NTNU – Trondheim
Norwegian University of
Science and Technology

DUNE: Unified Navigation Environment for the REMUS 100 AUV

Implementation, Simulator Development, and
Field Experiments

Sigurd Andreas Holsen

Master of Science in Engineering and ICT
Submission date: June 2015
Supervisor: Asgeir Johan Sørensen, IMT
Co-supervisor: Petter Norgren, IMT

Norwegian University of Science and Technology
Department of Marine Technology

NTNU Trondheim
Norwegian University of Science and Technology
Department of Marine Technology

MASTER THESIS IN MARINE CYBERNETICS

SPRING 2015

FOR

STUD. TECH. Sigurd Andreas Holsen

DUNE: Unified Navigation Environment for the REMUS 100 AUV

Implementation, Simulator Development, and Field Experiments

Work Description

Autonomous Underwater Vehicles (AUVs) are commonly used for underwater surveys as they have high maneuverability and can cover great distances. NTNU owns one such vehicle: the REMUS 100 developed by Hydroid. The REMUS 100 vehicle enables software extensions by use of the Hugin SDK plugin API. The API gives low-level access to the states of the vehicle and makes it possible to override the vehicles behavior.

Development of algorithms for navigation and control often involves complex systems and the Hugin SDK gives little aid to the structuring of such systems. To cope with this complexity, we have chosen to use the open source robot framework DUNE. By making an interface to DUNE from a Hugin SDK plugin, the control of the vehicle can be written using DUNE and leverage its field tested software modules for underwater applications. DUNE is also used by Department of Engineering and Cybernetics and Department of Marine Technology at NTNU. Using the same robot framework will allow for reuse of code across applications.

Scope of work

1. Development of an interface between DUNE and Hugin SDK through a plugin.
2. Development of a DUNE configuration for the REMUS 100 vehicle.
3. Development of an altitude controller in DUNE to move the vehicle with a constant distance to the seabed. The controller will take the Doppler Velocity Log (DVL) ranges as input.
4. Development of a simulator for the vehicle to be able to test the controller.
5. Field testing of the interface, DUNE configuration, and controller in Hopavågen.

NTNU Trondheim
Norwegian University of Science and Technology
Department of Marine Technology

6. Writing of documentation and guidelines for DUNE development on REMUS.

The report shall be written in English and edited as a research report including literature survey, description of mathematical models, description of control algorithms, simulation results, model test results, discussion and a conclusion including a proposal for further work. Source code should be provided on a CD with code listing enclosed in appendix. It is supposed that Department of Marine Technology, NTNU, can use the results freely in its research work, unless otherwise agreed upon, by referring to the student's work. The thesis should be submitted in three copies within June 10.

Advisers: PhD Candidate Petter Norgren

Professor Asgeir J. Sørensen
Supervisor

Abstract

This master thesis presents a new software system for developing control systems on the REMUS 100 autonomous underwater vehicle (AUV). Software written to control autonomous vehicles generally involve many components, and issues such as communication, synchronization, and modularity will arise. Many software frameworks have been written to address these issues. To ease the development, we have implemented an interface between the REMUS vehicle and the open source DUNE framework, leveraging its field-tested software for AUVs. To be able to simulate software written in DUNE, we have implemented an interface to an AUV simulator, AUVSim.

We have used DUNE to control the heading and the altitude of the vehicle. Two altitude controllers have been presented that use range measurements from a Doppler velocity log (DVL) as control input. The auto altitude controller uses the ranges to estimate the altitude of the vehicle. The highpass altitude controller uses an additional estimate of the seabed slope, resulting in a reduced altitude error on large bathymetric variations. Both altitude controllers have been implemented in DUNE and simulated in AUVSim with good results.

A field test was carried out in Hopavågen April 20-23, 2015 to test the two altitude controllers and the heading control via DUNE. The field test showed good result for the auto altitude controller. However, the highpass controller proved to be sensitive to noise in the range measurements from the DVL and did therefore give a noisy control output. Control of heading was not conducted due to an error in the REMUS API. This error was fixed after the field test was complete.

The field tests proved that DUNE was a good fit for controlling the REMUS 100 AUV. The NTNU REMUS 100 can now use DUNE as a development platform, allowing reuse of code from related research activities.

Sammendrag

Denne masteroppgaven presenter et nytt programvaresystem for utvikling av styringssystem for den autonome undervannsfarkosten REMUS 100. Programvare skrevet for å kontrollere autonome fartøy vil generelt sett inneholde mange komponenter, og problemstillinger som kommunikasjon, synkronisering, og modularitet vil fort oppstå. Mange programvarerammeverk har blitt utviklet for å bistå i disse utfordringene. For å tilrettelegge for videre utvikling har vi utviklet et grensesnitt mellom REMUS og DUNE, som er et rammeverk med åpen kildekode. DUNE er mye brukt for å styre undervannsfarkoster, noe som kan bidra til økt produktivitet for videre utvikling på REMUS. For å gjøre det mulig å simulere progamvaren, har vi implementert et grensesnitt til fartøysimulatoren AUVSim.

