1998-03

Real-time modeling of cross-body flow for torpedo tube recovery of the Phoenix Autonomous Underwater Vehicle (AUV)

Byrne, Kevin Michael

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

REAL-TIME MODELING OF CROSS -BODY FLOW FOR TORPEDO TUBE RECOVERY OF THE *PHOENIX* AUTONOMOUS UNDERWATER VEHICLE (AUV)

by

Kevin Michael Byrne

March 1998

Thesis Advisor: Don Brutzman
Second Reader: Robert B. McGhee

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1998	Master's Thesis

4. TITLE AND SUBTITLE

REAL-TIME MODELING OF CROSS-BODY FLOW FOR TORPEDO TUBE RECOVERY OF THE PHOENIX AUTONOMOUS UNDERWATER VEHICLE (AUV)

5. FUNDING NUMBERS

6. AUTHOR(S)

Kevin Michael Byrne

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

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

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT

A virtual world provides an exceptional resource for the testing and development of an Autonomous Underwater Vehicle (AUV). The difficulties associated with the underwater environment are numerous and complex. In order to properly verify vehicle results in the laboratory such a world must accurately model the physics associated with the vehicle, its submerged hydrodynamics characteristics, and interactions with the environment. Environmental effects such as wave motion, currents, and flow forces created by bodies moving through the water can cause unpredicted performance variations and failures in the ocean environment. The current *Phoenix* AUV virtual world includes steady-state ocean currents, but does not take into account the environmental effects of waves and flow forces induced by adjacent vehicles (such as a moving submarine docking target).

This work provides a thorough real-time simulation of these complex factors using physically based models. The problem is broken down into wave motion effects, submarine-induced flow fields, and virtual sensors to improve AUV motion control. Simulated testing is performed across a range of easy to worst-case scenarios in order to justify assumptions. Extensive testing using virtual sensors is used to develop adequate control algorithms in the presence of turbulent cross-body flow.

The result of this research is an enhanced virtual world which more accurately depicts the ocean environment, along with the models and control algorithms required to design and operate an AUV (continued next page)

14. SUBJECT TERMS

Virtual environment, simulation-based design, cross-body flow, autonomous underwater vehicle (AUV), platform-independent simulation.

15. NUMBER OF PAGES

228

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard From 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-102

13. ABSTRACT (Continued)
during submarine launch and recovery. A platform independent approach to virtual environment simulation is presented through the use of the Virtual Reality Modeling Language (VRML) and Java. Finally, simulation test results provide strong evidence that AUV control with actual cross-body flow sensors can enable stable navigation, first through a turbulent flow field and then for subsequent docking with a moving submarine.

Approved for public release; distribution is unlimited

REAL-TIME MODELING OF CROSS-BODY FLOW FOR TORPEDO TUBE RECOVERY OF THE *PHOENIX* AUTONOMOUS UNDERWATER VEHICLE (AUV)

Kevin Michael Byrne
Lieutenant, United States Navy
B.S., State University of New York Maritime College, 1991

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1998

ABSTRACT

A virtual world provides an exceptional resource for the testing and development of an Autonomous Underwater Vehicle (AUV). The difficulties associated with the underwater environment are numerous and complex. In order to properly verify vehicle results in the laboratory such a world must accurately model the physics associated with the vehicle, its submerged hydrodynamics characteristics, and interactions with the environment. Environmental effects such as wave motion, currents, and flow forces created by bodies moving through the water can cause unpredicted performance variations and failures in the ocean environment. The current *Phoenix* AUV virtual world includes steady-state ocean currents, but does not take into account the environmental effects of waves and flow forces induced by adjacent vehicles (such as a moving submarine docking target).

This work provides a thorough real-time simulation of these complex factors using physically based models. The problem is broken down into wave motion effects, submarine-induced flow fields, and virtual sensors to improve AUV motion control. Each set of forces are thoroughly analyzed and realistically simulated in real-time through the algorithms developed. In order to maintain real-time response, perturbations in the flow field caused by the AUV itself are assumed to be negligible. Simulated testing is performed across a range of easy to worst-case scenarios in order to justify assumptions. Extensive testing using virtual sensors is used to develop adequate control algorithms in the presence of turbulent cross-body flow.

The result of this research is an enhanced virtual world which more accurately depicts the ocean environment, along with the models and control algorithms required to design and operate an AUV during submarine launch and recovery. A platform independent approach to virtual environment simulation is presented through the use of the Virtual Reality Modeling Language (VRML) and Java. Finally, simulation test results provide strong evidence that AUV control with actual cross-body flow sensors can enable stable navigation, first through a turbulent flow field and then for subsequent docking with a moving submarine.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS, ACRONYMS, AND SYMBOLS

ADV	Acoustic Doppler Velocimeter
API	Application Programming Interface
AUV	Autonomous Underwater Vehicle
CBF	Cross-body Flow
CFD	Computational Fluid Dynamics
DIS	Distributed Interactive Simulation
EOM	Equations of Motion
ESPDU	Entity State Protocol Data Unit
GPS	Global Positioning System
NPS	Naval Postgraduate School
OOD	Officer Of the Deck
PC	Personal Computer
PDU	Protocol Data Unit
PLD	Programmable Logic Device
P-M	Pierson-Moskowitz [wave spectrum]
RBM	Rational Behavior Model
SBD	Simulation-Based Design
VRML	Virtual Reality Modeling Language
kt	Knot(s)
yd	Yard(s)
A	Amplitude
$S(\omega)$	Spectral Density Function
g	Acceleration of Gravity
Hs	Significant Wave Height
lbs	Pounds
KHz	Kilohertz
MHz	Megahertz

t	Time
ω	Frequency

ACKNOWLEDGMENTS

First of all, I would like to acknowledge the unfailing love, devotion, and unconditional support which I have received throughout this experience from my wife, Jennifer. She has given up so much and expected little in return.

Secondly, I wish to extend my deepest gratitude to Dr. Don Brutzman whose support, guidance, knowledge, and enthusiasm have been a constant inspiration to me. His patience and positive attitude were invaluable to this research. I would also like to thank my second reader, Dr. Robert B. McGhee, for his support and careful review of this work. Thirdly, I am very grateful for the support I received from the AUV Research Group throughout the past year. Dr. Anthony Healy, Dr. Dave Marco, LCDR Jeff Reidel and CDR Mike Holden all provided advice and guidance during this entire project.

Finally, I would like to thank Caroline DelTheil and Didier Leandri for bringing this project to light.

I. INTRODUCTION

A. BACKGROUND

The end of the cold war has shifted the international balance of military strength. Today's United States Navy is undergoing a major reorganization. Our naval mission has moved from an open-water strategy to littoral warfare. A paradigm shift of such proportions brings with it the need for new strategies, technologies, and insights. Meanwhile public pressure demands reduced military funding and resources. These factors are creating challenging situations as a smaller military attempts to meet broader range of missions with fewer resources. Highly capable, low-cost underwater robots provide promising new capabilities which might be used to enhance military readiness while relieving the stress associated with broader mission goals.

While robots are not a valid solution for every problem domain, mine warfare is a mission area that is extremely pertinent. Mine warfare is a naval tactic that can be easily used by any potential enemy. It is a low-cost, low-risk measure which is very effective and hard to oppose.

The Naval Postgraduate School (NPS) Autonomous Underwater Vehicle (AUV) Research Group is actively working to provide a solution to this defense problem. The *Phoenix* AUV is a low-cost robot designed for mine detection. One of the research group's goal is to demonstrate that autonomous underwater robots are a solution which can provide underway units the ability to search areas for mines and obstacles from a safe distance. Figure 1.1 shows the Phoenix AUV deployed during in-water testing.

In addition to in-water robot testing, the NPS has a fully operational virtual environment which is used for simulation-based design (SBD). This provides a low-cost development environment for many possible robot technologies, reducing both project cost and time to deliver operational devices. The virtual environment gives researchers the ability to thoroughly test future devices in diverse operating conditions.

Figure 1.1. The *Phoenix* AUV deployed for in water testing (Brutzman, 1998).

B. MOTIVATION

Virtual environments provide a realistic arena for the testing and development of future vehicle technologies. It is necessary to ensure that simulations are physically based and accurate in order to support proper testing and development. This type of simulation-based design (SBD) can be used to develop the tools that the military will need to transition to the next century. This thesis presents solutions to previously unsolved underwater robot challenges, new capabilities in mine warfare and advancements in SBD techniques.

1. Mine Warfare

The NPS AUV research group has been striving with great success to provide a low-cost robot solution to mine detection and classification since 1986. The next logical step is to provide a sound method for the forward deployment and retrieval of AUV technology. Demonstrating that an AUV can be released and recovered underway "closes the loop" for AUVs by providing fully deployable technological solutions.

Submerged vehicle launch has a relatively easy solution. Submarines have been launching various objects through torpedo tubes since World War I. While some modifications to existing tube hardware may be required, a clear path to the launching solution exists. The most difficult problem for submarine deployment is recovery. Recovery is essential for mission data analysis and AUV re-use. This work provides an important missing link: autonomous vehicle control through turbulent water flow while docking. This new capability provides submarines the potential to effectively engage in counter mine warfare through deployment of recoverable AUVs.

2. Platform Independence

A second motivation for this research is commonly referred to (in the computer science domain) as platform independence. In the context of the problem at hand it is taken to mean providing the ability to simulate complex virtual environments on whatever computer resources are available, regardless of make, architecture, or operating system. As computational power has increased in the recent past, complex simulations are no longer limited to users with high-end graphics workstations. Personal computers have the capacity to manage applications that were previously unavailable.

The vision this work has pursued is one in which anyone anywhere with network connectivity can view and actively participate in complex simulation exercises. We are attempting to build a closer link between those involved in design and testing and the end user of technology. While the platform-independence issue is not directly related to solving torpedo tube docking of an AUV, its importance cannot be underestimated. It drives home the point that simulation for spatial awareness can be used anywhere.

C. OBJECTIVES

The objective of this research is to design a method of simulating AUV control in a true ocean environment in order to accurately test and develop algorithms for moving torpedo tube recovery. To achieve this goal there are several sub-problems to address:

- Wave motion must be accurately simulated for numerous sea states. This is significant due to the unpredictability of the ocean environment. For a prototype AUV to be fully tested, all possible sea conditions must be available.

- Foreign-body-induced flow forces must be depicted as realistically as currently

possible. These types of forces pose the most difficult problems to the development of control algorithms for several reasons. First, they are extremely dynamic. The state of flow forces are continuously changing and influenced by many independent factors, many of which are not yet completely understood. Second, they occur in the regions where robot control is most crucial, i.e. the areas where a control failure could cause devastating damage to both the robot and recovery vehicle.

- Extensibility needs to be considered when modeling flow fields. While a significant amount is known about the nature and behavior of complex fluid flow, there is still much to be discovered. By creating a methodology which allows for the upgrade of simulation flow field data as the science of fluid dynamics advances, the accuracy and lifetime of the virtual world is enhanced.

- Refinement of the equations of motion is also necessary. The ability to model a vehicles behavior based on its size and shape as forces act upon it becomes another concern when trying to ensure the behavioral accuracy of such an environment. In previous versions of the Phoenix AUV's Virtual World the equations of motion were based solely on a cylindrical body shape.

- The source code for the Phoenix AUV Virtual World is distributed openly. Unfortunately, due to the computational complexity of such a model its use has previously been feasible for only those users with high-end Silicon Graphics workstations. With the capabilities of personal computers rapidly increasing and the introduction of platform-independent languages (such as Virtual Reality Modeling Language (VRML) and Java) it has become possible to move this simulation into the platform-independent domain. To this end a platform-independent implementation is also provided.

D. THESIS OUTLINE

This chapter describes the background, motivation, and objectives of creating a virtual world which accurately models the ocean environment. Chapter II discusses the background of the Autonomous Underwater Vehicle (AUV) at the Naval Postgraduate School (NPS) including its purpose, history and related research projects. Chapter III evaluates the goals of this work providing a clear and concise problem statement. Chapter IV provides in-depth discussion of design considerations and hydrodynamics modeling related to the simulation. Chapter V addresses changes required to the AUV execution level, both hardware and software, needed to equip the vehicle so it can successfully operate in such an environment. Chapter VI and Chapter VII describe the simulation

results found in this research, including run-time performance, simulation limitations and robot control/sensor upgrades. Chapter VIII presents conclusions, research contributions and recommendations for future work.

II. RELATED WORK

A. INTRODUCTION

The technology involved in virtual environment development and robot simulation encompasses many disciplines. While these two tasks seem to be well suited for one another, the fields of study that produce the theories employed by each are vastly different. This chapter looks at the important areas evaluated for use in this research. The topics range from cutting-edge computer graphics to mechanical engineering practices that have been around for many decades. Nevertheless, all of these techniques are needed to create the solution to this difficult problem.

Specific related-work topics examined in this chapter include *Phoenix* AUV hardware and software, underwater virtual world modeling, distributed control, the Virtual Reality Modeling Language (VRML), the Java programming language, the Distributed Interactive Simulation (DIS) protocol, the DIS-Java-VRML project, and Computational Fluid Dynamics (CFD).

B. *PHOENIX* AUV

An Autonomous Underwater Vehicle (AUV) is a self-contained underwater robot typically combining a multitude of sensors and controllers. The *Phoenix* AUV is an incarnation of this type of robot developed to demonstrate the abilities of a low-cost autonomous platform. The *Phoenix* architecture can be broken down into two major categories: hardware and software.

1. Hardware Architecture

The Naval Postgraduate School's (NPS) *Phoenix* AUV is a complex robot, comprised of a single water-tight compartment which contains various motors, controllers, servo-amplifiers, and computers. The internal component layout is shown in Figure 2.1 . Figure 2.2 shows the an external view of the hardware layout.

The main processing power inside the *Phoenix* comes from two computers. A Gespac M68030 is used to run the execution level software while a Sun Voyager Sparc 5 Workstation runs the tactical and strategic level software (Brutzman, 1998). The specifics of each software level will be discussed in the following section. The Gespac computer runs the OS-9 operating system allowing for use of

real-time multitasking functions when controlling the vehicle devices (Byrnes, 1993). The Sun Voyager 5 runs SunOS 5.4. These computers are networked together via an Ethernet inside the vehicle. This allows for the machines to easily communicate. It also has advantages in terms of remote monitoring. The Ethernet optionally provides Internet connectivity to the boat through a tether. The tether can be used to monitor each process, collect data, or to intervene when an operational fault occurs.

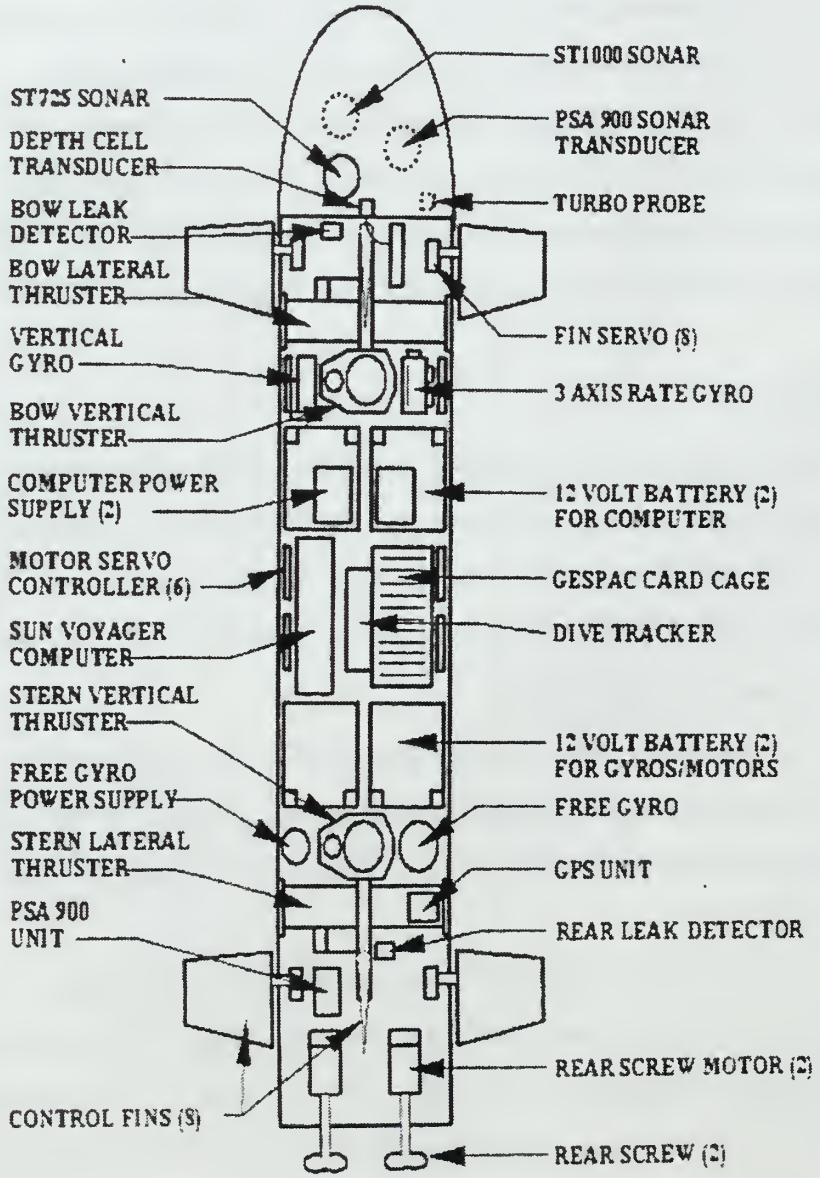


Figure 2.1. Internal view of the Phoenix AUVs component layout (Marco, 1996).

Other interesting pieces of gear include two high-resolution sonar units, a Global Positioning System, and an inertial navigation package. The sonar units provide excellent detect and classification abilities. They have 1 cm resolution out to a maximum range of 30 meters. Additionally, the ST725 (725 KHz) has a 1° wide by 24° vertical beam, and the ST1000 (1 MHz) a conical beam of 1° (Brutzman et al, 1998). All of these devices are used to provide a fully autonomous robot with significant operational and navigational capabilities.

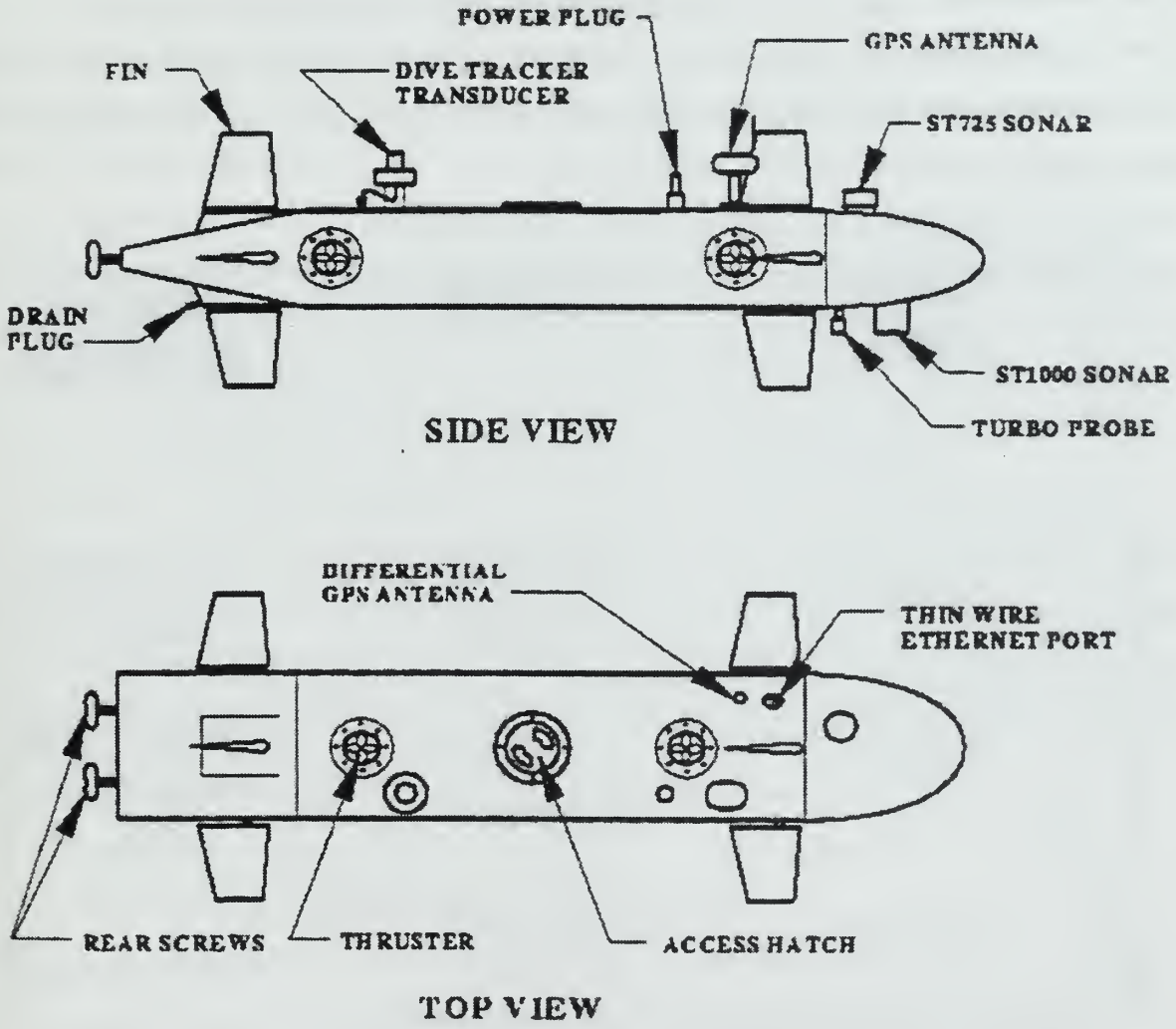


Figure 2.2. External view of the Phoenix AUVs component layout (Marco, 1996).

2. Software Architecture

The software architecture of the Phoenix AUV is a tri-level design called the Rational Behavior Model (RBM). The RBM architecture consists of three separate software layers, each layer having its own functional requirements, implementation restrictions, and component interfaces (Byrnes, 1993)(Byrnes, 1996)(Marco, Healey, McGhee, 1996). RBM divides robot control into functional blocks which mimic those of a submarine operational structure. Thus the use of RBM in the NPS vehicle is well suited to the thinking patterns of students involved in the project.

The RBM divides responsibilities into areas of open-ended strategic planning, soft real-time tactical concern, and hard real-time execution level tasks. Figure 2.3 shows the relationship between strategic, tactical and execution levels in the RBM.

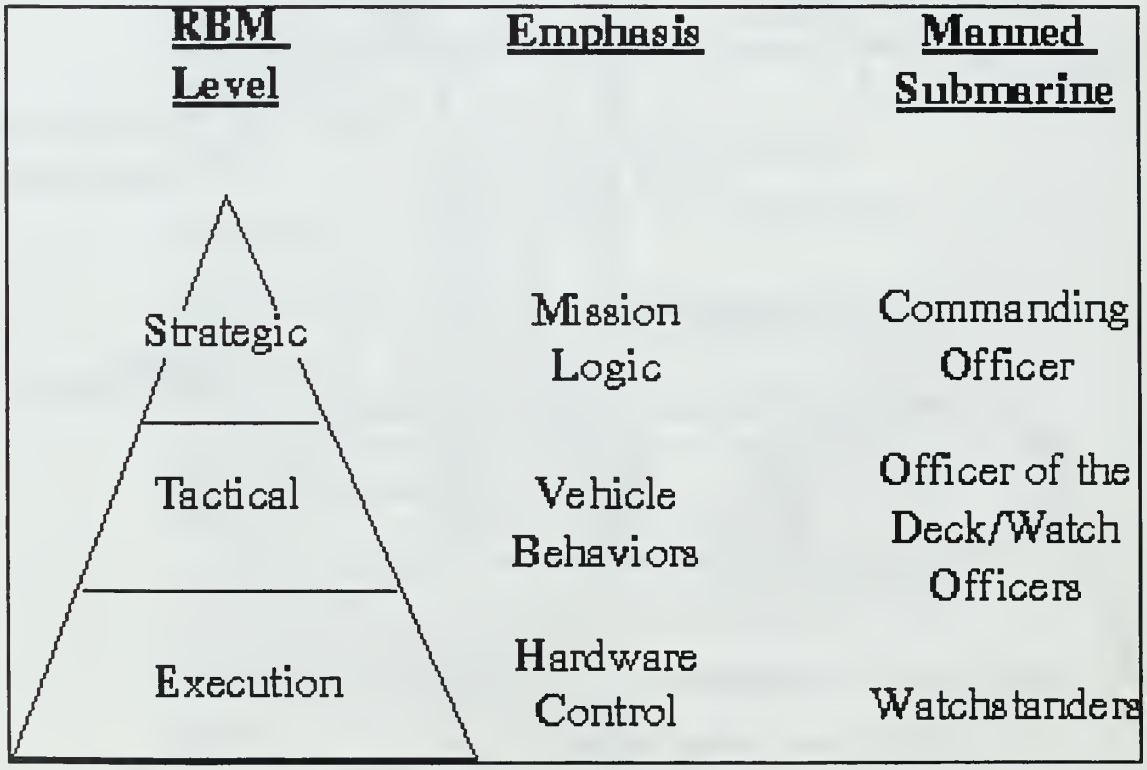


Figure 2.3. The Rational Behavior Model Architecture Pyramid (Holden, 1995)

The execution level provides the interface between software and hardware. It is designed to meet all of the systems hard real-time requirements. The execution level is responsible for the

underlying stability of the vehicle, the control of individual devices, and providing data to the tactical level (Byrnes, 1993)(Byrnes, 1996). In terms of an underway watch team organization, the execution level performs the tasks normally assigned to individual watchstanders.

The level of command above the watchstanders in the underway watch team is the Officer Of the Deck (OOD). The OOD's responsibilities are concerned with the tactical picture: what individual tasks need to be completed to reach a goal. In the RBM this functionality is contained in the tactical level. The tactical level does not operate on hard real-time deadlines, rather it operates in terms of discrete events (Byrnes, 1993)(Byrnes, 1996). It provides a software level that interfaces with both the execution and the strategic levels, thus giving strategic level indication of vehicle state and completed tasks, and execution level commands.

The highest level of the RBM is the strategic level. This portion corresponds to the role of a commanding officer. It is not concerned with the specifics of task completion. Instead the issues that the strategic level monitors are the completion of mission goals. Inside the strategic level resides the mission specification. Through symbolic computing it uses a set of rules coupled with an inference engine to direct (and respond to) the tactical level (Byrnes, 1993)((Byrnes, 1996).

The RBM is a complex architecture, but it greatly simplifies AUV design and operation through appropriate levels of abstraction. By setting clear boundaries between areas of responsibility, RBM enables robot control to be defined by separate applications with predefined interfaces. RBM also allows naval students, who are intimately familiar with an at-sea watch structure, to apply real-world experience to complicated control problems. Using an architecture designed for both robot and human requirements has been a crucial advantage.

C. *PHOENIX* AUV VIRTUAL ENVIRONMENT

Development of an AUV poses a number of unique problems. Chief among those problems is the fact that during the actual in-water testing of robot hardware and software, it is often impossible to observe or communicate with the vehicle. Analysis is typically limited to post-exercise data review alone. This situation confronts designers with a difficult development process. Physical remoteness and inability to observe effectively takes away one of the human mind's greatest strengths: the ability to visualize. To overcome this problem, a virtual world was developed which models salient

characteristics of the ocean environment from the robot's perspective (Brutzman, 1994). This effectively puts humans back into middle of the testing and development loop, and allows developers to visualize robot behavior under diverse conditions.

The virtual environment provides an area of underwater terrain in which testing and development can be observed. Figure 2.4 is a recent view of the underwater world showing all of the major objects that are contained, including *Phoenix* at the surface just right of center. The implementation of the *Phoenix* AUVs virtual world is broken into three major sections. One portion represents the physical side of the operating environment, a second is robot software (and optionally hardware), and a third provides an interactive3D graphics window into the virtual environment.

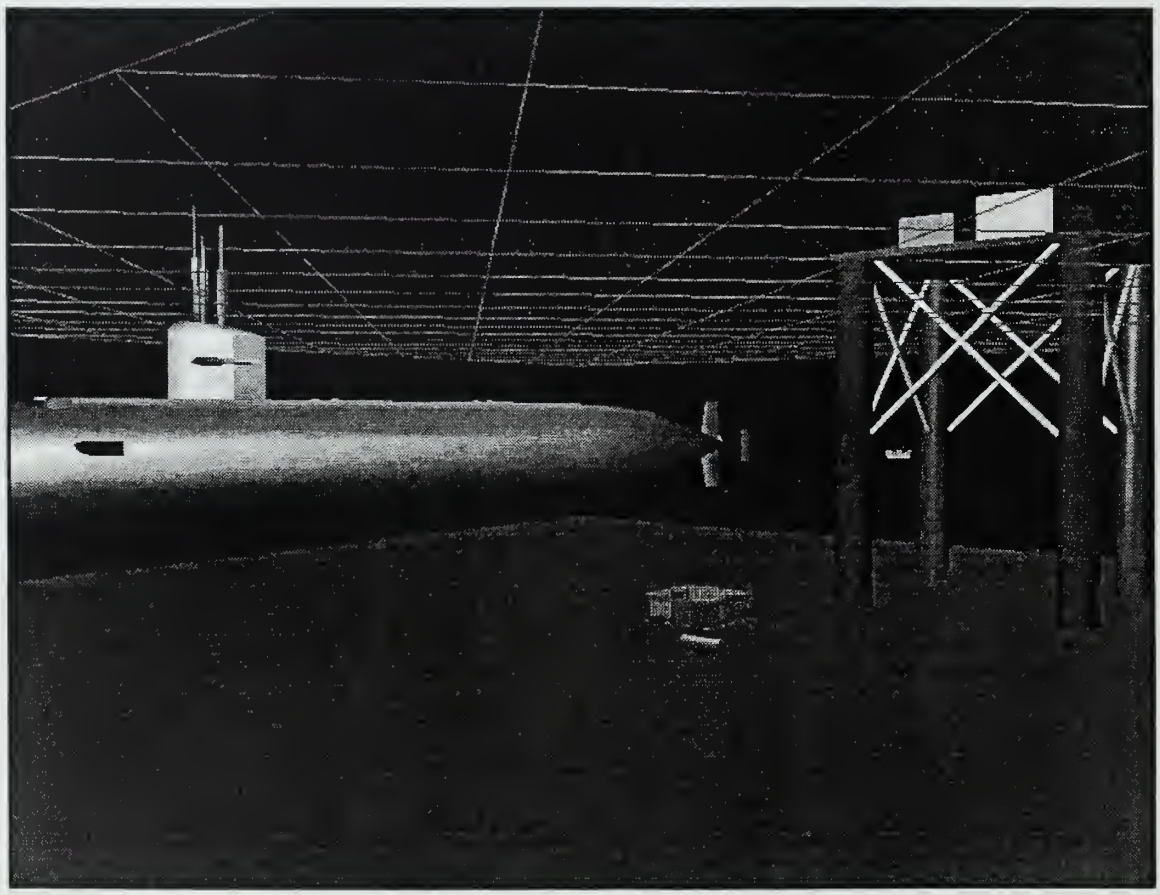


Figure 2.4 - The *Phoenix* AUV Virtual World.

The most complex of the modules which comprise the virtual environment is the one which models the vehicle hydrodynamics. Inside this program all aspects of vehicle motion are considered. Using a Newton-Euler approach to the derivation of the six degree-of-freedom equations of motion (Brutzman, 1994)(Healey, 1992), the program provides a very accurate rendition of the environment. The dynamics program takes input from the vehicle, describing the state of all of its devices, and calculates in complete detail the responses the vehicle is expected to receive from the environment. Other physical models includereal-time sonar detection, Global Positioning System (GPS), and acoustic navigation.

Another component of the virtual environment is the AUV software. This code performs the task of controlling all of the devices associated with the AUV, including propellers, thrusters, sonars, inertial navigation systems and any other hardware installed on *Phoenix*. This execution level control is coupled with the more sophisticated robot intelligence provided by the strategic and tactical levels. Communication is conducted through all levels of the RBM architecture to determine how to deploy each one of these devices. After determining the state of each sensor and effector, the execution level sends out the commands placing them in the appropriate state. Since in the virtual simulation the devices are typically not physically present, they are positioned via telemetry vector message interchange with the dynamics program. The effects of all the devices are determined by the dynamics program models, and then proper responses are sent back to the execution level software. This query-response interchange is incorporated into the sense and act phases of the execution level's sense-decide-act cycle. This design architecture enables the robot to run and respond to various stimuli in the same manner in the virtual world as in the real world, since the robot software in each case is identical (Burns, 1996).

The final portion of the virtual environment gives the user an interactive 3D graphics window into the environment. Referred to as the viewer, this program allows observation of all aspects of the simulation. Virtual representations are provided to indicate the robots employment of each sensor and effector. This visualization has repeatedly been shown to be essential to the development process. Figure 2.5 shows the animations associated with the robots use of thrusters and propellers.

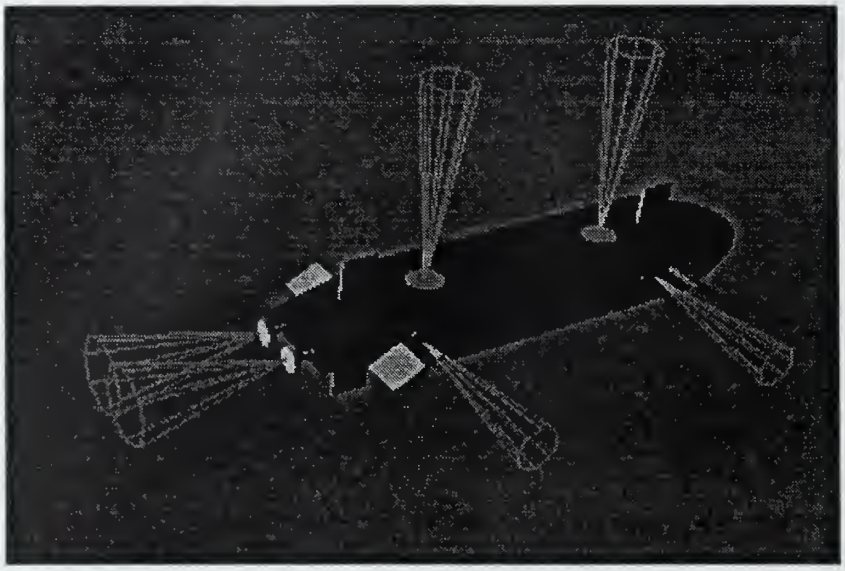


Figure 2.5. *Phoenix* AUV in the virtual environment demonstrating use of thrusters and main motors.

This type of visual representation also gives some intuition into how the AUV employs its sonar assets. Sonar visualization was used to implement and refine a control algorithm which enables the *Phoenix* AUV to detect and classify objects in the water (Davis, 1996). In the case of a detected tube like object the algorithm was further enhanced to allow the vehicle to safely begin entering the tube. Figure 2.6 is a screen capture of this type of mission being executed in the virtual environment.

All of these components are networked together to provide an integrated development environment. Multiple simultaneous viewers are enabled via use of the Distributed Interactive Simulation (DIS) protocol (IEEE, 1993)(IEEE, 1994a)(IEEE, 1994b). The position, orientation and state of the vehicle are multicast across the network via Entity State Protocol Data Units (ESPDU). The viewer application listens to the network for these packets, extracts the information from them, and incorporates it into the scene rendered in the virtual world. Decoupling graphics viewers from robot software and virtual world models provides a scalable approach that permits multiple researchers to evaluate robot mission progress.

The versatility of the virtual world was further demonstrated by its use for prototype modeling of an optical sensor, used for AUV guidance and control without sonar, while docking with a stationary torpedo tube (DelTheil, 1997).

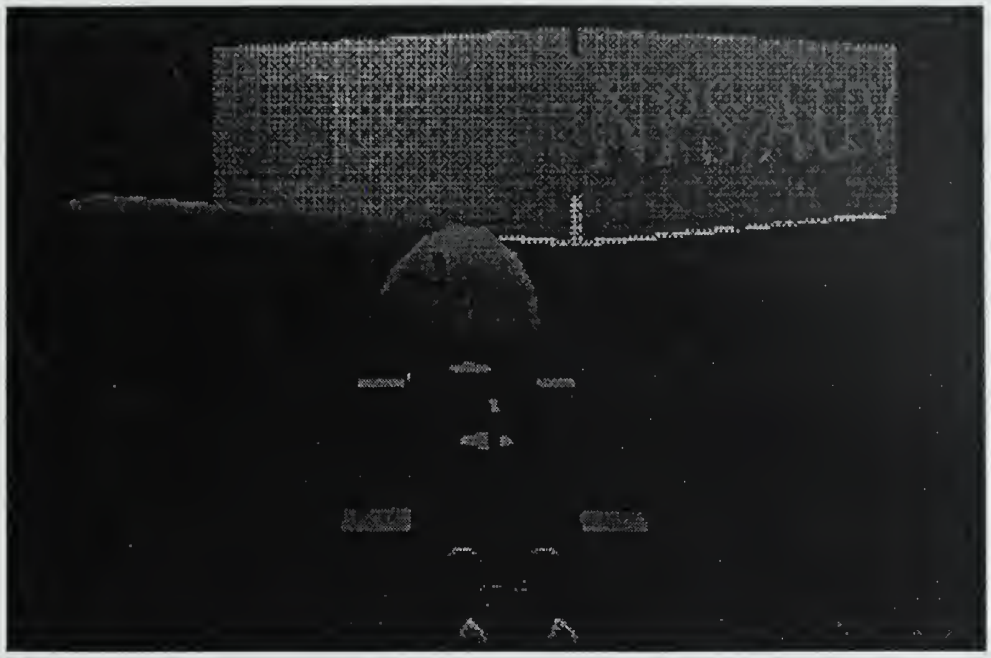


Figure 2.6. *Phoenix* AUV Using Sonar to detect and Classify a Torpedo Tube (Davis, 1996).

A virtual environment provides an outstanding arena for robot testing and development. It affords system engineers the opportunity to observe equipment operation in a safe controlled environment. This type of technological advance is a large step forward for the underwater robot community.

D. DISTRIBUTED ANALOG/DIGITAL CONTROL DEVELOPMENT

Computing power has been increasing at an amazing rate. The average lifetime for new technology in the computer industry is only nine months, with processing power increasing by an order of magnitude every two years. One area in which similar advances are being made is networking. These advances have come in transfer rate increases and reliability improvements. This allows system designers to leverage the network (and all the resources attached to it) as additional assets.

The network provides an unlimited number of additional resources to any computing system. Similar advances are also being made in control technology for data acquisition systems. Smart controllers present the opportunity to create autonomous device controls which can monitor device

operation, output necessary data readings, and react properly when provided operational commands from remote stations. One company leading the way in this field is Echelon. They have developed a series of programmable devices which all communicate over a network using a proprietary network protocol called LonTalk (Young, 1998).

Use of small specialized processors can give any system significant performance improvements. By offloading portions of specialized control code a central monitor application becomes more concise, allowing it to execute more quickly and efficiently. It also adds to system robustness. No longer will a single fault halt operation. Other devices not affected by a single control board failure will continue to operate normally, leaving the monitoring application to adjust for a single device failure.

Distributed control is the direction in which many data acquisition systems are moving. Systems as diverse as elevator control systems and high-voltage air conditioning systems all gain from distributed control. This technology can also give the *Phoenix* AUV execution level a welcome upgrade (Young, 1998).

E. VIRTUAL REALITY MODELING LANGUAGE (VRML)

As the Internet continues to expand and gain in popularity, many new technologies are being developed to utilize this medium. In the past, browsing web pages has been restricted to two dimensions. The Virtual Reality Modeling Language (VRML) brings three-space to the Web.

VRML is an interpreted language that allows developers to create content-rich three-dimensional (3D) worlds which can be viewed across the Internet inside a web browser. At its core VRML is a specification for describing 3D worlds through a text-based file format (Ames, 1997)(VRML, 1997).

```
#VRML V2.0 utf8

Group {
   Children [
      Viewpoint   {
           description    Initial view
           position          6 -1 0
           orientation    0 1 0 1.57
       }
       Shape {
           geometry Sphere {   radius 1 }
           appearance Appearance {
         texture ImageTexture {
          url    earth-topo.png
         }  }  }

        Transform   {
            translation   0 -2 1.25
            rotation     0 1  0      1.57
            Children [
                Shape {
                    geometry Text {
                   string [" Hello" "world!"]
}
                appearance Appearance {
              material Material {
          diffuseColor 0.1  0.5  1
                 }}}
            ]
   }
 ]
}
```

Figure 2.7. VRML source code hello_world.wrl taken from (Brutzman, 1998a).

While VRML is a powerful object description language, it is also a simple language to learn. The novice can quickly learn enough to develop his first program. Figure 2.7 is a programming listing for the basic "Hello World" program found in so many programming texts (Brutzman, 1998a). The results of this small scene description are displayed in Figure 2.8. This demonstrates just how simple VRML makes 3D authoring.

Figure 2.8. Output of *Hello_world.wrl*.

The power of VRML is its ability to create dynamic environments. It fully supports animation, user interaction, and advanced object behaviors through scripts (Hartman, 1997). After describing objects inside a world a developer has many options regarding how to use those objects. Animation can be performed through predetermined routes, execution of scripts, or dynamically using outside

applications to manipulate objects inside the world. The overall affect is the creation of portable virtual environments which are visually pleasing and truly interesting. The greatest promise of VRML with respect to this project is the possibility of 3D visualization of AUV missions using any web browser.

F. JAVA

Java is a fully functional programming language which was first released in 1995 by Sun Microsystems. It is a solidly engineered language that was created with many ideal design goals in mind. Table 2.1 gives the key features as described by the authors of Java (Cornell, 1997). The key features of interest related to the work described in this thesis are that it is architecture neutral, object oriented, and portable (Cornell, 1997).

Java is an architecture-neutral language. What this means is that when a Java program is compiled the compiler creates a neutral file format that contains byte codes of the compiled program. These byte codes can then be executed on many different processors with the Java run-time environment present. The run-time system interprets the byte codes and translates the information into native machine code for execution.

Java Design Goals	Functionality of Concern for this Thesis
Simple	Yes
Object-Oriented	Yes
Distributed	No
Robust	No
Secure	No
Architecture Neutral	Yes
Portable	Yes
Interpreted	No
High Performance	Yes
Multithreaded	No

Dynamic	No

Table 2.1. Design goals of the Java programming language, contrasted with thesis goals.

The object-oriented programming paradigm provides numerous useful characteristics. Java fully supports data hiding, encapsulation, inheritance and code reuse through this object-oriented approach. This type paradigm focuses on the data being manipulated by an application instead of how each step of the manipulation takes place. It gives the developer the ability to write code once and use it many times in many different applications.

Portability brings Java to the Web. There are no implementation-dependent aspects of the Java specification (Cornell, 1997). This means that the binary data is stored in a fixed format which eliminates the problems of running code on various platforms. Through this type of implementation and the use of standard libraries which define portable interfaces, Java byte codes can be retrieved across the Internet and run on local platforms, independent of the machine architecture.

As the world wide web continues to increase in popularity, Java is positioned to be the language of choice. Its well-designed class library provides all the functionality required to develop professional applications. These applications can be easily distributed via the Internet and run on any platform which has the Java run-time environment present.

G. DISTRIBUTED INTERACTIVE SIMULATION (DIS) PROTOCOL

The Distributed Interactive Simulation (DIS) protocol describes a standard of communications between entities in distributed simulations (IEEE 93, 94a, 94b). It is well suited for general usage in networked virtual environments due to the standardization of object interactions. This allows many users in remote locations to view or participate in a simulation as long as the standard object interface is followed.

Information is passed between entities through the use of protocol data units (PDUs). Figure 2.9 demonstrates the architecture of a distributed simulation using DIS. There are 27 different types of PDUs defined for use. Each one addresses a different possible interaction between entities. Types of PDUs range from the most common Entity State PDU, to the more rarely used Electromagnetic Emission PDU. The Entity State PDU is the primary PDU used, containing information about an

entity's position, posture, linear and angular velocities and accelerations. It is sent across the network by an object whenever one of the its entity state parameters changes by a threshold amount or a designated time period has expired. All other entities that are concerned with the position information of the sending entity will listen for the PDU and upon receipt will integrate that information into the rendered scene.

Networking provides a significant advantage when working in virtual environments. It allows objects which are being operated on remote workstations to be viewed locally. In a complex world objects can be offloaded to idle processors while they are rendered by the local machine. This type of network interaction is possible through the standards defined by the DIS protocol.

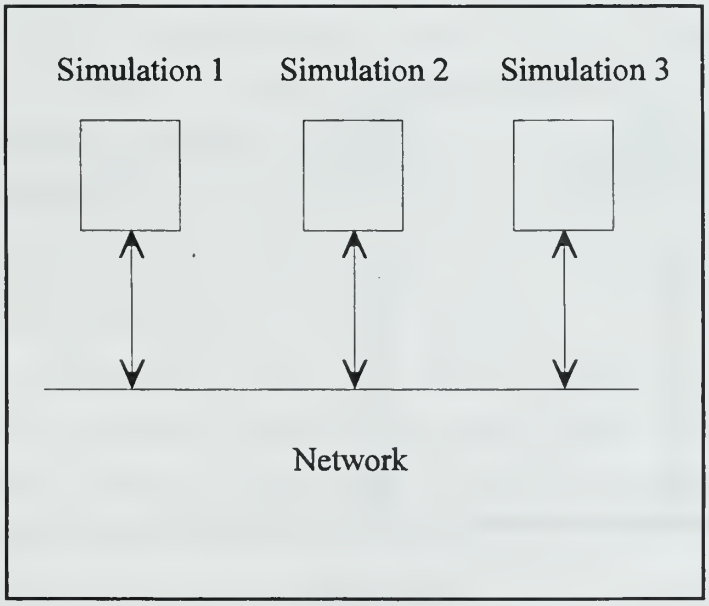


Figure 2.9. Network Connectivity of a DIS simulation.

H. DIS-JAVA-VRML

In an effort to bring large-scale distributed simulations to the personal computer domain a working group has been formed to integrate DIS with Java and VRML. A VRML Consortium (www.vrml.org) working group is a technical committee which tries to solve specific technical problems. The DIS-Java-VRML working group was chartered with numerous goals aimed at making these technologies work together.

Some specific objectives of the DIS-Java-VRML working group include completing a freely available Java implementation of the DIS protocol, producing a set of references and recommended practices for mapping between DIS and VRML worlds, create various DIS utilities in Java, and to also create some standard physics and math libraries to be used in these simulations. More information for regarding the working group can be found at [http://www.stl.nps.navy.mil/dis-java-vrml].

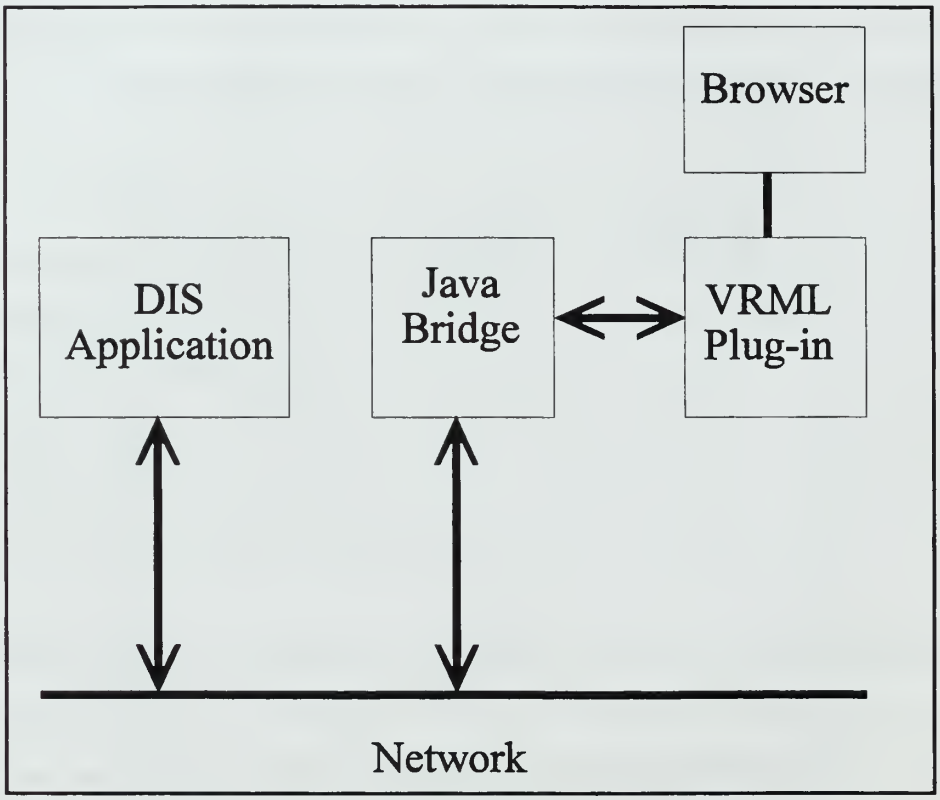


Figure 2.10. DIS-Java-VRML interaction layout.

The overall layout of a simulation using the DIS-Java-VRML library is quite unique. The library handles the interactions between the network, browser, and VRML plugin. Figure 2.10 outlines the interactions handled by the library.

Combining the capabilities of DIS, Java and VRML can quickly allow developers to create content-rich networked simulations that are available to anyone with access to the Internet. The

possibilities of simulation content and complexity are endless. With access via the Web simulations can be run anywhere in the world regardless of the locality of participants and simulation monitors. This integration truly adds new dimensions to virtual environment use. It continues to move the Web into three-space.

I. COMPUTATIONAL FLUID DYNAMICS (CFD)

Computational Fluid Dynamics (CFD) is a field of study concerned with the prediction of fluid motion about bodies of arbitrary shape. Supercomputers are typically used to solve numeric approximations that describe the fluid flow.

Looking more closely at the term CFD, this branch of science is considered computational because of the use of high-speed computing resources. The fluid flows are typically modeled and analyzed using large sets of Navier-Stokes partial differential equations. Solving these equations for specific fluid-flow cases is computationally intense because no closed-form solutions exist, except in trivially special cases (Scientific Computing Group at Indiana University, 1998). Thus any given problem may take days of computational cycles to solve. Solution of CFD problems are generally considered to be among the grand challenges of supercomputing.

CFD also refers to the analysis of fluids. Fluid refers to anything that isn't a solid, thus both air and water are considered. A more technical definition classifies fluids "as any substance which cannot remain at rest under a sliding, or shearing, stress (Scientific Computing Group at Indiana University, 1998)."

Finally, CFD dynamics refers to the study of objects in motion. In dynamics one is concerned with an objects motion and the forces associated with that motion. This is very different from kinematics, which is concerned with the relationships of motion quantities regardless of the forces induced by that motion (Healey, 1998). Kinematics models are typically less realistic than dynamics models. In general the resolution of kinematics models are demonstrably inadequate for AUV motion prediction.

Overall, CFD is a numerically intensive science. Solutions to CFD problems are extremely complex and often require the most advanced computer systems to solve. Thus research in this field is extremely important, providing the basis for understanding complex flow interactions. From there

simpler representations of flow interactions can be created which provide a general model for testing. The approach taken by this thesis completely avoids the field of supercomputing CFD, and instead seeks PC-based or workstation-based solutions which produce imperfect but adequate results in realtime for human operator and robot use.

J. SUMMARY

Many disciplines are needed to complete any complex project. The basis for the solution to the problem of torpedo tube docking of the *Phoenix* AUV epitomizes that notion. Simulation-based research and design draws from the newest available technologies. While the environmental forces which must be modeled in this type of simulation have been around since the beginning of time, only recently has technology been created to help man solve problems in many disciplines. A broad level of knowledge must be used to arrive at a correct solution when considering large and complex problems.

This chapter presented an overview of many topics related to the solutions presented in this thesis. The *Phoenix* AUV hardware, software and underwater virtual world modeling were discussed because they provide the basis for the testing presented. Other areas such as distributed control, the Virtual Reality Modeling Language (VRML), the Java programming language, the Distributed Interactive Simulation (DIS) protocol, the DIS-Java-VRML project and Computational Fluid Dynamics (CFD) also play a role important roles in solving the problem of torpedo tube recovery. While no one topic provides all answers, a combination provides a well-rounded solution.

III. PROBLEM STATEMENT

As the mission of the United States Navy continues to focus on littoral warfare, the importance of mine warfare becomes more apparent. Mine warfare has historically been one of the most difficult tasks performed by naval units. The majority of the tactical burden has often fallen on submarines since their operational areas often coincide with the areas of most value to enemy forces. One outstanding tool for mine detection is an Autonomous Underwater Vehicle (AUV). This type of robot can be used to scour forward areas using high-frequency sonars and global positioning systems (GPS) to detect and neutralize mines, easing the burden of mine detection on all forward-deployed units.

The majority of technical issues preventing this type of AUV deployment have been solved. The only problem remaining before this type of vehicle deployment can be executed is the vehicle recovery system. Vehicle recovery poses many difficult problems. The evolution itself is very dangerous for both the recovering submarine and the AUV. Even the smallest mistake can place the submarine at great risk. If the recovery does not run smoothly, damage can range from complete loss of the AUV to a breach of the water-tight integrity of the recovering submarine, thus threatening the safety of her crew. There is an intolerably small margin of error.

In an effort to conquer the unresolved issues associated with mine warfare, the *Phoenix* AUV has been created as a research and development platform. It is used to test the newest equipment on the market and to develop control algorithms which employs this equipment most effectively. Even in stand-alone development the risk of vehicle loss is very high. While robot code is written with safety of the vehicle in mind, robot testing is inherently dangerous. For that reason a virtual environment was created that is used to test software and hardware prior to in-water testing (Brutzman, 1994).

A virtual world provides an exceptional resource for the testing and development of AUV technology. The difficulties associated with the underwater environment are numerous and complex. In order to properly validate the results from such a world, one must accurately model the physics associated with the vehicle, its submerged hydrodynamics characteristics, and the environment. Environmental effects such wave motion, currents, and flow forces created by bodies moving through the water can cause significant variance in the testing environment. The current version of

the Phoenix AUV Virtual World includes steady-state ocean currents, but does not take into account the localized environmental effects of waves and body-induced flow forces.

In an effort to provide this type of realistic simulation environment, the effects of environmental factors have been completely integrated into the hydrodynamic simulation. This work provides a sound real-time simulation of these complex factors using physically based models. The problem is broken down into wave motion effects, body-induced flow fields, and AUV motion control. Each one is thoroughly analyzed and realistically simulated in real-time through the algorithms developed.

The result of this research is a Virtual World which accurately depicts the ocean environment. It can be used to test and develop the control algorithms required to operate an Autonomous Underwater Vehicle in any situation without risk. This environment thus provides a safe and physically accurate arena in which the problem of torpedo tube recovery can be carefully examined.

Another issue evaluated in this work is platform independence. As research and development money becomes scarce, the availability of high-end graphics workstations is also becoming rare. With the advent of platform independent languages such as Virtual Reality Modeling Language (VRML) and Java, the ability to run complex three-dimensional (3D) simulations on personal computers has arrived. In addition to providing a realistic virtual environment for development of new technology this work strives to make that simulation available for anyone to use. By using web-based technologies anyone can view and interact with the simulation and development process, further advancing the marriage between developer and end user.

The principal problem addressed by this thesis is that of torpedo tube recovery of the Phoenix AUV. It employs a physically based virtual environment to simulate the forces encountered during such an evolution. The goal is to provide an overall solution to the problems associated with torpedo tube recovery through simulation-based design (SBD).

