Adaptive Rapid Environmental Assessment System Simulation Framework

by

Ding Wang

B.A. in Automotive Engineering (1997)

and

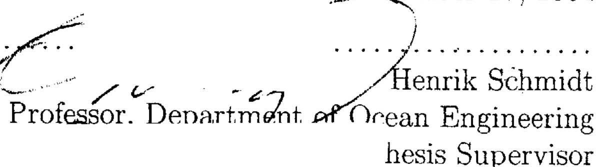
S.M. in Physics (2000)

Tsinghua University, Beijing, China

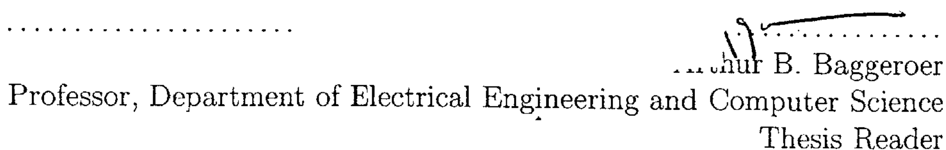
Submitted to the Department of Ocean Engineering and the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of
Master of Science in Ocean Engineering
and
Master of Science in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 2005

© Massachusetts Institute of Technology 2005. All rights reserved.

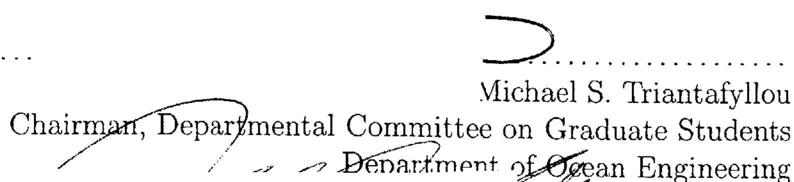
Author ....
Department of Ocean Engineering
December 17, 2004



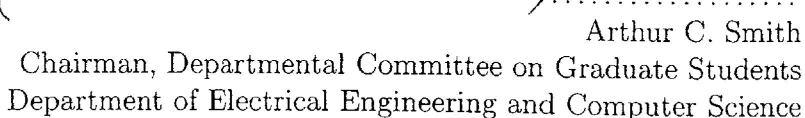
Certified by ....
Henrik Schmidt
Professor, Department of Ocean Engineering
Thesis Supervisor



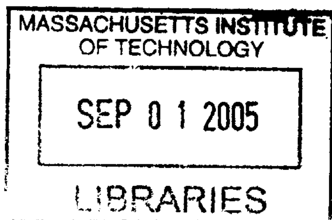
Certified by ....
Arthur B. Baggeroer
Professor, Department of Electrical Engineering and Computer Science
Thesis Reader



Accepted by ....
Michael S. Triantafyllou
Chairman, Departmental Committee on Graduate Students
Department of Ocean Engineering



Accepted by ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students
Department of Electrical Engineering and Computer Science



MASSACHUSETTS INSTITUTE OF TECHNOLOGY
SEP 0 1 2005
LIBRARIES

BARKER

Adaptive Rapid Environmental Assessment System Simulation Framework

by

Ding Wang

Submitted to the Department of Ocean Engineering
and the Department of Electrical Engineering and Computer Science
on December 17, 2004, in partial fulfillment of the
requirements for the degrees of
Master of Science in Ocean Engineering
and
Master of Science in Electrical Engineering and Computer Science

Abstract

Adaptive Rapid Environmental Assessment (AREA) is a new concept for minimizing the non model-based sonar performance prediction uncertainty and improving the model-based sonar performance by adaptive and rapid in situ measurement in the ocean environment. In this thesis, a possible structure of the AREA system has been developed; an AREA System Simulation Framework has been constructed using C++, which can simulate how AREA system will work and be utilized to determine the optimal or sub-optimal sampling strategies. A user's manual for the simulation framework, and specifications of all important C++ classes are included.

Thesis Supervisor: Henrik Schmidt
Title: Professor, Department of Ocean Engineering

Thesis Reader: Arthur B. Baggeroer
Title: Professor, Department of Electrical Engineering and Computer Science

Acknowledgments

First and foremost I would like to thank my supervisor Prof. Henrik Schmidt for all of the guidance, ideas and support that he has graciously provided throughout the course of this thesis. He always gave me courage and confidence when I felt perplexed and depressed. I want to also thank Prof. Arthur Baggeroer for reading this thesis.

Much of my graduate student's time was spent in my office and interacting with fellow students, and I would like to acknowledge a few in particular: George Dikos, a genius, for giving me the very important inspiration; Harish Mukundan, another genius, for helping me a lot about Adobe Acrobat and Latex; Da Guo, for sharing his thesis experience with me.

None of this would have been possible without the support and confidence of my parents. I haven't been back to my home town for almost eight years. I am so grateful for their understanding. The love and support from my girl friend, Ling Li, has seen me through all stages of this thesis, from discussions about methodologies of doing research to biology and artificial intelligence development.

Finally, I would like to thank the Office of Naval Research for providing funding for this project under the Capturing Uncertainty DRI.

Contents

1 Introduction **12**

2 AREA Concept **17**

3 AREA Simulation Framework **23**

4 How To Install And Use AREA Simulation Framework **30**

4.1 Set Up 30

4.2 Examples 36

5 Summary **41**

A Classes and Files **42**

A.1 AVU_ASD.h 43

A.1.1 Data Members 43

A.1.2 Member Functions & Operators 44

A.2 AVU_SSD.h 46

A.2.1 Data Members 46

A.2.2 Member Functions & Operators 47

A.3 AVU_SSD_ASD.h 49

A.3.1 Data Members 49

A.3.2 Member Functions & Operators 50

A.4 Bathymetry.h 52

A.4.1 Data Members 52

A.4.2 Member Functions & Operators 53
A.5 Candidate_points_generate.h 55
A.5.1 Functions Defined In This File 55
A.6 ControlAgent.h 57
A.6.1 Data Members 57
A.6.2 Member Functions & Operators 58
A.7 DetectionRange.h 61
A.7.1 Data Members 61
A.7.2 Member Functions & Operators 62
A.8 FixedWaterSensor.h 67
A.8.1 Data Members 67
A.8.2 Member Functions & Operators 68
A.9 fmat.h 70
A.9.1 Data Members 71
A.9.2 Member Functions & Operators 71
A.9.3 Functions And Operators Defined In This File 76
A.10 Greedy_algorithm.h 80
A.10.1 Functions Defined In This File 80
A.11 MatchedFieldProcessing.h 82
A.11.1 Data Members 82
A.11.2 Member Functions & Operators 83
A.12 ObjectiveAnalysis.h 86
A.12.1 Data Members 86
A.12.2 Member Functions & Operators 87
A.13 ObservationDatabase.h 89
A.13.1 Data Members 89
A.13.2 Member Functions & Operators 90
A.14 OceanPredictor.h 92
A.14.1 Data Members 92
A.14.2 Member Functions & Operators 93

A.15 PerfectSeabedDetector.h 95
A.15.1 Data Members 95
A.15.2 Member Functions & Operators 96
A.16 ram.h 99
A.16.1 Public Member Functions 99
A.17 Random.h 101
A.17.1 Data Members 101
A.17.2 Member Functions & Operators 101
A.18 Rollout.h 103
A.18.1 Functions Defined In This File 103
A.19 Search.h 105
A.19.1 Functions Defined In This File 105
A.20 SimulatedOcean.h 110
A.20.1 Data Members 110
A.20.2 Member Functions & Operators 111
A.21 SonarArray.h 116
A.21.1 Data Members 116
A.21.2 Member Functions & Operators 116
A.22 Sonar_Performance.h 119
A.22.1 Data Members 119
A.22.2 Member Functions & Operators 120
A.23 Sonar_SPC.h 123
A.23.1 Data Members 123
A.23.2 Member Functions & Operators 124
A.24 SoundField.h 127
A.24.1 Data Members 127
A.24.2 Member Functions & Operators 128
A.25 SoundSpeedGenerator.h 133
A.25.1 Data Members 133
A.25.2 Member Functions & Operators 134

A.26 StandardEnvironmentInfo.h 135
A.26.1 Data Members 135
A.26.2 Member Functions & Operators 136
A.27 StandardRamInfo.h 138
A.27.1 Data Members 139
A.27.2 Member Functions & Operators 140
A.27.3 Functions And Operators Defined In This File 140
A.28 Surveillance.h 142
A.28.1 Functions Defined In This File 142
A.29 SyntheticSeabed.h 145
A.29.1 Data Members 145
A.29.2 Member Functions & Operators 146
A.30 SyntheticStochasticWater.h 149
A.30.1 Data Members 149
A.30.2 Member Functions & Operators 150
A.31 SyntheticWater.h 154
A.31.1 Data Members 154
A.31.2 Member Functions & Operators 155
A.32 Total_cost.h 157
A.32.1 Functions Defined In This File 157
A.33 vec.h 161
A.33.1 Data Members 161
A.33.2 Member Functions & Operators 162
A.33.3 Functions And Operators Defined In This File 166

List of Figures

1-1 Illustration of how environmental uncertainties influences sonar performance. (a) shows an estimated water sound speed profile in shelf break region. (b) is the corresponding error field. We assume water sound speed profile is a Gaussian stochastic process. (c) shows non model-based sonar performance prediction uncertainty, where transmission loss and detection range are selected as sonar performance metric. In this case, the receiver is at 10m far from origin and 70m depth; a series of 50Hz source are at 50m depth and far to 10km; the detectable threshold is 65 dB. In this figure, strong TL uncertainty and detection range uncertainty can be observed. (d) shows model-based sonar performance, where a MFP sonar is used to localize a target at 15km range and 50m depth. In this case, we assume there is no ambient noise. In this figure, we can see that MFP localizations are usually several km away form the true target's location. 14

1-2 Multi-scale environmental assessment. The typical sonar systems performance is dependent on acoustic environment variability over a wide range of scales. Optimal environmental assessment will therefore be a compromise between conflicting requirements of coverage and resolution. By targeting areas of high sensitivity to the sonar system through in situ measurements, the deterministic assessment range will be shifted towards smaller scales. 15

2-1 Illustration of Adaptive Rapid Environmental Assessment System . . 17

2-2 Adaptive Rapid Environmental Assessment System wiring diagram . 18
2-3 Sequential diagram of *Adaptive Sampling Loop.* At each stage, the Observation Database will be first updated and the Ocean Predictor will do analysis; then the Control Agent will determine next sampling locations; following those commands, mobile sensors will do in situ measurements and new measurement results will be obtained, by which the Observation Database will be updated again. Repeating the *Adaptive Sampling Loop*, sampling points locations will be determined sequentially based on all the newest observation. 21
2-4 Sequential diagram of *Dynamic Programming.* 22

3-1 AREA simulation framework wiring diagram 24
3-2 Flow chart of AREA.cpp 24
3-3 Ocean Environment Simulator wiring diagram 26
3-4 Wiring diagram of Mobile Sensors Simulator, Fixed Platform Sensors Simulator, Sonar Array Simulator and Sonar Signal Processing Center 27
3-5 Wiring diagram of modules in control center 28
3-6 Simplified flow chart of Surveillance Module 29

4-1 Example 1. (a) shows the final estimated water sound speed profile and sampling path. (b) shows the error field and sampling path. . . . 36
4-2 Example 2. (a) shows the final estimated water sound speed profile and sampling path. (b) shows the error field and sampling path. . . . 37
4-3 Example 3. (a) shows the final estimated water sound speed profile and sampling path. (b) shows the error field and sampling path. (c) shows realizations of transmission loss. 38
4-4 Example 4. (a) shows the final estimated water sound speed profile and sampling path. (b) shows the error field and sampling path. (c) shows realizations of transmission loss. 39

4-5 Example 5. (a) shows the final estimated water sound speed profile and sampling path. (b) shows the error field and sampling path. (c) shows localizations of MFP. 40

A-1 Class diagram of class AUV_ASD 43
A-2 Class diagram of class AUV_SSD 46
A-3 Class diagram of class AUV_SSD_ASD 49
A-4 Class diagram of class Bathymetry 52
A-5 Flow chart of candidate_points_generate 56
A-6 Class diagram of class ControlAgent 57
A-7 Flow chart of DP_rollout 60
A-8 Class diagram of class DetectionRange 61
A-9 Flow chart of Run_TL_receiver 63
A-10 Flow chart of Run_TL_source 65
A-11 Flow chart of Run_DR 66
A-12 Class diagram of class FixedWaterSensor 67
A-13 Class diagram of class Fortran_matrix 70
A-14 Flow chart of greedy_algorithm 81
A-15 Class diagram of class MatchedFieldProcessing 82
A-16 Flow chart of Searching 84
A-17 Flow chart of Run 85
A-18 Class diagram of class ObjectiveAnalysis 86
A-19 Class diagram of class ObservationDatabase 89
A-20 Class diagram of class OceanPredictor 92
A-21 Class diagram of class PerfectSeabedDetector 95
A-22 Class diagram of class Random 101
A-23 Flow chart of rollout_once 104
A-24 Flow chart of the 1st overload of update 108
A-25 Flow chart of Transformer 109
A-26 Class diagram of class SimulatedOcean 110

A-27 Class diagram of class SonarArray 116
A-28 Class diagram of class Sonar_Performance 119
A-29 Class diagram of class Sonar_SPC 123
A-30 Class diagram of class SoundField 127
A-31 Flow chart of the 1st overload of Run 130
A-32 Flow chart of the 4th overload of Run 131
A-33 Flow chart of Output_Whole_Sound_Field 132
A-34 Class diagram of class SoundSpeedGenerator 133
A-35 Class diagram of class StandardEnvironmentInfo 135
A-36 Class diagram of class StandardRamInfo 138
A-37 Partial flow chart of Surveillance — Calculate mismatch displacement 143
A-38 Partial flow chart of Surveillance — Calculate TL_source 144
A-39 Class diagram of class SyntheticSeabed 145
A-40 Class diagram of class SyntheticStochasticWater 149
A-41 Flow chart of the 2nd overload of Output_Water_SoundSpeed 152
A-42 Flow chart of the 2nd overload of Output_All_Water_SoundSpeed . . 153
A-43 Class diagram of class SyntheticWater 154
A-44 Flow chart of total_cost_TL_receiver 158
A-45 Flow chart of points_filter 160
A-46 Class diagram of class Vector 161

Chapter 1

Introduction

In coastal regions, wind driven flow, tidal currents, river outflow, internal waves, solitary waves, fronts, eddies, thermal changes etc are usually dominant oceanographic processes. As a result, coastal ocean environment is often highly variable in time and space. In water column, the temperature profile, salinity profile, plankton distribution profile etc can vary in a complex dynamics driven by all oceanographic processes and their coupling; water depth is usually periodically changed by tides. So, in accordance, sound speed profile in the water also shows a complex variability. On the other hand, in the seabed, current flow interacts with bottom topography, thus bathymetric profile varies in time and space too, which in turn makes the dynamics of the water column extremely complex [1, 2, 3, 4, 5, 6].

Variability in the coastal ocean environment spans on multiple scales [7]. Current ocean prediction systems and conventional oceanographic measurement can not provide us with the ability to synoptically observe and accurately predict those dynamically interlocking, patchy and intermittent processes in coastal ocean, especially for those variabilities on small spatial scales and short temporal scales [8].

From an acoustic viewpoint, due to ocean variabilities especially small scale variabilities of the order of the acoustic wavelength of sonar systems, coastal ocean acoustic environment is largely unknown and too many uncertainties in terms of imperfect

sound speed profile, geo-acoustic models, reverberation levels, depth of thermocline, internal solitons etc, critically impacting the performance of acoustic systems [1].

For a non model-based sonar system, sonar performance is dependent on sound propagation properties in the ocean waveguide, the ocean acoustic environment. Therefore sonar performance can be viewed as a function of ocean acoustic environment,

$$SP = f(O)$$

where SP represents a sonar performance metric and O represents the ocean acoustic environment.

For a model-based sonar system such as Matched-Field Processing (MFP), sonar performance is dependent on both the real ocean acoustic environment and the environment model,

$$SP = f(O, O')$$

where O' is the environment model.

In the coastal ocean, due to the uncertain acoustic environment we can not even have high confidence in transmission loss (TL) estimate and consequently, non model-based sonar performance prediction has a significant uncertainty, i.e. SP must be treated as a random variable. In fact, a large part of sonar performance prediction uncertainty is indeed associated with ocean acoustic environmental uncertainties [1]. For model-based sonar, ocean acoustic environmental uncertainties makes environment model with very low confidence interval in predicting the real situation, thus sonar performance is often very unsatisfying [5]. (see Figure 1-1)

In such an uncertain or stochastic ocean environment, conventional oceanographic measurement systems can not capture the environmental uncertainties on short temporal scales and on spatial scales of the order of acoustic wavelengths, which are

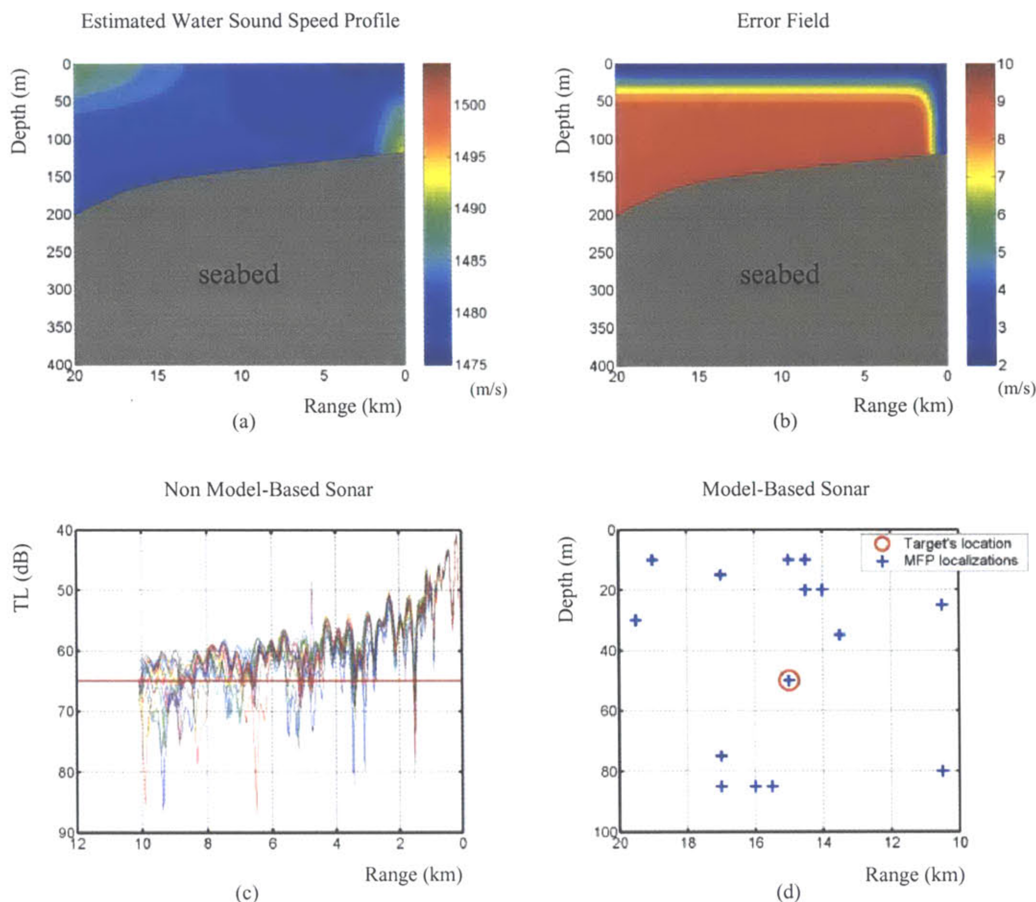


Figure 1-1: Illustration of how environmental uncertainties influences sonar performance. (a) shows an estimated water sound speed profile in shelf break region. (b) is the corresponding error field. We assume water sound speed profile is a Gaussian stochastic process. (c) shows non model-based sonar performance prediction uncertainty, where transmission loss and detection range are selected as sonar performance metric. In this case, the receiver is at 10m far from origin and 70m depth; a series of 50Hz source are at 50m depth and far to 10km; the detectable threshold is 65 dB. In this figure, strong TL uncertainty and detection range uncertainty can be observed. (d) shows model-based sonar performance, where a MFP sonar is used to localize a target at 15km range and 50m depth. In this case, we assume there is no ambient noise. In this figure, we can see that MFP localizations are usually several km away form the true target's location.

the most important scales to sonar performance, therefore local, high resolution in situ measurements are needed. Rapid deployable in situ measurement system has long been recognized as a very important tactical requirement to improve non model-based sonar performance prediction or improve model-based sonar performance, capturing

environmental uncertainties on scales from 10 to 1000 meters [1].(see Figure 1-2)

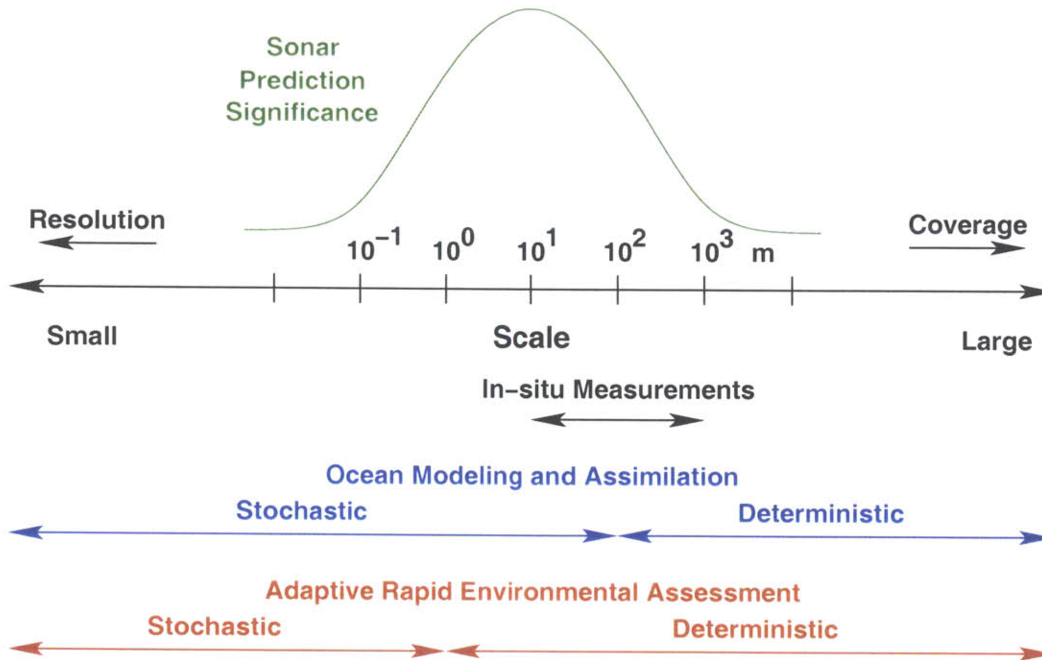


Figure 1-2: Multi-scale environmental assessment. The typical sonar systems performance is dependent on acoustic environment variability over a wide range of scales. Optimal environmental assessment will therefore be a compromise between conflicting requirements of coverage and resolution. By targeting areas of high sensitivity to the sonar system through in situ measurements, the deterministic assessment range will be shifted towards smaller scales.

However, the ocean area of interest is usually fairly large, whereas in situ measurement resources - mobile sensors etc - are very limited due to cost, time and performance constraints. Thus sampling strategies can make a significant difference in predicting sonar performance or improving sonar performance, but how to design them for real-time operation is a major problem. Data are collected over time and in such a situation adaptive sampling methods often lead to more efficient results than conventional fixed sampling techniques [9], in which sampling or data allocation is fixed and predetermined. On the contrary, an adaptive sampling strategy makes sampling decisions based on accruing data and all acquisition information. Sonar performance prediction uncertainty will be minimized with improved sonar performance

as a result.

Chapter 2

AREA Concept

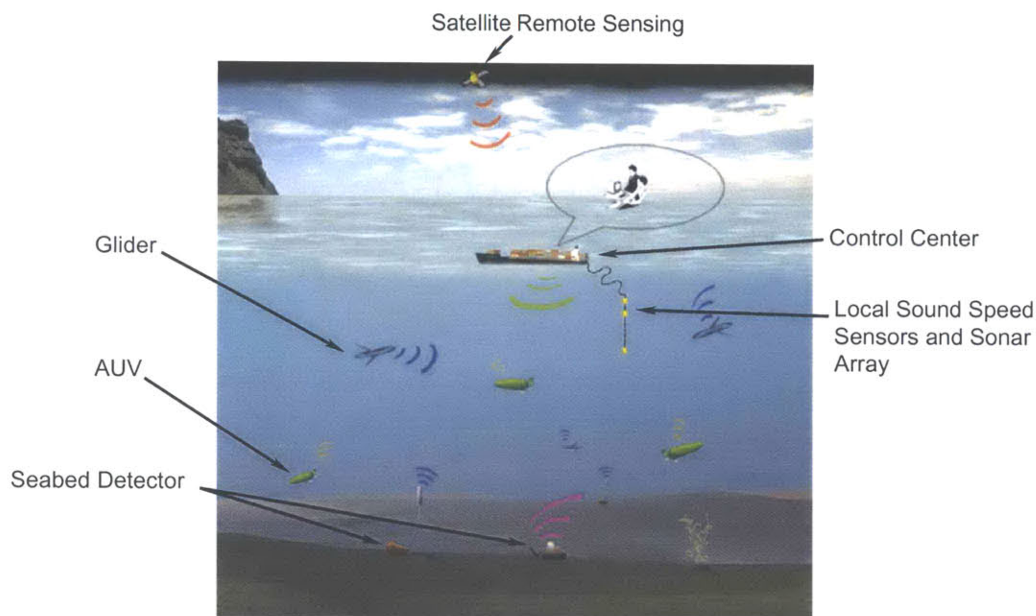


Figure 2-1: Illustration of Adaptive Rapid Environmental Assessment System

The *Adaptive Rapid Environmental Assessment* (AREA) concept has been proposed to reach that objective, trying to minimize the sonar performance prediction uncertainty or improve sonar performance by adaptively identifying an optimal deployment strategy of in situ measurement resources, capturing the uncertainty of the most critical and uncertain environmental parameters, within the existing operational constraints [1](see Figure 2-1). Moreover, the AREA system can

be used for minimizing oceanographic information uncertainties, biological information uncertainties and sound propagation uncertainties. In that regard, AREA is a multi-purpose adaptive sampling system.

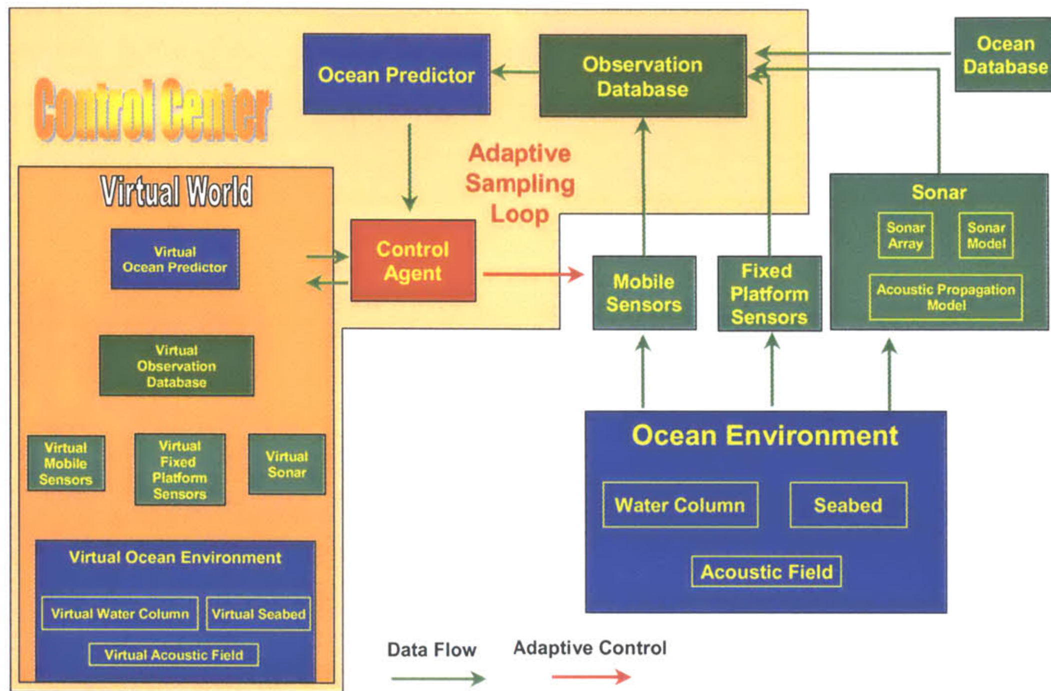


Figure 2-2: Adaptive Rapid Environmental Assessment System wiring diagram

As shown in Figure 2-2, an operational AREA system has 5 main components:

1. The real ocean environment: this is the object of observation and measurement, in which sensors can receive all kinds of information including temperature, salinity, density, attenuation coefficient, acoustic signals and seabed geophysical & geographic information, etc. Basically, the ocean environment can be divided into 3 sub-components: water column, seabed and acoustic field.

2. Sonar: the sonar component includes hydrophone arrays and signal processing system. It can receive acoustic signals and process signals, then detect or localize target. For different sonar, we may have different sonar performance metrics.

3. Mobile sensors: a mobile sensor consist of two parts: one or more sensors that can measure some sort of oceanographic or bathymetric information; a platform that can carry sensor and move in water or on seabed according to commands. Most likely, due to its excellent mobility and fast technical progress, Autonomous Underwater Vehicle (AUV) will be the platform for mobile sensor. An AUV can receive and execute commands, and also it can transmit new measurement results and its own status' information.

4. Fixed platform sensors: basically these sensors have the same functions as mobile sensors but with fixed platforms. They could be any oceanographic or bathymetric measurement devices such as local XBT, local CDT, satellite or acoustic remote sensing system and seabed detector etc. Fixed platform sensors will give AREA initial information about the ocean area of interest.

5. Control center: the control center is the heart and brain of AREA. People or computers can directly operate and control the whole AREA system through the control center. Basically, it consists of 3 modules: Observation database module, ocean predictor module and control agent module. Observation database module is an interface and data storage, which can communicate with all sensors and sonar, then store received data, and activate ocean predictor module and then transfer data. Due to very fast progress in underwater communication techniques, we can assume that the observation database module can communicate well with any other part in real-time. The ocean predictor module can process received data and predict the ocean. It provides us with an estimation of the stochastic ocean environment and corresponding error field. An ocean database may be needed to predict more accurately. Control agent module works as a decision maker, which can generate optimal commands based on the received data and analysis results. This module is very complicated. Depending on the decision making algorithm, the control agent may be structured differently. For most sophisticated algorithms, it usually possesses a vir-

tual world - a mirror of the whole AREA system - and 'play' all possible controls in the virtual world, then select the one with optimal virtual consequence as command. This module is the main object of attention in the AREA project.

Besides the above 5 main objects, an ocean database may be needed, which contains fundamental principles of ocean circulation in ocean area of interest. As mentioned before, it can significantly improve ocean prediction and consequently AREA system may work significantly better. So it could be very important.

In reality, the AREA system starts with initialization - updating observation database according to the latest ocean database, latest measurements by the fixed platform sensors, mobile sensors' initial information and sonar configuration information etc. After initialization, the control center will run the ocean predictor module and generate preliminary environmental predictions. All initial information and analysis results will then be collected by the control agent module where a sampling strategy programme will run and work out commands such as next sampling locations for the mobile sensors . Those commands will be sent to the mobile platforms through communication channels. Following the commands, the mobile sensors will approach the next sampling locations and complete the measurements. The new in situ measurements will be sent back to the control center and the observation database will be updated again. This is the *Adaptive Sampling Loop* (see Figure 2-1 and Figure 2-3). As AUV techniques develop, it is very possible that control center will be decentralized, i.e. it will be distributed in mobile sensors and mobile sensors will finally become as intelligent sensors.

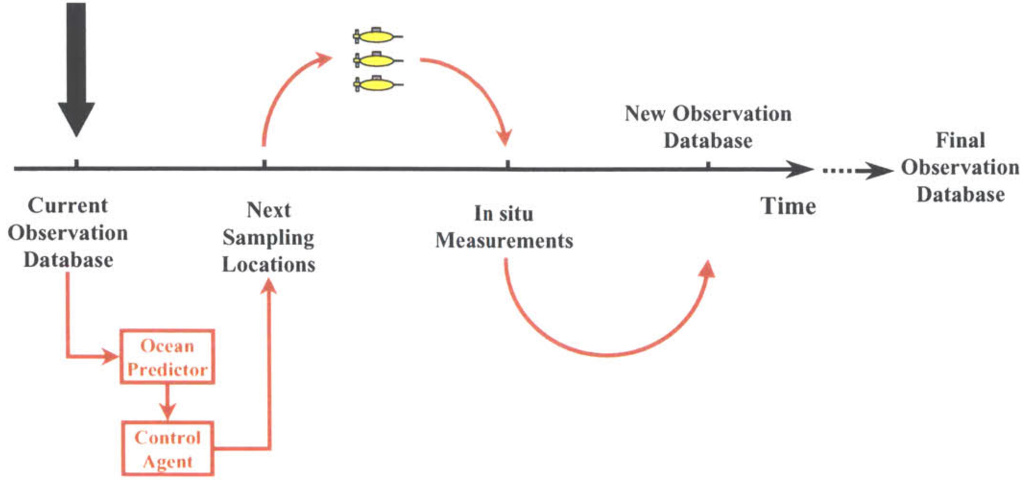


Figure 2-3: Sequential diagram of *Adaptive Sampling Loop*. At each stage, the Observation Database will be first updated and the Ocean Predictor will do analysis; then the Control Agent will determine next sampling locations; following those commands, mobile sensors will do in situ measurements and new measurement results will be obtained, by which the Observation Database will be updated again. Repeating the *Adaptive Sampling Loop*, sampling points locations will be determined sequentially based on all the newest observation.

Adaptive sampling problem can be expressed as a typical ***Dynamic Programming*** **(DP)** or ***Reinforcement Learning*** **(RL)** problem with finite horizon [10] (see Figure 2-4), where in our case, the content of the Observation Database can be selected as state x; commands from the Control Agent can be selected as control u; in situ measurements results can be selected as random disturbance ω; electric power used in one *Adaptive Sampling Loop* or some other factors can be selected as cost per stage g, which is a function of state, control and disturbance; the final sonar performance or sonar performance prediction uncertainty can be selected as terminate stage cost g_N. As shown in Figure 2-4, at stage k we have current state x_k, base on which a control can be generated from control function μ_k; after operating the control, a random disturbance ω_k will happen, whose probability density function (pdf) can be written as $P_k(\omega_k\,|x_k, u_k)$; then following the state equation:

$$x_{k+1} = f_k\left(x_k, u_k, \omega_k\right)_{k=0,1,2,\cdots,N-1}$$

state x_{k+1} at stage $k+1$ will be reached with a cost. At the end, terminate state x_N

will be reached and the terminate stage cost $g_N(x_N)$ will happen then. The goal in the DP problem is to find a control function sequence

$$\{\mu_0(x_0), \mu_1(x_1), \cdots, \mu_{N-1}(x_{N-1})\} \;\; s.t. \; min \; E\left\{g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k, \omega_k)\right\}$$

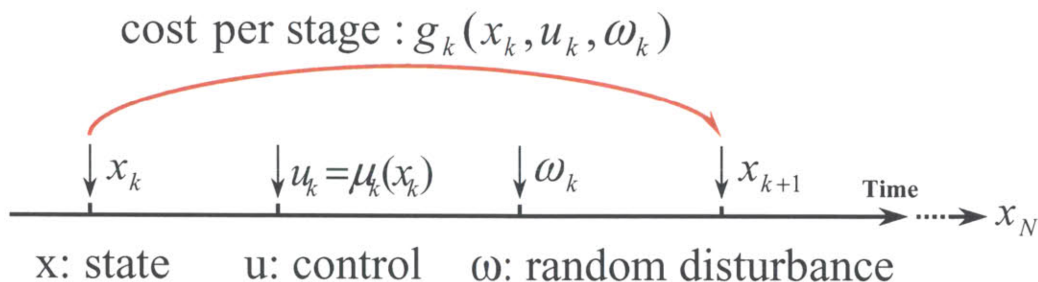


Figure 2-4: Sequential diagram of *Dynamic Programming.*

For DP or RL problem, there are a lot of simple or complex methods to find the optimal or sub-optimal control function sequence. In our case, the state space and control space could be very huge; the state equation, the disturbance's pdf and the terminate cost are so complicated that no explicit formula exists; So, to solve the DP problem, find the optimal or sub-optimal sampling strategies and test their optimization effects before doing very costly on-site experiments, an *Adaptive Rapid Environmental Assessment Simulation Framework* is needed, by which we can also observe how AREA system will work and test if real-time adaptive sampling is feasible. The Adaptive Rapid Environmental Assessment Simulation Framework provides a training and learning tool for control agent.