Vi har brukt DUNE til å styre kursen og altituden til farkosten. For å styre altituden har vi utviklet to regulatorer som bruker avstandsmålinger fra en Doppler velocity log (DVL) som innputt. Auto-regulatoren bruker avstandsmålingene til å estimere fartøyets altitude. Høypass-regulatoren kompanserer i tillegg for stigningen til havbunnen, noe som kan gi lavere regulatorfeil. Begge regulatorene har blitt implementert i DUNE og simulert i AUVSim.

En felttest ble gjennomført i Hopavågen fra 20. til 23. april 2015 for å teste de to altituderegulatorene. Felttesten viste gode resultater for auto-regulatoren, men støy i avstandsmålingene fra DVLen gjorde at resultatene fra høypass-regulatoren ble støyete. Styring av kurs ble ikke gjennomført på grunn av en feil i REMUS sitt API. Denne feilen ble identifisert og rettet etter at felttesten var gjennomført.

Resultatene fra felttesten viste at DUNE var godt egnet som utviklingsplatform for REMUS 100.

Acknowledgements

I would like to thank my supervisor Asgeir J. Sørensen and co-supervisor Petter Norgren for their contributions to this master thesis. Petter has patiently read through this thesis giving me advice on structure and theory. He spent a week week with me in Hopavågen to get the results presented in this thesis. Asgeir has given me invaluable guidance on academic writing and control theory. I would also like to thank Kristian Klausen for teaching me the basics of DUNE and Neptus and for responding to all my emails.

Last I want to thank my office mates at Tyholt.

Nomenclature

AUV - Autonomous Underwater Vehicle

DOF - Degrees Of Freedom

DUNE - DUNE Unified Navigational Environment

DVL - Doppler Velocity Log

GCS - Ground Control System

GPS - Global Positioning System

LSTS - Underwater Systems and Technology Laboratory (Portuguese)

MBE - Multibeam Echosounder

RECON - Remote Control Interface

ROS - Robot Operating System

ROV - Remotely Operated Vehicle

RPM - Revolutions Per Minute

SBC - Single Board Computer

SNAME - Society of Naval Architects and Marine Engineers

UDP - User Datagram Protocol

UUV - Unmanned Underwater Vehicle

VIP - Vehicle Interface Program

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

The aim of this thesis is to facilitate software development for the REMUS 100 autonomous underwater vehicle (AUV). Currently the only way to add functionality to the AUV is through the HUGIN SDK plugin API. Development of algorithms for navigation and control often involves complex systems and the HUGIN SDK gives little aid to the structuring of such systems.

Multiple software frameworks have been written to cope with the complexity of developing robotic software. Most of these frameworks support separation of the software into multiple user-defined components that interact with each other in a transparent manner. With smaller components it becomes easier to reason about the system.

In addition, they include a big amount of `plumbing` code required for an AUV to function. This include modules for data logging, synchronization and communication between components, noise filtering and more. To do research on high-level topics like maneuvering or autonomy it is necessary to have implemented solutions for these tasks. By using a field-tested software framework, the focus can be centered on development of new features that will result in increased productivity.

DUNE is a software framework made by the Underwater Systems and Technology Laboratory (LSTS, 2014), a part of the Faculty of Engineering, University of Porto. It has been tested in the field numerous times, is the software running on the AUVs produced by LSTS, and is actively developed.

1.2 Background

1.2.1 Unmanned Underwater Vehicles

Unmanned underwater vehicles (UUVs) are a growing research field and have many applications. Their use cases include bathymetric surveys, pipeline surveys, cable maintenance, marine archeology, and marine biology. Autonomous underwater vehicles is a category of underwater vehicles that is untethered and can operate autonomously without human intervention. They are commonly used for underwater surveys as they have high maneuverability and can cover great distances. AUVs are often underactuated, which means that they do not have actuators in all degrees-of-freedom (DOF) (Wadoo and Kachroo, 2010). Most AUVs have a stern propeller to control forward speed and two fins to control pitch and heading.

Another category of underwater vehicles is the remotely operated vehicle (ROV). ROVs are tethered and most often controlled by an operator located onshore or onboard a surface vessel. They are often equipped with manipulator arms that can be used for missions that require interaction.