IV. HYDRODYNAMICS MODELING

A. INTRODUCTION

This chapter presents the theory behind the implementation of cross-body flow in the *Phoenix* virtual environment. The problem of modeling cross-body flow is broken down into several components, each of which is analyzed in depth. An overview first outlines the components of the equations of motion algorithms with some detail on the individual parts. The high-resolution buoyancy model is described as the basis for modeling flow forces on the AUV. Wave-motion simulation is examined in detail, followed by body-induced flow simulation and square hull versus round hull adjustments to the equations of motion.

B. OVERVIEW

Virtual environments are a very useful tool in the research and development process. Their use can provide sound simulation-based designs. The ability to test and redesign during development allows for relatively easy correction of design flaws and can save valuable time and money in the process. Nevertheless any simulation is only as good as the physical model it is based on. A virtual ocean environment which fails to address the physical forces that are present in the real ocean provides little insight during development, perhaps guaranteeing the failure of the project.

The elements of nature must be completely integrated into any simulation environment if it is to be used as a true test platform. Additionally, the ocean environment has unique characteristics which make its simulation more complex. Factors such as buoyancy, wave motion, and body-induced flow forces are among the most computationally complex to model. They are all significant and cannot be overlooked when developing a true simulation environment. Figure 4.1 shows the overall flow of information within the hydrodynamics model. The separate sections indicate areas of code that handle specific calculations which are calculated during each time step. The overall hydrodynamics model is described in (Brutzman 94a, 94b, 98). The simple buoyancy model is described in (Bacon, 96). This thesis implements the shaded blocks in Figure 4.1. Throughout the following discussion of theoretical basis, simplifications were made to ensure true real-time performance of the simulation. Assumptions are made when the effect of their simplification do not

effect the accuracy of the model, while providing performance improvement. Table 4.1 provides a summary of assumptions which will be presented and justified later in the chapter. This chapter provides the theoretical basis for a high-resolution buoyancy model, physically based wave motion simulation, external submarine body-induced flow field simulation, more precise modeling of square hulls versus round hulls, and refinements to the equations of motion (EOM).

Topic	Assumptions
Wave motion simulation	1. Wave motion effects vehicle position and orientation are due to the movement of water across the vehicle body as waves move past.
	2. The length of the *Phoenix* AUV is small enough that measuring wave height above the vehicle at ½ foot increments gives a realistic representation of the wave forces felt by the vehicle.
Environmental Factors	1. The effects of ocean current are felt by both the AUV and the submarine, thus the relative motion caused by steady-state current can be ignored locally.
	2. The time variation of environmental factors such as change in sea state is slow and can be ignored for the duration of a docking evolution.
AUV docking approach	1. The AUV will always approach the submarines torpedo tube from aft in order to maintain stability and minimize risk of collision.
	2. The AUV approach course will be such that it never passes through the turbulence caused by the submarines propeller(s).
Flat-plate fluid flow theory	1. The submarine is large enough (when viewed from the AUV) that the hull appears as a flat plate.
	2. The majority of drag across the submarine as it moves through the water is pressure drag vice skin friction drag.

Table 4.1. Overview of assumptions made to implement wave motion and flow models.

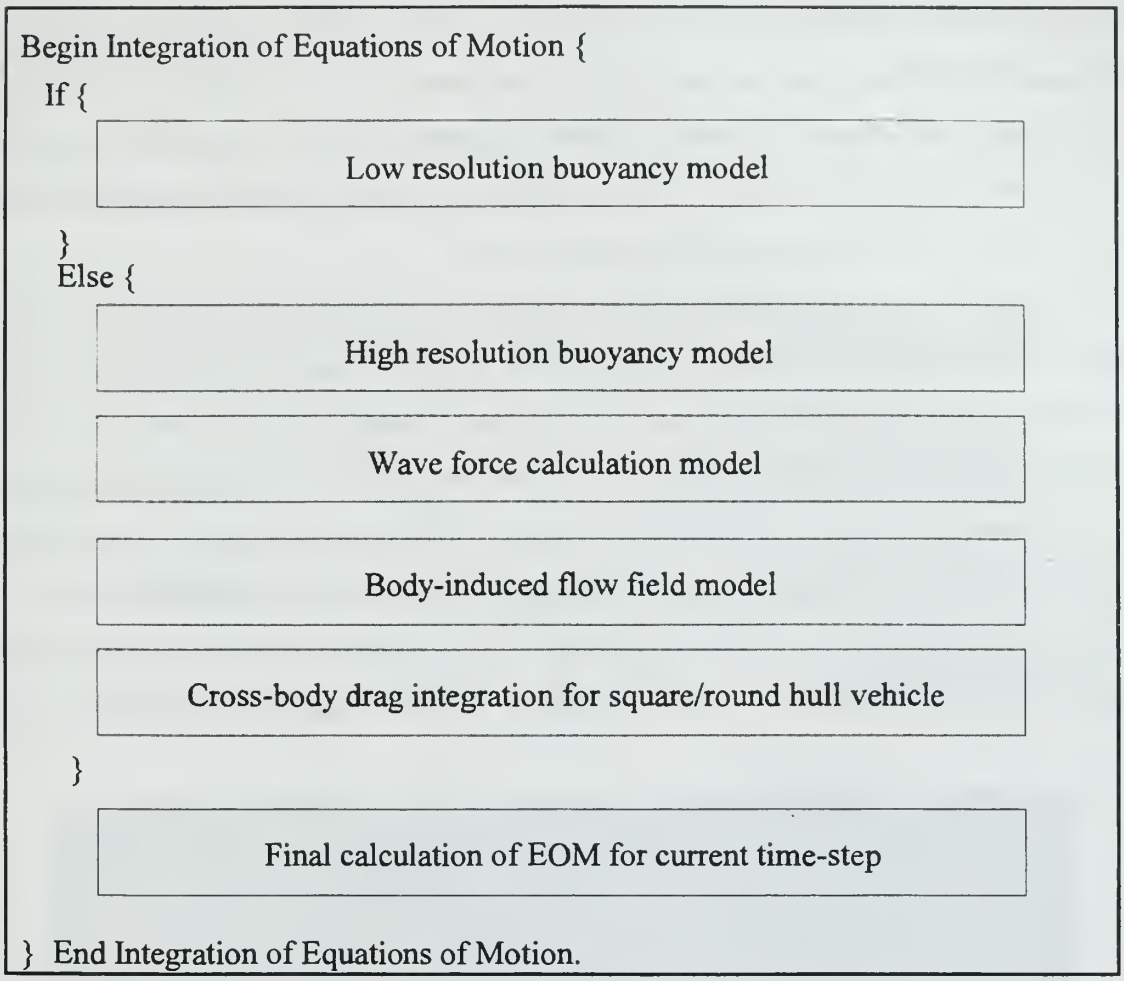


Figure 4.1. Flow of information within hydrodynamic model of the Phoenix AUV virtual environment.

C. BUOYANCY MODEL

Simulation of ocean-going vehicles poses many unique problems. They differ from land-based vehicles by exhibiting six degrees of freedom (DOF) in their movement. They primarily differ from air vehicles (which do move in six degrees) with respect to buoyancy forces. Buoyancy forces differ significantly from the lift experienced in air vehicles. One major difference between these types of forces in the modeling and simulation context is fairly straight forward: for an air vehicle, once the dimensions are known, the lift force exerted on the plane is proportional to air speed. From these forces one can easily calculate position and orientation. In the sea-going vehicle domain, however the dynamics model isn't as simple. Vehicle buoyancy is a major contributor to determining vehicle

position and orientation, and this quantity is truly dynamic when at the surface. Buoyancy varies based on the amount of water displaced at that particular time step, which is an instantly changing irregular 3D volumetric integral. Therefore, in order to maintain real-time response, the calculation must be optimized and flexible. This is especially important in the underwater domain since buoyancy determines whether the vehicle can maintain depth or sink.

The original virtual world hydrodynamics model only handled neutrally buoyant vehicles (Brutzman, 1994). A later refinement estimated buoyancy using box approximations for volume and center of buoyancy (Bacon, 1995). This approach provided reasonably accurate simulation when fully or partially submerged, but may be insufficient when the submersible is continuously operating on the surface or at shallow depths in a surf zone. In this thesis, a high-resolution model is presented that precisely approximates volume and center of buoyancy by evaluating the submersible over 15 separate slices. Each slice has its own buoyancy (and center of buoyancy) that are approximated and calculated every time step. Figure 4.2 shows this type of partitioning applied to the *Phoenix* AUV.

Figure 4.2. AUV broken into buoyancy model slices.

This buoyancy model works well and accurately models vehicle response in a variety of surfaced and broached conditions. When submerged, each piece of the AUV retains its full buoyant force giving the AUV neutral buoyancy. On the surface, the portions of the boat which are out of the

water are subtracted from the net buoyancy giving an approximately correct value for that condition. Additional impacts of buoyancy on a shifted center of buoyancy are calculated piecewise and then summed, using the same computational model provided in (Bacon, 1996). Figure 4.3 graphically represents the buoyancy model from (Bacon, 1996).

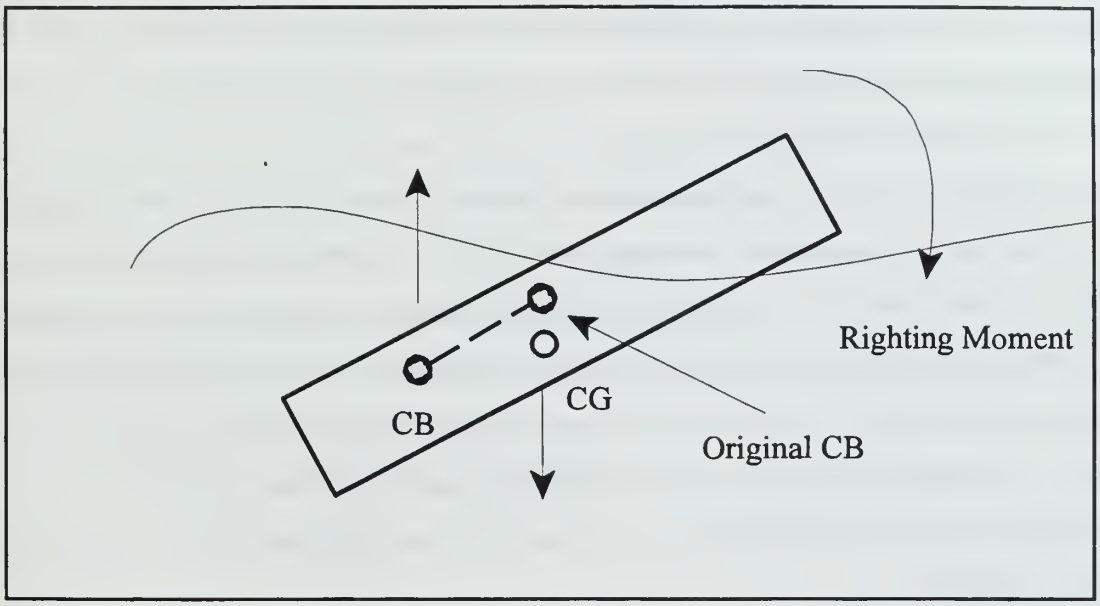


Figure 4.3. Effect of submerged body exiting water on center of buoyancy (Bacon, 1996)

The only condition not considered previously was the effect of ocean waves on vehicle buoyancy. Since the model worked so well for the boundary conditions it seemed appropriate to extend it by adding the needed functionality to accurately model sea state effects on vehicle motion. In order to do so there are several factors to consider and assumptions to make. Issues to address range from what forces are that wave motion produces, how these can be estimated, and finally how are they applied to the vehicle to produce an accurate simulation.

The first assumption to be evaluated applies to the effect wave motion has on the vehicle itself. In other words, how do the forces created by passing waves cause the position and orientation of the AUV to change? When discussing underwater hydrodynamics one often arrives at the effects on a submerged body by multiplying the flow which is present across the body via cross-body drag

calculations. This is the proper method to use when the body is moving through a flow. Wave motion causes the water surrounding the vehicle to move as a whole. As the wave moves down the length of the vehicle the water column surrounding the vehicle is elevated until the crest passes and then it is lowered through the trough. The movement of the water column has an effect on the position and orientation the vehicle. The cross-body drag present is large enough not to be ignored. With that in mind it becomes evident that the most accurate way to simulate wave effects is by evaluating the movement of the water column surrounding the vehicle at every time step, and treating that movement as piecewise forces, derived from vertical and horizontal velocities.

Having decided on the proper interaction between wave and vehicle, one must now evaluate the interval at which to measure the water column. The length of the *Phoenix* AUV is relatively small, 7.3 ft, when compared to the average wavelength of a low sea state. In a sea state of 1 the average wavelength is 20 ft. From the buoyancy model already in place calculations are performed for 15 segments along the body. Continuing this convention for the surrounding water column provides a measurement every 6 inches. This accuracy is more than sufficient given the relative size of the *Phoenix* AUV as compared to the wave. This methodology allows the hydrodynamics model to calculate a force vector representing the water column surrounding the vehicle at the center of each of the 15 body segments.

The final issue to address when discussing the extension of the buoyancy model to include wave effects is how to apply these new force vectors to the vehicle. Superposition of forces is performed in a way which is physically accurate and provides a realistic animation in the virtual environment.

As a wave moves along the length of the *Phoenix* AUV's body force vectors are created representing the direction and magnitude with which the water column is moving at that time step. Using these vectors it is now possible to adjust the buoyancy of each vehicle segment to include wave motion. For each individual segment the buoyancy and wave force vectors are calculated. Then the overall effect on vehicle position and orientation is arrived at by adjusting vehicle buoyancy and the center of buoyancy. Vehicle buoyancy and vehicle center of buoyance are determined using the (Bacon, 1996) methods. The calculations are based on the equation:

$$Buoyancy = \rho g \iiint dV$$

where ρ is the density of water,

g is gravity, and

$\iiint dV$ is the volumetric displacement of a submerged (or partially submerged) body section

at any given time.

This provides an estimated value for buoyancy which is based on the body segments which are actually displacing water. If it is the case that a portion of the vehicle is exposed due to a passing wave, that section does not contribute to the vehicle overall buoyancy and the center of buoyancy is adjusted.

The forces created by the flow of the water particles moved by the wave are applied to the vehicle via cross-body drag calculations. The wave forces are originally determined in world reference frame as velocities. Thus these values must be translated into the local frame and applied along the length of the vehicle. The translation is done using the following equation:

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} X-dot \\ Y-dot \\ Z-dot \end{bmatrix} \begin{bmatrix} \\ R \\ \\ \end{bmatrix}$$

where u, v, and w are body reference frame velocities, X-dot, Y-dot, and Z-dot represent wave velocity in the global reference frame, and [R] is the rotation matrix (Healey, 1998).

This provides a approximation which is both visually accurate and physically correct. The development of an accurate buoyancy model has led to significant advances in the simulation of underwater vehicle characteristics. It is now possible to simulate proper vehicle behavior when submerged, surfaced, or operating in the surf zone. Taking into account simplifying assumptions (such as how wave motion affects vehicle position and orientation) it allows for real-time modeling while maintaining a physically correct basis. Through extending this model to include the effects of wave motion on vehicle dynamics, another step has been made towards accurately simulating all aspects of the ocean environment. One crucial final step remains which is deferred as future work: in-water validation of predicted model results. Nevertheless, current behavior is visually and algorithmically correct enough to justify development of more robust vehicle control laws.

D. WAVE MOTION SIMULATION

> Underwater vehicle design and construction is almost completely preoccupied with environmental considerations. The ocean completely surrounds the vehicle, affects the slightest nuance of vehicle motion and poses a constant hazard to vehicle survivability. Many of the effects of the surrounding environment on a robot vehicle are unique to the underwater domain. Vehicles move through the ocean by attempting to control complex forces and reactions in a predictable and reliable manner. Thus understanding these forces is a key requirement in the development and control of both simple and sophisticated vehicle behaviors (Brutzman, 1994).

With this insight one realizes that in order to provide an arena for the proper development of such a complex robot, the art and science of modeling underwater environmental disturbances must be mastered. These effects must be coupled and studied with underwater vehicle underwater control and dynamic behavior in order to accurately model reality.

Environmental disturbances play a significant role in marine control applications. Their effects dictate how vehicles are designed, constructed and eventually driven. For these reasons the physics of the sea have been studied for many years. The major areas of interest can be broken down into three broad categories: wind, ocean currents, and wind-generated waves.

Each one of these forces is important, having a significant effect on both the novice and expert ocean traveler. The wind plays a major role in the design of ocean-going vehicles, but in the underwater domain its direct affects are minimal. For this reason the introduction of wind to the underwater virtual environment dynamics model is not required. It will be left to those interested in surface modeling and simulation to implement wind in their appropriate environments.

Ocean currents are another environmental disturbance which needs to be evaluated. Any ocean navigator recognizes the effects of set and drift. Ignoring their influence can be a fatal mistake. These currents are also applicable when discussing submerged vehicles. They exist throughout the world and have a large effect in terms of vehicle control. In fact the majority of areas where a robot of this type would be employed have significant currents (i.e. harbors or river outlets) and so ocean current must be dealt with.

This work uses two complementary approaches to the simulation of ocean currents. The first addresses the local frame of reference and the second the global frame of reference. Locally, the AUV is influenced by a set and drift which are present in the area of operation. The direction and force

associated with the current is calculated and factored into the position calculation at every time step. This provides a very simple and accurate modeling of local currents and their affects on the AUV.

The driving force behind much of this simulation is to provide an ability to accurately simulate the forces an AUV will likely encounter while trying to rendezvous with a submarine in the open ocean. For this type of maneuver both vehicles remain in a relatively small area. For example, if the entire evolution was to take 1 hour with a submarine at a maximum speed of 3 kt then the total area traversed is only 6000 yds. This is a small area when contrasted with the vast expanse of the ocean. Thus, while in the global frame of reference there may be many different currents to evaluate and apply to vehicles in the vicinity, for our purposes it can be assumed that both the AUV and submarine are subject to the same set and drift. This assumption provides a useful advantage. Since both vehicles are influenced by an equivalent set and drift the relative motion between the two vehicles induced by these currents are insignificant. This result provides additional computational simplification: relative motion between the AUV and submarine due to steady-state ocean current (set and drift) no longer needs to be calculated.

Ocean currents are a major factor in both ocean navigation and ocean simulation. For that reason the virtual environment developed for the Phoenix AUV fully accounts for the effects of these environmental forces.

Wind-generated waves affect both surface vessels and submersibles which operate at shallow depths. The process of wave generation due to wind begins with small wavelets appearing on the water surface. This increases the drag force which in turn allows short waves to grow. These short waves continue to grow until they finally break and their energy is dissipated. It is observed that a developing sea or storm starts with high frequencies creating a spectrum with peak at a relative high frequency. A storm which has been blowing for a long time (and has reached quasi-equilibrium) is said to create a fully developed sea. After the wind has stopped blowing, low frequency decaying sea or swell is formed. These long waves form a spectrum with a low peak frequency. Wind-generated waves are usually represented as a sum of a large number of wave components (Fossen, 1990).

As early as 1952 researchers were developing mathematical representations of wind-generated wave phenomena (Fossen, 1990). Their efforts laid the groundwork for the definition of a wave-field spectral-density function. In addition, a large amount of data has been collected via

observations. By comparing observed data with the mathematical formulations it has been concluded that the spectral density of the energy spectrum as a function of wave frequency is sufficient to describe a wave environment of fully developed long-crested seas (Reidel, Healey, 1997). This frequency spectrum can be represented as

$$S(\omega) = \frac{\alpha g^2}{\omega^5}\left[-\beta\left[\frac{g}{V\omega}\right]^4\right] \tag{4.1}$$

where α and β are empirical constants defining the spectrum, g is the acceleration of gravity, ω is the frequency, and V is the wind velocity. Equation (4.1) describes a general frequency spectrum which can be used to fit many observations. To make this formulation more specific there are several alternative values for α and β. One can use the Neumann formula, Pierson-Moskowitz (P-M) formula, the Bretschneider formula, or the International Ship Structure Formula to name a few (Fossen, 1990). The most common of these is the P-M spectrum. In the P-M spectrum typical values are $\alpha = 0.0081$ and $\beta = 0.74$.

Inserting values for α, β, and g along with some simplification based on the relationship between significant wave height *(Hs)* and wind velocity *(V)*, a simplified version of the P-M spectrum can be arrived at (Reidel, Healey, 1997). Formula (4.2) is the simplified P-M spectrum.

$$S(\omega) = \frac{8.384}{\omega^5}\exp\left(\frac{-33.52}{H_s^2\omega^4}\right) \tag{4.2}$$

Using the P-M spectrum provides the spectral density. This information is used to find the wave amplitude, which is needed in order to apply the movement of the water column to the AUV as described in section B above. The wave amplitude can easily be represented in terms of spectral density as follows:

$$A^2 = 2S(\omega)\Delta\omega \tag{4.3}$$

Here A is the amplitude and $\Delta\omega$ is the difference in successive wave frequencies (Fossen, 1990). From this equation the amplitude of a wave of interest to the position and orientation calculations of

the AUV can be arrived at. From here it is necessary to compute the value of wave amplitude at the prescribed intervals along the vehicle body. Calculating and combining these many values quickly becomes computationally expensive. The final formulation for this approach to wave simulation is

$$WaveHeight = \sqrt{2\left(\frac{8.384}{\omega^5}\exp\left(\frac{-33.52}{H_s^2\omega^4}\right)\right)\Delta\omega} * \left(\begin{array}{l}\sin\left(\left(t*freq\right)+\left(\lambda*dx\right)\right)* \\ \cos\left(Waveheading - AUVheading\right)\end{array}\right) \quad (4.4)$$

where t is time in seconds, *freq* is wave frequency in radians per second, λ is wavelength in ft, and dx is the distance along the vehicle body.

As indicated by the above derivation, wave spectra are complicated and computationally expensive. It is difficult to perform this type of analysis as part of a real-time simulation. Luckily the diligence and hard work of researchers over the past 45 years alleviates the computational burden through published data tables for various wave spectra. These tables are the result of countless hours of hard work and provide a solid basis for wave simulation. Table 4.2 is an excerpt from a table found in (Bertaux 1976). It gives all the pertinent data required to approximate the P-M spectrum in any sea state ranging from 0-9. The fields of interest are significant wave height, frequency, and wavelength. With this information the state of a wave at any given time step along the body of the AUV can be calculated.

Sea State	Average Significant Wave Height (ft)	Average Period (seconds)	Average Wave Length (ft)	Minimum Duration (hours)
0	0.05	0.5	1.0	18 min.
1	0.18	1.4	6.7	39 min.
2	0.6	2.4	20.0	1.7
3	2.9	4.6	71.0	6.6
3	4.3	5.4	99.0	9.2
6	6.4	6.3	134.0	12.0
6	11.0	4.6	212.0	20.0
7	21.0	10.3	363.0	34.0
8	36.0	12.5	534.0	52.0
9	64.0	16.3	910.0	88.0

Table 4.2. Characteristics of a fully arisen sea. Excerpts taken from (Bertaux, 1976).

Having this data in the form of a lookup table at program run time gives the ability to dynamically apply the affects of a fully developed sea state to the vehicle. The computational advantage gained is tremendous. Equation 4.1 shows how a single wave can be applied to one section of the vehicle using the lookup values.

$$WaveHeight = H_s * \begin{pmatrix} \sin\left(\left(t * freq\right) + \left(\lambda * dx\right)\right) * \\ \cos\left(Waveheading - AUVheading\right) \end{pmatrix} \qquad (4.5)$$

where, Hs is the significant wave height, t is time, λ is wave length, and dx is the distance along the AUV body. This allows the instantaneous height of a wave to be calculated for each segment of the AUV body. This height is then transformed into a buoyancy force as previously described in section B above.

Lookup tables also present the possibility of changing sea state during simulation. Although this functionality is currently implemented it is important to point out that sea state cannot change in nature instantaneously. Nevertheless, looking ahead to long-term scenarios simulating multiple days at sea, it is a worthy feature and was included in the implementation.

Wind-generated waves have an important role when attempting to simulate the physical nature of the ocean environment. They are the most complex of the environmental disturbances adding significant computational complexity to ocean simulation. Despite this complexity their workings are well known. Over 40 years of study have lead to the ability to accurately simulate this phenomenon in a real-time virtual environment.

Environmental disturbances are major factors to consider when simulating the ocean environment. They are an ever-present force which all sea going vessels must deal with, whether surfaced or submerged. Wind, ocean currents and wind-generated waves are significant factors which must be accurately simulated to guarantee the success of any vehicle developed for operation at sea.

E. COMPLEX FLOW-FIELD SIMULATION

Another field which must be addressed in terms of creating a physically based underwater simulation environment is fluid mechanics. Fluid mechanics is an area of study concerned with observing fluid behaviors in order to utilize and control the effects of fluid movement for the benefit of society (James, Haberman, 1988). There are many laws describing the behavior of fluids in motion and various methods of applying them. These laws provide the insight needed to successfully model important aspects of the ocean environment.

The forces generated by fluid movement are of particular concern for the problem at hand: torpedo tube docking of an AUV. When a body moves through a liquid it displaces an amount equal to its volume. This displaced volume of fluid generates forces as it moves and in turn can apply substantial force to other bodies in the area. These forces become significant when considering torpedo tube docking for several reasons.

Torpedo tube docking is a high-risk evolution. There are many things which must be evaluated before this type of exercise can be conducted. A primary area of concern is of safety, for personnel and for both vehicles. A mistake or accident can place the submarine and her crew in great danger.

Depending on the nature of the accident, damage can range from compromising the submarines water-tight integrity, to creating a noise hazard making the submarine easily detectable by adversaries, to crippling the submarine by damaging the propeller or towed sonar array.

In order to avoid the above-mentioned problems, it is of paramount importance that the development of AUV technology be thoroughly tested. To that end it is necessary to ensure that this type of flow simulation can be done in real time. Real-time feedback provides useful insight into vehicle behavior in such a complex environment. It gives both designers and users a chance to view vehicle behavior and actively discuss improvements. The simulation-based design (SBD) methodology is a major factor in assuring that finished products meet user requirements. For that reason, it is essential for robot development.

Another aspect to the importance of body-induced flow has to do with the relative size of the AUV versus that of a submarine. Figure 4.4 shows the difference in size between the two vehicles. The overall submerged displacement of a 688-class submarine is 6900 tons, with a length of 360 ft and a 30 ft beam. When this is compared to the AUV, which in the case of *Phoenix* is 435 lbs displacement, 7 ft length and 1.5 ft beam, it becomes obvious that the force of the water displaced as a submarine moves in the area of the AUV must be evaluated and accounted for.

Flow instabilities are also present along the hull of the submarine. These instabilities, although small when compared to the amount of flow generated by the moving submarine, can be enough to cause major AUV control problems. Large variations in the force and magnitude of movement surrounding the vehicle must be planned for during AUV testing and development. By accurately simulating these variations control algorithms can be tested to ensure vehicle stability in even the worst-case flow situation.

The reasons why this type of physically based simulation is needed are plentiful. The questions to address now include the methodology used in creating such a simulation and any assumptions made to ensure real-time performance.

Intuition tells that since these local forces exist, they must be applied to every vehicle they affect. In other words, the submarine creates a significant field which must be felt by the AUV and any other vehicles around, while the AUV simultaneously generates its own field which affects the submarine. Herein lies the first simplification. Looking again at the size difference between the

submarine and AUV in Figure 4.4 makes it obvious that the displacement force created by the AUV is not significant from the submarines perspective. In fact, it can be ignored completely. Since water displacement is entirely dependent on the size and shape of the vehicle doing the displacement, it is necessary to determine the induced flow on a vehicle-by-vehicle basis, taking into account all the details of the hull in question. This quickly becomes too computationally expensive for a real-time simulation system. Nevertheless, limiting the calculations to one side of the interaction reduces the problem by one half, a significant improvement.

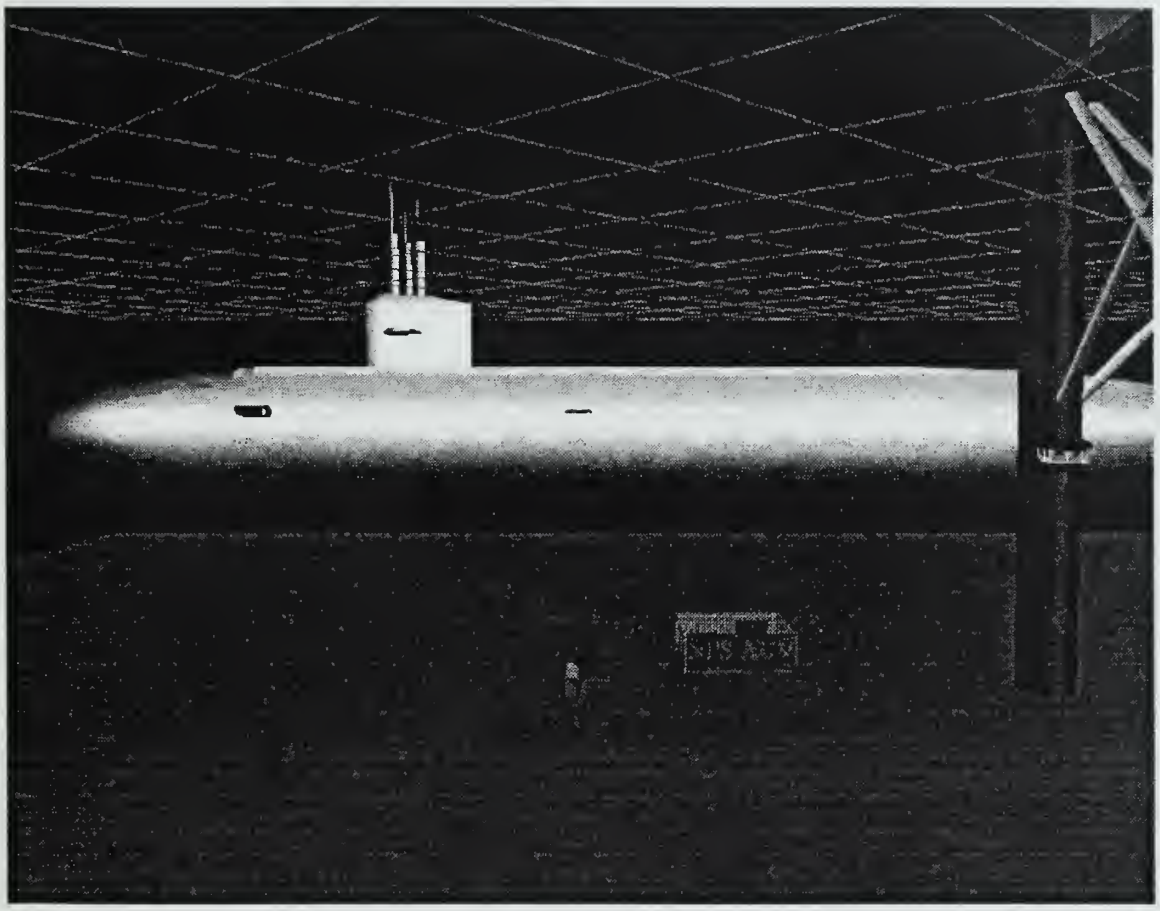


Figure 4.4. *Phoenix* AUV size (center of image) versus 688 class submarine.

Inside the virtual environment an area of influence which surrounds the submarine was created to represent the flow field. This volume encapsulates all the possible positions that the AUV can be in

which are affected by the presence of the submarine. Figures 4.5 and 4.6 give a visual representation of this field, with Figure 4.5 showing the side view. It demonstrates that the field exists from bow to stern of the submarine. This size field allows modeling the approach of the AUV from any position along the hull of the submarine.

Figure 4.5. Side view of 688 class submarine surrounded by its field of influence.

While forces do exist forward and aft of the submarine, they are not of concern when considering a torpedo tube docking solution, since it is assumed that for the docking evolution the AUV will always approach from aft of the torpedo tube door, and it will not take a path which is crosses any of the turbulent flow created behind the submarines propeller. These are reasonably valid assumptions. An approach from aft of the torpedo tube door is a necessary fact. This is because the submarine must always maintain forward headway to ensure adequate depth and heading control. If the AUV were to make an attempt at docking from forward of the submarine, the relative speed would be too large to ensure safety and proper control for the evolution. Therefore, the orientation of the torpedo tube door must allow rear entry. Figure 4.7 shows a proposed outer door configuration for AUV recovery. This provides a unique advantage when conducting the recovery evolution. Adjusting the door to move outward has an advantage of being a relatively simple modification to the current outer torpedo-tube door configuration, and also provides a sheltered lee for the AUV to move into. This lee creates a volume of water for the AUV to perform difficult portions of the docking maneuver while sheltered from most open-water flow.

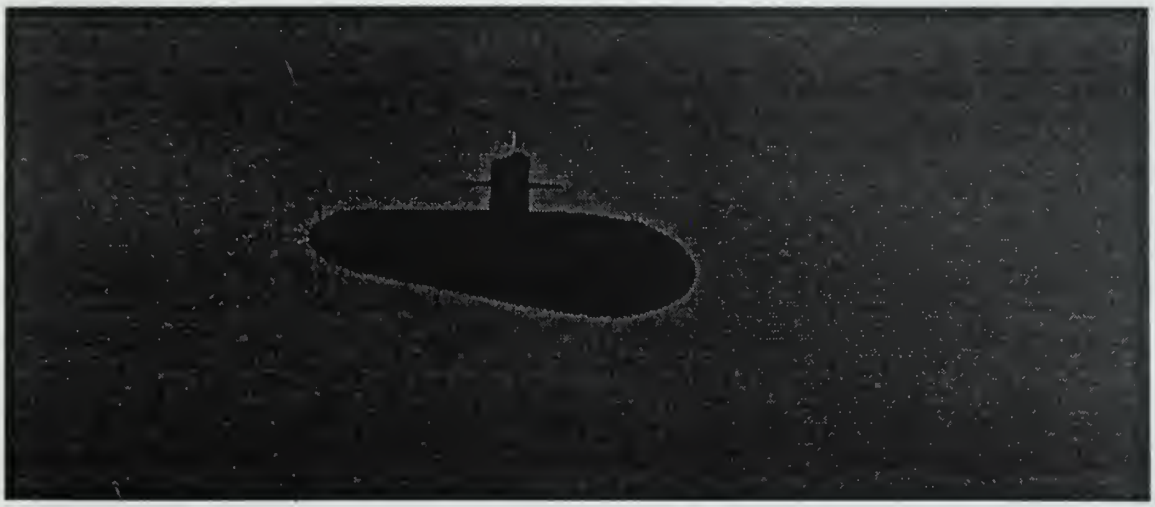


Figure 4.6. Front view of 688 class submarine surrounded by its field of influence.

An approach from the area directly astern of the submarine is not a feasible alternative. The reason behind this is inherent in the AUV mission. One of the primary missions for an AUV is mine detection and avoidance. The circumstances under which this type of mission is conducted are normally those associated with a higher degree of military readiness due to the presence of a possible threat. Standard operating procedure for a submarine in that type of environment requires deployment of a towed array for enhanced enemy detection and acoustical monitoring. With such a tactically valuable (and expensive) piece of equipment trailing from the stern of the submarine, this path becomes unavailable for AUV recovery. Thus the AUV is expected to choose an approach from behind that is along one side of the submarine, vice fully astern.

With that in mind, Figure 4.6 gives a better view of the relative area enclosed inside this flow field. The cylindrical area extends a distance of 30 ft from the side of the hull giving the area a total diameter of 90 ft. Outside this arbitrary volume submarine-induced flow forces are assumed to be negligible.

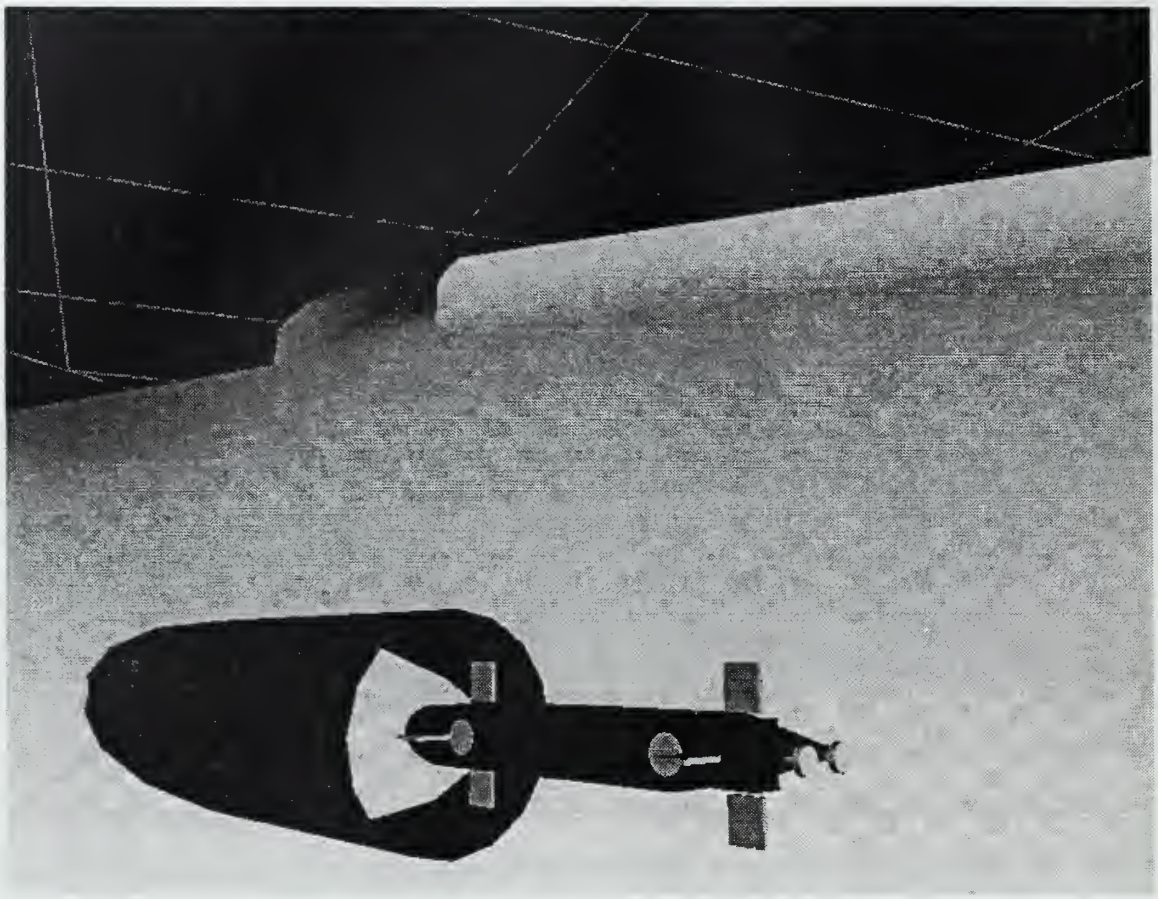


Figure 4.7. AUV docking with outward opening torpedo tube door.

Having examined the size and orientation of the computational model for the flow field, it is now necessary to examine what makes up this virtual flow field. The field is comprised of vectors at ½ ft intervals. Each one is contains a flow component in the X, Y, and Z direction. The vector represents the total amount of flow force (in knots) felt by the vehicle hull at that location relative to the submarine. Graphically, one planar slice of the flow-field velocity looks like Figure 4.8.

This type of grid extends to cover the entire volume within the cylindrical area of influence surrounding the submarine. The orientation is such that the innermost row of flow vectors is flush with the hull and the outermost follows a line 30 ft out from the hull. An exact flow vector within the grid is easily found through position comparison between the AUV and submarine.

Figure 4.8. Grid Level inside submarine flow field.

The construction of the flow field provides some distinct advantages in terms of hydrodynamic modeling. In section B of this chapter the high-resolution buoyancy model was presented dividing the AUV body into 15 slices. With an AUV length of 7.8 ft this breakdown corresponds rather nicely to ½ ft per slice. Thus there is a direct correspondence between the size of the grid and the distance between the center of each section along the AUV body. This allows for rapid flow vector cross-referencing and application during the cross-body drag calculations. There is no need to interpolate between grid positions when retrieving flow vectors for each subsection of the vehicle. In fact, the vehicle can move through the flow field at any random orientation and an exact position is rapidly determined for the flow force component seen by each section.

The flow field design eases computational complexity in another area as well. After the vehicle position is determined and the flow force vector retrieved it must be applied to the vehicle through the equations of motion (EOM). However, by tying the flow field vectors to the submarine (global) coordinate system these forces are not in the AUV (local) coordinate system. In order to use them in the EOM we must translate them into the local system. Looking at the flow field from the AUV point of view, with the submarine on a course of North, it can be said that no matter where the vehicle moves in the world coordinate system these velocities will be present. Flow field contributions are essentially analogous to the world velocities X-dot, Y-dot, Z-dot, where X-dot represents the linear velocity along the North-South axis, Y-dot is the linear velocity along the East-West axis, and Z-dot is the linear velocity along the depth axis. This gives a direct relationship

between the flow vector velocities and those which can be used in the vehicle's hydrodynamic modeling. The velocities can be rotated from the world coordinate system to the local frame of reference as follows:

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} X-dot \\ Y-dot \\ Z-dot \end{bmatrix} \begin{bmatrix} R \end{bmatrix} \tag{4.6}$$

where u is surge, v is sway, w is heave, and R is the (already calculated) rotation matrix. At this point we finally have quantities that can be factored into the calculation of the vehicle's cross-body drag and incorporated into the EOM.

Up to this point, a distinct methodology has been presented for deriving flow forces and applying them to the vehicle being affected. The next step is to elaborate on the actual data that is used to model these complex flow interactions. While looking at this problem several objectives come to mind. Typical computational fluid dynamics (CFD) techniques are far too computationally complex for a real-time system, so flow data must be precalculated whenever possible to support the simulation.

Extensibility is at the core of this approach to flow modeling. By importing flow data at run-time, the virtual environment can be used as a test bed for numerous flow regimes and control environments. The simulation is no longer bound to the specific case for which it was developed (i.e. tube entry). The data used can represent any type of flow desired. Additionally as advances in the field are made, data files can be upgraded to provide a more accurate representation of the fluid's physical behavior. The only requirement is that the data files maintain a readable format, and that requirement too can easily be manipulated.

To create the data needed, a generation program was developed based on Fortran source code from (Schetz, 1965). The original program generated a flow profile at a single point along a flat plate using a two-dimensional (2D) approach to boundary layer incompressible turbulent flow. In order to meet the needs of this simulation the code was converted to C++ and modified to include the flow models required. The program also generates output data files which are imported into the virtual

environment when a docking simulation is initiated. The code for this program is included in Appendix D.

There are two models which are used to create the flow profile down the length of the submarine: one for areas of low turbulence, and one for areas of high turbulence. The majority of the submarine hull is included in the areas of low turbulence; for these sections a flat plate fluid flow model is used. The turbulent portions use a tube-level flow model.

1. Flat-Plate Fluid-Flow Theory

The total drag on a body is due to the sum of two types of drag: pressure drag and skin friction drag. In many cases one of the two types of drag is dominant (John, Haberman, 1988). In the case of a submarine moving through the water, pressure drag dominates.

One flow model which has many similarities to the application in which this data is going to be used is the flat-plate fluid-flow model. It is used to model uniform flow over a flat plate aligned with the direction of the flow. Since the flow in question is created by the submarine moving in a specific direction through the water, it will always be the case that flow is aligned with the flat plate (i.e. submarine hull).

Additionally flat plate theory assumes that over 90% of the drag caused by flow is pressure drag, with only a small fraction due to skin friction. Again this is exactly the case for a submarine moving through the water. The shape and special hull treatment of a submarine are designed specifically to reduce skin friction and reduce undesirable side effects: increased noise levels, reduced propulsion plant efficiency, etc. It can intuitively be asserted that the majority of drag felt by a submarine is pressure drag due to the amount of water it must displace to move through the water.

One remaining question regarding model suitability is whether or not the submarine appears to be a flat plate from the perspective of the AUV. Figure 4.9 shows a picture of the AUV next to the upper 1/3 of a 688 class submarine. What it demonstrates is the fact that the side of the submarine extends 10 ft above the AUV and 20 ft below, looking very much like the AUV against a wall or flat plate. For another perspective, one can look at Figure 4.4 to get a wide angle view.

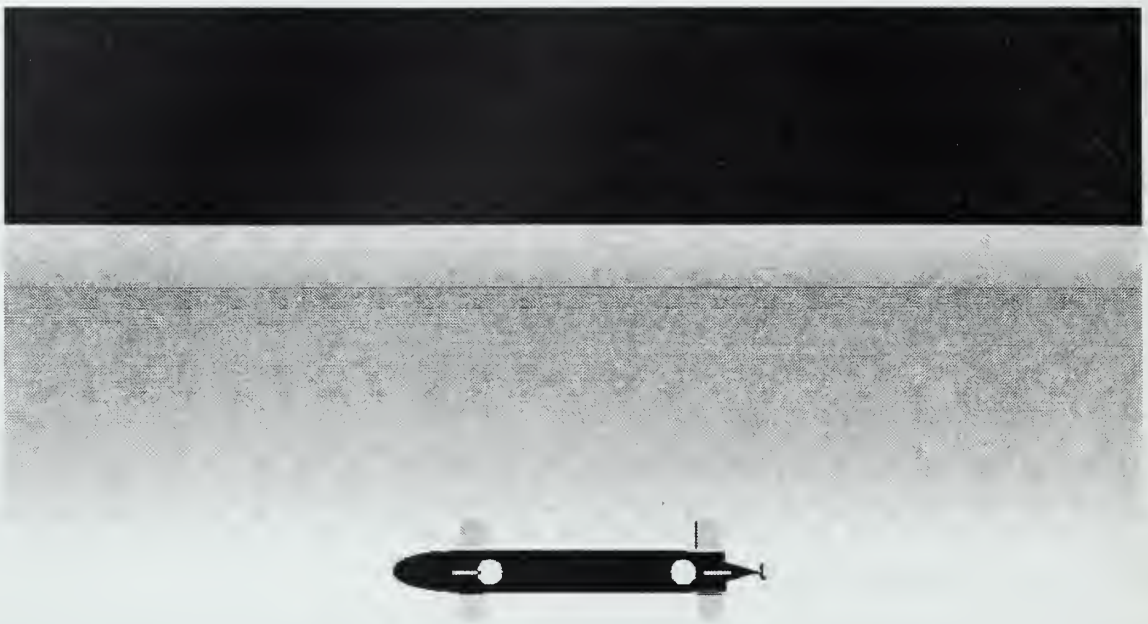


Figure 4.9. Phoenix AUV seen adjacent and parallel to the upper hull of a 688 class submarine.

After viewing the comparison it becomes readily obvious that using flat plate fluid flow model to simulate the flow field in the areas of low turbulence along the submarine hull is a good approximation.

The use of this type of model provides excellent simplification of the run-time flow calculations. It creates simple flow vectors. In fact, they only have one component vice three. Due to the assumption which said that over 90% of the drag which is present is due to pressure drag, not friction drag, two of the three components drop out. The flow force is only present along the axis of the plate. This means that of the three flow vector components, X-dot, Y-dot, and Z-dot, only Y-dot is a nonzero number. The overall profile extending out from the hull is shown in Figure 4.10. What Figure 4.10 shows is how flow changes as one moves out from the hull of the submarine.

Initially flow is at 0% of the open water velocity and it rapidly increases to 100% as the distance from the hull decreases. This demonstrates that the effects of the submarine's presence are larger as the AUV approaches to the hull. The distance at which flow returns to the open water value is approximately 25-28 ft. For that reason the flow field extends 30 ft from the hull, which gives a small buffer for insertion of more severe flow profiles. It is also interesting to note that as the distance of the vehicle moves from the bow to the stern the percentage of open water flow seen by the vehicle moves toward 100% more rapidly. This is an expected phenomenon when using a flat-plate approximation.

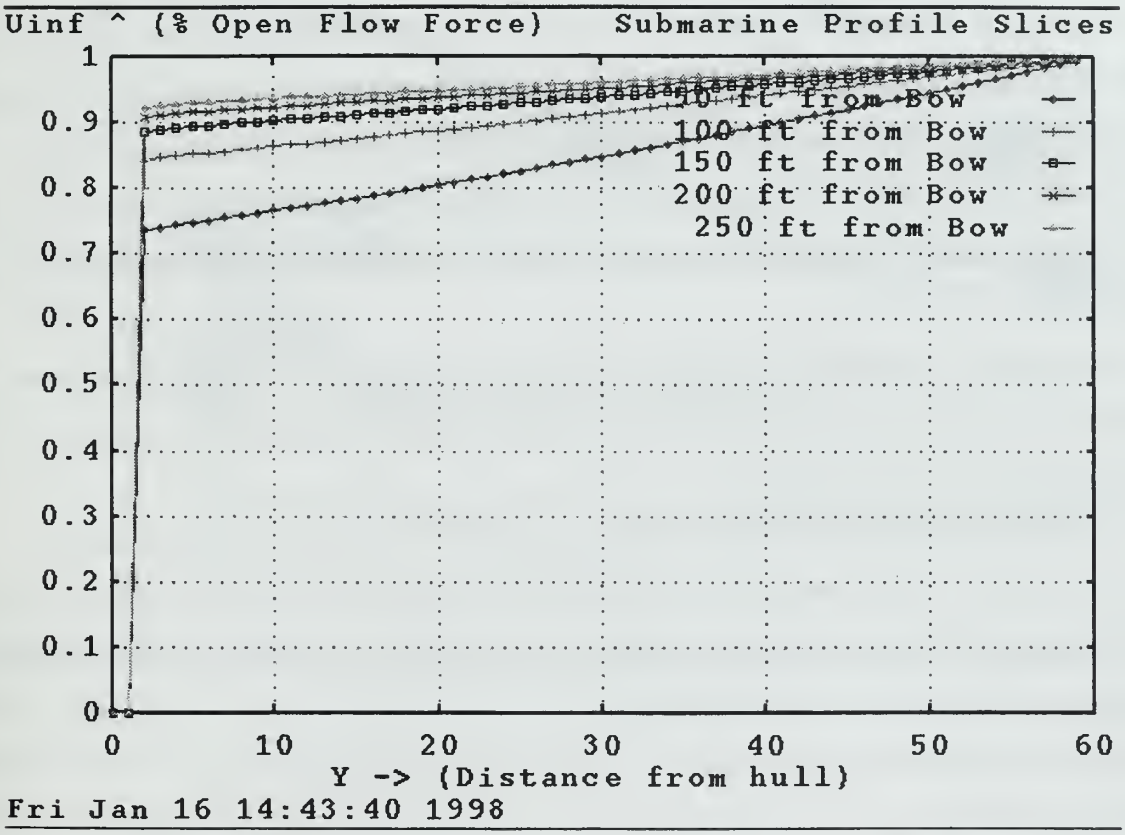


Figure 4.10. Flat-plate flow profile (generated by flow generation code) versus distance from the hull of a 688 submarine, shown at 5 locations along the hull.

The flat-plate fluid flow model provides an excellent match for areas within the submarines field of influence where low turbulence is expected. The assumptions inherent in the theory correspond almost directly with the characteristics of the problem being addressed. This approach

also provides a nice computational advantage since this profile can be used for the majority of the submarine.

2. Tube-Level Fluid Flow

Some areas along the submarines hull cannot be approximated by the flat plate model, since they are subject to much more complex flow interactions. For the submarine docking problem at hand, the area of concern surrounds the open torpedo tube door, beginning slightly ahead of the door and continuing back along the hull until flow is no longer disturbed by the instabilities caused by the open door.

This type of flow profile is similar to those experienced when viewing flow over a cavity. In this case the torpedo tube outer door acts as a shield and the tube area is the cavity. The behavior of flow in this type of situation is very complex and poorly defined. There is a great deal of active research being done on flow fields since many aspects of flow behavior are poorly understood. What is known gives enough of a picture of the flow interaction to make this simulation as accurate as possible.

To accurately model this type of flow there are three portions to take into account: the flow approaching the tube, the flow inside the cavity (and directly aft of the tube) and the rest of the flow path from aft of the tube to the stern.

The flow area forward of the tube is easily modeled. As flow moves along the hull the protruding torpedo tube door forces an outward movement of the flow. In this area each flow vector now has a magnitude in the x direction, out from the hull, and the y direction, along the hull. Figure 4.11 portrays the overall flow picture in this area.

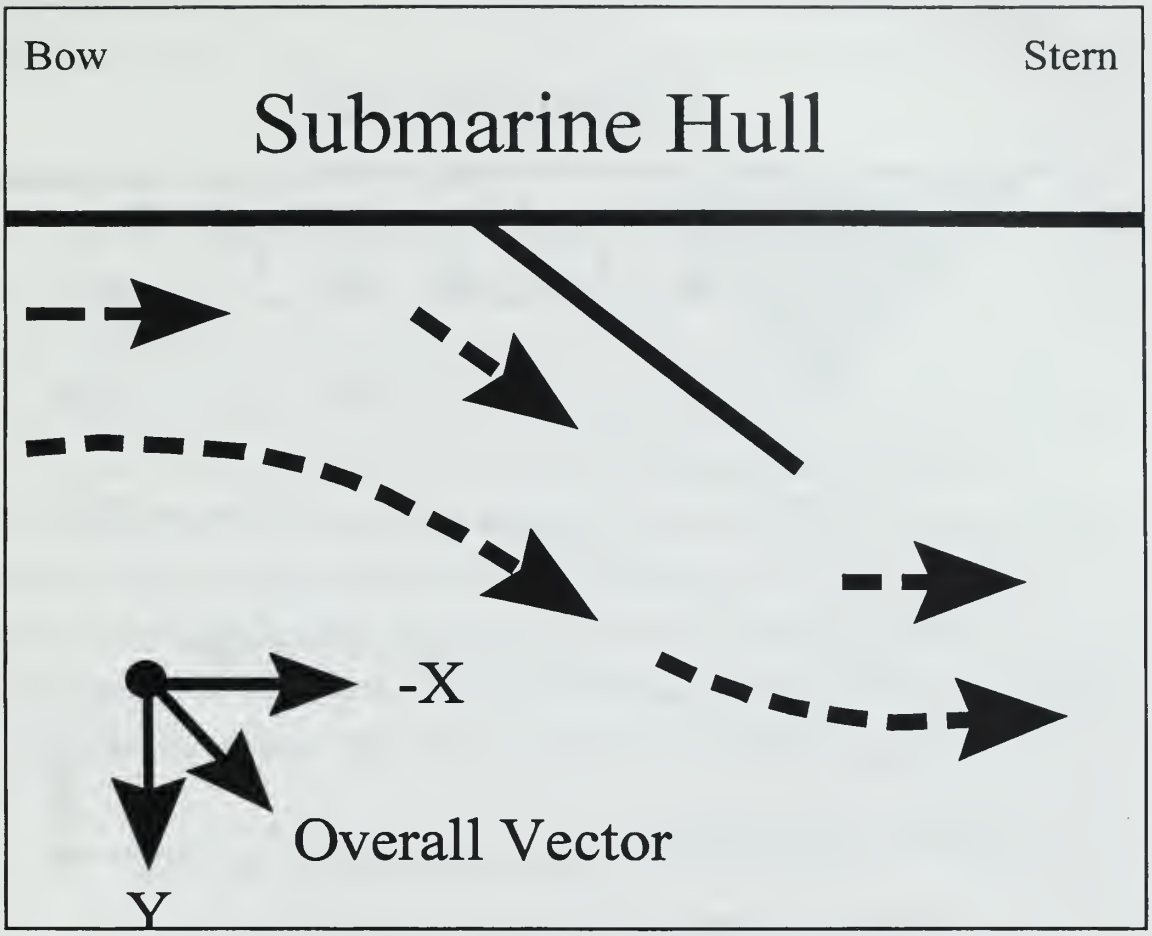


Figure 4.11. Flow interaction as it approaches an open torpedo tube door.