Chapter 3

AREA Simulation Framework

An AREA simulating system has been created using C++ - an object-oriented language. Due to the object-oriented feature, every real object can have a corresponding simulated object in the computer, which can simulate all functions that the real one has. So, basically each component in the real AREA system has a corresponding module in the AREA simulation framework. However, the sonar system is divided into 2 modules: sonar array simulator and sonar signal processing center. The Control center is directly replaced with the observation database module, ocean predictor module and control agent module. For the control agent, several different sampling strategy algorithms have been embedded. In the end, a surveillance module and an output module were built to monitor the whole system and output results. In this way, the AREA simulation framework is upgradeable and flexible; and its structure is simpler and close to a real AREA system.

As shown in Figure 3-1, the structure of AREA simulation framework is like an integrated circuit board. AREA.cpp is the C++ main file containing the 'main' function. It works like a human-computer interface where we can input almost all parameters for each module, select options and start running programme (see Figure 2-2). AREA.cpp provides a working environment to all the other modules like the main board in PC to peripherals.

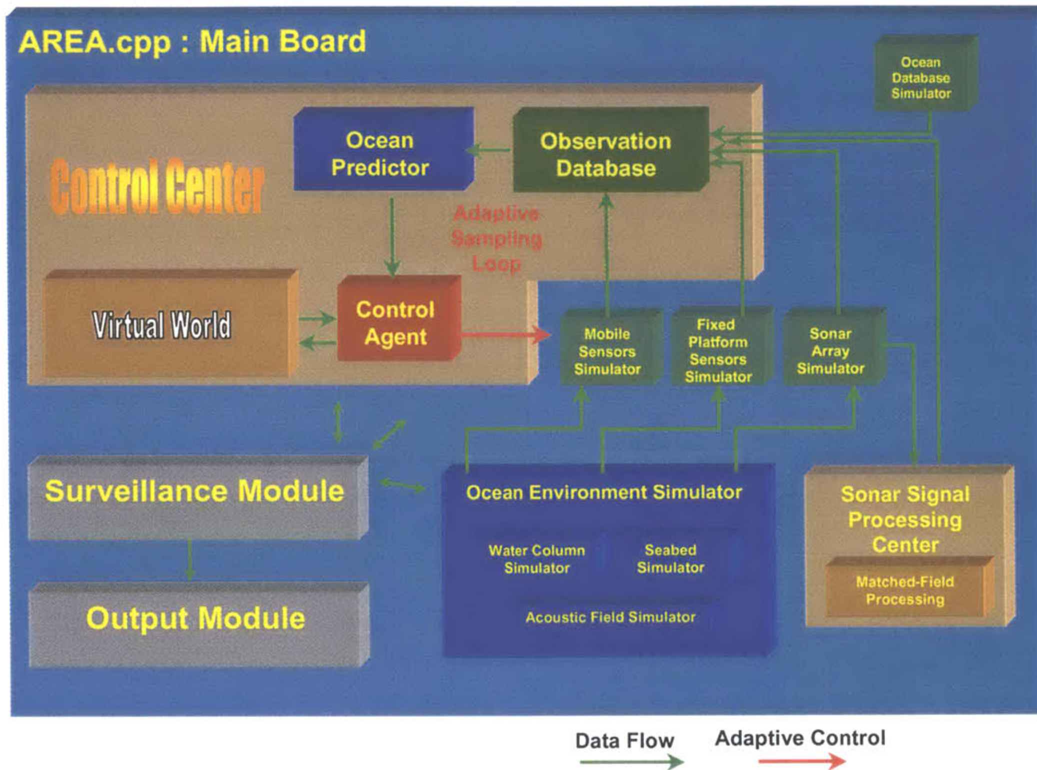


Figure 3-1: AREA simulation framework wiring diagram

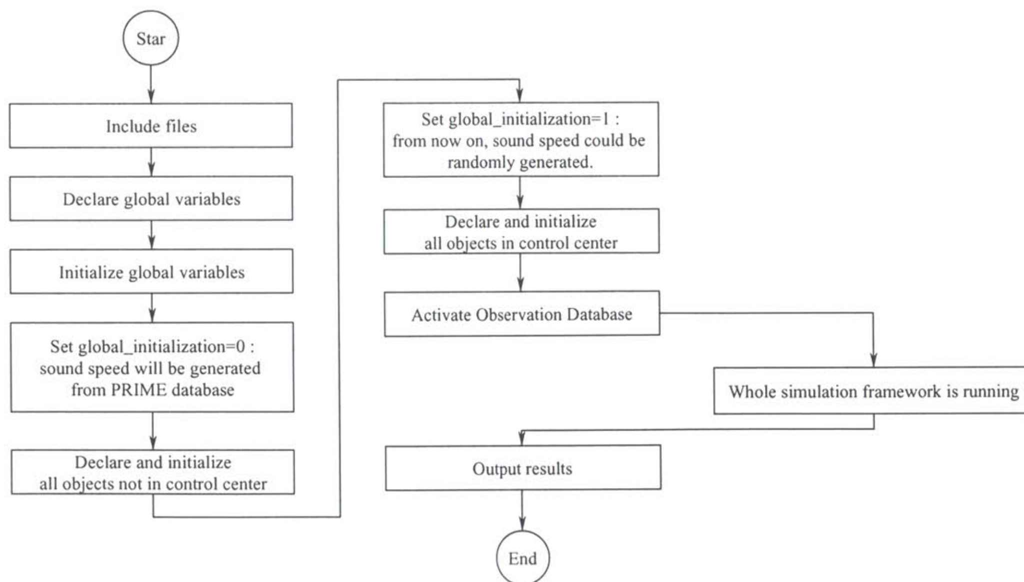


Figure 3-2: Flow chart of AREA.cpp

The Ocean Environment Simulator module is supposed to provide sensors and sonar arrays with oceanographic information, bathymetric information and acoustic signals. It includes 3 sub-modules: Water Column Simulator, Seabed Simulator, Acoustic Field Simulator (see Figure 3-2). Water Column Simulator and Seabed Simulator simulate the ocean environment in water column and seabed respectively. The Acoustic Field Simulator can generate the acoustic field according to water and seabed environment and sound source parameters input from AREA.cpp. The current acoustic model is RAM. At present since only PRIMER Shelf Break experiment database is available, the Ocean Environment Simulator can only create a simulated ocean for that scenario. There are two ways to do this (see Figure 2-3):
1. Through the database, an environment realization can be re-created. In this way, the Ocean Environment Simulator can only represent a certain ocean environment.
2. Through a replica of Ocean Predictor and some initial information from the PRIMER Shelf Break experiment database, a mean and a standard deviation of the ocean environment can be obtained. Thus by assuming the ocean environment is a Gaussian stochastic process, the Ocean Environment Simulator can provide an uncertain ocean environment. Details about the two ocean environment simulation algorithms will be given in Appendix A.20, A.31, A.30.

The Mobile Sensors Simulator module can be called and input controlling parameters by the Control Agent module. The Mobile Sensors Simulator can simulate how real mobile sensors move in the ocean and measure in situ by calling the Ocean Environment Simulator to output information at those measurement locations. By configuring the sensors and platforms differently, this module can simulate many sort of mobile sensors such as XBT carried on ship, CTD carried on AUV, hydrophones carried on AUV, or both of them carried on AUV.

The Fixed Platform Sensors Simulator module can retrieve oceanographic information and/or bathymetric information from the Ocean Environment Simulator as conventional oceanographic sensors do in ocean. The Fixed Platform Sensors Simula-

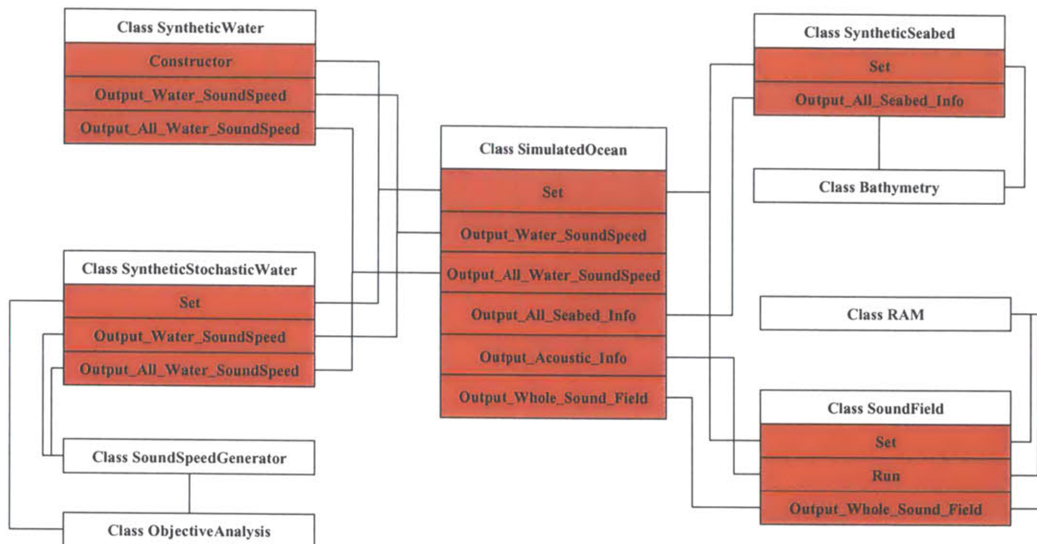


Figure 3-3: Ocean Environment Simulator wiring diagram

tor may include several different objects, each of them corresponding to one particular sensor, which could be local CDT, satellite or acoustic remote sensing and a seabed mapping device. Because of the flexibility, this module can be quickly adapted according to requirement.

The Sonar Array Simulator module simulates a hydrophone array, which can call the Ocean Environment Simulator and retrieve data from the Acoustic Field Simulator. Acoustic signals received by the Sonar Array Simulator and signals received by the Mobile Sensors Simulator will be processed in the Sonar Signal Processing Center. The Sonar Signal Processing Center is a software package containing different sonar models and acoustic models; however, currently only *Matched-Field Processing* (MFP) method and RAM are included.

The Observation Database is the first module in the control center. Its function is to sequentially call and receive data output from the Ocean Database Simulator, Sonar Signal Processing Center, Sonar Array Simulator, Fixed Platform Sensors Simulator, Mobile Sensors Simulator and store the data. In fact, the whole simulation framework starts from the Observation Database calling and collecting initial infor-

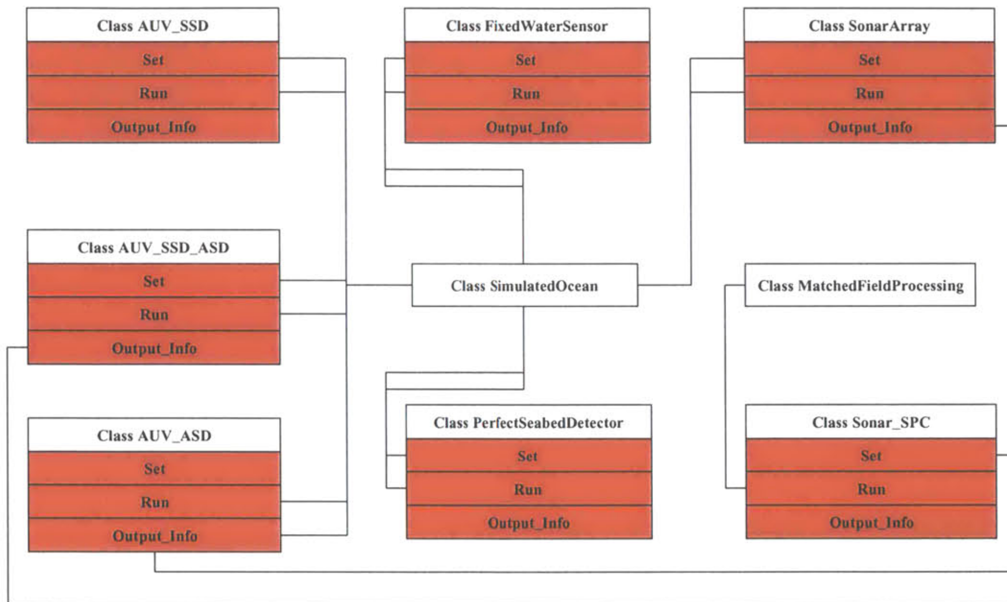


Figure 3-4: Wiring diagram of Mobile Sensors Simulator, Fixed Platform Sensors Simulator, Sonar Array Simulator and Sonar Signal Processing Center

mation from those modules.

After the Observation Database finishes collecting all necessary initial information, it will call and activate module Ocean Predictor. This module uses some estimation algorithms such as an objective analysis technique to predict the ocean acoustic environment and simultaneously provide the error field.

At the end, the Control Agent will be called and passed those initial information and analysis results. Based on all information and according to adaptive sampling algorithm, the Control Agent may create a virtual world for trial purpose and determine optimal or sub-optimal commands through a complicated procedure. Details about the decision making procedure are out of the range of this thesis, but a major AREA research issue.

Once commands are determined, Mobile Sensors Simulator will be called and ex-

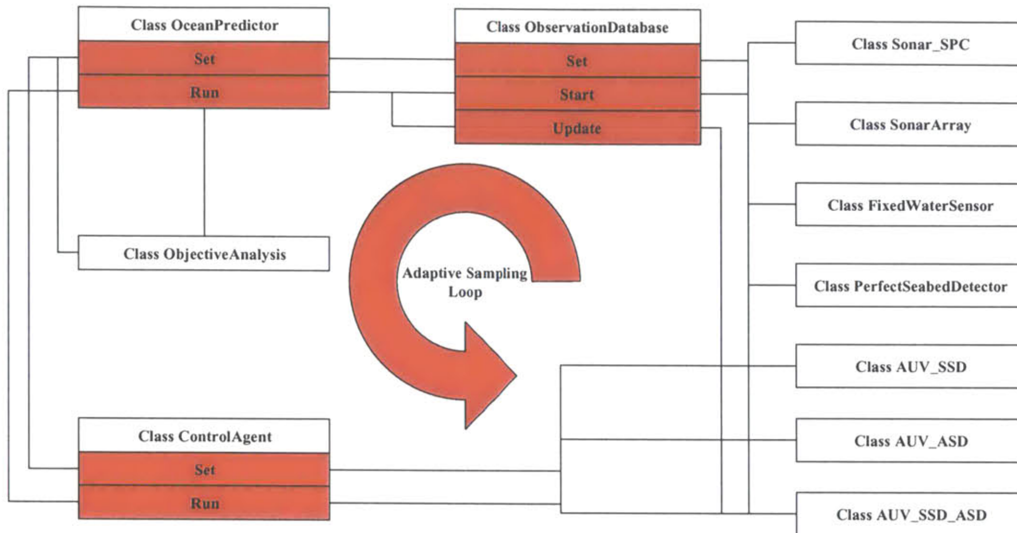


Figure 3-5: Wiring diagram of modules in control center

ecute those commands to obtain the newest data. After that, Observation Database will be called and updated. Then, the adaptive sampling loop will be repeated again until the Mobile Sensors Simulator finishes all in situ measurements.

When all the above modules are running, a very special module - the Surveillance Module keeps watching all processes and records all interesting intermediate results. In the end, the Surveillance Module will send all records to Output Module through which results will be output into a file.

Note:

1. Since we don't have any ocean database for Ocean Predictor, there's currently no Ocean Database Simulator in the simulation framework. But it is easy to add in this module later.

2. In this chapter, we simply introduced the structure and functions of the simulation framework. For more details, please refer to Appendix and original files.

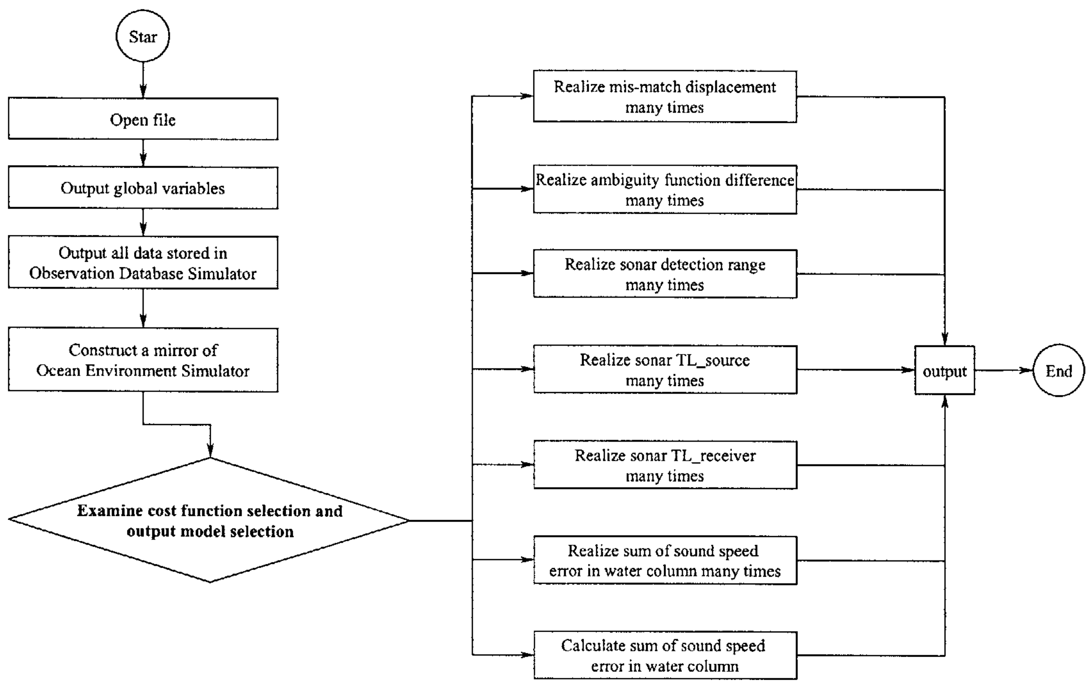


Figure 3-6: Simplified flow chart of Surveillance Module

Chapter 4

How To Install And Use AREA Simulation Framework

4.1 Set Up

Running Environment: Red Hat Linux 7.2 (kernel 2.4.7-10)
Installation Method: just copy folder AREA to the destination directory

After installation, we need to first set up all system parameters and options. Basically, this can be done in file `.../AREA/src/AREA_DP_RL_AS_v7/src/AREA.cpp`. This file is the main file of the whole simulation system. Global variables are first declared and assigned, and then global options are selected (see Figure 2-2). In AREA simulation framework, metric unit is adopted unless explicitly specified.

To initialize simulation framework, we need to first set up the following global variables and options step by step:

1. `global_start and global_end`

   Vector global_start contains latitude and longitude of start point; and global_end contains those of end point. They determine the ocean area of interest. Currently due to the limit of our HOPS OAG data file, latitude of the start point

should be set around 40.25 degree and latitude of the end point should be set around 40.0 degree; longitude of both the start and end point should be set around -71.0 degree. In the current simulation framework we only consider a 2-D problem in the vertical plane defined by the start and end points. In local coordinates, the origin is the surface point at the start point.

2. `global_rmax and global_zmax`

   These two scalars define the maximum horizontal and vertical computation range in RAM code.

3. `global_water_c_n, global_water_c_m, global_water_grid_z, global_water_grid_r`

   Vector global_water_grid_r and global_water_grid_z are the horizontal and vertical axis in ocean water column discretization grid respectively. global_water_c_m and global_water_c_n are their lengths.

4. `global_seabed_grid_r, global_seabed_speed_z, global_seabed_density_z, global_seabed_attn_z`

   In seabed, for sound speed, density and attenuation coefficient, we have different discretization grids. global_seabed_grid_r works as the common horizontal axis. global_seabed_speed_z is the vertical axis for sound speed's grid. global_seabed_density_z is the vertical axis for density's grid. global_seabed_attn_z is the vertical axis for attenuation coefficient's grid.

5. `global_bathymetry_resolution`

   This item defines the water-seabed interface line resolution.

6. `global_frequency, global_source_r, global_source_z`

   global_frequency is the sound source frequency (assume that there is a CW source in the ocean). global_source_r, global_source_z define source location with respect to the origin (assume that the source is in the plane defined by the start and end point). This sound source could be the target to localize.

7. `global_fleet_config`

   This vector has 7 elements. The 1st element stores AUV_SSD's quantity.

The 2nd element stores AUV_SSD_ASD's quantity. The 3rd element stores AUV_ASD's quantity. The 4th element stores Perfect_Seabed_Detector's quantity. The 5th element stores Fixed_Water_Sensor's quantity. The 6th stores Sonar_Array's quantity. For the 7th element, if it's equal to 1 then Sonar_SPC will do initialization for MFP; if it's not, then Sonar_SPC will do nothing.

8. `global_auv_ssd_init_location_r, global_auv_ssd_init_location_z`
   These 2 vectors store initial locations of all AUV_SSD objects.

9. `global_auv_ssd_asd_init_location_r, global_auv_ssd_asd_init_location_z`
   These 2 vectors store initial locations of all AUV_SSD_ASD objects.

10. `global_auv_asd_init_location_r, global_auv_asd_init_location_z`
    These 2 vectors store initial locations of all AUV_ASD objects.

11. `global_n_fixed_water_detector, global_fixed_water_detector_location_r, global_fixed_water_detector_location_z`
    Vector global_n_fixed_water_detector contains sensor quantity information in each FixedWaterSensor object. Vector array global_fixed_water_detector_location_r and global_fixed_water_detector_location_z contain locations of all sensors in each FixedWaterSensor object.

12. `global_n_receivers, global_sonar_array_r, global_sonar_array_z`
    Vector global_n_receivers contains hydrophone quantity information in each SonarArray object. Vector array global_sonar_array_r and global_sonar_array_z contain locations of all hydrophones in each SonarArray object.

13. `global_replica_r, global_replica_z`
    Vector global_replica_r and global_replica_z are the horizontal and vertical axis of replica sources grid used in MFP.

14. `global_dB_threshold`
    This scalar defines detection threshold in dB used in Sonar SPC for detection range estimation.

15. `global_virtual_receivers_r, global_virtual_receivers_z`

    These 2 vectors define the line along which TL_receiver will be calculated (see item 21).

16. `global_target_r, global_target_z`

    These 2 vectors define the line along which TL_source will be calculated (see item 21).

17. `global_hydrophone_r, global_hydrophone_z`

    Define a hydrophone's location. This hydrophone is used in calculating TL_source.

18. `global_Lr, global_Lz`

    These are sound speed correlation lengths in r and z direction in water column.

19. `global_sig_c, global_sig_n`

    These are a priori sound speed standard deviation (m/s) and noise sound speed standard deviation (m/s) respectively.

20. `global_acoustic_model_selection`

    In the simulation framework, there could be several acoustic models available. global_acoustic_model_selection indicates which model is selected. Now, we only have RAM, so global_acoustic_model_selection must be equal to 1.

21. `global_cost_function_selection`

    This option indicates which cost function will be selected to minimize.

    `global_cost_function_selection=1`: summation of sound speed standard deviations in water column provided by objective analysis is selected as cost function.

    `global_cost_function_selection=2`: basically, this selection is very similar to the selection 1. But in this selection, we will generate many realizations of sound speed in water column by Monte Carlo simulation and calculate sample variance for each water column point. Summation of those sample variances is selected as cost function.

global_cost_function_selection=3: this is reserved to biological information prediction uncertainty.

global_cost_function_selection=4: we have a virtual CW sound source and we calculate TLs along a level line, which is called TL_receiver. Summation of TL_receiver sample variance at all points is selected as cost function.

global_cost_function_selection=5: we calculate TLs at a hydrophone with a series of CW sound source located on a level line. This TL curve is called TL_source and summation of its sample variance at all points is selected as cost function.

global_cost_function_selection=7: based on the TL_source curve and global_dB_threshold, we can calculate sonar detection range, which is selected as cost function.

global_cost_function_selection=9: we have a MFP sonar and a virtual CW sound source. Ocean Predictor can provide us with the newest estimated water column sound speed profile which can be used as environment model in Sonar Signal Processing Center and the corresponding error field. Then we can realize the ocean environment including sound field many times by Monte Carlo simulation and Sonar Signal Processing Center will give us many realizations of ambiguity function. In addition, we can also use those true ocean environment realizations as our environment model in MFP and get many so-called real ambiguity functions. Summation of difference between so-called real ambiguity function and its corresponding ambiguity function output from Sonar Signal Processing Center is selected as cost function.

global_cost_function_selection=11: This selection is very similar to selection 9. However, now we use summation or average of mis-match displacements in ambiguity functions output from Sonar Signal Processing Center as cost function.

22. global_output_model_selection

This option indicates how we would like to generate results and output. if

global_output_model_selection=1, then we will output results without running Monte Carlo simulation; if global_output_model_selection=3, then we will output results and only run Monte Carlo simulations at the last stage.

23. `global_ControlAgent_model_selection`
    This option indicates which decision maker will be selected. if it's 0, then no any sampling strategy will be selected and no any in situ measurement will be done; if it's 1 or 2, then two different predetermined linear sampling strategies will be selected respectively; if it's 3, then an adaptive sampling strategy driven by greedy algorithm will be selected; if it's 4, then another adaptive sampling strategy driven by rollout algorithm based on greedy algorithm will be selected.

24. `global_operation_model`
    This option indicates if the simulated ocean uses certain ocean environment model or stochastic ocean environment model.

25. **`global_monitor_nrel, global_rollout_nrel, global_total_cost_nrel`**
    global_monitor_nrel is the times of Monte Carlo simulations realized in output. global_rollout_nrel is the times of Monte Carlo simulation in rollout algorithm. global_total_cost_nrel is the times of Monte Carlo simulation in computing total cost.

Refer to the file AREA.cpp for more details about how to set up global variables and options.

After initialization, we can compile and link, and then run the simulation framework:
In directory `.../AREA/src/AREA_DP_RL_AS_v7/src` type

```
>> make<Enter>
>> AREA<Enter>
```

4.2 Examples

1. global_acoustic_model_selection=1

   global_cost_function_selection=1

   global_output_model_selection=3

   global_ControlAgent_model_selection=2

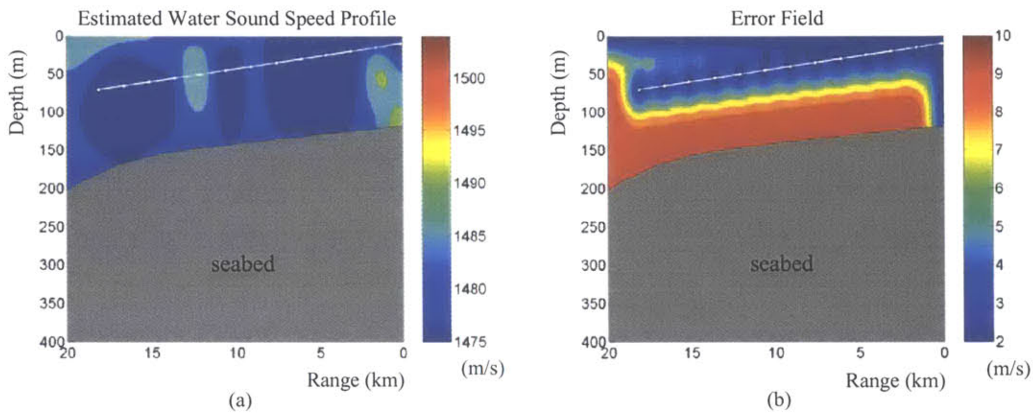


Figure 4-1: Example 1. (a) shows the final estimated water sound speed profile and sampling path. (b) shows the error field and sampling path.

   In this example, a predetermined linear sampling strategy is selected and summation of sound speed standard deviations in water column is selected as cost function. Figure 4-1 shows a realization in this scenario, in which $cost = 4746.3(m/s)$.

2. global_acoustic_model_selection=1

   global_cost_function_selection=1

   global_output_model_selection=3

   global_ControlAgent_model_selection=4

   In this example, adaptive sampling strategie driven by rollout algorithm base on greedy algorithm is selected and summation of sound speed standard deviations

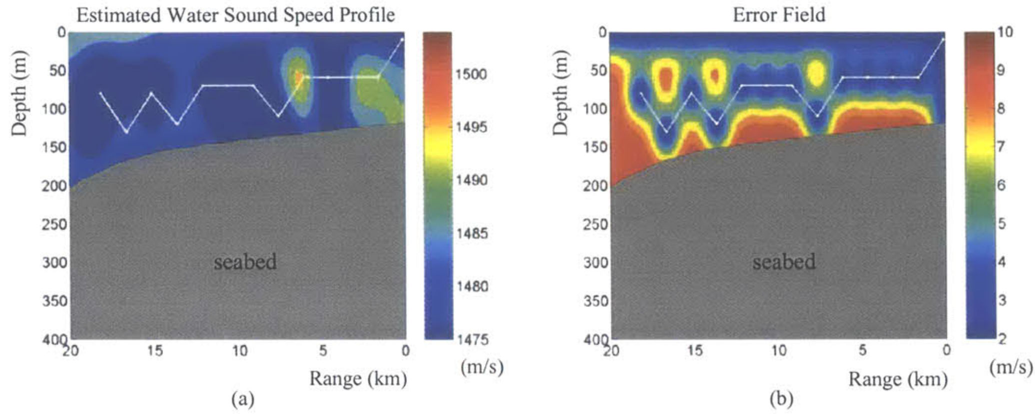


Figure 4-2: Example 2. (a) shows the final estimated water sound speed profile and sampling path. (b) shows the error field and sampling path.

in water column is selected as cost function. Figure 4-2 shows a realization in this scenario, in which $cost = 4307.77(m/s)$.

3. global_acoustic_model_selection=1
   global_cost_function_selection=4
   global_output_model_selection=3
   global_ControlAgent_model_selection=4

   In this example, we have a virtual $50Hz$ CW sound source at 50m depth and 15km range and we have a series of receivers distributed from 0km to 10km ranges and at 50m depth. Adaptive sampling strategy driven by rollout algorithm base on greedy algorithm is selected and summation of TL_receiver sample variance at all points is selected as cost function. Figure 4-3 shows a realization in this scenario, in which $cost = 8445.4(dB)$.

4. global_acoustic_model_selection=1
   global_cost_function_selection=5
   global_output_model_selection=3
   global_ControlAgent_model_selection=4

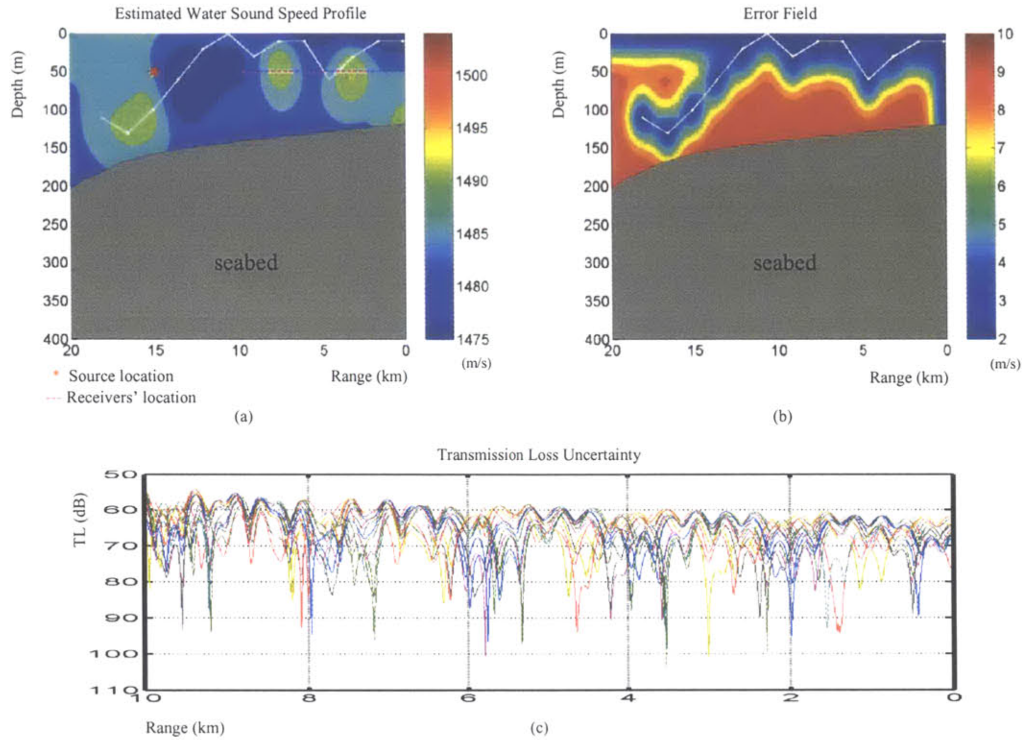


Figure 4-3: Example 3. (a) shows the final estimated water sound speed profile and sampling path. (b) shows the error field and sampling path. (c) shows realizations of transmission loss.

In this example, we have a hydrophone at 70m depth and 10m range; and we have a series of $50Hz$ CW sources distributed from 0km to 10km ranges and at 50m depth. Adaptive sampling strategy driven by rollout algorithm base on greedy algorithm is selected and summation of TL_source sample variance at all points is selected as cost function. Figure 4-4 shows a realization in this scenario, in which $cost = 265.79(dB)$.

5. global_acoustic_model_selection=1
   global_cost_function_selection=11
   global_output_model_selection=3
   global_ControlAgent_model_selection=3

In this example, we have a $50Hz$ CW sources at 15km range and 50m depth and

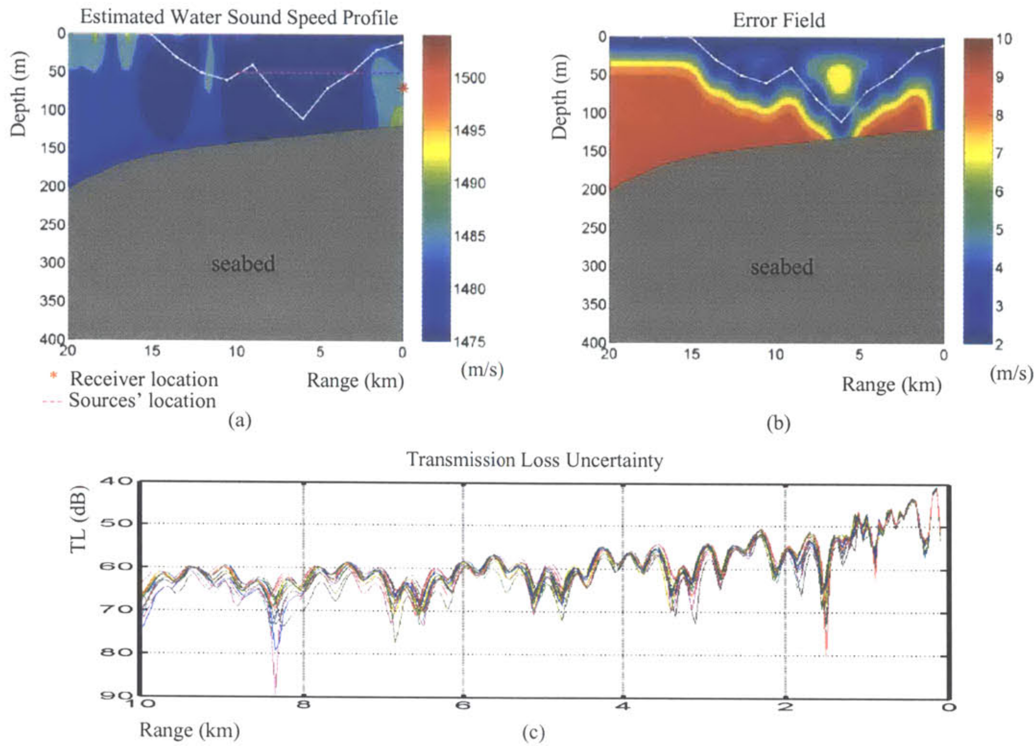


Figure 4-4: Example 4. (a) shows the final estimated water sound speed profile and sampling path. (b) shows the error field and sampling path. (c) shows realizations of transmission loss.

a 7-hydrophone sonar array at 70m range. Adaptive sampling strategy driven by greedy algorithm is selected and average of mis-match displacements in MFP is selected as cost function. Figure 4-5 shows a realization in this scenario, in which $cost = 1005.9(m)$.

