch and Recovery of an Unmanned Underwater Vehicle From a Submarine," *Proceedings of the 1996 Symposium on Autonomous Underwater Vehicle Technology*, pp. 276-282, Monterey California, June 1996.

Cheney, R., *Conduct of the Persian Gulf War, Final Report to Congress*, U.S. Department of Defense, Washington DC, April 1992.

Craig, J., *Introduction to Robotics: Mechanics and Control*, second edition, Addison-Wesley Publishing Company, Reading Massachusetts, 1989.

Davis, D., *CS4910 Class Project Computer Simulation of Control and Navigation of an Autonomous Underwater Vehicle*, Naval Postgraduate School, Monterey California, December 1995.

Dean, T., Firby, R. and Miller D., "Hierarchical Planning Involving Deadlines, Travel time and Resources," *Readings in Planning*, Morgan Kaufman Publishers, San Mateo California, 1990, pp. 369-386.

Ditang, W., Shouquan, K., Yulin, G., Yang, L. and Dalu, L., "A Launch and Recovery System for an Autonomous Underwater Vehicle Explorer," *Proceedings of the 1992 Symposium on Autonomous Underwater Vehicle Technology*, pp. 279-281, Washington D.C., June 1992.

Fishwick, P. A., *Simulation Model Design and Execution, Building Digital Worlds*, Prentice Hall, Englewood Cliffs New Jersey, 1995.

Florida Atlantic University Department of Ocean Engineering Advanced Marine Systems Laboratory World Wide Web Home Page, July 1996. Available at *http://www.oe.fau.edu/AMS/auv.html*, July 1996.

Gosling, J. and McGilton, H., *The Java Language Environment, a White Paper*, Sun Microsystems, San Jose California, May 1996. Available at *http://java.sun.com/doc/language_environment/*

Gwin, R. C. and Smith, J. T., "A Distributed Launch and Recovery System for an AUV and a Manned Submersible," *Proceedings of the 1992 Symposium on Autonomous Underwater Vehicle Technology*, pp. 267-278, Washington D.C., June 1992.

Healey, A. J., Marco, D. B., McGhee, R. B., Brutzman, D. P., Cristi, R., Papoulias, F. A. and Kwak, S. H., "Tactical/Execution Level Coordination for Hover Control of the NPS AUV II using Onboard Sonar Servoing," *Proceedings of the 1994 Symposium on Autonomous Underwater Vehicle Technology*, pp. 129-138, Cambridge Massachusetts, July 1994.

Healey, A. J., Marco, D. B., Oliviera, P., Pascoal, A., Silva, V. and Silvestre, C., "Strategic Level Mission Control--An Evaluation of CORAL and PROLOG Implementations for Mission Control Specifications," *Proceedings of the 1996 Symposium on Autonomous Underwater Vehicle Technology*, pp. 125-132, Monterey California, June 1996.

Holden, M. J., *Ada Implementation of Concurrent Execution of Multiple Tasks in the Strategic and Tactical Levels of the Rational Behavior Model for the NPS Phoenix Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey California, September 1995. Available at *http://www.cs.nps.navy.mil/research/auv/auv_thesisarchive.html*

Hopcroft, J. and Ullman, J., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Company, Reading Massachusetts, 1979, pp. 13-65.

Kanayama Y., *Introduction to Theoretical Robotics*, CS4313 Class Notes, Naval Postgraduate School, Monterey California, April 1996.

Leonhardt, B., *Mission Planning and Mission Control Software for the Phoenix Autonomous Underwater Vehicle (AUV): Implementation and Experimantal Study*, Master's Thesis, Naval Postgraduate School, Monterey California, March 1996. Available at *http://www.cs.nps.navy.mil/research/auv/auv_thesisarchive.html*

Marco, D. B. and Healey, A. J., "Local Area Navigation Using Sonar Feature Extraction and Model Based Predictive Control," *Proceedings of the 1996 Symposium on Autonomous Underwater Vehicle Technology*, pp. 67-77, Monterey California, June 1996.

Marco, D., Healey, A. and McGhee, R., "Autonomous Underwater Vehicles: Hybrid Control of Mission and Motion," *Journal of Autonomous Robots*, vol. 3, pp. 169-186, April 1996.

Massachusetts Institute of Technology Sea Grant College Program World Wide Web Home Page, July 1996. Available at *http://web.mit.edu/org/s/seagrant/www/mitsg.htm*

McClarin, D., *Discrete Asynchronous Kalman Filtering of Navigation Data for the Phoenix Autonomous Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey California, March 1996.

McGhee, R. B., *A Simplified Dynamic Model for Horizontal Plane Maneuvering by the NPS Model 2 AUV*, CS4314 Course Notes, Naval Postgraduate School, March 1991.