Figure 4.11 depicts how the flow moving along the hull is forced outwards creating a flow force vector which moves away from the hull and aft. After the end of the door is reached the flow interaction becomes very complex. In this area a dead zone is created inside the cavity. In the cavity area there are no significant flow forces at all. As displaced water moves back into the area behind the torpedo tube, a time-varying flow-profile is created. Vortices are created at varying frequencies along the path that follows the door. Figure 4.12 gives a top-down view of what the flow profile resembles at a given moment. The dead zone represented by the shaded area moves along with the submarine as the vortices are created directly aft of the area. Some small flow aft may exist in this dead zone if water is permitted to pass through openings where the door meets the hull. Such small flow may also help stabilize turbulence.

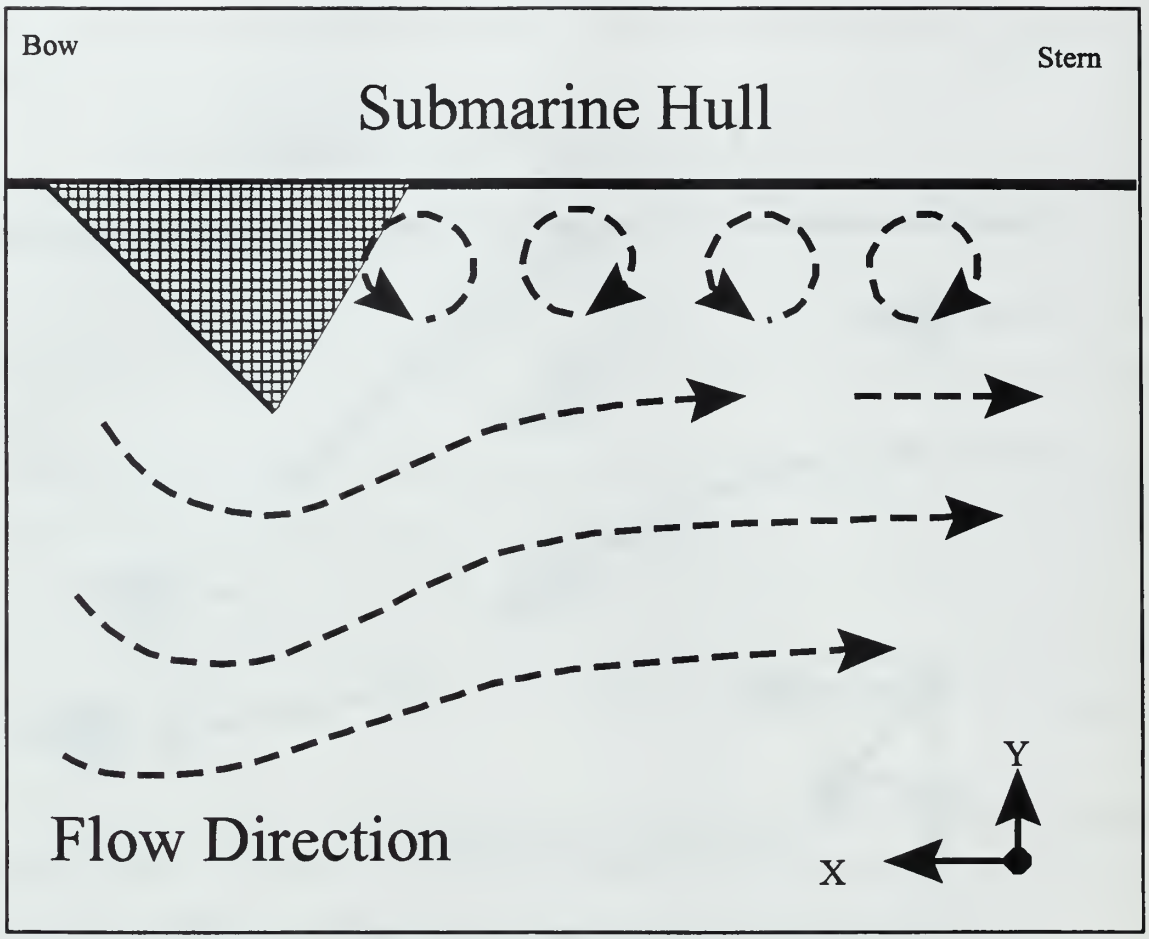


Figure 4.12. Flow movement aft of the torpedo tube door.

The final area to examine is aft of the tube disturbances. In this area flow has stabilized. All the complex interactions caused previously have subsided. Here it is again possible to model the flow field as a flat plate. The only additional disturbances present in this area are those created by pump suctions and discharges.

The flow interactions present on the tube level of the submarine are quite complex. This influence is time varying and not yet fully understood. The majority of the interactions take place on a small scale (less than inches) that a scale of ½ foot intervals between flow measurements can only approximate. In this simulation the variance of flow in this situation has been captured at a low level of detail. As advances are made in the understanding of fluid flow over cavities it will be possible to upgrade the resolution of flow vector data used. For the time being this model provides a plausibly

accurate testing environment for AUV interaction within complex flow situations. By examining approximate but worst-case conditions, an estimate of the magnitude of flow effects can be simulated.

F. EQUATIONS OF MOTION (EOM)

The *Phoenix* AUV virtual environment uses a Newton-Euler approach to the six degree of freedom (DOF) EOM. This accurately models the kinematics and dynamics of a rigid body vehicle moving without constraint (Brutzman, 1994)(Healey, 1998). These equations have been partially verified through extensive testing in the virtual environment coupled with in-water mission analysis. In all cases the results experienced in the virtual environment demonstrated proper behavior, as evidenced by similar results during in-water runs of identical missions. Additional testing is needed to quantify the effects of recent hardware improvements (such as larger shrouded propellers).

In order to properly integrate the flow forces previously discussed into the virtual environment, we examine the EOM looking for the proper terms to modify. Equation 4.7 is the sway equation of motion from (Brutzman, 1994) which is implemented in the *Phoenix* AUV virtual environment.

Sway Equation of Motion (4.7)

$$\left(m-\frac{\rho}{2}L^3Y_{\dot{v}}\right)\dot{v}+\left(-mz_G-\frac{\rho}{2}L^4Y_{\dot{p}}\right)\dot{p}+\left(mx_G-\frac{\rho}{2}L^4Y_{\dot{r}}\right)\dot{r}$$
$$=m\left[-ur+wp-x_Gpq+y_G\left(p^2+r^2\right)-z_Gqr\right]$$
$$+\frac{\rho}{2}L^4\left[Y_{pq}pq+Y_{qr}qr\right]$$
$$+\frac{\rho}{2}L^3\left[Y_{up}up+Y_{ur}ur+Y_{vq}vq+Y_{wp}wp+Y_{wr}wr\right]$$
$$+\frac{\rho}{2}L^2\left[Y_{uv}uv+Y_{vw}vw+u\left|u\right|\left(Y_{u|u|\delta_b}\delta_{rb}+Y_{u|u|\delta_s}\delta_{rs}\right)\right]$$
$$-\frac{\rho}{2}\int_{x\,tail}^{x\,nose}\left[C_{dy}h(x)(v+xr)^2+C_{dz}b(x)\left(w-xq\right)^2\right]\frac{(v+xr)}{U_{cf}(x)}dx$$
$$+(W-B)\cos(\theta)\sin(\phi)$$
$$+\left(\frac{2\ lb}{24^2\ volts}\right)\left[V_{bow-lateral}\left|V_{bow-lateral}\right|+V_{stern-lateral}\left|V_{stern-lateral}\right|\right]$$

The variables and coefficients in the sway equation of motion are defined in (Brutzman, 1994).

The terms of the EOM define all the major force contributors to vehicle motion. Similar equations exist which define surge, heave, roll, pitch, and yaw. These can be found in (Brutzman, 1994). For this discussion, the sway equation of motion is used as an example.

For the problem at hand it is necessary to integrate the additional flow force contributions into the EOM. Since the body-induced flow forces are primarily due to the cross-body drag of the water as it passes over the vehicle, the logical place to insert these factors is the term dealing with cross-body drag. To do so we must first examine precisely how these velocities induce drag.

1. Round Hull Derivation

Cross-body drag is calculated to incorporate the force generated by the motion of water over a rigid body. When determining the magnitude and direction of this force one must know the shape and size of the body being effected. In past versions of the EOM it has been assumed that the shape of the body was always cylindrical. This provides a solution to the six degree of freedom model that is general enough to accurately depict the cross-body drag for the majority of submerged vehicles.

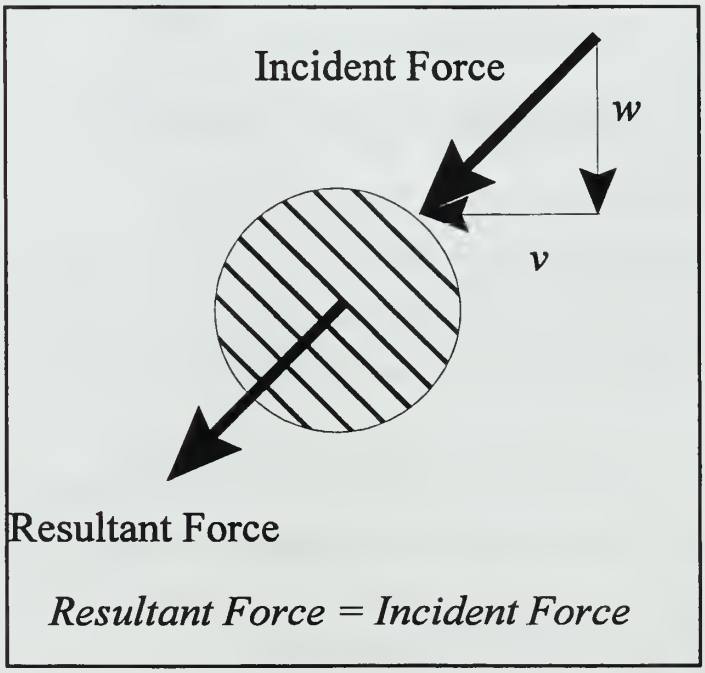


Figure 4.13. Flow force incident upon a round body.

For a round body the force applied by the water is always in the same direction as the original

force. When calculating the cross-body drag the v and w components can be found by the breaking the force up into the respective components, Fy and F_W, then normalizing. Figure 4.13 gives a visual representation of these forces and their components.

The terms can be defined mathematically as follows (Healey, 1998):

$$F_y = \frac{1}{2} C_D \rho v^2 dx \qquad F_w = \frac{1}{2} C_D \rho w^2 dx \qquad \text{(4.8) and (4.9)}$$

Going one step further it can be said that

$$v = v_0 + xr \qquad (4.10)$$

and

$$w = w_0 + xq \qquad (4.11)$$

Taking these facts and adding a term for normalization gives the final version of the cross-body drag formulation.

$$F_y = \frac{1}{2} C_D \rho (v + xr)^2 \frac{(v + xr)}{U_{cf}} dx \qquad (4.12)$$

$$F_w = \frac{1}{2} C_D \rho (w + xq)^2 \frac{(w + xq)}{U_{cf}} dx \qquad (4.13)$$

$$\frac{\rho}{2} \int_{x\,tail}^{x\,nose} \left[C_{dy} h(x)(v + xr)^2 + C_{dz} b(x)(w - xq)^2 \right] \frac{(v + xr)}{U_{cf}(x)} dx$$

These forces are incorporated into the EOM as one of the multiple terms present. Translating the forces into the rigid body's reference frame and integrating their effect along the horizontal axis of

the body results in the fifth term on the right hand side in the sway equation of motion.

This term represents the effects of flow forces across the spherical rigid body. As is evidenced by the derivation of the forces, shape of the rigid body does make a difference. In the case of the *Phoenix* AUV, which has a rectangular shape, this generic model is inaccurate.

2. Square Hull Derivation

When a flow force is incident upon a rigid body that does not have a spherical shape the direction of the resultant force is not necessarily the same direction the force came from. Figure 4.14 demonstrates this fact. Given a force incident upon a rigid body with a shape that is rectangular, the resultant force is not equal in magnitude or direction to the resultant force.

The initial formulation from the round hull derivation of cross-body drag is similar, but the terms cannot be normalized using U_{cf}. The reason for this is that U_{cf} is radialized and it is no longer the case that the forces are radially symmetric. This causes some differences to exist between the cross-body drag term in the EOM for a spherically shaped body versus the same term in the EOM for a non spherical body. The formulation for F_y and F_w in this case are given in (4.14) and (4.15) (Healey, 1998).

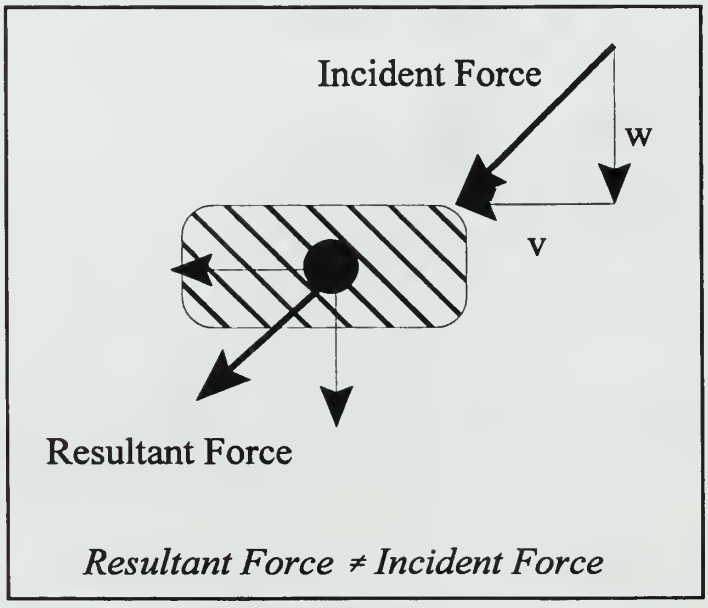


Figure 4.14. Flow force incident upon a non-spherical rigid body.

$$F_y = \frac{1}{2} C_D \rho (v + xr)^2 \, dx \tag{4.14}$$

$$F_w = \frac{1}{2} C_D \rho (w + xq)^2 \, dx \tag{4.15}$$

Taking these revised force formulations and incorporating them into the EOM to make a more specific set of equations yields the sway equation of motion given in (4.16). This equation provides an accurate evaluation of cross-body drag and is computationally less complex than the spherical hull case. Thus it provides two advantages. Similar specializations are performed for the equations of motion for heave, pitch, and yaw.

Square Hull Sway Equation of Motion (4.16)

$$\begin{aligned}
&\left(m - \frac{\rho}{2} L^3 Y_{\dot{v}}\right)\dot{v} + \left(-m z_G - \frac{\rho}{2} L^4 Y_{\dot{p}}\right)\dot{p} + \left(m x_G - \frac{\rho}{2} L^4 Y_{\dot{r}}\right)\dot{r} \\
&= m\left[-ur + wp - x_G pq + y_G (p^2 + r^2) - z_G qr\right] \\
&+ \frac{\rho}{2} L^4 \left[Y_{pq} pq + Y_{qr} qr\right] \\
&+ \frac{\rho}{2} L^3 \left[Y_{up} up + Y_{ur} ur + Y_{vq} vq + Y_{wp} wp + Y_{wr} wr\right] \\
&+ \frac{\rho}{2} L^2 \left[Y_{uv} uv + Y_{vw} vw + u|u|\left(Y_{u|u|\delta b} \delta_{rb} + Y_{u|u|\delta s} \delta_{rs}\right)\right] \\
&- \frac{\rho}{2} \int_{x\,tail}^{x\,nose} \left[C_{dy} h(x)(v + xr)^2 + C_{dz} b(x)(w - xq)^2\right] dx \\
&+ (W - B)\cos(\theta)\sin(\phi) \\
&+ \left(\frac{2\ lb}{24^2\ volts}\right)\left[V_{bow-lateral}|V_{bow-lateral}| + V_{stern-lateral}|V_{stern-lateral}|\right]
\end{aligned}$$

In order to improve the accuracy of the *Phoenix* AUV virtual environment without limiting its extensibility, both models are incorporated in the implementation. The user can select the shape of the hull being tested in the virtual environment, and based on that selection the appropriate version of the EOM will be used.

G. SUMMARY

The environment plays a major role in all aspects of AUV research and design. If a virtual environment is to act as a true test bed for newly engineered, devices it must take into account the forces of nature. The virtual environment used for testing and development of the *Phoenix* AUV incorporates many environmental factors into its simulation. The virtual environment is truly physically based. The enhancements added throughout this work incorporate a highly detailed buoyancy model, wave motion simulation based on the Pierson-Moskowitz wave spectrum, a detailed methodology for simulating body induced flow forces, and a specialization of the equations of motion to offer a higher resolution method for modeling cross-body drag on non spherical rigid bodies.

All of these factors serve to enhance the realistic behaviors which are present inside the *Phoenix* AUV's virtual environment. Improvements of this type can only better performance leading to improved design, testing, and final product.

V. IMPLEMENTATION

A. INTRODUCTION

As with any technically based research, there needs to be some proof of correctness for the various theories presented. This chapter examines two separate implementations of the *Phoenix* AUVs virtual environment. The initial implementation was done using C++ and Silicon Graphics OpenInventor Application Programmers Interface (API). This version runs solely on Silicon Graphics workstations. A second platform-independent, implementation was created to run on any machine upon which the Java runtime environment is present. Each version uses the DIS protocol for networking enabling the user to run a mix of viewer and dynamics versions if desired.

B. C++ AND *OPEN INVENTOR*

The virtual environment is primarily comprised of three components. In their original implementation the dynamics program was written in C++, robot execution level in C, and the viewer in C++ using the OpenInventor API (Brutzman, 1993). Each component was thoroughly tested and the performance was validated by real-world experiments. With this history in mind, the logical choice is to first implement the flow and buoyancy models in C++ before the transition to Java.

The wave model and the submarine-induced flow forces both relate to the environments effect on the AUV, thus both are implemented in the dynamics code. The code itself is located in a function called calculate_equations_of_motion() which is included in Appendix A.

The algorithm for the wave model uses the P-M spectrum as discussed in Chapter IV. For each time step, the height of the wave is calculated for the fifteen sections down the AUV body length. At each block, a force vector proportional to the wave height is assigned. After stepping down the length of the body the vectors are added and averaged to get an overall force that acts upon the entire AUV. This superposition vector is used to adjust the center of buoyancy of the vehicle prior to completing the integration of the equations of motion. The overall effect is a pitching moment that is proportional to the wave position over the body of the vehicle. Figure 5.1 presents the pseudocode for the wave algorithm.

```
for ( 1 to number of sections)    {
   Calculate wave motion buoyancy for this block
   if (depth > 20 ft) {
      Reduce wave buoyancy effect due to depth
   }
   Determine overall direction of wave motion
}  //end of for loop

for ( 1 to number of sections) {
   Adjust vehicle buoyancy based on wave motion
   Adjust center of buoyancy based on direction of wave motion and pitch angle
}
high-resolution buoyancy force calculation complete
```

Figure 5.1. Pseudocode for wave motion effect algorithm.

The next algorithm incorporated into the equations of motion provides the forces created by the submarine's flow field. As described in Chapter IV the flow field exists in the area of water surrounding the submarine. The implementation of this algorithm is more complex than that of the wave model. It requires several calculations for each section of the AUV body. Each one providing information for the next iteration down the body. Figure 5.2 contains pseudocode of the general algorithm.

The first step is to determine whether or not the AUV is inside the influence field of the submarine. This is done by comparing the position of the AUV to the position of center of the submarine. Having knowledge of the volume of water which falls into the flow field allows for quick determination of whether or not submarine flow interactions must be calculated.

```
Compare the  position of the AUV to that of the submarine
if (inside flowfield) {
  Set flowfield flag to TRUE;
}
for ( 1 to number of sections ) {
  Calculate the x-position of the current section
  Calculate the y-position of the current section
  Calculate the z-position of the current section
  Determine the position of the AUV relative to the submarine center
  Index into flow field matrix and retrieve the flow force at that point, without interpolation
}

Calculate a rotation matrix to translate x,y,z force components into body coordinates

for (1 to number of sections) {
    Translate current section forces into body coordinates

  //Cross Body Drag Contribution
   Calculate flow_field_sway_integral
   Calculate flow_field_surge_integral
   Calculate flow_field_heave_integral
   Calculate flow_field_roll_integral
   Calculate flow_field_pitch_integral
   Calculate flow_field_yaw_integral
}
Add flow field integrals to cross-body drag integrals
```

Figure 5.2. Pseudocode for flow field algorithm.

Once it is determined that the AUV is inside the flow field, more detailed calculations are performed. These include finding the position of each section's center and the position of that section inside the flow field. Figure 5.3 demonstrates the geometry of calculating the x position of a section. Knowing the heading of the AUV and the orientation of the world axis, each coordinate position can be determined using simple geometry. A similar method is used for determining the y and z values for a section of the hull.

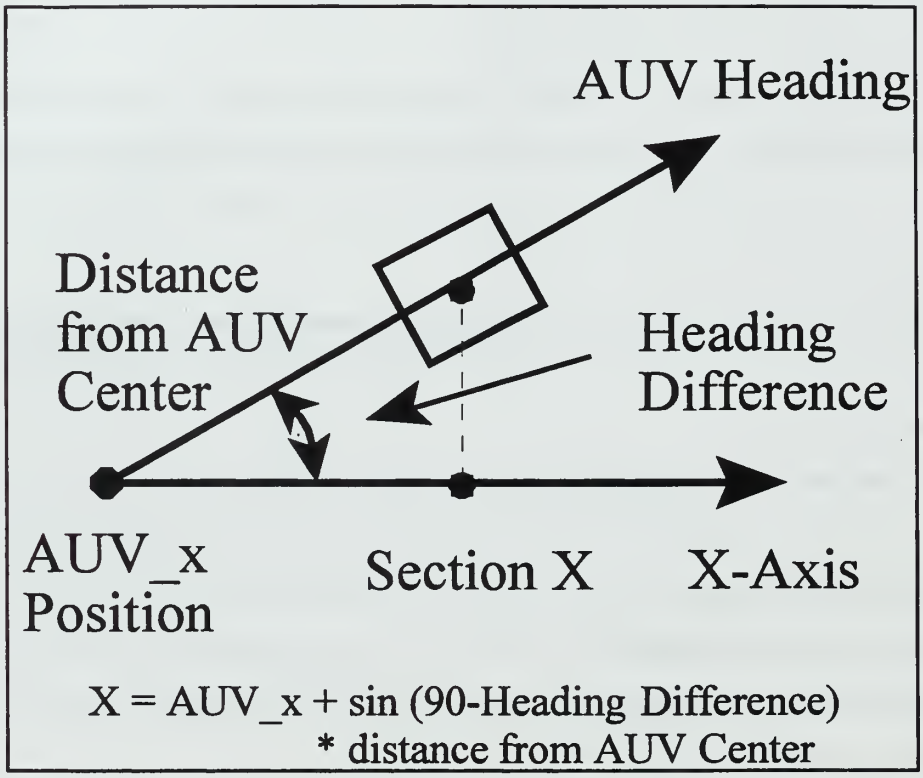


Figure 5.3. Geometry of calculating an AUV sections X position relative to the center of the AUV.

Once the x, y, and z coordinates of a body section have been determined, they are used to calculate the position inside the flow field. This process takes two steps. First, the relative position of the AUV to the submarine is determined, then the relative coordinates are converted into flow-field indices.

The X component represents the AUV position along the hull of the submarine with a value of zero ft meaning at the bow and 360 ft at the stern. The X value corresponds to the distance from the bow of the submarine in feet. This is determined by simply taking the difference between the X

position of the submarine and the X position of the AUV.

The next relative position component is the radial distance of the AUV from the centerline of the submarine. It combines both the y and z positional components into a single number (Equation 5.1). This is used because the construction of the flow field is such that the grid is anchored at the center of the submarine. To get at any particular position out from the hull the overall radial distance is needed as an index.

$$RadialDist = \sqrt{(Ydifference)^2 + (Zdifference)^2} \qquad (5.1)$$

The conversion step takes the X-position with the radial distance and then converts the pair to flow field indices. The numbers cannot be taken directly because the grid has a resolution of ½ ft increments. This causes the grid positions to range from zero to 720 along the hull and zero to sixty out from the hull. The conversion simply takes the calculated coordinate and makes it into an integer position which can be used in the flow field system.

Having the proper indices available it is now possible to retrieve the values of flow forces seen by the section of the AUV being considered. The flow induced forces are stored velocities in the world coordinate system. To apply them to the equations of motion in the local coordinate frame they are translated into body coordinates using equation (4.6). No interpolation is performed due to already high resolution, reducing computational complexity. The forces are then applied to the EOM by adding their effects into the calculation of cross-body drag.

The algorithms for wave motion and body-induced flow forces are tightly interlaced in the dynamics code. Many of the calculations required for the wave model are also needed for the flow field and vice versa. By conducting the computations in tandem the added execution time is kept to a minimum. It enables an already computationally complex virtual environment the ability to become more accurate, yet still run in real-time.

Other changes to the virtual environment involved additions to the viewer program. The viewer provides a window into the virtual environment. For the experiments conducted in this thesis it is necessary to visualize the AUVs approach and rendezvous with a submarine. In its initial incarnation the virtual environment did not contain a submarine. It was primarily used to develop

robot control algorithms for open water situations. It also provided a replica of the NPS test tank for small-area testing which might later be conducted in the actual tank. As the focus was moved away from small-area operations to open-water docking, a 688 class submarine model was added to the environment. The addition turned out to be an invaluable visualization tool and presented an added feeling of AUV scale in the open ocean.

Implementing the wave buoyancy model and the body-induced flow algorithm in C++ provided an excellent stepping stone in the development process. Knowing the original version of the virtual environment was validated and sound allowed for quick isolation of possible modeling errors. Any instabilities encountered were localized to either of the new algorithms. It also provided the groundwork for the later implementation of dynamics in Java. In summary: development and implementation of the high-resolution models was successful.

C. JAVA AND VIRTUAL REALITY MODELING LANGUAGE (VRML)

After proving the validity of the models proposed by this thesis, the next step was to provide a platform-independent version of the code. This was not possible using C++ and the OpenInventor API. C++ is plagued by compiler differences from one platform to another, and the OpenInventor API is primarily for Silicon Graphics workstations, although a port of the library to Windows95 has recently been completed. In any case the only way to provide true platform independence was to use languages which were not platform specific. For that reason Java in combination with VRML are the language of choice.

The first portion of the virtual environment converted was the dynamics program and associated functions. This was a relatively straightforward port of C++ to Java. While some problems were encountered due to differences in language functionality (i.e. object handling, operator overloading, pointers, etc.) it was more time consuming than complex. Appendix B contains a list description of the code for the virtual environment dynamics in Java. The functionality and object hierarchy of the dynamics program is the same in the Java and C++ version, as is most program syntax. Flow field matrices proved too large for current PC java implementation, so this section of code is commented out.

The second step in the move towards platform independence was to re-implement a viewer

program in a platform-neutral way. VRML was used to describe the virtual environment scene graph with Java as the language to animate the objects in the environment. This gives anyone with an Internet browser (and appropriate VRML plug-in) the ability to view the virtual environment.

The difficulties in porting the viewer to a platform independent scheme were primarily due to problems with Internet browsers and VRML plugins. Due to the early development stage of both of these technologies, many inconsistencies were encountered. These implementation problems were handled by the DIS-Java-VRML working group. Numerous work-arounds and problem solutions were developed in the working group forum. They provided the Java implementation of the DIS protocol and the bridge from multicast broadcast to unicast so the VRML scene can be animated via the script node. Figure 5.4 shows the underlying architecture of the Java-VRML version of the *Phoenix* AUV virtual environment. The source code for the VRML scene is available via references in Appendix B. The source for the DIS-Java-VRML library is available at [*http:www.stl.nps.navy.mil/dis-java-vrml*].

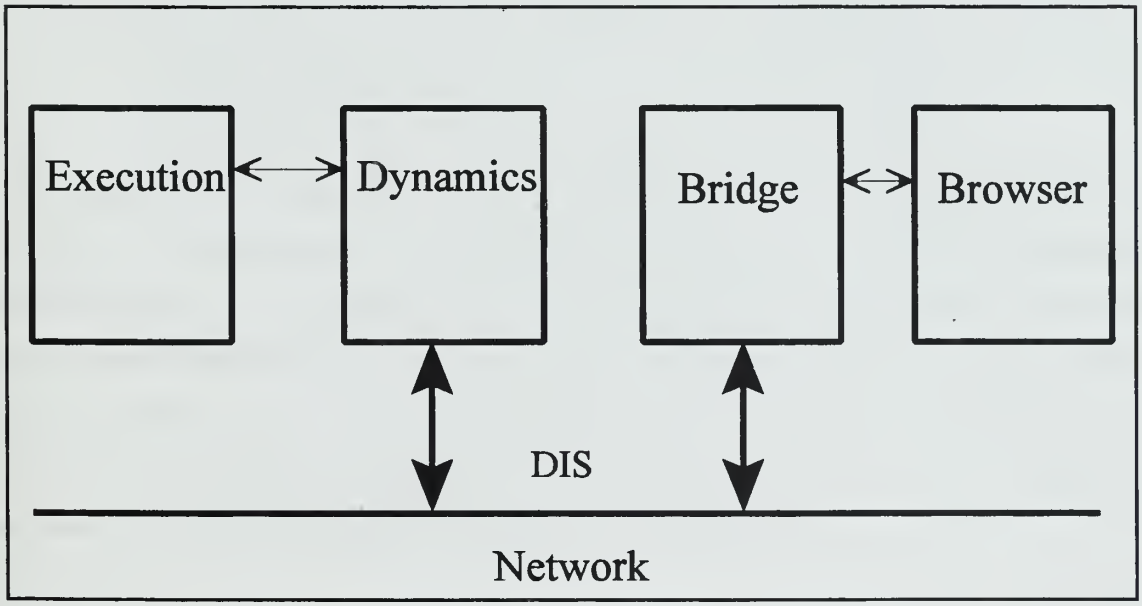


Figure 5.4. Platform-independent architecture for *Phoenix* AUV virtual environment.

The transition from a platform specific virtual environment to a platform-independent one is a large step forward in simulation technology. As personal computers become better and platform-independent languages more robust, this transition can only get easier.

D. SUMMARY

This chapter describes two separate implementations of the *Phoenix* AUV virtual environment. The C++/OpenInventor version is an extension of the original virtual environment, providing the speed and additional functionality needed to perform the SBD of torpedo tube recovery, while still using a validated base environment for quick isolation of problems. The DIS-Java-VRML implementation gives the virtual environment portability. It is now possible to view simulations from any machine having Internet connectivity.

VI. EXECUTION LEVEL AND VIRTUAL DOPPLER SONAR

A. INTRODUCTION

The *Phoenix* AUV execution level software controls all the hardware onboard the vehicle, ensuring all hard real-time deadlines are met. It uses a sense-decide-act loop to iterate through the process of polling sensor and effector state, deciding what actions are required and then commanding devices to the proper state. The devices that are controlled range from motors and servos to gyros and sonars. This chapter discusses a new sensor, a doppler sonar unit, which is simulated in the virtual environment and used for advanced control law testing.

B. TRITECH DS30 PRECISION DOPPLER SONAR

Doppler sonar works on the basic theory of measuring the frequency shift in a transmitted signal. The TRITECH DS30 precision doppler sonar is a highly accurate, reliable, compact unit designed for underwater vehicle use. It provides measurements of vehicle speed by analyzing the frequency shift in the back-scattered signal (MECCO, 1997). The DS30 is comprised of three major components: a digital micro controller, an analog control circuit, and a transducer.

The digital micro controller controls the transmitter, the receiver, a Programmable Logic Device (PLD), and manages data communications to an external control device. Data output provides a bottom speed vector, water mass speed vector, and the current depth. Both vectors can be presented in either polar or rectangular format. The speed vectors are given in meters per second, with an accuracy of one centimeter per second and depth indication is accurate to one centimeter. Communication with the sonar is conducted through a 9600 baud serial line. This line handles both data output and command input. Figure 6.1 gives the specification data for the DS30.

Power	24 VDC
Power consumption	200 mA average, 1 A peak
Operating frequency	1 MHz
Operating range for seabed tracking	2-30 meters
Tracking modes	Velocity relative to seabed & velocity relative to seawater
Data rate	Up to 5 updates per second
Communication	RS232 as standard, RS485 as option
Operating velocity	0-3.75 meters/second
Velocity accuracy	2.5 centimeters/second
Velocity resolution	0.5 centimeters/second
Transducer	4 beam Janus array
Configuration	Convex, beams @ 45° to vertical
Source level	217 dB re. 1 uPa @ 1 meter
Depth rating	1000 meters
Length	360 millimeters including connector
Body tube diameter	120 millimeters
Maximum diameter	130 millimeters
Weight in air	5.5 kilograms
Weight in water	2 kilograms

Figure 6.1. Tritech DS30 precision doppler sonar specification from (MECCO, 1997).

The DS30 analog control circuit is comprised of one receiving channel and one transmitting channel. It achieves a four-channel system by multiplexing the receiver/transmitter circuits to each transducer element. The transducer is constructed with four elements, each at 45° offset from the normal axis (MECCO, 1997). Figure 6.2 is a picture of the DS30 mounted on the front of the *Phoenix* AUV.

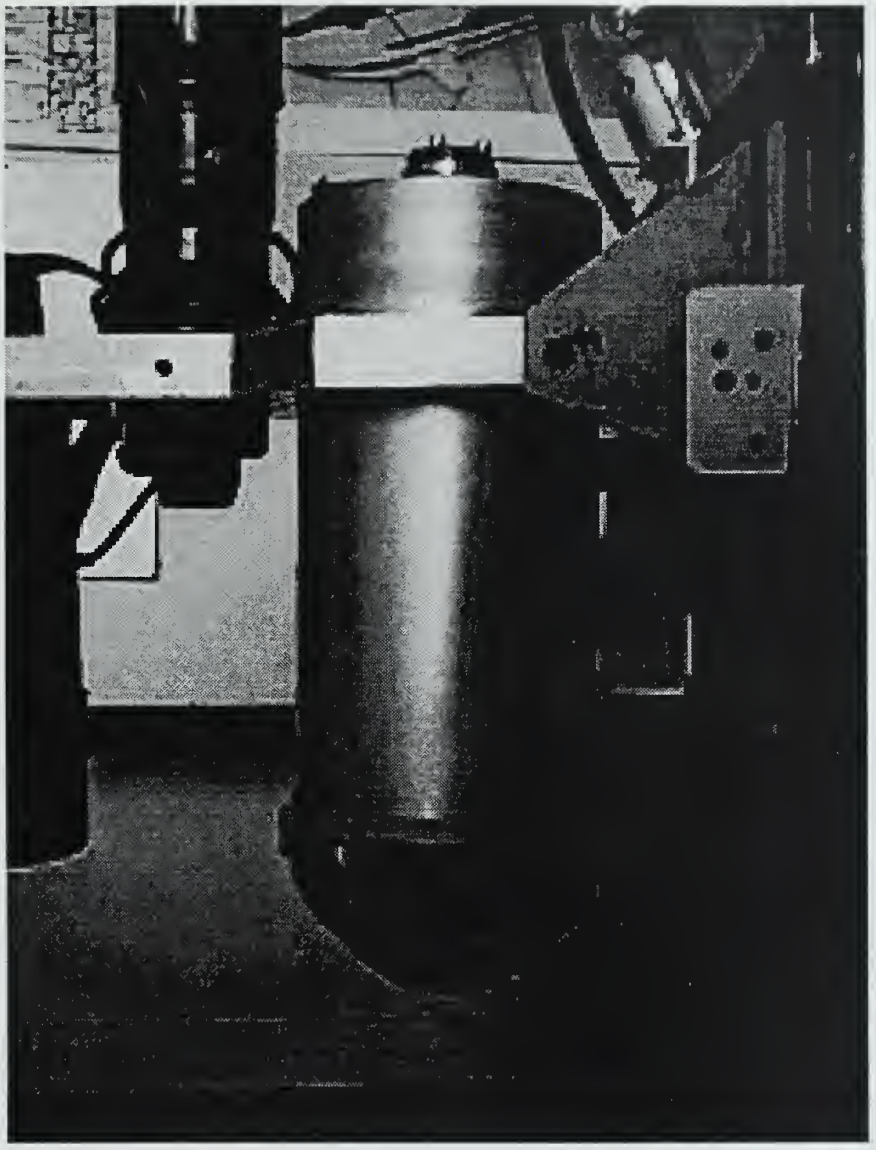


Figure 6.2. Tritech DS30 precision doppler sonar mounted on the nose of the *Phoenix* AUV.

The Tritech DS30 precision doppler sonar is a unit which is well suited to the *Phoenix* AUVs needs. It is an accurate sensor which can be easily integrated into the vehicle due to its low cost, low power requirements, and standard communication setup. The DS30 provides all the needed components to accurately measure cross-body flow and use that information for enhanced modes of AUV control.

C. SONTEK ACOUSTIC DOPPLER VELOCIMETER (ADV)

The Sontek acoustic doppler velocimeter (ADV) is another device which can be used to determine cross-body flow for the *Phoenix* AUV. It is an acoustic doppler current profiler and has the ability to determine water velocity in three component axis. The Sontek ADV works by measuring the velocity of a volume of fluid that is directly above its probe and has an accuracy of 0.1 millimeters/second. This type of technology is designed to accurately measure ocean current, and it is well suited to be used as a cross-body flow sensor on the *Phoenix* AUV. The specifications for this device are given in Figure 6.3. Eventually in-water testing will also examine whether the velocity update rate is sufficient fast for real-time maneuvering control.

Power	24 VDC
Power consumption	3 Watts average
Operating frequency	10 MHz
Data rate	0.1 to 25 Hz
Communication	RS232
Operating velocity	2.5 meters/second
Velocity resolution	0.1 millimeters/second
Depth rating	30 meters
Length	407.9 millimeters including connector
Body tube diameter	76.2 millimeters
Maximum diameter	133.4 millimeters

Figure 6.3. SonTek acoustic doppler velocimeter (ADV) specification from (SonTek, 1997).

The Sontek ADV is a new device which is specifically designed for shallow water operations.. Figure 6.4 is a picture of the SonTek ADV. This unit is currently being evaluated for use in the next incarnation of the Phoenix AUV.

Figure 6.4. Sontek acoustic doppler velocimeter (ADV).

D. VIRTUAL SIMULATION OF DOPPLER SONAR

Before a new sensor is integrated into the *Phoenix* AUV and deployed for in-water testing, SBD practices suggest that its use be simulated and evaluated first. To this end a virtual sensor was implemented to represent the Tritech DS30 precision doppler sonar. It was created to provide the same functionality in the virtual world as is expected from its performance in the open ocean environment. The simulation was developed in several steps to ensure accuracy of each portion of the model.

First, the doppler sonar was integrated into the execution level code in function closed_loop_control_module(). In the sense phase of the execution level's sense-decide-act loop, variables were added to read the sensor input. Since the true hardware is not present in the simulation this was accomplished by adding the needed parameters to the state vector. The state vector represents the value (or state) of every sensor and effector in the vehicle. This is the information that is sent to the dynamics model to provoke the appropriate forces, or measure the needed quantities, in the surrounding virtual environment.

The next step included the addition of a sensor model to the hydrodynamics code. This was necessary so the dynamics model might return proper values to the execution level, when the execution level indicated use of the doppler sonar via the state vector. For this thesis a simple zero-order model was constructed. The value returned by dynamics is the true error-free value of the quantity being measured. In other words, there is no error due to random noise or uncertainty inserted in the response. The assumption for beginning testing and evaluation is that the sensor will work exactly as described by the technical documentation on the sensor. Granted this is not always a valid assumption, nonetheless it is sufficient for initial testing. Random noise and errors can easily be incorporated at a later time, since it is more appropriate to examine performance failure modes after the sensor had been proven useful in the optimal case.

The final step for sensor integration is to provide a facility to exercise all the control and data modes of the DS30. To accomplish this step, commands must be added to the execution level command language. The device itself has a series of roughly ten commands, ranging from reset to designating sampling frequency. During initial device testing it is only necessary to accurately parse the output data. The data provided by the unit in its normal mode contains both the unit's speed over

ground and the speed of the water column. These values are all that is needed for the cross-body flow calculation input to the vehicle control laws. Thus, implementation of a full command language for the unit was deferred as future work.

The simulation of the Tritech DS30 precision sonar is a useful tool for testing and development of robot control modes. By separating the process into well defined components researchers are able to keep robot-specific code in one module and hydrodynamic code in another. The processes communicate via a state vector which is read by the robot execution level as if it were getting the data directly from the actual sensor. This makes the transition from the simulation environment to the real world transparent from the robot's point of view.

E. ENHANCED CONTROL LAWS

The addition of any useful sensor is an iterative process. In order to improve vehicle control the sensor must be evaluated, prototyped, and integrated into the existing system. The *Phoenix* AUV control laws are no exception. These laws are finely tuned to provide a properly damped control system. The addition of a doppler sonar which can provide cross-body flow information requires the adjustment of these laws to incorporate (and take advantage of) the new information available.

In order to use the information available from the doppler sonar unit, it is necessary to evaluate which positioning mechanism can best use this information. The question is primarily whether to adjust the control of the rudders, the fore and aft thrusters or both sets of effectors. In the case of the rudders, which are primarily used during forward transit, cross-body flow information is not significant. Since the employment of this sensor is envisioned to be a mechanism which allows the AUV to predict turbulent flow areas before the entire body is pushed unstable by them, rudders are not the most effective control devices. Instead the virtual cross-body flow sensor is used for adjusting thruster control laws.

Thrusters can be used to orient the vehicle horizontally and counteract cross-body flow quite effectively. As the AUV moves into a turbulent area the sensed cross-body flow can be used to activate a thruster force counteracting the instability caused by the turbulence. Thus the thruster control laws are adjusted to include a term for cross-body flow data. Figure 6.5 gives the new thruster control law.

```
AUV_stern_lateral =  ( - k_thruster_psi *  normalize2(psi-psi_command)
                     - k_thruster_r   * r)
                     + k_thruster_hover   * cross_track_distance
                     - k_thruster_current   * AUV_oceancurrent_x
                                            * sin_psi
                     + k_thruster_current   * AUV_oceancurrent_y
                                            * cos_psi
                     + k_sway_hover         * v
                     + k_thruster_current   * cross_body_flow_u[12];
```

Figure 6.5. New thruster control law for the AUV stern lateral thruster.

This new control law can be written in two ways: as a straight sensor input or as a smart sensor. The straight sensor input takes the value sensed by the doppler sonar and uses it in the forward lateral thruster control law since that is the relative location of the physical sensor. A smart sensor is one which "dead reckons" the sensed cross-body flow using the vehicles recent movement history and can predict the flow at both forward and aft lateral thrusters. In this case both thrusters can be effectively employed to counteract the turbulent flow encountered. Both of these control law options were implemented and tested. The results are presented in Chapter VII.

The integration of a new sensor into AUV control is a significant task. Simulation testing shows that the doppler sonar sensor provides useful information which needs to be integrated into vehicle control. This section presented an alteration to the vehicle control laws for cross-body thrusters in addition to two methodologies for sensor employment. These are the first attempts at harnessing the wealth of information available from such a useful piece of equipment.

F. SUMMARY

The abilities of the *Phoenix* AUV to see the environment in which it operates are limited by its sensor suite. Improving the way the AUV observes the environment and increasing sensory input

provides additional information for vehicle control law action. In order to thoroughly test the use of these sensors, they must be integrated into the execution level code and tested in the target environment. One such device which appears to greatly improve vehicle control is a precision doppler sonar. This chapter demonstrated the simulation of a precision doppler sonar and the integration of its output information into enhanced vehicle control laws. The results acquired from simulating such a sensor demonstrate how useful the information is to AUV control and the significance of sensor simulation in the vehicle design process.

VII. SIMULATION RESULTS

A. INTRODUCTION

This chapter outlines and presents experiments conducted to validate the simulations implemented in this thesis. The experimental design is addressed along with the measures used to qualify and quantify results. Then the final results are presented in concise tables which are supported by plots provided in Appendix C.

B. DESIGN OF EXPERIMENTS

When developing tests to validate the implemented cross-body flow and associated AUV control algorithms, it is necessary to examine two areas: the high-resolution buoyancy model with wave action effects, and the flow field interaction algorithm. In order to properly test each area separate experiments were designed. Each experiment focuses on the concerns associated with the particular application being tested.

For the testing of the high-resolution buoyancy model, a series of simple missions were conducted under various sea-state conditions. During these experiments the *Phoenix* AUV was placed on a base course heading into the sea at a speed which was high enough to allow the vehicle to maintain heading, while low enough to prevent vehicle control from masking the effects of the sea. The mission script used (*mission.script.SeaStateTest*) is included in Appendix C.

During these tests it was also necessary to determine specific factors which might be used to quantify and qualify the results that were found. In terms of vehicle stability while heading into a sea the primary factors of concern are maximum pitch angle and pitch rate. These parameters are appropriate because they directly indicate the vehicle's stability and ability to maintain control as it moves through the seas.

The termination consideration for these tests is determination of what sea state to end the analysis. While the hydrodynamics model may be able to produce a sea state ranging from one to nine, at some point the vehicle becomes so unstable that its presence is not worthwhile. Thus the analysis range from minimal sea states (1) to a sea state in which the vehicles stability was in questionable for greater than 50% of the run.

In the first group of experiments, the high-resolution buoyancy model is tested in sea states ranging from one to a sea in which the vehicle does not maintain stability. During each exercise the AUV proceeded on a course directly into the sea at a speed high enough to maintain steerage. These runs fully exercised the high resolution buoyancy model and the wave motion simulation. The experiments are designated SS.1 through SS.5 corresponding to sea states one through five.

The second set of experiments are aimed at testing the flow-field simulation and vehicle control using cross-body flow sensor input. The goal is to bracket the torpedo tube docking problem by running experiments in flow conditions which ranged from lower than expected turbulence levels to well above expected turbulence levels. Additionally, the results with the cross-body sensor available are compared to runs with the sensor absent. Table 7.1 shows the naming convention for all of the experiment variations.

	No Flow Field	Normal Flow Field	Extreme Flow Field
No Flow Sensor	Experiment CBF.1	Experiment CBF.2	Experiment CBF.3
Flow Sensor	Experiment CBF.4	Experiment CBF.5	Experiment CBF.6
Smart Control Sensor	Experiment CBF.7	Experiment CBF.8	Experiment CBF.9

Table 7.1. Variation of conditions for experimental cross-body flow (CBF) missions.

While running the CBF missions, three criteria were chosen to quantify observed results: vehicle distance from track, time to regain track in turbulence and whether or not the vehicle collided with the submarine hull during the docking mission. These parameters are appropriate because they directly address AUV survivability during torpedo tube recovery. If flow perturbations cause significant variance from the preplanned track, then AUV endurance and control become a concern. If collision occurs, then safety of the AUV and the submarine become significant. Thus these metrics provide a useful measure of AUV performance in the presence of turbulent flow. In each run the commanded path was identical, as specified by the mission script *mission.script.FlowFieldTestLoop* included in Appendix C.

The combined results of these experiments provide a sound measure of the algorithms developed in this thesis. They address the performance of the high-resolution buoyancy model, the

wave motion simulation, the turbulent flow field simulation, and the cross-body flow sensor control algorithm. These simulation experiments also serve to illustrate the accuracy of the physically based models they are derived from. Any errors in a cross-coupled model as complex as vehicle dynamics will almost certainly cause vehicle instability in the virtual environment.

C. RESULTS

The experiments described in the previous section were conducted ising the Phoenix AUV execution level and the C++ implementation of the virtual environment. The results were measured in terms of the metrics discussed, collected in the form of parameter graphs included in Appendix C and summary tables presented in this chapter.

The experiment which exercised the high-resolution buoyancy model and the wave motion simulation provided interesting results. The vehicle was able to maintain stability in sea states ranging from zero to five. In sea state five the vehicle was unstable for roughly 60% of the 5 minutes that the run lasted. Nevertheless, the control algorithms were able to maintain a relatively stable attitude while the vehicle was moved by large wave swells. Vehicle pitch rate and maximum pitch angle varied greatly between sea states as expected. Table 7.2 contains the data addressing these metrics. Worth noting is the dramatic increase in pitch rate and pitch angle when the sea state progressed from four to five. It is likely that shorter sampling rates, modified control coefficients and the predictive control algorithm specified in (Riedell, Healey, 1998) can improve performance even further. Surprisingly the vehicle was able to maintain control in sea states well above what was expected. Further in-water testing is definitely needed to validate these results.

Experiment	SS.1	SS.2	SS.3	SS.4	SS.5
Pitch rate (deg/sec)	0.2	1.0	1.5	4.5	11.0
Maximum pitch angle (deg)	0.35 (0.15 avg)	1.0 (1.0 avg.)	3.0 (1.5 avg.)	7.0 (4.5 avg)	40.0 (11.0 avg.)

Approximate amount of time vehicle was unstable	0 %	0 %	1 %	15 %	60 %

Table 7.2. Experimental results of AUV stability in various sea states.

The results from the experiment which tested vehicle control in the flow field are also significant. For the most part the results are as expected. The first metric used, collision with the hull, gives a boolean result for each run. Table 4.3 contains the data collected for runs under all of the various conditions. In the no flow and normal flow conditions, vehicle control was stable enough to prevent the AUV from colliding with the submarine hull. In the extreme flow case the AUV collided with the hull in every case, regardless of sensor control. The point at which collision occurred was at the pump suction inlet along the hull. In the extreme case the suction flow simulates a flow of 1.3 knots vice 1.0 knot in the normal flow case. This slight increase in flow force creates a significant problem for AUV control. Despite flow turbulence near the torpedo tube door, no collisions occured in the door area.

Flow Regime	No Sensor	Simple Control Sensor	Smart Control Sensor
No Flow	CBF.1: No Collision	CBF.2: No Collision	CBF.3: No Collision
Normal Flow Profile	CBF.4: No Collision	CBF.5: No Collision	CBF.6: No Collision
Extreme Flow Profile	CBF.7: Collision at pump suction only	CBF.8: Collision at pump suction only	CBF.9: Collision at pump suction only

Table 7.3. Cross-body flow (CBF) experimental results of AUV collision with submarine hull.

The other measures evaluated are the overall distance the vehicle departed from its pre-planned track due to turbulent flow and how long it took to return to track after departure. Departure from track was measured in the most turbulent areas within the flow field: the pump suction, pump discharge, and torpedo tube door docking. Table 7.4 presents the results of these measures at the three points for each experiment. These results are as expected when moving from one flow condition to another. Yet the results within each condition show that the cross-body flow sensor input to thruster

control has no significant effect on distance from track, but does aid in the time needed to get back on base course. The lower return times are most likely due to the fact that the thrusters are helping to stabilize the vehicle when the cross-body flow sensor is used. More accurate testing of the adjusted control laws is needed. Nevertheless, the control results are promising. The additional sensor does in fact provide some additional thruster control ability. It is left to future researchers to implement a more effective control law.

Flow Condition	Position	No Sensor	Simple Control Sensor	Smart Control Sensor
No Flow	Distance from track at pump discharge (feet)	0	0	0
No Flow	Time to regain track (seconds)	0	0	0
No Flow	Distance from track at pump suction (feet)	0	0	0
No Flow	Time to regain track (seconds)	0	0	0
No Flow	Distance from track at torpedo tube entry (inches)	0	0	0
No Flow	Time to regain track (seconds)	0	0	0
Normal Flow Profile	Distance from track at pump discharge (feet)	13	43	13
Normal Flow Profile	Time to regain track (seconds)	55	52	45
Normal Flow Profile	Distance from track at pump suction (feet)	●	5	●
Normal Flow Profile	Time to regain track (seconds)	44	43	40
Normal Flow Profile	Distance from track at torpedo tube entry (inches)	8.7	6.5	6.0

Normal Flow Profile	Time to regain track (seconds)	N/A	N/A	N/A
Extreme Flow Profile	Distance from track at pump discharge (feet)	18	18	18
Extreme Flow Profile	Time to regain track (seconds)	60	57	56
Extreme Flow Profile	Distance from track at pump suction (feet)	10	10	10
Extreme Flow Profile	Time to regain track (seconds)	47	44	42
Extreme Flow Profile	Distance from track at torpedo tube entry (inches)	10.3	9.6	8.7
Extreme Flow Profile	Time to regain track (seconds)	N/A	N/A	N/A

Table 7.4. AUV distance from track under various cross-body flow (CBF) experiment conditions.

The results arrived at in these experiments provide useful insight into the algorithms implemented in this thesis. The high-resolution buoyancy model, the wave motion simulation and the turbulent flow field simulation appear to be accurate and give consistent results which are in line with expectations. On the other hand, the experiments also demonstrate that the control algorithms which use doppler sonar input for cross-body flow measurement need to be tuned. The virtual environment thus provides a useful tool for control law testing, which can be further improved by incorporation of results from in-water validation tests.

D. SUMMARY

Experiments are a useful tool in any researcher's repertoire. They serve to verify the theories upon which technological innovations are based. This chapter presents the design of experiments that are performed in simulation and used to test the models developed in this thesis. The experiments address testing of the high-resolution buoyancy model, the wave motion simulation, turbulent flow-field simulation, and enhanced vehicle control using a doppler sonar employed as a cross-body flow

sensor. Additionally, design of experiments and the metrics used to measure results are discussed to provide the reader with a good understanding of what success is based on. These experiments are a useful means to rigorously test the *Phoenix* AUV dynamics model. The simulation results give hard data demonstrating the stability and accuracy of the hydrodynamics model and associated cross-body flow control laws.

VIII. CONCLUSIONS AND RECOMMENDATIONS

A. CONTEXT

This thesis has taken an in-depth look at methods of modeling environmental effects in a virtual environment. The net result is a virtual environment for the NPS *Phoenix* AUV which is more robust and better simulates the environment for which the AUV is being designed. These improvements are aimed at enhancing the SBD process, allowing engineers to rigorously test the performance of AUV systems prior to deployment in the vehicle.

B. RESEARCH CONTRIBUTIONS AND CONCLUSIONS

Throughout this thesis the intention has been to provide solutions to real-world problems. With that in mind, even simulation results provide useful contributions to the modeling community along with interesting experimental results for those concerned with autonomous robot simulation. The simulation enhancements include a high-resolution buoyancy model for wave simulation, an extensible body-induced flow methodology, and an approach to platform-independent distributed simulation environments.

The high-resolution buoyancy model divides the modeled vehicle into fifteen separate sections. Each one is then evaluated for its contribution to the overall vehicle buoyancy. This approximation gives an accurate representation of vehicle posture at shallow depths in various sea states. It proved to be quite useful when evaluating vehicle operation in various broach postures. Once fully submerged, at a depth where no portion of the vehicle is consistently exposed, its more accurate modeling characteristics were less apparent, again as expected. The high-resolution buoyancy model is a needed improvement with no noticeable consequence in terms of real-time performance.

The ability to test AUV control in various sea states also turned out to be a significant improvement in vehicle modeling. The effects of wave motion come into play in shallow-water operations as well as during submarine docking evolutions. At shallow the forces of wave motion cause changes in vehicle velocities, accelerations and buoyancy. These factors need to be considered when fine-tuning control algorithms. They bring to light possibilities of over-sensitve control laws which can cause vehicle hunting and instability. During docking evolutions at submarine periscope depth, wave movement is also a factor. Although increasing depth for this type of operation reduces

wave-induced forces, they are still present and need to be dealt with.