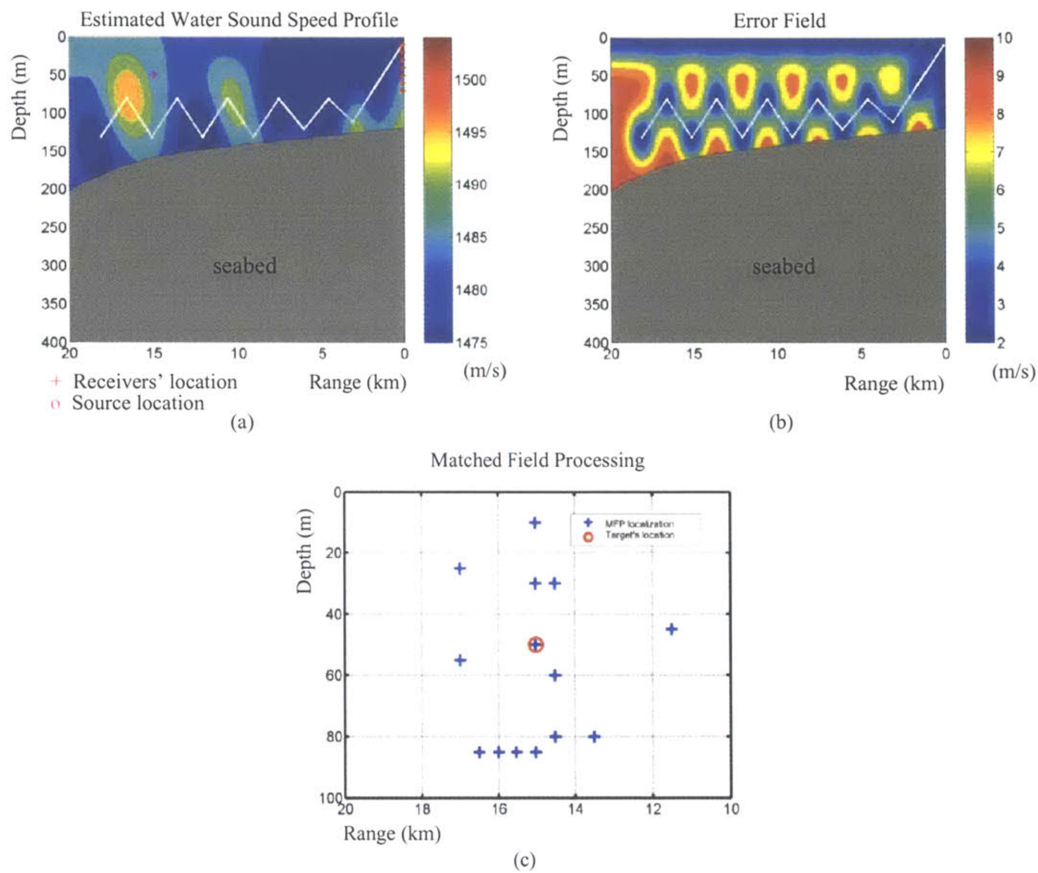


Figure 4-5: Example 5. (a) shows the final estimated water sound speed profile and sampling path. (b) shows the error field and sampling path. (c) shows localizations of MFP.

Chapter 5

Summary

The coastal environment is characterized by variability on small spatial scales and short temporal scales, which leads to the most significant ocean acoustic environmental uncertainties with respect to non model-based sonar performance prediction and model-based sonar performance. The AREA concept was proposed in [1] to capture those uncertainties by adaptive and rapid in situ measurement. A possible structure of the AREA system has been introduced and demonstrated in this thesis. Function of the 5 main components and working sequence were discussed. The central component of the AREA system is the *Adaptive Sampling Loop*, which has been described in detail. To find the optimal or sub-optimal sampling strategies and test their optimization effects before doing very costly on-site experiments, an AREA Simulation Framework has been constructed in C++. The structure of the simulation framework and its main modules were discussed, and interactions between those modules were illustrated. In the end, we showed how to set up and initialize the simulation framework and some examples were given. Moreover, specifications of all important C++ classes and files were documented in Appendix.

Appendix A

Classes and Files

Appendix A includes specifications of all classes and function files created by Ding Wang. For those inherited from Pierre Elisseeff, only the important classes and files are selected and described.

A.1 AVU_ASD.h

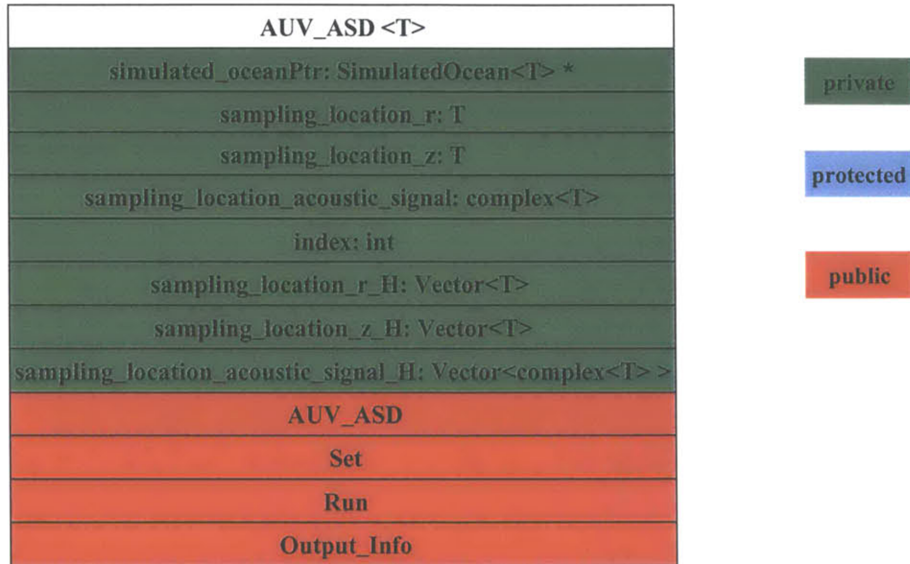


Figure A-1: Class diagram of class AUV_ASD

In this header file, class AUV_ASD is defined, which can simulate an hydrophone carried on AUV.

A.1.1 Data Members

1. SimulatedOcean<T> * simulated_oceanPtr — private; this is the pointer pointing to the simulated ocean.

2. T sampling_location_r — private; This is the horizontal coordinates of current sampling point.

3. T sampling_location_z — private; This is the vertical coordinates of current sampling point.

4. complex<T> sampling_location_acoustic_signal — private; this is the acoustical signal received at current sampling point.

5. int index — private; this is the index of current AUV_ASD object (we may have several AUV-carring ASD).

6. `Vector<T> sampling_location_r_H` — private; this is the history record of `sampling_location_r`.

7. `Vector<T> sampling_location_z_H` — private; this is the history record of `sampling_location_z`.

8. `Vector<complex<T> > sampling_location_acoustic_signal_H` — private; this is the history record of `sampling_location_acoustic_signal`.

A.1.2 Member Functions & Operators

1. Name: `AUV_ASD`

   Overloads:

   - `AUV_ASD(void)`

   Description:

   public; Constructor. Nothing is done in construction.

2. Name: `Set`

   Overloads:

   - `void Set(SimulatedOcean<T> * simulated_oceanPtr_, const T & init_location_r_, const T & init_location_z_, const int & index_)`
   - `void Set(SimulatedOcean<T> * simulated_oceanPtr_, const Vector<T> & sampling_location_r_H_, const Vector<T> & sampling_location_z_H_, const Vector<complex<T> > sampling_location_acoustic_signal_H_, const int & index_)`

   Description:

   public; The first overload can set up AUV_ASD initial status including initial location, its index number and connecting to the simulated ocean. All the other data members will be automatically generated. By the second overload, we can set up the above 5 data members manually. This can be used in virtual world.

3. Name: `Run`

   Overloads:

   - `void Run(const Vector<T> &start_, const Vector<T> &end_, const T & target_r_, const T & target_z_)`

- `void Run(const T &target_r_, const T &target_z_)`

Description:

public; This function simulates how AUV_ASDs process commands. In the first overload, start_ and end_ are dummy, however this is a flexible interface for upgrading.

4. Name: `Output_Info`

   Overloads:

   - `void Output_Info(T & sampling_location_r_, T & sampling_location_z_, complex<T> & sampling_location_acoustic_signal_, int & index_, Vector<T> & sampling_location_r_H_, Vector<T> & sampling_location_z_H_, Vector<complex<T> > & sampling_location_acoustic_signal_H_)`
   - `void Output_Info(T & sampling_location_r_, T & sampling_location_z_, complex<T> & sampling_location_acoustic_signal_, int & index_)`

   Description:

   public; The first overload simulates how AUV_ASD communicates with headquarter and how to transfer all data it has. The second one is a simple version of the first one, which could be used in virtual world.

A.2 AVU_SSD.h

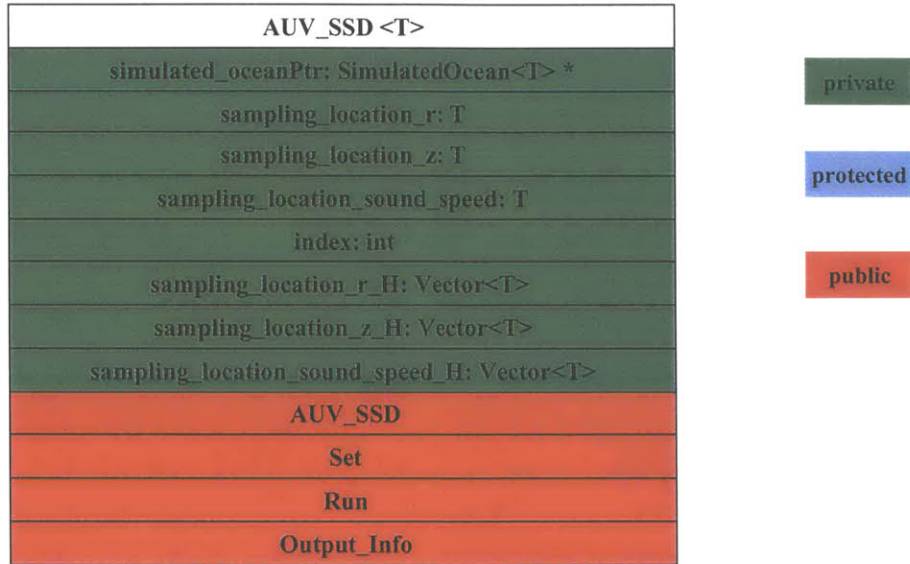


Figure A-2: Class diagram of class AUV_SSD

In this header file, class AUV_SSD is defined, which can simulate an sound speed sensor carried on AUV.

A.2.1 Data Members

1. `SimulatedOcean<T> * simulated_oceanPtr` — private; this is the pointer pointing to the simulated ocean.

2. `T sampling_location_r` — private; This is the horizontal coordinates of current sampling point.

3. `T sampling_location_z` — private; This is the vertical coordinates of current sampling point.

4. `T sampling_location_sound_speed` — private; This is the sound speed value at current sampling point.

5. `int index` — private; this is the index of current AUV_SSD object (we may have several AUV-carring SSD).

6. Vector<T> sampling_location_r_H — private; this is the history record of sampling_location_r.

7. Vector<T> sampling_location_z_H — private; this is the history record of sampling_location_z.

8. Vector<T> sampling_location_sound_speed_H — private; this is the history record of sampling_location_sound_speed.

A.2.2 Member Functions & Operators

1. Name: AUV_SSD

   Overloads:

   - AUV_SSD(void)

   Description:
   public; Constructor. Nothing is done in construction.

   Name: Set

   Overloads:

   - void Set(SimulatedOcean<T> * simulated_oceanPtr_, const T & init_location_r_, const T & init_location_z_, const int & index_)
   - void Set(SimulatedOcean<T> * simulated_oceanPtr_, const Vector<T> & sampling_location_r_H_, const Vector<T> & sampling_location_z_H_, const Vector<T> & sampling_location_sound_speed_H_, const int & index_)
   -

   Description:
   public; The first overload can set up AUV_SSD initial status including initial location, its index number and connecting to the simulated ocean. All the other data members will be automatically generated. By the second overload, we can set up the above 5 data members manually. This can be used in virtual world.

2. Name: Run

   Overloads:

   - void Run(const Vector<T> &start_, const Vector<T> &end_, const T & target_r_, const T & target_z_)

- void Run(const T &target_r_, const T &target_z_)

Description:
public; This function simulates how AUV_SSDs process commands. In the first overload, start_ and end_ are dummy, however this is a flexible interface for upgrading.

3. Name: Output_Info
Overloads:

- void Output_Info(T & sampling_location_r_, T & sampling_location_z_, T & sampling_location_sound_speed_, int & index_, Vector<T> & sampling_location_r_H_, Vector<T> & sampling_location_z_H_, Vector<T> & sampling_location_sound_speed_H_)
- void Output_Info(T & sampling_location_r_, T & sampling_location_z_, T & sampling_location_sound_speed_, int & index_)

Description:
public; The first overload simulates how AUV_SSD communicates with headquarter and how to transfer all data it has. The second one is a simple version of the first one, which could be used in virtual world.

A.3 AVU_SSD_ASD.h

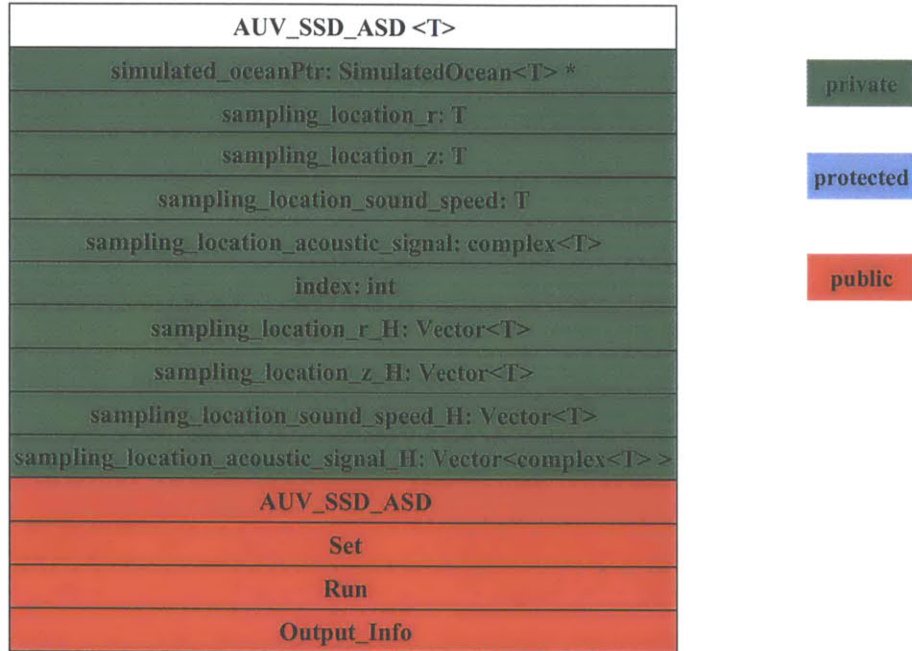


Figure A-3: Class diagram of class AUV_SSD_ASD

In this header file, class AUV_SSD_ASD is defined, which can simulate an AUV carrying a sound speed sensor and a hydrophone.

A.3.1 Data Members

1. `SimulatedOcean<T> * simulated_oceanPtr` — private; this is the pointer pointing to the simulated ocean.

2. `T sampling_location_r` — private; This is the horizontal coordinates of current sampling point.

3. `T sampling_location_z` — private; This is the vertical coordinates of current sampling point.

4. `T sampling_location_sound_speed` — private; This is the sound speed value at current sampling point.

5. complex<T> sampling_location_acoustic_signal — private; this is the acoustical signal received at current sampling point.

6. int index — private; this is the index of current AUV_SSD_ASD object (we may have several AUV-carring SSD_ASD).

7. Vector<T> sampling_location_r_H — private; this is the history record of sampling_location_r.

8. Vector<T> sampling_location_z_H — private; this is the history record of sampling_location_z.

9. Vector<T> sampling_location_sound_speed_H — private; this is the history record of sampling_location_sound_speed.

10. Vector<complex<T> > sampling_location_acoustic_signal_H — private; this is the history record of sampling_location_acoustic_signal.

A.3.2 Member Functions & Operators

1. Name: AUV_SSD_ASD

   Overloads:

   - AUV_SSD_ASD(void)

   Description:

   public; Constructor. Nothing is done in construction.

2. Name: Set

   Overloads:

   - void Set(SimulatedOcean<T> * simulated_oceanPtr_, const T & init_location_r_, const T & init_location_z_, const int & index_)
   - void Set(SimulatedOcean<T> * simulated_oceanPtr_, const Vector<T> & sampling_location_r_H_, const Vector<T> & sampling_location_z_H_, const Vector<T> & sampling_location_sound_speed_H_, const Vector<complex<T> > & sampling_location_acoustic_signal_H_, const int & index_)

   Description:

   public; The first overload can set up AUV_SSD_ASD initial status including initial

location, its index number and connecting to the simulated ocean. All the other data members will be automatically generated. By the second overload, we can set up the above 6 data members manually. this can be used in virtual world.

3. Name: Run

   Overloads:

   - void Run(const Vector<T> &start_, const Vector<T> &end_, const T & target_r_, const T & target_z_)
   - void Run(const T &target_r_, const T &target_z_)

   Description:
   public; This function simulates how AUV_SSD_ASDs process commands. In the first overload, start_ and end_ are dummy, however this is a flexible interface for upgrading.

4. Name: Output_Info

   Overloads:

   - void Output_Info(T & sampling_location_r_, T & sampling_location_z_, T & sampling_location_sound_speed_, complex<T> & sampling_location_acoustic_signal_, int & index_, Vector<T> & sampling_location_r_H_, Vector<T> & sampling_location_z_H_, Vector<T> & sampling_location_sound_speed_H_, Vector<complex<T> > & sampling_location_acoustic_signal_H_)
   - void Output_Info(T & sampling_location_r_, T & sampling_location_z_, T & sampling_location_sound_speed_, complex<T> & sampling_location_acoustic_signal_, int & index_)

   Description:
   public; The first overload simulates how AUV_SSD_ASD communicates with head-quarter and how to transfer all data it has. The second is a simple version of the first one, which could be used in virtual world.

A.4 Bathymetry.h

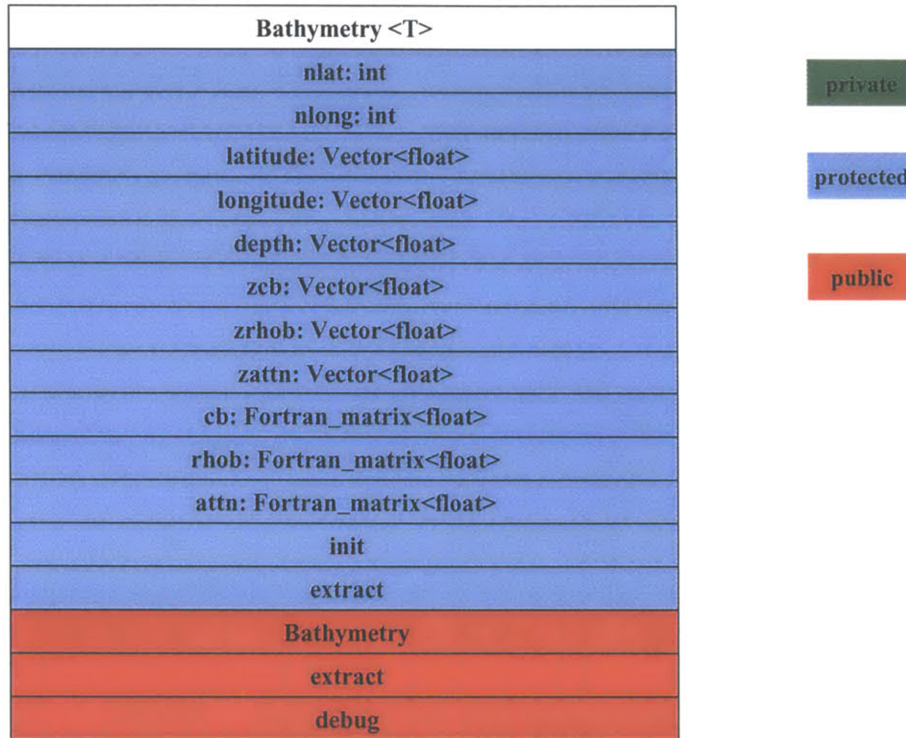


Figure A-4: Class diagram of class Bathymetry

This file was created by Pierre Elisseeff. In this file, class Bathymetry is defined, which enables querying a bathymetry HOPS data file (netcdf format).

A.4.1 Data Members

1. int nlat — protected; not clear.

2. int nlong — protected; not clear.

3. Vector<float> latitude — protected; not clear.

4. Vector<float> longitude — protected; not clear.

5. Vector<float> depth — protected; not clear.

6. Vector<float> zcb — protected; not used.

7. `Vector<float> zrhob` — protected; not used.

8. `Vector<float> zattn` — protected; not used.

9. `Fortran_matrix<float> cb` — protected; not used.

10. `Fortran_matrix<float> rhob` — protected; not used.

11. `Fortran_matrix<float> attn` — protected; not used.

A.4.2 Member Functions & Operators

1. Name: `init`

   Overloads:

   - `void init( char* grids_file_name )`

   Description:

   protected; `grids_file_name` is the database file. this function uploads database.

2. Name: `extract`

   Overloads:

   - `void extract( const Vector<T> &start, const Vector<T> &end, const T &res, Vector<T> &rb, Vector<T> &zb )`
   - `T extract( const Vector<T> &start, T x, T y )`

   Description:

   The first overload is public, in which we input latitude and longitude of start and end points, input resolution, the water-seabed interface line will be generated and output to rb and zb. The second overload is protected, which is an internal function. we can input start point's latitude and longitude and 2-D horizontal local coordinates, then depth at that point will be output.

3. Name: `Bathymetry`

   Overloads:

   - `Bathymetry( void )`

- `Bathymetry( char* g )`

Description:

public; Constructor. In the first overload, default data file will be used; in the second overload, we can input another data file through `g`.

4. Name: `debug`

   Overloads:

   - `void debug( char *file )`

   Description:

   protected; This function is for debug.

A.5 Candidate_points_generate.h

Function `candidate_points_generate` is defined in this file, which determines all possible candidate points for AUV visiting in the next step.

A.5.1 Functions Defined In This File

1. Name: `candidate_points_generate`

   Overloads:

   - `void candidate_points_generate(`
     `const ObservationDatabase<T> & observation_database_, Vector<T> & candidate_points_r_, Vector<T> & candidate_points_z_)`

   Description:

   By inputting `observation_database_` and according to water-seabed interface contained in observation database and AUV performance limit, this function will output candidate points location (see Figure A-5).

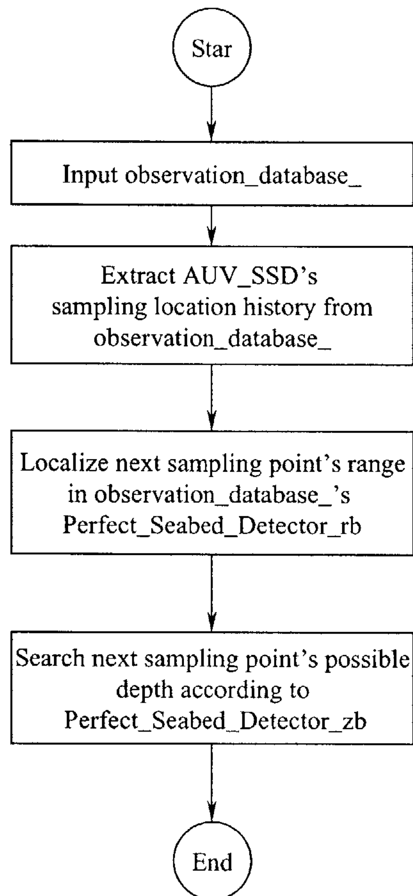


Figure A-5: Flow chart of candidate_points_generate

A.6 ControlAgent.h

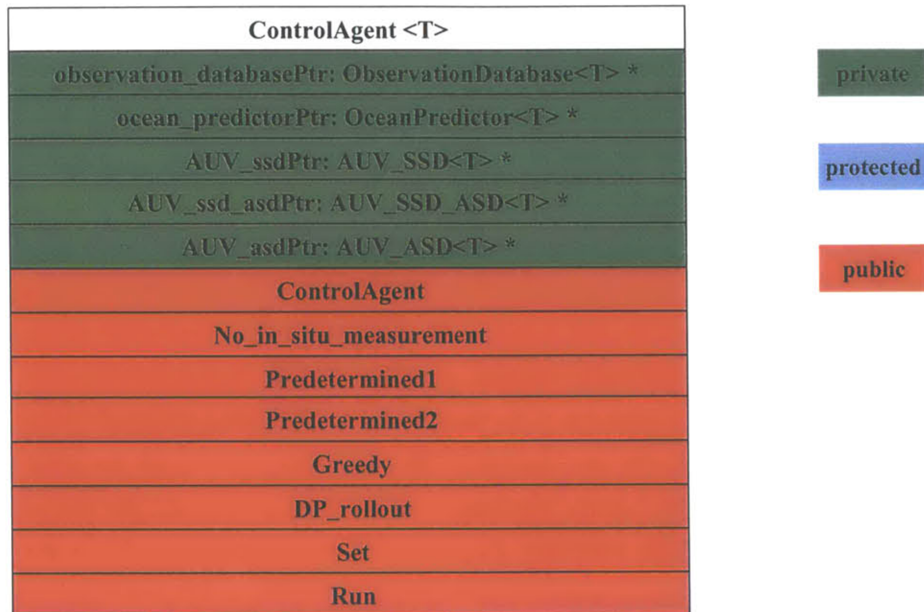


Figure A-6: Class diagram of class ControlAgent

Class ControlAgent is the kernel part of the whole programme. It will determine the next sampling point location based on all current information. Control Agent is the most important module in control center.

A.6.1 Data Members

1. `ObservationDatabase<T> * observation_databasePtr` — private; this pointer points to the Observation Database in control center.

2. `OceanPredictor<T> * ocean_predictorPtr` — private; this pointer points to the Ocean Predictor.

3. `AUV_SSD<T> * AUV_ssdPtr` — private; this is the pointer pointing to those AUV_SSDs that can be controlled by Control Agent.

4. `AUV_SSD_ASD<T> * AUV_ssd_asdPtr` — private; this is the pointer pointing to those AUV_SSD_ASDs that can be controlled by Control Agent.

5. `AUV_ASD<T> * AUV_asdPtr` — private; this is the pointer pointing to those AUV_ASDs that can be controlled by Control Agent.

A.6.2 Member Functions & Operators

1. Name: `ControlAgent`

   Overloads:

   - `ControlAgent(void)`

   Description:

   public; Constructor function. Nothing is done in construction.

2. Name: `Set`

   Overloads:

   - `void Set(ObservationDatabase<T> * observation_databasePtr_, OceanPredictor<T> * ocean_predictorPtr_)`

   Description:

   public; This function let Control Agent connect to Observation Database and Ocean Predictor.

3. Name: `No_in_situ_measurement`

   Overloads:

   - `void No_in_situ_measurement(void)`

   Description:

   public; In this function no in situ measurement will be done and it is mainly for comparison.

4. Name: `Predetermined1`

   Overloads:

   - `void Predetermined1(void)`

   Description:

   public; This function will produce a predetermined linear route for only 1 AUV_SSD.

5. Name: `Predetermined2`

   Overloads:

   - `void Predetermined2(void)`

   Description:
   public; This function will produce another predetermined linear route for only 1 AUV_SSD.

6. Name: `Greedy`

   Overloads:

   - `void Greedy(void)`

   Description:
   public; This function will produce an adaptive route by selecting the point with biggest error — greedy algorithm. Now, only one AUV_SSD is allowed.

7. Name: `DP_rollout`

   Overloads:

   - `void DP_rollout(void)`

   Description:
   public; This function will generate a sub-optimal route for a single AUV_SSD by rollout algorithm based on greedy algorithm.

8. Name: `Run`

   Overloads:

   - `void Run(void)`

   Description:
   public; This function will call another function from `Predetermined1`, `Predetermined2`, `Greedy`, `DP_rollout` based on `global_ControlAgent_model_selection`. This function is the interface function that will be called by Ocean Predictor in the programme.

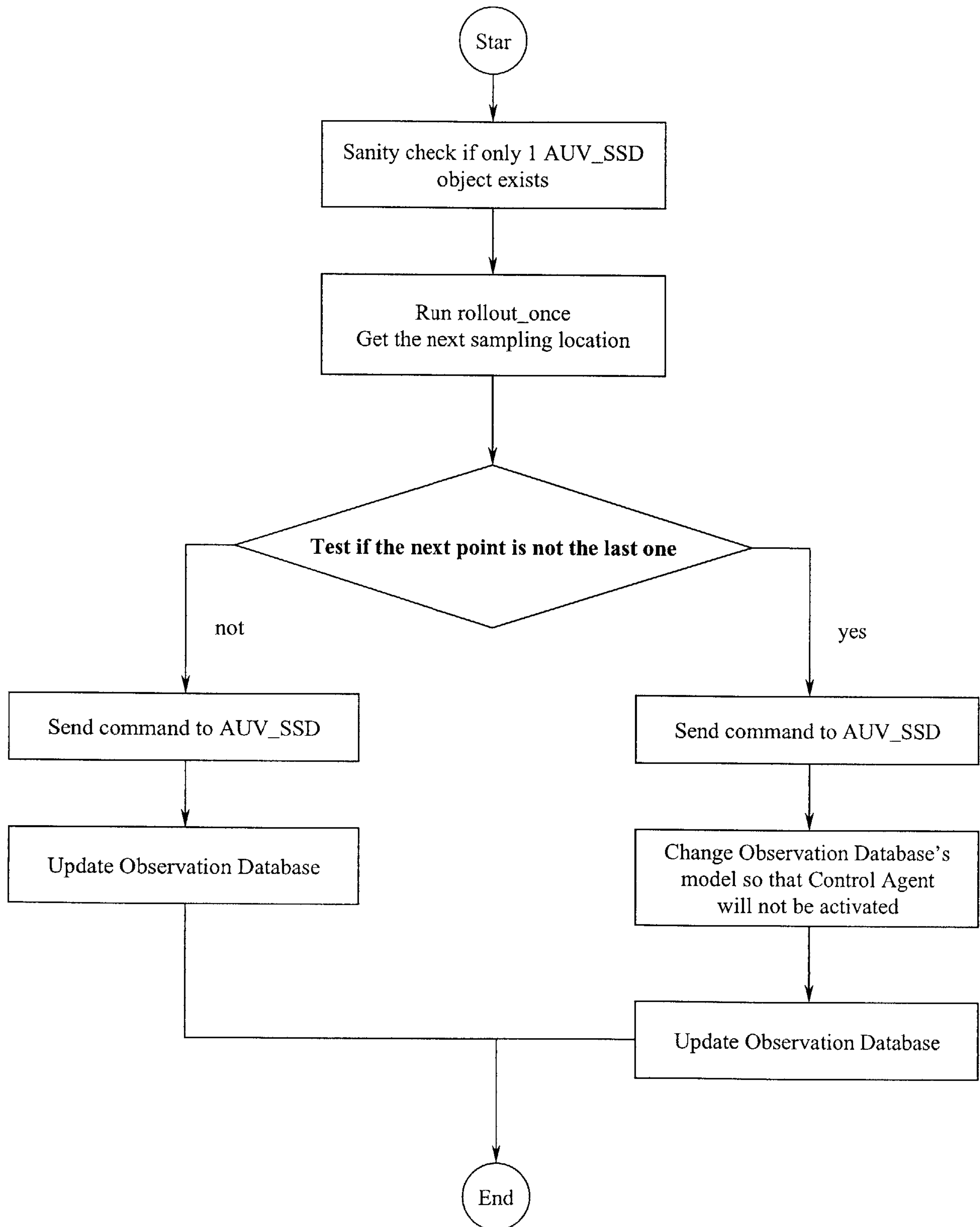


Figure A-7: Flow chart of DP_rollout

A.7 DetectionRange.h

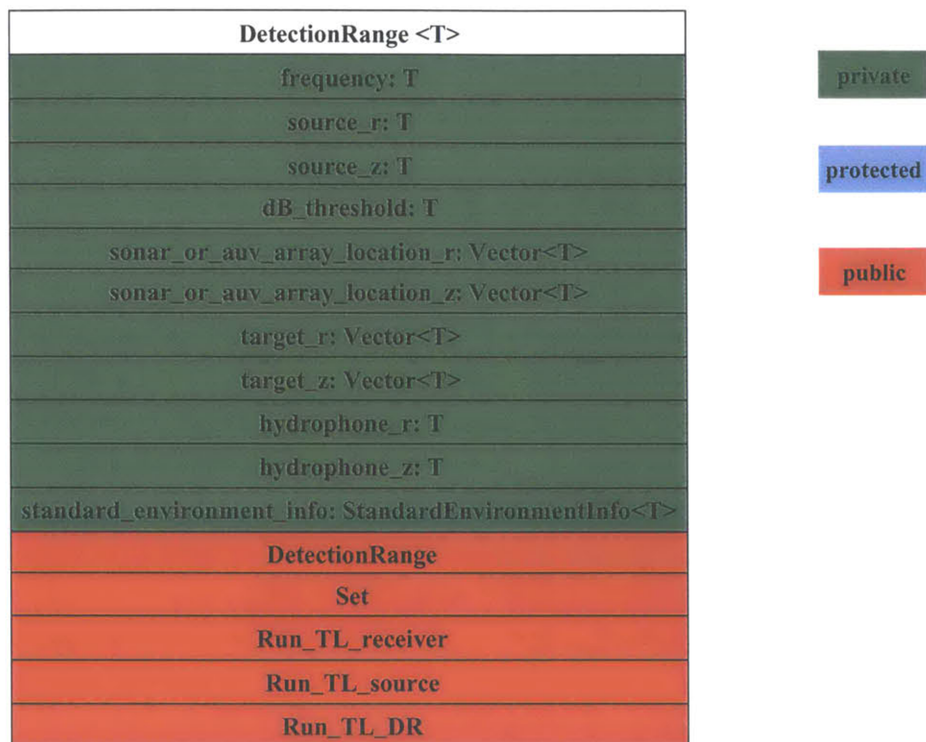


Figure A-8: Class diagram of class DetectionRange

Class DetectionRange is defined in this file, which can calculate transmission loss or calculate detection range.

A.7.1 Data Members

1. `T frequency` — private; this is CW sound source frequency or sonar central frequency.

2. `T source_r` — private; this is sound source horizontal location.

3. `T source_z` — private; this is sound source vertical location.

4. `T dB_threshold` — private; this is the detectable sound strength in dB.

5. `Vector<T> sonar_or_auv_array_location_r` — private; this is the horizontal locations of hydrophones used in function `Run_TL_receiver`.

6. Vector<T> sonar_or_auv_array_location_z — private; this is the vertical locations of hydrophones used in function Run_TL_receiver.

7. Vector<T> target_r — private; this is the horizontal locations of a series of virtual CW sound source used in function Run_TL_source.