1.2.2 Doppler Velocity Log

Most AUVs are equipped with a Doppler velocity log (DVL). A DVL sends out acoustic signals in multiple directions. When these beams hit a surface, they are reflected back to the DVL. The time between sending and receiving yields the distance along the beam, and a Doppler shift of the frequency gives the velocity. Three beams are necessary to get the velocity of the vehicle, but most DVLs have four beams making it more robust. When an UUV is relative close to the seabed, the DVL can be used as navigation input to estimate the position of the vehicle. Usage of DVL for navigation have been presented by Candeloro et al. (2012) and Dukan et al. (2011). Usage of the DVL range measurements for altitude control have been presented by Dukan and Sørensen (2012).

1.2.3 The REMUS 100 AUV

The AUV used in this thesis is a REMUS 100, first developed by the Woods Hole Oceanographic Institution. The AUV is now developed by Hydroid, a spinoff company from Woods Hole owned by KONGSBERG Group. The vehicle is equipped with two DVLs: one on the upper side of the vehicle and one underneath. Both DVLs are made by Teledyne RD Instruments (2014). In addition, the AUV is

Figure 1.1: DVL made by Teledyne RD Instruments (2014).

equipped with an altimeter measuring the distance straight down. This is the sensor that the Hydroid software uses for altitude control.

Figure 1.2: The REMUS 100 AUV developed by Hydroid.

REMUS is equipped with two horizontal fins called the stern planes, and two vertical fins called the rudder planes. The stern planes control the pitch of the vehicle while the rudder planes control the heading. Both the stern planes and the rudder planes are moving together, thus they cannot be controlled individually.

The REMUS 100 vehicle has multiple computers responsible for the parts of the system. The *Payload-processor computer* (PP computer) handles the plugins and all the sensors except for navigation. The PP computer is running the Windows 7 Embedded operating system. The *REMUS computer* runs the low-level control system responsible for setting the propeller speed and fin position.

1.2.4 HUGIN SDK

HUGIN SDK is the plugin framework originally created for the HUGIN AUV developed by KONGSBERG Maritime. It has since been ported to support the REMUS AUV. The framework is written in C++ and contains common building blocks for plugin creation including multithreading, synchronization, and file access.

In addition, it includes an interface to retrieve data from the AUV. Typical data includes estimated state from the observer, ranges from the DVL, and data from the side scan sonar.

1.2.5 DUNE

DUNE is made by the Underwater Systems and Technology Laboratory (LSTS, 2014), a part of the Faculty of Engineering, University of Porto. The development of DUNE started with the development of another software package, Neptus (Dias et al., 2005).

Neptus is graphical user interface (GUI) for operating multiple unmanned vehicles. This includes mission planning, simulation, execution, and review and analysis of mission data. A key point of the Neptus software is that it was designed for use with different types of vehicles including AUVs, ASVs, ROVs, and UAVs. This was also the reason for its creation.

DUNE is the runtime environment for the vehicles on-board software (Pinto et al., 2012). It was created because of the need to control networks of vehicles and to facilitate further research on unmanned vehicles. DUNE, together with Neptus, forms the LSTS-Toolchain for unmanned vehicles. A more thorough discussion about DUNE is given in Chapter 2.

1.2.6 AUVSim

AUVSim is a simulator for REMUS 100 created by PhD candidate Petter Norgren (Norgren and Skjetne, 2015). The simulator is implemented in MATLAB and Simulink. All simulations in this thesis have been performed by AUVSim.

1.3 Previous Work

1.3.1 Software Frameworks for Unmanned Vehicles

Different software frameworks are developed for different use cases. This section will present some of the frameworks and the purpose for their creation.

Orocos

Orocos (Open Robot Control Software) is a European project, started on September 1st, 2001. The project aimed at becoming a general-purpose and open robot control software package (Bruyninckx, 2001) since no other open source control software packages existed at the time. Orocos systems are separated into components that may be developed individually and connected together either at compile time or run time.

Orocos has a high focus on real time applications through its real time toolchain that provides fine grained control over how the software components interact. In addition, Orocos includes libraries for bayesian filtering like Kalman filters, dynamic bayesian networks, and particle filters, and libraries for kinematics and dynamics like kinematic chains and forward kinematics.

ROS

ROS (Robot Operating System) is maybe the most popular middleware for robotic applications. The software platform is a result of a collaboration between Willow Garage and Stanford University that started around 2007 (Magyar et al., 2015). ROS has been built from the ground up to encourage collaborative robotics software development (Quigley et al., 2009). ROS software are developed in packages that are easily distributed and shared. As of June 2015, over 3500 packages have been developed.

The official description of ROS is:

> ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers (ROS, 2014).

An evaluation of ROS and DUNE is presented in Chapter 2.

Rock

Rock (Robot Construction Kit) is an open source software framework for development of robotic systems. It is based on the real time toolchain in Orocos and

provides several ready-to-use drivers and modules.

Meinecke et al. (2013) have used Rock in the control system aboard the Marum Hybrid-ROV for under-ice operations. Natarajan et al. (2012) have used Rock for offline parameter identification using on-board sensors for an AUV.