Another improvement in functionality of the virtual environment is the ability to simulate body-induced flow forces. The methodology used for this simulation is completely extensible. As researchers desire to change flow conditions, a simple data file replacement can import the new flow field into the virtual environment. By bracketing the submarine docking problem with worst-case and best-case flow instability, simulation results indicate that a feasible solution exists. A slight modification to current torpedo tube door mechanisms might thus provide an avenue to AUV recovery by naval submarines.

The use of a doppler sonar to determine cross-body flow is also evaluated. This type of sensor, having the ability to provide speed over ground or speed through the water, enabled enhanced AUV control in complex flow fields. Its employment allows the robot to predict and compensate for movement instability using real-time flow condition feedback. Initial evaluation of doppler sonar demonstrates that the sensor, when properly used, provides irreplaceably valuable inputs for vehicle control.

Finally, this thesis shows that platform-independent 3D real-time simulations are possible. The use of platform-neutral programming languages coupled with the rapidly increasing performance of personal computers has brought the ability to run complex distributed simulations anytime, anywhere. As network bandwidth continues to improve and PC performance is enhanced, platform-independent simulations will continue to get better and become more popular.

C. RECOMMENDATIONS FOR FUTURE WORK

On the technological frontier there are always things to do. Breakthroughs in technology happen at an amazing rate, with each new discovery bringing a new piece of gear or programming paradigm to light. As these developments occur it will continue to be necessary to thoroughly test and evaluate new technologies. The virtual environment is the ideal place for testing potential AUV hardware and software.

This thesis falls short in the test and evaluation of the modeling technology proposed due to the lack of in-water tests. To remedy this situation, a series of tests need to be conducted to validate both the wave model and the complex body-induced flow interaction algorithm. These additions to

the virtual environment provide exceptional insight into vehicle behavior, but these results need not be broadly accepted until all doubt is laid to rest through validating in-water tests.

Another area for future consideration involves both the execution level code and the dynamics code. The current versions of these programs use standard British units. Yet the DIS protocol requires metric units in its broadcast standard. This difference caused some inaccurate results during the prototyping stages. In some cases formulations appeared to be correct but unit differences caused erroneous results. After extensive troubleshooting all units were corrected and the results verified. For future development a single set of units (metric since DIS requires it) needs to be implemented in both the execution level code and the dynamics code.

This thesis also proposes that a doppler sonar be used as a cross-body flow indicator onboard the *Phoenix* AUV. The simulation model for the doppler sensor used in this thesis was a simple one, lacking any noise distribution. Nonetheless, simulation of such a sensor demonstrates it can provide significant control enhancements. Further work is needed in simulation enhancement. Comparisons need be made between perfect data and expected (noisy) real-world data. As the NPS AUV research group moves towards the third incarnation of the *Phoenix* AUV it will be interesting install and test the DS30 doppler sonar. An instrument of this nature will likely enable very precise control of the robot in dangerous operating environments.

Another useful extension for robot development will be the integration of a depth-sensing model coupled with real-world terrain topology (Leaver, 1998). It is also useful to move the virtual world into the domain of testing sensor and effector performance in various acoustic environments. This is a significant step forward from the generic environment testing currently performed, enabling researchers to test equipment in a virtual Monterey Bay, then test in the real bay. It will likely eliminate errors normally attributed to environmental considerations.

Other sensors to be enhanced in virtual simulation are the ST725 and ST1000 sonars. These sonars were modeled using several scan modes, employed in numerous different execution level tactics by (Davis, 1996). While the modes are accessible to all for low-level control, a simplification is required allowing for easier scan mode selection. Addition of manual steering along a true bearing during the final stages of thesis testing added a new sensor value: lateral range (and range rate) to the submarine maintaining steady course and speed. An enumeration of all sonar modes and their addition to the execution command language will be useful in future tactic development.

Animation is a vital part of any virtual environment simulation. Helping humans visualize the interactions taking place in the environment. It is one of the key reasons virtual simulations are even created. The *Phoenix* AUVs virtual environment is an irreplaceable resource. Continued use of virtual environment visualization and experimental validation will continue to provide invaluable insight.

APPENDIX A. VIRTUAL ENVIRONMENT C++ CODE

1. UUVBody.C Excerpt

```
////////////////////////////////////////////////////////////////////////
/*
 Program:           UUVBody.C

 Description:       Six degree-of-freedom underwater vehicle hydrodynamics
                    based on Healey model

 Revised:           24 February 98

 System:            Irix 5.3

 Compiler:          ANSI C++

 Compilation:       irix> make UUVBody.o
                    irix> CC UUVBody.C -lm -c -g +w

                    -c == Produce binaries only, suppressing the link phase.
                    +w == Warn about all questionable constructs.

 Author:            Don Brutzman                    brutzman@nps.navy.mil
                    Code UW/Br
                    Naval Postgraduate School       408.656.2149 work
                    Monterey CA 93943-5000          408.656.3679 fax

 EOM Revisions:     Jeff Riedel, FEB 97:  removed extra cross-body flow terms
                    Kevin Byrne, FEB 98:  high-resolution buoyancy, cross-body flow

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

                    Brutzman, Donald P., Software Reference:  A Virtual World
                    for an Autonomous Underwater Vehicle, technical report
                    NPS-CS-010-94, Naval Postgraduate School, Monterey
                    California, December 1994.  The accompanying public
                    electronic distribution of this reference includes source
                    code and executable programs.  World-Wide Web (WWW)
                    Uniform Resource Locator (URL) is
                    http://www.stl.nps.navy.mil/~auv

 Advisors:          Dr. Mike Zyda, Dr. Bob McGhee and Dr. Tony Healey

 References:        Healey, A.J. and Lienard, D., "Multivariable Sliding Mode
                    Control for Autonomous DIving and Steering of Unmanned
                    Underwater Vehicles," IEEE Journal of Oceanic Engineering,
                    vol. 18 no. 3, July 1993, pp. 327-339.

                    Yuh, J., "Modeling and Control of Underwater Robotic
                    Vehicle," IEEE Transactions on Systems, Man and Cybernetics,
                    vol. 20 no. 6, November/December 1990, pp. 1475-1483.

                    Press, William H., Teukolsky, Saul A., Vetterling,
                    William T. and Flannery, Brian P., "Numerical Recipes in C,"
                    second edition, Cambridge University Press, Cambridge
                    England, 1992.

                    Marco, David, "Autonomous Control of Underwater Vehicles
                    and Local Area Maneuvering," Ph.D. dissertation, Naval
```

```
                    Postgraduate School, Monterey California, September 1996.

                    Fossen, Thor I., _Guidance and Control of Ocean Vehicles_,
                    John Wiley and Sons, Chichester England, 1994.

                    Bacon, Daniel Keith, Jr. "Integration of a Submarine into
                    NPSNET," Master's Thesis, Naval Postgraduate School,
                    Monterey, California, September 1995.  Available via
                    http://www.npsnet.nps.navy.mil/npsnet/publications.html

 Status:            Equations of motion tested satisfactorily,
                        verification against in-water tests remains.
                    Added buoyancy and center-of-buoyancy changes at surface
                        based on Dan Bacon's thesis work.
                    Housekeeping: move utilities to math_utilities.c


 Future work:       Comments and suggestions are welcome!


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

//*******                  Excerpt Follows                          ********//

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

void UUVBody:: integrate_equations_of_motion ()
{
   int MAX_ACCELERATIONS_EXCEEDED = FALSE;

   current_uuv_time = AUV_time;

   double dt        = current_uuv_time - time_of_posture_value ();

   if (dt < 0.0)    // mission clock was reset, rezero the dynamics model
   {
       current_uuv_time   = AUV_time;
       set_time_of_posture (AUV_time);
       set_velocities    (0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
       set_accelerations (0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
       dt = 0.0;
       U  = 0.0;
       V  = 0.0;
       W  = 0.0;
       P  = 0.0;
       Q  = 0.0;
       R  = 0.0;
   }

   double rho2      = rho / 2.0;
   double L2        = L * L;
   double L3        = L * L * L;
   double L4        = L * L * L * L;
   double L5        = L * L * L * L * L;

   // note that sign is not preserved in the following squared variables
   //      in order to present consistent naming with Healey reference paper.
   //      To preserve sign, use (U * fabs (U)) etc.
   double P2        = P * P;
   double Q2        = Q * Q;
   double R2        = R * R;
// double U2        = U * U;
   double V2        = V * V;
   double W2        = W * W;
```

```
   // calculate world coordinate posture rates, use holding variables for speed

   double PHI       = phi_value   ();
   double THETA     = theta_value ();
   double PSI       = psi_value   ();

   double sinPHI    = sin (   PHI );
   double cosPHI    = cos (   PHI );
   double sinTHETA  = sin ( THETA );
   double cosTHETA  = cos ( THETA );
   double sinPSI    = sin (   PSI );
   double cosPSI    = cos (   PSI );

   // clamp inputs to max values allowed in hydrodynamics coefficients file - //
   if (MAX_RPM > 0.0)
   {
      clamp (& AUV_port_rpm, -MAX_RPM,    MAX_RPM,    "AUV_port_rpm");
      clamp (& AUV_stbd_rpm, -MAX_RPM,    MAX_RPM,    "AUV_stbd_rpm");
   }
   if (MAX_PLANE > 0.0)
   {
      clamp (& AUV_delta_planes, -radians (MAX_PLANE),  radians (MAX_PLANE),
              "AUV_delta_planes");
   }
   if (MAX_RUDDER > 0.0)
   {
      clamp (& AUV_delta_rudder, -radians (MAX_RUDDER), radians (MAX_RUDDER),
              "AUV_delta_rudder");
   }
   if (MAX_THRUSTER > 0.0)
   {
    clamp(& AUV_bow_lateral,   -MAX_THRUSTER,MAX_THRUSTER,"AUV_bow_lateral");
    clamp(& AUV_stern_lateral, -MAX_THRUSTER,MAX_THRUSTER,"AUV_stern_lateral");
    clamp(& AUV_bow_vertical,  -MAX_THRUSTER,MAX_THRUSTER,"AUV_bow_vertical");
    clamp(& AUV_stern_vertical,-MAX_THRUSTER,MAX_THRUSTER,"AUV_stern_vertical");
   }

   // finish initializations  - - - - - - - - - - - - - - - - - - - - - - - - //
   double delta_planes_stern =   AUV_delta_planes;
   double delta_planes_bow   = - AUV_delta_planes;
   double delta_rudder_stern =   AUV_delta_rudder;
   double delta_rudder_bow   = - AUV_delta_rudder;

   // Zero ordered thruster values if no thrusters present
   AUV_bow_lateral   *= THRUSTERS;
   AUV_stern_lateral *= THRUSTERS;
   AUV_bow_vertical  *= THRUSTERS;
   AUV_stern_vertical*= THRUSTERS;

// double EPSILON   = epsilon (); // no longer used in revised model

//*************************Flag for Wave Model
//Moved Variable definition for visibility throughout both models
double  sway_integral = 0.0;
double heave_integral = 0.0;
double pitch_integral = 0.0;
double   yaw_integral = 0.0;
double  roll_integral = 0.0;
double surge_integral = 0.0;
double U_cf_x;



if (WAVE_BOUYANCY_MODEL == 0)  {
```

```
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - //
// calculate neutral buoyancy using center of buoyancy near surface  - - - //

if      (AUV_z <= H / 2.0)          /* transition, calculate broach extent */
{
   if   (AUV_z >= -(H / 2.0))       /* broach region, reduce buoyancy      */

        revisedBuoyancy = Buoyancy * (AUV_z + H/2.0)/ H;

   else revisedBuoyancy = 0.0;      /* completely out of the water         */
}
else    revisedBuoyancy = Buoyancy; /* > H/2, no broach, normal submerged  */

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

        This picture shows the condition (AUV_z == H / 2.0) which is the
        transition point above which revisedBuoyancy begins to drop off.

        revisedBuoyancy will = 0 when (AUV_z <= - H / 2.0)

        Severe buoyancy changes result when AUV position magically begins
        at depths so shallow that the AUV is initially above the surface.
-z
 |
 |
 -     - H
 |
 -     - H/2
 |
 0       0   ~~~~~~~~ surface ~~~~~~~~
 |                                  (|                          |\
 +     + H/2                       ( |  ___ AUV_z = + H/2 ___  | \ 0  .. . .
 |                                 ( |                          | / 0
 +     + H                          (|__________________________|/
 |
 |
+z

depth down (positive increasing z)

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

// if boat is broaching and pitch THETA is positive, perform an approximate
// calculation of how center of buoyancy CB moves back towards stern

// nose_length is defined in UUVmodel.H and stays fixed

if      ((THETA == 0.0) || (AUV_z >= H / 2.0))
{
   revised_x_B  = x_B; // prevent divide-by-zero case and too-deep case
}
else if (THETA > 0.0)
{
   surface_length = AUV_z /     sinTHETA;
}
else if (THETA < 0.0)
{
   surface_length = AUV_z / (- sinTHETA); // roughly symmetric fore+aft
}
else
{
   cout << "Unexpected case in revised CB calculation!" << endl;
   revised_x_B  = x_B; // prevent divide by zero case
}
```

```
   if ((THETA != 0.0) && (surface_length < nose_length) && (AUV_z <= H / 2.0))
   // move x_CB aft (fwd) but only if nose (stern) broaches the surface
   {
      revised_x_B  = x_B - (nose_length - surface_length) * sinTHETA / 2.0;
   }

   if (TRACE || TRACE_EOM || (revisedBuoyancy != Buoyancy))
   {
      cout << "revisedBuoyancy = " << revisedBuoyancy << ", ";
      cout << "Weight = "          << Weight          << ", ";
      cout <<  endl;
      cout << " surface_length = "  << surface_length  << ", ";
      cout << "nose_length = "     << nose_length     << ", ";
      cout << "revised_x_B = "     << revised_x_B     << endl;
   }

////////////////////////////////////////////////////////////////////////////

   // integrate drag forces over the vehicle - - - - - - - - - - - - - - - -//
   // corresponding to cross-body flow.  Use cross-sectional slices. - - - - -//

   double dx;

   // traverse longitudinal centerline:  index through x coordinate arrays
   for (int x_index = 0; x_index < cross_sections-1; x_index ++)
   {
        dx = fabs (xx [x_index] - xx [x_index + 1]);

        U_cf_x = sqrt (  square (V + xx [x_index] * R)

                       + square (W - xx [x_index] * Q));

        if (U_cf_x > 1.0E-6) // arbitrary small non-0 minimum
        {
            sway_integral  +=   rho2 * ( C_dy * hh [x_index]

                                         * square ((V + xx [x_index] * R)) )

// removed from model                      + C_dz * bb [x_index]
//                                                * square ((W - xx [x_index] * Q)))

                                      * (V + xx [x_index] * R) * dx / U_cf_x;


            heave_integral +=   rho2 * (

// removed from model                        C_dy * hh [x_index]
//                                                * square ((V + xx [x_index] * R))

                                        + C_dz * bb [x_index]
                                               * square ((W - xx [x_index] * Q)))

                                      * (W - xx [x_index] * Q) * dx / U_cf_x;


            pitch_integral +=   rho2 * (

// removed from model                        C_dy * hh [x_index]
//                                                * square ((V + xx [x_index] * R)) )

                                        + C_dz * bb [x_index]

                                               * square ((W - xx [x_index] * Q)))

                                      * (W - xx [x_index] * Q)
```

```
                                        // ^ note sign correction

                                      * xx [x_index] * dx / U_cf_x;


              yaw_integral +=   rho2 * ( C_dy * hh [x_index]

                                             * square ((V + xx [x_index] * R)) )

// removed from model                    + C_dz * bb [x_index]
//                                           * square ((W - xx [x_index] * Q)))

                                      * (V + xx [x_index] * R)

                                      * xx [x_index] * dx / U_cf_x;
         }
   }
   if (TRACE || TRACE_EOM)
   {
      cout << "dx = " << dx << ", U_cf_x = " << U_cf_x
           << ", sway_integral  = " << sway_integral << endl;

      cout << "dx = " << dx << ", U_cf_x = " << U_cf_x
           << ", heave_integral = " << heave_integral << endl;

      cout << "dx = " << dx << ", U_cf_x = " << U_cf_x
           << ", pitch_integral = " << pitch_integral << endl;

      cout << "dx = " << dx << ", U_cf_x = " << U_cf_x
           << ", yaw_integral   = " << yaw_integral << endl;
   }

}  // end old bouyancy model
else if (WAVE_BOUYANCY_MODEL == TRUE) {

   int    in_sub_flow_field = 0;
   int    pw_flowfield_x[cross_sections];
   int    pw_flowfield_r[cross_sections];

   //required variables for piecewise calculations of wave motion effects
   double pw_AUV_x[cross_sections];
   double pw_AUV_y[cross_sections];
   double pw_AUV_z[cross_sections];
   double pw_nose_length[cross_sections];
   double pw_surface_length[cross_sections];
   double pw_dx[cross_sections];
   double pw_revised_x_B[cross_sections];
   double pushup[cross_sections];
   double pw_revisedBouyancy[cross_sections];
   double x_difference;
   double y_difference;
   double z_difference;
   double AUV_TTube_z_difference;
   double AUV_SUB_Course_difference = 0.0;
   double grid_x_difference = 0.0;
   double grid_r_difference = 0.0;
   double flow_force_direction[cross_sections];
   double K_waves = 0.4;      //This is a factor used to reduce wave effects. Otherwise
vehicle goes unstable.
   double temp_doppler_stw_u = 0.0;
   double temp_doppler_stw_v = 0.0;

   Vector3D U_waves[cross_sections];
   Vector3D  pw_UVW;                   //Holds piecewise flow velocities in AUV frmae of
reference
```

```
   Vector3D flow_force_magnitude[cross_sections];    //This holds x-dot, y-dot, z-dot in
sub ref frame

   Hmatrix flow_rotation_matrix;                //This is used to move flow vector from
the sub's (ft/sec)

   //Additions to the equations of motion
   double flow_field_sway_integral  = 0.0;
   double flow_field_surge_integral = 0.0;
   double flow_field_heave_integral = 0.0;
   double flow_field_roll_integral  = 0.0;
   double flow_field_pitch_integral = 0.0;
   double flow_field_yaw_integral   = 0.0;

   if (SUBMARINE_DOCKING == TRUE) {

     //Check to se if AUV is in the influence field of the submarine
     //This conversion uses 0.3048 meters per foot or 3.281 ft per meter
     x_difference = (AUV_x - submarine_x);
     y_difference = (AUV_y - submarine_y);
     z_difference = (AUV_z - submarine_z);

     //The order of AUV and TT is reversed to get sign correct since +z is down
     AUV_TTube_z_difference = torpedotube_z - AUV_z;

     //All box calculations are in feet, here we convert to meters and
     //then compare
     //The 15in y  calc accounts for sub diameter of 30 ft, radius = 15 ft
     if ((fabs(x_difference)) <= (flowfieldbox_length * FLOWFIELDLENGTH )  &&
         (fabs(y_difference)) <= (flowfieldbox_width  * FLOWFIELDWIDTH + 15.0 )  &&
         (fabs(z_difference)) <= (flowfieldbox_height * 40.0            )) {

           //set flag to perform piecewise calculations
           in_sub_flow_field = 1;

           //calculate difference in AUV and sub course + speed
           AUV_SUB_Course_difference = submarine_course - AUV_heading;

     } //end of if in flow field
   }   //end if SUBMARINE_DOCKING


   //Loop through body to Initialize all Arrays, perform piecewise calculations
   for (int x_index = 0; x_index <= cross_sections - 1; x_index++)
   {
      pw_dx[x_index] = fabs(xx[x_index] - xx[x_index + 1]);

      if (x_index == 0) {
         pw_nose_length[x_index] = pw_dx[x_index]/2.0;
      }
      else {
         pw_nose_length[x_index] = pw_dx[x_index]/2.0 + pw_nose_length[x_index - 1] +
pw_dx[x_index - 1]/2.0;
      }

      //Calculate pushup - the amount this sections pw_AUV_z differes from the overall
AUV_Z
      pushup[x_index] = (xx[x_index] + pw_dx[x_index]/2.0) * sinTHETA;

      //Calculate pw_AUV_z
      pw_AUV_z[x_index] = AUV_z - pushup[x_index];


      //Here we perform all calulations for piecewise flow field forces
      //if AUV is in sub torpedotube area
```

```
      if (in_sub_flow_field == TRUE) {

        //Calculate an exact x & y for each section
        pw_AUV_x[x_index] = AUV_x + (sin(90 - AUV_SUB_Course_difference) *
                                    (xx[x_index] + pw_dx[x_index]/2.0));

        pw_AUV_y[x_index] = AUV_y + (sin(AUV_SUB_Course_difference) *
                                    (xx[x_index] + pw_dx[x_index]/2.0));


        //Translate the x and y into grid coordinates based on position relative to sub
        //this takes position in feet and gives diff in ft
        grid_x_difference = ((double)submarine_x -  pw_AUV_x[x_index]);
        grid_r_difference = sqrt ((pow((double)submarine_y -  pw_AUV_y[x_index], 2)) +
                                  (pow((double)submarine_z -  pw_AUV_z[x_index], 2)));

        //Assuming each integer differnce equals one foot, this translates the difference
        //between sub and auv (x,y) into a coordinate in the grids reference. The
        //grid starts with (0,0) at the bow and (720, 0) at the stern. The center ofthe
sub
        // is actually at grid position (360, 0).
        if (grid_x_difference >= 0) {
           pw_flowfield_x[x_index] = 360 + (int)(2 * grid_x_difference);
        }
        else {
           pw_flowfield_x[x_index] = 360 + (int)(2 * grid_x_difference);
        }


        //Here 15 is subtracted to account for submarine radius (15 ft = 30 .5 ft
segments)
        pw_flowfield_r[x_index] = (int)((grid_r_difference - 15.0) * 2.0);


        //Check to make sure pw_flowfield x and y are valid
        if ((pw_flowfield_x[x_index] >=  FLOWFIELDLENGTH - 1) ||
            (pw_flowfield_x[x_index] <  0)) {

          if (TRACE) {
            //print error message
            cout << "*********************************************" << endl
                 << "pw_flowfield_x[x_index] for AUV section " << x_index
                 << " was calculated as "  << pw_flowfield_x[x_index] << endl;

            cout << "Submarine X = " << submarine_x  << " ft   Submarine_y = "
                 << submarine_y << " ft" <<endl
                 << "pw_AUV_x    = " << pw_AUV_x[x_index] << " ft   pw_AUV_y = "
                 << pw_AUV_y[x_index] << " ft"
                 << endl;
            cout << "Value reset to 360" << endl;
          }
          //Reset the values to middle of grid
          pw_flowfield_x[x_index] = 360;
        }

        if (pw_flowfield_r[x_index] >=  FLOWFIELDWIDTH - 1) {

          if (TRACE) {
            //print error message
            cout << "*********************************************" << endl
                 << "pw_flowfield_r[x_index] for AUV section " << x_index
                 << " was calculated as "  << pw_flowfield_r[x_index] << endl;

            cout << "Submarine X = " << submarine_x  << " ft   Submarine_y = "
                 << submarine_y  << " ft" <<endl
```

```
                << "pw_AUV_x    = " << pw_AUV_x[x_index]  << " ft   pw_AUV_y = "
                << pw_AUV_y[x_index] << " ft"
                << endl;
           cout << "Value reset to 1 ft from hull" << endl;
         }

         //This case is reached most when AUV hits hull Therefore to keep flow
         //force consistent I reset the flow field index to 1, or 6" from hull
         //Reset the value to next to hull
         pw_flowfield_r[x_index] = 60;
       }
       else if (pw_flowfield_r[x_index] <  0) {

         if (TRACE) {
            //print error message
            cout << "*********************************************" << endl
                 << "pw_flowfield_r[x_index] for AUV section " << x_index
                 << " was calculated as "  << pw_flowfield_r[x_index] << endl;

            cout << "Submarine X = " << submarine_x  << " ft   Submarine_y = "
                 << submarine_y  << " ft" <<endl
                 << "pw_AUV_x    = " << pw_AUV_x[x_index]  << " ft   pw_AUV_y = "
                 << pw_AUV_y[x_index] << " ft"
                 << endl;
            cout << "Value reset to 1 ft from hull" << endl;
         }
         //This case is reached when AUV hits hull Therefore to keep flow
         //force consistent I reset the flow field index to 1, or 6" from hull
         //Reset the value to next to hull
         pw_flowfield_r[x_index] = 1;
       }


        //Determine which flow grid to use based on pw_AUV_z and selected model
        if (((fabs(AUV_TTube_z_difference) <= torpedotube_height) && ( FLOW_FIELD_MODE ==
3))
                                                  || (FLOW_FIELD_MODE == 2))  {


            //the direction should always be submarine_course + flow field direction
offset
            //The flow magnitude here is converted to ft/sec by multiplying by
            /*
                 ft/sec = knots * 2000 yds/hr* 3 ft/yd  * hr/60 min * min/60 sec  = 1.667
* knots

*/

            //Now decide which level of the tube flow fields to use
            if (AUV_TTube_z_difference > 3.0) {
               //The AUV is in the above tube zone

               //Next select the appropriate speed matrix
               switch ((int) submarine_speed) {
                  case 1:
                     flow_force_direction[x_index] = submarine_course +

abovetubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].direction;
                     flow_force_magnitude[x_index].setValue

(abovetubelevel1ktgrid[pw_flowfield_x[x_index]]
```

```
[pw_flowfield_r[x_index]].x_magnitude * 1.667,
abovetubelevel1ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].y_magnitude * 1.667,
abovetubelevel1ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;
                        break;
                    case 2:
                        flow_force_direction[x_index] = submarine_course +
abovetubelevel2ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].direction;
                        flow_force_magnitude[x_index].setValue
(abovetubelevel2ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].x_magnitude * 1.667,
abovetubelevel2ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].y_magnitude * 1.667,
abovetubelevel2ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;
                        break;
                    case 3:
                        flow_force_direction[x_index] = submarine_course +
abovetubelevel3ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].direction;
                        flow_force_magnitude[x_index].setValue
(abovetubelevel3ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].x_magnitude * 1.667,
abovetubelevel3ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].y_magnitude * 1.667,
abovetubelevel3ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;
                        break;
                    default:
                        cerr << "The submarine is moving to fast for the AUV to dock with."
<< endl;
                        break;
                }
            }
```

```
            else if (AUV_TTube_z_difference > 1.0) {
              //THe AUV is at the upper tube edge zone


               //Next select the appropriate speed matrix
               switch ((int) submarine_speed) {
                  case 1:
                     flow_force_direction[x_index] = submarine_course +

uppertubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].direction;
                     flow_force_magnitude[x_index].setValue

(uppertubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].x_magnitude * 1.667,

uppertubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].y_magnitude * 1.667,

uppertubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;

                     break;

                  case 2:

                     flow_force_direction[x_index] = submarine_course +

uppertubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].direction;
                     flow_force_magnitude[x_index].setValue

(uppertubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].x_magnitude * 1.667,

uppertubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].y_magnitude * 1.667,

uppertubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;

                     break;

                  case 3:

                     flow_force_direction[x_index] = submarine_course +

uppertubelevel3ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].direction;
                     flow_force_magnitude[x_index].setValue

(uppertubelevel3ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].x_magnitude * 1.667,

uppertubelevel3ktgrid[pw_flowfield_x[x_index]]
```

```
[pw_flowfield_r[x_index]].y_magnitude * 1.667,

uppertubelevel3ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;

                        break;

                    default:
                        cerr << "The submarine is moving to fast for the AUV to dock with."
<< endl;
                        break;
                }

            }

            else if (AUV_TTube_z_difference > -1.0) {
                //The AUV is in the center of the tube zone


                //Next select the appropriate speed matrix
                switch ((int) submarine_speed) {
                    case 1:
                        flow_force_direction[x_index] = submarine_course +

centertubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].direction;
                        flow_force_magnitude[x_index].setValue

(centertubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].x_magnitude * 1.667,

centertubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].y_magnitude * 1.667,

centertubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;

                        break;

                    case 2:

                        flow_force_direction[x_index] = submarine_course +

centertubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].direction;
                        flow_force_magnitude[x_index].setValue

(centertubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].x_magnitude * 1.667,

centertubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].y_magnitude * 1.667,

centertubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;
```

```
                        break;

                     case 3:

                        flow_force_direction[x_index] = submarine_course +

centertubelevel3ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].direction;
                        flow_force_magnitude[x_index].setValue

(centertubelevel3ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].x_magnitude * 1.667,

centertubelevel3ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].y_magnitude * 1.667,

centertubelevel3ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;

                        break;

                     default:
                        cerr << "The submarine is moving to fast for the AUV to dock with."
<< endl;
                        break;
                  }


               }

               else if (AUV_TTube_z_difference > -3.0) {
                  //The AUV is in the lower tube edge zone


                  //Next select the appropriate speed matrix
                  switch ((int) submarine_speed) {
                     case 1:
                        flow_force_direction[x_index] = submarine_course +

lowertubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].direction;
                        flow_force_magnitude[x_index].setValue

(lowertubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].x_magnitude * 1.667,

lowertubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].y_magnitude * 1.667,

lowertubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;

                        break;

                     case 2:

                        flow_force_direction[x_index] = submarine_course +
```

```
lowertubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].direction;
                        flow_force_magnitude[x_index].setValue

(lowertubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].x_magnitude * 1.667,

lowertubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].y_magnitude * 1.667,

lowertubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;

                        break;

                    case 3:

                        flow_force_direction[x_index] = submarine_course +

lowertubelevel3ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].direction;
                        flow_force_magnitude[x_index].setValue

(lowertubelevel3ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].x_magnitude * 1.667,

lowertubelevel3ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].y_magnitude * 1.667,

lowertubelevel3ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;

                        break;

                    default:
                        cerr << "The submarine is moving to fast for the AUV to dock with."
<< endl;
                        break;
                }


            }

            else {
              //The AUV is in the below tube zone


               //Next select the appropriate speed matrix
               switch ((int) submarine_speed) {
                  case 1:
                     flow_force_direction[x_index] = submarine_course +

belowtubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].direction;
                        flow_force_magnitude[x_index].setValue
```

```
(belowtubelevel1ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].x_magnitude * 1.667,
belowtubelevel1ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].y_magnitude * 1.667,
belowtubelevel1ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;
                        break;
                    case 2:
                        flow_force_direction[x_index] = submarine_course +
belowtubelevel2ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].direction;
                        flow_force_magnitude[x_index].setValue
(belowtubelevel2ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].x_magnitude * 1.667,
belowtubelevel2ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].y_magnitude * 1.667,
belowtubelevel2ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;
                        break;
                    case 3:
                        flow_force_direction[x_index] = submarine_course +
belowtubelevel3ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].direction;
                        flow_force_magnitude[x_index].setValue
(belowtubelevel3ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].x_magnitude * 1.667,
belowtubelevel3ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].y_magnitude * 1.667,
belowtubelevel3ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;
                        break;
                    default:
                        cerr << "The submarine is moving to fast for the AUV to dock with."
<< endl;
                        break;
```

```
            }

        } //End of If that decides tube level

    }   //End of if which decides flat/tube profile

    //This is the case of being in a flat plate field region
    else  {


        //Next select the appropriate speed matrix
        switch ((int) submarine_speed) {
           case 1:
              flow_force_direction[x_index] = submarine_course +

nontubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].direction;
              flow_force_magnitude[x_index].setValue

(nontubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].x_magnitude * 1.667,

nontubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].y_magnitude * 1.667,

nontubelevel1ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;

              break;

           case 2:

              flow_force_direction[x_index] = submarine_course +

nontubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].direction;
              flow_force_magnitude[x_index].setValue

(nontubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].x_magnitude * 1.667,

nontubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].y_magnitude * 1.667,

nontubelevel2ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;

              break;

           case 3:

              flow_force_direction[x_index] = submarine_course +

nontubelevel3ktgrid[pw_flowfield_x[x_index]]

[pw_flowfield_r[x_index]].direction;
```

```
                        flow_force_magnitude[x_index].setValue
(nontubelevel3ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].x_magnitude * 1.667,
nontubelevel3ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].y_magnitude * 1.667,
nontubelevel3ktgrid[pw_flowfield_x[x_index]]
[pw_flowfield_r[x_index]].z_magnitude * 1.667) ;
                        break;
                     default:
                        cerr << "The submarine is moving to fast for the AUV to dock with."
<< endl;
                        break;
                }
         }  //end of else for  flat plate region


      }   //end of in flow field calculations

      //Check Bouyancy adjustment
      if      (pw_AUV_z[x_index] <= H / 2.0)          // transition, calculate broach
extent
      {
         if   (pw_AUV_z[x_index] >= -(H / 2.0))       // broach region, reduce buoyancy
         {
              pw_revisedBouyancy[x_index] = (Buoyancy/cross_sections) * (pw_AUV_z[x_index]
+ H/2.0)/H;
         }
         else
         {
             pw_revisedBouyancy[x_index] = 0.0;      // completely out of the water
         }
      }

      else
      {
          pw_revisedBouyancy[x_index] = (Buoyancy/cross_sections);  // > H/2, no broach,
normal submerged
      }

      //Global wave force effects in ft per second
      U_waves[x_index].setValue ( K_waves *
                                  (SeaState[SEASTATE].H_s * SeaState[SEASTATE].freq1*
                                       (cos ( SeaState[SEASTATE].freq1*AUV_time +
SeaState[SEASTATE].wavelength*pw_nose_length[x_index] ))) // +
                                   //SeaState[SEASTATE].H_s * SeaState[SEASTATE].freq2*
                                   //    (cos ( SeaState[SEASTATE].freq2*AUV_time +
                                   //
SeaState[SEASTATE].wavelength*pw_nose_length[x_index] )) +
                                   //SeaState[SEASTATE].H_s * SeaState[SEASTATE].freq3*
                                   //    (cos ( SeaState[SEASTATE].freq3*AUV_time  +
                                   //
SeaState[SEASTATE].wavelength*pw_nose_length[x_index] )))
                                   *(cos (heading_wave_1 - AUV_psi))
                                     ,
```

```
                                    0.0,

                                    K_waves *
                                    (SeaState[SEASTATE].H_s * SeaState[SEASTATE].freq1*
                                         (cos ( SeaState[SEASTATE].freq1*AUV_time + 90.0 +

SeaState[SEASTATE].wavelength*pw_nose_length[x_index] )))  // +
                                    //SeaState[SEASTATE].H_s * SeaState[SEASTATE].freq2*
                                    //     (cos ( SeaState[SEASTATE].freq2*AUV_time + 90.0 +
                                    //
SeaState[SEASTATE].wavelength*pw_nose_length[x_index] )) +
                                    //SeaState[SEASTATE].H_s * SeaState[SEASTATE].freq3*
                                    //     (cos ( SeaState[SEASTATE].freq3*AUV_time + 90.0 +
                                    //
SeaState[SEASTATE].wavelength*pw_nose_length[x_index])))
                                     *(cos (heading_wave_1 - AUV_psi))

                                    );

      //At depth > 20  we reduce the wave motion effect linearly, deeper than 100' wave
effect is negligible
      if (AUV_z > 20.0)
      {

        U_waves[x_index].setValue ( U_waves[x_index][1] * ((100.0-AUV_z)/100.0),
                                    0.0,
                                    U_waves[x_index][3] * ((100.0-AUV_z)/100.0) );

      } else if (AUV_z > 100) {

         U_waves[x_index].setValue ( 0.0, 0.0, 0.0);

      }

      //Check for revised_x_B adjustment
      if      ((THETA == 0.0) || (pw_AUV_z[x_index] >= H / 2.0))
      {
         pw_revised_x_B[x_index]  = xx[x_index] + (pw_dx[x_index]/2.0); // prevent
divide-by-zero case and too-deep case
      }
      else if (THETA > 0.0)
      {
         pw_surface_length[x_index] = pw_AUV_z[x_index] / sinTHETA;

      }
      else if (THETA < 0.0)
      {
         pw_surface_length[x_index] = pw_AUV_z[x_index] / (- sinTHETA);

      }
      else
      {
         cout << "Unexpected case in revised CB calculation!" << endl;
         pw_revised_x_B[x_index]  = xx[x_index] + (pw_dx[x_index]/2.0); // prevent divide
by zero case
      }

      if ((THETA != 0.0) && (pw_surface_length[x_index] < pw_nose_length[x_index]) &&
(pw_AUV_z[x_index] <= H / 2.0))
      // move x_CB aft (fwd) but only if nose (stern) broaches the surface
      {
         pw_revised_x_B[x_index]  = (xx[x_index] + (pw_dx[x_index]/2.0)) -
(pw_nose_length[x_index] - pw_surface_length[x_index]) * sinTHETA / 2.0;
         //cout << "pw_revised_x_B in case one(nose out) = " << pw_revised_x_B[x_index]<<
endl;
```

```
      }
      else {
         pw_revised_x_B[x_index] =  (xx[x_index] + (pw_dx[x_index]/2.0));
         //cout << "pw_revised_x_B in case 2= " << pw_revised_x_B[x_index]<< endl;
      }

     if (TRACE || TRACE_EOM)
     {
         cout << "AUV_Z = " << AUV_z   << endl;

         cout << x_index << "  pw_dx = " << pw_dx[x_index] << endl;

         cout << "xx = " << xx[x_index] << endl;

         cout << "sinTHETA = " << sinTHETA << endl;

         cout << "pushup = " << pushup[x_index]<< endl;

         cout << "pw_AUV_z = " << pw_AUV_z[x_index]<< endl;

         cout << "pw_nose_length = " << pw_nose_length[x_index]<< endl;

         cout << "pw_surface_length = " << pw_surface_length[x_index]<< endl;

         cout << "pw_revisedBouyancy = " << pw_revisedBouyancy[x_index]<< endl;

         cout << "pw_revised_x_B = " << pw_revised_x_B[x_index]<< endl;
      }

    } //end for loop

    //Loop to sum up piecewise bouyancy and x_b effects
    revisedBuoyancy = 0.0;
    revised_x_B     = 0.0;

    for (int x1_index = 0; x1_index <= cross_sections - 1; x1_index++)  {

        revisedBuoyancy = revisedBuoyancy + pw_revisedBouyancy[x1_index];

        revised_x_B = revised_x_B + ((xx[x1_index] + pw_dx[x1_index]/2.0) -
(pw_revised_x_B[x1_index]));

    }  //end for loop

      revised_x_B = x_B - revised_x_B;

      if (TRACE || TRACE_EOM)
      {
         cout << "revisedBuoyancy = " <<  revisedBuoyancy << ", ";
         cout << "Weight = "          <<  Weight          << ", ";
         cout <<  endl;
         cout << " surface_length = "  << surface_length  << ", ";
         cout << "nose_length = "     <<  nose_length     << ", ";
         cout << "revised_x_B = "     <<  revised_x_B     << endl;
      }

////////////////////////////////////////////////////////////////////////////

   // integrate drag forces over the vehicle - - - - - - - - - - - - - - - -//
   // corresponding to cross-body flow.  Use cross-sectional slices. - - - - -//

   //This section of code creates the rotation matrix which will be used later to
   //transform flow filed components from the sub's reference frame to the AUV's
   flow_rotation_matrix.set_identity();
   flow_rotation_matrix.rotate  (submarine_roll - AUV_phi,
```

```
                                    submarine_pitch - AUV_theta,
                                    submarine_course - AUV_psi);

   //This starts the summation of cross body drag forces. The following if statement
   //allows for 2 different modes of cross body calculations, one for  a circular hull
   //and one for a square hull. The type of Hull is required to be defined in UUVmodel.H

   if (SQUARE_HULL == TRUE) {

      // traverse longitudinal centerline:  index through x coordinate arrays
      for (int x2_index = 0; x2_index <= cross_sections - 1; x2_index++)  {

              //Calculate the effects of sub flow field
              if (in_sub_flow_field == 1) {

                //here flow forces are due to flow field + wave motion
                flow_force_magnitude[x2_index] = flow_force_magnitude[x2_index] +
U_waves[x2_index];

              }
              else {

                 //here flow forces are due to wave motion only
                 flow_force_magnitude[x2_index] =  U_waves[x2_index];

              }


              //This gets U, V, W from (x-dot, y-dot, z-dot)*rotation matrix transpose
              pw_UVW.setValue( flow_rotation_matrix * flow_force_magnitude[x2_index]);


              if (x2_index == 1) {
                  temp_doppler_stw_u = pw_UVW[1];
                  temp_doppler_stw_v = pw_UVW[2];
              }


              //-------------------------------------------
              // these integrals are for wave and flowinduced drag forces

                 flow_field_sway_integral  +=  rho2 * ( C_dy * hh [x2_index]

                                                    * pw_UVW[2] * fabs(pw_UVW[2]))

                                                    * pw_dx[x2_index];


                 flow_field_surge_integral = 0.0;

                 flow_field_heave_integral +=   rho2 * ( C_dz * bb [x2_index]

                                                    * pw_UVW[3] * fabs(pw_UVW[3]))

                                                    * pw_dx[x2_index];


                 flow_field_roll_integral =  0.0;

                 flow_field_pitch_integral +=   rho2 * ( C_dz * bb [x2_index]

                                                    * pw_UVW[3] * fabs(pw_UVW[3]))
```

```
                                                * xx [x2_index] * pw_dx[x2_index];


            flow_field_yaw_integral   +=   rho2 * ( C_dy * hh [x2_index]
                                                * pw_UVW[2] * fabs(pw_UVW[2]))
                                                * xx [x2_index] * pw_dx[x2_index];
         //-------------------------------------------
         // these integrals are for rigidbody velocity drag forces
         sway_integral  +=   rho2 * ( C_dy * hh [x2_index]
                                    * square ((V + xx [x2_index] * R  )) )
                                    * pw_dx[x2_index];
         heave_integral +=   rho2 * ( C_dz * bb [x2_index]
                                    * square ((W - xx [x2_index] * Q)))
                                    * pw_dx[x2_index];
         pitch_integral +=   rho2 * ( C_dz * bb [x2_index]
                                    * square ((W - xx [x2_index] * Q)))
                                    * xx [x2_index] * pw_dx[x2_index];

         yaw_integral   +=   rho2 * ( C_dy * hh [x2_index]
                                    * square ((V  + xx [x2_index] * R )) )
                                    * xx [x2_index] * pw_dx[x2_index];
         roll_integral  +=  0.0;



      if (TRACE || TRACE_EOM)
      {
         cout << "dx = " << pw_dx[x2_index] << ", U_cf_x = " << U_cf_x
         << ", sway_integral  = " << sway_integral << endl;

         cout << "dx = " << pw_dx[x2_index] << ", U_cf_x = " << U_cf_x
         << ", heave_integral = " << heave_integral << endl;

         cout << "dx = " << pw_dx[x2_index] << ", U_cf_x = " << U_cf_x
         << ", pitch_integral = " << pitch_integral << endl;

         cout << "dx = " << pw_dx[x2_index] << ", U_cf_x = " << U_cf_x
         << ", yaw_integral   = " << yaw_integral << endl;
      }

   } //end for loop

} //End of the square hull case

//This starts the round hull case of cross body drag
```

```
   else {

      // traverse longitudinal centerline:  index through x coordinate arrays
      for (int x2_index = 0; x2_index <= cross_sections - 1; x2_index++)  {
          U_cf_x = sqrt (  square (V + xx [x2_index] * R)

                         + square (W - xx [x2_index] * Q));

          if (U_cf_x > 1.0E-6) // arbitrary small non-0 minimum
          {



              //Calculate the effects of sub flow field
              if (in_sub_flow_field == 1) {

                //here flow forces are due to flow field + wave motion
                flow_force_magnitude[x2_index] = flow_force_magnitude[x2_index] +
U_waves[x2_index];

              }
              else {

                 //here flow forces are due to wave motion only
                 flow_force_magnitude[x2_index] =  U_waves[x2_index];

              }


              //This gets U, V, W from (x-dot, y-dot, z-dot)*rotation matrix transpose
              pw_UVW.setValue( flow_rotation_matrix * flow_force_magnitude[x2_index]);


              if (x2_index == 1) {
                  temp_doppler_stw_u = pw_UVW[1];
                  temp_doppler_stw_v = pw_UVW[2];
              }


              //-------------------------------------------
              // these integrals are for wave and flowinduced drag forces

                 flow_field_sway_integral  +=   rho2 * ( C_dy * hh [x2_index]

                                                      * pw_UVW[2] * fabs(pw_UVW[2]))

                                                      * pw_dx[x2_index];


                 flow_field_surge_integral = 0.0;


                 flow_field_heave_integral +=   rho2 * ( C_dz * bb [x2_index]

                                                      * pw_UVW[3] * fabs(pw_UVW[3]))

                                                      * pw_dx[x2_index];



                 flow_field_roll_integral =  0.0;


                 flow_field_pitch_integral +=   rho2 * ( C_dz * bb [x2_index]
```

```
                                                    * pw_UVW[3] * fabs(pw_UVW[3]))
                                                    * xx [x2_index] * pw_dx[x2_index];


                flow_field_yaw_integral   +=   rho2 * ( C_dy * hh [x2_index]
                                                    * pw_UVW[2] * fabs(pw_UVW[2]))
                                                    * xx [x2_index] * pw_dx[x2_index];
            //-------------------------------------------
            // these integrals are for rigidbody velocity drag forces
            sway_integral  +=   rho2 * ( C_dy * hh [x2_index]
                                       * square ((V + xx [x2_index] * R  )) )
                                       * (V + xx [x2_index] * R) * pw_dx[x2_index] /
U_cf_x;
            heave_integral +=   rho2 * ( C_dz * bb [x2_index]
                                       * square ((W - xx [x2_index] * Q)))
                                       * (W - xx [x2_index] * Q) * pw_dx[x2_index] /
U_cf_x;

            pitch_integral +=   rho2 * ( C_dz * bb [x2_index]
                                       * square ((W - xx [x2_index] * Q)))
                                       * (W - xx [x2_index] * Q)
                                       * xx [x2_index] * pw_dx[x2_index] / U_cf_x;

            yaw_integral   +=   rho2 * ( C_dy * hh [x2_index]
                                       * square ((V + xx [x2_index] * R )) )
                                       * (V + xx [x2_index] * R)
                                       * xx [x2_index] * pw_dx[x2_index] / U_cf_x;

        }   //end of if (U_cf_x > 1.0E-6)

        if (TRACE || TRACE_EOM)
        {
           cout << "dx = " << pw_dx[x2_index] << ", U_cf_x = " << U_cf_x
           << ", sway_integral  = " << sway_integral << endl;

           cout << "dx = " << pw_dx[x2_index] << ", U_cf_x = " << U_cf_x
           << ", heave_integral = " << heave_integral << endl;

           cout << "dx = " << pw_dx[x2_index] << ", U_cf_x = " << U_cf_x
           << ", pitch_integral = " << pitch_integral << endl;

           cout << "dx = " << pw_dx[x2_index] << ", U_cf_x = " << U_cf_x
           << ", yaw_integral   = " << yaw_integral << endl;
        }
```

```
      } //end for loop

   }  //End of the round hull case of cross body drag

   //Add effects of flow field integral's to eom integrals
   sway_integral  +=     flow_field_sway_integral;
   heave_integral +=     flow_field_heave_integral;
   pitch_integral +=     flow_field_pitch_integral;
   yaw_integral   +=     flow_field_yaw_integral;
   roll_integral  +=     flow_field_roll_integral;
   surge_integral +=    flow_field_surge_integral; // unused

   //set doppler velocities for speed through water in ft/sec
   doppler_stw_u =  temp_doppler_stw_u;
   doppler_stw_v =  temp_doppler_stw_v;

}  //end new bouyancy model


////////////////////////////////////////////////////////////////////////////

// debug section.  selectively set sway/heave/pitch/yaw integrals to zero to
//                 isolate problems.  also see zeroing of rhs values.

//  sway_integral = 0.0;
//  heave_integral = 0.0;
//  pitch_integral = 0.0;
//   yaw_integral = 0.0;

////////////////////////////////////////////////////////////////////////////

   // reduce efficiency if propellers operating astern - - - - - - - - - - -//
   double port_propeller_efficiency, stbd_propeller_efficiency;

   if (AUV_port_rpm >= 0.0) port_propeller_efficiency = 1.0;
   else                     port_propeller_efficiency = X_astern_efficiency;

   if (AUV_stbd_rpm >= 0.0) stbd_propeller_efficiency = 1.0;
   else                     stbd_propeller_efficiency = X_astern_efficiency;

////////////////////////////////////////////////////////////////////////////
// calculate Equations of Motion right-hand sides                         //
////////////////////////////////////////////////////////////////////////////

   rhs [SURGE] = // Surge Motion Equation right hand side ----------------//

         m * ((V * R) - (W * Q) + x_G * (Q2 + R2) - y_G * P*Q - z_G * P*R)

       + rho2 * L4 * (  X_pp * P2  + X_qq * Q2
                      + X_rr * R2  + X_pr * P*R)

       + rho2 * L3 * (  X_wq * W*Q + X_vp * V*P + X_vr * V*R

                      + U*Q * (  X_uq_delta_bow     * delta_planes_bow
                               + X_uq_delta_stern   * delta_planes_stern)

                      + U*R * (  X_ur_delta_rudder  * delta_rudder_bow
                               + X_ur_delta_rudder  * delta_rudder_stern)
                     )

       + rho2 * L2 * (  X_vv * V2  + X_ww * W2

                      + U*V * (  X_uv_delta_rudder  * delta_rudder_stern)

                      + U*W * (  X_uw_delta_bow     * delta_planes_bow
```

```
                                + X_uw_delta_stern   * delta_planes_stern)

                        + U * fabs (U)  * (  X_uu_delta_b_delta_b
                                             * delta_planes_bow
                                             * delta_planes_bow

                                           + X_uu_delta_s_delta_s
                                             * delta_planes_stern
                                             * delta_planes_stern

                                           + X_uu_delta_r_delta_r
                                             * delta_rudder_bow
                                             * delta_rudder_bow

                                           + X_uu_delta_r_delta_r
                                             * delta_rudder_stern
                                             * delta_rudder_stern)
                      )


       - (Weight - revisedBuoyancy) * sinTHETA

       // EPSILON terms have been removed due to revised equations of motion

       // + rho2 * L3 *   X_qdsn  * U*Q * delta_planes_stern * EPSILON
       // + rho2 * L2 * EPSILON * ( X_wdsn  * U*W * delta_planes_stern
       //
       //                          + X_dsdsn * U2       * delta_planes_stern
       //                                               * delta_planes_stern)

       // X_propulsion surge force (derived using expressions in Healey paper)
       // note that SPEED_PER_RPM is associated with work of two propellors

       + rho2 * L2 * C_d0 *  square (SPEED_PER_RPM)

                   *  0.5 * (  AUV_port_rpm * fabs (AUV_port_rpm)
                                            * port_propeller_efficiency

                             + AUV_stbd_rpm * fabs (AUV_stbd_rpm)
                                            * stbd_propeller_efficiency)

       // X_resistance surge drag (derived using expressions in Healey paper)

       - rho2 * L2 * C_d0 * U * fabs (U);

////////////////////////////////////////////////////////////////////////////////

if (TRACE || TRACE_EOM || (rhs [SURGE] >= MAX_SURGE))   // Surge TRACE
{
cout << "* surge term1=" <<  m * ((V * R) - (W * Q)
        + x_G * (Q2 + R2) - y_G * P*Q - z_G * P*R)<< endl;

cout << "term2=" <<    + rho2 * L4 * (  X_pp * P2  + X_qq * Q2
                       + X_rr * R2  + X_pr * P*R)
     << endl;

cout << "term3=" <<    + rho2 * L3 * (  X_wq * W*Q + X_vp * V*P + X_vr * V*R

                       + U*Q * (  X_uq_delta_bow     * delta_planes_bow
                                + X_uq_delta_stern   * delta_planes_stern)

                       + U*R * (  X_ur_delta_rudder  * delta_rudder_stern
                                + X_ur_delta_rudder  * delta_rudder_bow)
                      )
     << endl;
```

```
cout << "term4=" <<    + rho2 * L2 * (  X_vv * V2  + X_ww * W2

                      + U*V * (  X_uv_delta_rudder  * delta_rudder_stern)

                      + U*W * (  X_uw_delta_bow     * delta_planes_bow
                               + X_uw_delta_stern   * delta_planes_stern)

                      + U * fabs (U)  * (  X_uu_delta_b_delta_b
                                           * delta_planes_bow
                                           * delta_planes_bow

                                         + X_uu_delta_s_delta_s
                                           * delta_planes_stern
                                           * delta_planes_stern

                                         + X_uu_delta_r_delta_r
                                           * delta_rudder_bow
                                           * delta_rudder_bow

                                         + X_uu_delta_r_delta_r
                                           * delta_rudder_stern
                                           * delta_rudder_stern)
                    )
     << endl;

cout << "term5=" << - (Weight - revisedBuoyancy) * sinTHETA
     << endl;

cout << "term6,term7=" <<   "EPSILON terms, no longer used"
     << endl;

// cout << "term6=" <<  rho2 * L3 *   X_qdsn  * U*Q * delta_planes_stern
//      * EPSILON << endl;
//
// cout << "term7=" <<  rho2 * L2 * EPSILON * ( X_wdsn  * U*W
//                                                * delta_planes_stern
//                                  + X_dsdsn * U2        * delta_planes_stern
//                                                * delta_planes_stern)
//      << endl;

cout << "term8=" <<  + rho2 * L2 * C_d0 *  square (SPEED_PER_RPM)

                  *  0.5 * (  AUV_port_rpm * fabs (AUV_port_rpm)
                                           * port_propeller_efficiency

                           + AUV_stbd_rpm * fabs (AUV_stbd_rpm)
                                           * stbd_propeller_efficiency)
     << endl;

cout << "term9=" << - rho2 * L2 * C_d0 * U * fabs (U)
     << endl;
}

////////////////////////////////////////////////////////////////////////////

    rhs [SWAY ] = // Sway  Motion Equation right hand side ------------------//

         m * (- (U * R) + (W * P) - x_G * (P * Q)

                                  + y_G * (P2 + R2)

                                  - z_G * (Q  * R))

       + rho2 * L4 * (  Y_pq    * P*Q    + Y_qr   * Q*R)
```

```
        + rho2 * L3 * (  Y_up    * U*P    + Y_ur   * U*R
                       + Y_vq    * V*Q    + Y_wp   * W*P    +  Y_wr * W*R)
        + rho2 * L2 * (  Y_uv    * U*V    + Y_vw   * V*W
                       + U*fabs(U) * Y_uu_delta_rb * delta_rudder_bow
                       + U*fabs(U) * Y_uu_delta_rs * delta_rudder_stern)
        - sway_integral
        + (Weight - revisedBuoyancy) * cosTHETA * sinPHI
        -  (2.0 / (24.0 * 24.0))  // each thruster 2.0 lb per 24V signal squared
           * (  AUV_bow_lateral   * fabs (AUV_bow_lateral)
              + AUV_stern_lateral * fabs (AUV_stern_lateral));

////////////////////////////////////////////////////////////////////////////

if (TRACE || TRACE_EOM || (rhs [SWAY] >= MAX_SWAY))   // Sway TRACE
{
cout << "* sway term1=" <<  m * (- (U * R) + (W * P)
                                - x_G * (P * Q)
                                + y_G * (P2 + R2)
                                - z_G * (Q  * R))
     << endl;

cout << "term2=" <<   + rho2 * L4 * (  Y_pq    * P*Q    + Y_qr   * Q*R)
     << endl;

cout << "term3=" <<   + rho2 * L3 * (  Y_up    * U*P    +  Y_ur  * U*R
                      + Y_vq    * V*Q    + Y_wp   * W*P    +  Y_wr * W*R)
     << endl;

cout << "term4=" <<   + rho2 * L2 * (  Y_uv     * U*V    + Y_vw   * V*W
                      + U*fabs(U) * Y_uu_delta_rb * delta_rudder_bow
                      + U*fabs(U) * Y_uu_delta_rs * delta_rudder_stern)
     << endl;

cout << "term5=" <<   - sway_integral << " sway_integral"
     << endl;

cout << "term6=" <<   + (Weight - revisedBuoyancy) * cosTHETA * sinPHI
     << endl;

cout << "term7=" <<   -  (2.0 / (24.0 * 24.0))
                     // each thruster 2.0 lb per 24V signal squared
                  * (  AUV_bow_lateral   * fabs (AUV_bow_lateral)
                     + AUV_stern_lateral * fabs (AUV_stern_lateral))
     << endl;
}

////////////////////////////////////////////////////////////////////////////

   rhs [HEAVE] = // Heave Motion Equation right hand side --------------------//
```

```
        m * (  (U * Q) - (V * P) - x_G * (P * R) - y_G * (Q * R)
                                                 + z_G * (P2 + Q2))
      + rho2 * L4 * (  Z_pp    * P2     + Z_pr * P*R    + Z_rr * R2)
      + rho2 * L3 * (  Z_uq    * U*Q    + Z_vp * V*P    + Z_vr * V*R)
      + rho2 * L2 * (  Z_uw    * U*W    + Z_vv * V2
                     + ( U*fabs(U) * Z_uu_delta_b * delta_planes_bow  )
                     + ( U*fabs(U) * Z_uu_delta_s * delta_planes_stern))
      - heave_integral
      + (Weight - revisedBuoyancy) * cosTHETA * cosPHI
      // EPSILON terms have been removed due to revised equations of motion
      // + rho2 * L3 *    Z_qn  * U*Q * EPSILON
      // + rho2 * L2 * (  Z_wn  * U*W
      //                + Z_dsn * U*fabs(U) * delta_planes_stern) * EPSILON
      +  (2.0 / (24.0 * 24.0))  // each thruster 2.0 lb per 24V signal squared
         * ( AUV_bow_vertical   * fabs (AUV_bow_vertical)  +
             AUV_stern_vertical * fabs (AUV_stern_vertical));

////////////////////////////////////////////////////////////////////////////////

if (TRACE || TRACE_EOM || (rhs [HEAVE] >= MAX_HEAVE))   // Heave TRACE
{
cout << "* heave term1=" <<  m * (  (U * Q) - (V * P) - x_G * (P * R)
                                                 - y_G * (Q * R)
                                                 + z_G * (P2 + Q2))
     << endl;

cout << "term2=" <<   + rho2 * L4 * (  Z_pp    * P2     + Z_pr    * P*R
        + Z_rr * R2) << endl;

cout << "term3=" <<   + rho2 * L3 * (  Z_uq    * U*Q    + Z_vp * V*P
        + Z_vr * V*R) << endl;

cout << "term4=" <<   + rho2 * L2 * (  Z_uw    * U*W    + Z_vv  * V2
                      + ( U*fabs(U) * Z_uu_delta_b * delta_planes_bow  )
                      + ( U*fabs(U) * Z_uu_delta_s * delta_planes_stern))
     << endl;

cout << "term5=" <<   - heave_integral << " heave_integral"
     << endl;

cout << "term6=" <<   + (Weight - revisedBuoyancy) * cosTHETA * cosPHI
     << endl;

cout << "term7, term8=" <<   "no longer used"
     << endl;

cout << "term9=" <<   +  (2.0 / (24.0 * 24.0))
         // each thruster 2.0 lb per 24V signal squared
               * ( AUV_bow_vertical   * fabs (AUV_bow_vertical)  +
```

```
                    AUV_stern_vertical * fabs (AUV_stern_vertical) )
     << endl;
}

//////////////////////////////////////////////////////////////////////////////

   rhs [ROLL ] = // Roll  Motion Equation right hand side --------------------//

       - (I_z - I_y) * Q*R - I_xy * P*R + I_yz * (Q2 - R2) + I_xz * P*Q

       - m * ( y_G * ( -U*Q + V*P) - z_G  * ( U*R - W*P))

       + rho2 * L5  * (  K_pq * P*Q  + K_qr * Q*R

                       + K_pp * P * fabs(P)
                       + K_p  * P           )  // hovering roll drag

       + rho2 * L4  * ( K_up * fabs(U)*P  + K_ur * U*R  + K_vq * V*Q

                      + K_wp * W*P  + K_wr * W*R)

       + rho2 * L3  * (  K_uv * U*V  + K_vw * V*W

                       - U*fabs(U) * 0.5 * (  K_uu_planes * delta_planes_bow
                                            + K_uu_planes * delta_planes_stern)

                       - U*fabs(U) * 0.5 * (  K_uu_rudder * delta_rudder_bow
                                            + K_uu_rudder * delta_rudder_stern))

       //Added roll integral for square hull model
       + roll_integral

       // expected: opposed plane directions ^ cause negation & cancellation

       + (y_G * Weight - y_B * revisedBuoyancy) * cosTHETA * cosPHI

       - (z_G * Weight - z_B * revisedBuoyancy) * cosTHETA * sinPHI;

       // EPSILON terms have been removed due to revised equations of motion
       // + rho2 * L4 * K_pn * U*P * EPSILON

       // + rho2 * L3 * U*fabs(U)  * K_prop; // oversimplified, in error

//////////////////////////////////////////////////////////////////////////////

if (TRACE || TRACE_EOM || (rhs [ROLL] >= MAX_ROLL))   // Roll TRACE
{
cout << "* roll term1=" << - (I_z - I_y) * Q*R - I_xy * P*R + I_yz * (Q2 - R2)
                       + I_xz * P*Q
     << endl;

cout << "term2=" << - m * ( y_G * ( -U*Q + V*P) - z_G  * ( U*R - W*P))
     << endl;

cout << "term3=" << + rho2 * L5  * (  K_pq * P*Q  + K_qr * Q*R

                       + K_pp * P * fabs(P)
                       + K_p  * P           )  // hovering roll drag
     << endl;

cout << "term4=" << + rho2 * L4  * ( K_up  * fabs(U)*P  + K_ur * U*R

                       + K_vq * V*Q + K_wp * W*P  + K_wr * W*R)
     << endl;
```

```
cout << "term5=" << + rho2 * L3  * (  K_uv * U*V  + K_vw * V*W

                      - U*fabs(U) * 0.5 * (  K_uu_planes * delta_planes_bow
                                           + K_uu_planes * delta_planes_stern)

                      - U*fabs(U) * 0.5 * (  K_uu_rudder * delta_rudder_bow
                                           + K_uu_rudder * delta_rudder_stern))
       // expected: opposed plane directions ^ cause negation & cancellation
     << endl;

cout << "term6=" << + (y_G * Weight - y_B * revisedBuoyancy) * cosTHETA * cosPHI
     << endl;

cout << "term7=" << - (z_G * Weight - z_B * revisedBuoyancy) * cosTHETA * sinPHI
     << endl;

cout << "term8,term9=" <<   "EPSILON terms, no longer used"
     << endl;

// cout << "term8=" << + rho2 * L4 * K_pn * U*P * EPSILON
//      << endl;

// cout << "term9=" << + rho2 * L3 * U*fabs(U)  * K_prop
//     << endl;
}

//////////////////////////////////////////////////////////////////////////////

   rhs [PITCH] = // Pitch Motion Equation right hand side ----------------------//

       - (I_x - I_z) * P*R + I_xy * Q*R - I_yz * P*Q - I_xz * (P2 - R2)

       +  m * ( x_G * ( -U*Q + V*P) - z_G  * ( - V*R + W*Q))

       + rho2 * L5  * (  M_pp * P2   + M_pr * P*R  + M_rr * R*fabs (R)

                       + M_q  * Q
                       + M_qq * Q * fabs(Q))  // hovering pitch drag

       + rho2 * L4  * ( M_uq * U*Q  + M_vp * V*P  + M_vr * V*R)

       + rho2 * L3  * ( M_uw * U*W  + M_vv * V2

                      + U*fabs(U) * (  M_uu_delta_bow   * delta_planes_bow
                                     + M_uu_delta_stern * delta_planes_stern))

       + pitch_integral  // note sign corrections to Healey pitch_integral

       - (x_G * Weight   -  revised_x_B * revisedBuoyancy) * cosTHETA * cosPHI

       - (z_G * Weight   -  z_B * revisedBuoyancy) * sinTHETA

       +  (2.0 / (24.0 * 24.0))  // each thruster 2.0 lb per 24V signal squared
                                 // multiplied by respective moment arms
                                 // x_bow_vertical (+), x_stern_vert (-)

        * ( (AUV_bow_vertical   * fabs (AUV_bow_vertical)   * x_bow_vertical)
           +(AUV_stern_vertical * fabs (AUV_stern_vertical) * x_stern_vertical));

       // EPSILON terms have been removed due to revised equations of motion
       // + rho2 * L4  *  M_qn * U*Q * EPSILON
       // + rho2 * L3  * (M_wn * U*W + M_dsn * U*fabs(U) * delta_planes_stern)
       //              * EPSILON;

//////////////////////////////////////////////////////////////////////////////
```

```
if (TRACE || TRACE_EOM || (rhs [PITCH] >= MAX_PITCH))   // Pitch TRACE
{
cout << "* pitch term1=" << - (I_x - I_z) * P*R + I_xy * Q*R - I_yz * P*Q
        - I_xz * (P2 - R2) << endl;

cout << "term2=" << +  m * ( x_G * ( -U*Q + V*P) - z_G  * ( - V*R + W*Q))
     << endl;

cout << "term3=" << + rho2 * L5  * (  M_pp * P2   + M_pr * P*R  + M_rr
                                                                 * R*fabs (R)
                       + M_q  * Q
                       + M_qq * Q * fabs(Q))  // hovering pitch drag
     << endl;

cout << "term4=" << + rho2 * L4  * ( M_uq * U*Q  + M_vp * V*P  + M_vr * V*R)
     << endl;

cout << "term5=" << + rho2 * L3  * ( M_uw * U*W  + M_vv * V2

                      + U*fabs(U) * (  M_uu_delta_bow   * delta_planes_bow
                                     + M_uu_delta_stern * delta_planes_stern))
     << endl;

cout << "term6=" << + pitch_integral << " pitch_integral"
     << endl;

cout << "term7=" << - (x_G * Weight   -  revised_x_B * revisedBuoyancy)
                      * cosTHETA * cosPHI
     << endl;

cout << "term8=" << - (z_G * Weight   -  z_B * revisedBuoyancy) * sinTHETA
     << endl;

cout << "term9=" << +  (2.0 / (24.0 * 24.0))
                                  // each thruster 2.0 lb per 24V signal squared
                                  // multiplied by respective moment arms
                                  // x_bow_vertical (+), x_stern_vert (-)

        * ( (AUV_bow_vertical   * fabs (AUV_bow_vertical)   * x_bow_vertical)
           +(AUV_stern_vertical * fabs (AUV_stern_vertical) * x_stern_vertical))
     << endl;

cout << "term10,term11=" <<   "EPSILON terms, no longer used"
     << endl;

// cout << "term10=" << + rho2 * L4  *  M_qn * U*Q * EPSILON
//      << endl;

// cout << "term11=" << + rho2 * L3  * (M_wn * U*W + M_dsn * U*fabs(U)
//                                                    * delta_planes_stern)
//                      * EPSILON
//     << endl;
}

////////////////////////////////////////////////////////////////////////////

   rhs [YAW  ] = // Yaw   Motion Equation right hand side --------------------//

       - (I_y - I_x) * P*Q + I_xy * (P2 - Q2) + I_yz * P*R - I_xz * Q*R

       -  m * ( x_G * ( U*R - W*P) - y_G   * ( - V*R + W*Q))

       + rho2 * L5  * (  N_pq * P*Q  + N_qr * Q*R

                       + N_r  * R
```

```
                        + N_rr * R * fabs (R))  // hovering yaw drag

      + rho2 * L4  * ( N_up * U*P  + N_ur * U*R  + N_vq * V*Q

                       + N_wp * W*P  + N_wr * W*R)

      + rho2 * L3  * ( N_uv * U*V  + N_vw * V*W

                       + U*fabs(U) * N_uu_delta_rb * delta_rudder_bow
                       - U*fabs(U) * N_uu_delta_rs * delta_rudder_stern)

      - yaw_integral

      + (x_G * Weight  -  revised_x_B * revisedBuoyancy) * cosTHETA * sinPHI

      + (y_G * Weight  -  y_B * revisedBuoyancy) * sinTHETA

      -  (2.0 / (24.0 * 24.0)) // each thruster 2.0 lb per 24V signal squared
                               // multiplied by respective moment arms
         * ( (AUV_bow_lateral   * fabs (AUV_bow_lateral)   * x_bow_lateral   )
            +(AUV_stern_lateral * fabs (AUV_stern_lateral) * x_stern_lateral ))

      - rho2 * L2 * C_d0

                  * ( square (SPEED_PER_RPM) *  0.5 // propeller yaw

                     * (  AUV_port_rpm * fabs(AUV_port_rpm) * y_port_propeller
                                       * port_propeller_efficiency

                        + AUV_stbd_rpm * fabs(AUV_stbd_rpm) * y_stbd_propeller
                                       * stbd_propeller_efficiency)

// *** revision:  removed ( - U * fabs(U) ) term from dissertation, incorrect
//                          - U * fabs(U));
                      );

////////////////////////////////////////////////////////////////////////////

if (TRACE || TRACE_EOM || (rhs [YAW] >= MAX_YAW))   // Yaw TRACE
{
cout << "* yaw term1=" <<  - (I_y - I_x) * P*Q + I_xy * (P2 - Q2)
                         + I_yz * P*R - I_xz * Q*R
     << endl;

cout << "term2=" <<   -  m * ( x_G * ( U*R - W*P) - y_G   * ( - V*R + W*Q))
     << endl;

cout << "term3=" <<   + rho2 * L5  * (  N_pq * P*Q  + N_qr * Q*R

                       + N_r  * R
                       + N_rr * R * fabs (R))  // hovering yaw drag
     << endl;

cout << "term4=" <<   + rho2 * L4  * ( N_up * U*P  + N_ur * U*R  + N_vq * V*Q

                       + N_wp * W*P  + N_wr * W*R)
     << endl;

cout << "term5=" << + rho2 * L3  * ( N_uv * U*V  + N_vw * V*W

                       + U*fabs(U) * N_uu_delta_rb * delta_rudder_bow
                       - U*fabs(U) * N_uu_delta_rs * delta_rudder_stern)
     << endl;

cout << "term6=" <<  - yaw_integral << " yaw_integral"
```

```
     << endl;

cout << "term7=" <<  + (x_G * Weight  -  revised_x_B * revisedBuoyancy)
                    * cosTHETA * sinPHI
     << endl;

cout << "term8=" <<  + (y_G * Weight  -  y_B * revisedBuoyancy) * sinTHETA
     << endl;

cout << "term9=" << -  (2.0 / (24.0 * 24.0))
                      // each thruster 2.0 lb per 24V signal squared
                      // multiplied by respective moment arms
          * ( (AUV_bow_lateral   * fabs (AUV_bow_lateral)   * x_bow_lateral   )
             +(AUV_stern_lateral * fabs (AUV_stern_lateral) * x_stern_lateral ))
     << endl;

cout << "term10=" << - rho2 * L2 * C_d0

                  * ( square (SPEED_PER_RPM) *  0.5 // propeller yaw

                     * (  AUV_port_rpm * fabs(AUV_port_rpm) * y_port_propeller
                                       * port_propeller_efficiency

                        + AUV_stbd_rpm * fabs(AUV_stbd_rpm) * y_stbd_propeller
                                       * stbd_propeller_efficiency)

// *** revision:  removed ( - U * fabs(U) ) term from dissertation, incorrect
//                          - U * fabs(U));

                       )
     << endl;
}

////////////////////////////////////////////////////////////////////////////

// debug section.  selectively set rhs values to zero to isolate problems.
//                 also see zeroing of sway/heave/pitch/yaw integrals.

// rhs [SURGE] = 0.0;
// rhs [SWAY ] = 0.0;
// rhs [HEAVE] = 0.0;
// rhs [ROLL ] = 0.0;
// rhs [PITCH] = 0.0;
// rhs [YAW  ] = 0.0;

    MAX_ACCELERATIONS_EXCEEDED =
        ((rhs [SURGE] >= MAX_SURGE) || (rhs [SWAY ] >= MAX_SWAY ) ||
         (rhs [HEAVE] >= MAX_HEAVE) || (rhs [ROLL ] >= MAX_ROLL ) ||
         (rhs [PITCH] >= MAX_PITCH) || (rhs [YAW  ] >= MAX_YAW  ));

if (TRACE)
{
   cout << "      SURGE  = " <<      SURGE  << endl;
   cout << "      SWAY   = " <<      SWAY   << endl;
   cout << "      HEAVE  = " <<      HEAVE  << endl;
   cout << "      ROLL   = " <<      ROLL   << endl;
   cout << "      PITCH  = " <<      PITCH  << endl;
   cout << "      YAW    = " <<      YAW    << endl;
}
if (TRACE || TRACE_EOM)
{
   cout << "      SURGE  = " <<      SURGE  << endl;
   cout << "      SWAY   = " <<      SWAY   << endl;
   cout << "      HEAVE  = " <<      HEAVE  << endl;
   cout << "      ROLL   = " <<      ROLL   << endl;
```

```
   cout << "     PITCH  = " <<      PITCH  << endl;
   cout << "     YAW    = " <<      YAW    << endl;

   cout << "rhs [SURGE] = " << rhs [SURGE] << endl;
   cout << "rhs [SWAY ] = " << rhs [SWAY ] << endl;
   cout << "rhs [HEAVE] = " << rhs [HEAVE] << endl;
   cout << "rhs [ROLL ] = " << rhs [ROLL ] << endl;
   cout << "rhs [PITCH] = " << rhs [PITCH] << endl;
   cout << "rhs [YAW  ] = " << rhs [YAW  ] << endl;

// cout << "mass_inverse:   ";   print_matrix6x6 (mass_inverse);
}
if (TRACE || TRACE_EOM || MAX_ACCELERATIONS_EXCEEDED)
{
   cout << "velocities:     <" << U << ", " << V << ", " << W << ", "
                              << P << ", " << Q << ", " << R << ">" << endl;
   cout << "RHS:            ";   print_matrix6 (rhs);
}

////////////////////////////////////////////////////////////////////////////

   // calculate new accelerations matrix using mass_inverse & rhs, print -----//

   multiply6x6_6 (mass_inverse, rhs, new_acceleration);

   if (TRACE || TRACE_EOM)
   {
       cout << "Accelerations:  ";   print_matrix6 (new_acceleration);
   }

   // limit accelerations ----------------------------------------------------//

   if (CLAMP)  // values are for NPS AUV, consider parameterizing
   {
      clamp (& new_acceleration [SURGE], -MAX_SURGE, MAX_SURGE,
              "new_acceleration [SURGE]");
      clamp (& new_acceleration [SWAY ], -MAX_SWAY , MAX_SWAY ,
              "new_acceleration [SWAY ]");
      clamp (& new_acceleration [HEAVE], -MAX_HEAVE, MAX_HEAVE,
              "new_acceleration [HEAVE]");

      clamp (& new_acceleration [ROLL ], -MAX_ROLL , MAX_ROLL ,
              "new_acceleration [ROLL ]");
      clamp (& new_acceleration [PITCH], -MAX_PITCH, MAX_PITCH,
              "new_acceleration [PITCH]");
      clamp (& new_acceleration [YAW  ], -MAX_YAW  , MAX_YAW  ,
              "new_acceleration [YAW  ]");
   }
                .

   // find velocities by integrating averaged accelerations -----------------//
   //      (Heun integration)

   new_velocity [SURGE] = 0.5 * (u_dot + new_acceleration [SURGE]) * dt + U;
   new_velocity [SWAY ] = 0.5 * (v_dot + new_acceleration [SWAY ]) * dt + V;
   new_velocity [HEAVE] = 0.5 * (w_dot + new_acceleration [HEAVE]) * dt + W;
   new_velocity [ROLL ] = 0.5 * (p_dot + new_acceleration [ROLL ]) * dt + P;
   new_velocity [PITCH] = 0.5 * (q_dot + new_acceleration [PITCH]) * dt + Q;
   new_velocity [YAW  ] = 0.5 * (r_dot + new_acceleration [YAW  ]) * dt + R;

   // find velocities by integrating instantaneous accelerations
   //      (Euler integration)
   //      (this method is less accurate and is not used, although at small
   //       timesteps the difference is negligible)
```

```
// new_velocity [SURGE] = (new_acceleration [SURGE]) * dt + U;
// new_velocity [SWAY ] = (new_acceleration [SWAY ]) * dt + V;
// new_velocity [HEAVE] = (new_acceleration [HEAVE]) * dt + W;
// new_velocity [ROLL ] = (new_acceleration [ROLL ]) * dt + P;
// new_velocity [PITCH] = (new_acceleration [PITCH]) * dt + Q;
// new_velocity [YAW  ] = (new_acceleration [YAW  ]) * dt + R;


// Note that surge velocity may be negative under model constraints
// but reverse stability is a problem.  Originally clamped non-negative.

if (CLAMP)
{
  clamp (& new_velocity [SURGE], -MAX_SURGE, MAX_SURGE,
          "new_velocity [SURGE] velocity");
}

// update UUVBody state accelerations to newly-calculated values   --------//

u_dot = new_acceleration [SURGE];
v_dot = new_acceleration [SWAY ];
w_dot = new_acceleration [HEAVE];
p_dot = new_acceleration [ROLL ];
q_dot = new_acceleration [PITCH];
r_dot = new_acceleration [YAW  ];

// calculate world coordinate system linear & angular velocities   --------//

//  see Cooke Figure 10 for corrections to Healey equations for x/y/z_dot:
//  also Healey course notes eqn (26) and Frank-McGhee corrected paper (A.8)

x_dot = AUV_oceancurrent_x

      + U *  cos (PSI) * cos (THETA)

      + V * (cos (PSI) * sin (THETA) * sin (PHI) - sin (PSI) * cos(PHI))

      + W * (cos (PSI) * sin (THETA) * cos (PHI) + sin (PSI) * sin(PHI));

y_dot = AUV_oceancurrent_y

      + U *  sin (PSI) * cos (THETA)

      + V * (sin (PSI) * sin (THETA) * sin (PHI) + cos (PSI) * cos(PHI))

      + W * (sin (PSI) * sin (THETA) * cos (PHI) - cos (PSI) * sin(PHI));

z_dot = AUV_oceancurrent_z

      - U * sin (THETA)

      + V * cos (THETA) * sin (PHI)

      + W * cos (THETA) * cos (PHI);


phi_dot   = P + Q * sin (PHI) * tan (THETA)

              + R * cos (PHI) * tan (THETA);

theta_dot =     Q * cos (PHI)

              - R * sin (PHI);
```

```
if  (cos (THETA) == 0.0)
{
     cout << "UUVBody::integrate_equations_of_motion ():  "        << endl;
     cout << "   cos (THETA) == 0.0 so psi_dot set equal to zero." << endl;
     psi_dot = 0.0;
}
else psi_dot = (Q * sin (PHI) + R * cos (PHI)) / cos (THETA);


Vector3D linear_rates =  Vector3D (x_dot, y_dot, z_dot);
if (TRACE || TRACE_EOM)
{
     cout << endl;
     cout << "<x_dot, y_dot, z_dot>         = " << linear_rates << endl;
     cout << "                     magnitude = " << linear_rates.magnitude ()
          << endl;
}

Vector3D euler_rates =  Vector3D (phi_dot, theta_dot, psi_dot);
if (TRACE || TRACE_EOM)
{
     cout << "<phi_dot, theta_dot, psi_dot> = " << euler_rates << endl;
     cout << "                     magnitude = " << euler_rates.magnitude ()
          << endl;
}

// calculate world coordinate system homogenous transform matrix   --------//

Hmatrix Hincremental = Hmatrix (); // default initialization
Hincremental.set_orientation ( P * dt, Q * dt, R * dt );
Hincremental.rotate      ( PHI,    THETA,   PSI    );

double omega_x = Hincremental.phi_value   ();
double omega_y = Hincremental.theta_value ();
double omega_z = Hincremental.psi_value   ();

Vector3D world_rates =  Vector3D (omega_x, omega_y, omega_z);
if (TRACE || TRACE_EOM)
{
     cout << "<omega_x, omega_y, omega_z> = " << world_rates << endl;
     cout << "                   magnitude = " << world_rates.magnitude ()
          << endl;
}

Hmatrix Hrevised1 = Hmatrix ();   // default initialization
Hrevised1.incremental_rotation    ( phi_dot, theta_dot, psi_dot, dt );
Hrevised1.incremental_translation ( U, V, W, dt );

Hmatrix Hproduct1 = Hprevious * Hrevised1;
Hproduct1.incremental_translation (AUV_oceancurrent_x,
                                   AUV_oceancurrent_y,
                                   AUV_oceancurrent_z, dt);

Hprevious = Hproduct1;

// translate and rotate and update time in RigidBody state ----------------//
// note world coordinate system is used by RigidBody:

set_angular_velocities (phi_dot, theta_dot, psi_dot);

set_linear_velocities  (  x_dot,     y_dot,   z_dot);

set_time_of_posture    (current_uuv_time);

update_Hmatrix         (dt);
```

```
    if (TRACE)
    {
        cout << "incremental hmatrix = ";
        Hincremental.print_hmatrix ();
        cout << "revised1 hmatrix = ";
        Hrevised1.print_hmatrix ();
        cout << "product1 hmatrix = ";
        Hproduct1.print_hmatrix ();

        cout << "original hmatrix = ";
        hmatrix.print_hmatrix ();
    }

    if (TRACE)  cout << "substituting product1 hmatrix" << endl;
    hmatrix = Hproduct1;


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

    // Save body-coordinate-system velocities for the next loop:

    U = new_velocity [SURGE];
    V = new_velocity [SWAY ];
    W = new_velocity [HEAVE];
    P = new_velocity [ROLL ];
    Q = new_velocity [PITCH];
    R = new_velocity [YAW  ];

// cout << "world U =" << U << ", x_dot     = " << x_dot     << endl;
// cout << "world V =" << V << ", y_dot     = " << y_dot     << endl;
// cout << "world W =" << W << ", z_dot     = " << z_dot     << endl;
// cout << "world P =" << P << ", phi_dot   = " << phi_dot   << endl;
// cout << "world Q =" << Q << ", theta_dot = " << theta_dot << endl;
// cout << "world R =" << R << ", psi_dot   = " << psi_dot   << endl;

    // ---------------------------------------------------------------------
    // update all hydrodynamics-model-provided state variables in AUV_globals.h
    //        prior to retransmittal to AUV via AUVsocket

    AUV_time      = current_uuv_time; // mission time

    AUV_x         = x_value     (); //  x    position in world coordinates
    AUV_y         = y_value     (); //  y    position in world coordinates
    AUV_z         = z_value     (); //  z    position in world coordinates
    AUV_phi       = phi_value   (); //  roll  posture in world coordinates
    AUV_theta     = theta_value (); //  pitch posture in world coordinates
    AUV_psi       = psi_value   (); //  yaw   posture in world coordinates

    AUV_speed=new_velocity [SURGE]; //  paddlewheel speed = u = surge

    AUV_u   = new_velocity [SURGE]; //  surge  linear velocity along x-axis
    AUV_v   = new_velocity [SWAY ]; //  sway   linear velocity along y-axis
    AUV_w   = new_velocity [HEAVE]; //  heave  linear velocity along x-axis
    AUV_p   = new_velocity [ROLL ]; //  roll  angular velocity about x-axis
    AUV_q   = new_velocity [PITCH]; //  pitch angular velocity about y-axis
    AUV_r   = new_velocity [YAW  ]; //  yaw   angular velocity about z-axis

    AUV_u_dot     = u_dot;          //     linear acceleration along x-axis
    AUV_v_dot     = v_dot;          //     linear acceleration along y-axis
    AUV_w_dot     = w_dot;          //     linear acceleration along x-axis
    AUV_p_dot     = p_dot;          //    angular acceleration about x-axis
    AUV_q_dot     = q_dot;          //    angular acceleration about y-axis
    AUV_r_dot     = r_dot;          //    angular acceleration about z-axis

    AUV_x_dot     = x_dot;          //      Euler velocity along North-axis
```

```
   AUV_y_dot     = y_dot;          //      Euler velocity along  East-axis
   AUV_z_dot     = z_dot;          //      Euler velocity along Depth-axis
   AUV_phi_dot   = phi_dot;        // Euler rotation rate about North-axis
   AUV_theta_dot = theta_dot;      // Euler rotation rate about  East-axis
   AUV_psi_dot   = psi_dot;        // Euler rotation rate about Depth-axis

   divetracker_range1 = sqrt (sqr (AUV_x - DiveTracker1_x) +
                              sqr (AUV_y - DiveTracker1_y) +
                              sqr (AUV_z - DiveTracker1_z));

   divetracker_range2 = sqrt (sqr (AUV_x - DiveTracker2_x) +
                              sqr (AUV_y - DiveTracker2_y) +
                              sqr (AUV_z - DiveTracker2_z));

  //-----------------------------------
  //set value of doppler sonar outputs
  //-----------------------------------
  //doppler speed over ground in meters/sec
  doppler_sog_u = U * 0.3048;
  doppler_sog_v = V * 0.3048;

  //doppler speed through water in meters/sec
  doppler_stw_u = doppler_stw_u *  0.3048;
  doppler_stw_v = doppler_stw_v *  0.3048;

  //doppler altitude returns height of AUV above bottom in meters, I assume total depth of
100 meters
  doppler_altitude = 100.0 - AUV_z;

   if (FALSE && TRACE_EOM && MAX_ACCELERATIONS_EXCEEDED)
   {
       char  user_pause;
       cout << "==== Hit enter to continue... ====";
       cin  >> user_pause;
       cout << endl;
   }

   return; // integrate_equations_of_motion () complete
}
```