McGhee, R. B., Clynch, J. R., Healey, A. J., Kwak, S. H., Brutzman, D. P., Yun, X. P., Norton, N. A., Whalen R. H., Bachmann, E. R., Gay, D. L. and Schubert, W. R., "An Experimental Study of an Integrated GPS/INS System for Shallow-Water AUV Navigation (SANS)," *Proceedings of the 9th International Symposium on Unmanned Untethered Submersible Technology*, September 1995.

McKusick, M., Bostic, K., Karels, M. and Quarterman, J., *The Design and Implementation of the 4.4 BSD Operating System*, Addison-Wesley Publishing Company, Reading Massachusetts, 1996, pp. 137-146.

Murphy, R., "Coordination and Control of Sensing for Mobility Using Action-Oriented Perception," *AI-Based Mobile Robots*, MIT/AAAI Press, Cambridge Massachusetts, 1996.

Oliveira, P., Pascoal, A., Silva, V. and Silvestre, C., "Design, Development, and Testing of a Mission Control System for the MARIUS AUV," *Proceedings of the 3rd Workshop on Mobile Robots for Subsea Environments*, Toulon France, March 1996.

Peterson, J., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs New Jersey, 1981.

Proceedings of the 1996 Symposium on Autonomous Underwater Vehicle Technology, Monterey California, June 1996.

Proceedings of the 9th International Symposium on Unmanned Untethered Submersible Technology, Durham New Hampshire, September 1995.

Rae, G. J. S., "A Fuzzy Rule Based Docking Procedure for Autonomous Underwater Vehicles," *Proceedings of the Oceans 92 Conference*, pp. 539-546, Newport RI, October 1992.

Rae, G. J. S., Smith, S. M., Anderson, D. T. and Shein, A. M., "A Fuzzy Logic Docking Procedure for Two Moving Underwater Vehicles," *Proceedings of the 1993 American Control Conference*, pp. 580-584, San Francisco CA, June 1993.

Rowe, N., *Artificial Intelligence Through Prolog*, pp. 110-112, 263-281, Prentice Hall, Englewood Cliffs New Jersey, 1988.

Scrivener, A., *Acoustic Underwater Navigation of the Phoenix Autonomous Underwater Vehicle Using the Divetracker System*, Master's Thesis, Naval Postgraduate School, Monterey California, March 1996.

Smith, S. M. and Dunn, S. E., "The Ocean Voyager II: An AUV Designed for Coastal Oceanography," *Proceedings of the 1994 Symmposium on Autonomus Underwater Vehicle Technology*, pp. 139-147, Cambridge Massachusetts, July 1994.

Stefik, M., "Planning with Constraints (MOLGEN: Part 1)," *Readings in Planning*, Morgan Kaufman Publishers, San Mateo California, 1990, pp. 171-185.

Stevens, Richard W., *Advanced Programming in the Unix Environment*, Addison Wesley Publishing Company, Menlo Park California, 1992.

Tate, A., Hendler, J. and Drummond, M., "A Review of AI Planning Techniques," *Readings in Planning*, Morgan Kaufman Publishers, San Mateo California, 1990, pp. 26-49.

Tate, A., "Generating Project Networks," *Readings in Planning*, Morgan Kaufman Publishers, San Mateo California, 1990, pp. 291-296.

Tritech International Ltd., *ST-1000 Sonar: User's Manual*, Mike E. Chapman Company, Duvall Washington, 1992.

Tritech Internation Ltd., *ST-725 Sonar: User's Manual*, Mike E. Chapman Company, Duvall Washington, 1992.

Urick, R. J., *Principles of Underwater Sound*, 3rd edition, McGraw Hill Book Company, New York New York, 1983, pp. 406-416.

The VRML Repository World Wide Web Home Page, San Diego Supercomputer Center, August 1996. Available at *http://sdsc.edu/vrml/*

Vere, S., "Planning in Time: windows and Durations for Activities and Goals," *Readings in Planning*, Morgan Kaufman Publishers, San Mateo California, 1990, pp. 297-318.

Wernecke, J., *The Inventor Mentor, Programming Object-Oriented 3D Graphics with Open Inventor*^{TM}, Addison-Wesley Publishing Company, Reading Massachusetts, 1994.

White, K. A., Smith, S. M., Ganesan, K., Kronen, D., Rae, J. S. and Lagenbach, R. M., "Performance Results of a Fuzzy Behavioral Altitude Flight Controller and Rendezvous and Docking of an Autonomous Underwater Vehicle with Fuzzy Control," *Proceedings of the 1996 Symposium on Autonomous Underwater Vehicle Technology*, pp. 117-124, Monterey California, June 1996.

Wielemaker J. and Anjerwierden, A., *XPCE Reference Manual*, University of Amsterdam, Amsterdam Netherlands, 1994.

Wind River Systems, *VxWorks Programmer's Guide 5.2*, Wind River Systems, San Jose California, March 1995.

Winston, P., *Artificial Intelligence*, third edition, Addison-Wesley Publishing Company, Reading Massachusetts, 1992, pp. 50-53.