1.4 Contributions

The contributions of this thesis are the following:

- A HUGIN SDK plugin for interfacing the REMUS 100 AUV with DUNE. This interface makes it possible to implement software extensions for REMUS 100 using the DUNE framework and leverage its field testes software modules for underwater applications. The architecture and design decisions for the software are given in Chapter 4 and Chapter 7.
- A beam range simulator capable of simulating DVL and MBE beam ranges is presented in Section 4.5.
- A altitude controller using beam range measurements from a DVL. The new altitude controller, named the highpass controller, uses the slope of the seabed to reduce the altitude error experienced when the slope of the seabed changes. The theory for highpass controller is given in Chapter 3 and has been simulated in Chapter 5. A field test using the REMUS 100 AUV was carried out in Hopavågen, Norway April 20-23, 2015. The field testing results are found in Chapter 6.
- Guidelines for DUNE development on REMUS based on the experiences from the field test. The guidelines are presented in Appendix A.

1.5 Organization of Thesis

Chapter 2 presents an evaluation of two software frameworks ROS and DUNE. The frameworks are compared in terms of configurability, ease of use, portability, user interface, and applicability for AUVs.

Chapter 3 presents modeling of the REMUS 100 AUV and development of two altitude controllers. The focus is on modeling and estimation of the altitude and the seabed slope using range measurements from a DVL. Two altitude controllers are presented that use these range measurements as input.

Chapter 4 presents the software written to control the REMUS AUV during the field tests. A HUGIN SDK plugin have been developed to interface the vehicle with DUNE. Finally, the AUVSim simulator is connected to DUNE allowing for simulation of control systems written in DUNE.

Chapter 5 presents the simulation results using the software and the altitude controllers given in the previous chapters. The AUVSim depth controller is tuned to resemble the depth controller on REMUS. The altitude controllers presented in Chapter 3 is simulated in AUVSim for three different scenarios.

Chapter 6 presents the execution of the field testing in Hopavågen. The chapter includes organization of the field testing, tuning of the REMUS depth controller, and the field test results.

Chapter 7 presents a new architecture for interfacing DUNE with the REMUS AUV based on the experiences from Hopavågen.

Appendix A presents a set of guidelines for DUNE development on the REMUS 100 AUV.

Appendix B presents the control flow for path planning and maneuvering in DUNE.

Appendix C presents new simulation results using a PID depth controller as opposed to the PI depth controller implemented on REMUS 100.

Appendix D presents the raw test results from the field test in Hopavågen.

Appendix E presents the attachments to this thesis.

Chapter 2

Evaluation of Software Frameworks for AUVs

This chapter presents an evaluation of software frameworks for AUVs. The two software frameworks that have been evaluated are DUNE and ROS. The frameworks are compared in terms of configurability, ease of use, portability, user interface, and applicability for AUVs. The result of the evaluation was the choice of using DUNE for the REMUS 100 AUV.

The chapter is based on what was done in the project thesis (Holsen, 2014), and has been repeated because of its relevance to this thesis.

2.1 Program Structure

DUNE

The base entity of a DUNE program is the *Task*. Tasks can be viewed as small configurable subprograms that run in parallel with other tasks. DUNE tasks can be diverse; some are responsible for interfacing with hardware, others are part of the chain of command translating higher-level goals to low-level actuator commands or monitoring the state of the vehicle (Faria et al., 2014). The communication between different tasks are done by message passing. Each task may subscribe to and broadcast messages. A depth controller task may subscribe to messages about desired depth and broadcast messages about thruster actuation.

A corner stone of the LSTS-Toolchain is the Inter-Module Communication (IMC)

protocol (Martins et al., 2009). This protocol describes the different data types and messages that can be sent between tasks. All message types in IMC is described in a single XML file `IMC.xml` that can be compiled into C++ and Java classes. Examples of different messages are `IMC::EstimatedState`, `IMC::DesiredControl`, and `IMC::SetThrusterActuation`. A depth controller for an AUV may listen to `IMC::EstimatedState` to get the current depth of the vehicle, and then after some calculations broadcast `IMC::DesiredControl`. The `IMC::DesiredControl` message may then be received by a thrust allocation task that in turn broadcasts `IMC::SetThrusterActuation` and `IMC::SetServoPosition`. Subscription and broadcasting of IMC messages in DUNE are handled by the *IMC Bus*. An illustration of the message passing can be seen in Figure 2.1