APPENDIX B. VIRTUAL ENVIRONMENT JAVA/VRML CODE

1. Java Source Code

This appendix includes the files needed for the Phoenix AUV dynamics to run in Java with the virtual environment done in VRML. Since the functionality of the C++ and Java version are the same the actual source code is not included. The source code is freely distributed at http://www.stl.nps.navy.mil/~auv. Please feel free to download a complete version of the code if it is required. The complete list of Java files needed to run the virtual environment follows:

dynamics.java	UUVBody.java
AUVglobals.java	UUVmodel.java
SonarModel.java	AUVsocket.java
AUVmodel.java	FlowFileReader.java
RigidBody.java	DISNetworkedRigidBody.java
Hmatrix.java	Vector3D.java
MathU.java	Console.java

Additionally, the DIS-Java-VRML library is required to compile the program. This can be obtained free of charge at *http://www.stl.nps.navy.mil/dis-java-vrml.*

2. AUVvirtual.wrl

```
#VRML V2.0 utf8

#This file creates a Virtual world for the Phoenix AUV
#Author:  Kevin Byrne
#Date :   28 January 1998

##########################################
#This is the externproto to link in DIS pdu's


EXTERNPROTO EspduReadTransformTrace [

field                   SFString     marking          # 0..11 character label for
entity
field           SFTime               readInterval  # seconds between DIS updates
field           SFString     address                    # multicast address or
"unicast"
field           SFInt32              port               # port number

exposedField MFNode             children
field           SFVec3f              translation
field                   SFRotation   rotation
exposedField SFVec3f            scale
exposedField SFRotation   scaleOrientation
field           SFVec3f              bboxCenter
field           SFVec3f              bboxSize
exposedField SFVec3f            center
eventIn                 MFNode              addChildren
eventIn                 MFNode              removeChildren

] [ "EspduReadTransform.wrl"
    "../JavaViaScriptNode/EspduReadTransform.wrl" # local or remote URLs for the
EXTERNPROTO

"http://www.stl.nps.navy.mil/dis-java-vrml/mil/navy/nps/JavaViaScriptNode/EspduReadTransfo
rm.wrl"
  ]


EspduReadTransformTrace {

       marking      "Phoenix AUV"

       readInterval 2                  # seconds between DIS reads

# do not modify address/port while using  unicast-only browser, run bridge instead
#                   address             "224.2.244.141"     # NPS AUV exercise default,
multicast
#                   port                3111          # NPS AUV exercise default

              children Inline {
                    url    [      "phoenix_auv.wrl"
                                  "http://web.nps.navy.mil/~kmbyrne/AUVvw/phoenix_auv.wrl"
              ]
       }

       translation  2 -2 0        # offset for initial location/orientation,

}


#End of Proto
#################################################################
```

```
Group {
   children [
      #This Section states Navigation info
      NavigationInfo  {
         type   "EXAMINE"       #In the end should be FLY
         speed  10.0
         avatarSize    [0.26, 1.6, 0.75]
      },
      #This section adds a background to the scene
      Background {
       skyColor [
             0.0 0.2 0.7,
             0.0 0.5 1.0,
             0.4 0.8 1.0  ]
       skyAngle [1.309, 1.571]
      },

      #This Section creates the Ocean Floor
      Transform  {
       translation -75.0 -30.0 -75.0
       children Shape {
             appearance Appearance {
                   material Material {
                          ambientIntensity 0.50
                          diffuseColor 0.3 0.11 0.00
                          specularColor 0.75 0.33 0.00
                          shininess 0.10
                   }
             }
             geometry ElevationGrid {
                   xDimension 15
                   zDimension 15
                   xSpacing   10
                   zSpacing   10
                   solid      FALSE
                          creaseAngle 0.785
                   height [0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                           0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
                     ]
             }
        }
      },


      #This section adds the sea surface
      #It uses an indexed face set
      Shape {
        appearance Appearance {
                   material Material {
                          ambientIntensity 0.70
                          diffuseColor 0.0 0.0 1.0
                          specularColor 0.0 0.0 1.00
```

```
                    shininess 0.10
                    transparency 0.3
                }
        }
        geometry IndexedFaceSet {
                coord   Coordinate {
                    point [ -75  0  75,
                           -75  0 -75,
                            75  0 -75,
                            75  0  75,]
                }
                coordIndex [  0, 1, 2, 3, ]
                solid FALSE

        }
},

#This section adds the Sun
DirectionalLight {
 direction 0.0 -1.0 0.0
},

#This Section places a 688 Class Submarine in the scene
Transform {
   translation 0.0 -5.0 40.0
   rotation    0.0 1.0 0.0 3.142
  #This sub must be scaled down from 600 M to ~100 M
  scale 0.1666 0.1666 0.1666
  children [
       Inline {
         bboxSize 500.0 300.0 300.0
         url "688.wrl"
       }
 ]
},

#This transform places the oil rig in the scene
Transform {
   translation -45.0 7.0 -10.0
   children [
     Inline  {
       bboxSize 200.0 200.0 200.0
       url "oil_rig.wrl"
     }
   ]

},

#This places a tube on the sea floor
Transform {
   translation 0.0 -30.0 0.0
   rotation    1.0 0.0 0.0 1.571
   children  [
     Shape {
       appearance Appearance {
          material Material {
             diffuseColor 0.8 1.0 0.0
          }
       }

       geometry Cylinder {
          height 6.0
          top    FALSE
          bottom FALSE
       }
```

```
            }
        ]

        },
    ]
}
#end of file AUVvirtual.wrl
```

3. Oil_rig.wrl

```
#VRML V2.0 utf8

#This file creates an oil rig for the Phoenix AUV VW
#Author:  Kevin Byrne
#Date :   28 January 1998


NavigationInfo {
  type [ "EXAMINE" "ALL" ]


}

Viewpoint {
    position 17 35 0
    orientation 0 1 0 0
    description "On Oil Rig"
}

Group {
  children [

    #This creates the left  forward leg
    Transform {
       translation 0.0 -10.0 0.0
       children  [
          Shape {
             appearance Appearance {
                material DEF blueMetal Material {
                   diffuseColor 0.4 0.4 1.0
                }
             }

             geometry DEF Leg Cylinder {
                radius 3.0
                height 55.0
             }
          }
       ]
    },

    #Back Left Leg
    Transform {
       translation  0.0 -10.0 -35.0
          children  [
             Shape {
                appearance Appearance {
                   material USE blueMetal
                }

                geometry USE Leg
             }
          ]
     },

   #Back Right Leg
    Transform {
       translation  35.0 -10.0 -35.0
          children  [
             Shape {
                appearance Appearance {
                   material USE blueMetal
                }
```

```
            geometry USE Leg
         }
      ]
   },

#Front Right Leg
 Transform {
    translation  35.0 -10.0 0.0
       children  [
          Shape {
             appearance Appearance {
                material USE blueMetal
             }

             geometry USE Leg
          }
       ]
   },

#This creates the left  forward crossbeam
 Transform {
    translation  17.5 10.0 0.0
    rotation     0.0 0.0 1.0 1.2
    children  [
       Shape {
          appearance Appearance {
             material DEF white Material {
                diffuseColor 1.0 1.0 1.0
             }
          }

          geometry DEF CrossLeg Cylinder {
             radius 0.8
             height 38.0
          }
       }
    ]
 },

#Left forward cross beam 2
 Transform {
    translation  17.5 10.0 0.0
    rotation     0.0 0.0 1.0  -1.2
       children  [
          Shape {
             appearance Appearance {
                material USE white
             }

             geometry USE CrossLeg
          }
       ]
   },

#Left forward cross beam 3
  Transform {
     translation  0.0 10.0 -17.5
     rotation     1.0 0.0 0.0  1.2
        children  [
           Shape {
              appearance Appearance {
                 material USE white
              }
```

```
                geometry USE CrossLeg
            }
        ]
    },


 #Left forward cross beam 4
   Transform {
      translation  0.0 10.0 -17.5
      rotation     1.0 0.0 0.0  -1.2
         children  [
            Shape {
               appearance Appearance {
                  material USE white
               }

               geometry USE CrossLeg
            }
         ]
    },


#Left forward cross beam 5
   Transform {
      translation  35.0 10.0 -17.5
      rotation     1.0 0.0 0.0  1.2
         children  [
            Shape {
               appearance Appearance {
                  material USE white
               }

               geometry USE CrossLeg
            }
         ]
    },


 #Left forward cross beam 6
   Transform {
      translation  35.0 10.0 -17.5
      rotation     1.0 0.0 0.0  -1.2
         children  [
            Shape {
               appearance Appearance {
                  material USE white
               }

               geometry USE CrossLeg
            }
         ]
    },


  #Left forward cross beam 7
   Transform {
      translation  17.5 10.0 -35.0
      rotation     0.0 0.0 1.0  1.2
         children  [
            Shape {
               appearance Appearance {
                  material USE white
               }

               geometry USE CrossLeg
```

```
            }
         ]
   },
#Left forward cross beam 8
 Transform {
    translation  17.5 10.0 -35.0
    rotation     0.0 0.0 1.0  -1.2
       children  [
          Shape {
             appearance Appearance {
                material USE white
             }

             geometry USE CrossLeg
          }
       ]
   },

#This creates the bottom platform
 Transform {
    translation  17.5 17.5 -17.5
    children  [
       Shape {
          appearance Appearance {
             material Material {
                diffuseColor 1.0 0.2 0.2
             }
          }

          geometry Box {
             size 50.0 2.0 50.0
          }
       }
    ]
 },

 #This places a simple box-like building on the oil rig
 Transform {
    translation  27.5 22.5 -17.5
    children  [
       Shape {
          appearance Appearance {
             material Material {
                diffuseColor 1.0 1.0 0.0
             }
          }

          geometry Box {
             size 15.0 8.0 10.0
          }
       }
    ]
 }

#This places a second simple box-like building on the oil rig
 Transform {
    translation  5.0 21.5 -17.5
    children  [
       Shape {
          appearance Appearance {
             material Material {
                diffuseColor 0.4 1.0 0.4
             }
          }
```

```
            geometry Box {
               size 10.0 6.0 10.0
            }
         }
      ]
   }


   ]   # end of oilRig group children

}  # end of oilRig group

#end of file oil_rig.wrl
```