8. Vector<T> target_z — private; this is the vertical locations of a series of virtual CW sound source used in function Run_TL_source.

9. T hydrophone_r — private; this is the receiver's horizontal location used in function Run_TL_source.

10. T hydrophone_z — private; this is the receiver's vertical location used in function Run_TL_source.

11. StandardEnvironmentInfo<T> standard_environment_info — private; This is ocean acoustic environment, including all information needed for computation.

A.7.2 Member Functions & Operators

1. Name: DetectionRange

   Overloads:

   - DetectionRange(void)

   Description:

   public; Constructor function. Nothing is done in construction.

2. Name: Set

   Overloads:

   - void Set(const T & frequency_, const T & source_r_, const T & source_z_, const T & dB_threshold_, const Vector<T> & sonar_or_auv_array_location_r_, const Vector<T> & sonar_or_auv_array_location_z_, const StandardEnvironmentInfo<T> & Standard_Environment_Info_)

   Description:

   public; Input and setup all data members.

3. Name: `Run_TL_receiver`

   Overloads:

   - `void Run_TL_receiver(Vector<T> & TL_)`

   Description:

   public; This function computes TLs at points of
   `(sonar_or_auv_array_location_r, sonar_or_auv_array_location_z)` with CW sound source at `(global_source_r, global_source_z)`.

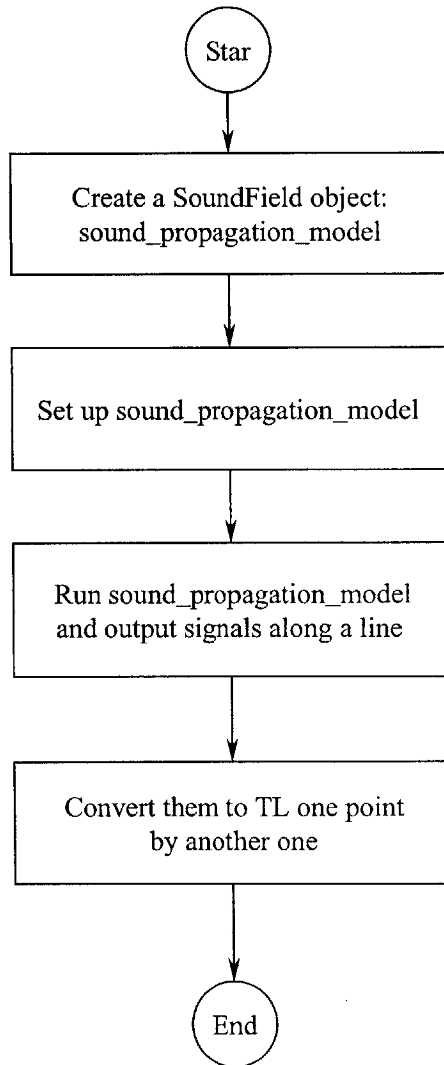


Figure A-9: Flow chart of Run_TL_receiver

4. Name: `Run_TL_source`

   Overloads:

   - `void Run_TL_source(Vector<T> & TL_)`

   Description:

   public; This function computes TLs at `(global_hydrophone_r, global_hydrophone_z)` with CW sound source at points of `(global_target_r, global_target_z)`.

5. Name: `Run_DR`

   Overloads:

   - `void Run_DR(T & detection_range_)`

   Description:

   public; This function computes detection range, based on `dB_threshold` and output from `Run_TL_source`.

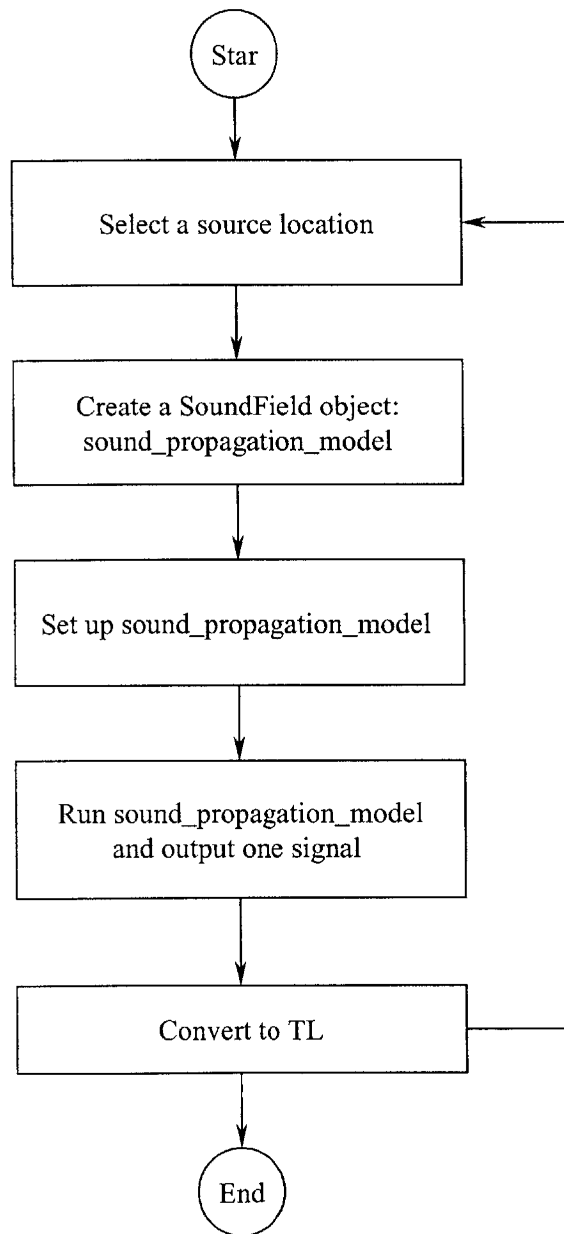


Figure A-10: Flow chart of Run_TL_source

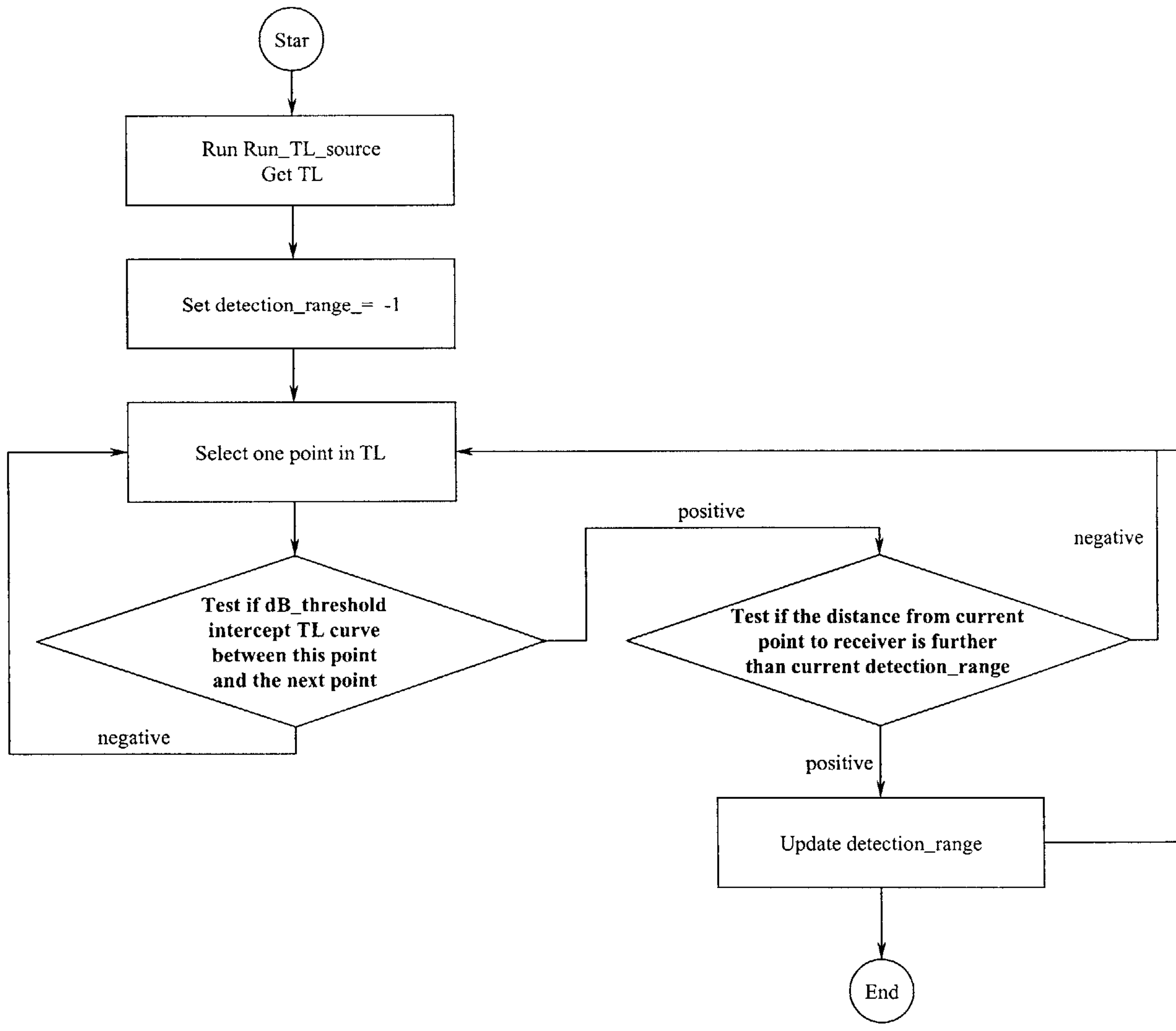


Figure A-11: Flow chart of Run_DR

A.8 FixedWaterSensor.h

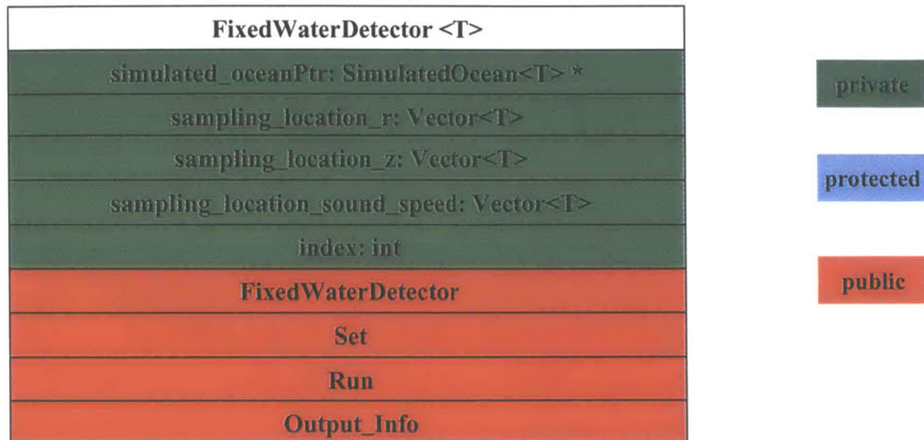


Figure A-12: Class diagram of class FixedWaterSensor

Class FixedWaterSensor is defined in this file, which can simulate local water sound speed sensors (array).

A.8.1 Data Members

1. `SimulatedOcean<T> * simulated_oceanPtr` — private; this is the pointer pointing to the simulated ocean.

2. `Vector<T> sampling_location_r` — private; This is the horizontal coordinates of sensors.

3. `Vector<T> sampling_location_z` — private; This is the vertical coordinates of sensors.

4. `Vector<T> sampling_location_sound_speed` — private; This is the sound speed value at sensors.

5. `int index` — private; this is the index of current FixedWaterSensor object.

A.8.2 Member Functions & Operators

1. Name: `FixedWaterSensor`

   Overloads:

   - `FixedWaterSensor(void)`

   Description:

   public; Constructor. Nothing is done in construction.

2. Name: `Set`

   Overloads:

   - `void Set(SimulatedOcean<T> * simulated_oceanPtr_, const Vector<T> & sampling_location_r_, const Vector<T> & sampling_location_z_, const int & index_)`
   - `void Set(SimulatedOcean<T> * simulated_oceanPtr_, const Vector<T> & sampling_location_r_, const Vector<T> & sampling_location_z_, const Vector<T> & sampling_location_sound_speed_, const int & index_)`

   Description:

   public; In the first overload we manually set up 4 data members and sensors will automatically measure sound speeds in the simulated ocean and then output to `sampling_location_sound_speed`. In the second overload, we manually set up all data members.

3. Name: `Run`

   Overloads:

   - `Vector<T> Run(void)`

   Description:

   public; This function forces sensors to measure sound speeds and output them.

4. Name: `Output_Info`

   Overloads:

- `void Output_Info(Vector<T> & sampling_location_r_,`
  `Vector<T> & sampling_location_z_, Vector<T> & sampling_location_sound_speed_,`
  `int & index_)`
- `void Output_Info(Vector<T> & sampling_location_sound_speed_, int & index_)`

Description:

public; The first overload outputs all information. The second one is a simple version of the first overload. Only measurement results and index number will be output.

A.9 fmat.h

This header file originates from Template Numerical Toolkit (TNT). It has been added and changed by Pierre Elisseeff and Ding Wang. In this file, the class Fortran_matrix is defined, which owns most properties of matrix in Fortran. Moreover, many useful functions and operators for matrix are constructed. Fortran_matrix is 1-offset.

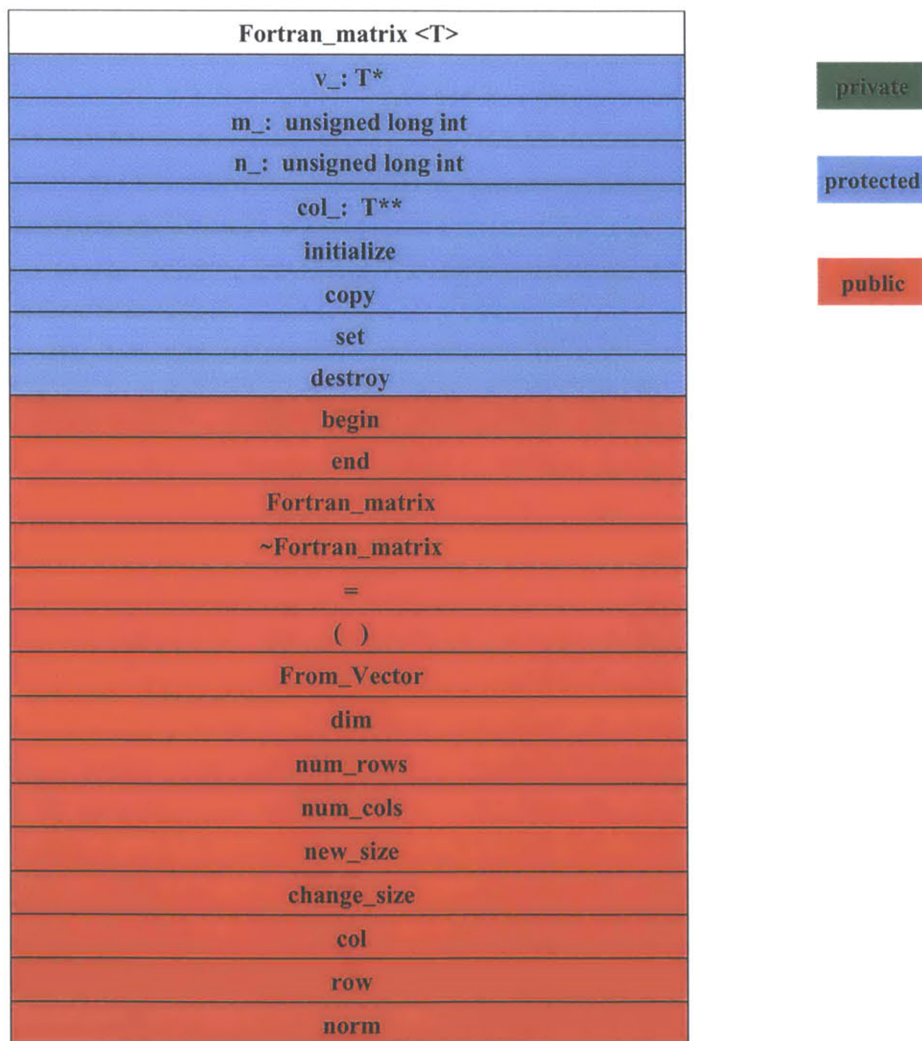


Figure A-13: Class diagram of class Fortran_matrix

A.9.1 Data Members

1. `T* v_` — protected; this is the 0-offset array containing elements of matrix.

2. `unsigned long int m_` — protected; this is the number of rows in matrix.

3. `unsigned long int n_` — protected; this is the number of columns in matrix.

4. `T** col_` — protected; this is a pointer array which stores pointers pointing to all the first row elements.

A.9.2 Member Functions & Operators

1. Name: `initialize`

   Overloads:

   - `void initialize(unsigned long int M, unsigned long int N)`

   Description:
   protected; This is an internal function to create `v_` and `col_`. 1-offset pointers to the first row elements are assigned to `col_` and `col_` itself is also adjusted to be 1-offset.

2. Name: `copy`

   Overloads:

   - `void copy(const T* v)`

   Description:
   protected; This function copy `v[ ]` to `v_[ ]`. Note that this function must be used after initialize and M×N in function initialize must be equal to the length of `v`. It is not so clear about what this function does when `TNT_UNROLL` is defined, but when `TNT_UNROLL` is not defined, it just copies `v` to `v_` piece by piece.

3. Name: `set`

   Overloads:

   - `void set(const T& val)`

Description:

protected; this function is similar to function copy, but now the input val must be a scalar. In this function, `val` is assigned to all elements of `v_[ ]` piece by piece.

4. Name: `destroy`

   Overloads:

   - `void destroy( )`

   Description:

   protected; this function destructs `v_` and `col_` and free space.

5. Name: `begin`

   Overloads:

   - `T* begin( )`
   - `const T* begin( )`

   Description:

   public; This function returns back the pointer pointing to the first element. In the second overload it is a constant function and the returned pointer points to a constant datum.

6. Name: `end`

   Overloads:

   - `T* end( )`
   - `const T* end( )`

   Description:

   public; This function returns back the pointer pointing to the last element. In the second overload it is a constant function and the returned pointer points to a constant datum.

7. Name: `Fortran_matrix`

   Overloads:

   - `Fortran_matrix( )`

- `Fortran_matrix(const Fortran_matrix<T> &A)`
- `Fortran_matrix(unsigned long int M, unsigned long int N, const T& value = T(0))`
- `Fortran_matrix(unsigned long int M, unsigned long int N, const T* v)`
- `Fortran_matrix(unsigned long int M, unsigned long int N, char *s)`
- `Fortran_matrix(unsigned long int M, unsigned long int N, const Vector<T> & x)`

Description:
public; This is the constructor function. The 1st overload constructs a null matrix; the 2nd overload constructs a copy of matrix `A`; the 3rd overload constructs a M×N matrix and assign scalar value to each element; the 4th overload constructs a M×N matrix copy of M×N-element array v; the 5th overload constructs a matrix copy of N-element string `s`. the 6th overload constructs a M×N matrix copy of M×N-element vector `x`.

8. Name: ~ `Fortran_matrix`
   Overloads:

   - ~ `Fortran_matrix( )`

   Description:
   public; This is the destructor function. It deletes the matrix and frees space.

9. Name: =
   Overloads:

   - `Fortran_matrix<T>& operator=(const Fortran_matrix<T> &A)`
   - `Fortran_matrix<T> operator=(const Fortran_coordinate_matrix<T> &A)`
   - `Fortran_matrix<T>& operator=(const T& scalar)`

   Description:
   public; The 1st overload assigns matrix A to the matrix at left of '='; it is not clear about what the 2nd overload does; Refer to the original file for details; the 3rd overload assigns a scalar to each element of the matrix at left of '='.

10. Name: `( )`

    Overloads:

    - `inline reference operator( )(unsigned long int i, unsigned long int j)`
    - `inline const_reference operator( ) (unsigned long int i, unsigned long int j) const`

    Description:
    public; By this 1-offset sign, an element of matrix can be extracted, e.g. x(i, j) is the element at i th row and j th column of x. The 2nd overload is a constant operator and return back a constant reference.
    this operator has other 2 overloads:

    - `Region operator( )(const Index1D &I, const Index1D &J)`
    - `const_Region operator( )(const Index1D &I, const Index1D &J) const`

    However, it is not clear about what these 2 overloads do. Refer to the original file for more details.

11. Name: `From_Vector`

    Overloads:

    - `Fortran_matrix<T> From_Vector(unsigned long int M, unsigned long int N, const Vector<T> & x)`

    Description:
    public; This function resizes the matrix and columnwise copy M×N-element vector **x** to a M×N matrix.

12. Name: `dim`

    Overloads:

    - `unsigned long int dim(unsigned long int d) const`
    - `Vector<int> dim(void)`

Description:

public; This function outputs matrix's dimension. the 1st overload outputs matrix's row number or column number: when `d=1` this function outputs row number, when `d=2` it outputs column number; the 2nd overload outputs row and column number together in a vector.

13. Name: `num_rows`

    Overloads:

    - `unsigned long int num_rows( ) const`

    Description:

    public; This function returns back the row number.

14. Name: `num_cols`

    Overloads:

    - `unsigned long int num_cols( ) const`

    Description:

    public; This function returns back the column number.

15. Name: `newsize`

    Overloads:

    - `Fortran_matrix<T>& newsize(unsigned long int M, unsigned long int N)`

    Description:

    public; This function can change the matrix's dimension to be M×N by destroying and creating. So content of the matrix could be changed. Refer to the file for details.

16. Name: `change_size`

    Overloads:

    - `Fortran_matrix<T> & change_size(unsigned long int M, unsigned long int N)`

Description:

public; This function can also change the matrix's dimension to be M×N. But the content of vector is kept. Refer to the file for details.

17. Name: col

    Overloads:

    - Fortran_matrix<T> col (unsigned long int i)

    Description:

    public; This function can output the whole i th column of the matrix.

18. Name: row

    Overloads:

    - Fortran_matrix<T> row (unsigned long int i)

    Description:

    public; This function can output the whole i th row of the matrix.

19. Name: norm

    Overloads:

    - T norm ( )

    Description:

    public; This function is only applicable to matrix with 1 column. it outputs the norm of the column.

A.9.3 Functions And Operators Defined In This File

1. Name: <<

    Overloads:

    - ostream& operator<<(ostream &s, const Fortran_matrix<T> &A)

    Description:

    By this operator, matrix A's dimension information and content can be output by I/O stream s. E.g. cout<<x<<endl; .

2. Name: >>

   Overloads:

   - istream& operator>>(istream &s, Fortran_matrix<T> &A)

   Description:

   By this operator, matrix A's dimension information and content can be input from I/O stream s. E.g. cin>>x; .

3. Name: +

   Overloads:

   - Fortran_matrix<T> operator+(const Fortran_matrix<T> &A,
     const Fortran_matrix<T> &B)

   Description:

   '+' let matrix A be able to plus another matrix B which has the same dimension. It returns back the summation.

4. Name: -

   Overloads:

   - Fortran_matrix<T> operator-(const Fortran_matrix<T> &A,
     const Fortran_matrix<T> &B)

   Description:

   '−' let matrix A be able to subtracted by matrix B which has the same dimension. It returns back the result.

5. Name: mult_element

   Overloads:

   - Fortran_matrix<T> mult_element(const Fortran_matrix<T> &A,
     const Fortran_matrix<T> &B)

   Description:

   'mult_element' let matrix A be able to elementwise multiply matrix B which has the same dimension. This is function is the same as '.*' in MATLAB.

6. Name: `transpose`

   Overloads:

   - `Fortran_matrix<T> transpose(const Fortran_matrix<T> &A)`

   Description:

   this function just returns transpose of `A`.

7. Name: `transconj`

   Overloads:

   - `Fortran_matrix< complex<T> > transconj(const Fortran_matrix< complex<T> > &A)`

   Description:

   this function just returns hermitian of `A`.

8. Name: `matmult`

   Overloads:

   - `inline Fortran_matrix<T> matmult(const Fortran_matrix<T> &A, const Fortran_matrix<T> &B)`
   - `inline int matmult(Fortran_matrix<T>& C, const Fortran_matrix<T> &A, const Fortran_matrix<T> &B)`
   - `Vector<T> matmult(const Fortran_matrix<T> &A, const Vector<T> &x)`

   Description:

   the 1st overload just returns back multiplication of `A` and `B`; in the 2nd one, matrix `A` times matrix `B` and store the result in matrix `C`; in the 3rd one, matrix `A` times a vector and generate another vector.

9. Name: `*`

   Overloads:

   - `inline Fortran_matrix<T> operator*(const Fortran_matrix<T> &A, const Fortran_matrix<T> &B)`

- `inline Vector<T> operator*(const Fortran_matrix<T> &A, const Vector<T> &x)`
- `inline Fortran_matrix<T> operator*(const Fortran_matrix<T> &A, const TT &x)`

Description:

the 1st overload just returns back multiplication of `A` and `B`; in the 2nd overload, matrix `A` times a vector and generate another vector; in the 3rd one, matrix `A` times a scalar `x`.

10. Name: `/`

    Overloads:

    - `inline Fortran_matrix<T> operator/(const Fortran_matrix<T> &A, const TT &x)`

    Description:

    By this operator, matrix `A` is divided by a scalar `x`.

A.10 Greedy_algorithm.h

Function greedy_algorithm will generate a whole sampling process based on greedy algorithm with respect to sound speed standard deviation.

A.10.1 Functions Defined In This File

1. Name: `greedy_algorithm`

   Overloads:

   - `void greedy_algorithm(ObservationDatabase<T> & virtual_observation_database, OceanPredictor<T> & virtual_ocean_predictor, AUV_SSD<T> virtual_AUV_ssd[ ])`

   Description:

   Through this function, `virtual_AUV_ssd` will finish the sampling path by selecting the point with biggest sound speed standard deviation as the next sampling point, and virtual_observation_database will record all measurement results. Note that the 3 inputs must connect to each other and `virtual_AUV_ssd` must connect to a simulated ocean. Now, only one single `virtual_AUV_ssd` is allowed.

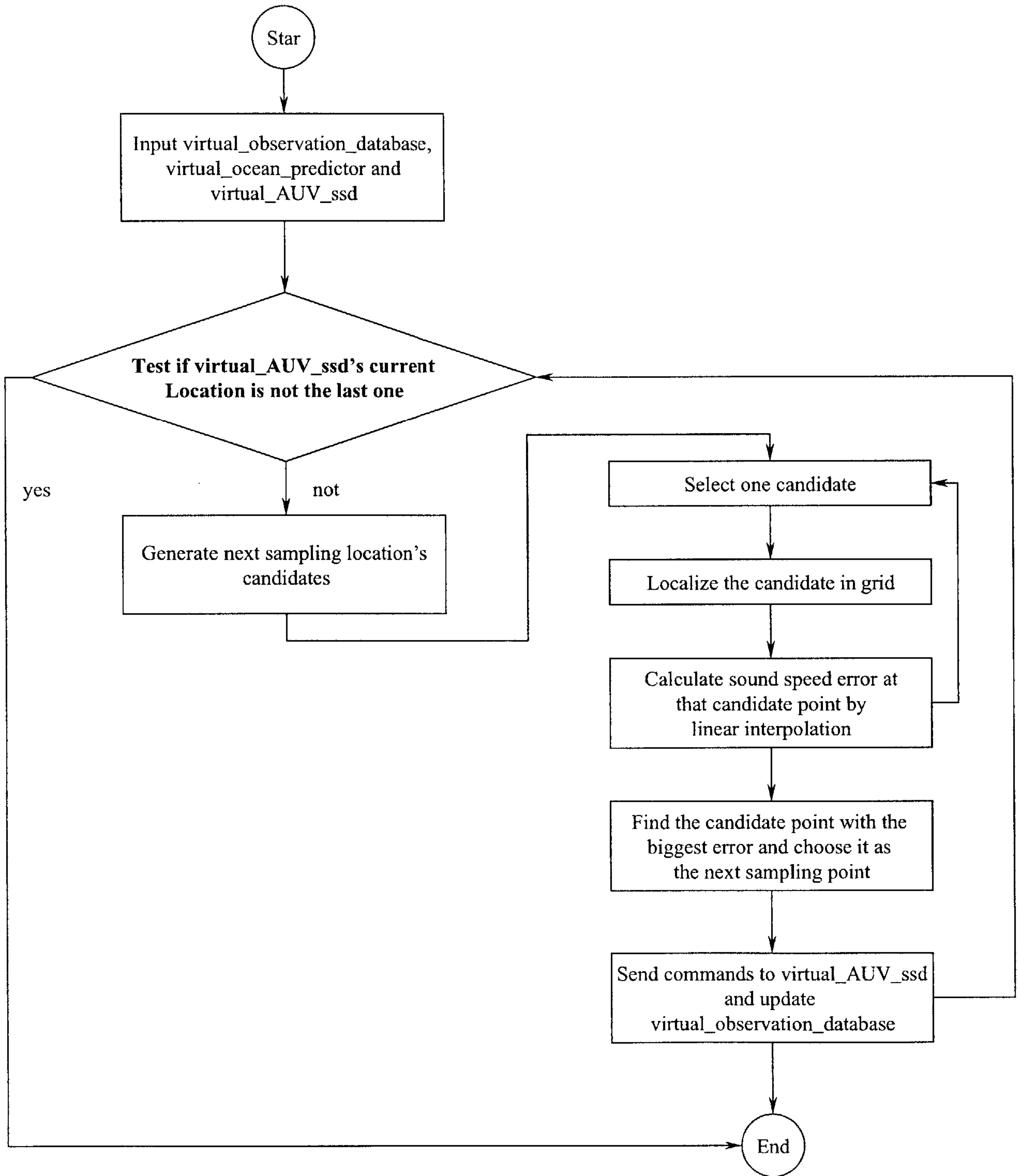


Figure A-14: Flow chart of greedy_algorithm

A.11 MatchedFieldProcessing.h

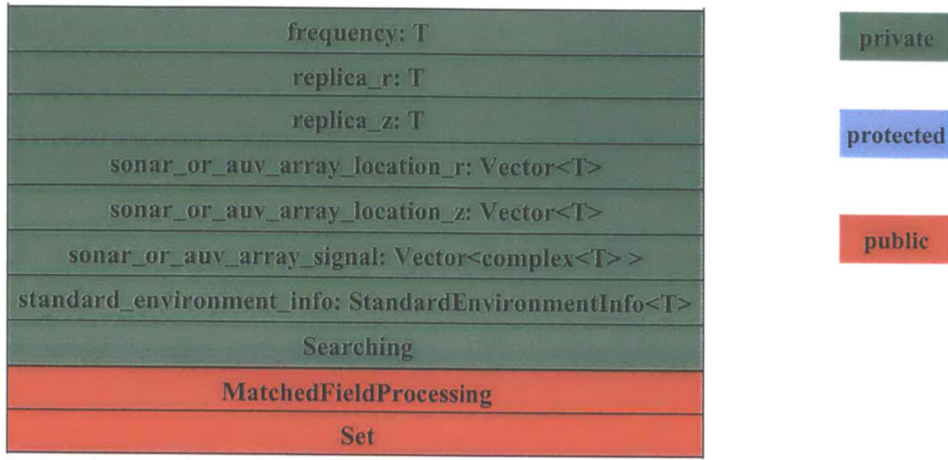


Figure A-15: Class diagram of class MatchedFieldProcessing

In this file, class MatchedFieldProcessing is defined, which can do CW matched field processing and find main lobe peak and max side lobe peak.

A.11.1 Data Members

1. `T frequency` — private; this is the central frequency of sonar and sound source.
2. `Vector<T> replica_r` — private; this is the horizontal axis of replica sources grid.
3. `Vector<T> replica_z` — private; this is the vertical axis of replica sources grid.
4. `Vector<T> sonar_or_auv_array_location_r` — private; this is the horizontal coordinates of hydrophones.
5. `Vector<T> sonar_or_auv_array_location_z` — private; this is the vertical coordinates of hydrophones.
6. `Vector<complex<T> > sonar_or_auv_array_signal` — private; this is the signals received by hydrophones.
7. `StandardEnvironmentInfo<T> standard_environment_info` — private; this is the built-in environment model in sonar system, including all information needed for computation.

A.11.2 Member Functions & Operators

1. Name: Searching

   Overloads:

   - void Searching(Fortran_matrix<T> & ambiguity_function_, Vector<int> & main_lobe_peak_index_, Vector<int> & max_side_lobe_peak_index_)

   Description:

   private; This function is to find the main lobe peak and max side lobe peak in ambiguity_function_ and then output indices of them.

2. Name: MatchedFieldProcessing

   Overloads:

   - MatchedFieldProcessing(void)

   Description:

   public; Constructor. Nothing is done in construction.

3. Name: Set

   Overloads:

   - void Set(const T & frequency_, const Vector<T> & replica_r_, const Vector<T> & replica_z_, const Vector<T> & sonar_or_auv_array_location_r_, const Vector<T> & sonar_or_auv_array_location_z_, const Vector<complex<T> > & sonar_or_auv_array_signal_, const StandardEnvironmentInfo<T> & Standard_Environment_Info_)

   Description:

   public; By this function, we can input and set up all data members.

4. Name: Run

   Overloads:

   - void Run(Fortran_matrix<T> & ambiguity_function_, Vector<int> & main_lobe_peak_index_, Vector<int> & max_side_lobe_peak_index_)

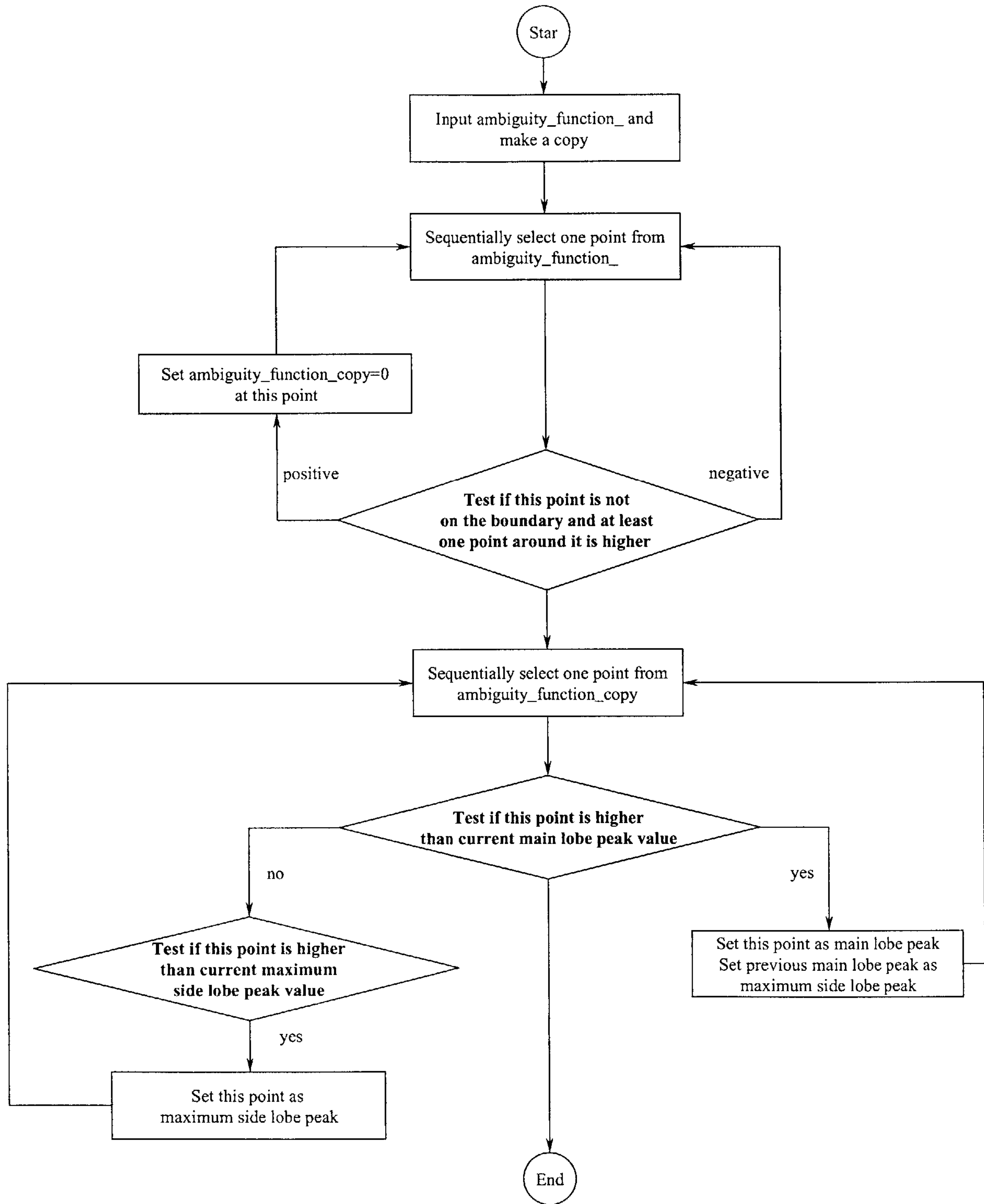


Figure A-16: Flow chart of Searching

Description:

public; This function will run matched field processing and output ambiguity function, index of main lobe peak and index of max side lobe peak.