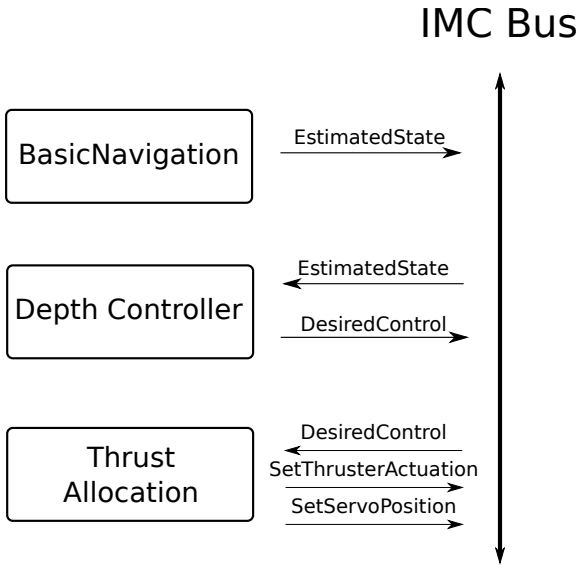


Figure 2.1: Message passing using IMC.

The benefit of this approach is that it makes it easy to replace a task. As long as the new task broadcasts the same messages as the replaced one, the rest of system should behave the same. It also makes it easy to retrieve information from the system since all messages are available to all tasks.

The tasks in DUNE are supported by the DUNE core library. The core library consists of all the *plumbing* code necessary for the tasks to function. This includes support for concurrency, monitoring, filtering, control algorithms, modeling, and the IMC protocol among others. Other applications that want to interact with DUNE may link to the core library.

The core library together with the tasks are compiled into a single binary executable. The executable will contain all tasks and hardware drivers. An unmanned aerial vehicle (UAV) may use the same executable as an AUV, and then run the executable with a different configuration. Configuration files are discussed in Section 2.2.

ROS

The basic entity of a ROS application is the *node*. A node is a program that performs computations. A ROS application is typically comprised of many such nodes, each responsible for one part of the application. The node in ROS differs from the tasks in DUNE in that each ROS node is compiled into its own executable while all DUNE tasks are compiled into the same DUNE executable.

This leads to a different architecture for communication between nodes. The single executable in DUNE makes it easy to locate the different tasks to find the receiver. In ROS, this responsibility is given to a special program called *master*. Each of the nodes have to connect to *master* to communicate with other nodes.

As in DUNE, ROS uses message passing to communicate between nodes. The types of messages are specified in the language-neutral interface definition language (IDL) where each message type is defined in a separate file. The IDL offers the same functionality as the IMC protocol in DUNE.

The message passing is done by subscribing and publishing to different *topics*. This allows for more fine-grained control of the message passing as there may be multiple topics using the same message type. An example of a message passing graph is shown in Figure 2.2. The nodes in the figure are showed as ellipsoids while the topics are showed as rectangles.

The way message passing works for the two platforms is different. In DUNE, all messages are sent to and received from a common message bus. In ROS, the messages are sent directly between the nodes using the peer-to-peer model. The difference lies in how the nodes/tasks establish the connection. In ROS, the connection between the nodes and the topics is explicitly established when setting up the connection. The connection procedure for ROS is shown in Figure 2.3.

Source Code Hierarchy

The source code in ROS is separated into packages that may contain any ROS file. A ROS package may reside in a ROS workspace for development, or be installed in the ROS package directory as a binary. This separation makes it easy