4. 688.wrl

```
#VRML V2.0 utf8
#This file creates a submarine
#Author:  Kevin Byrne
#Date :   2 Dec 1997


#This Section builds a 688 Class Submarine
Transform {

       children [
       #The Submarine Hull
       Group {
              children     [
                  NavigationInfo {
                     avatarSize    [ 0.3, 1.6, 0.75 ]
                  },
                  Viewpoint {
                     position      -600 0 0
                     orientation   0 1.0 0 -1.571
                     fieldOfView   0.8
                     description   "Upper Foward End of 688, looking Aft"
                  },
                  Viewpoint {
                     position      -40 20 -110
                     orientation   0 1.0 0 3.14
                     fieldOfView   0.95993
                     description   "Upper STBD Side of 688, looking to Port"
                  },
                  Viewpoint {
                     position      350 50 10
                     orientation   0 1.0 0 1.571
                     fieldOfView   0.8
                     description   "Upper Aft End of 688, looking Fwd"
                  },
                  Viewpoint {
                     position      -40 20 120
                     orientation   0 0 0 0
                     fieldOfView   0.95993
                     description   "Upper Port Side of 688, looking to STBD"
                  },

                 Viewpoint {
                     position      -350 -60 10
                     orientation   0 1.0 0 -1.571
                     fieldOfView   0.8
                     description   "Lower Foward End of 688, looking Aft"
                  },
                  Viewpoint {
                     position      -80 -20 -110
                     orientation   0 1.0 0 3.14
                     fieldOfView   0.95993
                     description   "Lower STBD Side of 688, looking to Port"
                  },
                  Viewpoint {
                     position      350 -50 10
                     orientation   0 1.0 0 1.571
                     fieldOfView   0.8
                     description   "Lower Aft End of 688, looking Fwd"
                  },
                  Viewpoint {
                     position      -80 -20 120
                     orientation   0 0 0 0
                     fieldOfView   0.95993
```

```
   description   "Lower Port Side of 688, looking to STBD"
},

Transform {
    rotation    0.577417 -0.577317 -0.577317  4.18889
   center       0 2.5 10
   children
         Shape {
             appearance          Appearance {
             material DEF _688_Hull Material {
             ambientIntensity       0.1
             diffuseColor    0.1 0.1 0.1
             specularColor   0.1 0.1 0.1
          }

       }

       geometry  IndexedFaceSet {
          coord Coordinate {
              point     [     1.73 -160 5.33,
                       0 -162.5 4.92,              0 -160 5.6,
                       0.25 -165 4.23,             0.25 -167.5 3.55,
                       0.46 -175 1.43,             0 -172.5 2.18,
                       0 -175 1.5,           5.6 -160 0,
                       4.95 -162.5 0,              5.33 -160 1.73,
                       4.23 -165 0.25,             3.55 -167.5 0.25,
                       1.43 -175 0.46,             2.18 -172.5 0,
                       1.5 -175 0,           18 -172.5 10,
                       17.8 -170 10,         18.2 -170 10,
                       18.2 -167.5 10,             17.8 -167.5 10,
                       18 -166 10,           0 -172.5 15,
                       0 -166 15,                  0.2 -170 15,
                       17.8 -170 0.2,              17.8 -167.5 0.2,
                       18 -166 0,                  18.2 -170 0,
                       18 -172.5 0,          0.2 -167.5 15,
                       4.53 -160 3.29,             0.88 -175 1.21,
                       3.29 -160 4.53,             -3.29 -160 4.53,
                       -0.46 -175 1.43,            -0.88 -175 1.21,
                       -4.53 -160 3.29,            -1.21 -175 0.88,
                       -5.33 -160 1.73,            -1.43 -175 0.46,
                       -0.25 -165 4.23,            -0.2 -167.5 15,
                       -0.25 -167.5 3.55,        -0.2 -170 15,
                       -4.23 -165 0.25,            -17.8 -167.5 0.2,
                       -4.95 -162.5 0,             -3.55 -167.5 0.25,
                       -2.18 -172.5 0,             -18 -172.5 0,
                       -18.2 -170 10,              -18.2 -170 0,
                       -18.2 -167.5 10,            -18.2 -167.5 0,
                       -18 -166 10,          -18 -166 0,
                       -17.8 -167.5 10,            -17.8 -170 10,
                       -17.8 -170 0.2,             -18 -172.5 10,
                       -1.5 -175 0,          -5.6 -160 0,
                       -1.73 -160 5.33,            -1.73 -160 -5.33,
                       0 -162.5 -4.92,             0 -160 -5.6,
                       -0.25 -165 -4.23,           -0.25 -167.5 -3.55,
                       -0.46 -175 -1.43,           0 -172.5 -2.18,
                       0 -175 -1.5,          -5.33 -160 -1.73,
                       -4.23 -165 -0.25,           -3.55 -167.5 -0.25,
                       -1.43 -175 -0.46,           -18 -172.5 -10,
                       -17.8 -170 -10,             -18.2 -170 -10,
                       -18.2 -167.5 -10,           -17.8 -167.5 -10,
                       -18 -166 -10,         0 -172.5 -15,
                       0 -166 -15,           -0.2 -170 -15,
                       -17.8 -170 -0.2,            -17.8 -167.5 -0.2,
                       -0.2 -167.5 -15,            -4.53 -160 -3.29,
                       -0.88 -175 -1.21,           -3.29 -160 -4.53,
```

```
3.29 -160 -4.53,
0.88 -175 -1.21,
1.21 -175 -0.88,
1.43 -175 -0.46,
0.2 -167.5 -15,
0.2 -170 -15,
17.8 -167.5 -0.2,
18.2 -170 -10,
18.2 -167.5 0,
17.8 -167.5 -10,
17.8 -170 -0.2,
1.73 -160 -5.33,
1.6 179.6 0,
1.29 179.6 0.94,
0.5 179.6 1.52,
3.42 178 1.1,
2.11 178 2.91,
3.6 178 0,
5.33 176 1.73,
1.73 176 5.33,
0 178 3.6,
6.71 172 4.88,
4.88 172 6.71,
11.2 164 0,
9.06 164 6.58,
2.56 172 7.9,
14 152 0,
13.31 152 4.33,
4.33 152 13.31,
0 152 14,
12.14 140 8.82,
8.82 140 12.14,
15 140 0,
12.7 128 9.23,
4.85 128 14.93,
15.7 128 0,
12.82 116 9.32,
4.89 116 15.07,
15.85 116 0,
12.94 100 9.4,
0 100 16,
15.22 90 4.94,
12.94 90 9.4,
4.94 90 15.22,
0 90 16,
15.22 30 4.94,
12.94 30 9.4,
9.4 30 12.94,
4.94 30 15.22,
0 30 16,
16 30 0,
12.94 0 9.4,
4.94 0 15.22,
16 0 0,
12.94 -30 9.4,
4.94 -30 15.22,
16 -30 0,
12.94 -60 9.4,
4.94 -60 15.22,
16 -60 0,
12.94 -80 9.4,
4.94 -80 15.22,
16 -80 0,
14.93 -90 4.85,
4.85 -90 14.93,
0.46 -175 -1.43,
4.53 -160 -3.29,
5.33 -160 -1.73,
0.25 -165 -4.23,
0.25 -167.5 -3.55,
4.23 -165 -0.25,
3.55 -167.5 -0.25,
18.2 -167.5 -10,
18 -166 -10,
17.8 -170 -10,
18 -172.5 -10,
0 180 0,
1.52 179.6 0.5,
0.94 179.6 1.29,
0 179.6 1.6,
2.91 178 2.11,
1.1 178 3.42,
5.6 176 0,
3.29 176 4.53,
0 176 5.6,
8.3 172 0,
4.53 176 3.29,
0 172 8.3,
7.9 172 2.56,
6.58 164 9.06,
0 164 11.2,
10.65 164 3.46,
8.23 152 11.33,
3.46 164 10.65,
14.27 140 4.64,
11.33 152 8.23,
4.64 140 14.27,
14.93 128 4.85,
9.23 128 12.7,
0 140 15,
15.07 116 4.89,
9.32 116 12.82,
0 128 15.7,
16 100 0,
9.4 100 12.94,
0 116 15.85,
15.22 100 4.94,
9.4 90 12.94,
4.94 100 15.22,
16 60 0,
15.22 60 4.94,
12.94 60 9.4,
9.4 60 12.94,
4.94 60 15.22,
0 60 16,
15.22 0 4.94,
9.4 0 12.94,
0 0 16,
15.22 -30 4.94,
9.4 -30 12.94,
0 -30 16,
15.22 -60 4.94,
9.4 -60 12.94,
0 -60 16,
15.22 -80 4.94,
9.4 -80 12.94,
0 -80 16,
15.7 -90 0,
9.23 -90 12.7,
0 -90 15.7,
```

```
15 -110 0,
12.7 -90 9.23,
4.64 -110 14.27,
14 -120 0,
12.14 -110 8.82,
4.33 -120 13.31,
10.65 -139 3.46,
11.33 -120 8.23,
3.46 -139 10.65,
7.9 -150 2.56,
2.56 -150 7.9,
8.3 -150 0,
0 -150 8.3,
0 -150 -8.3,
6.71 -150 -4.88,
3.46 -139 -10.65,
6.58 -139 -9.06,
10.65 -139 -3.46,
0 -120 -14,
11.33 -120 -8.23,
4.64 -110 -14.27,
12.14 -110 -8.82,
4.85 -90 -14.93,
9.23 -90 -12.7,
14.93 -90 -4.85,
0 -90 -15.7,
12.94 -80 -9.4,
4.94 -60 -15.22,
9.4 -60 -12.94,
15.22 -60 -4.94,
0 -60 -16,
12.94 -30 -9.4,
4.94 0 -15.22,
9.4 0 -12.94,
15.22 0 -4.94,
0 0 -16,
12.94 30 -9.4,
4.94 60 -15.22,
9.4 60 -12.94,
15.22 60 -4.94,
0 90 -16,
9.4 100 -12.94,
12.94 100 -9.4,
15.22 100 -4.94,
16 90 0,
0 100 -16,
12.82 116 -9.32,
4.85 128 -14.93,
9.23 128 -12.7,
14.93 128 -4.85,
0 140 -15,
12.14 140 -8.82,
4.33 152 -13.31,
8.23 152 -11.33,
13.31 152 -4.33,
6.58 164 -9.06,
10.65 164 -3.46,
0 164 -11.2,
6.71 172 -4.88,
1.73 176 -5.33,
3.29 176 -4.53,
5.33 176 -1.73,
0 176 -5.6,
2.91 178 -2.11,
0.5 179.6 -1.52,
14.27 -110 4.64,
8.82 -110 12.14,
0 -110 15,
13.31 -120 4.33,
8.23 -120 11.33,
0 -120 14,
9.06 -139 6.58,
6.58 -139 9.06,
11.2 -139 0,
4.88 -150 6.71,
0 -139 11.2,
6.71 -150 4.88,
2.56 -150 -7.9,
4.88 -150 -6.71,
7.9 -150 -2.56,
0 -139 -11.2,
9.06 -139 -6.58,
4.33 -120 -13.31,
8.23 -120 -11.33,
13.31 -120 -4.33,
8.82 -110 -12.14,
14.27 -110 -4.64,
0 -110 -15,
12.7 -90 -9.23,
4.94 -80 -15.22,
9.4 -80 -12.94,
15.22 -80 -4.94,
0 -80 -16,
12.94 -60 -9.4,
4.94 -30 -15.22,
9.4 -30 -12.94,
15.22 -30 -4.94,
0 -30 -16,
12.94 0 -9.4,
4.94 30 -15.22,
9.4 30 -12.94,
15.22 30 -4.94,
0 30 -16,
12.94 60 -9.4,
4.94 100 -15.22,
4.94 90 -15.22,
9.4 90 -12.94,
12.94 90 -9.4,
15.22 90 -4.94,
4.89 116 -15.07,
9.32 116 -12.82,
15.07 116 -4.89,
0 128 -15.7,
12.7 128 -9.23,
4.64 140 -14.27,
8.82 140 -12.14,
14.27 140 -4.64,
0 152 -14,
11.33 152 -8.23,
3.46 164 -10.65,
9.06 164 -6.58,
2.56 172 -7.9,
4.88 172 -6.71,
7.9 172 -2.56,
0 172 -8.3,
4.53 176 -3.29,
1.1 178 -3.42,
2.11 178 -2.91,
3.42 178 -1.1,
0 179.6 -1.6,
```

```
0.94 179.6 -1.29,                1.29 179.6 -0.94,
1.52 179.6 -0.5,                 1 89 21,
0 90 21,                         1 89 16,
3 83 21,                         3 83 16,
3 71 21,                         3 71 16,
0 60 21,                         1 89 25,
0 90 25,                         3 83 25,
3 82 24,                         3 77 24,
3 71 25,                         0 60 25,
1 89 29,                         0 90 29,
3 83 29,                         3 81 26,
3 77 26,                         3 71 29,
0 60 29,                         1 89 34,
0 90 34,                         3 83 34,
3 71 34,                         0 60 34,
0 86 36,                         0 81 36,
9 71 25,                         9 75 26,
9 79 26,                         9 81 25,
9 80 24.3,                       9 75 24.3,
15 71 25,                        15 75 26,
15 77 26,                        15 79 25,
15 78 24.6,                      15 75 24.6,
-4.94 60 15.22,                  -3 71 16,
-3 83 16,                        -4.94 90 15.22,
-1 89 16,                        -9.4 90 12.94,
-9.4 60 12.94,                   -12.94 90 9.4,
-12.94 60 9.4,                   -15.22 90 4.94,
-15.22 60 4.94,                  -16 90 0,
-16 60 0,                        -15.22 90 -4.94,
-15.22 60 -4.94,                 -12.94 90 -9.4,
-12.94 60 -9.4,                  -9.4 90 -12.94,
-9.4 60 -12.94,                  -4.94 90 -15.22,
-4.94 60 -15.22,                 0 60 -16,
-15 77 26,                       -15 78 24.6,
-15 79 25,                       -15 75 24.6,
-15 75 26,                       -15 71 25,
-9 71 25,                        -9 75 24.3,
-9 80 24.3,                      -9 81 25,
-9 79 26,                        -9 75 26,
-3 71 25,          -3 77 24,
-3 82 24,                        -3 83 25,
-3 81 26,                        -3 77 26,
-3 71 34,                        -3 83 34,
-1 89 34,                        -3 71 29,
-3 83 29,                        -1 89 29,
-1 89 25,                        -3 71 21,
-3 83 21,                        -1 89 21,
-1.6 179.6 0,            -1.52 179.6 -0.5,
-1.29 179.6 -0.94,       -0.94 179.6 -1.29,
-0.5 179.6 -1.52,                -3.42 178 -1.1,
-2.91 178 -2.11,                 -2.11 178 -2.91,
-1.1 178 -3.42,                         -3.6 178 0,
-5.6 176 0,                      -5.33 176 -1.73,
-3.29 176 -4.53,                 -1.73 176 -5.33,
0 178 -3.6,                      -8.3 172 0,
-6.71 172 -3.88,                 -4.53 176 -3.29,
-4.88 172 -6.71,                 -11.2 164 0,
-7.9 172 -2.56,                  -9.06 164 -6.58,
-6.58 164 -9.06,                 -2.56 172 -7.9,
-14 152 0,                       -10.65 164 -3.46,
-13.31 152 -4.33,                -8.23 152 -11.33,
-4.33 152 -13.31,                -3.46 164 -10.65,
-14.27 140 -4.64,                -12.14 140 -8.82,
-11.33 152 -8.23,                -8.82 140 -12.14,
-4.64 140 -14.27,                -15 140 0,
```

```
-14.93 128 -4.85,                -12.7 128 -9.23,
-9.23 128 -12.7,                 -4.85 128 -14.93,
-15.7 128 0,                     -15.07 116 -4.89,
-12.82 116 -9.32,                -9.32 116 -12.82,
-4.89 116 -15.07,                -15.85 116 0,
-16 100 0,                       -12.94 100 -9.4,
-9.4 100 -12.94,                 0 116 -15.85,
-15.22 100 -4.94,                -4.94 100 -15.22,
-15.22 30 -4.94,                 -12.94 30 -9.4,
-9.4 30 -12.94,                  -4.94 30 -15.22,
-16 30 0,                        -15.22 0 -4.94,
-12.94 0 -9.4,                   -9.4 0 -12.94,
-4.94 0 -15.22,                  -16 0 0,
-15.22 -30 -4.94,                -12.94 -30 -9.4,
-9.4 -30 -12.94,                 -4.94 -30 -15.22,
-16 -30 0,                       -15.22 -60 -4.94,
-12.94 -60 -9.4,                 -9.4 -60 -12.94,
-4.94 -60 -15.22,                -16 -60 0,
-15.22 -80 -4.94,                -12.94 -80 -9.4,
-9.4 -80 -12.94,                 -4.94 -80 -15.22,
-16 -80 0,                       -15.7 -90 0,
-14.93 -90 -4.85,                -9.23 -90 -12.7,
-4.85 -90 -14.93,                -15 -110 0,
-14.27 -110 -4.64,               -12.7 -90 -9.23,
-8.82 -110 -12.14,         -4.64 -110 -14.27,
-14 -120 0,                -13.31 -120 -4.33,
-12.14 -110 -8.82,         -8.23 -120 -11.33,
-4.33 -120 -13.31,         -10.65 -139 -3.46,
-9.06 -139 -6.58,          -11.33 -120 -8.23,
-6.58 -139 -9.06,          -3.46 -139 -10.65,
-11.2 -139 0,        -7.9 -150 -2.56,
-4.88 -150 -6.71,          -2.56 -150 -7.9,
-8.3 -150 0,               -6.71 -150 -4.88,
-2.56 -150 7.9,            -4.88 -150 6.71,
-6.71 -150 4.88,           -7.9 -150 2.56,
-3.46 -139 10.65,          -6.58 -139 9.06,
-9.06 -139 6.58,           -10.65 -139 3.46,
-4.33 -120 13.31,          -8.23 -120 11.33,
-11.33 -120 8.23,          -13.31 -120 4.33,
-4.64 -110 14.27,          -8.82 -110 12.14,
-12.14 -110 8.82,          -14.27 -110 4.64,
-4.85 -90 14.93,           -9.23 -90 12.7,
-12.7 -90 9.23,            -14.93 -90 4.85,
-4.94 -80 15.22,           -9.4 -80 12.94,
-12.94 -80 9.4,            -15.22 -80 4.94,
-4.94 -60 15.22,           -9.4 -60 12.94,
-12.94 -60 9.4,            -15.22 -60 4.94,
-4.94 -30 15.22,           -9.4 -30 12.94,
-12.94 -30 9.4,            -15.22 -30 4.94,
-4.94 0 15.22,             -9.4 0 12.94,
-12.94 0 9.4,        -15.22 0 4.94,
-4.94 30 15.22,            -9.4 30 12.94,
-12.94 30 9.4,             -15.22 30 4.94,
-4.94 100 15.22,           -9.4 100 12.94,
-12.94 100 9.4,            -15.22 100 4.94,
-4.89 116 15.07,           -9.32 116 12.82,
-12.82 116 9.32,           -15.07 116 4.89,
-4.85 128 14.93,           -9.23 128 12.7,
-12.7 128 9.23,            -14.93 128 4.85,
-4.64 140 14.27,           -8.82 140 12.14,
-12.14 140 8.82,           -14.27 140 4.64,
-4.33 152 13.31,           -8.23 152 11.33,
-11.33 152 8.23,           -13.31 152 4.33,
-3.46 164 10.65,           -6.58 164 9.06,
-9.06 164 6.58,            -10.65 164 3.46,
```

```
                    -2.56 172 7.9,              -4.88 172 6.71,
                    -6.71 172 4.88,             -7.9 172 2.56,
                    -1.73 176 5.33,             -3.29 176 4.53,
                    -4.53 176 3.29,             -5.33 176 1.73,
                    -1.1 178 3.42,              -2.11 178 2.91,
                    -2.91 178 2.11,             -3.42 178 1.1,
                    -0.5 179.6 1.52,            -0.94 179.6 1.29,
                    -1.29 179.6 0.94,           -1.52 179.6 0.5,
                    1.21 -175 0.88,             -1.21 -175 -0.88 ]
}
        creaseAngle  1.5708
        solid FALSE
        coordIndex    [ 0, 1, 2, -1, 0, 3, 1, -1,
                        0, 4, 3, -1, 0, 5, 4, -1,
                        5, 6, 4, -1, 5, 7, 6, -1,
                        8, 9, 10, -1, 9, 11, 10, -1,
                        11, 12, 10, -1, 10, 12, 13, -1,
                        12, 14, 13, -1, 14, 15, 13, -1,
                        16, 17, 18, -1, 19, 20, 21, -1,
                        18, 20, 19, -1, 22, 23, 24, -1,
                        16, 25, 17, -1, 17, 26, 20, -1,
                        20, 26, 21, -1, 21, 27, 19, -1,
                        19, 28, 18, -1, 18, 29, 16, -1,
                        14, 12, 29, -1, 12, 26, 25, -1,
                        11, 9, 26, -1, 6, 22, 4, -1,
                        4, 30, 3, -1, 3, 23, 1, -1,
                        10, 13, 31, -1, 31, 32, 33, -1,
                        33, 5, 0, -1, 34, 35, 36, -1,
                        37, 36, 38, -1, 39, 37, 40, -1,
                        41, 23, 42, -1, 43, 42, 44, -1,
                        6, 43, 22, -1, 45, 46, 47, -1,
                        48, 46, 45, -1, 49, 50, 48, -1,
                        51, 50, 52, -1, 53, 52, 54, -1,
                        55, 53, 56, -1, 57, 55, 46, -1,
                        58, 46, 59, -1, 60, 59, 50, -1,
                        22, 44, 23, -1, 51, 57, 58, -1,
                        53, 55, 57, -1, 60, 51, 58, -1,
                        49, 40, 61, -1, 48, 40, 49, -1,
                        39, 40, 48, -1, 45, 39, 48, -1,
                        47, 39, 45, -1, 62, 39, 47, -1,
                        35, 6, 7, -1, 35, 43, 6, -1,
                        63, 43, 35, -1, 63, 41, 43, -1,
                        63, 1, 41, -1, 63, 2, 1, -1,
                        64, 65, 66, -1, 64, 67, 65, -1,
                        64, 68, 67, -1, 64, 69, 68, -1,
                        69, 70, 68, -1, 69, 71, 70, -1,
                        62, 47, 72, -1, 47, 73, 72, -1,
                        73, 74, 72, -1, 72, 74, 75, -1,
                        74, 49, 75, -1, 49, 61, 75, -1,
                        76, 77, 78, -1, 79, 80, 81, -1,
                        78, 80, 79, -1, 82, 83, 84, -1,
                        76, 85, 77, -1, 77, 86, 80, -1,
                        80, 86, 81, -1, 81, 56, 79, -1,
                        79, 52, 78, -1, 78, 50, 76, -1,
                        49, 74, 50, -1, 74, 86, 85, -1,
                        73, 47, 86, -1, 70, 82, 68, -1,
                        68, 87, 67, -1, 67, 83, 65, -1,
                        72, 75, 88, -1, 88, 89, 90, -1,
                        90, 69, 64, -1, 91, 92, 93, -1,
                        94, 93, 95, -1, 96, 94, 97, -1,
                        98, 83, 99, -1, 100, 99, 101, -1,
                        70, 100, 82, -1, 102, 103, 9, -1,
                        104, 103, 102, -1, 14, 29, 104, -1,
                        105, 29, 28, -1, 106, 28, 107, -1,
                        108, 106, 27, -1, 109, 108, 103, -1,
```

```
110, 103, 111, -1, 112, 111, 29, -1,
82, 101, 83, -1, 105, 109, 110, -1,
106, 108, 109, -1, 112, 105, 110, -1,
14, 97, 15, -1, 104, 97, 14, -1,
96, 97, 104, -1, 102, 96, 104, -1,
9, 96, 102, -1, 8, 96, 9, -1,
92, 70, 71, -1, 92, 100, 70, -1,
113, 100, 92, -1, 113, 98, 100, -1,
113, 65, 98, -1, 113, 66, 65, -1,
114, 115, 116, -1, 114, 116, 117, -1,
114, 117, 118, -1, 114, 118, 119, -1,
114, 119, 120, -1, 115, 121, 116, -1,
116, 122, 117, -1, 117, 123, 118, -1,
118, 123, 119, -1, 119, 124, 120, -1,
125, 126, 121, -1, 121, 127, 122, -1,
122, 128, 123, -1, 123, 129, 124, -1,
124, 130, 131, -1, 126, 132, 127, -1,
127, 133, 134, -1, 134, 135, 128, -1,
128, 135, 129, -1, 129, 136, 130, -1,
132, 137, 138, -1, 138, 139, 133, -1,
133, 140, 135, -1, 135, 140, 141, -1,
141, 142, 136, -1, 137, 143, 144, -1,
144, 145, 139, -1, 139, 146, 140, -1,
140, 147, 148, -1, 148, 149, 142, -1,
143, 150, 145, -1, 145, 151, 152, -1,
152, 153, 146, -1, 146, 153, 147, -1,
147, 154, 149, -1, 155, 156, 150, -1,
150, 157, 151, -1, 151, 158, 153, -1,
153, 158, 154, -1, 154, 159, 160, -1,
161, 162, 156, -1, 156, 163, 157, -1,
157, 164, 158, -1, 158, 164, 159, -1,
159, 165, 166, -1, 167, 168, 162, -1,
162, 169, 163, -1, 163, 170, 164, -1,
164, 170, 165, -1, 165, 171, 172, -1,
168, 173, 174, -1, 174, 175, 169, -1,
169, 176, 170, -1, 170, 177, 178, -1,
178, 179, 171, -1, 180, 181, 182, -1,
182, 183, 184, -1, 184, 185, 186, -1,
186, 187, 188, -1, 188, 189, 190, -1,
191, 192, 181, -1, 181, 193, 183, -1,
183, 194, 185, -1, 185, 195, 187, -1,
187, 196, 189, -1, 197, 198, 192, -1,
192, 199, 193, -1, 193, 200, 194, -1,
194, 201, 195, -1, 195, 202, 196, -1,
203, 204, 198, -1, 198, 205, 199, -1,
199, 206, 200, -1, 200, 207, 201, -1,
201, 208, 202, -1, 209, 210, 204, -1,
204, 211, 205, -1, 205, 212, 206, -1,
206, 213, 207, -1, 207, 214, 208, -1,
215, 216, 210, -1, 210, 217, 211, -1,
211, 218, 212, -1, 212, 219, 213, -1,
213, 220, 214, -1, 216, 221, 217, -1,
217, 222, 223, -1, 223, 224, 218, -1,
218, 225, 219, -1, 219, 226, 220, -1,
221, 227, 222, -1, 222, 228, 229, -1,
229, 230, 224, -1, 224, 231, 225, -1,
225, 232, 226, -1, 227, 233, 228, -1,
228, 234, 235, -1, 235, 236, 230, -1,
230, 236, 231, -1, 231, 237, 232, -1,
238, 239, 233, -1, 233, 239, 234, -1,
234, 240, 236, -1, 236, 241, 237, -1,
237, 241, 242, -1, 243, 10, 239, -1,
239, 10, 244, -1, 244, 33, 240, -1,
240, 0, 241, -1, 241, 0, 245, -1,
246, 247, 113, -1, 248, 113, 91, -1,
```

```
249, 91, 94, -1, 250, 249, 96, -1,
243, 96, 8, -1, 251, 252, 246, -1,
253, 246, 248, -1, 254, 248, 249, -1,
255, 254, 250, -1, 238, 250, 243, -1,
256, 257, 251, -1, 258, 256, 253, -1,
259, 253, 254, -1, 260, 254, 255, -1,
227, 255, 238, -1, 261, 257, 256, -1,
262, 256, 258, -1, 263, 258, 259, -1,
264, 263, 260, -1, 221, 264, 227, -1,
265, 266, 261, -1, 267, 261, 262, -1,
268, 262, 263, -1, 269, 268, 264, -1,
216, 269, 221, -1, 270, 271, 265, -1,
272, 265, 267, -1, 273, 267, 268, -1,
274, 273, 269, -1, 215, 274, 216, -1,
275, 276, 270, -1, 277, 270, 272, -1,
278, 272, 273, -1, 279, 273, 274, -1,
209, 274, 215, -1, 280, 281, 275, -1,
282, 275, 277, -1, 283, 277, 278, -1,
284, 278, 279, -1, 203, 279, 209, -1,
285, 286, 280, -1, 287, 280, 282, -1,
288, 282, 283, -1, 289, 283, 284, -1,
197, 284, 203, -1, 290, 291, 285, -1,
292, 285, 287, -1, 293, 287, 288, -1,
294, 288, 289, -1, 191, 289, 197, -1,
295, 296, 290, -1, 297, 290, 292, -1,
298, 292, 293, -1, 299, 293, 294, -1,
180, 294, 191, -1, 300, 301, 302, -1,
303, 302, 304, -1, 305, 304, 306, -1,
307, 306, 308, -1, 168, 308, 309, -1,
310, 311, 300, -1, 312, 310, 303, -1,
313, 303, 305, -1, 314, 305, 307, -1,
167, 314, 168, -1, 315, 316, 310, -1,
317, 315, 312, -1, 318, 312, 313, -1,
319, 313, 314, -1, 161, 314, 167, -1,
320, 321, 315, -1, 322, 320, 317, -1,
323, 317, 318, -1, 324, 318, 319, -1,
155, 319, 161, -1, 325, 326, 320, -1,
327, 325, 322, -1, 328, 322, 323, -1,
329, 323, 324, -1, 143, 324, 155, -1,
330, 326, 325, -1, 331, 325, 327, -1,
332, 327, 328, -1, 333, 332, 329, -1,
137, 333, 143, -1, 334, 335, 330, -1,
336, 334, 331, -1, 337, 331, 332, -1,
338, 332, 333, -1, 132, 338, 137, -1,
339, 340, 334, -1, 341, 339, 336, -1,
342, 336, 337, -1, 343, 337, 338, -1,
126, 343, 132, -1, 344, 345, 339, -1,
346, 339, 341, -1, 347, 341, 342, -1,
348, 347, 343, -1, 125, 348, 126, -1,
349, 350, 344, -1, 351, 349, 346, -1,
352, 346, 347, -1, 353, 347, 348, -1,
115, 348, 125, -1, 114, 350, 349, -1,
114, 349, 351, -1, 114, 351, 352, -1,
114, 352, 353, -1, 114, 353, 115, -1,
179, 354, 355, -1, 356, 357, 354, -1,
358, 359, 357, -1, 360, 361, 359, -1,
355, 362, 363, -1, 354, 364, 362, -1,
357, 365, 364, -1, 357, 366, 365, -1,
357, 359, 366, -1, 359, 367, 366, -1,
359, 368, 367, -1, 363, 369, 370, -1,
362, 371, 369, -1, 364, 372, 371, -1,
372, 373, 371, -1, 373, 374, 371, -1,
373, 367, 374, -1, 367, 375, 374, -1,
370, 376, 377, -1, 369, 378, 376, -1,
371, 379, 378, -1, 374, 380, 379, -1,
```

```
381, 377, 376, -1, 381, 376, 378, -1,
381, 378, 382, -1, 382, 378, 379, -1,
382, 379, 380, -1, 383, 367, 384, -1,
384, 373, 385, -1, 385, 372, 386, -1,
386, 365, 387, -1, 387, 366, 388, -1,
388, 367, 383, -1, 389, 384, 390, -1,
390, 384, 391, -1, 391, 385, 392, -1,
392, 387, 393, -1, 393, 388, 394, -1,
394, 383, 389, -1, 390, 394, 389, -1,
391, 394, 390, -1, 391, 392, 393, -1,
301, 295, 302, -1, 302, 297, 304, -1,
304, 298, 306, -1, 306, 299, 308, -1,
308, 180, 309, -1, 309, 182, 173, -1,
173, 184, 175, -1, 175, 186, 176, -1,
176, 188, 177, -1, 177, 356, 179, -1,
177, 358, 356, -1, 177, 188, 358, -1,
188, 360, 358, -1, 188, 190, 360, -1,
395, 396, 190, -1, 395, 397, 396, -1,
398, 397, 395, -1, 398, 399, 397, -1,
398, 179, 399, -1, 400, 395, 401, -1,
402, 401, 403, -1, 404, 403, 405, -1,
406, 405, 407, -1, 408, 407, 409, -1,
410, 409, 411, -1, 412, 411, 413, -1,
414, 413, 415, -1, 301, 415, 416, -1,
417, 418, 419, -1, 417, 420, 418, -1,
421, 422, 420, -1, 420, 423, 424, -1,
418, 424, 425, -1, 419, 425, 426, -1,
417, 419, 427, -1, 421, 417, 428, -1,
422, 428, 423, -1, 424, 429, 430, -1,
425, 430, 431, -1, 426, 431, 432, -1,
427, 426, 433, -1, 428, 427, 434, -1,
423, 428, 429, -1, 382, 380, 435, -1,
382, 435, 436, -1, 381, 382, 436, -1,
381, 436, 437, -1, 381, 437, 377, -1,
438, 380, 375, -1, 439, 435, 438, -1,
440, 436, 439, -1, 370, 437, 440, -1,
429, 375, 368, -1, 434, 438, 429, -1,
434, 439, 438, -1, 433, 439, 434, -1,
432, 439, 433, -1, 441, 439, 432, -1,
363, 440, 441, -1, 442, 368, 361, -1,
442, 430, 429, -1, 443, 430, 442, -1,
443, 431, 430, -1, 443, 432, 431, -1,
444, 432, 443, -1, 355, 441, 444, -1,
396, 361, 190, -1, 397, 442, 396, -1,
399, 443, 397, -1, 179, 444, 399, -1,
114, 445, 446, -1, 114, 446, 447, -1,
114, 447, 448, -1, 114, 448, 449, -1,
114, 449, 350, -1, 445, 450, 446, -1,
446, 451, 447, -1, 447, 452, 448, -1,
448, 452, 449, -1, 449, 453, 350, -1,
454, 455, 450, -1, 450, 456, 451, -1,
451, 457, 452, -1, 452, 458, 453, -1,
453, 345, 459, -1, 455, 460, 456, -1,
456, 461, 462, -1, 462, 463, 457, -1,
457, 463, 458, -1, 458, 340, 345, -1,
460, 464, 465, -1, 465, 466, 461, -1,
461, 467, 463, -1, 463, 467, 468, -1,
468, 335, 340, -1, 464, 469, 470, -1,
470, 471, 466, -1, 466, 472, 467, -1,
467, 473, 474, -1, 474, 326, 335, -1,
469, 475, 471, -1, 471, 476, 477, -1,
477, 478, 472, -1, 472, 478, 473, -1,
473, 479, 326, -1, 480, 481, 475, -1,
475, 482, 476, -1, 476, 483, 478, -1,
478, 483, 479, -1, 479, 484, 321, -1,
```

```
485, 486, 481, -1, 481, 487, 482, -1,
482, 488, 483, -1, 483, 488, 484, -1,
484, 489, 316, -1, 490, 491, 486, -1,
486, 492, 487, -1, 487, 493, 488, -1,
488, 493, 489, -1, 489, 311, 494, -1,
491, 408, 495, -1, 495, 410, 492, -1,
492, 412, 493, -1, 493, 414, 496, -1,
496, 301, 311, -1, 407, 497, 409, -1,
409, 498, 411, -1, 411, 499, 413, -1,
413, 500, 415, -1, 415, 296, 416, -1,
501, 502, 497, -1, 497, 503, 498, -1,
498, 504, 499, -1, 499, 505, 500, -1,
500, 291, 296, -1, 506, 507, 502, -1,
502, 508, 503, -1, 503, 509, 504, -1,
504, 510, 505, -1, 505, 286, 291, -1,
511, 512, 507, -1, 507, 513, 508, -1,
508, 514, 509, -1, 509, 515, 510, -1,
510, 281, 286, -1, 516, 517, 512, -1,
512, 518, 513, -1, 513, 519, 514, -1,
514, 520, 515, -1, 515, 276, 281, -1,
521, 522, 517, -1, 517, 523, 518, -1,
518, 524, 519, -1, 519, 525, 520, -1,
520, 271, 276, -1, 522, 526, 523, -1,
523, 527, 528, -1, 528, 529, 524, -1,
524, 530, 525, -1, 525, 266, 271, -1,
526, 531, 527, -1, 527, 532, 533, -1,
533, 534, 529, -1, 529, 535, 530, -1,
530, 257, 266, -1, 531, 536, 532, -1,
532, 537, 538, -1, 538, 539, 534, -1,
534, 539, 535, -1, 535, 540, 257, -1,
541, 542, 536, -1, 536, 542, 537, -1,
537, 543, 539, -1, 539, 544, 540, -1,
540, 544, 252, -1, 545, 72, 542, -1,
542, 72, 546, -1, 546, 90, 543, -1,
543, 64, 544, -1, 544, 64, 247, -1,
547, 245, 63, -1, 548, 63, 34, -1,
549, 34, 37, -1, 550, 549, 39, -1,
545, 39, 62, -1, 551, 242, 547, -1,
552, 547, 548, -1, 553, 548, 549, -1,
554, 553, 550, -1, 541, 550, 545, -1,
555, 232, 551, -1, 556, 555, 552, -1,
557, 552, 553, -1, 558, 553, 554, -1,
531, 554, 541, -1, 559, 232, 555, -1,
560, 555, 556, -1, 561, 556, 557, -1,
562, 561, 558, -1, 526, 562, 531, -1,
563, 226, 559, -1, 564, 559, 560, -1,
565, 560, 561, -1, 566, 565, 562, -1,
522, 566, 526, -1, 567, 220, 563, -1,
568, 563, 564, -1, 569, 564, 565, -1,
570, 569, 566, -1, 521, 570, 522, -1,
571, 214, 567, -1, 572, 567, 568, -1,
573, 568, 569, -1, 574, 569, 570, -1,
516, 570, 521, -1, 575, 208, 571, -1,
576, 571, 572, -1, 577, 572, 573, -1,
578, 573, 574, -1, 511, 574, 516, -1,
579, 202, 575, -1, 580, 575, 576, -1,
581, 576, 577, -1, 582, 577, 578, -1,
506, 578, 511, -1, 583, 196, 579, -1,
584, 579, 580, -1, 585, 580, 581, -1,
586, 581, 582, -1, 501, 582, 506, -1,
395, 189, 583, -1, 401, 583, 584, -1,
403, 584, 585, -1, 405, 585, 586, -1,
407, 586, 501, -1, 587, 179, 398, -1,
588, 398, 400, -1, 589, 400, 402, -1,
590, 402, 404, -1, 491, 404, 406, -1,
```

```
591, 171, 587, -1, 592, 591, 588, -1,
593, 588, 589, -1, 594, 589, 590, -1,
490, 594, 491, -1, 595, 166, 591, -1,
596, 595, 592, -1, 597, 592, 593, -1,
598, 593, 594, -1, 485, 594, 490, -1,
599, 160, 595, -1, 600, 599, 596, -1,
601, 596, 597, -1, 602, 597, 598, -1,
480, 598, 485, -1, 603, 149, 599, -1,
604, 603, 600, -1, 605, 600, 601, -1,
606, 601, 602, -1, 469, 602, 480, -1,
607, 149, 603, -1, 608, 603, 604, -1,
609, 604, 605, -1, 610, 609, 606, -1,
464, 610, 469, -1, 611, 142, 607, -1,
612, 611, 608, -1, 613, 608, 609, -1,
614, 609, 610, -1, 460, 614, 464, -1,
615, 136, 611, -1, 616, 615, 612, -1,
617, 612, 613, -1, 618, 613, 614, -1,
455, 618, 460, -1, 619, 130, 615, -1,
620, 615, 616, -1, 621, 616, 617, -1,
622, 621, 618, -1, 454, 622, 455, -1,
623, 120, 619, -1, 624, 623, 620, -1,
625, 620, 621, -1, 626, 621, 622, -1,
445, 622, 454, -1, 114, 120, 623, -1,
114, 623, 624, -1, 114, 624, 625, -1,
114, 625, 626, -1, 114, 626, 445, -1,
20, 18, 17, -1, 24, 23, 30, -1,
25, 16, 29, -1, 26, 17, 25, -1,
21, 26, 27, -1, 19, 27, 28, -1,
28, 19, 107, -1, 29, 18, 28, -1,
29, 12, 25, -1, 26, 12, 11, -1,
26, 9, 27, -1, 4, 22, 24, -1,
30, 4, 24, -1, 23, 3, 30, -1,
31, 13, 627, -1, 32, 31, 627, -1,
5, 33, 32, -1, 35, 34, 63, -1,
36, 37, 34, -1, 40, 37, 38, -1,
23, 41, 1, -1, 42, 43, 41, -1,
22, 43, 44, -1, 47, 46, 56, -1,
46, 48, 59, -1, 48, 50, 59, -1,
50, 51, 60, -1, 52, 53, 51, -1,
56, 53, 52, -1, 46, 55, 56, -1,
46, 58, 57, -1, 59, 60, 58, -1,
23, 44, 42, -1, 57, 51, 53, -1,
80, 78, 77, -1, 84, 83, 87, -1,
85, 76, 50, -1, 86, 77, 85, -1,
81, 86, 56, -1, 79, 56, 52, -1,
52, 79, 54, -1, 50, 78, 52, -1,
50, 74, 85, -1, 86, 74, 73, -1,
86, 47, 56, -1, 68, 82, 84, -1,
87, 68, 84, -1, 83, 67, 87, -1,
88, 75, 628, -1, 89, 88, 628, -1,
69, 90, 89, -1, 92, 91, 113, -1,
93, 94, 91, -1, 97, 94, 95, -1,
83, 98, 65, -1, 99, 100, 98, -1,
82, 100, 101, -1, 9, 103, 27, -1,
103, 104, 111, -1, 104, 29, 111, -1,
29, 105, 112, -1, 28, 106, 105, -1,
27, 106, 28, -1, 103, 108, 27, -1,
103, 110, 109, -1, 111, 112, 110, -1,
83, 101, 99, -1, 109, 105, 106, -1,
121, 115, 125, -1, 122, 116, 121, -1,
123, 117, 122, -1, 119, 123, 124, -1,
120, 124, 131, -1, 121, 126, 127, -1,
122, 127, 134, -1, 128, 122, 134, -1,
129, 123, 128, -1, 130, 124, 129, -1,
127, 132, 138, -1, 133, 127, 138, -1,
```

```
135, 134, 133, -1, 129, 135, 141, -1,
136, 129, 141, -1, 138, 137, 144, -1,
139, 138, 144, -1, 140, 133, 139, -1,
141, 140, 148, -1, 142, 141, 148, -1,
144, 143, 145, -1, 139, 145, 152, -1,
146, 139, 152, -1, 147, 140, 146, -1,
149, 148, 147, -1, 150, 143, 155, -1,
151, 145, 150, -1, 153, 152, 151, -1,
147, 153, 154, -1, 149, 154, 160, -1,
156, 155, 161, -1, 157, 150, 156, -1,
158, 151, 157, -1, 154, 158, 159, -1,
160, 159, 166, -1, 162, 161, 167, -1,
163, 156, 162, -1, 164, 157, 163, -1,
159, 164, 165, -1, 166, 165, 172, -1,
162, 168, 174, -1, 169, 162, 174, -1,
170, 163, 169, -1, 165, 170, 178, -1,
171, 165, 178, -1, 173, 168, 309, -1,
175, 174, 173, -1, 176, 169, 175, -1,
177, 170, 176, -1, 179, 178, 177, -1,
181, 180, 191, -1, 183, 182, 181, -1,
185, 184, 183, -1, 187, 186, 185, -1,
189, 188, 187, -1, 192, 191, 197, -1,
193, 181, 192, -1, 194, 183, 193, -1,
195, 185, 194, -1, 196, 187, 195, -1,
198, 197, 203, -1, 199, 192, 198, -1,
200, 193, 199, -1, 201, 194, 200, -1,
202, 195, 201, -1, 204, 203, 209, -1,
205, 198, 204, -1, 206, 199, 205, -1,
207, 200, 206, -1, 208, 201, 207, -1,
210, 209, 215, -1, 211, 204, 210, -1,
212, 205, 211, -1, 213, 206, 212, -1,
214, 207, 213, -1, 210, 216, 217, -1,
211, 217, 223, -1, 218, 211, 223, -1,
219, 212, 218, -1, 220, 213, 219, -1,
217, 221, 222, -1, 223, 222, 229, -1,
224, 223, 229, -1, 225, 218, 224, -1,
226, 219, 225, -1, 222, 227, 228, -1,
229, 228, 235, -1, 230, 229, 235, -1,
231, 224, 230, -1, 232, 225, 231, -1,
233, 227, 238, -1, 234, 228, 233, -1,
236, 235, 234, -1, 231, 236, 237, -1,
232, 237, 242, -1, 239, 238, 243, -1,
234, 239, 244, -1, 240, 234, 244, -1,
241, 236, 240, -1, 242, 241, 245, -1,
10, 243, 8, -1, 244, 10, 31, -1,
33, 244, 31, -1, 0, 240, 33, -1,
245, 0, 2, -1, 113, 247, 66, -1,
113, 248, 246, -1, 91, 249, 248, -1,
96, 249, 94, -1, 96, 243, 250, -1,
246, 252, 247, -1, 246, 253, 251, -1,
248, 254, 253, -1, 250, 254, 249, -1,
250, 238, 255, -1, 251, 257, 252, -1,
253, 256, 251, -1, 253, 259, 258, -1,
254, 260, 259, -1, 255, 227, 260, -1,
257, 261, 266, -1, 256, 262, 261, -1,
258, 263, 262, -1, 260, 263, 259, -1,
227, 264, 260, -1, 266, 265, 271, -1,
261, 267, 265, -1, 262, 268, 267, -1,
264, 268, 263, -1, 221, 269, 264, -1,
271, 270, 276, -1, 265, 272, 270, -1,
267, 273, 272, -1, 269, 273, 268, -1,
216, 274, 269, -1, 276, 275, 281, -1,
270, 277, 275, -1, 272, 278, 277, -1,
273, 279, 278, -1, 274, 209, 279, -1,
281, 280, 286, -1, 275, 282, 280, -1,
```

```
277, 283, 282, -1, 278, 284, 283, -1,
279, 203, 284, -1, 286, 285, 291, -1,
280, 287, 285, -1, 282, 288, 287, -1,
283, 289, 288, -1, 284, 197, 289, -1,
291, 290, 296, -1, 285, 292, 290, -1,
287, 293, 292, -1, 288, 294, 293, -1,
289, 191, 294, -1, 296, 295, 416, -1,
290, 297, 295, -1, 292, 298, 297, -1,
293, 299, 298, -1, 294, 180, 299, -1,
301, 300, 311, -1, 302, 303, 300, -1,
304, 305, 303, -1, 306, 307, 305, -1,
308, 168, 307, -1, 311, 310, 494, -1,
303, 310, 300, -1, 303, 313, 312, -1,
305, 314, 313, -1, 168, 314, 307, -1,
310, 316, 494, -1, 312, 315, 310, -1,
312, 318, 317, -1, 313, 319, 318, -1,
314, 161, 319, -1, 315, 321, 316, -1,
317, 320, 315, -1, 317, 323, 322, -1,
318, 324, 323, -1, 319, 155, 324, -1,
320, 326, 321, -1, 322, 325, 320, -1,
322, 328, 327, -1, 323, 329, 328, -1,
324, 143, 329, -1, 326, 330, 335, -1,
325, 331, 330, -1, 327, 332, 331, -1,
329, 332, 328, -1, 143, 333, 329, -1,
335, 334, 340, -1, 331, 334, 330, -1,
331, 337, 336, -1, 332, 338, 337, -1,
137, 338, 333, -1, 340, 339, 345, -1,
336, 339, 334, -1, 336, 342, 341, -1,
337, 343, 342, -1, 132, 343, 338, -1,
345, 344, 459, -1, 339, 346, 344, -1,
341, 347, 346, -1, 343, 347, 342, -1,
126, 348, 343, -1, 344, 350, 459, -1,
346, 349, 344, -1, 346, 352, 351, -1,
347, 353, 352, -1, 348, 115, 353, -1,
354, 179, 356, -1, 357, 356, 358, -1,
359, 358, 360, -1, 361, 360, 190, -1,
362, 355, 354, -1, 364, 354, 357, -1,
368, 359, 361, -1, 369, 363, 362, -1,
371, 362, 364, -1, 375, 367, 368, -1,
376, 370, 369, -1, 378, 369, 371, -1,
379, 371, 374, -1, 380, 374, 375, -1,
384, 367, 373, -1, 385, 373, 372, -1,
386, 372, 364, -1, 365, 386, 364, -1,
366, 387, 365, -1, 367, 388, 366, -1,
384, 389, 383, -1, 391, 384, 385, -1,
392, 385, 386, -1, 387, 392, 386, -1,
388, 393, 387, -1, 383, 394, 388, -1,
394, 391, 393, -1, 295, 301, 416, -1,
297, 302, 295, -1, 298, 304, 297, -1,
299, 306, 298, -1, 180, 308, 299, -1,
182, 309, 180, -1, 184, 173, 182, -1,
186, 175, 184, -1, 188, 176, 186, -1,
395, 400, 398, -1, 401, 402, 400, -1,
403, 404, 402, -1, 405, 406, 404, -1,
407, 408, 406, -1, 409, 410, 408, -1,
411, 412, 410, -1, 413, 414, 412, -1,
415, 301, 414, -1, 420, 417, 421, -1,
423, 420, 422, -1, 424, 418, 420, -1,
425, 419, 418, -1, 427, 419, 426, -1,
428, 417, 427, -1, 428, 422, 421, -1,
429, 424, 423, -1, 430, 425, 424, -1,
431, 426, 425, -1, 433, 426, 432, -1,
434, 427, 433, -1, 429, 428, 434, -1,
380, 438, 435, -1, 435, 439, 436, -1,
436, 440, 437, -1, 437, 370, 377, -1,
```

```
375, 429, 438, -1, 439, 441, 440, -1,
440, 363, 370, -1, 368, 442, 429, -1,
432, 444, 441, -1, 441, 355, 363, -1,
361, 396, 442, -1, 442, 397, 443, -1,
443, 399, 444, -1, 444, 179, 355, -1,
450, 445, 454, -1, 451, 446, 450, -1,
452, 447, 451, -1, 449, 452, 453, -1,
350, 453, 459, -1, 450, 455, 456, -1,
451, 456, 462, -1, 457, 451, 462, -1,
458, 452, 457, -1, 345, 453, 458, -1,
456, 460, 465, -1, 461, 456, 465, -1,
463, 462, 461, -1, 458, 463, 468, -1,
340, 458, 468, -1, 465, 464, 470, -1,
466, 465, 470, -1, 467, 461, 466, -1,
468, 467, 474, -1, 335, 468, 474, -1,
470, 469, 471, -1, 466, 471, 477, -1,
472, 466, 477, -1, 473, 467, 472, -1,
326, 474, 473, -1, 475, 469, 480, -1,
476, 471, 475, -1, 478, 477, 476, -1,
473, 478, 479, -1, 326, 479, 321, -1,
481, 480, 485, -1, 482, 475, 481, -1,
483, 476, 482, -1, 479, 483, 484, -1,
321, 484, 316, -1, 486, 485, 490, -1,
487, 481, 486, -1, 488, 482, 487, -1,
484, 488, 489, -1, 316, 489, 494, -1,
486, 491, 495, -1, 492, 486, 495, -1,
493, 487, 492, -1, 489, 493, 496, -1,
311, 489, 496, -1, 408, 491, 406, -1,
410, 495, 408, -1, 412, 492, 410, -1,
414, 493, 412, -1, 301, 496, 414, -1,
497, 407, 501, -1, 498, 409, 497, -1,
499, 411, 498, -1, 500, 413, 499, -1,
296, 415, 500, -1, 502, 501, 506, -1,
503, 497, 502, -1, 504, 498, 503, -1,
505, 499, 504, -1, 291, 500, 505, -1,
507, 506, 511, -1, 508, 502, 507, -1,
509, 503, 508, -1, 510, 504, 509, -1,
286, 505, 510, -1, 512, 511, 516, -1,
513, 507, 512, -1, 514, 508, 513, -1,
515, 509, 514, -1, 281, 510, 515, -1,
517, 516, 521, -1, 518, 512, 517, -1,
519, 513, 518, -1, 520, 514, 519, -1,
276, 515, 520, -1, 517, 522, 523, -1,
518, 523, 528, -1, 524, 518, 528, -1,
525, 519, 524, -1, 271, 520, 525, -1,
523, 526, 527, -1, 528, 527, 533, -1,
529, 528, 533, -1, 530, 524, 529, -1,
266, 525, 530, -1, 527, 531, 532, -1,
533, 532, 538, -1, 534, 533, 538, -1,
535, 529, 534, -1, 257, 530, 535, -1,
536, 531, 541, -1, 537, 532, 536, -1,
539, 538, 537, -1, 535, 539, 540, -1,
257, 540, 252, -1, 542, 541, 545, -1,
537, 542, 546, -1, 543, 537, 546, -1,
544, 539, 543, -1, 252, 544, 247, -1,
72, 545, 62, -1, 546, 72, 88, -1,
90, 546, 88, -1, 64, 543, 90, -1,
247, 64, 66, -1, 63, 245, 2, -1,
63, 548, 547, -1, 34, 549, 548, -1,
39, 549, 37, -1, 39, 545, 550, -1,
547, 242, 245, -1, 547, 552, 551, -1,
548, 553, 552, -1, 550, 553, 549, -1,
550, 541, 554, -1, 551, 232, 242, -1,
552, 555, 551, -1, 552, 557, 556, -1,
553, 558, 557, -1, 554, 531, 558, -1,
```

```
                                    232, 559, 226, -1, 555, 560, 559, -1,
                                    556, 561, 560, -1, 558, 561, 557, -1,
                                    531, 562, 558, -1, 226, 563, 220, -1,
                                    559, 564, 563, -1, 560, 565, 564, -1,
                                    562, 565, 561, -1, 526, 566, 562, -1,
                                    220, 567, 214, -1, 563, 568, 567, -1,
                                    564, 569, 568, -1, 566, 569, 565, -1,
                                    522, 570, 566, -1, 214, 571, 208, -1,
                                    567, 572, 571, -1, 568, 573, 572, -1,
                                    569, 574, 573, -1, 570, 516, 574, -1,
                                    208, 575, 202, -1, 571, 576, 575, -1,
                                    572, 577, 576, -1, 573, 578, 577, -1,
                                    574, 511, 578, -1, 202, 579, 196, -1,
                                    575, 580, 579, -1, 576, 581, 580, -1,
                                    577, 582, 581, -1, 578, 506, 582, -1,
                                    196, 583, 189, -1, 579, 584, 583, -1,
                                    580, 585, 584, -1, 581, 586, 585, -1,
                                    582, 501, 586, -1, 189, 395, 190, -1,
                                    583, 401, 395, -1, 584, 403, 401, -1,
                                    585, 405, 403, -1, 586, 407, 405, -1,
                                    179, 587, 171, -1, 398, 588, 587, -1,
                                    400, 589, 588, -1, 402, 590, 589, -1,
                                    404, 491, 590, -1, 171, 591, 172, -1,
                                    588, 591, 587, -1, 588, 593, 592, -1,
                                    589, 594, 593, -1, 491, 594, 590, -1,
                                    591, 166, 172, -1, 592, 595, 591, -1,
                                    592, 597, 596, -1, 593, 598, 597, -1,
                                    594, 485, 598, -1, 595, 160, 166, -1,
                                    596, 599, 595, -1, 596, 601, 600, -1,
                                    597, 602, 601, -1, 598, 480, 602, -1,
                                    599, 149, 160, -1, 600, 603, 599, -1,
                                    600, 605, 604, -1, 601, 606, 605, -1,
                                    602, 469, 606, -1, 149, 607, 142, -1,
                                    603, 608, 607, -1, 604, 609, 608, -1,
                                    606, 609, 605, -1, 469, 610, 606, -1,
                                    142, 611, 136, -1, 608, 611, 607, -1,
                                    608, 613, 612, -1, 609, 614, 613, -1,
                                    464, 614, 610, -1, 136, 615, 130, -1,
                                    612, 615, 611, -1, 612, 617, 616, -1,
                                    613, 618, 617, -1, 460, 618, 614, -1,
                                    130, 619, 131, -1, 615, 620, 619, -1,
                                    616, 621, 620, -1, 618, 621, 617, -1,
                                    455, 622, 618, -1, 619, 120, 131, -1,
                                    620, 623, 619, -1, 620, 625, 624, -1,
                                    621, 626, 625, -1, 622, 445, 626, -1 ]
                    colorIndex    -1
                    normalIndex   -1
                }

            }

        }
    ]

},
#The Propeller
 Transform {
            translation 178.5 -7.5 9.5
              rotation    0.0 0.0 1.0 1.571
            scale       2.5 2.5 2.5
            children [
    DEF propeller_movement Transform {
            rotation 1.0 0.0 0.0 0.0
            children
```

```
                    Group {
                    children [
                          #The center hub of the propeller
                          Shape {
                          appearance DEF Bronze Appearance {
                                material Material {
                                            diffuseColor 1.0 1.0 0.0
                                 }  #end material
                          }    #end appearence
                          geometry Cylinder {
                                       radius 0.7
                                       height 0.5
                          }  # end geometry
                    },    #end shape

                          #Blade 1, oriented to stick out of the  right side of the
hub
                          DEF Blade Transform  {
                          rotation       1.0   0.0   0.0  1.048
                          translation    2.0   0.0   0.0
                          scale          2.0   0.1  0.5
                          children   Shape {
                                appearance  USE Bronze
                                geometry Sphere {}
                          }   #end shape

                          },        #end transform
                          #Blade 2
                          Transform {
                          rotation 0.0 1.0 0.0 1.26
                          children USE Blade
                          },
                          #Blade 3
                          Transform {
                          rotation 0.0 1.0 0.0 2.52
                          children USE Blade
                    },

                          #Blade 4
                          Transform {
                          rotation 0.0 1.0 0.0 3.78
                          children USE Blade
                          },

                          #Blade 5
                          Transform {
                          rotation 0.0 1.0 0.0 5.04
                          children USE Blade
                          }

                     ]      # end of children in group
               }           # end of Propeller Group
               },     #end of propeller_position Transform
                  DEF Blade_Clock TimeSensor {
                      cycleInterval 3.0
                       startTime     1.0
                      loop          TRUE
               },
               DEF Blade_Path OrientationInterpolator {
                      key [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
                      keyValue [0.0 1.0 0.0  0.0,
                               0.0 1.0 0.0 0.628,
                               0.0 1.0 0.0 1.256,
                               0.0 1.0 0.0 1.884,
                               0.0 1.0 0.0 2.512,
```

```
                        0.0 1.0 0.0 3.14,
                        0.0 1.0 0.0 3.768,
                        0.0 1.0 0.0 4.396,
                        0.0 1.0 0.0 5.024,
                        0.0 1.0 0.0 5.652 ]

        }
    ]

    }
     #The Vortex Dissipater
     Transform {
          translation 182.5 -7.5 9.5
          rotation    0.0 0.0 1.0 -1.571
          children Shape {
                 appearance Appearance  {
                 material USE _688_Hull
          }
             geometry Cone {
                 height 6.0
                  bottomRadius 1.7
          }
          }
    },

    #This section creates the periscope
    Transform {
          translation -76.0 32.0 10.0
          children [
                 Shape  {
                        appearance Appearance {
                               material  DEF Scope_color Material {
                                      diffuseColor   0.75 0.75 0.75
                               }
                        }
                        geometry Cylinder {
                               radius 0.2
                               height 9.0
                        }
                 },
                 Transform {
                        translation -0.2 4.3 0.0
                        scale 1.2 1.0 0.80
                        children Shape  {
                               appearance Appearance {
                                      material USE Scope_color
                               }
                               geometry Cylinder {
                                      radius 0.3
                                      height 0.4
                               }
                        }
                 }
          ]
    }
    ]
}

ROUTE Blade_Clock.fraction_changed TO Blade_Path.set_fraction
ROUTE Blade_Path.value_changed TO propeller_movement.set_rotation
```

5. Phoenix_auv.wrl

```
#VRML V2.0 utf8

#Model of the Naval Postgraduate School Center for Autonomous
#  Underwater Vehicle (AUV) Research's "Phoenix" AUV.
#  Authors: Martin Whitfield, Don Brutzman, Kevin Byrne

Viewpoint {
    position 0 0 2
    orientation 0 1 0 0
    description "Stbd Beam"
}

Viewpoint {
    position 2 0 2
    orientation 0 1 0 .707
    description "Stbd Bow"
}

Viewpoint {
    position 2 0 0
    orientation 0 1 0 1.4
    description "Bow"
}

Viewpoint {
    position 2 0 -2
    orientation 0 1 0 2.3562
    description "Port Bow"
}

Viewpoint {
    position 0 0 -2
    orientation 0 1 0 3.14159267
    description "Port Beam"
}

Viewpoint {
    position -2 0 -2
    orientation 0 1 0 3.9270
    description "Astern Port"
}

Viewpoint {
    position -2 0 0
    orientation 0 1 0 -1.4
    description "Astern"
}

Viewpoint {
    position -2 0 2
    orientation 0 1 0 -.707
    description "Astern Stbd"
}

Viewpoint {
    position 0 0 2
    orientation 0 1 0 0
    description "Stbd Beam"
}

Viewpoint {
    position 0 1.5 1.5
    orientation 1 0 0 -.707
```

```
    description "Above Stbd Beam"
}

Viewpoint {
    position 2 2 2
    orientation -.6786 .6786 -.2811 1.0961
    description "Above Stbd Bow"
}

Viewpoint {
    position 2 2 0
    orientation -.3574 .8629 .3574 1.7178
    description "Above Bow"
}

Viewpoint {
    position 2 2 -2
    orientation -.3780 .9125 .1566 2.4189
    description "Above Port Bow"
}

Viewpoint {
    position 0 2 -2
    orientation 1 0 0 3.9270
    description "Above Port Beam"
}

Viewpoint {
    position -2 2 -2
    orientation -.1566 .9125 .3780 3.8643
    description "Above Astern Port"
}

Viewpoint {
    position -2 2 0
    orientation  .3574 .8629 .3574 4.5654
    description "Above Stern"
}

Viewpoint {
    position -1.5 0 0
    orientation 0 1 0 -1.4
    description "Close Astern"
}

Viewpoint {
    position -2 0 1
    orientation 0 1 0 -.707
    description "Close Astern Stbd"
}


Viewpoint {
    position -1.11 0 .5
    orientation 0 1 0 0
    description "Close Stbd Stern"
}

Viewpoint {
    position -1.11 0 2
    orientation 0 1 0 0
    description "Stbd Stern"
}

Viewpoint {
```

```
    position 0 -1.5 1.5
    orientation 1 0 0 .707
    description "Below Stbd Beam"
}

NavigationInfo {
  type [ "EXAMINE" "ALL" ]
}

Group {             # DEF AUV
  children [

    # Fwd Top Plane
    Transform{
      translation .6223 .13335 0
      children[

        #A Plane Shape
        DEF A_Plane Shape{
          appearance Appearance{
            material Material {diffuseColor  .3 .2 0}
          } #end Appearance
          geometry IndexedFaceSet {
            coord Coordinate{
              point[ .0635,  0,     -.0127,   #0
                     .0381,  .1778, -.0127,   #1
                    -.0381,  .1778, -.0127,   #2
                    -.0889,  0,     -.0127,   #3

                     .0635,  0,      .0127,   #4
                     .0381,  .1778,  .0127,   #5
                    -.0381,  .1778,  .0127,   #6
                    -.0889,  0,      .0127    #7
              ] #end Points
            } #end Coordinates

            coordIndex[ 0, 3, 2, 1, -1,
                        4, 5, 6, 7, -1,
                        0, 1, 5, 4, -1,
                        1, 2, 6, 5, -1,
                        2, 3, 7, 6, -1,
                        0, 3, 7, 4, -1
            ] #end coordIndex
            creaseAngle 3.14159
          } #end IndexedFaceSet
        } #end Shape
      ] #end Transform children
    } #end Transform

    # Aft Top Plane
    Transform{
      translation -.7747 .13335 0
      children[
        USE A_Plane
      ]# end Transform children
    }#end Transform

    #Fwd Bottom Plane
    Transform{
      rotation 1 0 0 3.14159267
      translation .6223 -.13335 0
      children[
        USE A_Plane
      ]#end Transform children
    }#end Transform
```

```
#Aft Bottom Plane
Transform{
  rotation 1 0 0 3.14159267
  translation -.7747 -.13335 0
  children[
    USE A_Plane
  ]#end Transform children
}#end Transform

#Stbd Fwd Plane
Transform{
  rotation 1 0 0 1.5708
  translation .6223 0 .20955
  children[
    USE A_Plane
  ]#end Transform children
}#end Transform

#Stbd Aft Plane
Transform{
  rotation 1 0 0 1.5708
  translation -.7747 0 .20955
  children[
    USE A_Plane
  ]#end Transform children
}#end Transform

#Port Fwd Plane
Transform{
  rotation 1 0 0 -1.5708
  translation .6223 0 -.20955
  children[
    USE A_Plane
  ]#end Transform children
}#end Transform

#Port Aft Plane
Transform{
  rotation 1 0 0 -1.5708
  translation -.7747 0 -.20955
  children[
    USE A_Plane
  ]#end Transform children
}#end Transform

#Fwd Vert Thruster
Transform{
  translation .3302 0 0
  children[
    Shape{
      appearance Appearance{
        material Material {diffuseColor .2 .2 .2}
      }
      geometry Cylinder {height .29 radius .0635} # {height .2737 radius .0635}
    }#end Shape
  ]#end Children
}#end Transform

#Aft Vert Thruster
Transform{
  translation -.4953 0 0
  children[
    Shape{
      appearance Appearance{
        material Material {diffuseColor .2 .2 .2}
```

```
      }
      geometry Cylinder {height .29 radius .0635} # {height .2737 radius .0635}
    }#end Shape
  ]#end Children
}#end Transform

#Fwd Horiz Thruster
Transform{
  rotation 1 0 0 1.5708
  translation .4699 0 0
  children[
    Shape{
      appearance Appearance{
        material Material {diffuseColor .2 .2 .2}
      }
      geometry Cylinder {height .44 radius .0635} # {height .4231 radius .0635}
    }#end Shape
  ]#end Children
}#end Transform


#Aft Horiz Thruster
Transform{
  rotation 1 0 0 1.5708
  translation -.6223 0 0
  children[
    Shape{
      appearance Appearance{
        material Material {diffuseColor .2 .2 .2}
      }
      geometry Cylinder {height .44 radius .0635} # {height .4231 radius .0635}
    } #end Shape
  ] #end Children
} #end Transform

#Hull
Group{
  children[

    #Bow Cowling
    Shape{
      appearance Appearance{
        material Material {diffuseColor  0 0 .8}
      } #end Appearance

      geometry IndexedFaceSet {
        coord Coordinate{
          point[ .6985,  .13335, -.20955,  #0 Start of Bow Cowling
                 .6985,  .13335,  .20955,  #1
                 .6985, -.13335,  .20955,  #2
                 .6985, -.13335, -.20955,  #3

                1.05,   .085,    0,        #4
                1.05,   0,      .1143,     #5
                1.05,  -.085,    0,        #6
                1.05,   0,     -.1143,     #7

                1.05,   .04572,  -.098985,  #8
                1.05,   .079188, -.05715,   #9
                1.05,   .079188,  .05715,   #10
                1.05,   .04572,   .098985,  #11

                1.05,  -.04572,   .098985,  #12
                1.05,  -.079188,  .05715,   #13
                1.05,  -.079188, -.05715,   #14
```

```
          1.05, -.04572,  -.098985,  #15

          1.1,    .04064,   .02032,  #16
          1.1,    .02032,   .06096,  #17
          1.1,   -.02032,   .06096,  #18
          1.1,   -.04064,   .02032,  #19
          1.1,   -.04064,  -.02032,  #20
          1.1,   -.02032,  -.06096,  #21
          1.1,    .02032,  -.06096,  #22
          1.1,    .04064,  -.02032,  #23

          1.11,  0,        0,        #24

          -.6985,  .13335,  .20955,  #25 Start of Stern Cowling
          -.6985,  .13335, -.20955,  #26
          -.6985, -.13335, -.20955,  #27
          -.6985, -.13335,  .20955,  #28
          -1.1303,  0,      .20955,  #29
          -1.1303,  0,     -.20955,  #30

         -.6985,   .13335,    .0635,  #31 Start of Rudder Post
         -.8509,   .13335,    .0635,  #32
         -.8509,   .13335,   -.0635,  #33
         -.6985,   .13335,   -.0635,  #34
         -.6985,  -.13335,    .0635,  #35
         -.8509,  -.13335,    .0635,  #36
         -.8509,  -.13335,   -.0635,  #37
         -.6985,  -.13335,   -.0635,  #38

  ] #end Points
} #end Coordinates

coordIndex[ 0, 26, 34, 33, 32, 31, 25, 1, -1, #Hull
            1, 25, 29, 28, 2,  -1,
            2, 28, 35, 36, 37, 38, 27, 3, -1,
            0, 3,  27, 30, 26, -1,

            0, 4, 1, -1,   #Bow Cowling
            0, 1, 4, -1,
            1, 5, 2, -1,
            1, 2, 5, -1,
            2, 6, 3, -1,
            2, 3, 6, -1,
            3, 7, 0, -1,
            3, 0, 7, -1,

            7, 0, 8, -1,
            7, 8, 0, -1,
            8, 0, 9, -1,
            8, 9, 0, -1,
            9, 0, 4, -1,
            9, 4, 0, -1,

            4,  1, 10, -1,
            4, 10,  1, -1,
           10,  1, 11, -1,
           10, 11,  1, -1,
           11,  1,  5, -1,
           11,  5,  1, -1,

            5,  2, 12,  -1,
            5, 12,  2,  -1,
           12,  2, 13, -1,
           12, 13,  2, -1,
           13,  2,  6,  -1,
```

```
                    13,  6,  2,  -1,

                     6,  3, 14,  -1,
                     6, 14,  3,  -1,
                    14,  3, 15,  -1,
                    14, 15,  3,  -1,
                    15,  3,  7,  -1
                    15,  7,  3,  -1,

                    4, 10, 16, -1,
                    10, 11, 16, -1,
                    11, 5, 17,  -1,
                    5, 12, 18, -1,
                    12, 13, 19, -1,
                    13, 6, 19, -1,
                    6, 14, 20, -1,
                    14, 15, 20, -1,
                    15, 7, 21, -1,
                    7, 8, 22, -1,
                    8, 9, 23, -1,
                    9, 4, 23, -1,

                    4, 16, 23, -1,
                    11, 17, 16,  -1,
                    5, 18, 17, -1,
                    12, 19, 18,  -1,
                    6, 20, 19, -1,
                    20, 15, 21,  -1,
                    21, 7, 22,  -1,
                    22, 8, 23,  -1,

                    23, 16, 24,  -1,
                    16, 17, 24,  -1,
                    17, 18, 24,  -1,
                    18, 19, 24,  -1,
                    19, 20, 24,  -1,
                    20, 21, 24,  -1,
                    21, 22, 24,  -1,
                    22, 23, 24,  -1,

                    26, 27, 30, -1,      #Start of Stern Cowling
                    25, 26, 30, 29, -1,
                    25, 29, 28, -1,
                    27, 28, 29, 30, -1,

                    31, 32, 36, 35, -1, #Start of Rudder Post
                    32, 33, 37, 36, -1,
                    34, 38, 37, 33, -1,

        ] #end coordIndex
      creaseAngle 3.14159

      } #end IndexedFaceSet
    } #end Shape

  ] #end Hull Group Children
} #end Hull Group

#The Stbd screw
Transform{
  translation -1.1557 0 .09525
  children[
    Group{              # DEF Stbd_Screw
      children[
```

```
DEF Stbd_Blade Group{
children[
  Transform{
     rotation 0 1 0 -.39
     children[
        Shape{
          appearance Appearance{
            material Material {diffuseColor  .226 .197 0}
          } #end Appearance

          geometry IndexedFaceSet {
            coord Coordinate{
              point[ 0,  0,      -.00508,    #0
                     0,  .02540,  -.02032,    #1
                     0,  .04572,  -.01524,    #2
                     0,  .05080,  -.00508,    #3
                     0,  .05080,   .00508,    #4
                     0,  .04572,   .01524,    #5
                     0,  .02540,   .02032,    #6
                     0,  0,       .00508      #7
              ] #end Points
            } #end Coordinates

            coordIndex[ 0, 1, 2, 3, 4, 5, 6, 7, -1,
                        0, 7, 6, 5, 4, 3, 2, 1, -1
            ] #end coordIndex
          } #end IndexedFaceSet
        } #end A_Blade Shape
        ]#end transform children
      } #end transform
    ] #end group children
   } #end A_Blade Group

        Transform{
          rotation 1 0 0 1.5708
          children[ USE Stbd_Blade ]
        } #end Transform

        Transform{
         rotation 1 0 0 3.14159267
          children[ USE Stbd_Blade ]
        } #end Transform

        Transform{
          rotation 1 0 0 -1.5708
          children[ USE Stbd_Blade ]
        } #end Transform

        #The shaft
        Transform{
          rotation 0 0 1 1.5708
          translation .0281 0 0
          children[
            Shape {
              appearance Appearance{
                material Material {diffuseColor  .226 .197 0}
              } #end Appearance
              geometry Cylinder {radius .008 height .0762}
            } #end Shape
          ] #end children
        } #end Transform

        #The shaft end cap
        Transform{
          rotation 0 0 1 1.5708
```

```
          translation -.015 0 0
          children[
            Shape {
              appearance Appearance{
                material Material {diffuseColor  .226 .197 0}
              } #end Appearance
              geometry Cone {bottomRadius .008 height .01}
            } #end Shape
          ] #end children
        } #end Transform
      ] #end Screw Group Children
    } #end Screw Group
  ] #end Transform Children
 } #end Transform

#The Port screw
 Transform{
   translation -1.1557 0 -.09525
   children[
     Group{              # DEF Port_Screw
       children[

 DEF Port_Blade Group{
 children[
   Transform{
      rotation 0 1 0 .39
      children[
         Shape{
           appearance Appearance{
             material Material {diffuseColor  .226 .197 0}
           } #end Appearance

           geometry IndexedFaceSet {
             coord Coordinate{
               point[ 0,  0,       -.00508,   #0
                      0,  .02540,  -.02032,   #1
                      0,  .04572,  -.01524,   #2
                      0,  .05080,  -.00508,   #3
                      0,  .05080,   .00508,   #4
                      0,  .04572,   .01524,   #5
                      0,  .02540,   .02032,   #6
                      0,  0,       .00508     #7
             ] #end Points
           } #end Coordinates

           coordIndex[ 0, 1, 2, 3, 4, 5, 6, 7, -1,
                       0, 7, 6, 5, 4, 3, 2, 1, -1
           ] #end coordIndex
         } #end IndexedFaceSet
       } #end A_Blade Shape
       ]#end transform children
      } #end transform
     ] #end group children
    } #end A_Blade Group

        Transform{
          rotation 1 0 0 1.5708
          children[ USE Port_Blade ]
        } #end Transform

        Transform{
         rotation 1 0 0 3.14159267
          children[ USE Port_Blade ]
        } #end Transform
```

```
            Transform{
              rotation 1 0 0 -1.5708
              children[ USE Port_Blade ]
            } #end Transform

            #The shaft
            Transform{
              rotation 0 0 1 1.5708
              translation .0281 0 0
              children[
                Shape {
                  appearance Appearance{
                    material Material {diffuseColor  .226 .197 0}
                  } #end Appearance
                  geometry Cylinder {radius .008 height .0762}
                } #end Shape
              ] #end children
            } #end Transform

            #The shaft end cap
            Transform{
              rotation 0 0 1 1.5708
              translation -.015 0 0
              children[
                Shape {
                  appearance Appearance{
                    material Material {diffuseColor  .226 .197 0}
                  } #end Appearance
                  geometry Cone {bottomRadius .008 height .01}
                } #end Shape
              ] #end children
            } #end Transform
          ] #end Screw Group Children
        } #end Screw Group
      ] #end Transform Children
    } #end Transform

  ]#end AUV Group children

} #end AUV Group

#end auv.wrl
```