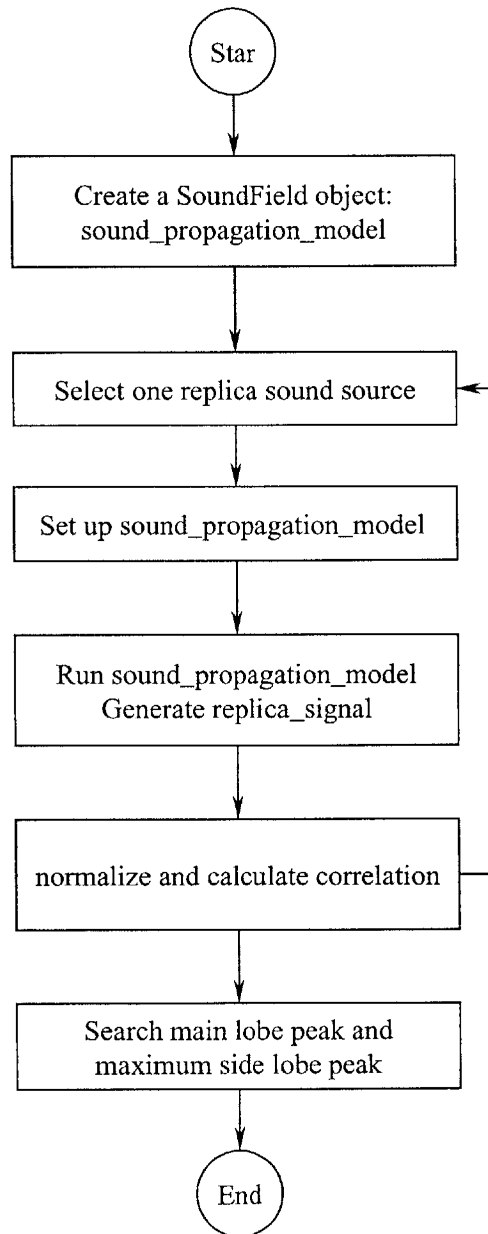


Figure A-17: Flow chart of Run

A.12 ObjectiveAnalysis.h

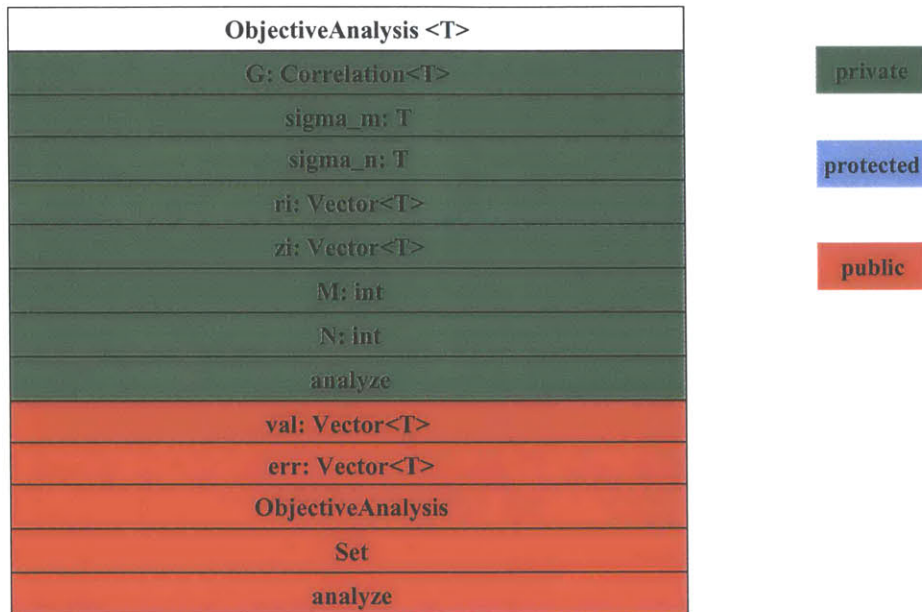


Figure A-18: Class diagram of class ObjectiveAnalysis

This file was created by Pierre Elisseeff and adapted by Ding Wang. In this file, class ObjectiveAnalysis is defined, which can objectively analyzes a set of raw data points on a vertical plane (2D analysis).

A.12.1 Data Members

1. `Correlation<T> G` — private; this is the correlation function used in this class, which describes horizontal and vertical correlation in water sound speed profile.

2. `T sigma_m` — private; this is a priori sound speed field standard deviation.

3. `T sigma_n` — private; this is a priori sound speed noise standard deviation.

4. `Vector<T> ri` — private; this is the horizontal axis of a grid.

5. `Vector<T> zi` — private; this is the vertical axis of a grid.

6. `int M` — private; `M` is the length of vector `zi`.

7. int N — private; N is the length of vector ri.

8. Vector<T> val — public; this is the mean field of sound speed profile. this data member is for output.

9. Vector<T> err — public; this is the standard deviation field of sound speed profile. this data member is for output.

A.12.2 Member Functions & Operators

1. Name: ObjectiveAnalysis

   Overloads:

   - ObjectiveAnalysis(void)

   Description:

   public; Constructor. Nothing is done in construction.

2. Name: Set

   Overloads:

   - void Set(const Correlation<T> &G_, const T &sigma_m_, const T &sigma_n_, const Vector<T> &ri_, const Vector<T> &zi_)

   Description:

   public; Through this function, all private data members will be set up.

3. Name: analyze

   Overloads:

   - Vector<T> analyze(const Vector<T> &rd, const Vector<T> &zd, const Vector<T> &cd, const Fortran_matrix<T> &R, const Fortran_matrix<T> &E, const T &r1, const T &z1)
   - Vector<T> analyze(const Vector<T> &rd, const Vector<T> &zd, const Vector<T> &cd)
   - void analyze(const Vector<T> &rd, const Vector<T> &zd, const Vector<T> &cd, Fortran_matrix<T> & val_matrix_, Fortran_matrix<T> & err_matrix_)

Description:

The first overload is private, in which `rd` and `zd` are measurement locations' coordinates and `cd` is the corresponding sound speeds, `R` is a priori noise covariance matrix, `E` is observation matrix. This overload computes field estimate for one grid point along with associated error. Assumes the data is zero-mean. Refer to the original file for more details.

The second overload is public, in which `rd`, `zd` and `cd` are the same as in the first overload. This overload can compute field estimate along with associated error for the whole output grid using the raw data.

The third overload is public, in which `rd`, `zd` and `cd` are the same as before. `val_matrix_` and `err_matrix_` are mean and error field of sound speed profile. They are the same as data members `val` and `err` respectively, but in matrix format. Through this overload, we can do the same thing as in the second overload and moreover, we can directly output `val` and `err` in matrix format.

A.13 ObservationDatabase.h

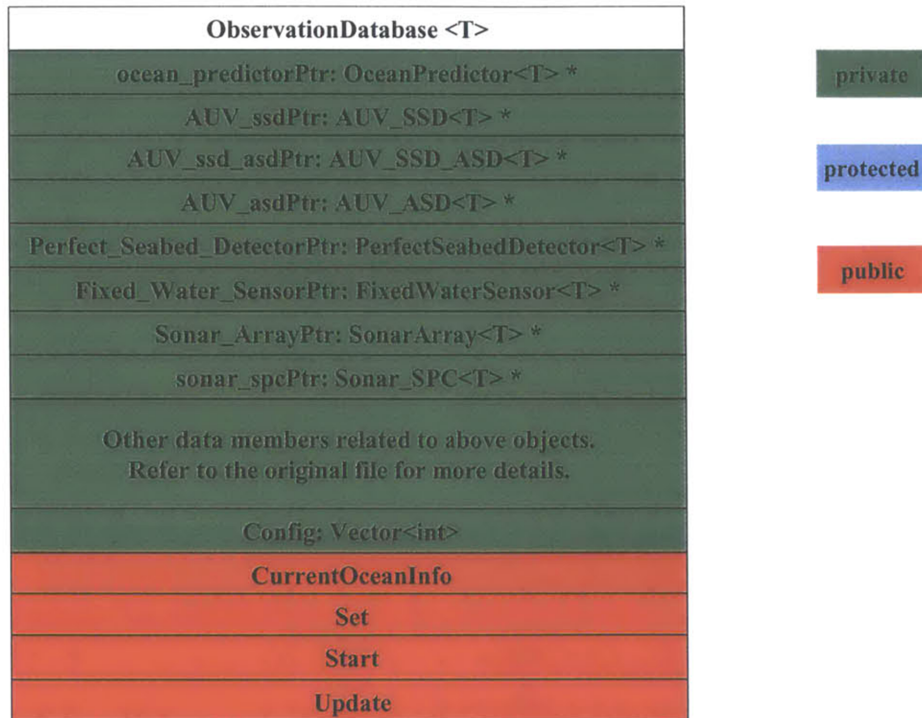


Figure A-19: Class diagram of class ObservationDatabase

Class ObservationDatabase is the communication part of control center, which is used to communicate with external world and store observation data from ocean water column and seabed. Observation Database is an important module in control center.

A.13.1 Data Members

1. `OceanPredictor<T> * ocean_predictorPtr` — private; this pointer points to the ocean predictor.

2. `AUV_SSD<T> * AUV_ssdPtr` — private; this is the pointer pointing to the first AUV_SSD object.

3. `AUV_SSD_ASD<T> * AUV_ssd_asdPtr` — private; this is the pointer pointing to the first AUV_SSD_ASD object.

4. AUV_ASD<T> * AUV_asdPtr — private; this is the pointer pointing to the first AUV_ASD object.

5. PerfectSeabedDetector<T> * Perfect_Seabed_DetectorPtr — private; this is the pointer pointing to the first PerfectSeabedDetector object.

6. FixedWaterSensor<T> * Fixed_Water_SensorPtr — private; this is the pointer pointer pointing to the first FixedWaterSensor object.

7. SonarArray<T> * Sonar_ArrayPtr — private; this is the pointer pointing to the first SonarArray object.

8. Sonar_SPC<T> * sonar_spcPtr — private; this is the pointer pointing to the Sonar_SPC object.

9. Vector<int> Config — private; this is to contain **global_fleet_config**.

10. In this class, it has a lot of data members pointing to data members of AUV_SSD, AUV_SSD_ASD, AUV_ASD, PerfectSeabedDetector, FixedWaterSensor, SonarArray and Sonar_SPC. Since there are too many such data members, it is not suitable to introduce them one by one here. Please refer to the original file for more details.

A.13.2 Member Functions & Operators

1. Name: ObservationDatabase

   Overloads:

   - ObservationDatabase(void)

   Description:

   public; Constructor function. Nothing is done in construction.

2. Name: Set

   Overloads:

   - void Set(OceanPredictor<T> * ocean_predictorPtr_, AUV_SSD<T> * AUV_ssdPtr_, AUV_SSD_ASD<T> * AUV_ssd_asdPtr_, AUV_ASD<T> * AUV_asdPtr_, PerfectSeabedDetector<T> * Perfect_Seabed_DetectorPtr_, FixedWaterSensor<T>

* Fixed_Water_SensorPtr_, SonarArray<T> * Sonar_ArrayPtr_, Sonar_SPC<T>
* sonar_spcPtr_, const Vector<int> & Config_)

Description:

public; Through this initialization function, ObservationDatabase connects to all external sensors, sonar system and OceanPredictor in control center.

3. Name: **Start**

   Overloads:

   - void Start(void)

   Description:

   public; This function starts the whole simulation framework. By this function, ObservationDatabase communicates with all external sensors, sonar system and collects all useful information from them, and then activate Ocean Predictor.

4. Name: **Update**

   Overloads:

   - void Update(void)

   Description:

   public; By this function, ObservationDatabase can update all information related to mobile sensors, and then activate Ocean Predictor.

A.14 OceanPredictor.h

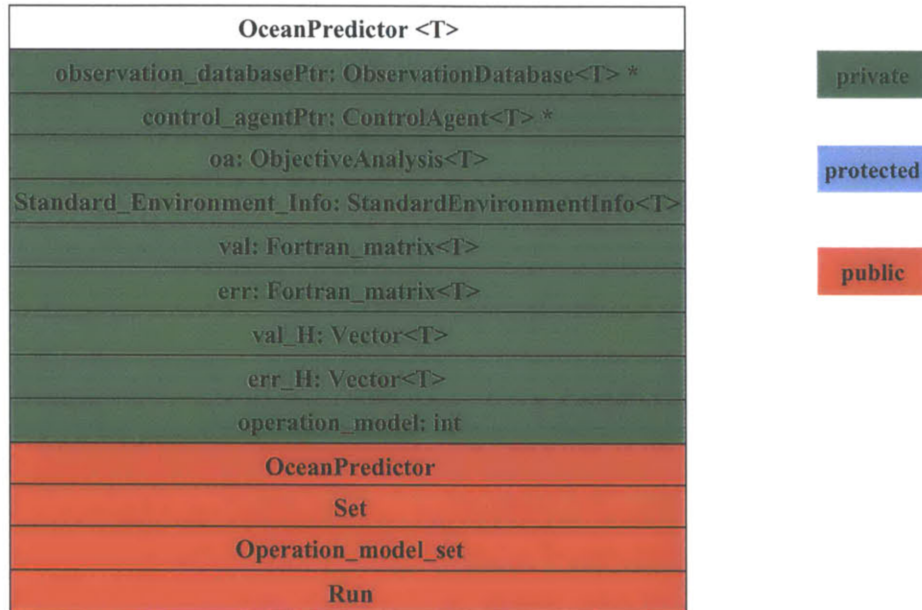


Figure A-20: Class diagram of class OceanPredictor

In this file, class OceanPredictor is defined, which is used to analyze the raw data, estimate ocean acoustic environment, output mean and standard deviation. It should include two parts, one for water column and one for seabed. Now, the seabed is supposed to be known exactly, so only water column part exists. Ocean Predictor is an important module in control center.

A.14.1 Data Members

1. `ObservationDatabase<T> * observation_databasePtr` — private; this pointer points to the Observation Database.

2. `ControlAgent<T> * control_agentPtr` — private; this pointer points to the Control Agent.

3. `ObjectiveAnalysis<T> oa` — private; By this object member, we can objectively analyze raw measurement data in water column.

4. StandardEnvironmentInfo<T> Standard_Environment_Info — private; Currently it is useless, but it will be used in upgrading.

5. Fortran_matrix<T> val — private; Analysis result: mean value of water Sound Speed Profile.

6. Fortran_matrix<T> err — private; Analysis result: standard deviation of water Sound Speed Profile.

7. Vector<T> val_H — private; history record of val. Note that from val to val_H, it's columnwise.

8. Vector<T> err_H — private; history record of err. Note that from err to err_H, it's columnwise.

9. int operation_model — private; this is an internal indicator. 0: Ocean Predictor will call Control Agent, this is for running in real world; 1: Ocean Predictor will not call Control Agent, this is for running in virtual world.

A.14.2 Member Functions & Operators

1. Name: OceanPredictor

   Overloads:

   - OceanPredictor(void)

   Description:

   public; Constructor function. Nothing is done in construction.

2. Name: Set

   Overloads:

   - void Set(ObservationDatabase<T> * observation_databasePtr_, ControlAgent<T> * control_agent_Ptr_, const Correlation<T> & G_, const T & sig_c_, const T & sig_n_, const Vector<T> & water_grid_r_, const Vector<T> & water_grid_z_)
   - void Set(ObservationDatabase<T> * observation_databasePtr_, const Correlation<T> & G_, const T & sig_c_, const T & sig_n_, const Vector<T> & water_grid_r_, const Vector<T> & water_grid_z_)

- void Set(ObservationDatabase<T> * observation_databasePtr_)

Description:

public; The first overload can build up all connections, set up oa and set operation_model=0. this is used in real world.

The second overload can build up all necessary connections and set up oa and set operation_model=1. this is used in virtual world.

The third overload can connect Observation Database and set operation_model=1. this is used in virtual world.

3. Name: Operation_model_set

   Overloads:

   - void Operation_model_set(const int operation_model_)

   Description:

   public; By this function, we can set operation_model to be 0 or 1.

4. Name: Run

   Overloads:

   - void Run(void)
   - void Run(Fortran_matrix<T> & val_, Fortran_matrix<T> & err_)

   Description:

   public; The first overload will run objective analysis and activate Control Agent if operation_model=0. The second overload will run objective analysis and output mean and standard deviation of water Sound Speed Profile.

A.15 PerfectSeabedDetector.h

Figure A-21: Class diagram of class PerfectSeabedDetector

In this file, class PerfectSeabedDetector is defined, which can simulate an ideal seabed detector and output all informations about seabed.

A.15.1 Data Members

1. `SimulatedOcean<T> * simulated_oceanPtr` — private; this is the pointer pointing to the simulated ocean.

2. `Vector<T> start` — private; It's 2-element vector containing latitude and longitude coordinates of start point.

3. `Vector<T> end` — private; It's 2-element vector containing latitude and longitude coordinates of end point.

4. T resolution — private; this is the resolution for the water-seabed interface line.

5. Vector<T> seabed_grid_r — private; this is the common horizontal axis in seabed.

6. Vector<T> seabed_speed_grid_z — private; this is vertical axis for sound speed in seabed.

7. Fortran_matrix<T> seabed_c — private; this is the 2-D sound speed profile in seabed.

8. Vector<T> seabed_density_grid_z — private; this is the vertical axis for density in seabed.

9. Fortran_matrix<T> seabed_density — private; this is the 2-D density profile in seabed.

10. Vector<T> seabed_attn_grid_z — private; this is the vertical axis for density in seabed.

11. Fortran_matrix<T> seabed_attn — private; this is the 2-D attenuation coefficients profile in seabed.

12. Vector<T> rb — private; this is the horizontal coordinates of grid points on the water-seabed interface line.

13. Vector<T> zb — private; this is the vertical coordinates of grid points on the water-seabed interface line.

14. int index — private; this is the index number of current PerfectSeabedDetector object.

A.15.2 Member Functions & Operators

1. Name: PerfectSeabedDetector

   Overloads:

   - PerfectSeabedDetector(void)

   Description:

   public; Constructor. Nothing is done in construction.

2. Name: Set

   Overloads:

   - void Set(SimulatedOcean<T> * simulated_oceanPtr_, const Vector<T> & start_, const Vector<T> & end_, const T & resolution_, const int & index_)
   - void Set(SimulatedOcean<T> * simulated_oceanPtr_, const Vector<T> & start_, const Vector<T> & end_, const T &resolution_, const Vector<T> & rb_, const Vector<T> & zb_, const Vector<T> & seabed_grid_r_, const Vector<T> & seabed_speed_z_, const Fortran_matrix<T> & seabed_c_, const Vector<T> & seabed_density_z_, const Fortran_matrix<T> & seabed_density_, const Vector<T> & seabed_attn_z_, const Fortran_matrix<T> & seabed_attn_, const int & index_)

   Description:

   public; The first overload can set up fundamental data members and then automatically measure all seabed environmental parameters. By the second, we can manually set up all data members including all seabed environmental parameters.

3. Name: Run

   Overloads:

   - void Run(void)

   Description:

   public; Suppose we have known all fundamental data members, this function forces detector to do measurement.

4. Name: Output_Info

   Overloads:

   - void Output_Info(Vector<T> & start_, Vector<T> & end_, T & resolution_, Vector<T> & rb_, Vector<T> & zb_, Vector<T> & seabed_grid_r_, Vector<T> &seabed_speed_z_, Fortran_matrix<T> &seabed_c_, Vector<T> &seabed_density_z_, Fortran_matrix<T> &seabed_density_, Vector<T> &seabed_attn_z_, Fortran_matrix<T> &seabed_attn_, int & index_)

- void Output_Info(Vector<T> & rb_, Vector<T> & zb_, Vector<T> & seabed_grid_r_, Vector<T> &seabed_speed_z_, Fortran_matrix<T> &seabed_c_, Vector<T> &seabed_density_z_, Fortran_matrix<T> &seabed_density_, Vector<T> &seabed_attn_z_, Fortran_matrix<T> &seabed_attn_, int & index_)

Description:
public; The first overload outputs information about location of detector and all seabed environmental parameters. The second overload is a simple version of the first overload, it only outputs all seabed environmental parameters.

A.16 ram.h

This file was created by Pierre Elisseeff and adapted by Ding Wang. Class Ram is defined in this file, by which underwater sound field can be calculated by parabolic equation method. Since this file is pretty big and has too many data members, we will only introduce public member functions here. Refer to the original file and class StandardRamInfo for more details.

A.16.1 Public Member Functions

1. Name: Ram

   Overloads:

   - Ram(void)
   - Ram(const Ram<T> &ram)
   - Ram(const T &freq_, const T &zr_, const T &zs_, const T &rmax_, const T &dr_, const int &ndr_, const T &zmax_, const T &dz_, const int &ndz_, const T &zmplt_, const T &c0_, const int &np_, const int &ns_, const T &rs_, const Vector<T> &rb_input_, const Vector<T> &zb_input_, const Vector<T> &zcw_input_, const Fortran_matrix<T> &cw_input_, const Vector<T> &zcb_input_, const Fortran_matrix<T> &cb_input_, const Vector<T> &zrhob_input_, const Fortran_matrix<T> &rhob_input_, const Vector<T> &zattn_input_, const Fortran_matrix<T> &attn_input_, const Vector<T> &rp_input_)

   Description:

   public; Constructors. The first constructor does nothing. The second constructor makes a copy of ram, which is another object of class Ram. In the third one, we can input and set up all necessary data members. Refer to class StandardRamInfo for details about input parameters.

2. Name: set

   Overloads:

- void Set(const T &freq_, const T &zr_, const T &zs_, const T &rmax_, const T &dr_, const int &ndr_, const T &zmax_, const T &dz_, const int &ndz_, const T &zmplt_, const T &c0_, const int &np_, const int &ns_, const T &rs_, const Vector<T> &rb_input_, const Vector<T> &zb_input_, const Vector<T> &zcw_input_, const Fortran_matrix<T> &cw_input_, const Vector<T> &zcb_input_, const Fortran_matrix<T> &cb_input_, const Vector<T> &zrhob_input_, const Fortran_matrix<T> &rhob_input_, const Vector<T> &zattn_input_, const Fortran_matrix<T> &attn_input_, const Vector<T> &rp_input_)
- void Set(const StandardRamInfo<T> & Standard_Ram_Info_)

Description:
public; By the first overload, we can input and set up all necessary data members. The second overload actually does the same thing as the first one, but now all necessary data members are packed in **Standard_Ram_Info_**. Refer to class StandardRamInfo for details about input parameters.

3. Name: run
   Overloads:

   - Fortran_matrix< complex<T> > run(int option)

   Description:
   public; This function returns back the whole sound field. if option=1, then verbose output mode is selected; if option=0, non-verbose output mode is selected.

A.17 Random.h

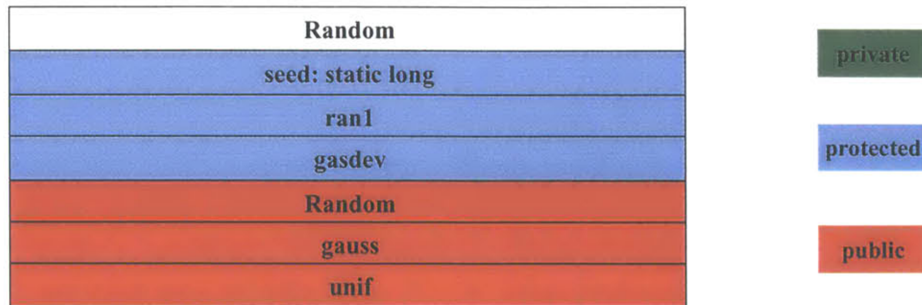


Figure A-22: Class diagram of class Random

The original file was constructed by Pierre Elisseeff or someone else and adapted by Ding Wang. In this file, class Random is defined, which can generate realizations of a gaussian random variable with zero mean, unit variance, or generate realizations of a random variable uniformly distributed in (0, 1). This file has been changed significantly, please refer to the original file for details.

A.17.1 Data Members

1. `static long seed` — protected; this is the seed for random number generation.

A.17.2 Member Functions & Operators

1. Name: `ran1`

   Overloads:

   - `float ran1( long &idum )`

   Description:

   protected; This is an internal function to generate a uniformly distributed random number. `idum` is used to pass seed.

2. Name: `gasdev`

   Overloads:

- `float gasdev( long &idum )`

Description:

protected; This is an internal function to generate a gaussian random number. `idum` is used to pass seed.

3. Name: `Random`

   Overloads:

   - `Random( void )`

   Description:

   public; Constructor.

4. Name: `gauss`

   Overloads:

   - `float gauss( void )`

   Description:

   public; This is an interface function to output a gaussian random number with zero mean, unit variance.

5. Name: `unif`

   Overloads:

   - `float unif( void )`

   Description:

   public; This is an interface function to output a random number uniformly distributed in (0, 1).

A.18 Rollout.h

Function rollout_once is defined in this file. This function will rollout heuristic algorithm such as greedy algorithm once and generate the suboptimal next sampling location.

A.18.1 Functions Defined In This File

1. Name: `rollout_once`

   Overloads:

   - `void rollout_once(const ObservationDatabase<T> * const observation_databasePtr_, const OceanPredictor<T> * const ocean_predictorPtr_, T & next_r_, T & next_z_)`

   Description:

   `observation_databasePtr_` and `ocean_predictorPtr_` point to Observation Database and Ocean Predictor. From them a virtual ocean world will be constructed and at present rollout algorithm based on greedy algorithm will be run once. The next sampling location will be output through `next_r_` and `next_z_`.

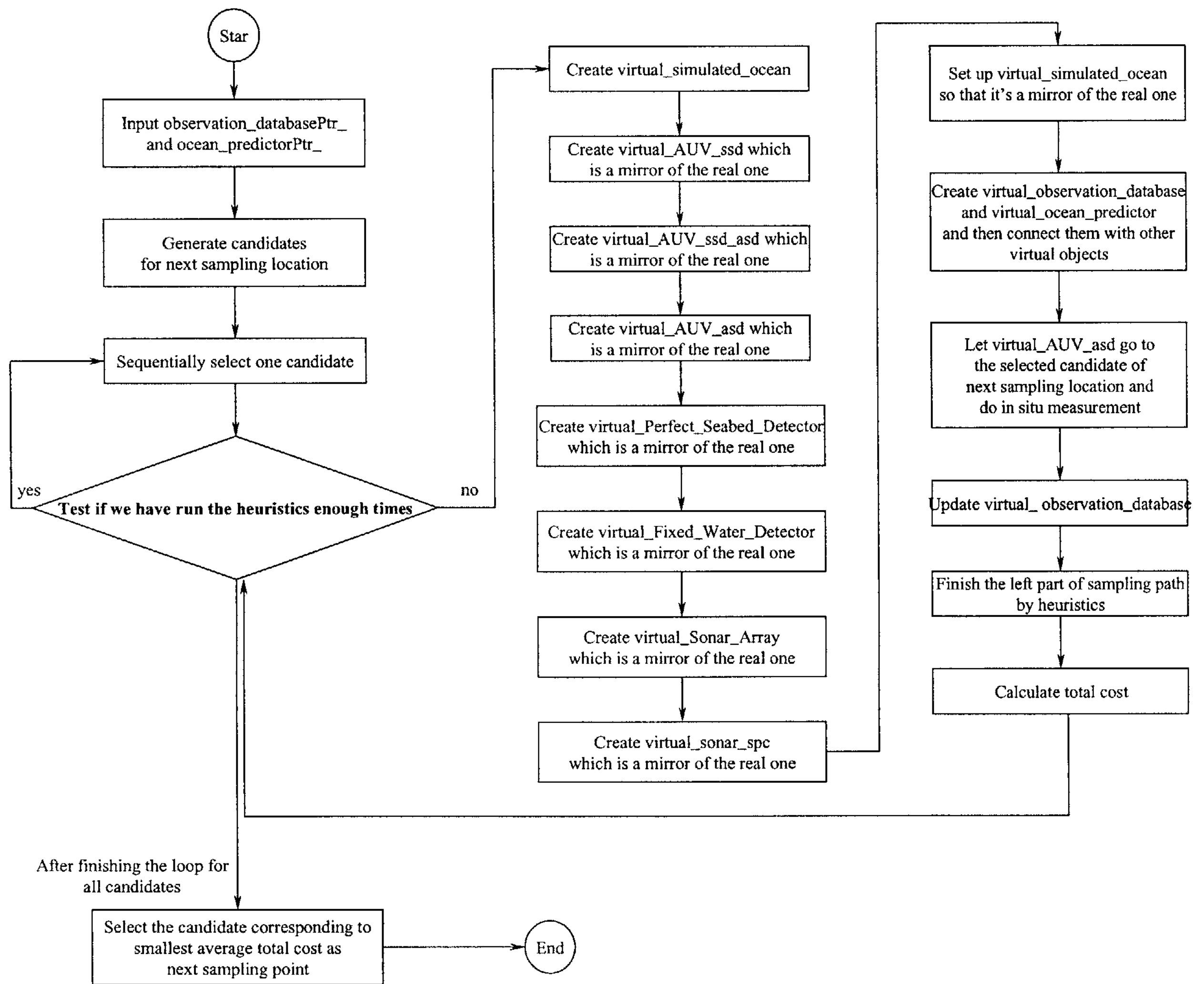


Figure A-23: Flow chart of rollout_once

A.19 Search.h

This file contains several functions used frequently.

A.19.1 Functions Defined In This File

1. Name: `sum`

   Overloads:

   - `T Sum(const Vector<T> & v_)`
   - `T Sum(const Fortran_matrix<T> & matrix_)`

   Description:

   The 2 overloads can calculate summation of all elements in a vector or a matrix respectively.

2. Name: `Search_Max`

   Overloads:

   - `void Search_Max(const Vector<T> v_, int & position_)`
   - `void Search_Max(const int total_n_, Vector<T> v_, Vector<int> & position_)`

   Description:

   In the first overload, this function can find the maximum element in vector `v_` and return back the corresponding position. In the second overload, this function can find the `total_n_` biggest elements and return back the corresponding positions.

3. Name: `Search_Min`

   Overloads:

   - `void Search_Min(const Vector<T> v_, int & position_)`

   Description:

   This function find the minimum element in vector `v_` and return back the corresponding index.

4. Name: min

   Overloads:

   - T min(const Vector<T> & x)

   Description:

   This function returns back the minimum element in vector x.

5. Name: SearchIndex

   Overloads:

   - void SearchIndex(const T & source_r_, const Vector<T> & candidate_vector_, int & upper_index_, int & lower_index_)
   - void SearchIndex(const T & source_r_, const Vector<T> & candidate_vector_, int & upper_index_, int & lower_index_, T & ratio_to_lower_index_)

   Description:

   This function will localize source_r_ in candidate_vector_ which is an ascending or descending vector and output the nearest two elements' indices. In addition, in the second overload,
   ratio: $\frac{source_r_ - candidate_vector_(lower_index_)}{candidate_vector_(upper_index_) - candidate_vector_(lower_index_)}$ will be output too. If source_r_ is just equal to an element, then upper_index_=lower_index_ and ratio_to_lower_index_=0.

6. Name: LinearInterpolation

   Overloads:

   - T LinearInterpolation(const int & r_upper_index_, const int & r_lower_index_, const T & r_ratio_to_lower_index_, const int & z_upper_index_, const int & z_lower_index_, const T & z_ratio_to_lower_index_, const Fortran_matrix<T> & water_c_temp_)
   - complex<T> LinearInterpolation(const int & r_upper_index_, const int & r_lower_index_, const T & r_ratio_to_lower_index_, const int & z_upper_index_, const int & z_lower_index_, const T & z_ratio_to_lower_index_, const Fortran_matrix<complex<T> > & sound_field_)

Description:

This function is to do 2-D linear interpolation. `r_upper_index_`, `r_lower_index_`, `r_ratio_to_lower_index_`, `z_upper_index_` define 4 adjacent points in matrix `water_c_temp_`. `r_ratio_to_lower_index_` and `z_ratio_to_lower_index_` are local coordinates of the interpolation point. The second overload is specially for complex number.

7. Name: `Update`

   Overloads:

   - `void Update(const Vector<T> & original_, Vector<T> & updated_, const int & upper_index_, const int & lower_index_, const T & ratio_to_lower_index_)`
   - `void Update(const Fortran_matrix<T> & original_, Fortran_matrix<T> & updated_, const int & upper_index_, const int & lower_index_, const T & ratio_to_lower_index_)`

   Description:

   In the first overload, we input vector `original_` and a point positioned by `upper_index_`, `lower_index_`, `ratio_to_lower_index_`. Then we construct a new vector which is a truncation of `original_` from the first element to that point but in reverse order. In the second overload, we do the same thing to a Fortran_matrix `original_` by truncating it at a vertical line whose horizontal position is determined by `upper_index_`, `lower_index_`, `ratio_to_lower_index_`.