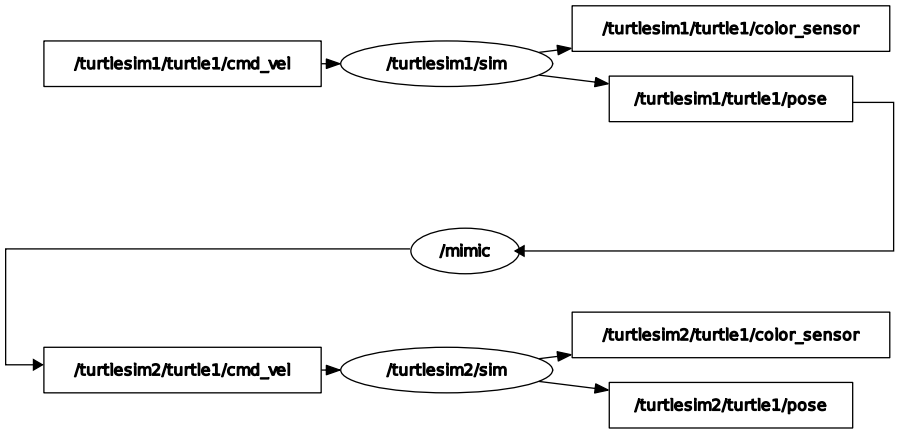


Figure 2.2: Communication graph in ROS made with `rqt_graph`

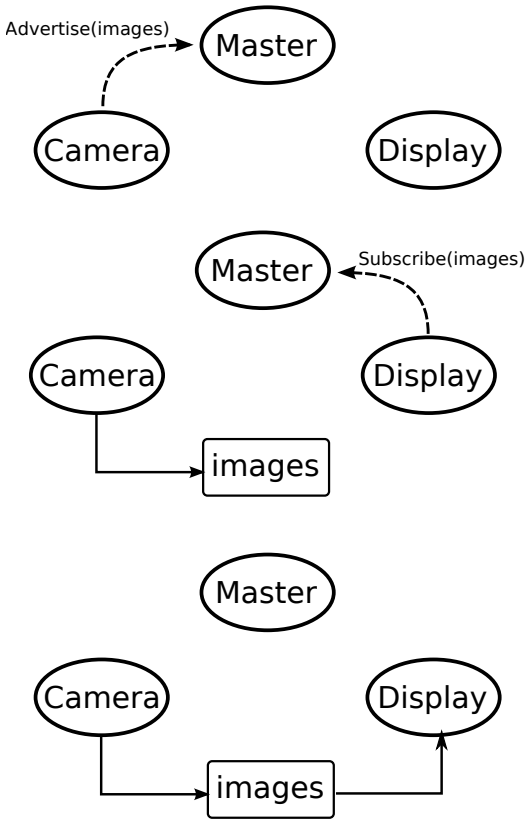


Figure 2.3: Connection setup in ROS between two nodes through the master channel. Courtesy of DeMarco et al. (2011).

to collaborate. If new functionality is needed in ROS, a new package can be created. The package may then be distributed to others as either binaries or source code. Another developer can then choose to install the new functionality.

2.2 Run Configurations

DUNE

Which tasks should be run is specified in a configuration file. In addition, the configuration file includes configuration parameters for each of the tasks. Different AUVs can use the same tasks, but with different configurations.

Configuration files may also be included in other configuration files forming a tree-like structure. For the AUVs that LSTS operates, most of the task parameters will be the same. These common parameters lies in the `etc/auv` folder in the DUNE source code. The parameters that differ can be overridden in parent files. An illustration of the tree structure for the SeaCon-1 AUV is shown in Figure 2.4

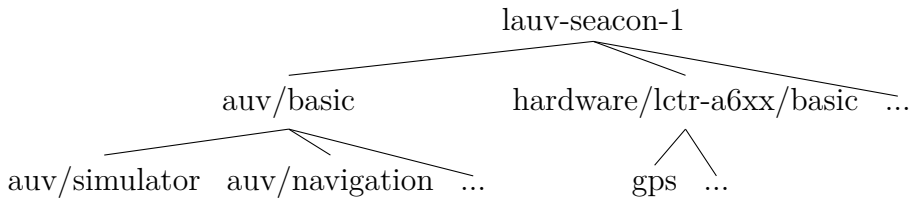


Figure 2.4: Tree structure comprising the configuration files for the SeaCon-1 AUV

DUNE can be run in different *profiles*. The four most used profiles are *Simulation*, *Hardware*, *Always*, and *Never*. All task entries in a configuration file must state what profiles it allows. Hardware sensors should only be enabled in *Hardware* and simulation tasks should only be enabled in *Simulation*. A task using the *Always* profile will always be enabled, and a task using the *Never* profile will never be enabled. This way it is possible to use the same configuration file for both real missions and simulations.

ROS

A ROS program called `roslaunch` makes it possible to run multiple nodes at once. `roslaunch` reads a configuration file in XML-syntax ands starts the appropriate

nodes. One important difference with DUNE tasks is that after the nodes have been launched, it's still possible to start new nodes and connect them to the others. This is not possible with a DUNE task. When a node start, it connects to the master channel to connect to the intended topics. This may be useful during debugging to launch plotting tools or other logging software.

The `roslaunch` program also has the functionality to start ROS nodes on multiple computers. For large-scale robots with many computational entities, this can be useful.

2.3 User Interface

2.3.1 Neptus

DUNE is tightly integrated with the Ground Control System (GCS) Neptus. The GCS is the user interface the operator interacts with to control the vehicles. This includes both setting up mission plans, executing mission plans, and reviewing the mission afterwards (Dias et al., 2006).

The goal of the Neptus framework is stated by Dias et al. (2005):

> This infrastructure, the Neptus framework, goal is to support the coordinated operation of heterogeneous teams which include autonomous and remotely operated underwater, surface, land, and air vehicles and people.

A key property of the Neptus framework is that it offers different interfaces for different use cases. The operation console is different for AUVs, ROVs, and UAVs. ROVs are for instance often equipped with cameras, therefore the operation console for ROVs includes a camera view (Dias et al., 2006). For UAVs, the operation console includes customized views for altitude and attitude. These customizations are not hard-coded, but can be configured using XML. New types of views can be written as plugins for Neptus and then configured to be included in the console.