APPENDIX C. EXPERIMENTAL SCRIPTS AND RESULT DATA

1. Mission.script.SeaStateTest

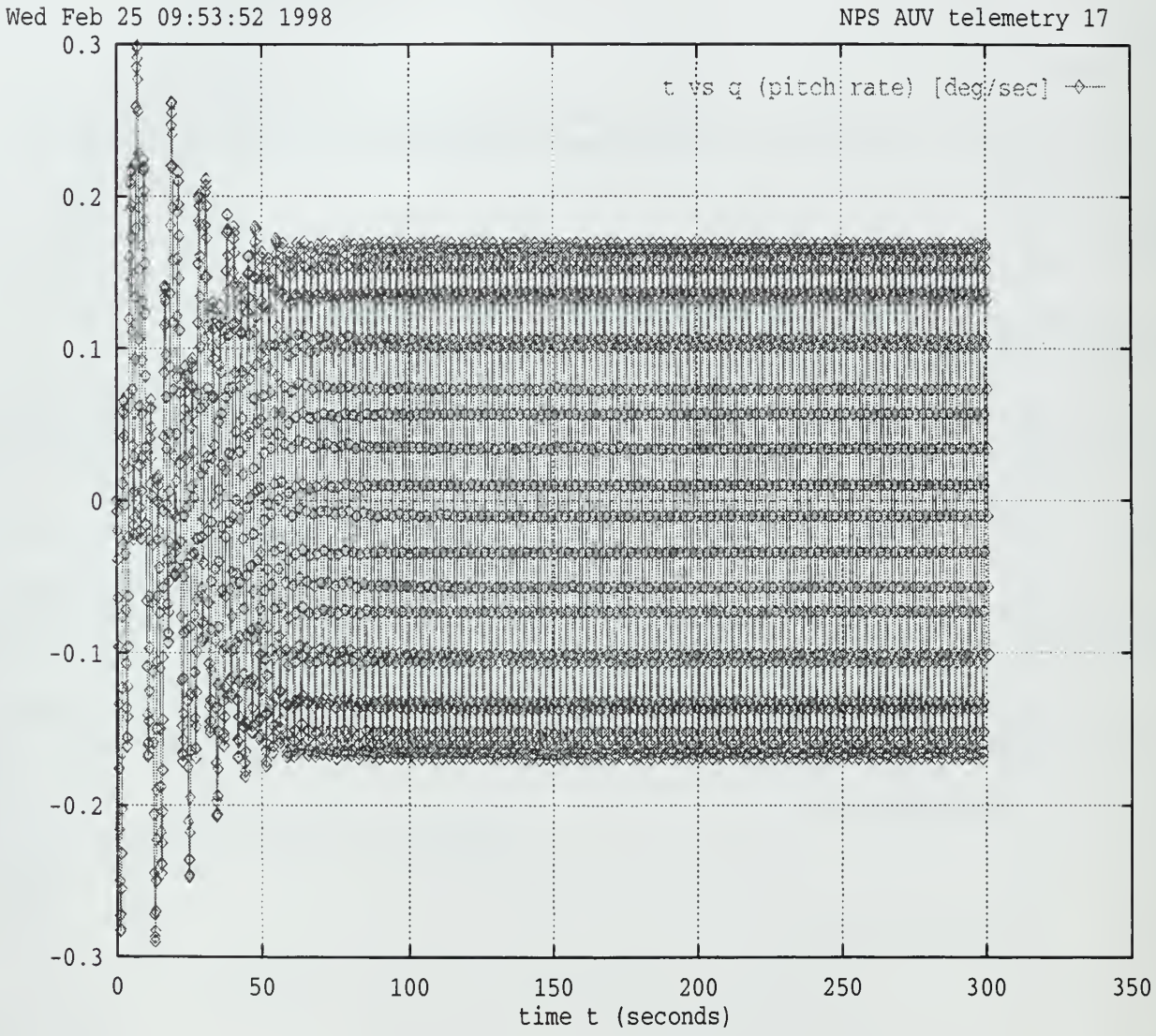
```
# your mission is
# Sea State Test

# mission.script.SeaStateTest

# ,,,,

# initial position
position    -180 50  2


# drive straight into seas
course 000
depth  2
rpm    350

#run test for 5 minutes
wait   300

#done, stop
rpm 0
wait 60

# test complete
```

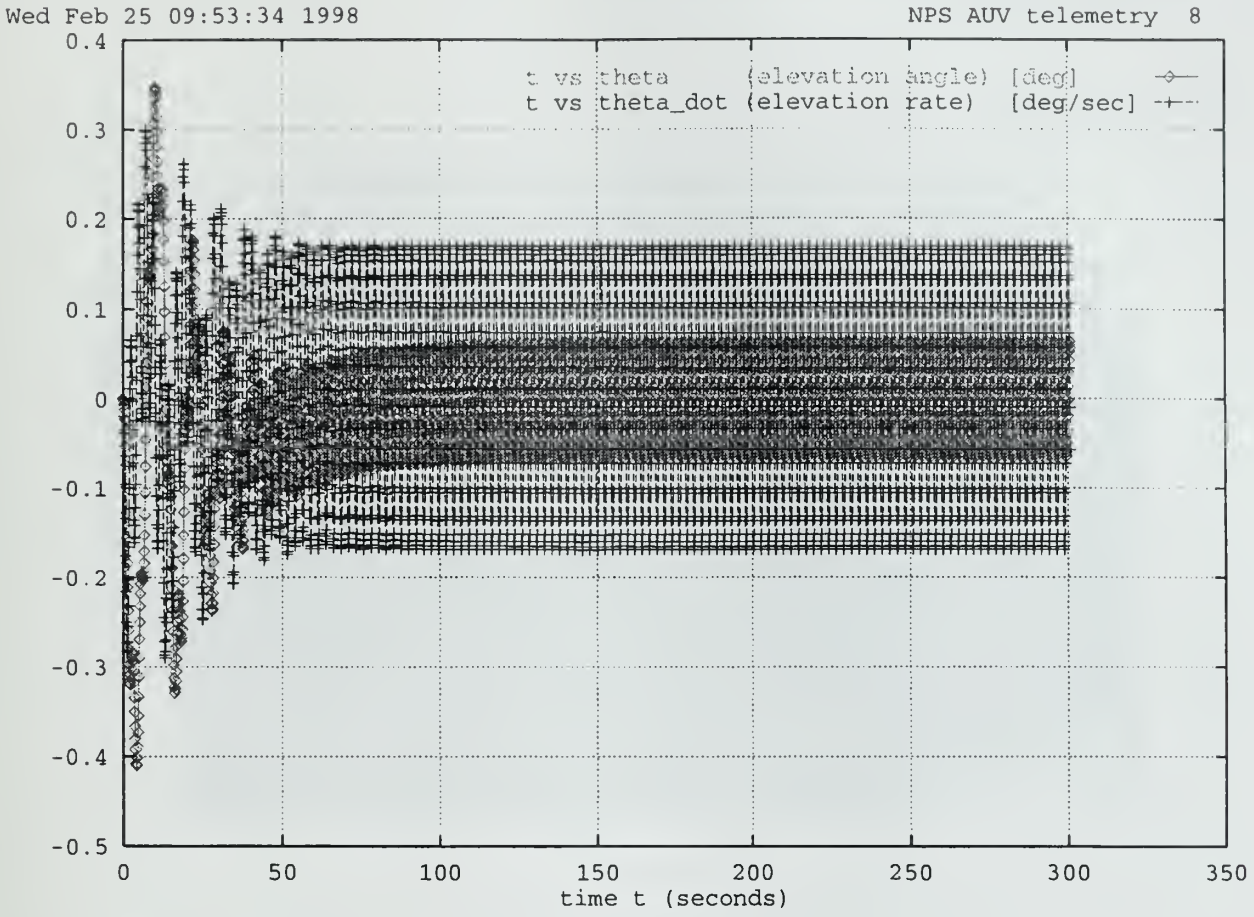
2. Mission.script.FlowFieldTestLoop

```
# your mission is
# flow field test loop

# mission.script.FlowFieldTestLoop

# ,,,,

# shift DS30 Precision Doppler Sonar mode
# to track speed through water, not speed over ground

# ,,,,

# hull is at y distance of 83 feet

# ,,,,

# initial position inside hull
position    117 88  43
orientation   0  0 335

standoff-distance 2.0

# launch from lower port torpedo tube
hover 122 85.5 43 335
wait   10

# drive out of tube
rpm  700
wait  20

# go to surface and turn south
depth    2
course 180
wait    90

# operate at surface first, then go deep
rpm    0
wait  60

thrusters-on
rpm  700
depth 70
wait  60

# drive to aft end of submarine
standoff-distance 4.0
hover -130 75 33 000
```

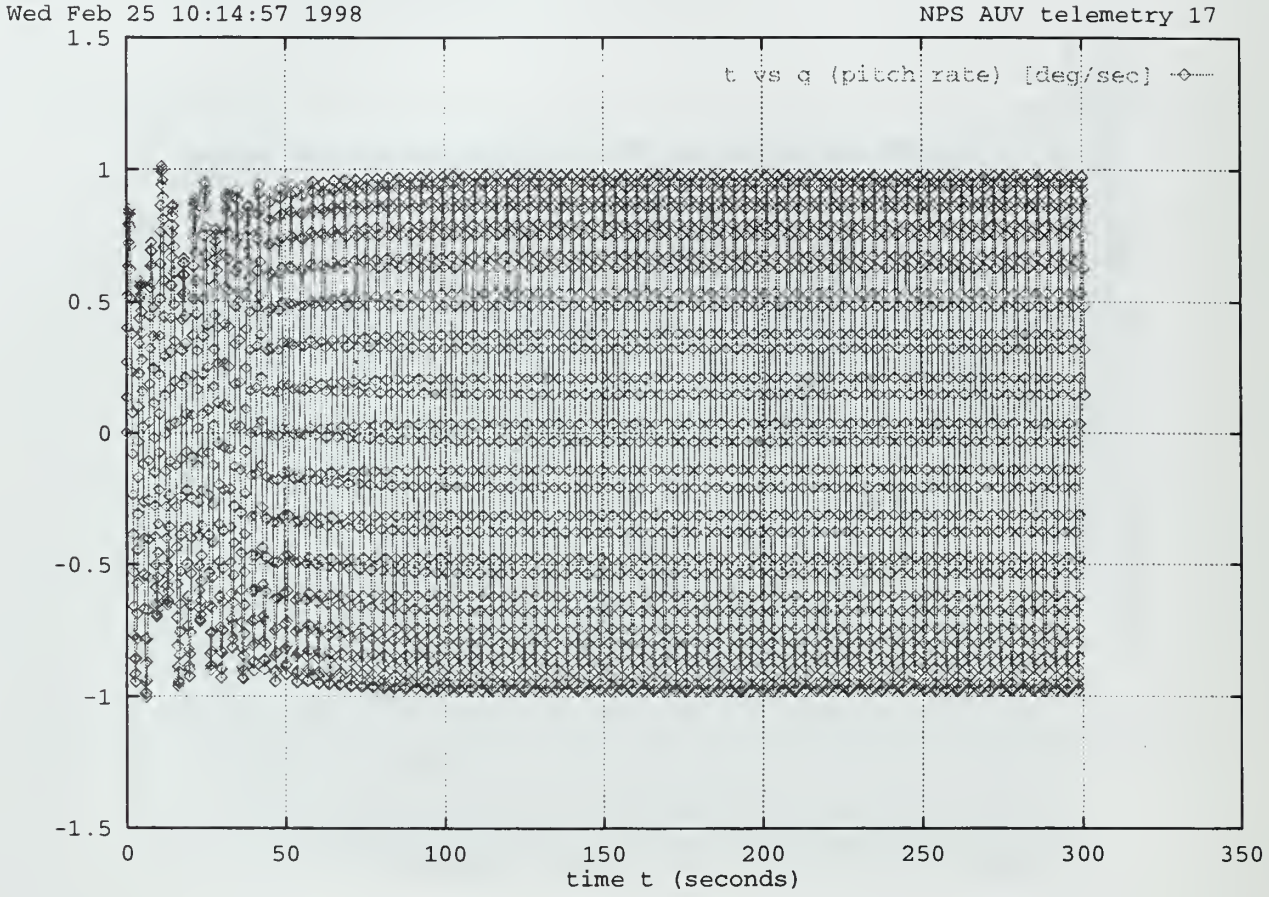
```
# steer collision avoidance sonar
# to track the submarine hull
SONAR_725 090 30 1

# wider scan for tracking sonar
SCAN-WIDTH 45

wait    10
hover-off

# take position just aft of the pump discharge
rpm    400
course 000
depth   33
wait    60

# stabilize after pump discharge
waypoint -25 76 33

# drive through pump suction
course 000
rpm    400
depth   33
wait     5

# stabilize after pump suction
waypoint 90 80 33

# dock with torpedo upper port tube,
# then hover with nose in tube

standoff-distance 0.5

course  025

hover    108.5 84 33

# move in

hover    117   88 33


# stabilize for next iteration
wait 10
hover-off

# docking complete
```

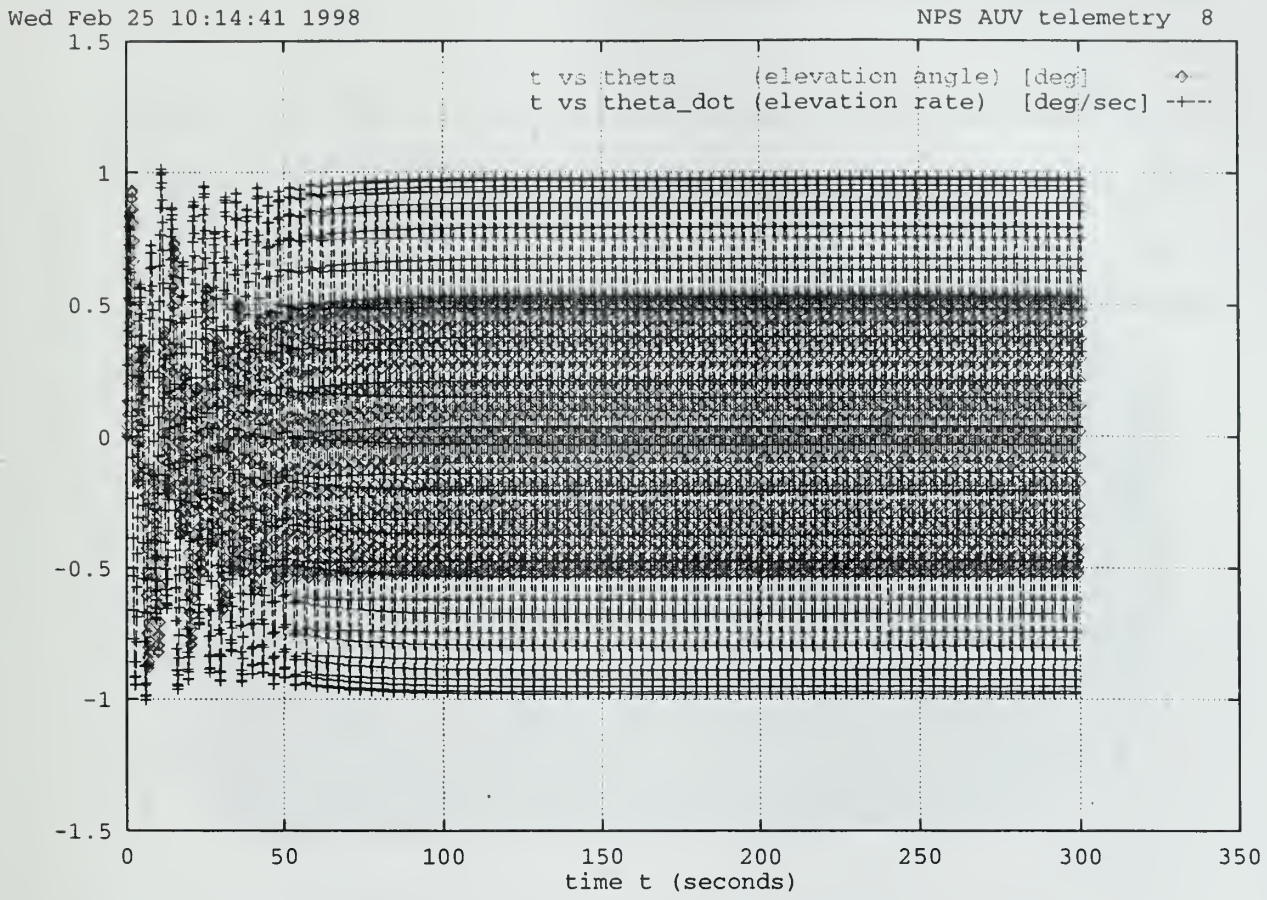
3. SEA STATE 1 SIMULATION DATA



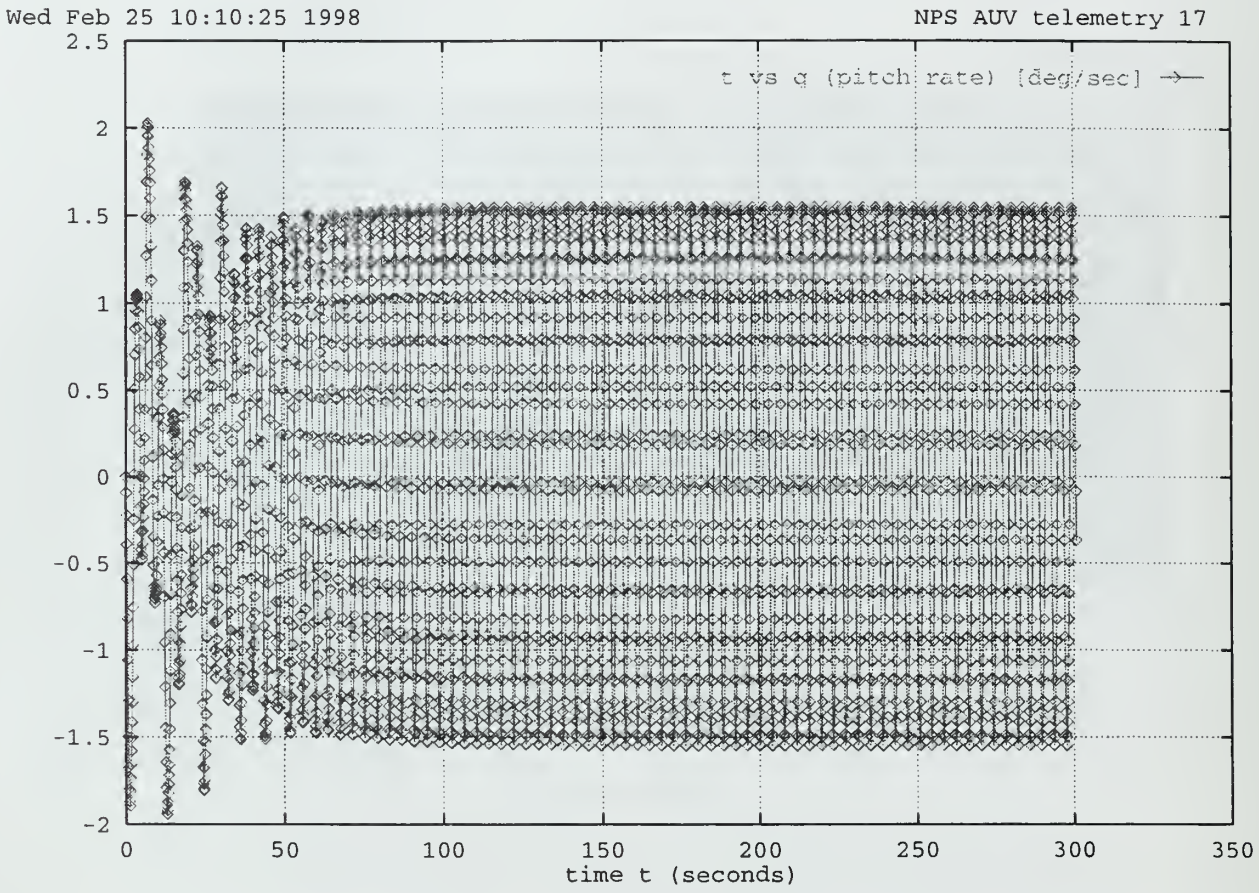

Wed Feb 25 09:53:34 1998
NPS AUV telemetry 8
t vs theta (elevation angle) [deg]
t vs theta_dot (elevation rate) [deg/sec]
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
0
50
100
150
200
250
300
350
time t (seconds)

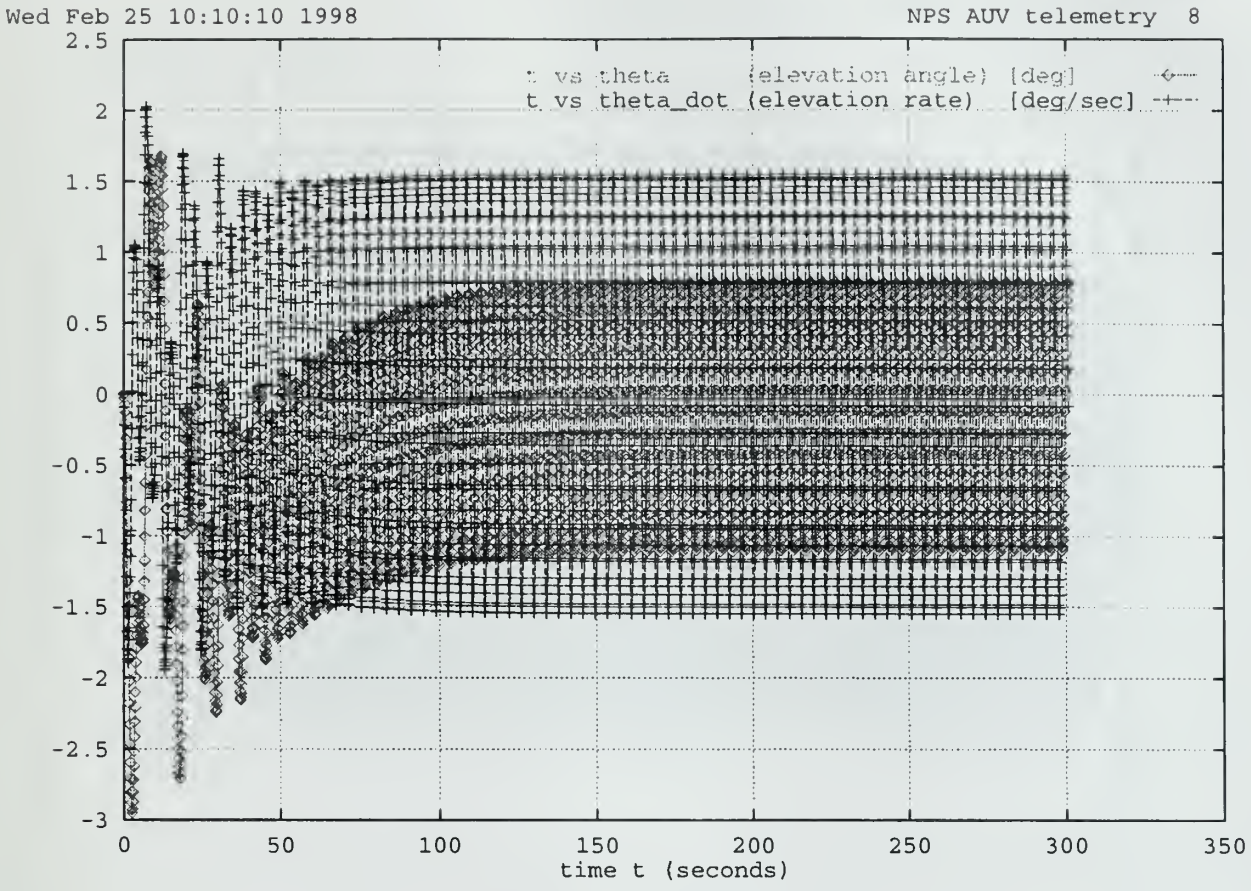
4. SEA STATE 2 SIMULATION DATA



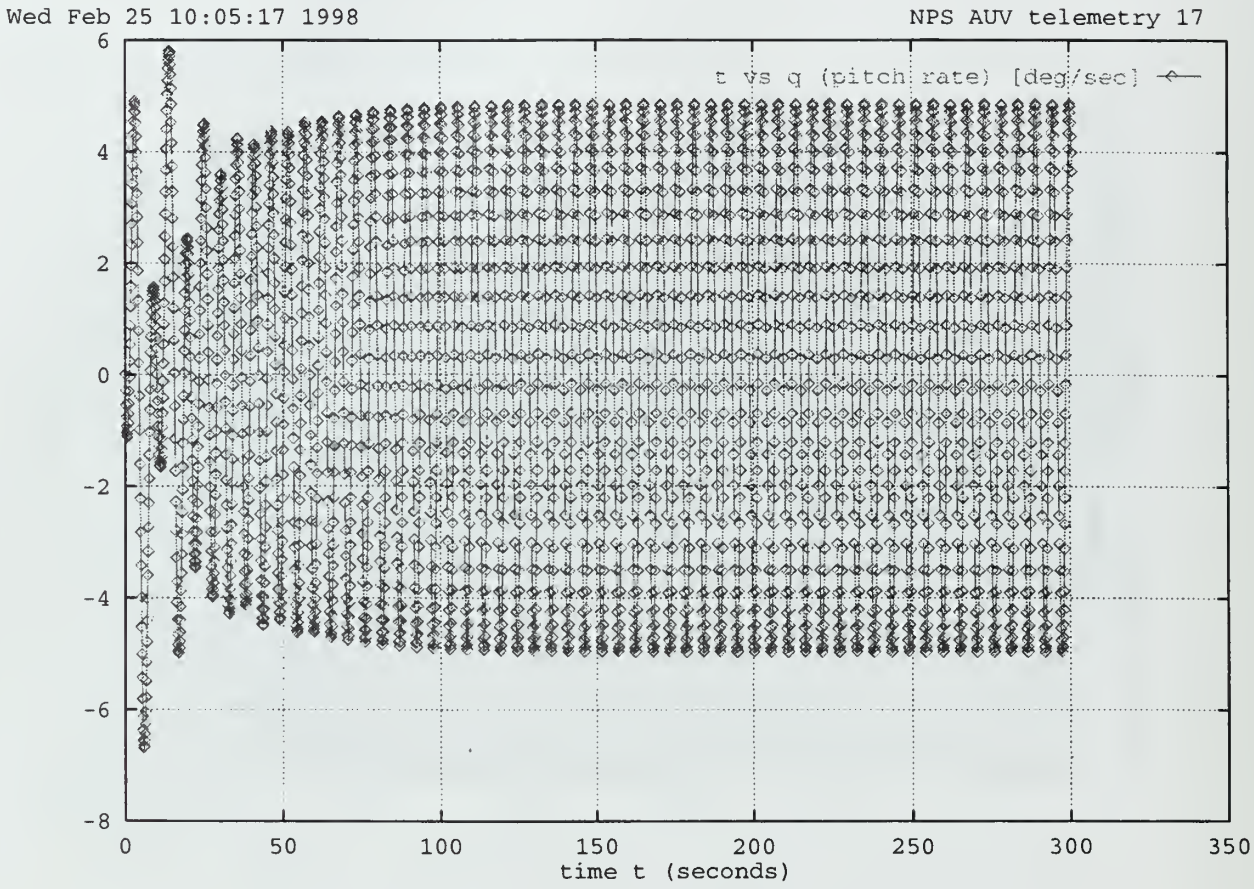

Wed Feb 25 10:14:41 1998
NPS AUV telemetry 8
t vs theta (elevation angle) [deg]
t vs theta_dot (elevation rate) [deg/sec]
1.5
1
0.5
0
-0.5
-1
-1.5
0
50
100
150
200
250
300
350
time t (seconds)

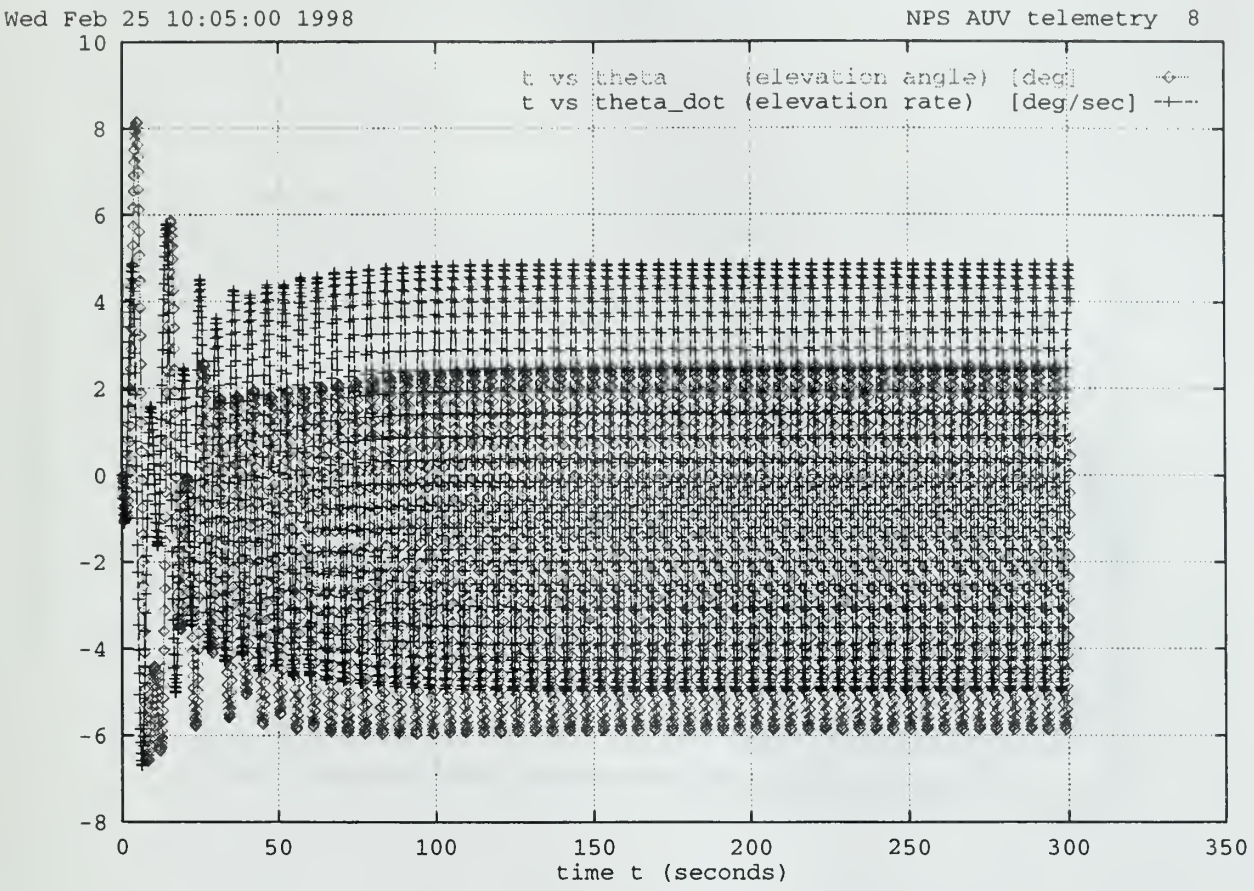
5. SEA STATE 3 SIMULATION DATA



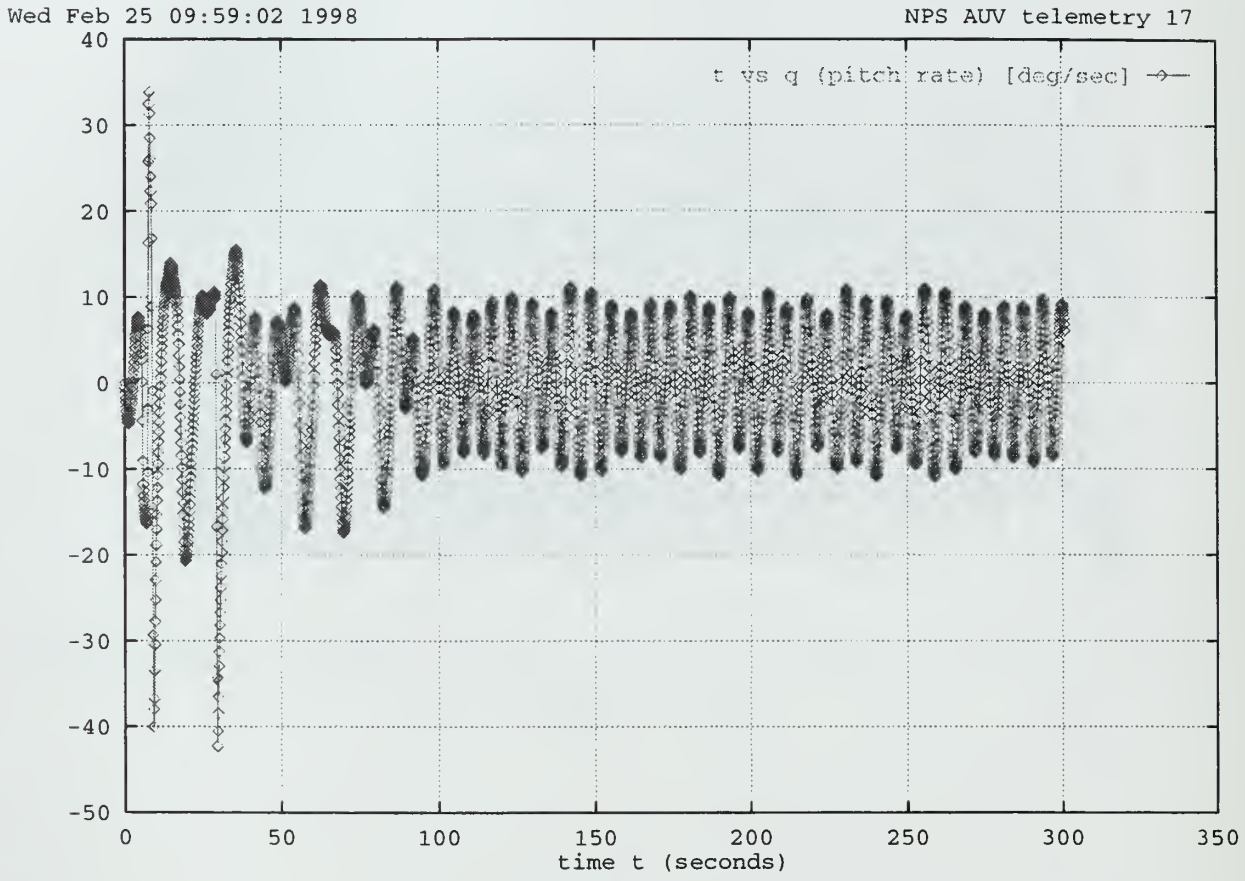

Wed Feb 25 10:10:10 1998
NPS AUV telemetry 8
t vs theta (elevation angle) [deg]
t vs theta_dot (elevation rate) [deg/sec]
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-3
0
50
100
150
200
250
300
350
time t (seconds)

6. SEA STATE 4 SIMULATION DATA

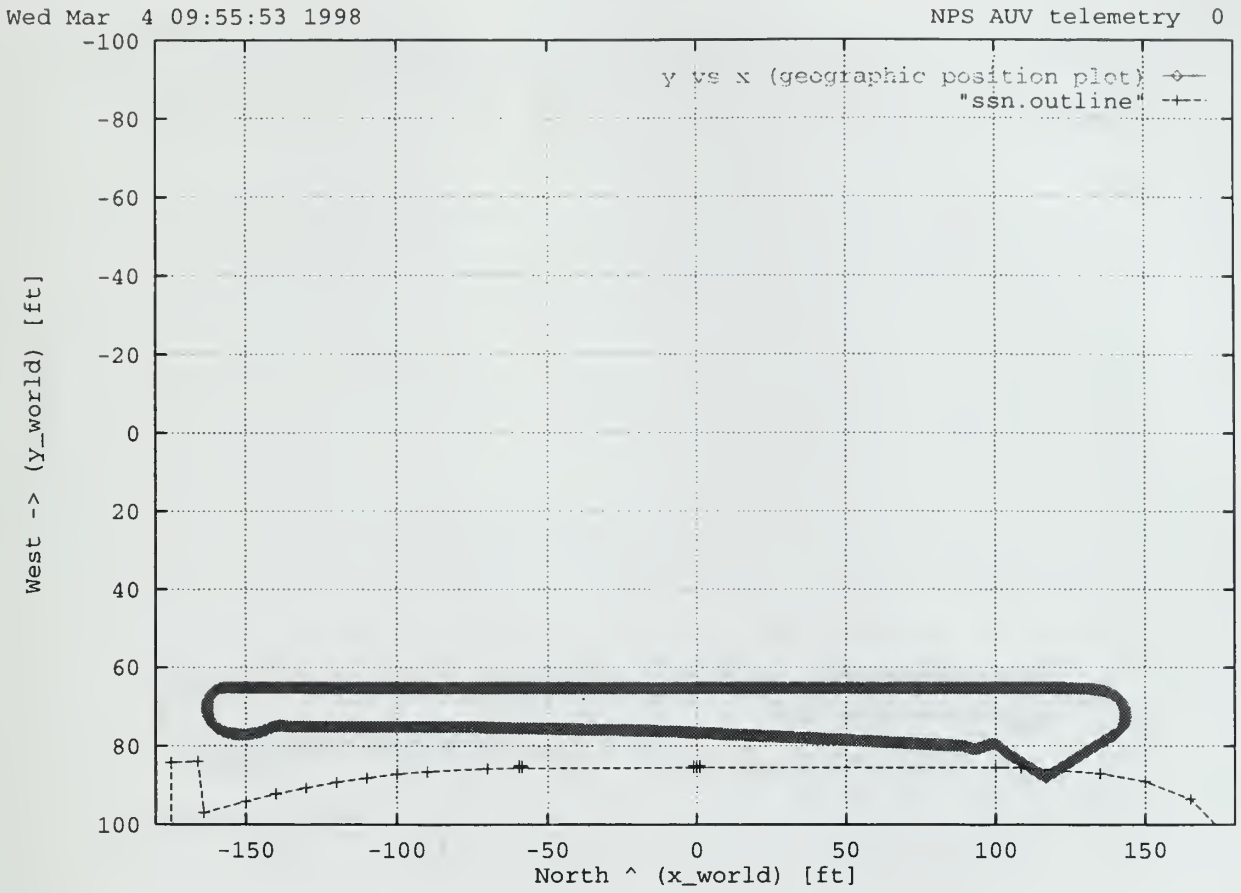



Wed Feb 25 10:05:00 1998
NPS AUV telemetry 8
t vs theta (elevation angle) [deg]
t vs theta_dot (elevation rate) [deg/sec]
10
8
6
4
2
0
-2
-4
-6
-8
0
50
100
150
200
250
300
350
time t (seconds)