8. Name: `Transformer`

   Overloads:

   - `void Transformer(const T & frequency_, const T & source_r_, const T & source_z_, const StandardEnvironmentInfo<T> & standard_environment_info_, const int & select_option_, StandardRamInfo<T> & Standard_Ram_Info_)`

   Description:

   `frequency_`, `source_r_`, `source_z_` and `standard_environment_info_` include all informations needed to calculate sound filed. However, usually acoustic codes always assume that source is at origin. So, this function will do coordinates transformation to `standard_environment_info_` such that the new origin will be just at the source

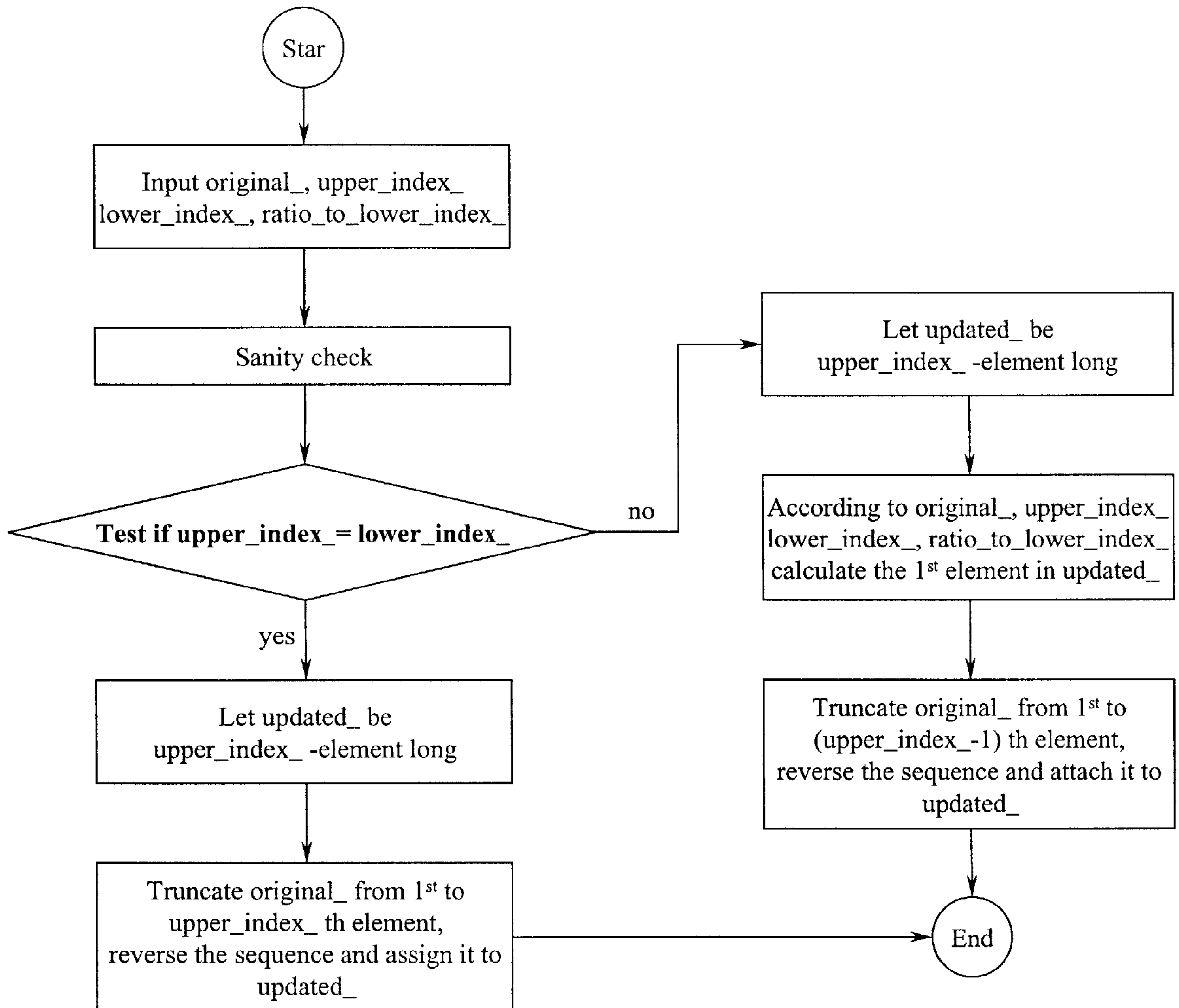


Figure A-24: Flow chart of the 1st overload of update

and then pack all informations into `Standard_Ram_Info_`. `select_option_` is used to double check if RAM is selected, so it has to be 1.

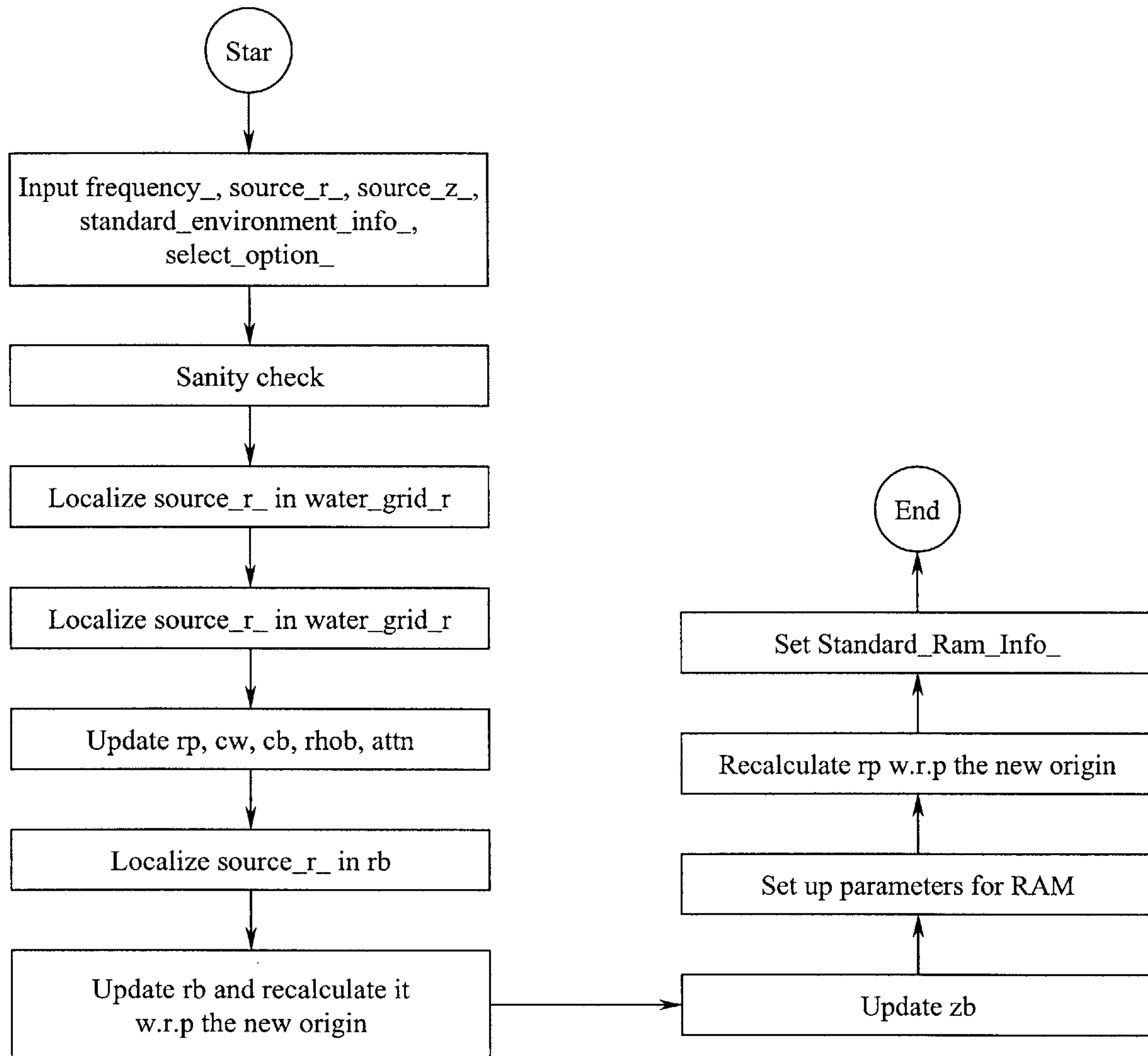


Figure A-25: Flow chart of Transformer

A.20 SimulatedOcean.h

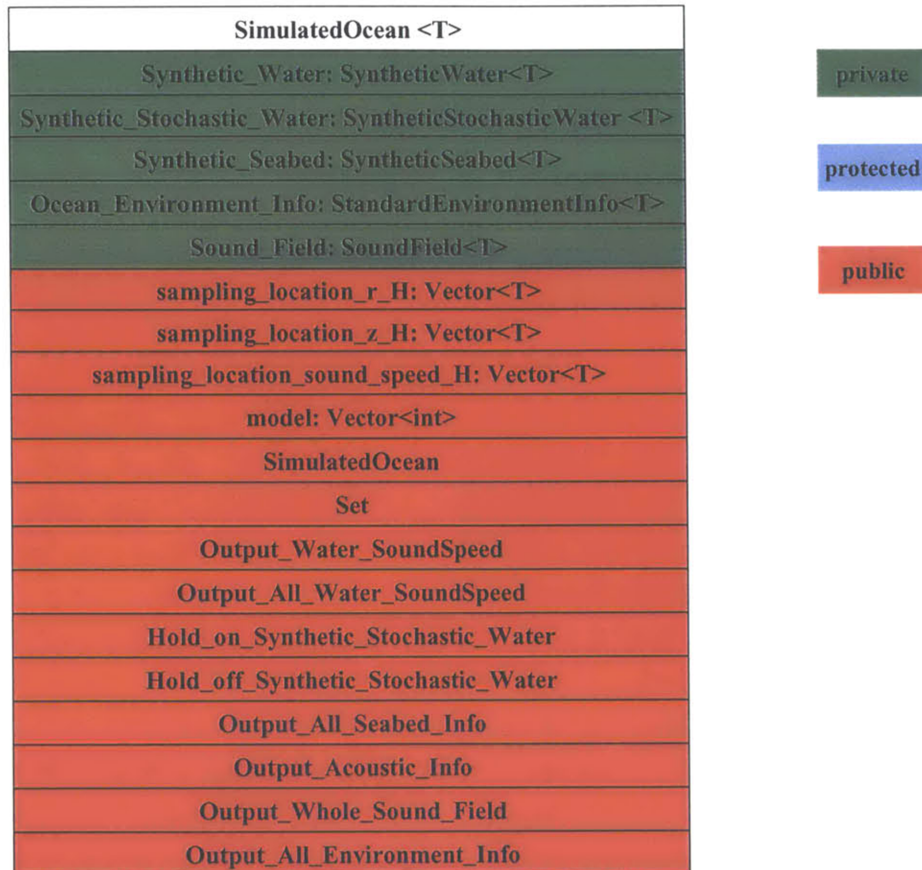


Figure A-26: Class diagram of class SimulatedOcean

Class SimulatedOcean can simulate the real ocean environment including water column, seabed and acoustic sound field. Model for water column could be certain or uncertain.

A.20.1 Data Members

1. `SyntheticWater<T> Synthetic_Water` — private; this object provides a certain model for water column.

2. `SyntheticStochasticWater <T> Synthetic_Stochastic_Water` — private; this object provides an uncertain model for water column.

3. SyntheticSeabed<T> Synthetic_Seabed — private; This object provides a seabed environment model.

4. StandardEnvironmentInfo<T> Ocean_Environment_Info — private; This object can store all water column and seabed informations. It's needed to build up Sound_Field.

5. SoundField<T> Sound_Field — private; This object can calculate acoustic sound field.

6. Vector<T> sampling_location_r_H — public; this is used to store every measurement's horizontal location.

7. Vector<T> sampling_location_z_H — public; this is used to store every measurement's vertical location.

8. Vector<T> sampling_location_sound_speed_H — public; this is used to store every sound speed measurement's result.

9. Vector<int> model — public; this is used to select certain or uncertain water column model.

A.20.2 Member Functions & Operators

1. Name: SimulatedOcean

   Overloads:

   - SimulatedOcean(void)
   - SimulatedOcean(char* oag, char* grid)

   Description:

   public; Constructors. In the first overload, we use default data files; in the second one, other data files can be input. oag is the data file for water column. grid is the data file for seabed.

2. Name: Set

   Overloads:

   - void Set(void)

- void Set(const Vector<int> model_)
- void Set(const Vector<T> & sampling_location_r_, const Vector<T> & sampling_location_z_, const Vector<T> & sampling_location_sound_speed_, const Vector<int> model_)

Description:
public; Those are initialization functions. The first overload can only be used after data member 'model' has been assigned value. It does initialization according to model selection. By the second overload, we can assign a value to model and then do initialization. By the third overload, we can replicate a measurement history, select a model and run initialization. This overload is usually used to construct a virtual ocean world which is a mirror of the simulated ocean.

3. Name: Output_Water_SoundSpeed
   Overloads:

   - T Output_Water_SoundSpeed(const Vector<T> &start_, const Vector<T> &end_, const T &r_, const T &z_)
   - T Output_Water_SoundSpeed(const T &target_r_, const T &target_z_)
   - Vector<T> Output_Water_SoundSpeed(const Vector<T> &target_r_, const Vector<T> &target_z_)

   Description:
   public; r_ and z_ are location coordinates of sampling point(s). This function outputs corresponding sound speeds. In the 1st overload, we need to input start point's and end point's latitude and longitude and it only applies to a single sampling point situation. In the 2nd and 3rd overloads the default start point and end point will be used. The 2nd overload only applies to a single sampling point. The 3rd one can output several points together.

4. Name: Output_All_Water_SoundSpeed
   Overloads:

   - Fortran_matrix<T> Output_All_Water_SoundSpeed(const Vector<T> &start, const Vector<T> &end, const Vector<T> &ri, const Vector<T> &zi)

- Fortran_matrix<T> Output_All_Water_SoundSpeed(const Vector<T> &ri, const Vector<T> &zi)

Description:
public; This function can output water sound speed at all points of the grid defined by vector ri and zi. In the first overload , we need to input start point's and end point's latitude and longitude. In the second overload, the default start point and end point will be used.

5. Name: Hold_on_Synthetic_Stochastic_Water
Overloads:

   - void Hold_on_Synthetic_Stochastic_Water(void)

   Description:
   public; This function let Synthetic_Stochastic_Water hold on.

6. Name: Hold_off_Synthetic_Stochastic_Water
Overloads:

   - void Hold_off_Synthetic_Stochastic_Water(void)

   Description:
   public; This function let Synthetic_Stochastic_Water hold off.

7. Name: Output_All_Seabed_Info
Overloads:

   - void Output_All_Seabed_Info(const Vector<T> & start_location_, const Vector<T> & end_location_, const T & bathymetry_resolution_, Vector<T> & seabed_grid_r_, Vector<T> &seabed_speed_grid_z_, Fortran_matrix<T> &seabed_c_, Vector<T> &seabed_density_grid_z_, Fortran_matrix<T> &seabed_density_, Vector<T> &seabed_attn_grid_z_, Fortran_matrix<T> &seabed_attn_, Vector<T> & rb_, Vector<T> & zb_)
   - void Output_All_Seabed_Info(Vector<T> & seabed_grid_r_, Vector<T> &seabed_speed_grid_z_, Fortran_matrix<T> &seabed_c_, Vector<T>

&seabed_density_grid_z_, Fortran_matrix<T> &seabed_density_, Vector<T> &seabed_attn_grid_z_, Fortran_matrix<T> &seabed_attn_, Vector<T> & rb_, Vector<T> & zb_)

Description:
public; In the first overload, we need to input start point's and end point's coordinates and bathymetry resolution, the function can output all seabed information in the vertical plane defined by start point and end point. In the second overload, default start, end points and bathymetry resolution will be used.

8. Name: Output_Acoustic_Info

   Overloads:

   - void Output_Acoustic_Info(const Vector<T> & sonar_or_auv_array_location_r_, const Vector<T> & sonar_or_auv_array_location_z_, Vector<complex<T> > & sonar_or_auv_array_signal_)
   - void Output_Acoustic_Info(const T & sonar_or_auv_array_location_r_, const T & sonar_or_auv_array_location_z_, complex<T> & sonar_or_auv_array_signal_)

   Description:
   public; sonar_or_auv_array_location_r_ and sonar_or_auv_array_location_z_ are hydrophone's location. In the first overload, we can input several hydrophones' locations simultaneously and output acoustic signals at those hydrophones. In the second overload, we can only do that for a single hydrophone. Note that sound source's properties are defined in AREA.cpp.

9. Name: Output_Whole_Sound_Field

   Overloads:

   - void Output_Whole_Sound_Field(const Vector<T> & grid_r_, const Vector<T> & grid_z_, Fortran_matrix<complex<T> > & Whole_Sound_Field_const_)

   Description:
   public; vector grid_r_ and grid_z_ define a grid. This function outputs the whole sound field with respect to the grid. Note that sound source's properties are defined in AREA.cpp.

10. Name: `Output_All_Environment_Info`

    Overloads:

    - `void SimulatedOcean<T>::Output_All_Environment_Info(StandardEnvironmentInfo<T> & Standard_Environment_Info_)`

    Description:

    public; This function packs all information about water column and seabed and output it. Refer to the original file for more details.

A.21 SonarArray.h

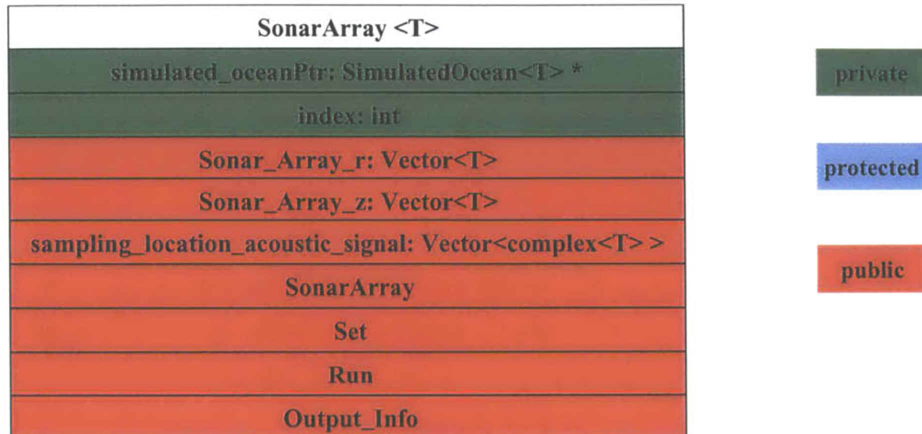


Figure A-27: Class diagram of class SonarArray

In this file, class SonarArray is defined, which can simulate position-fixed sonar array — receive sound signals and output them.

A.21.1 Data Members

1. `SimulatedOcean<T> * simulated_oceanPtr` — private; this is the pointer pointing to the simulated ocean, by which handle can be passed.

2. `int index` — private; this is the index number of current SonarArray object.

3. `Vector<T> Sonar_Array_r` — public; horizontal coordinates of all sensors.

4. `Vector<T> Sonar_Array_z` — public; vertical coordinates of all sensors.

5. `Vector<complex<T> > sampling_location_acoustic_signal` — public; acoustical signals received at all sensors.

A.21.2 Member Functions & Operators

1. Name: `SonarArray`

   Overloads:

- `SonarArray(void)`

Description:

public; Constructor. Nothing is done in construction.

2. Name: `Set`

   Overloads:

   - `void Set(SimulatedOcean<T> *simulated_oceanPtr_, const Vector<T> &Sonar_Array_r_, const Vector<T> &Sonar_Array_z_, const int & index_)`
   - `void Set(SimulatedOcean<T> *simulated_oceanPtr_, const Vector<T> &Sonar_Array_r_, const Vector<T> &Sonar_Array_z_, const Vector<complex<T> > & sampling_location_acoustic_signal_, const int & index_)`

   Description:

   public; In the first overload, we can setup all data members except `sampling_location_acoustic_signal_` which will be automatically obtained from simulated ocean. In the second overload, all data members can be set up manually. It could be used in virtual ocean world.

3. Name: `Run`

   Overloads:

   - `Vector<complex <T> > Run(void)`

   Description:

   public; This function forces sonar array receive acoustic signals again and output them.

4. Name: `Output_Info`

   Overloads:

   - `void Output_Info(Vector<T> & Sonar_Array_r_, Vector<T> & Sonar_Array_z_, Vector<complex<T> > & sampling_location_acoustic_signal_, int & index_)`

- `void Output_Info(Vector<complex<T> > & sampling_location_acoustic_signal_, int & index_)`
- `void Output_Info(Vector<T> & Sonar_Array_r_, Vector<T> & Sonar_Array_z_, int & index_)`

Description:
public; The first overload outputs all information of SonarArray. The second one only outputs acoustic signals and index number. The third one only outputs location information and index number.

A.22 Sonar_Performance.h

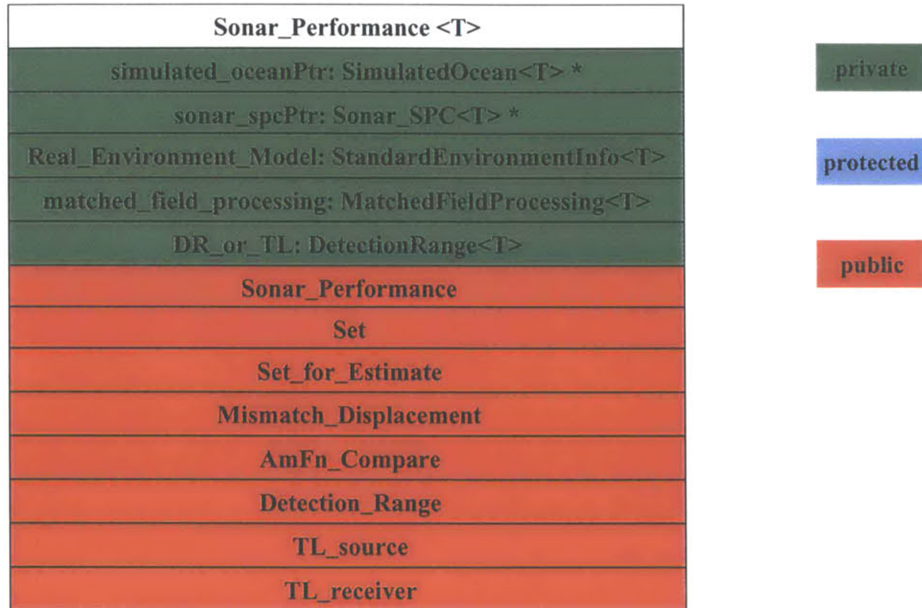


Figure A-28: Class diagram of class Sonar_Performance

In this file, class Sonar_Performance is defined. This class can analyze or predict sonar performance according to the selection of sonar performance metric.

A.22.1 Data Members

1. `SimulatedOcean<T> * simulated_oceanPtr` — private; this is the pointer pointing to the simulated ocean. `simulated_oceanPtr` is needed for extracting true simulated ocean environment. By this pointer handle can be passed.

2. `Sonar_SPC<T> * sonar_spcPtr` — private; this is the pointer pointing to the sonar signal processing center. `sonar_spcPtr` is needed for extracting sonar output. By this pointer handle can be passed.

3. `StandardEnvironmentInfo<T> Real_Environment_Model` — private; this is to store the true environment information of simulated ocean or our estimation.

4. `MatchedFieldProcessing<T> matched_field_processing` — private; By this object member, we can do matched field processing.

5. DetectionRange<T> DR_or_TL — private; By this object member, we can calculate detection range or transmission loss.

A.22.2 Member Functions & Operators

1. Name: Sonar_Performance

   Overloads:

   - Sonar_Performance(void)

   Description:

   public; Constructor. Nothing is done in construction.

2. Name: Set

   Overloads:

   - void Set(SimulatedOcean<T> * simulated_oceanPtr_, Sonar_SPC<T> * sonar_spcPtr_)

   Description:

   public; Through this function, we can setup simulated_oceanPtr and sonar_spcPtr. This function is used for analyzing sonar performance.

3. Name: Set_for_Estimate

   Overloads:

   - Set_for_Estimate(Sonar_SPC<T> * sonar_spcPtr_, const StandardEnvironmentInfo<T> & Estimated_Ocean_Environment_)

   Description:

   public; Through this function, we can setup sonar_spcPtr and input our estimation of the simulated ocean environment to Real_Environment_Model. This function is for predicting sonar performance.

4. Name: Mismatch_Displacement

   Overloads:

   - void Mismatch_Displacement(Vector<T> & main_lobe_peak_, Vector<T> & max_side_lobe_peak_, Vector<T> & source_real_location_)

Description:

public; If in sonar signal processing center Matched-Filed processing is used, this function can output locations of main lobe peak and max side lobe peak. The real location of sound source will be output too.

5. Name: `AmFn_Compare`

   Overloads:

   - `void AmFn_Compare(Fortran_matrix<T> & ambiguity_function_, Fortran_matrix<T> & real_ambiguity_function_)`

   Description:

   public; `ambiguity_function_` is the ambiguity function output from sonar. In this function another ambiguity function based on the true environment information of simulated ocean or our estimation is calculated, which is output as `real_ambiguity_function_`.

6. Name: `Detection_Range`

   Overloads:

   - `void Detection_Range(const T & dB_threshold_, T & detection_range_)`

   Description:

   public; In this function,transmission loss along a line which is defined in AREA.cpp will be calculated and according to `dB_threshold_`, detection range will be found and output.

7. Name: `TL_source`

   Overloads:

   - `void TL_source(Vector<T> & TL_)`

   Description:

   public; In this function, `TL_source` will be calculated and output. Refer to class DetectionRange for explanation for `TL_source`.

8. Name: `TL_receiver`

   Overloads:

   - `void TL_receiver(Vector<T> & TL_)`

   Description:

   public; In this function, `TL_receiver` will be calculated and output. Refer to class DetectionRange for explanation for `TL_receiver`.

A.23 Sonar_SPC.h

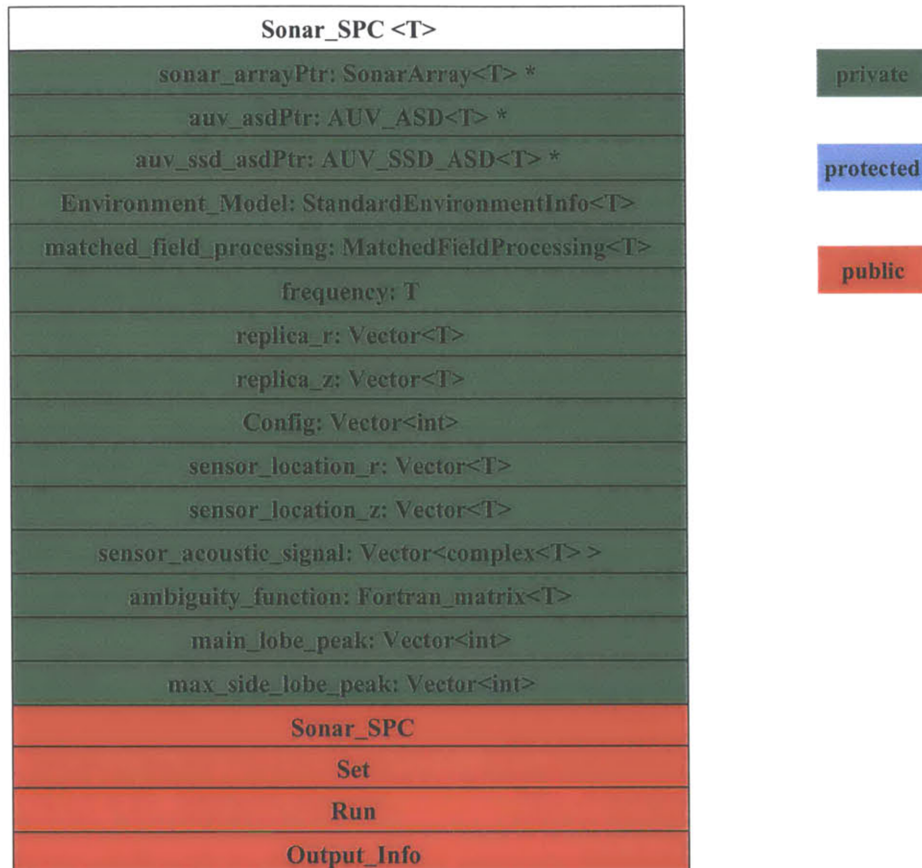


Figure A-29: Class diagram of class Sonar_SPC

In this file, class Sonar_SPC is defined. Sonar_SPC means sonar signal processing center, which is a part of sonar system. Sonar_SPC processes acoustic signals received by SonarArray. Now, our sonar system is a MFP sonar, in Sonar_SPC it does matched field processing by calling an object of class MatchedFieldProcessing.

A.23.1 Data Members

1. `SonarArray<T> * sonar_arrayPtr` — private; this is the pointer pointing to an object of class SonarArray. By this pointer handle can be passed.

2. `AUV_ASD<T> * auv_asdPtr` — private; this is the pointer pointing to an object of

class AUV_ASD. By this pointer handle can be passed.

3. AUV_SSD_ASD<T> * auv_ssd_asdPtr — private; this is the pointer pointing to an object of class AUV_SSD_ASD.

4. StandardEnvironmentInfo<T> Environment_Model — private; This data member stores the ocean acoustic environment model used in sonar system.

5. MatchedFieldProcessing<T> matched_field_processing — private; By this object member, we can run Matched-Field Processing.

6. T frequency — private; This is the central frequency of sonar system.

7. Vector<T> replica_r — private; horizontal axis of the grid for replica source locations.

8. Vector<T> replica_z — private; vertical axis of the grid for replica source locations.

9. Vector<complex<T> > sensor_acoustic_signal — private; acoustic signals received by all sensors.

10. Fortran_matrix<T> ambiguity_function — private; this matrix stores ambiguity function values at all replica source locations.

11. Vector<int> main_lobe_peak — private; This vector stores horizontal index and vertical index of main lobe peak.

12. Vector<int> max_side_lobe_peak — private; This vector stores horizontal index and vertical index of the biggest side lobe peak.

A.23.2 Member Functions & Operators

1. Name: Sonar_SPC

   Overloads:

   - Sonar_SPC(void)

   Description:

   public; Constructor. Nothing is done in construction.

2. Name: Set

   Overloads:

   - void Set(SonarArray_T> * sonar_arrayPtr_, AUV_ASD<T> * auv_asdPtr_, AUV_SSD_ASD<T> * auv_ssd_asdPtr_, const StandardEnvironmentInfo<T> & Environment_Model_, const T & frequency_, const Vector<T> & replica_r_, const Vector<T> & replica_z_, const Vector<int> & Config_)
   - void Set(SonarArray<T> * sonar_arrayPtr_, AUV_ASD<T> * auv_asdPtr_, AUV_SSD_ASD<T> * auv_ssd_asdPtr_, const StandardEnvironmentInfo<T> & Environment_Model_, const T & frequency_, const Vector<T> & replica_r_, const Vector<T> & replica_z_, const Vector<int> & Config_, const Vector<T> & sensor_location_r_, const Vector<T> & sensor_location_z_, const Vector<complex<T> > & sensor_acoustic_signal_)

   Description:
   public; In the first overload, we will input 8 data members. All the other data members will be automatically extracted and generated. In the second overload, we will input and setup all data members.

3. Name: Run

   Overloads:

   - void Run(Fortran_matrix<T> & ambiguity_function_, Vector<int> & main_lobe_peak_, Vector<int> & max_side_lobe_peak_)
   - void Run(void)

   Description:
   public; In the first overload, this function will do matched-field processing and output all results. In the second overload, this function will do matched-field processing and store all results in data members.

4. Name: Output_Info

   Overloads:

   - void Output_Info(StandardEnvironmentInfo<T> & Environment_Model_, T & frequency_, Vector<T> & replica_r_, Vector<T> & replica_z_, Vector<int>

& Config_, Vector<T> & sensor_location_r_, Vector<T> & sensor_location_z_, Vector<complex<T> > & sensor_acoustic_signal_, Fortran_matrix<T> & ambiguity_function_, Vector<int> & main_lobe_peak_, Vector<int> & max_side_lobe_peak_)

- void Output_Info(Fortran_matrix<T> & ambiguity_function_, Vector<int> & main_lobe_peak_, Vector<int> & max_side_lobe_peak_)
- void Output_Info(StandardEnvironmentInfo<T> & Environment_Model_, T & frequency_, Vector<T> & replica_r_, Vector<T> & replica_z_, Vector<int> & Config_, Vector<T> & sensor_location_r_, Vector<T> & sensor_location_z_, Vector<complex<T> > & sensor_acoustic_signal_)

Description:
public; In the first overload, all information will be output. In the second overload, results from Matched-Filed processing will be output. In the third one, all information except Matched-Field processing results will be output.

A.24 SoundField.h

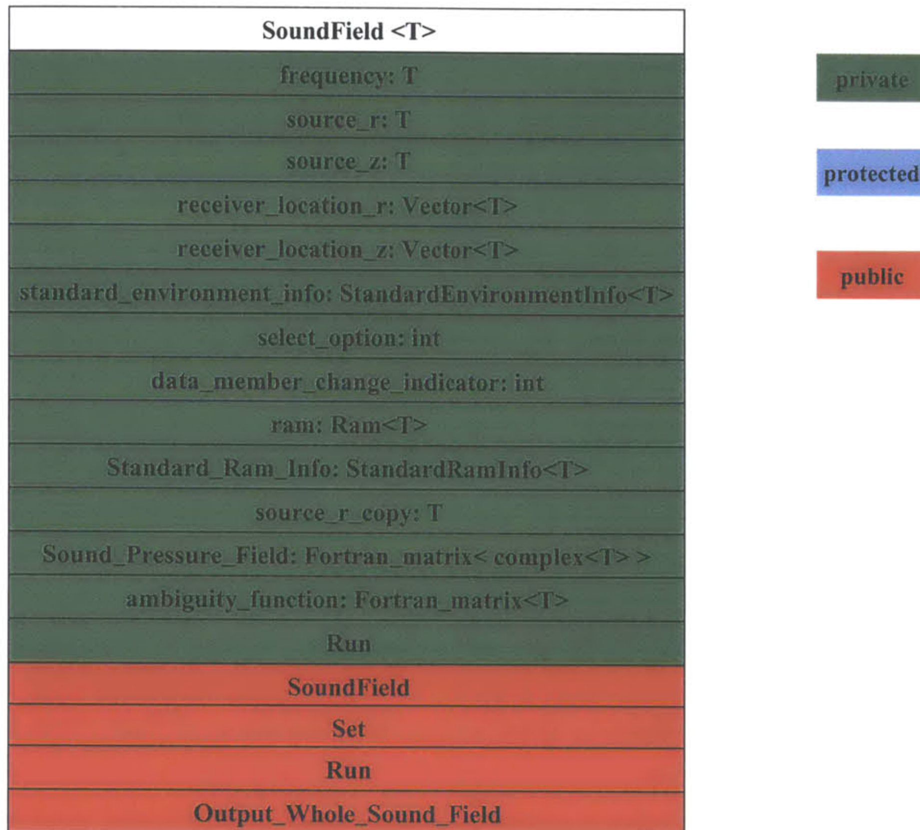


Figure A-30: Class diagram of class SoundField

Class SoundField is defined in this file, which can generate the whole sound pressure field according to request. RAM and SEALAB could be used in this class.

A.24.1 Data Members

1. `T frequency` — private; sound frequency.
2. `T source_r` — private; this is source horizontal coordinates.
3. `T source_z` — private; this is source vertical coordinates.
4. `Vector<T> receiver_location_r` — private; These are receivers' horizontal locations.

5. `Vector<T> receiver_location_z` — private; These are receivers' vertical locations.

6. `StandardEnvironmentInfo<T> standard_environment_info` — private; this object stores all water column and seabed environmental parameters.

7. `int select_option` — private; if 1, select RAM; if 2, select SEALAB.

8. `int data_member_change_indicator` — private; this variable indicates if data members get changed or not. If 1, then some data members has been changed; if 0, no data member has been changed. If data members get changed, then some codes will be run again.

9. `Ram<T> ram` — private; This object may be used to calculate sound field.

10. `StandardRamInfo<T> Standard_Ram_Info` — private; this object stores all inputs needed by class Ram.

11. `T source_r_copy` — private; this is a copy of `source_r`.

12. `Fortran_matrix< complex<T> > Sound_Pressure_Field` — private; this matrix will store sound pressure field.

A.24.2 Member Functions & Operators

1. Name: `SoundField`

   Overloads:

   - `SoundField(void)`

   Description:

   public; Constructor.

2. Name: `Set`

   Overloads:

   - `void Set(const T & frequency_, const T & source_r_, const T & source_z_, const StandardEnvironmentInfo<T> & standard_environment_info_, const int & select_option_)`

- void Set(const T & frequency_, const T & source_r_, const T & source_z_, const Vector<T> & receiver_location_r_, const Vector<T> & receiver_location_z_, const StandardEnvironmentInfo<T> & standard_environment_info_, const int & select_option_)
- void Set(const Vector<T> & receiver_location_r_, const Vector<T> & receiver_location_z_)

Description:
public; In the first overload, we set up source's parameters and environmental parameters and model selection. In the second overload, we set up source's parameters, receiver's parameters and environmental parameters and model selection. In the third overload, we only set up receivers' location.

3. Name: Run
   Overloads:

   - void Run(void)
   - void Run(const Vector<T> & receiver_location_r_, const Vector<T> & receiver_location_z_, Vector<complex<T> > & signal_at_receiver_)
   - void Run(const T & receiver_location_r_, const T & receiver_location_z_, complex<T> & signal_at_receiver_)
   - void Run(Vector< complex<T> > & signal_at_receiver_)

   Description:
   The first overload is private, which is an internal function which will call ram or sealab and generate sound pressure field. Other overloads are public. In the second one, we can input several receivers' coordinates and get back corresponding acoustic signals. The third one is almost the same as the second one, but only applies to a single receiver. In the fourth one, we suppose receiver(s)' location has been input; all corresponding acoustic signal(s) will be output.