Another goal of the Neptus project is to control multiple vehicles at once. Control of two AUVs in Neptus is shown in Figure 2.5. Field experiments and simulations with multiple vehicles have been performed by Pinto et al. (2013) and Marques et al. (2015). With the development of the NVL language (Marques et al., 2015), LSTS have executed coordinated missions where AUVs and an UAV have operated together using Neptus.

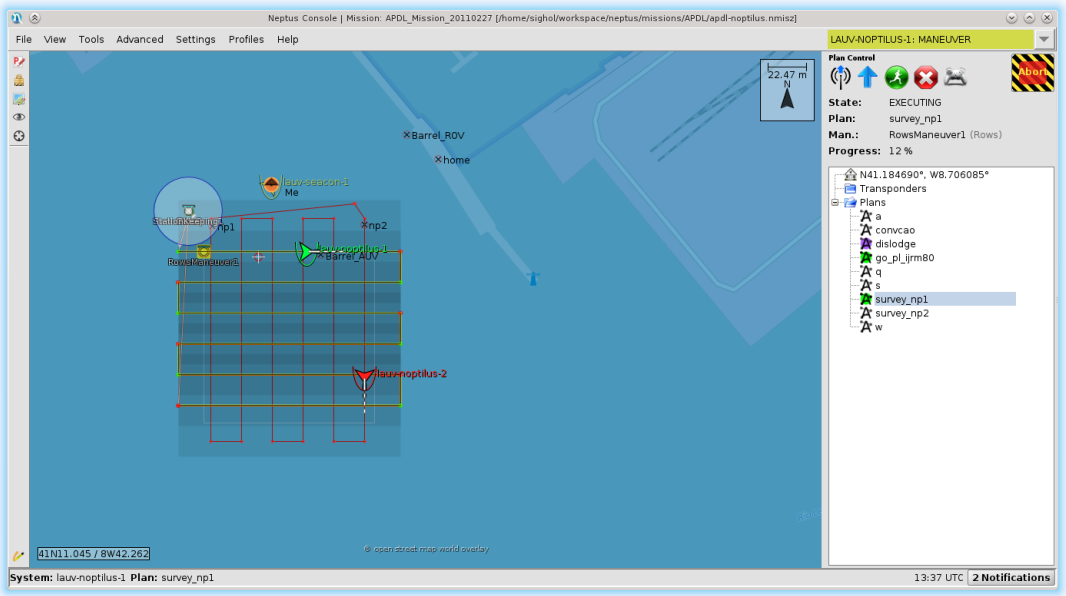


Figure 2.5: Neptus operator console. Simultaneous control of two AUVs.

For mission planning, Neptus offers a set of maneuvers that is linked together to form a mission plan. The types of maneuvers include waypoints, lawnmower patterns, and loitering. Mission plans created with Neptus may also be simulated before the they are executed.

Neptus provides a specialized application (Neptus MRA) to inspect and analyze the mission data after an operation (Pinto et al., 2012). After a mission or a maneuver has been performed, Neptus can requests a log of all the messages sent and received by DUNE. This data file can be read and analyzed by Neptus MRA, see Figure 2.6

The MRA has support for some of the data formats used by multibeam echosounders (MBE) and sidescan sonars. The MBE data can be presented in a 3D plot, while the sidescan sonar data can be presented as a sidescan plot (Figure 2.7) plotted directly on the map. A video demonstrating these capabilities have been published on `http://youtu.be/bBafWYAiPAY`.