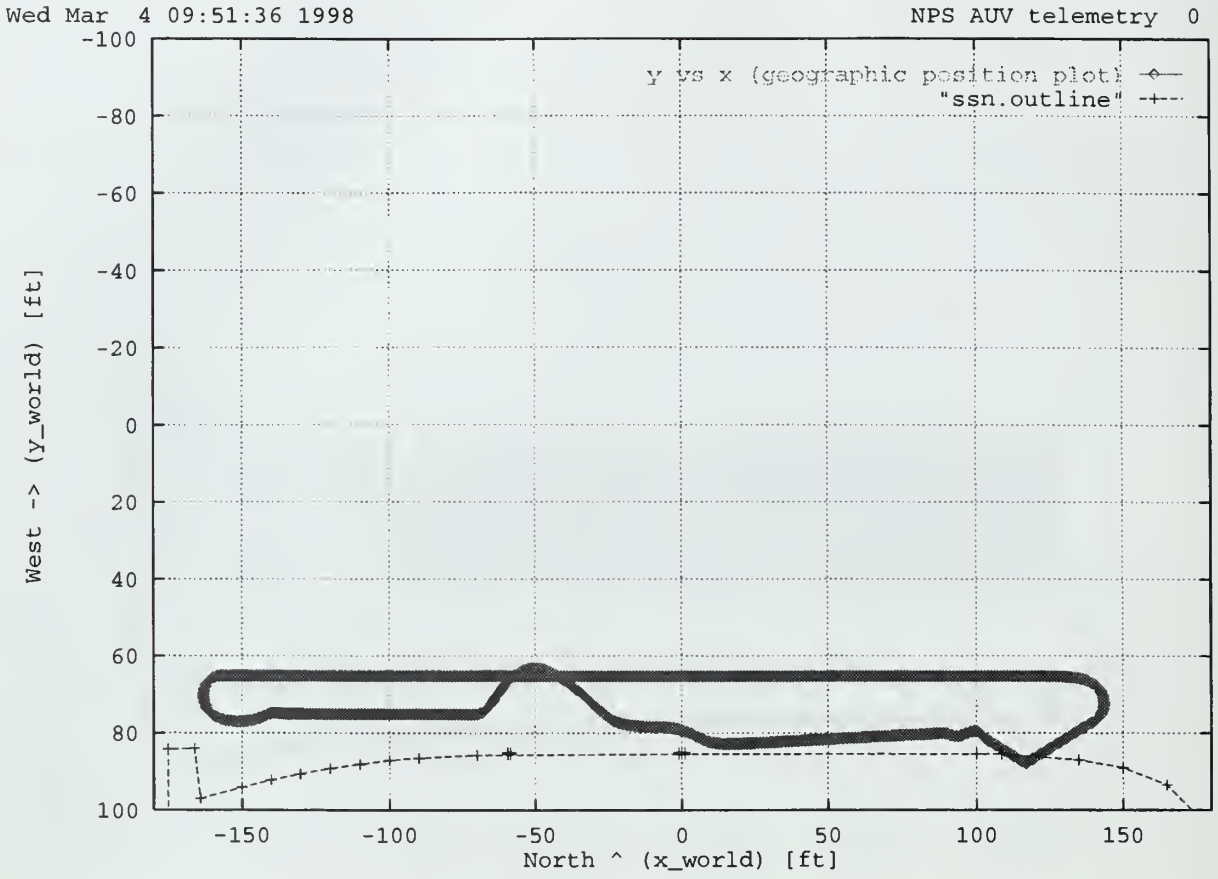
7. SEA STATE 5 SIMULATION DATA



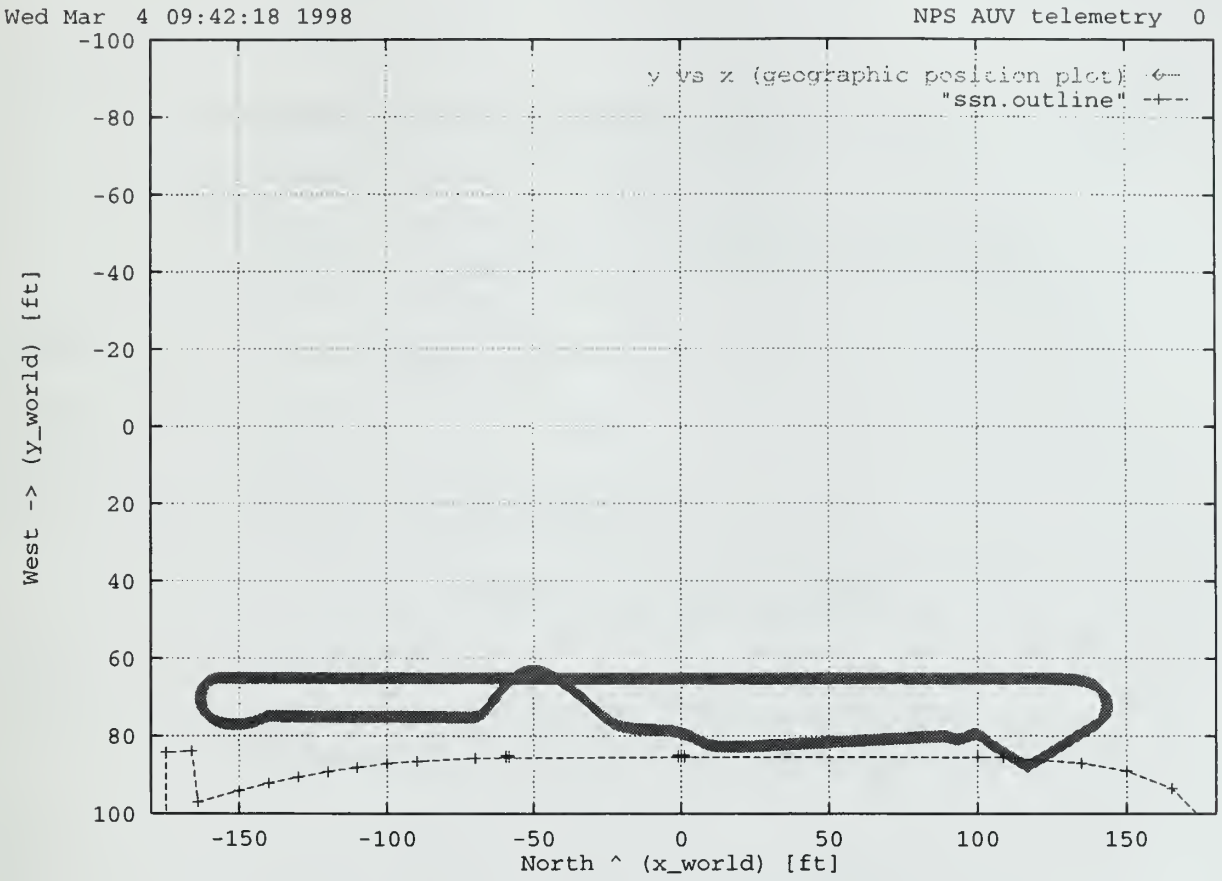
8. X VERSUS Y FOR NO-FLOW SIMULATION



9. X VERSUS Y FOR NORMAL FLOW SIMULATION



10. X VERSUS Y FOR EXTREME FLOW RUN



APPENDIX D. FLOW GENERATION CODE

```
////////////////////////////////////////////////////////////////////////////
/*
 Program:           FlowFieldGenerator.C

 Description:       This program creates the data required for a complex flow field
                    associated with a submarine driving through the water. It uses
                    Flat Plate Fluid Flow theory to create a series of files which
                    contain the data used by the Phoenix AUVs Virtual environment.

                    This program is based upon a program which was written in fortran
                    called  ITBL (Incompressible Turbulent Boundary Layer) from a
                    mechanical Engineering text. The book was called Boundary Layer
                   Analysis, by Joseph A. Schetz.

 Revised:           26 January 98

 System:            Irix 5.3

 Compiler:          ANSI C++

 Compilation:       irix> make FlowFieldGenerator.o
                    irix> CC FlowFieldGenerator.C -lm -c -g +w

                    -c == Produce binaries only, suppressing the link phase.
                    +w == Warn about all questionable constructs.

 Author:            Kevin Byrne

 Thesis:            Byrne, Kevin M., Real-Time Modeling of Cross-Body
                    Flow for Torpedo Tube Recovery of the Phoenix Autonomous
                    Underwater Vehicle, Masters Thesis, Naval Postgraduate
                    School, Monterey California, March 1998.

 Advisors:          Dr. Don Brutzman, Dr. Bob McGhee

 References:        Schetz, Joseph A., _Boundary Layer Analysis_,
                           Prentice Hall, Englewood Cliffs, NJ, 1992.

Notes:


//      2-d Boundary Layer Computation,  Incompressible,
//      Turbulent, 1st Order,  Implicit
//
//      Mixing-length or Eddy-viscosity Model or Tke Model.
//
//      Equations Are Dimensionless Using Freestream Velocity,
//      Uinf, Viscosity, Muinf, and Density, Rhoinf, and a
//      Reference Length, L; X/l, Y/l, U/uinf, Also
//      Re = Rhoinf*uinf*l/muinf
//      Pick L = 1.0.
//
//      Other Variables:
//      Rkap, Kappa in the Mean Flow Turbulence Models
//      Ypa, Y Sub A+
//      Del, Starting Boundary Layer Thickness
//      Duedx, Derivative of Edge Velocity in the X Direction
//      Red, Reynolds Number Based on Delta
//      Usue, Ustar/uedge
//      A,b,c, Splitting up the Boundary Layer Equations
//
```

```
//      Sample Problem of a Flat Plate with Uinf = 10.0. Start at
//      X = 5.0. Goto X = 6.0. Take Nu(=muinf/rhoinf) = 1.0e-5. Rex=5.0e6.
//
//      Use Simple Integral Solution to Get Initial Values.
//      Delta = 0.0856. Cf = 0.002665. Other Flows Can Be Set by User.
////////////////////////////////////////////////////////////////////////////

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <math.h>

#define TRUE  1
#define FALSE 0

struct FlowGridElements {
  double direction;          //direction  relative to sub heading
  double x_magnitude;        //The magnitudes are dimensionless
  double y_magnitude;        //forces in the submarine reference frame
  double z_magnitude;     .
};

const int FLOWFIELDLENGTH = 721;  //2 * lenght of sub om .5 ft inc
const int FLOWFIELDWIDTH  = 61;   //4 * the number of cross_sections on AUV


//These grids are used to pass the initial flow profile from the flat
//Plate model to the tube level model. The tube level model refines
//these arraysin the areas of interest.
FlowGridElements global1ktgrid[FLOWFIELDLENGTH][FLOWFIELDWIDTH] = {0};
FlowGridElements global2ktgrid[FLOWFIELDLENGTH][FLOWFIELDWIDTH] = {0};
FlowGridElements global3ktgrid[FLOWFIELDLENGTH][FLOWFIELDWIDTH] = {0};


//Local Constants
const double RKAP = 0.41;
const double YPA = 9.7;

//---------------------------------------------------------------------------
void eddy( int NNX, int MMAX, int MEST, const double RE, double DY,
                  double  U0[],  double UE[], double T[], double CF[])   {

                                                    .
   double    RMUT  = 0.0;
   double    Y     = 0.0;
   double    YP    = 0.0;
   double    DELST = 0.0;


   for (int ie = 2; ie <= MEST; ie++)  {
      DELST = DELST + DY*(1.0-0.5*(U0[ie-1]+U0[ie])/UE[NNX]);
   }


   //   CLAUSER EDDY VISCOSITY MODEL
   RMUT = 0.018*RE*UE[NNX]*DELST;

   for (int ig = 1; ig <= MMAX; ig++)  {

      Y  = (ig-1) * DY;
      YP = Y*UE[NNX]*RE*sqrt(0.5*CF[NNX-1]);

      //REICHART MODEL FOR COMPLETE INNER REGION
      T[ig] = RKAP*(YP-YPA*tanh(YP/YPA));
```

```
      if (T[ig] > RMUT)  {
         T[ig] = RMUT;
      }
   }

   return;

}   //end of function eddy


//---------------------------------------------------------------------------
void TRID ( int MM, double A[], double B[], double C[], double R[], double S[]) {


   double GAM[550];
   double RP[550];
   double DENO;

   //   THOMAS ALGORITHM
   GAM[1] = C[1] / B[1];
   RP[1] = R[1] / B[1];

   for (int ih = 2; ih <= MM; ih++)  {

      DENO = B[ih] - A[ih] * GAM[ih-1];
      GAM[ih] = C[ih] / DENO;
      RP[ih] = (R[ih]-A[ih]*RP[ih-1]) / DENO;
   }

   S[MM] = RP[MM];

   for (int ii = 1; ii <= MM-1; ii++)  {
      S[MM-ii] = RP[MM-ii]-GAM[MM-ii]*S[MM-ii+1];

   }

   return;

}   //end of function TRID


//---------------------------------------------------------------------------
void flatPlateFlowFieldGenerator ( void ) {

//      YOU MUST GIVE INITIAL X (XI), FINAL X (XF), (CNU), (UNINF),
//      (NMAX), (MMAX) AND (DY)
//      PICK MMAX BASED ON INITIAL BOUNDARY LAYER THICKNESS AND
//      NUMBER OF POINTS ACROSS THE LAYER.  USE AT LEAST 400 ACROSS
//      DELTA.  ADD AT LEAST 100 POINTS ABOVE DELTA.
//
//      PICK NMAX BASED ON LENGTH OF REGION AND DX DESIRED.  DX
//      CAN BE OF THE ORDER OF INITIAL DELTA/FIVE. TAKE L = 1.0.

   //Constant Variables
   const int ARRAY_SIZE = 550;
   const double    XI = 0.0;          //Initial X position where flow hits plate
   const double    CNU = 0.000001;

   //---------------------------------------
   //Variables I needed to add for iteration down sub Hull

   int current_distance     = 0;
   int index_difference       = 0;
```

```
   double submarine_length_meters = 111;   //Length in meters = 360 ft, truly 109.7m
   //double submarine_speed  = 1.0;

   struct flowRecord {
      double x_magnitude;
      double y_magnitude;
      double z_magnitude;
      double direction;
   };

   flowRecord nontubelevelgrid[FLOWFIELDLENGTH][FLOWFIELDWIDTH];


   //Array declararations
   double U[ARRAY_SIZE];
   double U0[ARRAY_SIZE];
   double V[ARRAY_SIZE];
   double V0[ARRAY_SIZE];
   double CF[150];
   double A[ARRAY_SIZE];
   double B[ARRAY_SIZE];
   double C[ARRAY_SIZE];
   double R[ARRAY_SIZE];
   double TKEO[ARRAY_SIZE];
   double TMU[ARRAY_SIZE];
   double UE[150];
   double DUEDX[150];
   double Y[ARRAY_SIZE];                    //distance from hull feet

   //Varibles Required For Fluid Caluations
   int       MMAX  = 60;            //MMAX originally was 525, NMAX originally was 101
   int       NMAX  = 15;
   int       MEST  = 401;           //This is the M index for the initial Delta
   int       FM1   = MEST-1;
   int       NMAXP = NMAX;
   int       MMAXP = MEST+100;
   int       NNX;

   double    XF = 0.0;               //The position at which the profile is generated
   double    DY = 0.02/MMAX;         // step distance away from hull per calcualtion(last
value.035)
   double    DX;                     // distance from the start of plate
   double    DENO;
   double    DEL = 0.00015;
   double    Y0D;
   double    pump_outlet_jet_factor;      //Holds value for pump force reduction sw
discharge
   double    pump_inlet_jet_factor;       //Holds value for pump force reductionsw suction
   double    pump_outlet_jet_speed = 2.5; //Holds value for pump force (knots)
   double    pump_inlet_jet_speed  = 1.0; //Holds value for pump force (knots)

   double    pumpSuctionPosition_ft   = 180.5; //The position along hte hull of suction
(ft)
   double    pumpDischargePosition_ft = 245.5; //The position along hte hull of discharge
(ft)

   double    suctionBegin_m    = (pumpSuctionPosition_ft - 6.5) * 0.3048; //Begin of
suction forces(m)
   double    suctionEnd_m      = (pumpSuctionPosition_ft + 6.5) * 0.3048; //End of suction
forces(m)

   double    dischargeBegin_m  = (pumpDischargePosition_ft - 6.5) * 0.3048; //Begin of
discharge forces(m)
   double    dischargeEnd_m    = (pumpDischargePosition_ft + 6.5) * 0.3048; //End of
discharge forces(m)
```

```
   //-----------------------------------------------------------------
   //Output streams to hold the generated flow fields for later usage
   //      flatplateflowfield1kt.data - Holds the entire flatplate model flow field
   //            for a submarine speed of 1 kt.
   //      flatplateflowfield2kt.data - Holds the entire flatplate model flow field
   //            for a submarine speed of 2 kt.
   //      flatplateflowfield3kt.data - Holds the entire flatplate model flow field
   //            for a submarine speed of 3 kt.
   //
   //      These others are just for data verification.
   //      flatprofile.data  - Holds flat plate data for sub profile
   //      flatslice50.data  - Holds flat plate data for slice at 50 ft
   //      flatslice100.data - Holds flat plate data for slice at 100 ft
   //      flatslice150.data - Holds flat plate data for slice at 150 ft
   //      flatslice200.data - Holds flat plate data for slice at 200 ft
   //      flatslice250.data - Holds flat plate data for slice at 250 ft


   ofstream plate1ktOutput ("flatplateflowfield1kt.data", ios::out);
   ofstream plate2ktOutput ("flatplateflowfield2kt.data", ios::out);
   ofstream plate3ktOutput ("flatplateflowfield3kt.data", ios::out);
   ofstream plateProfileOutput ("flatprofile.data", ios::out);
   ofstream plateSlice50Output ("flatslice50.data", ios::out);
   ofstream plateSlice100Output("flatslice100.data", ios::out);
   ofstream plateSlice150Output("flatslice150.data", ios::out);
   ofstream plateSlice200Output("flatslice200.data", ios::out);
   ofstream plateSlice250Output("flatslice250.data", ios::out);

for (int submarine_speed = 1; submarine_speed < 4; submarine_speed++) {

   int last_distance_filled = 0;
   double    UINF = (double)submarine_speed; //This is the flow strength in open water
   //This  declared  here due to dependence on other variables
   double    RE = UINF * submarine_length_meters / CNU;


   //-----------------------------------------------------------------
   //Initialize flow field to zero prior to each speed iteration
   for ( int row = 0; row < FLOWFIELDLENGTH; row++) {
      for (int col = 1; col < FLOWFIELDWIDTH; col++) {

         nontubelevelgrid[row][col].x_magnitude = 0.0;
         nontubelevelgrid[row][col].y_magnitude = 0.0;
         nontubelevelgrid[row][col].z_magnitude = 0.0;
         nontubelevelgrid[row][col].direction = 180.0;
      }
   }



   //-----------------------------------------------------------------
   //This is the main loop. It generates the Flow field from bow to stern
   //in 1 meter increments. Each profile starts from the hull and
   //goes outward until flow force = Uinf (~30 ft)
   for (int generationloop = 1; generationloop < submarine_length_meters  ;
generationloop++) {

     //Flag for file output
     // firstEntry = 1;
      //This increments XF by 1 m each time. The loop will run from bow to stern
      XF = (double)generationloop;
      DX = (XF-XI) / (NMAX-1);

      //initialize UE and DUEDX arrays they are only 150 elements large
      for ( int iw = 0; iw <= NMAX; iw++) {
```

```
  UE[iw]    = 1.0;
  DUEDX[iw] = 0.0;
}

//additional variable which depend on nitialized arrays
CF[1] = 0.001;
double USUE = sqrt(CF[1]/2.0);
double RED = RE * DEL * USUE * UE[1];

//initialize other arrays, these are 550 elements large
for ( int ix = 1; ix <= MMAX; ix++) {

   U[ix]     = UE[1];
   U0[ix]    = UE[1];
   V[ix]     = 0.0;
   V0[ix]    = 0.0;
   TKEO[ix]  = 0.0;
   TMU[ix]   = 0.0;
}


//        NO SLIP CONDITION
U[1] = 0.0;
U0[1] = 0.0;

//-------------------------------------------------------------
//The initial profiles of U and V can be changed by the user.
//MEST is the M index for the initial Delta.
//Assume a Coles Wake Law Initial Velocity Profile
//-------------------------------------------------------------
for (int iy = 2; iy <= MEST; iy++) {

   Y0D = (double)(iy-1) / (double)FM1;
   U0[iy] = USUE*UE[1]*
            (1.0/RKAP*log(Y0D*RED) + 4.90 + 0.51/RKAP
            * 2.0 * pow((sin(Y0D*1.5708)) ,2));
   V0[iy] = 0.0;
}


//By this point all initialization is done, U0 and V0 are initial U+V profiles
int  done_200 = FALSE;
int iz = 2;

while (( iz <= NMAX ) && (done_200 == FALSE)) {

  NNX = iz;
  U0[MMAX] = UE[iz];
  V0[MMAX] = 0.0;

  eddy(NNX, MMAX, MEST, RE, DY, U0, UE,  TMU, CF);

  B[1] = 1.0;
  C[1] = 0.0;
  R[1] = 0.0;
  A[MMAX] = 0.0;
  B[MMAX] = 1.0;
  C[MMAX] = 0.0;
  R[MMAX] = UE[iz];
  DENO = RE*DY*DY;

  for (int ia = 2; ia <= MMAX-1; ia++) {

      A[ia] = -0.5*V0[ia]/DY-(1.0+TMU[ia-1])/DENO;
      B[ia] = U0[ia]/DX+(2.0+TMU[ia-1]+TMU[ia])/DENO;
```

```
      C[ia] = 0.5*V0[ia]/DY-(1.0+TMU[ia])/DENO;
      R[ia] = UE[iz]*DUEDX[iz]+U0[ia]*U0[ia]/DX;

   }

   TRID(MMAX, A, B, C, R, U);

   for (int ib = 2; ib <= MMAX - 1; ib++) {
      V[ib] = V[ib-1]-(0.5*DY/DX)*(U[ib]-U0[ib]+U[ib-1]-U0[ib-1]);
   }

   int done = FALSE;
   int ic = MEST - 10;

   while ((ic <= MMAX) && (done == FALSE))  {
      if (U[ic] > 0.99*UE[iz]) {
         MEST = ic;
         MMAXP = MEST+100;
         done = TRUE;
      }
      ic++;
   }

   //This steps in the X-direction from front of plate to current position
   for (int id = 2; id <= MMAX; id++)  {

      U0[id] = U[id];
      V0[id] = V[id];
   }


   CF[iz] = (4.0*U0[2]-U0[3])/(pow(UE[iz],2)*DY*RE);

   //Check if near Seperation, if so this profile is done
   if (CF[iz] < 0.0001) {

      NMAXP = iz;
      done_200 = TRUE;
   }
   iz++;
} //end of for 200 loop

if (MMAXP > MMAX) {
   MMAXP = MMAX;
}


//-----------------------------------------------------------
//This section puts data in seperate files for later use.
//The data is formatted in the following order:
//    X-dir flow component Y-dir Z-dir vector direction
//All values are unitless. This allows scaling during usage.
//-----------------------------------------------------------

//Calculate the number of feet down the hull we are
current_distance = (int)(generationloop/.3048) * 2;

//Check to ensure we have a good ft increment on hull
if (current_distance > 720 ) {
   cout << " Distance along hull exceeded 360 ft, reset to 360 ft (720)" << endl;
   current_distance = 720;
}

if (current_distance < 0 ) {
   cout << " Distance along hull below 0 ft, reset to 1" << endl;
```

```
         current_distance = 0;
      }

      //Calculate the number of rows to be filled.This is needed because the
      //dynamics model needs a flow field with 0.5 ft increments, and this
      //generates a field of 1 meter increments. We interpolate to fill
      //in the missing data
      index_difference = current_distance - last_distance_filled;

      //Output routine to put Values in proper arrays and files
      //A loop is used to access each U value for this position on
      //Hull. The generationloop index represents the distance along
      // the hull in meters
      for (int ij = 0; ij < MMAXP; ij++)  {

         //Distance from the hull in feet
         Y[ij] =(ij-1)*DY* submarine_length_meters / 0.3048 ;


         //-------------------------------------------------
         //This section does array output
         //output of flow field into flowfield arrays
         //We must fill all .5 ft incremented array rows between
         //current_distance and  last_distance_filled

         int pass = 1;
         for (int arrayindex = last_distance_filled + 1; arrayindex < current_distance;
arrayindex++) {

            //-----------------------------------------------------------------
            // The output data is given in knots based on submarine speed. Dynamics
converts it to ft/sec
            // To convert knots to ft/sec kts*2000*3/60/60= 1.6667
            //In order to get this force into true x, y, z components it is necessary to
multiply the components
            //by a factor which relates them to the sub's refernce frame. Since for the
flat plate model
            //I assume x and z components are zero, only Y is adjusted . For the tube
level profile when
            //fully integrated each component will need to be adjusted.

            //reset pump jet force to one
            pump_outlet_jet_factor = 1.0;
            pump_inlet_jet_factor  = -1.0;



            for (int column = 0; column < FLOWFIELDWIDTH; column++) {

               nontubelevelgrid[arrayindex][column].x_magnitude = 0.0;
               nontubelevelgrid[arrayindex][column].y_magnitude = - U[column] *
submarine_speed;
               nontubelevelgrid[arrayindex][column].z_magnitude = 0.0;
               nontubelevelgrid[arrayindex][column].direction = 180.0;

               //This section adds a pump inlet 180 ft back on the hull. It starts out at
full
               //force and diminishes to 0 at 20 ft out from the hull. It assumes water is
sucked in at 2.5 kts
               if ((generationloop > suctionBegin_m) && (generationloop < suctionEnd_m)) {
                 nontubelevelgrid[arrayindex][column].x_magnitude = pump_inlet_jet_factor
                                                                    * pump_inlet_jet_speed;

                 if (pump_inlet_jet_factor < -0.2) {
                    pump_inlet_jet_factor = pump_inlet_jet_factor + 0.025;
```

```
                }
                else {
                   pump_inlet_jet_factor = 0.0;
                }

             }


             //This section adds a pump discharge jet 246 ft back on the hull. It starts
out at full
             //force and diminishes to 0 at 20 ft out from the hull.It assumes water is
discharged in at 2.5 kts
             else if ((generationloop > dischargeBegin_m) && (generationloop <
dischargeEnd_m)) {
                nontubelevelgrid[arrayindex][column].x_magnitude = pump_outlet_jet_factor
                                                                 * pump_outlet_jet_speed;

                if (pump_outlet_jet_factor > 0.2) {
                   pump_outlet_jet_factor = pump_outlet_jet_factor - 0.025;
                }
                else {
                   pump_outlet_jet_factor = 0.0;
                }

             }



             //Now write these values to the proper file
             switch ((int)submarine_speed) {

                case 1:
                   plate1ktOutput << arrayindex << " " << column << " "
                                  << nontubelevelgrid[arrayindex][column].x_magnitude <<
" "
                                  << nontubelevelgrid[arrayindex][column].y_magnitude <<
" "
                                  << nontubelevelgrid[arrayindex][column].z_magnitude <<
" "
                                  << nontubelevelgrid[arrayindex][column].direction
                                  << endl;

                   //Update the global array
                   global1ktgrid[arrayindex][column].x_magnitude =
nontubelevelgrid[arrayindex][column].x_magnitude;
                   global1ktgrid[arrayindex][column].y_magnitude =
nontubelevelgrid[arrayindex][column].y_magnitude;
                   global1ktgrid[arrayindex][column].z_magnitude =
nontubelevelgrid[arrayindex][column].z_magnitude;
                   global1ktgrid[arrayindex][column].direction   =
nontubelevelgrid[arrayindex][column].direction;
                   break;
                case 2:
                   plate2ktOutput << arrayindex << " " << column << " "
                                  << nontubelevelgrid[arrayindex][column].x_magnitude <<
" "
                                  << nontubelevelgrid[arrayindex][column].y_magnitude <<
" "
                                  << nontubelevelgrid[arrayindex][column].z_magnitude <<
" "
                                  << nontubelevelgrid[arrayindex][column].direction
                                  << endl;

                   //Update the global array
                   global2ktgrid[arrayindex][column].x_magnitude =
```

```
nontubelevelgrid[arrayindex][column].x_magnitude;
                    global2ktgrid[arrayindex][column].y_magnitude =
nontubelevelgrid[arrayindex][column].y_magnitude;
                    global2ktgrid[arrayindex][column].z_magnitude =
nontubelevelgrid[arrayindex][column].z_magnitude;
                    global2ktgrid[arrayindex][column].direction   =
nontubelevelgrid[arrayindex][column].direction;

                    break;
                 case 3:
                    plate3ktOutput << arrayindex << " " << column << " "
                                   << nontubelevelgrid[arrayindex][column].x_magnitude <<
" "
                                   << nontubelevelgrid[arrayindex][column].y_magnitude <<
" "
                                   << nontubelevelgrid[arrayindex][column].z_magnitude <<
" "
                                   << nontubelevelgrid[arrayindex][column].direction
                                   << endl;

                    //Update the global array
                    global3ktgrid[arrayindex][column].x_magnitude =
nontubelevelgrid[arrayindex][column].x_magnitude;
                    global3ktgrid[arrayindex][column].y_magnitude =
nontubelevelgrid[arrayindex][column].y_magnitude;
                    global3ktgrid[arrayindex][column].z_magnitude =
nontubelevelgrid[arrayindex][column].z_magnitude;
                    global3ktgrid[arrayindex][column].direction   =
nontubelevelgrid[arrayindex][column].direction;

                    break;
                 default:
                    cerr << "Invalid Submarine Speed" << endl;
                 break;
              }  // end switch
           } //end of column loop
           pass += 1;


        }

        //Fill current distance array, and write values to file
        nontubelevelgrid[current_distance][ij].x_magnitude = 0.0;
        nontubelevelgrid[current_distance][ij].y_magnitude = - U[ij] * submarine_speed;
        nontubelevelgrid[current_distance][ij].z_magnitude = 0.0;
        nontubelevelgrid[current_distance][ij].direction = 180.0;

             //This section adds a pump inlet 180 ft back on the hull. It starts out at
full
             //force and diminishes to 0 at 20 ft out from the hull. It assumes water is
sucked in at 2.5 kts
             if ((generationloop > suctionBegin_m) && (generationloop < suctionEnd_m)) {
               nontubelevelgrid[current_distance][ij].x_magnitude =
pump_inlet_jet_factor
                                                          * pump_inlet_jet_speed;

               if (pump_inlet_jet_factor < -0.2) {
                  pump_inlet_jet_factor = pump_inlet_jet_factor + 0.025;
               }
               else {
                  pump_inlet_jet_factor = 0.0;
               }

             }
```

```
            //This section adds a pump discharge jet 246 ft back on the hull. It starts
out at full
            //force and diminishes to 0 at 20 ft out from the hull.It assumes water is
discharged in at 2.5 kts
            else if ((generationloop > dischargeBegin_m) && (generationloop <
dischargeEnd_m)) {
              nontubelevelgrid[current_distance][ij].x_magnitude =
pump_outlet_jet_factor
                                                          * pump_outlet_jet_speed;

              if (pump_outlet_jet_factor > 0.2) {
                 pump_outlet_jet_factor = pump_outlet_jet_factor - 0.025;
              }
              else {
                 pump_outlet_jet_factor = 0.0;
              }

            }


      //Now write these values to the proper file
      switch ((int)submarine_speed) {
         case 1:
            plate1ktOutput << current_distance << " " << ij << " "
                           << nontubelevelgrid[current_distance][ij].x_magnitude << " "
                           << nontubelevelgrid[current_distance][ij].y_magnitude << " "
                           << nontubelevelgrid[current_distance][ij].z_magnitude << " "
                           << nontubelevelgrid[current_distance][ij].direction
                           << endl;

                  //Update the global array
                  global1ktgrid[current_distance][ij].x_magnitude =
nontubelevelgrid[current_distance][ij].x_magnitude;
                  global1ktgrid[current_distance][ij].y_magnitude =
nontubelevelgrid[current_distance][ij].y_magnitude;
                  global1ktgrid[current_distance][ij].z_magnitude =
nontubelevelgrid[current_distance][ij].z_magnitude;
                  global1ktgrid[current_distance][ij].direction   =
nontubelevelgrid[current_distance][ij].direction;

            break;
         case 2:
            plate2ktOutput << current_distance << " " << ij << " "
                           << nontubelevelgrid[current_distance][ij].x_magnitude << " "
                           << nontubelevelgrid[current_distance][ij].y_magnitude << " "
                           << nontubelevelgrid[current_distance][ij].z_magnitude << " "
                           << nontubelevelgrid[current_distance][ij].direction
                           << endl;

                  //Update the global array
                  global2ktgrid[current_distance][ij].x_magnitude =
nontubelevelgrid[current_distance][ij].x_magnitude;
                  global2ktgrid[current_distance][ij].y_magnitude =
nontubelevelgrid[current_distance][ij].y_magnitude;
                  global2ktgrid[current_distance][ij].z_magnitude =
nontubelevelgrid[current_distance][ij].z_magnitude;
                  global2ktgrid[current_distance][ij].direction   =
nontubelevelgrid[current_distance][ij].direction;

            break;
         case 3:
            plate3ktOutput << current_distance << " " << ij << " "
                           << nontubelevelgrid[current_distance][ij].x_magnitude << " "
                           << nontubelevelgrid[current_distance][ij].y_magnitude << " "
                           << nontubelevelgrid[current_distance][ij].z_magnitude << " "
```

```
                              << nontubelevelgrid[current_distance][ij].direction
                              << endl;

                     //Update the global array
                     global3ktgrid[current_distance][ij].x_magnitude =
nontubelevelgrid[current_distance][ij].x_magnitude;
                     global3ktgrid[current_distance][ij].y_magnitude =
nontubelevelgrid[current_distance][ij].y_magnitude;
                     global3ktgrid[current_distance][ij].z_magnitude =
nontubelevelgrid[current_distance][ij].z_magnitude;
                     global3ktgrid[current_distance][ij].direction   =
nontubelevelgrid[current_distance][ij].direction;

                  break;
               default:
                  cerr << "Invalid Submarine Speed" << endl;
                  break;
            }  // end switch


            pass = 1;
            last_distance_filled = current_distance;

            //-----------------------------------------------------------------
            //This section does file output for files that are used to visualize
            //field output over the whole sub length. They are generally used for
            //viewing only. This data is not in a UVW usable form
            if (U[ij] >= 0.99)   {
               plateProfileOutput      << XF << " "
                                       << Y[ij]
                                       << endl;
               //firstEntry = 0;
            }


            //Put output to file for flat plate slice at  50 ft
            if ((generationloop == 15) && (submarine_speed == 1))  {
               plateSlice50Output << Y[ij] << " " << U[ij]  << endl;
            }

            //Put output to file for flat plate slice at 100 ft
            if ((generationloop == 30) && (submarine_speed == 1))  {
               plateSlice100Output << Y[ij] << " " << U[ij] << endl;
            }
            //Put output to file for flat plate slice at 150 ft
            if ((generationloop == 45)  && (submarine_speed == 1)) {
               plateSlice150Output << Y[ij] << " " << U[ij]  << endl;
            }
            //Put output to file for flat plate slice at200 ft
            if ((generationloop == 60)  && (submarine_speed == 1)) {
               plateSlice200Output << Y[ij] << " " << U[ij]  << endl;
            }
            //Put output to file for flat plate slice at 250 ft
            if ((generationloop == 75)  && (submarine_speed == 1)) {
               plateSlice250Output << Y[ij] << " " << U[ij] << endl;
            }
         }

      } //end of generationloop
   } //end of submarine_speed loop



   //Close all output files
   plate1ktOutput.close();
```

```
   plate2ktOutput.close();
   plate3ktOutput.close();
   plateProfileOutput.close();
   plateSlice50Output.close();
   plateSlice100Output.close();
   plateSlice150Output.close();
   plateSlice200Output.close();
   plateSlice250Output.close();
   return;

}  //end of flatPlate function


//-------------------------------------------------------------------------
void  tubeLevelFlowFieldGenerator ( void ) {


   //-----------------------------------------------------------------
   //Arrays to hold all values as they are modified

   FlowGridElements above1kt [FLOWFIELDLENGTH][FLOWFIELDWIDTH];
   FlowGridElements upper1kt [FLOWFIELDLENGTH][FLOWFIELDWIDTH];
   FlowGridElements center1kt[FLOWFIELDLENGTH][FLOWFIELDWIDTH];
   FlowGridElements lower1kt [FLOWFIELDLENGTH][FLOWFIELDWIDTH];
   FlowGridElements below1kt [FLOWFIELDLENGTH][FLOWFIELDWIDTH];

   FlowGridElements above2kt [FLOWFIELDLENGTH][FLOWFIELDWIDTH];
   FlowGridElements upper2kt [FLOWFIELDLENGTH][FLOWFIELDWIDTH];
   FlowGridElements center2kt[FLOWFIELDLENGTH][FLOWFIELDWIDTH];
   FlowGridElements lower2kt [FLOWFIELDLENGTH][FLOWFIELDWIDTH];
   FlowGridElements below2kt [FLOWFIELDLENGTH][FLOWFIELDWIDTH];

   FlowGridElements above3kt [FLOWFIELDLENGTH][FLOWFIELDWIDTH];
   FlowGridElements upper3kt [FLOWFIELDLENGTH][FLOWFIELDWIDTH];
   FlowGridElements center3kt[FLOWFIELDLENGTH][FLOWFIELDWIDTH];
   FlowGridElements lower3kt [FLOWFIELDLENGTH][FLOWFIELDWIDTH];
   FlowGridElements below3kt [FLOWFIELDLENGTH][FLOWFIELDWIDTH];



   //-----------------------------------------------------------------
   //Output streams to hold the generated flow fields for later usage.
   //Five files are created for each sub speed to cover all major variations
   // in flow profile.
   //      abovetubelevel1kt.data - Holds the tube model flow field at
   //             1 ft above the tube using a submarine speed of 1 kt.
   //      uppertubelevel1kt.data - Holds the tube model flow field
   //            at the upper edge of the tube for a submarine speed  of 1 kt.
   //      centertubelevel1kt.data - Holds the tube model flow field at
   //            the center of the tube for a submarine speed of 3 kt.
   //      lowertubelevel1kt.data - Holds the tube model flow field at
   //            the lower edge of the tube for a submarine speed of 1 kt.
   //      belowtubelevel1kt.data - Holds the tube model flow field at
   //            1 ft below the tube using a submarine speed of 1 kt.
   //
   //  Files for other speeds are named using the same conventions.


   ofstream abovetubeLevel1ktOutput  ("abovetubelevel1kt.data", ios::out);
   ofstream upperLevel1ktOutput      ("uppertubelevel1kt.data", ios::out);
   ofstream centerLevel1ktOutput     ("centertubelevel1kt.data", ios::out);
   ofstream lowertubeLevel1ktOutput  ("lowertubelevel1kt.data", ios::out);
   ofstream belowLevel1ktOutput      ("belowtubelevel1kt.data", ios::out);
```

```
ofstream abovetubeLevel2ktOutput  ("abovetubelevel2kt.data", ios::out);
ofstream upperLevel2ktOutput      ("uppertubelevel2kt.data", ios::out);
ofstream centerLevel2ktOutput     ("centertubelevel2kt.data", ios::out);
ofstream lowertubeLevel2ktOutput  ("lowertubelevel2kt.data", ios::out);
ofstream belowLevel2ktOutput      ("belowtubelevel2kt.data", ios::out);

ofstream abovetubeLevel3ktOutput  ("abovetubelevel3kt.data", ios::out);
ofstream upperLevel3ktOutput      ("uppertubelevel3kt.data", ios::out);
ofstream centerLevel3ktOutput     ("centertubelevel3kt.data", ios::out);
ofstream lowertubeLevel3ktOutput  ("lowertubelevel3kt.data", ios::out);
ofstream belowLevel3ktOutput      ("belowtubelevel3kt.data", ios::out);


//-------------------------------------------------------------
//Initialize the tube level flow fields to those of the flat
//Plate fields
for (int row = 0; row < FLOWFIELDLENGTH; row++)  {
   for (int col = 0; col < FLOWFIELDWIDTH; col++)  {

      above1kt [row][col].x_magnitude =  global1ktgrid[row][col].x_magnitude ;
      above1kt [row][col].y_magnitude =  global1ktgrid[row][col].y_magnitude ;
      above1kt [row][col].z_magnitude =  global1ktgrid[row][col].z_magnitude ;
      above1kt [row][col].direction   =  global1ktgrid[row][col].direction ;

      upper1kt [row][col].x_magnitude =  global1ktgrid[row][col].x_magnitude ;
      upper1kt [row][col].y_magnitude =  global1ktgrid[row][col].y_magnitude ;
      upper1kt [row][col].z_magnitude =  global1ktgrid[row][col].z_magnitude ;
      upper1kt [row][col].direction   =  global1ktgrid[row][col].direction ;

      center1kt [row][col].x_magnitude =  global1ktgrid[row][col].x_magnitude ;
      center1kt [row][col].y_magnitude =  global1ktgrid[row][col].y_magnitude ;
      center1kt [row][col].z_magnitude =  global1ktgrid[row][col].z_magnitude ;
      center1kt [row][col].direction   =  global1ktgrid[row][col].direction ;

      lower1kt [row][col].x_magnitude =  global1ktgrid[row][col].x_magnitude ;
      lower1kt [row][col].y_magnitude =  global1ktgrid[row][col].y_magnitude ;
      lower1kt [row][col].z_magnitude =  global1ktgrid[row][col].z_magnitude ;
      lower1kt [row][col].direction   =  global1ktgrid[row][col].direction ;

      below1kt [row][col].x_magnitude =  global1ktgrid[row][col].x_magnitude ;
      below1kt [row][col].y_magnitude =  global1ktgrid[row][col].y_magnitude ;
      below1kt [row][col].z_magnitude =  global1ktgrid[row][col].z_magnitude ;
      below1kt [row][col].direction   =  global1ktgrid[row][col].direction ;


      above2kt [row][col].x_magnitude =  global2ktgrid[row][col].x_magnitude ;
      above2kt [row][col].y_magnitude =  global2ktgrid[row][col].y_magnitude ;
      above2kt [row][col].z_magnitude =  global2ktgrid[row][col].z_magnitude ;
      above2kt [row][col].direction   =  global2ktgrid[row][col].direction ;

      upper2kt [row][col].x_magnitude =  global2ktgrid[row][col].x_magnitude ;
      upper2kt [row][col].y_magnitude =  global2ktgrid[row][col].y_magnitude ;
      upper2kt [row][col].z_magnitude =  global2ktgrid[row][col].z_magnitude ;
      upper2kt [row][col].direction   =  global2ktgrid[row][col].direction ;

      center2kt [row][col].x_magnitude =  global2ktgrid[row][col].x_magnitude ;
      center2kt [row][col].y_magnitude =  global2ktgrid[row][col].y_magnitude ;
      center2kt [row][col].z_magnitude =  global2ktgrid[row][col].z_magnitude ;
      center2kt [row][col].direction   =  global2ktgrid[row][col].direction ;

      lower2kt [row][col].x_magnitude =  global2ktgrid[row][col].x_magnitude ;
      lower2kt [row][col].y_magnitude =  global2ktgrid[row][col].y_magnitude ;
      lower2kt [row][col].z_magnitude =  global2ktgrid[row][col].z_magnitude ;
      lower2kt [row][col].direction   =  global2ktgrid[row][col].direction ;
```

```
      below2kt [row][col].x_magnitude =  global2ktgrid[row][col].x_magnitude ;
      below2kt [row][col].y_magnitude =  global2ktgrid[row][col].y_magnitude ;
      below2kt [row][col].z_magnitude =  global2ktgrid[row][col].z_magnitude ;
      below2kt [row][col].direction   =  global2ktgrid[row][col].direction ;


      above3kt [row][col].x_magnitude =  global3ktgrid[row][col].x_magnitude ;
      above3kt [row][col].y_magnitude =  global3ktgrid[row][col].y_magnitude ;
      above3kt [row][col].z_magnitude =  global3ktgrid[row][col].z_magnitude ;
      above3kt [row][col].direction   =  global3ktgrid[row][col].direction ;

      upper3kt [row][col].x_magnitude =  global3ktgrid[row][col].x_magnitude ;
      upper3kt [row][col].y_magnitude =  global3ktgrid[row][col].y_magnitude ;
      upper3kt [row][col].z_magnitude =  global3ktgrid[row][col].z_magnitude ;
      upper3kt [row][col].direction   =  global3ktgrid[row][col].direction ;

      center3kt [row][col].x_magnitude =  global3ktgrid[row][col].x_magnitude ;
      center3kt [row][col].y_magnitude =  global3ktgrid[row][col].y_magnitude ;
      center3kt [row][col].z_magnitude =  global3ktgrid[row][col].z_magnitude ;
      center3kt [row][col].direction   =  global3ktgrid[row][col].direction ;

      lower3kt [row][col].x_magnitude =  global3ktgrid[row][col].x_magnitude ;
      lower3kt [row][col].y_magnitude =  global3ktgrid[row][col].y_magnitude ;
      lower3kt [row][col].z_magnitude =  global3ktgrid[row][col].z_magnitude ;
      lower3kt [row][col].direction   =  global3ktgrid[row][col].direction ;

      below3kt [row][col].x_magnitude =  global3ktgrid[row][col].x_magnitude ;
      below3kt [row][col].y_magnitude =  global3ktgrid[row][col].y_magnitude ;
      below3kt [row][col].z_magnitude =  global3ktgrid[row][col].z_magnitude ;
      below3kt [row][col].direction   =  global3ktgrid[row][col].direction ;

  }  //End of col loop
}   //End of Row loop


//-------------------------------------------------------------
//Update the tube level flow fields to show the tube flow
//disturbances
double before_tube_force = 1.0;
double after_tube_force  = -1.0;

for (int along_hull = 30; along_hull <= 60; along_hull++) {
   before_tube_force = 1.0;
   for (int out_from_hull = 0; out_from_hull <= 30 ; out_from_hull++)  {

      above1kt [along_hull][out_from_hull].x_magnitude =   before_tube_force * 1 ;
      upper1kt [along_hull][out_from_hull].x_magnitude =   before_tube_force * 1 ;
      center1kt[along_hull][out_from_hull].x_magnitude =   before_tube_force * 1 ;
      lower1kt [along_hull][out_from_hull].x_magnitude =   before_tube_force * 1 ;
      below1kt [along_hull][out_from_hull].x_magnitude =   before_tube_force * 1 ;
      above2kt [along_hull][out_from_hull].x_magnitude =   before_tube_force * 2 ;
      upper2kt [along_hull][out_from_hull].x_magnitude =   before_tube_force * 2 ;
      center2kt[along_hull][out_from_hull].x_magnitude =   before_tube_force * 2 ;
      lower2kt [along_hull][out_from_hull].x_magnitude =   before_tube_force * 2 ;
      below2kt [along_hull][out_from_hull].x_magnitude =   before_tube_force * 2 ;
      above3kt [along_hull][out_from_hull].x_magnitude =   before_tube_force * 3 ;
      upper3kt [along_hull][out_from_hull].x_magnitude =   before_tube_force * 3 ;
      center3kt[along_hull][out_from_hull].x_magnitude =   before_tube_force * 3 ;
      lower3kt [along_hull][out_from_hull].x_magnitude =   before_tube_force * 3 ;
      below3kt [along_hull][out_from_hull].x_magnitude =   before_tube_force * 3 ;

      before_tube_force = before_tube_force - 0.032;
   }
}
```

```
for ( along_hull = 61; along_hull <= 80; along_hull++) {
   after_tube_force = -1.0;
   for (int out_from_hull = 0; out_from_hull <= 30 ; out_from_hull++)  {

      above1kt [along_hull][out_from_hull].x_magnitude =   after_tube_force * 1 ;
      upper1kt [along_hull][out_from_hull].x_magnitude =   after_tube_force * 1 ;
      center1kt[along_hull][out_from_hull].x_magnitude =   after_tube_force * 1 ;
      lower1kt [along_hull][out_from_hull].x_magnitude =   after_tube_force * 1 ;
      below1kt [along_hull][out_from_hull].x_magnitude =   after_tube_force * 1 ;
      above2kt [along_hull][out_from_hull].x_magnitude =   after_tube_force * 2 ;
      upper2kt [along_hull][out_from_hull].x_magnitude =   after_tube_force * 2 ;
      center2kt[along_hull][out_from_hull].x_magnitude =   after_tube_force * 2 ;
      lower2kt [along_hull][out_from_hull].x_magnitude =   after_tube_force * 2 ;
      below2kt [along_hull][out_from_hull].x_magnitude =   after_tube_force * 2 ;
      above3kt [along_hull][out_from_hull].x_magnitude =   after_tube_force * 3 ;
      upper3kt [along_hull][out_from_hull].x_magnitude =   after_tube_force * 3 ;
      center3kt[along_hull][out_from_hull].x_magnitude =   after_tube_force * 3 ;
      lower3kt [along_hull][out_from_hull].x_magnitude =   after_tube_force * 3 ;
      below3kt [along_hull][out_from_hull].x_magnitude =   after_tube_force * 3 ;

      after_tube_force = after_tube_force + 0.032;
   }
}




//-------------------------------------------------------------
//Output the flow field arrays to the proper files
for (int row1 = 0; row1 < FLOWFIELDLENGTH; row1++)  {
   for (int col1 = 0; col1 < FLOWFIELDWIDTH; col1++)  {

      abovetubeLevel1ktOutput << row1 << " " << col1 << " "
                              << above1kt [row1][col1].x_magnitude  << " "
                              << above1kt [row1][col1].y_magnitude  << " "
                              << above1kt [row1][col1].z_magnitude  << " "
                              << above1kt [row1][col1].direction    << endl;

      upperLevel1ktOutput     << row1 << " " << col1 << " "
                              << upper1kt [row1][col1].x_magnitude  << " "
                              << upper1kt [row1][col1].y_magnitude  << " "
                              << upper1kt [row1][col1].z_magnitude  << " "
                              << upper1kt [row1][col1].direction    << endl;

      centerLevel1ktOutput    << row1 << " " << col1 << " "
                              << center1kt [row1][col1].x_magnitude << " "
                              << center1kt [row1][col1].y_magnitude << " "
                              << center1kt [row1][col1].z_magnitude << " "
                              << center1kt [row1][col1].direction   << endl;

      lowertubeLevel1ktOutput << row1 << " " << col1 << " "
                              << lower1kt [row1][col1].x_magnitude << " "
                              << lower1kt [row1][col1].y_magnitude << " "
                              << lower1kt [row1][col1].z_magnitude << " "
                              << lower1kt [row1][col1].direction   << endl;

      belowLevel1ktOutput     << row1 << " " << col1 << " "
                              << below1kt [row1][col1].x_magnitude << " "
                              << below1kt [row1][col1].y_magnitude << " "
                              << below1kt [row1][col1].z_magnitude << " "
                              << below1kt [row1][col1].direction   << endl;

      abovetubeLevel2ktOutput << row1 << " " << col1 << " "
                              << above2kt [row1][col1].x_magnitude  << " "
```

```
                              << above2kt [row1][col1].y_magnitude  << " "
                              << above2kt [row1][col1].z_magnitude  << " "
                              << above2kt [row1][col1].direction    << endl;

      upperLevel2ktOutput     << row1 << " " << col1 << " "
                              << upper2kt [row1][col1].x_magnitude  << " "
                              << upper2kt [row1][col1].y_magnitude  << " "
                              << upper2kt [row1][col1].z_magnitude  << " "
                              << upper2kt [row1][col1].direction    << endl;

      centerLevel2ktOutput    << row1 << " " << col1 << " "
                              << center2kt [row1][col1].x_magnitude << " "
                              << center2kt [row1][col1].y_magnitude << " "
                              << center2kt [row1][col1].z_magnitude << " "
                              << center2kt [row1][col1].direction   << endl;

      lowertubeLevel2ktOutput << row1 << " " << col1 << " "
                              << lower2kt [row1][col1].x_magnitude << " "
                              << lower2kt [row1][col1].y_magnitude << " "
                              << lower2kt [row1][col1].z_magnitude << " "
                              << lower2kt [row1][col1].direction   << endl;

      belowLevel2ktOutput     << row1 << " " << col1 << " "
                              << below2kt [row1][col1].x_magnitude << " "
                              << below2kt [row1][col1].y_magnitude << " "
                              << below2kt [row1][col1].z_magnitude << " "
                              << below2kt [row1][col1].direction   << endl;

      abovetubeLevel3ktOutput << row1 << " " << col1 << " "
                              << above3kt [row1][col1].x_magnitude  << " "
                              << above3kt [row1][col1].y_magnitude  << " "
                              << above3kt [row1][col1].z_magnitude  << " "
                              << above3kt [row1][col1].direction    << endl;

      upperLevel3ktOutput     << row1 << " " << col1 << " "
                              << upper3kt [row1][col1].x_magnitude  << " "
                              << upper3kt [row1][col1].y_magnitude  << " "
                              << upper3kt [row1][col1].z_magnitude  << " "
                              << upper3kt [row1][col1].direction    << endl;

      centerLevel3ktOutput    << row1 << " " << col1 << " "
                              << center3kt [row1][col1].x_magnitude << " "
                              << center3kt [row1][col1].y_magnitude << " "
                              << center3kt [row1][col1].z_magnitude << " "
                              << center3kt [row1][col1].direction   << endl;

      lowertubeLevel3ktOutput << row1 << " " << col1 << " "
                              << lower3kt [row1][col1].x_magnitude << " "
                              << lower3kt [row1][col1].y_magnitude << " "
                              << lower3kt [row1][col1].z_magnitude << " "
                              << lower3kt [row1][col1].direction   << endl;

      belowLevel3ktOutput     << row1 << " " << col1 << " "
                              << below3kt [row1][col1].x_magnitude << " "
                              << below3kt [row1][col1].y_magnitude << " "
                              << below3kt [row1][col1].z_magnitude << " "
                              << below3kt [row1][col1].direction   << endl;


  }  //End of col1 loop
}   //End of Row1 loop

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

```
   //Close all Files for later use by dynamics/gnuplot
   abovetubeLevel1ktOutput.close();
   upperLevel1ktOutput.close();
   centerLevel1ktOutput.close();
   lowertubeLevel1ktOutput.close();
   belowLevel1ktOutput.close();

   abovetubeLevel2ktOutput.close();
   upperLevel2ktOutput.close();
   centerLevel2ktOutput.close();
   lowertubeLevel2ktOutput.close();
   belowLevel2ktOutput.close();

   abovetubeLevel3ktOutput.close();
   upperLevel3ktOutput.close();
   centerLevel3ktOutput.close();
   lowertubeLevel3ktOutput.close();
   belowLevel3ktOutput.close();


   return;
}


//-------------------------------------------------------------------------
//This is the driver to run the  flateplate flow generation and
//tube level flow generation functions
main () {

        cout << "Starting the Flow Field Generation program." << endl;


        cout << "Generating the Flow Profiles for the Flat Plate Model Area." << endl;
        flatPlateFlowFieldGenerator ( );

        cout << endl << endl;
        cout << "The following File(s) were created for use by the Phoenix AUV UVW: " <<
endl;
        cout << "                        flatplateflowfield1kt.data" << endl;
        cout << "                        flatplateflowfield2kt.data" << endl;
        cout << "                        flatplateflowfield3kt.data" << endl;

        cout << "                        flatprofile.data" << endl;
        cout << "                        flatslice50.data"   << endl;
        cout << "                        flatslice100.data"   << endl;
        cout << "                        flatslice150.data"   << endl;
        cout << "                        flatslice200.data"   << endl;
        cout << "                        flatslice250.data"   << endl;
        cout << endl << endl;



        cout << "Creating the Flow Profiles for the Tube Level Flow Areas." << endl;
        tubeLevelFlowFieldGenerator ();
        cout << "The following File(s) were created for use by the Phoenix AUV UVW: " <<
endl;
        cout << "                        abovetubelevel1kt.data" << endl;
        cout << "                        uppertubelevel1kt.data" << endl;
        cout << "                        centertubelevel1kt.data" << endl;
        cout << "                        lowertubelevel1kt.data" << endl;
        cout << "                        belowtubelevel1kt.data" << endl;
        cout << endl;
        cout << "                        abovetubelevel2kt.data" << endl;
        cout << "                        uppertubelevel2kt.data" << endl;
        cout << "                        centertubelevel2kt.data" << endl;
```

```
        cout << "                       lowertubelevel2kt.data" << endl;
        cout << "                       belowtubelevel2kt.data" << endl;
        cout << endl;
        cout << "                       abovetubelevel3kt.data" << endl;
        cout << "                       uppertubelevel3kt.data" << endl;
        cout << "                       centertubelevel3kt.data" << endl;
        cout << "                       lowertubelevel3kt.data" << endl;
        cout << "                       belowtubelevel3kt.data" << endl;
        cout << endl << endl;


        cout << "Exiting the Flow Field Generation Program." << endl;

      return 0;
}  // end main
```

APPENDIX E. SIMULATION VIDEO

1. INTRODUCTION

The attached video appendix gives an overall view of the Phoenix AUV virtual environment. All major objects are described and viewed.

2. SURFACE BUOYANCY AND WAVE MOTION

In this segment the AUV is run on a course into the seas in various sea states. The test runs demonstrate a sea state of 1, 3, and 5 respectively.

3. PUMP OUTLETS/INLETS

This part of the demonstration shows the AUV driving past a pump discharge outlet followed by a pump suction inlet. It demonstrates how the effects of turbulent flow are felt by the AUV, and how the AUV maintains stability and continues on the preplanned course.

4. COMPLETE MISSION

This is the final portion of the simulation tape. It shows a complete torpedo tube launch and recovery mission. The AUV is launched from a lower port torpedo tube, proceeds into the open water, takes position at the submarines stern and then conducts a docking evolution with the upper port torpedo tube. Both inward and outward outer door openings are assumed, ans simply represented using cylinders.

5. INVOCATION INSTRUCTIONS

To reproduce this mission the following steps should be taken.

A. Start the viewer application as follows.

```
SGI> viewer
```

2. Start the dynamics portion of the program as follows

```
SGI> dynamics
```

OR

SGI> **dynamics_nosonar**

*****Insert dynamics Menu Capture*********

C. After dynamics is running select the option to conduct a torpedo tube docking evolution (it is letter "z").

D. Once all flow field arrays are initialized, select "l" to loop the dynamics with the execution level.

E. Finally, launch the execution application as follows:

SGI> **execution mission mission.script.FlowFieldGenerator remote <dynamics host name>**

F. You should now observe a torpedo tube launch and recovery mission in the viewer.

LIST OF REFERENCES

Ames, Andrea L., Nadeau, David R., Moreland, John L., *VRML 2.0 Sourcebook*, Second edition, John Wiley & Sons, New York, New York, 1997.

Bacon, Daniel K. Jr., *Integration of a Submarine into NPSNET*, Master's Thesis, Naval Postgraduate School, Monterey, California September 1995.

Berteaux, H.O., *Buoy Engineering*, John Wiley & Sons, New York, New York, 1976.

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

Brutzman, Don, The "Virtual Reality Modeling Language and Java," Communication of the ACM, Special issue on the Java programming language, to appear 1998. Available at *http://www.stl.nps.navy.mil/~brutzman/vrml/vrmljava.ps*

Brutzman, Don, Healey, Tony, Marco, Dave, and McGhee, Bob, "The *Phoenix* Autonomous Underwater Vehicle," *AI-Based Mobile Robots*, editors David Kortenkamp, Pete Bonasso and Robin Murphy, MIT/AAAI Press, Cambridge Massachusetts, to appear 1998. Available at *http://www.stl.nps.navy.mil/~auv/aimr.ps*

Burns, Mike, *Merging Virtual and Real Execution Level Software for the Phoenix Autonomus Underwater Vehicle*, Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1996. Available at *http://www.cs.nps.navy.mil/research/auv/thesispages/burns/cover.html*

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

Byrnes, Ronald B., Healey, Anthony J., McGhee, Robert B., Nelson, Michael L., Kwak, Se-Hung, Brutzman, Donald P., *The Rational Behavior Software Architecture for Intelligent Ships*, Naval Engineers Journal, March, 1996, pp. 43-54.

Cornell, G., and Hortsman, C., *Core JAVA*, SunSoft Press, Mountain View, California, 1997.

Davis, Duane, *Precision Maneuvering and Control of the Phoenix Autonomous Underwater Vehicle for Entering a Recovery Tube*, Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1996. Available at *http://www.cs.nps.navy.mil/research/auv/thesispages/davis/cover.html*

DelTheil, Caroline, Didier Leandri, Eric Hospital, Donald P. Brutzman, *An Optical Guidance System for the Recovery of an Unmanned Underwater Vehicle*, in proceedings from the Tenth International

Symposium on Unmanned Untethered Submersible Technology (UUST), Lee, NH, September 7-10, 1997, pp. 140-148.

IEEE Standard for Information Technology - Protocols for Distributed Interactive Simulation (DIS) Applications, version 2.0, Institute for Simulation and Training report IST-CR-93-15, University of Central Florida, Orlando Florida, May 28 1993.

IEEE Standard for Distributed Interactive Simulation -- Applications Protocols, Standards Proposal P1278.1 ballot draft, IEEE Standards Department, Piscataway New Jersey, November 1994.

IEEE Standard for Distributed Interactive Simulation -- Communication Architecture Requirements, Standards Proposal P1278.2 ballot draft, IEEE Standards Department, Piscataway New Jersey, November 1994.

Fossen, Thor, *Guidance and Control of Ocean Vehicles*, John Wiley & Sons, New York, New York, 1990.

Hartman, J., and Wernecke, J., *The VRML 2.0 Handbook*, Addison-Wesley Publishing Company, New York, New York, 1997.

Healey, A.J. and Lienard, D., "Multivariable Sliding Mode Control for Autonomous Diving and Steering of Unmanned Underwater Vehicles," *IEEE Journal of Oceanic Engineering*, vol. 18, no. 3, July 1993, pp. 327-339.

Healey, Anthony J., "Dynamics of Marine Vehicles," Course Notes, Naval Postgraduate School, Monterey California, Winter 1998.

Holden, Mike, *Ada Implementation of Concurrent Execution of Multiple Tasks in the Strategic and Tactical Levels in the Rational Behavior Model for the NPS Phoenix Autonomous Underwater Vehicle (AUV)*, Masters Thesis, Naval Postgraduate School, Monterey, California, September 1995.

John, James E.A. and Haberman, William L., *Introduction to Fluid Mechanics*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

Lea, R., Matsuda, K., and Miyashita, K., *JAVA for 3D and VML Worlds*, New Riders Publishing, Indianapolis, Indiana, 1997.

Leaver, Greg, *Monterey Bay Natural Marine Sanctuary Terrain Model*, Masters Thesis, Naval Postgraduate School, Monterey, California, June 1998.

Marco, D.B., Autonomous Control of Underwater Vehicles and Local Area Maneuvering, Dissertation, Naval Postgraduate School, Monterey, California, September 1996. Available at *http://www.cs.nps.navy.mil/research/auv*

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

Mecco, *Tritech DS30 Precision Doppler Sonar Operators Manual*, Duval, Washington, December. 1997.

Naughton, P., *The JAVA Handbook*, Osborne McGraw-Hill, Berkeley, California, 1997.

Riedel, J.S., Healey, A.J., *A Discrete Filter for the Forward Prediction of Sea Wave Effects on AUV Motions*, Naval Postgraduate School, Monterey, California September 1997.

Schetz, Joseph A., *Boundary Layer Analysis*, Prentice Hall, Englewood Cliffs, New Jersey, October 1992.

Scientific Computing Group at Indiana University, Web Pages at *http://www.cs.indiana.edu/scicomp/home.html,* Bloomington, Indiana, 1998.

SonTek, *SonTek Acoustic Doppler Velocimeter Introductory Documentation*, San Diego, California, June, 1997.

Young, Forest, *Phoenix Autonomous Underwater Vehicle (AUV): Networked Control of Multiple Analog and Digital Devices using Lontalk*, Master's Thesis, Naval Postgraduate School, Monterey, California, December, 1997.

INITIAL DISTRIBUTION LIST

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

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

3. Dr. Dan Boger, Chair, Code CS.....2
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943

4. Dr. Robert B. McGhee, Code CS/MZ.....1
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943

5. Dr. Donald P. Brutzman, Code UW/BR.....1
   Undersea Warfare Academic Group
   Naval Postgraduate School
   Monterey, California 93943

6. Kevin M. Byrne.....2
   49 Forest Ave.
   Oakdale, New York 11769

7. Dr. Mike Zyda, Code CS/MZ.....1
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943

8. CDR Mike Holden,USN, Code UW/HO.....1
   Undersea Warfare Academic Group
   Naval Postgraduate School
   Monterey, California 93943

9. Dr. Anthony Healey, Code ME/HY.....1
Mechanical Engineering Department
Naval Postgraduate School
Monterey, California 93943

10. Dr. Dave Marco, Code ME/MA.....1
Mechanical Engineering Department
Naval Postgraduate School
Monterey, California 93943

11. Dr. Jim Eagle, Code UW.....1
Undersea Warfare Academic Group
Naval Postgraduate School
Monterey, California 93943

12. Commander Naval Undersea Warfare Center Division.....1
1176 Howell Street
Attn: Erik Chaum, Code 2251, Building 1171-3
Combat Systems Engineering and Analysis Laboratory (CSEAL)
Newport, Rhode Island 02841-1708

13. Dr. James Bellingham.....1
Underwater Vehicles Laboratory, MIT Sea Grant College Program
Northeastern University
East Point, Nahant, Massachusetts 01908

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

15. Caroline DelTheil.....1
DGA-DCN, Center Technique des Systemes Navals
Dissuasion Lutte Sous Marine
DCN Toulon, BP 28, 83800 Toulon-Naval
France

16. Dr. John Leonard.....1
Underwater Vehicles Laboratory, MIT Sea Grant College Program
292 Main Street
Massachusetts Institute of Technology
Cambridge, Massachusetts 02142

17. Pr. Didier Leandri 1
Laboratoire LCPSI
Ecole Nationale d'Ingenieurs de Tarbes
47 av d'Azereix
65000 Tarbes
France

18. LTJG Barney Kossman, USN 1
Science Officer
Commander Submarine Development Squadron
137 Sylvester Road
San Diego, CA 92106

19. Dr. Naomi Leonard 1
Underwater Vehicles Laboratory, MIT Sea Grant College Program
292 Main Street
Massachusetts Institute of Technology
Cambridge, Massachusetts 02142

20. LCDR Jeff Reidel, USN, Code ME/RE 1
Mechanical Engineering Department
Naval Postgraduate School
Monterey, California 93943

21. CAPT Victor Fiebig, USN 1
PEO (UW) PMS 403
Crystal Park 1, Suite 817
Arlington, VA 22202

22. CAPT Jay Kistler, USN 1
N6M
2000 Navy Pentagon
Room 4C445
Washington, D.C. 20350-2000

23. George Phillips 1
CNO, N6M1
2000 Navy Pentagon
Room 4C445
Washington, D.C. 20350-2000