4. Name: Output_Whole_Sound_Field
   Overloads:

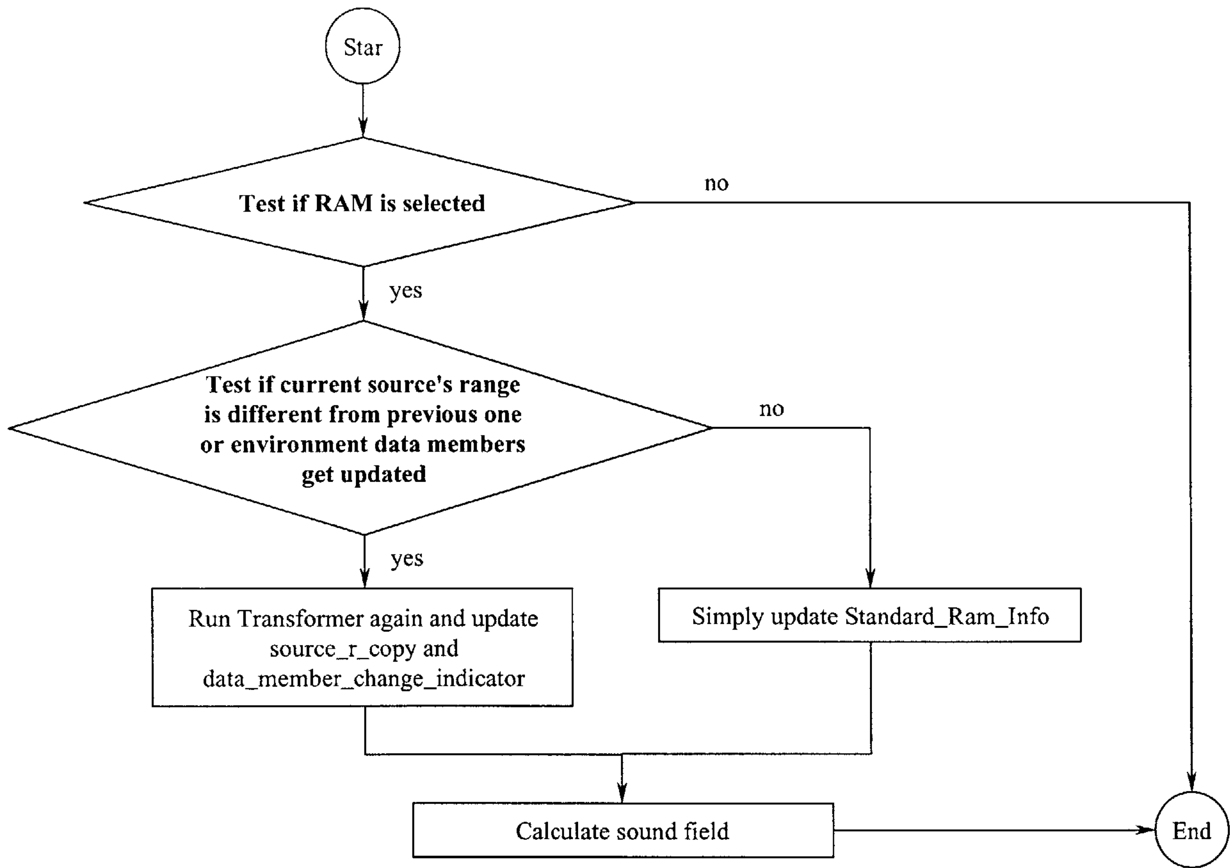


Figure A-31: Flow chart of the 1st overload of Run

- `void Output_Whole_Sound_Field(const Vector<T> & grid_r_, const Vector<T> & grid_z_, Fortran_matrix<complex<T> > & Whole_Sound_Field_)`

Description:

public; vector `grid_r_` and `grid_z_` define a grid. This function output the whole sound field with respect to the grid.

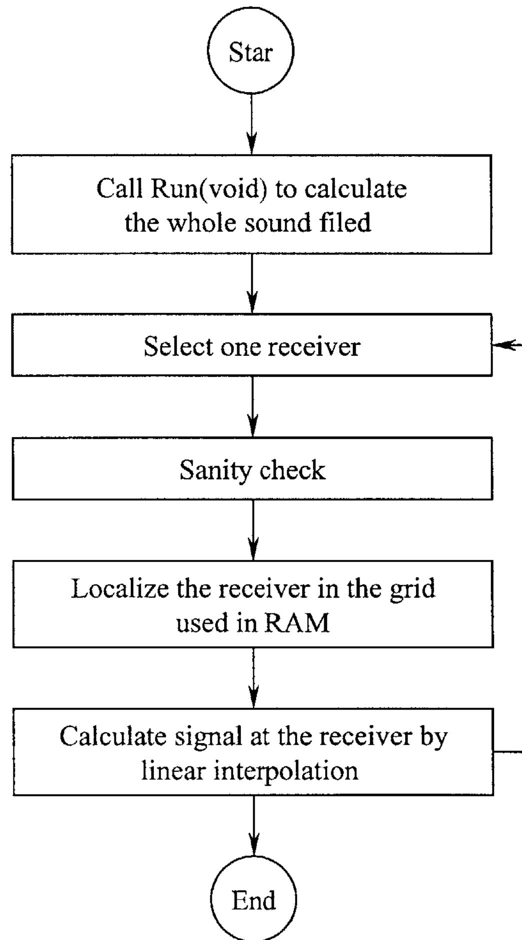


Figure A-32: Flow chart of the 4th overload of Run

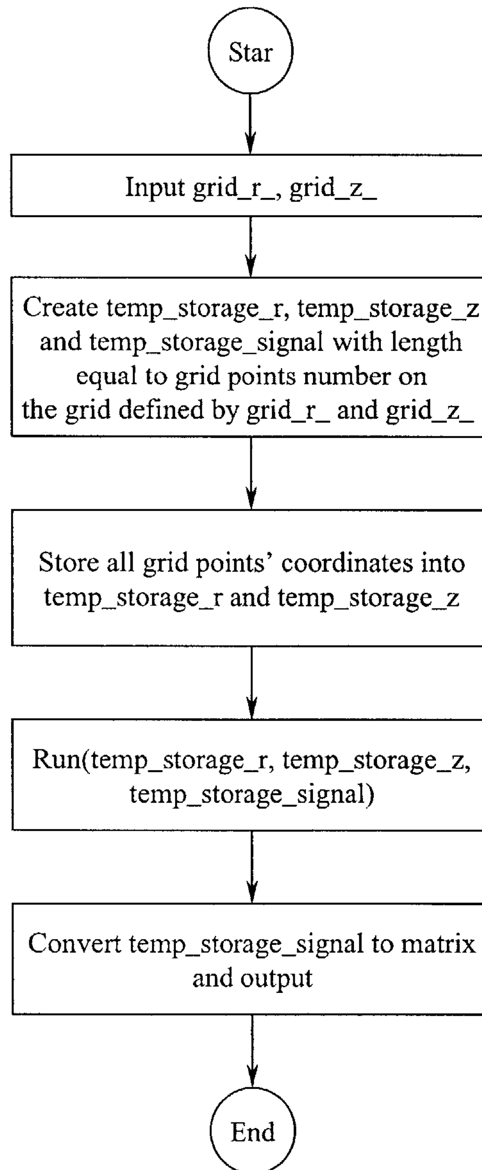


Figure A-33: Flow chart of Output_Whole_Sound_Field

A.25 SoundSpeedGenerator.h

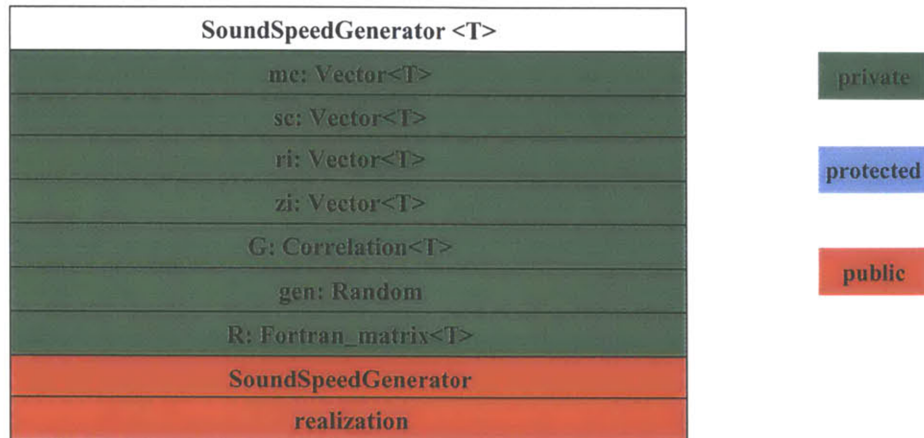


Figure A-34: Class diagram of class SoundSpeedGenerator

This file was created by Pierre Elisseeff. Ding Wang made some adaption. By inputting mean and error field of sound speed profile, a grid and correlation function, class SoundSpeedGenerator can output a realization of the whole sound speed profile according to Gaussian distribution.

A.25.1 Data Members

1. `Vector<T> mc` — private; this is the mean field of sound speed profile.
2. `Vector<T> sc` — private; this is the error field of sound speed profile.
3. `Vector<T> ri` — private; this is horizontal axis of a grid.
4. `Vector<T> zi` — private; this is vertical axis of a grid.
5. `Correlation<T> G` — private; This is the correlation function which describes horizontal and vertical correlation in sound speed profile.
6. `Random gen` — private; this is a random number generator.

A.25.2 Member Functions & Operators

1. Name: `SoundSpeedGenerator`

   Overloads:

   - `SoundSpeedGenerator( void )`
   - `SoundSpeedGenerator( const Fortran_matrix<T> &mc_, const Fortran_matrix<T> &sc_, const Vector<T> &ri_, const Vector<T> &zi_, const Correlation<T> &G_ )`
   - `SoundSpeedGenerator( const Vector<T> &mc_, const Vector<T> &sc_, const Vector<T> &ri_, const Vector<T> &zi_, const Correlation<T> &G_ )`

   Description:
   public; Constructor. The first overload is dummy, which will return back warning information. In programme, the first overload can't be used. In the second and third overload, we will input and set up all data members. In the second one, mean and error are in matrix format; however in the third one, they are in vector format. Refer to the original file for more details.

2. Name: `realization`

   Overloads:

   - `Fortran_matrix<T> realization ( void )`
   - `T realization (const T & r_, const T & z_)`
   - `Vector<T> realization (const Vector<T> & r_, const Vector<T> & z_)`

   Description:
   public; The first overload will generate the whole field and output them all. In the second and third overloads, a realization of the whole field will be generated, but only one or some points will be output respectively. `r_` and `z_` are coordinates of point(s) for output.

A.26 StandardEnvironmentInfo.h

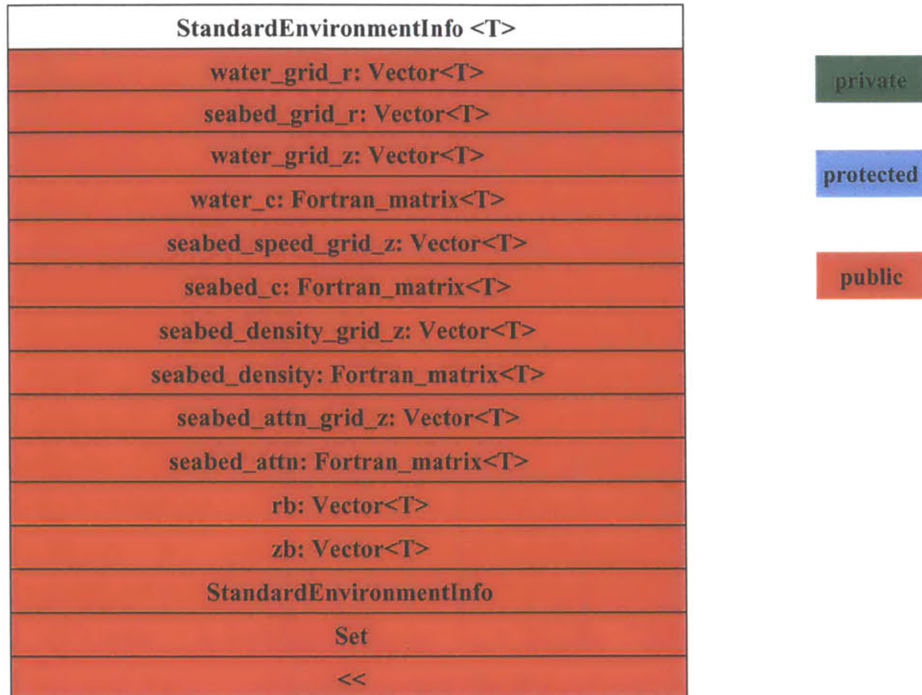


Figure A-35: Class diagram of class StandardEnvironmentInfo

Class StandardEnvironmentInfo is used to store water column's and seabed's physical parameters.

A.26.1 Data Members

1. `Vector<T> water_grid_r` — public; this is the horizontal axis in water column.

2. `Vector<T> seabed_grid_r` — public; this is the common horizontal axis in seabed.

3. `Vector<T> water_grid_z` — public; this is the vertical axis for water sound speed profile.

4. `Fortran_matrix<T> water_c` — public; this is 2-D water sound speed profile.

5. `Vector<T> seabed_speed_grid_z` — public; this is the vertical axis for seabed sound speed profile.

6. Fortran_matrix<T> seabed_c — public; this is the seabed sound speed profile.

7. Vector<T> seabed_density_grid_z — public; this is the vertical axis for seabed density profile.

8. Fortran_matrix<T> seabed_density — public; this is the seabed density profile.

9. Vector<T> seabed_attn_grid_z — public; this is the vertical axis for seabed attenuation profile.

10. Fortran_matrix<T> seabed_attn — public; this is the seabed attenuation profile.

A.26.2 Member Functions & Operators

1. Name: StandardEnvironmentInfo

   Overloads:

   - StandardEnvironmentInfo( )

   Description:

   public; Constructor. Nothing is done in construction.

2. Name: set

   Overloads:

   - void Set(const Vector<T> & water_grid_r_, const Vector<T> & seabed_grid_r_, const Vector<T> &water_grid_z_, const Fortran_matrix<T> &water_c_, const Vector<T> &seabed_speed_grid_z_, const Fortran_matrix<T> &seabed_c_, const Vector<T> &seabed_density_grid_z_, const Fortran_matrix<T> &seabed_density_, const Vector<T> &seabed_attn_grid_z_, const Fortran_matrix<T> &seabed_attn_, const Vector<T> &rb_, const Vector<T> &zb_)

   Description:

   public; By this function we can set up all data members.

3. Name: <<

   Overloads:

- `ostream& operator<<(ostream &s, const StandardEnvironmentInfo<T> & Standard_Environment_Info_)`

Description:

public; Through this operator we can output all data members by one command.

A.27 StandardRamInfo.h

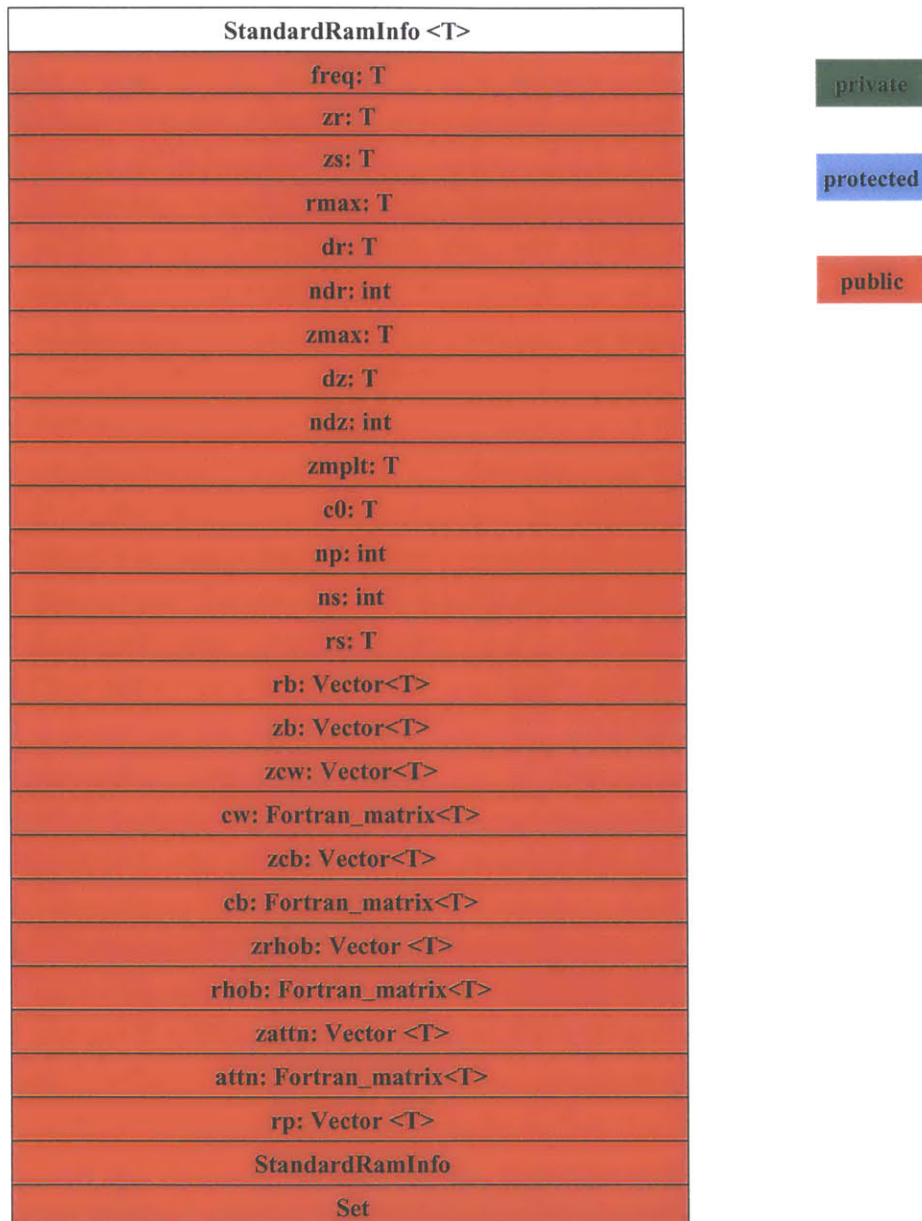


Figure A-36: Class diagram of class StandardRamInfo

Class StandardRamInfo is used to store all parameters needed by class Ram.

A.27.1 Data Members

1. T freq — public; sound frequency (Hz).
2. T zr — public; receiver depth, which is dummy but needed by Ram.
3. T zs — public; source depth (m).
4. T rmax — public; maximum range of computation grid (m).
5. T dr — public; horizontal step size.
6. int ndr — public; range axis decimation factor for output purposes.
7. T zmax — public; maximum depth of computation grid (m).
8. T dz — public; vertical step size.
9. int ndz — public; depth axis decimation factor for output purposes.
10. T zmplt — public; maximum depth of output grid (m).
11. T c0 — reference sound speed (m/s).
12. int np — number of Pade coefficients.
13. int ns — number of constraints.
14. T rs — stability range (m).
15. Vector<T> rb — bathymetry range axis (m).
16. Vector<T> zb — bathymetry depth axis (m).
17. Vector<T> zcw — water sound speed field depth axis (m).
18. Fortran_matrix<T> cw — water sound speed field (m/s).
19. Vector<T> zcb — bottom sound speed field depth axis (m).
20. Fortran_matrix<T> cb — bottom sound speed field (m/s).
21. Vector<T> zrhob — bottom density field depth axis (m).
22. Fortran_matrix<T> rhob — bottom density field (g/cm3).

23. `Vector<T> zattn` — bottom attenuation field depth axis (m).

24. `Fortran_matrix<T> attn` — bottom attenuation field (dB/lambda).

25. `Vector<T> rp` — range axis common to cw, cb, rhob and attn (m).

A.27.2 Member Functions & Operators

1. Name: `StandardRamInfo`

   Overloads:

   - `StandardRamInfo(void)`

   Description:

   public; Constructor. Nothing is done in construction.

2. Name: `set`

   Overloads:

   - `void Set(const T & freq_, const T & zr_, const T & zs_, const T & rmax_, const T & dr_, const int & ndr_, const T & zmax_, const T & dz_, const int & ndz_, const T & zmplt_, const T & c0_, const int & np_, const int & ns_, const T & rs_, const Vector<T> & rb_, const Vector<T> & zb_, const Vector<T> & zcw_, const Fortran_matrix<T> & cw_, const Vector<T> & zcb_, const Fortran_matrix<T> & cb_, const Vector<T> & zrhob_, const Fortran_matrix<T> & rhob_, const Vector<T> & zattn_, const Fortran_matrix<T> & attn_, const Vector<T> & rp_)`

   Description:

   public; By this function we can input and set up all data members.

A.27.3 Functions And Operators Defined In This File

1. Name: <<

   Overloads:

   - `ostream& operator<<(ostream &s, const StandardRamInfo<T> & Standard_Ram_Info_)`

Description:

public; Through this operator we can output all data members by one command.

A.28 Surveillance.h

See Figure 2-6.

Function Surveillant is defined in this file, by which we can extract history records from some objects and generate all results that we need.

A.28.1 Functions Defined In This File

1. Name: Surveillance

   Overloads:

   - void Surveillance(const int sonar_model_selection_,
     const int output_model_selection_,
     const ObservationDatabase<T> & Observation_Database_,
     const OceanPredictor<T> & Ocean_Predictor_)

   Description:
   This function is usually used in the end of the whole programme. History records in Observation_Database_ and Ocean_Predictor_ will be extracted and output. According to sonar_model_selection_ and output_model_selection_, different sonar performance metric will be selected and Monte Carlo simulations will be run in different manner. The result of sonar performance metric realizations will be output. In addition, some global variables will be output too. Refer to the original code for more details.

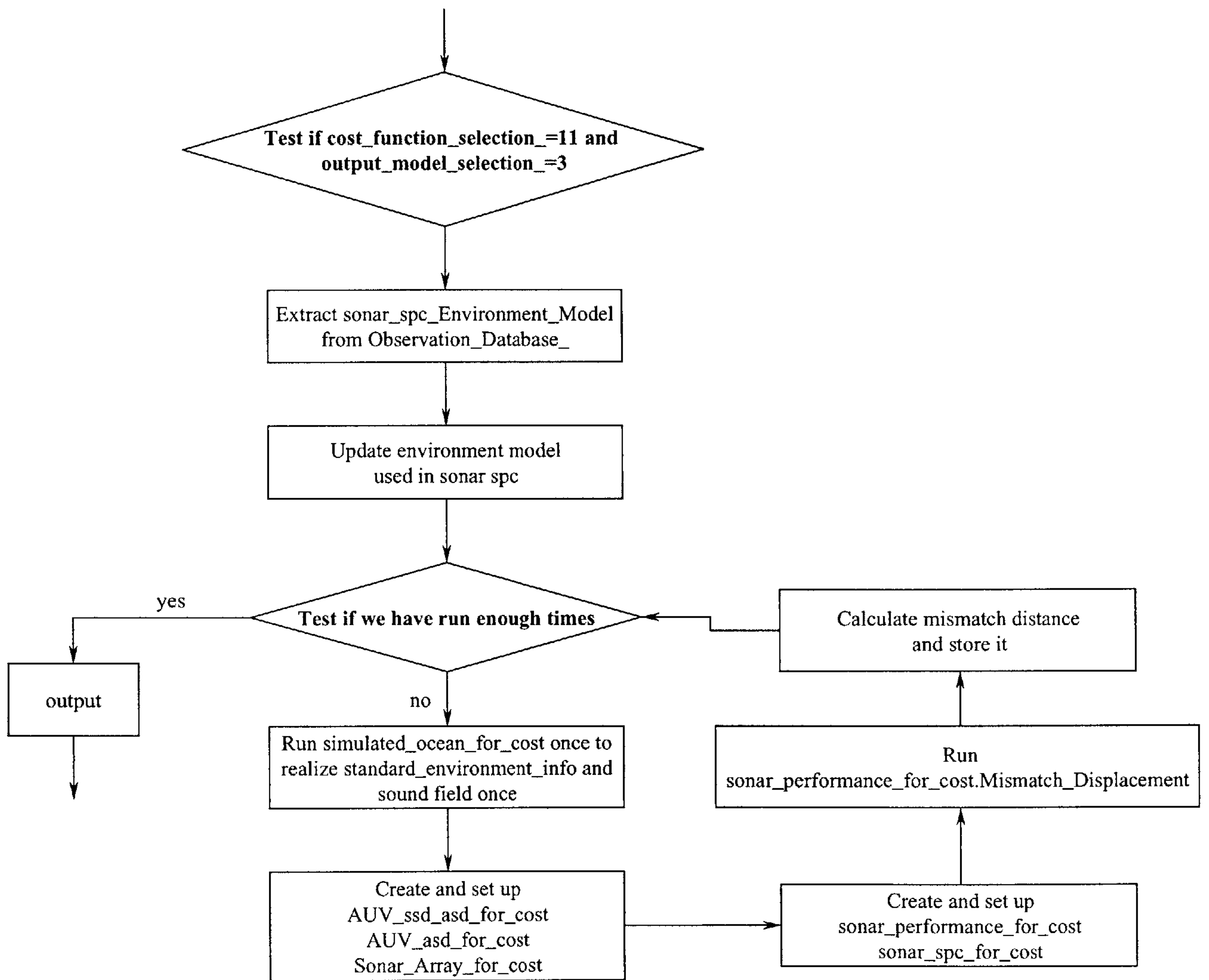


Figure A-37: Partial flow chart of Surveillance — Calculate mismatch displacement

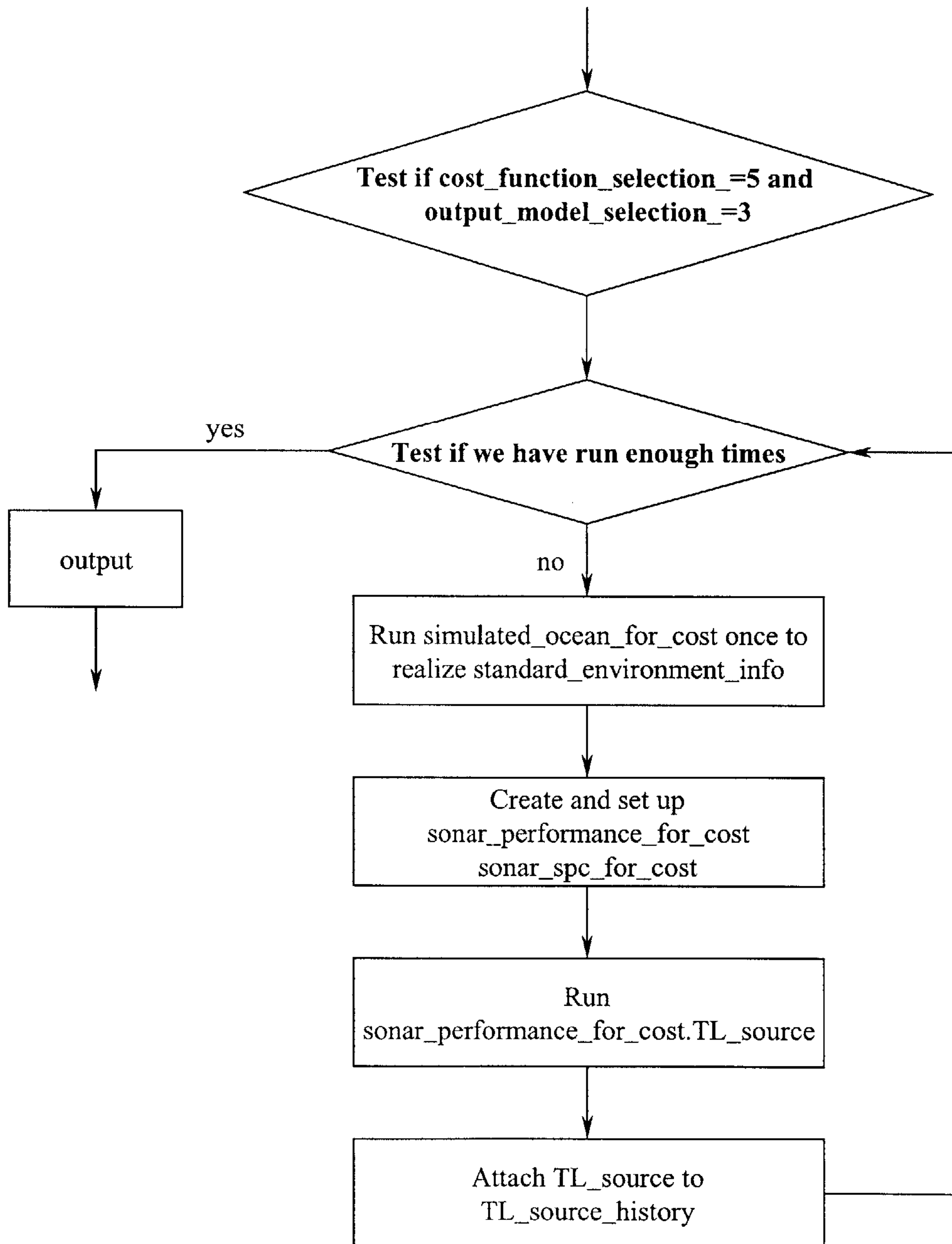


Figure A-38: Partial flow chart of Surveillance — Calculate TL_source

A.29 SyntheticSeabed.h

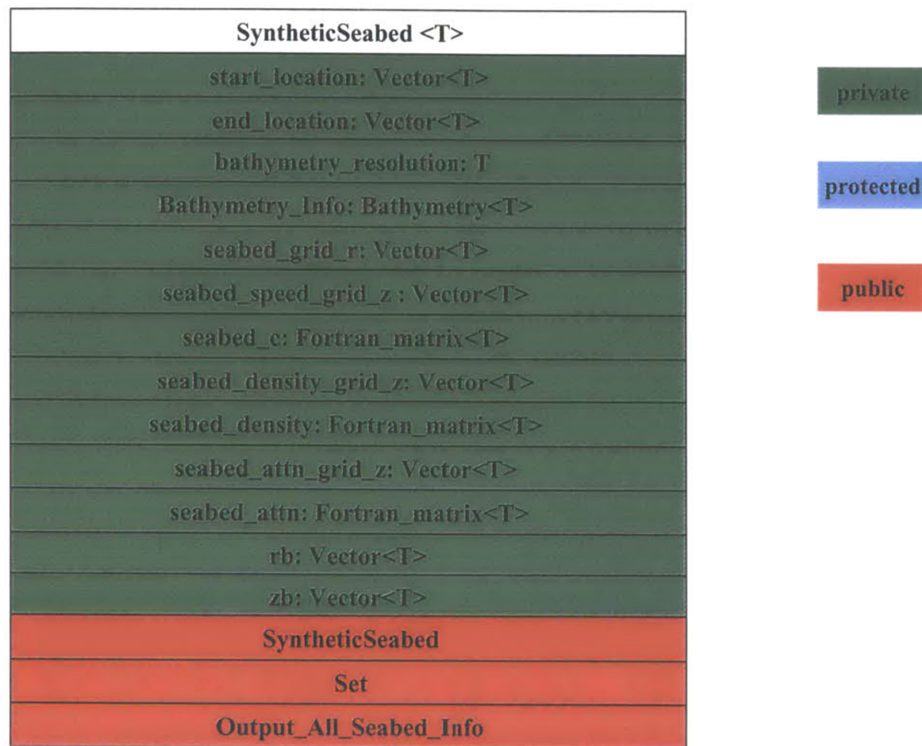


Figure A-39: Class diagram of class SyntheticSeabed

In this file, class SyntheticSeabed is defined, which can simulate the whole seabed acoustic environment and output environmental parameters according to requirements.

A.29.1 Data Members

1. `Vector<T> start_location` — private; this vector has 2 elements - latitude and longitude of start point.

2. `Vector<T> end_location` — private; this vector has 2 elements - latitude and longitude of end point.

3. `T bathymetry_resolution` — private; this is the resolution of water-seabed interface line.

4. Bathymetry<T> Bathymetry_Info — private; this object can query a bathymetry HOPS data file (netcdf format), and output the water-seabed interface line.

5. Vector<T> **seabed_grid_r** — private; this is common horizontal axis in seabed.

6. Vector<T> **seabed_speed_grid_z** — private; this is vertical axis for sound speed in seabed.

7. Fortran_matrix<T> **seabed_c** — private; this is the 2-D sound speed profile in seabed.

8. Vector<T> **seabed_density_grid_z** — private; this is the vertical axis for density in seabed.

9. Fortran_matrix<T> **seabed_density** — private; this is the 2-D density profile in seabed.

10. Vector<T> **seabed_attn_grid_z** — private; this is the vertical axis for density in seabed.

11. Fortran_matrix<T> **seabed_attn** — private; this is the 2-D attenuation coefficients profile in seabed.

12. Vector<T> **rb** — private; this is the horizontal coordinates of points on the water-seabed interface line.

13. Vector<T> **zb** — private; this is the vertical coordinates of points on the water-seabed interface line.

A.29.2 Member Functions & Operators

1. Name: SyntheticSeabed

   Overloads:

   - SyntheticSeabed(void)
   - SyntheticSeabed(char * grid)

Description:

public; this is the constructor function. the 1st overload uses default seabed database; the 2nd overload uses the database — 'grid'.

2. Name: Set

   Overloads:

   - void Set(const Vector<T> & seabed_grid_r_,
     const Vector<T> & seabed_speed_grid_z_, const Fortran_matrix<T> & seabed_c_,
     const Vector<T> & seabed_density_grid_z_,
     const Fortran_matrix<T> & seabed_density_,
     const Vector<T> & seabed_attn_grid_z_, const Fortran_matrix<T> & seabed_attn_,
     const Vector<T> & rb_, const Vector<T> & zb_)
   - void Set(const Vector<T> & seabed_grid_r_,
     const Vector<T> & seabed_speed_grid_z_,
     const Vector<T> & seabed_density_grid_z_,
     const Vector<T> & seabed_attn_grid_z_)
   - void Set(const Vector<T> & start_location_, const Vector<T> & end_location_,
     const T & bathymetry_resolution_, const Vector<T> & seabed_grid_r_,
     const Vector<T> & seabed_speed_grid_z_,
     const Vector<T> & seabed_density_grid_z_,
     const Vector<T> & seabed_attn_grid_z_)

   Description:

   public; Through the 1st overload, all seabed environmental parameters will be input and set up; in the 2nd overload, only the 4 grids are needed, corresponding sound speed profile, density profile, attenuation coefficient profile will be generated by built-in algorithm; in the 3rd overload, the 4 grids, start and end points' location and water-seabed interface line resolution are needed, sound speed profile, density profile, attenuation coefficient profile will be generated by built-in algorithm and water-seabed interface line will be extracted from database.

3. Name: Output_All_Seabed_Info

   Overloads:

- void Output_All_Seabed_Info(Vector<T> & seabed_grid_r_, Vector<T> & seabed_speed_grid_z_, Fortran_matrix<T> & seabed_c_, Vector<T> & seabed_density_grid_z_, Fortran_matrix<T> & seabed_density_, Vector<T> & seabed_attn_grid_z_, Fortran_matrix<T> & seabed_attn_, Vector<T> & rb_, Vector<T> & zb_)
- void Output_All_Seabed_Info(const Vector<T> & start_location_, const Vector<T> & end_location_, const T & bathymetry_resolution_, Vector<T> & seabed_grid_r_, Vector<T> & seabed_speed_grid_z_, Fortran_matrix<T> & seabed_c_, Vector<T> & seabed_density_grid_z_, Fortran_matrix<T> & seabed_density_, Vector<T> & seabed_attn_grid_z_, Fortran_matrix<T> & seabed_attn_, Vector<T> & rb_, Vector<T> & zb_)

Description:
public; By this function, we can output all seabed environmental parameters. In the 1st overload, the interface line is between start and end point that were input from function Set; this overload can not follow the 2nd Set overload. In the 2nd overload, we can input new start and end point; the interface line is between the new points. Refer to the original file for more details.

A.30 SyntheticStochasticWater.h

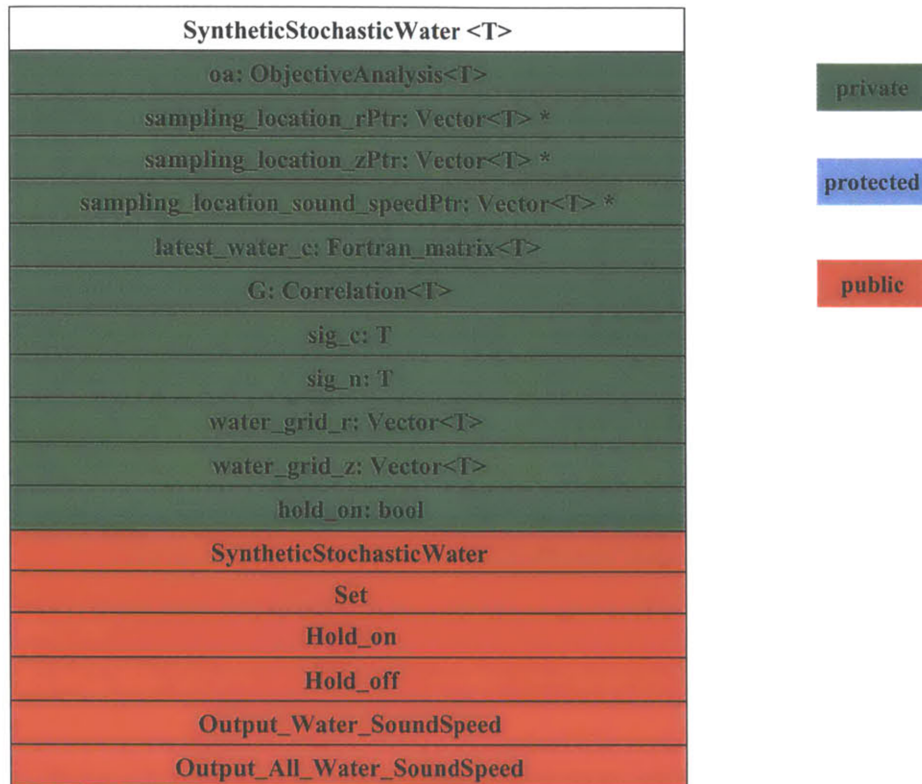


Figure A-40: Class diagram of class SyntheticStochasticWater

In this file, class SyntheticStochasticWater is defined, which simulates a stochastic ocean water acoustic environment by using ObjectiveAnalysis and SoundSpeedGenerator.

A.30.1 Data Members

1. `ObjectiveAnalysis<T> oa` — private; this object works as an sound speed profile estimator.

2. `Vector<T> * sampling_location_rPtr` — private; this is a pointer pointing to a vector containing sampling locations' horizontal coordinates.

3. `Vector<T> * sampling_location_zPtr` — private; this is a pointer pointing to a vector containing sampling locations' vertical coordinates.

4. Vector<T> * sampling_location_sound_speedPtr — private; this is a pointer pointing to a vector containing sound speed values at sampling locations.

5. Fortran_matrix<T> latest_water_c — private; this matrix contains the latest sound speed profile estimation.

6. Correlation<T> G — private; this is the correlation function.

7. T sig_c — private; a priori sound speed profile standard deviation.

8. T sig_n — private; a priori noise sound speed profile standard deviation.

9. Vector<T> water_grid_r — private; horizontal axis in water column.

10. Vector<T> water_grid_z — private; vertical axis in water column.

11. bool hold_on — private; if it's equal to 1, latest_water_c can't be refreshed; if it's equal to 0, latest_water_c can be refreshed.

A.30.2 Member Functions & Operators

1. Name: SyntheticStochasticWater

   Overloads:

   - SyntheticStochasticWater(void)

   Description:

   public; This is the constructor function. It sets hold_on=0.

2. Name: Set

   Overloads:

   - void Set(Vector<T> * sampling_location_rPtr_,
     Vector<T> * sampling_location_zPtr_,
     Vector<T> * sampling_location_sound_speedPtr_, const Correlation<T>
     & G_, const T & sig_c_, const T & sig_n_, const Vector<T> & water_grid_r_,
     const Vector<T> & water_grid_z_)

   Description:

   public; Through this function, all private data members will be input and set up.

3. Name: Hold_on

   Overloads:

   - void Hold_on(void)

   Description:

   public; This function sets hold_on=1.

4. Name: Hold_off

   Overloads:

   - void Hold_off(void)

   Description:

   public; This function sets hold_on=0.

5. Name: Output_Water_SoundSpeed

   Overloads:

   - T Output_Water_SoundSpeed(const T &target_r_, const T &target_z_)
   - Vector<T> Output_Water_SoundSpeed(const Vector<T> &target_r_, const Vector<T> &target_z_)

   Description:

   public; By inputting location coordinates of sampling points, this function outputs corresponding sound speeds. The 1st overload only applies to a single sampling point.

6. Name: Output_All_Water_SoundSpeed

   Overloads:

   - Fortran_matrix<T> Output_All_Water_SoundSpeed(const Vector<T> &start_, const Vector<T> &end_, const Vector<T> &ri_, const Vector<T> &zi_)
   - Fortran_matrix<T> Output_All_Water_SoundSpeed(const Vector<T> &ri_, const Vector<T> &zi_)

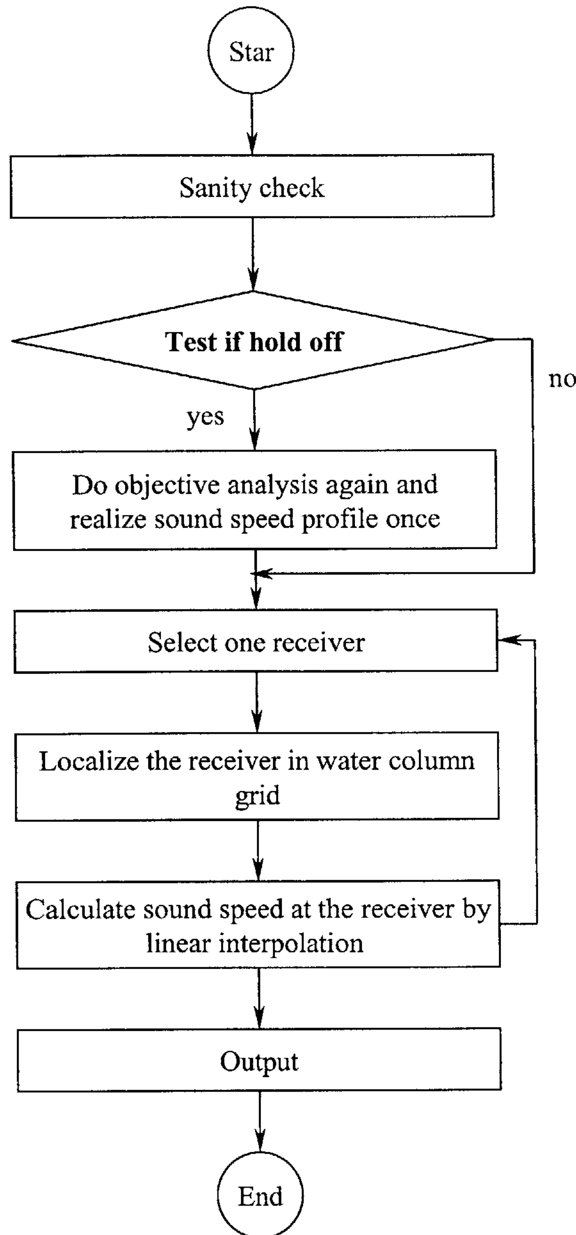


Figure A-41: Flow chart of the 2nd overload of Output_Water_SoundSpeed

Description:

public; This is a dummy function, the only objective is to make code in SimulatedOcean.h simple and consistent. Since in stochastic model, we can not know exactly the whole sound speed profile but mean and variance, so actually this function only returns back a realization of the whole sound speed profile with respect to horizontal axis `ri_` an vertical axis `zi_`. In the 1st overload, global latitude and longitude of start

and end points are dummy, but they are needed to make code in SimulatedOcean.h simple and consistent.

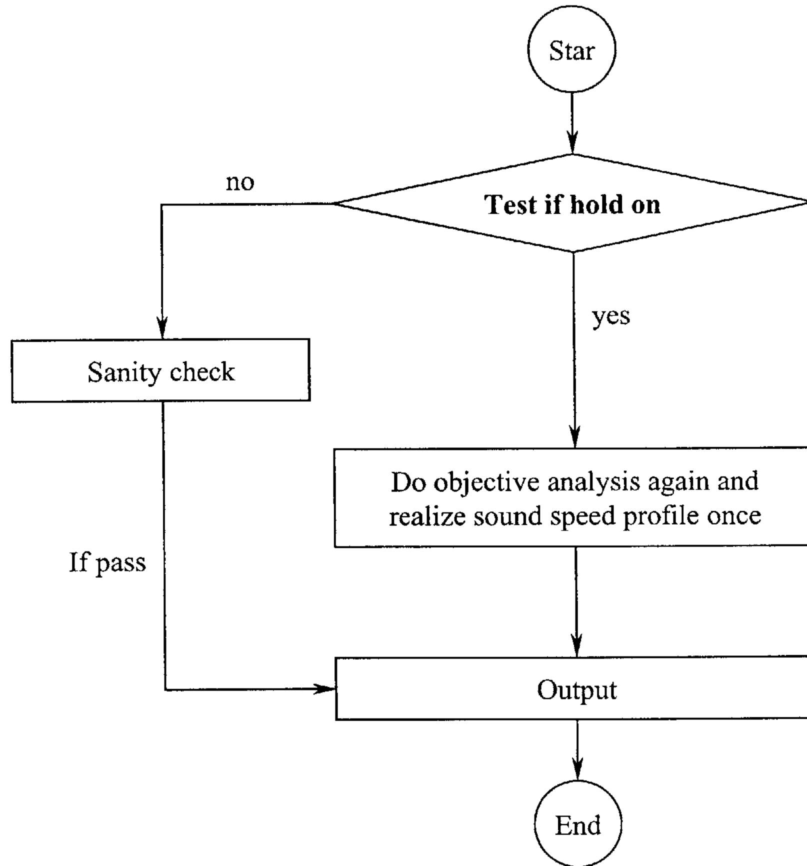


Figure A-42: Flow chart of the 2nd overload of Output_All_Water_SoundSpeed

A.31 SyntheticWater.h

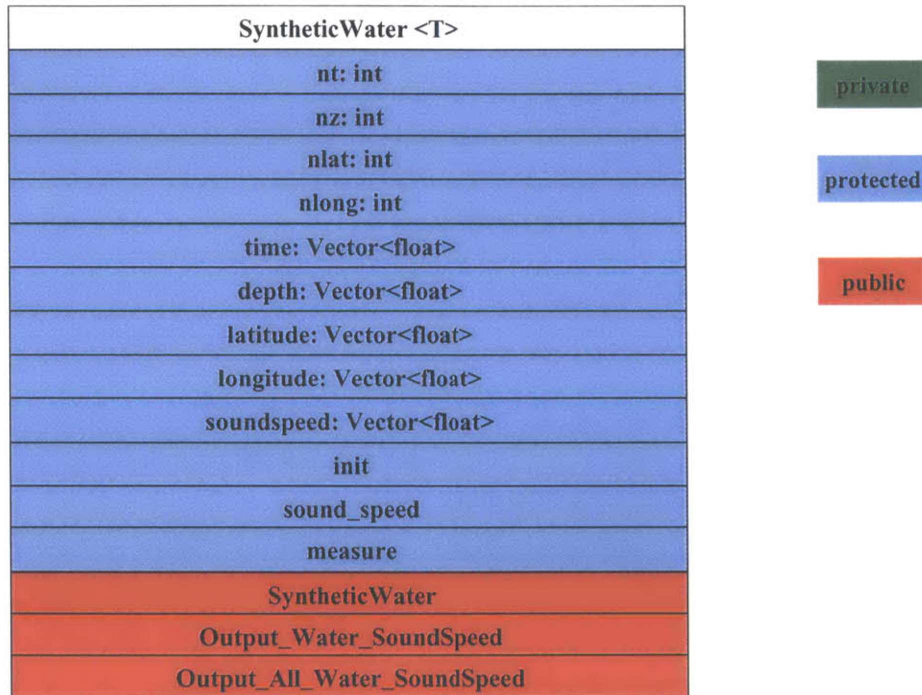


Figure A-43: Class diagram of class SyntheticWater

Class SyntheticWater was created by Pierre Elisseeff and adapted by Ding Wang. This class can simulate the real ocean water column using a HOPS OAG data file (netcdf format).

A.31.1 Data Members

1. `int nt` — protected; this is the length of vector time.
2. `int nz` — protected; this is the length of vector depth.
3. `int nlat` — protected; this is the length of vector latitude.
4. `int nlong` — protected; this is the length of vector longitude.
5. `Vector<float> time` — protected; this vector stores time axis.
6. `Vector<float> depth` — protected; this vector stores depth axis.

7. `Vector<float> latitude` — protected; this vector stores latitude axis.

8. `Vector<float> longitude` — protected; this vector stores longitude axis.

9. `Vector<float> soundspeed` — protected; this vector stores sound speed values at all grid points in the 4-D space defined by the above 4 axes.

A.31.2 Member Functions & Operators

1. Name: `init`

   Overloads:

   - `void init( char* oag_file_name)`

   Description:

   protected; This is an internal initialization helper function, which will setup all data members. It will open the data file: `oag_file_name`; upload the 4 axes and temperature and salinity information; and then compute sound speed values for the 4-D grid.

2. Name: `sound_speed`

   Overloads:

   - `T sound_speed(T s, T t, T d)`

   Description:

   protected; This function returns back the sound speed (m/sec) given values of salinity (ppt), temperature (deg C) and depth (m) using the formula of Mackenzie.

3. Name: `measure`

   Overloads:

   - `T measure( const Vector<T> &start, T x, T y, T z )`

   Description:

   protected; In this function, 2-D vector start is input as the new origin; x, y, z are local coordinates.

4. Name: SyntheticWater

   Overloads:

   - SyntheticWater(void)
   - SyntheticWater(char* o)

   Description:

   public; The first constructor uses default file: oag_AAcoustic_R.nc as data file; In the second constructor we can input another data file.

5. Name: Output_All_Water_SoundSpeed

   Overloads:

   - Fortran_matrix<T> Output_All_Water_SoundSpeed(const Vector<T> &start, const Vector<T> &end, const Vector<T> &ri, const Vector<T> &zi)

   Description:

   public; 2-D vector start and end are global coordinates of start and end points respectively. They define a vertical plane. Vector ri and zi define a grid on that plane. This function returns back sound speeds at all points on the grid.

6. Name: Output_Water_SoundSpeed

   Overloads:

   - Vector<T> Output_Water_SoundSpeed(const Vector<T> &start, const Vector<T> &end, const Vector<T> &r, const Vector<T> &z)

   Description:

   public; 2-D Vector start and end are global coordinates of start and end points respectively. They define a vertical plane. Vector ri and zi define a series of points. This function returns back sound speeds at those points.

A.32 Total_cost.h

This file contains several different cost calculating functions, by which we can obtain different total cost according to cost function selection.

A.32.1 Functions Defined In This File

1. Name: `total_cost_TL_receiver`

   Overloads:

   - `void total_cost_TL_receiver(SimulatedOcean<T> * virtual_simulated_oceanPtr_, Sonar_SPC<T> * virtual_sonar_spcPtr_, T & total_cost_)`

   Description:

   In this function, a simulated ocean and a sonar signal processing center are input. Based on them, sum of variance of `TL_receiver` will be calculated by Monte Carlo simulation and returned back as total cost. Refer to class DetectionRange for explanation for `TL_receiver`.

2. Name: `total_cost_TL_source`

   Overloads:

   - `void total_cost_TL_source(SimulatedOcean<T> * virtual_simulated_oceanPtr_, Sonar_SPC<T> * virtual_sonar_spcPtr_, T & total_cost_)`

   Description:

   In this function, a simulated ocean and a sonar signal processing center are input. Based on them, sum of variance of `TL_source` will be calculated by Monte Carlo simulation and returned back as total cost. Refer to class DetectionRange for explanation for `TL_source`.

3. Name: `total_cost_oceanography`

   Overloads:

   - `void total_cost_oceanography(SimulatedOcean<T> * virtual_simulated_oceanPtr_, T & total_cost_)`

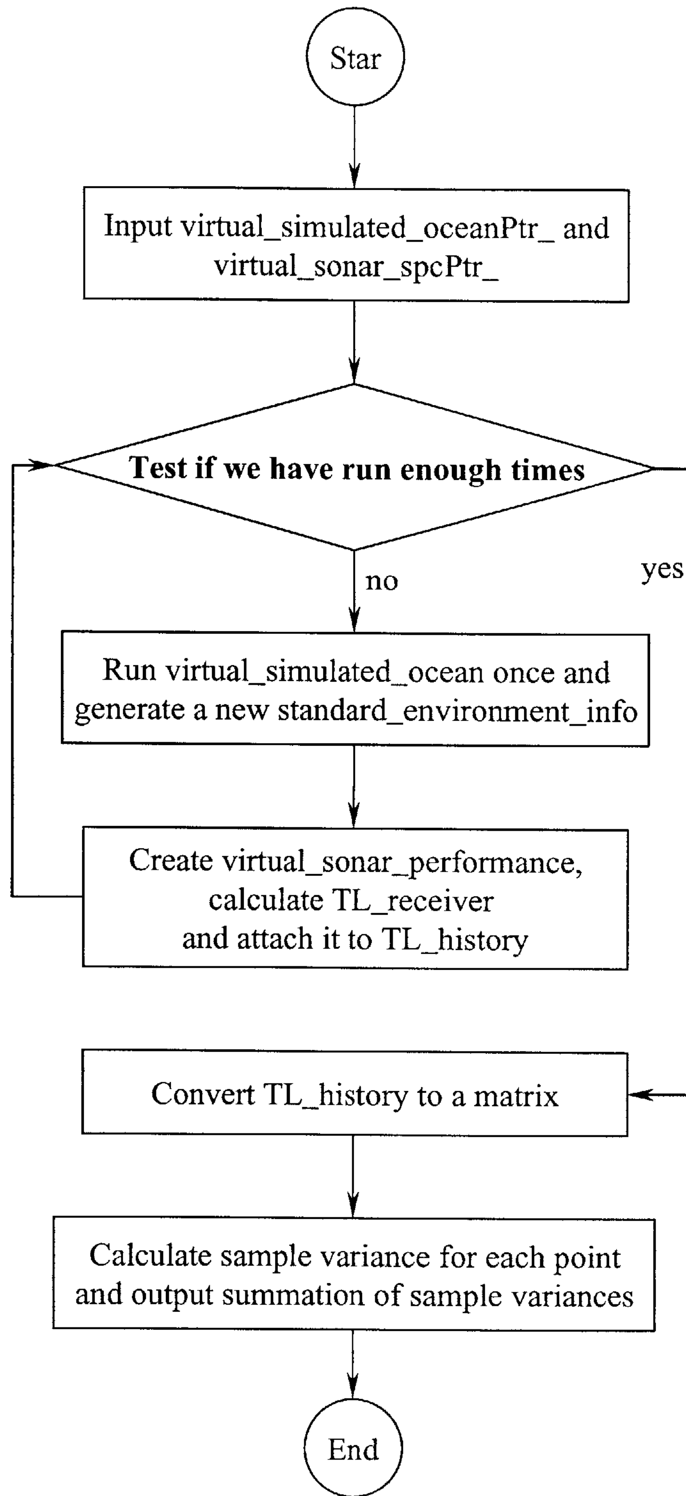


Figure A-44: Flow chart of total_cost_TL_receiver

Description:

In this function, a simulated ocean is input. Based on it, sum of variance of sound speed in water column will be calculated by Monte Carlo simulations and returned back as total cost.

4. Name: `total_cost_c_std`

   Overloads:

   - `void total_cost_c_std(SimulatedOcean<T> * virtual_simulated_oceanPtr_, const Fortran_matrix<T> & err_, T & total_cost_)`

   Description:

   This function is similar to function `total_cost_oceanography`; but here we directly input `err_` — standard deviations of sound speed in water column provided by ObjectiveAnalysis, to calculate total cost.

5. Name: `points_filter`

   Overloads:

   - `void points_filter( const StandardEnvironmentInfo<T> & standard_environment_info_, const Fortran_matrix<T> & water_c_matrix_, Vector<T> & water_c_history_vector_)`

   Description:

   According to water-seabed interface information contained in `standard_environment_info_`, this function picks out points that are really in water column from `water_c_matrix_` and store those points in `water_c_history_vector_`.

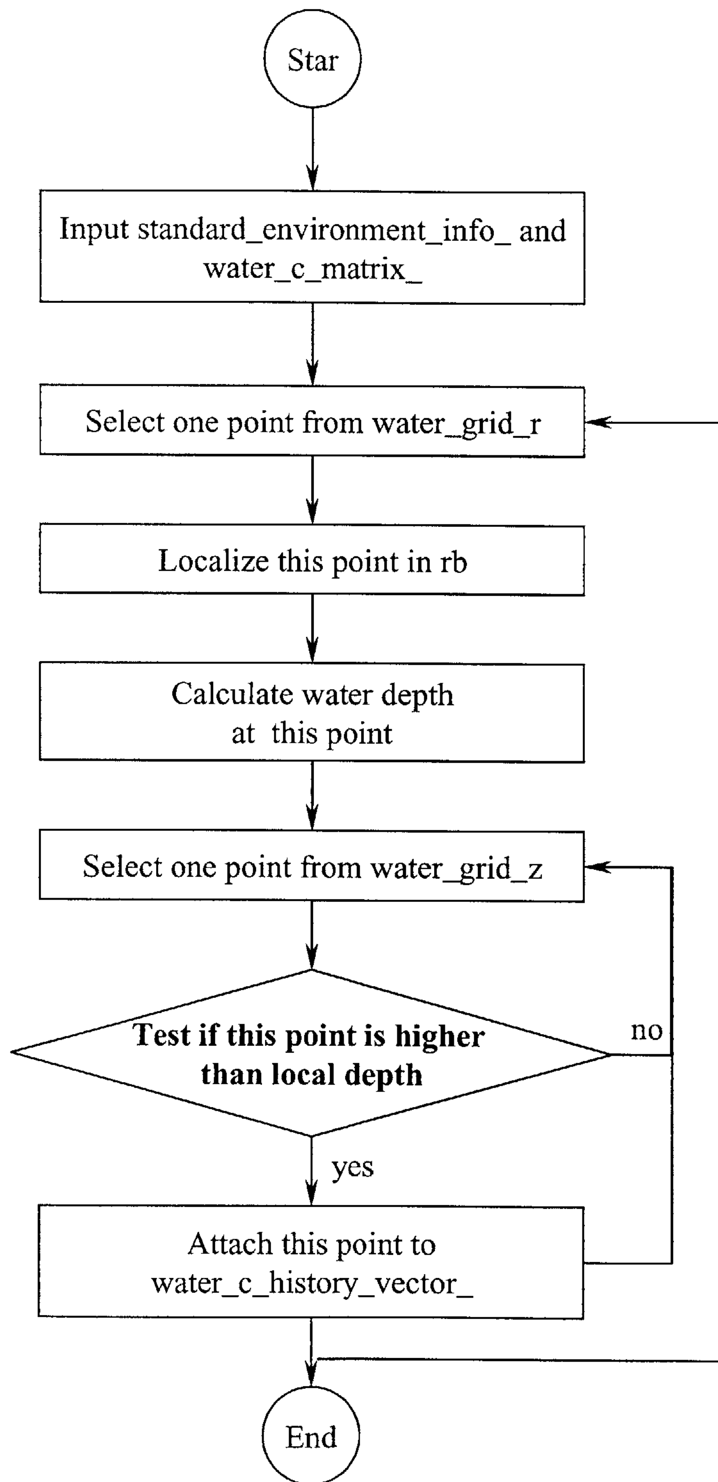


Figure A-45: Flow chart of points_filter

A.33 vec.h

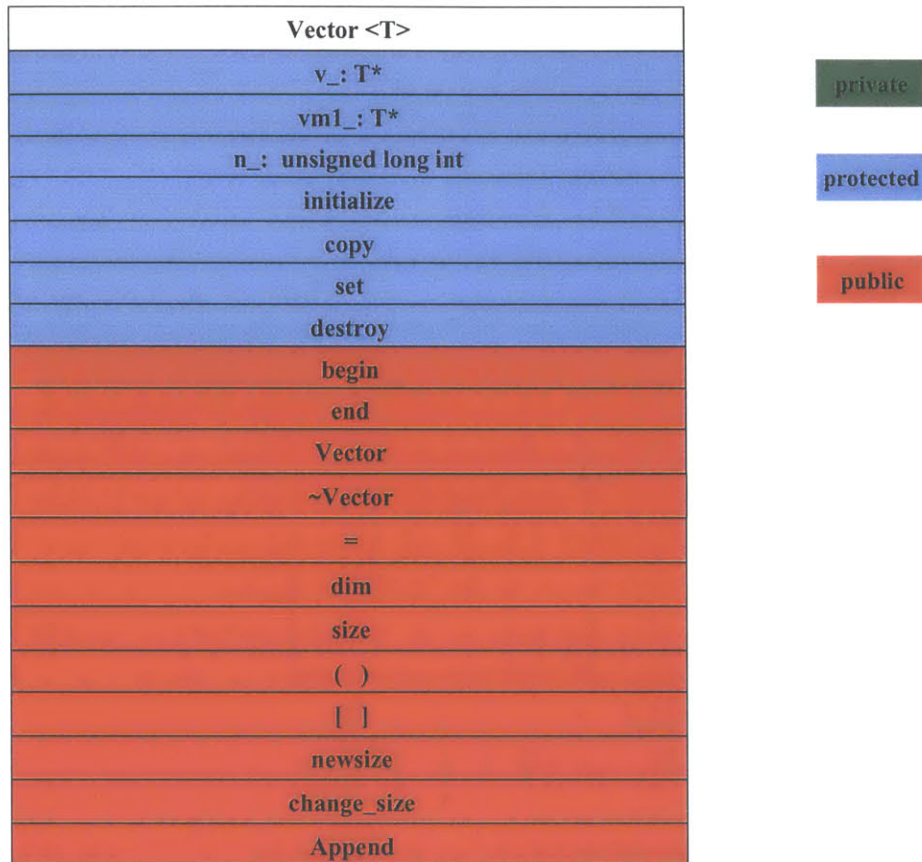


Figure A-46: Class diagram of class Vector

This header file originates from Template Numerical Toolkit (TNT). It has been added and changed by Pierre Elisseeff and Ding Wang. In this file, the class Vector is defined, which owns most properties of vector in mathematics. Moreover, many useful functions and operators for vector are constructed. Unlike the array data type in C++ which is 0-offset, vector is 1-offset.

A.33.1 Data Members

1. `T* v_` — protected; this is the 0-offset array containing elements of vector.

2. `T* vm1_` — protected; this is the 1-offset array containing elements of vector.

3. `unsigned long int n_` — protected; this is the length of the vector.

A.33.2 Member Functions & Operators

1. Name: `initialize`

   Overloads:

   - `void initialize(unsigned long int N)`

   Description:

   protected; This is an internal function to create `v_` with length `N` and then initialize `vm1_` and `n_`.

2. Name: `copy`

   Overloads:

   - `void copy(const T* v)`

   Description:

   protected; This function is to copy array `v` to `v_`. Note that this function must be used after initialize and the `N` in initialize must be equal to the length of `v`.

3. Name: `set`

   Overloads:

   - `void set(const T& val)`

   Description:

   protected; This function is to copy scalar `val` to each element in the vector.

4. Name: `destroy`

   Overloads:

   - `void destroy( )`

   Description:

   protected; This function destructs the vector and free space.

5. Name: `begin`

   Overloads:

- `iterator begin( )` — iterator is an alias of T*.
- `const iterator begin( ) const` — iterator is an alias of T*.

Description:

public; This function returns back the pointer pointing to the first element. In the second overload it is a constant function and the returned pointer points to a constant datum.

6. Name: `end`

   Overloads:

   - `iterator end( )` — iterator is an alias of T*.
   - `const iterator end( ) const` — iterator is an alias of T*.

   Description:

   public; This function returns back the pointer pointing to the last element. In the second overload it is a constant function and the returned pointer points to a constant datum.

7. Name: `Vector`

   Overloads:

   - `Vector( )`
   - `Vector(const Vector<T> &A)`
   - `Vector(size_type N, const T& value = T(0))` — size_type is an alias of unsigned long int.
   - `Vector(size_type N, const T* v)` — size_type is an alias of unsigned long int.
   - `Vector(size_type N, char *s)` — size_type is an alias of unsigned long int.

   Description:

   public; This is the constructor function. The 1st overload constructs a null vector; the 2nd overload constructs a copy of vector `A`; the 3rd overload constructs a vector with length `N` and assign scalar `value` to each element; the 4th overload constructs a vector

copy of N-element array v; the 5th overload constructs a vector copy of N-element string s.

8. Name: ~ `Vector`

   Overloads:

   - `~ Vector( )`

   Description:

   public; This is the destructor function. It deletes the vector and frees space.

9. Name: `=`

   Overloads:

   - `Vector<T> & operator=(const Vector<T> &A)`
   - `Vector<T> & operator=(const T& scalar)`

   Description:

   public; The 1st overload assigns vector A to the vector at left of '='; the 2nd overload assigns a scalar to each element of the vector at left of '='.

10. Name: `dim`

    Overloads:

    - `unsigned long int dim( ) const`

    Description:

    public; The function returns back the length of the vector.

11. Name: `size`

    Overloads:

    - `unsigned long int size( ) const`

    Description:

    public; The function returns back the length of the vector.

12. Name: `( )`

    Overloads:

- `inline reference operator( )(unsigned long int i)` — reference is an alias of T&
- `inline const_reference operator( ) (unsigned long int i) const` — const_reference is an alias of const T&

Description:
public; By this 1-offset sign, an element of vector can be extracted, e.g. x(i) is the i th element of x. The 2nd overload is a constant operator and return back a constant reference.

13. Name: `[ ]`
    Overloads:

    - `inline reference operator[ ](unsigned long int i)` — reference is an alias of T&
    - `inline const_reference operator[ ] (unsigned long int i) const` — const_reference is an alias of const T&

    Description:
    public; By this 0-offset sign, an element of vector can be extracted, e.g. x[i] is the i+1th element of x. The 2nd overload is a constant operator and return back a constant reference.

14. Name: `newsize`
    Overloads:

    - `Vector<T>& newsize(unsigned long int N)`

    Description:
    public; By this function, vector can be resized to `N` but its content may get lost. It returns back the new vector.

15. Name: `change_size`
    Overloads:

    - `Vector<T>& change_size(unsigned long int N)`

Description:
public; By this function, vector can be resized to N and its content will be kept as much as possible. It returns back the new vector.

16. Name: Append
    Overloads:
    - Vector<T>& Append(const Fortran_matrix<T> & mat_)
    - Vector<T>& Append(const Vector<T> & vec_)

    Description:
    public; This function can attach another vector vec_ to the original vector or columnwise attach a matrix mat_ to the original vector. It returns back the new vector.

A.33.3 Functions And Operators Defined In This File

1. Name: <<
   Overloads:
   - ostream& operator<<(ostream &s, const Vector<T> &A)

   Description:
   By this operator, vector A's length information and content can be output by I/O stream s. E.g. cout<<x<<endl; .

2. Name: >>
   Overloads:
   - istream& operator>><T>(istream &s, Vector<T> &A)

   Description:
   By this operator, vector A's length information and content can be input from I/O stream s. E.g. cin>>x; .

3. Name: +
   Overloads:
   - Vector<T> operator+(const Vector<T> &A, const Vector<T> &B)

- `Vector<T> operator+(const Vector<T> &A, const T &b)`
- `Vector<T> operator+(const T &b, const Vector<T> &A)`

Description:

'+' let vector `A` be able to plus another vector `B` or a scalar `b`. It returns back the summation.

4. Name: -

   Overloads:

   - `Vector<T> operator-(const Vector<T> &A, const Vector<T> &B)`
   - `Vector<T> operator-(const Vector<T> &A, const T &b)`
   - `Vector<T> operator-(const Vector<T> &A, const T &b)`

   Description:

   '–' let vector `A` be able to minus another vector `B` or a scalar `b`. Note that the 3rd overload has the same function as the 2nd overload. Result will be returned back.

5. Name: *

   Overloads:

   - `Vector<T> operator*(const Vector<T> &A, const Vector<T> &B)`
   - `Vector<T> operator*(const T &a, const Vector<T> &B)`
   - `Vector<T> operator*(const Vector<T> &B, const T &a)`

   Description:

   '*' let vector `A` be able to elementwise times another vector `B` or a scalar `a`. Result will be returned back.

6. Name: ==

   Overloads:

   - `bool operator==(const Vector<T> &A, const Vector<T> &B)`

   Description:

   '==' compares two vectors, if `A` and `B` are identical, returns 1; otherwise, returns 0.

7. Name: dot_prod

   Overloads:

   - T dot_prod(const Vector<T> &A, const Vector<T> &B)
   - complex<T> dot_prod(const Vector< complex<T>> &A, const Vector< complex<T>> &B)

   Description:

   This function provides dot product of two vectors A and B, which can be real or complex. Dot product result will be returned back.

8. Name: norm

   Overloads:

   - T norm(const Vector<T> &v)
   - T norm(const Vector< complex<T>> &v)

   Description:

   This function returns back the norm of a real or complex vector v.

9. Name: conj

   Overloads:

   - Vector< complex<T>> conj(const Vector< complex<T>> &v)

   Description:

   This function returns back the conjugate of a complex vector v.

10. Name: Append

    Overloads:

    - Vector<T> Append(const Vector<T> &A, const T &b)
    - Vector<T> Append(const Vector<T> &A, const Vector<T> &B)
    - Vector<complex<T>> Append(const Vector<complex<T>> &A, const Vector<complex<T>> &B)
    - Vector<T> Append(const Vector<T> &A, const Fortran_matrix<T> &C)

- `Vector<complex<T>> Append(const Vector<complex<T>> &A,`
  `const Fortran_matrix<complex<T>> &C)`

Description:

This function can attach a scalar `b` to a vector `A` or attach the right vector `B` to the left vector `A` or columnwise attach a matrix `C` to the left vector `A`. It returns back the new vector.

Bibliography

[1] H. Schmidt. Area: Adaptive rapid environmental assessment. In Pace and Jensen [11], pages 587–594.

[2] E. Coelho. Mesoscale — small scale oceanic variability effects on underwater acoustic signal propagation. In Pace and Jensen [11], pages 49–54.

[3] T. Evans S. Finette and C. Shen. Sub-mesoscale modeling of environmental variability in a shelf-slope region and the effect on acoustic fluctuations. In Pace and Jensen [11], pages 401–408.

[4] T. F. Duda. Relative influences of various environmental factors on 50-1000 Hz sound propagatioin shelf and slope areas. In Pace and Jensen [11], pages 393–400.

[5] S. Jesus A. Tolstoy and O. Rodríguez. Tidal effects on MFP via the intimate96 test. In Pace and Jensen [11], pages 457–464.

[6] T Akal. Effects of environmental variability on acoustic propagation loss in shallow water. In Pace and Jensen [11], pages 229–236.

[7] P.F.J. Lermusiaux A.R. Robinson, P. Abbot and L. Dillman. Transfer of uncertainties through physical-acoustical-sonar end-to-end systems: A conceptual basis. In Pace and Jensen [11], pages 603–610.

[8] N. M. Patrikalakis. Proposal to the National Science Fundationfor I-ADAPT, 2004.

[9] W.F. Rosenberger N. Flournoy and W.K.Wong, editors. *New Developments and Applications in Experimental Design*, volume 34 of *Lecture Notes - Monograph Series.* Institute of Math. Stat., 1998.

[10] D. P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, 2nd edition, 2001.

[11] N. G. Pace and F. B. Jensen, editors. *Impact of Littoral Environmental Variability on Acoustic Predictions and Sonar Performance.* Kluwer Academic Publishers, 2002.