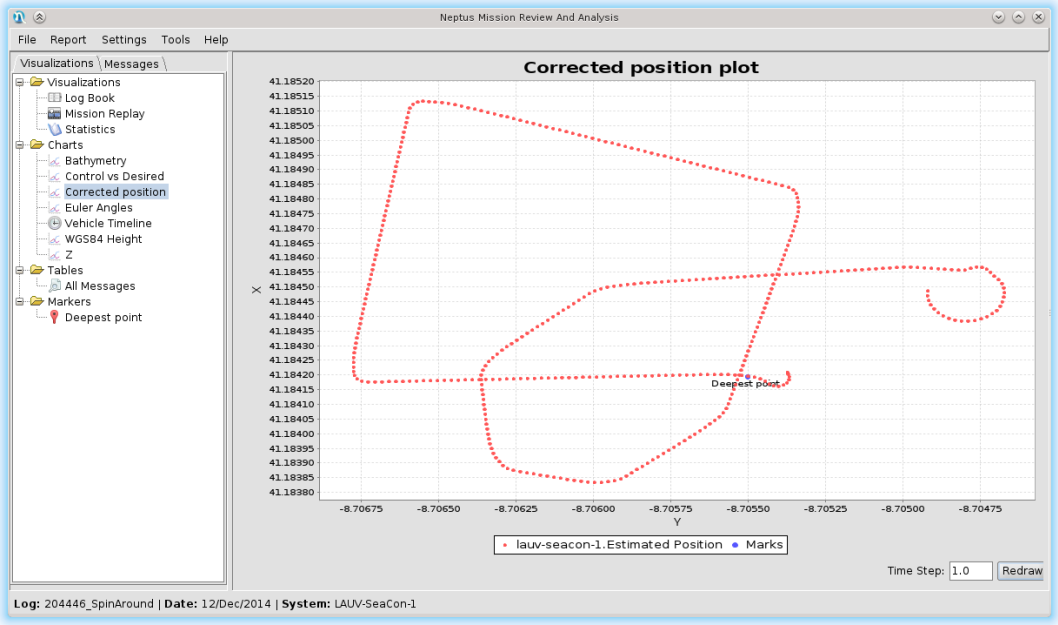


Figure 2.6: Neptus Mission Review and Analysis. Showing the xy-trajectory of an AUV mission.

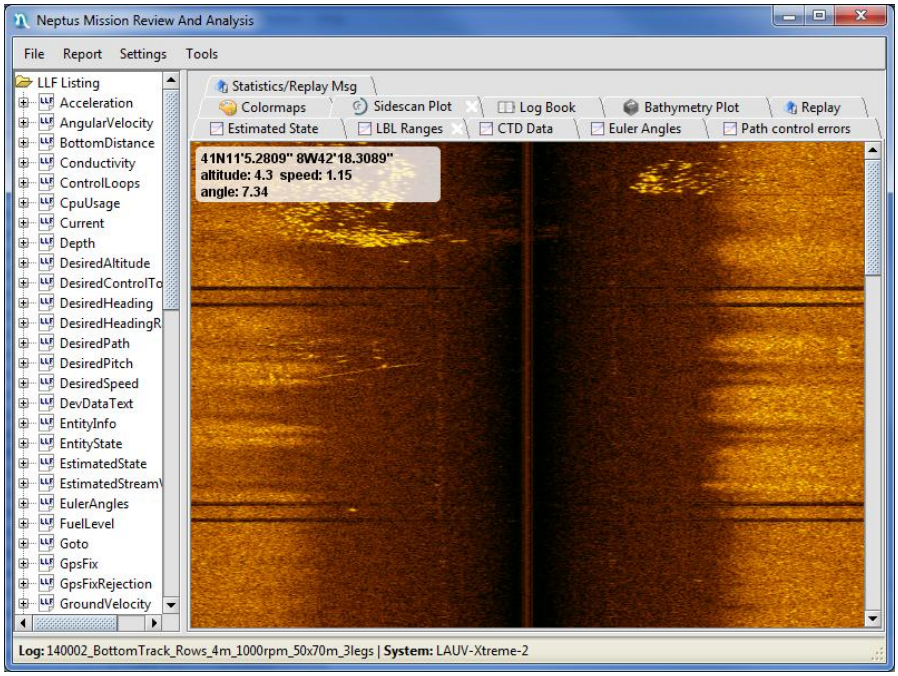


Figure 2.7: Neptus Mission Review and Analysis. Showing a Sidescan plot.

2.3.2 rqt

`rqt` is a Qt-based framework for GUI development for ROS (Thomas et al., 2014; Qt-Project, 2015). Most of the graphical tools offered by ROS is made with `rqt`. A benefit of this is that these tools can be docked in a single window creating custom interfaces (See Figure 2.8).

2.3.3 UWSim

UWSim is a software tool for visualization and simulation of underwater robotic missions (Prats et al., 2012). The visualization is done with OpenSceneGraph (2014) and osgOcean (2014). OpenSceneGraph is a high performance 3D graphics toolkit that is used both for scientific simulations and 3D games. osgOcean is an EU funded project that improves the graphics for underwater applications.

The core functionality of UWSim is to render underwater 3D visualizations. A simple example is a 3D plot of an AUV together with a bathymetry. The scene may be extended with other 3D models, like for example ship wrecks (see Figure 2.9).

The simulator in UWSim is purely kinematic. For each vehicle or object, it can take forces acting on the body as input and compute the next state. It also simulates different sensors, with or without noise. The different sensors include virtual cameras, localization sensors for position, and range sensors like altimeter, DVL, and MBE. The virtual camera can be used for pilot training or to test algorithms for image recognition.

With the simulator and the visualization, it is possible to simulate different control algorithms. UWSim exposes a ROS API making it possible to communicate with other ROS software. The control algorithms may then be written in any language that can interact with ROS. The dynamics simulations performed by Prats et al. (2012) was done in MATLAB.

One of the main goals of UWSim is that it should be modular and extensible. Research has been made to combine UWSim with the dynamic simulator Gazebo (Koenig and Howard, 2004; Kermorgant, 2014) which may give more realistic simulations for underwater behavior.

UWSim has support for collision detections. However, since the simulator is only kinematic, it does not support the dynamics of such collisions. This may improve in the future with research on Gazebo.

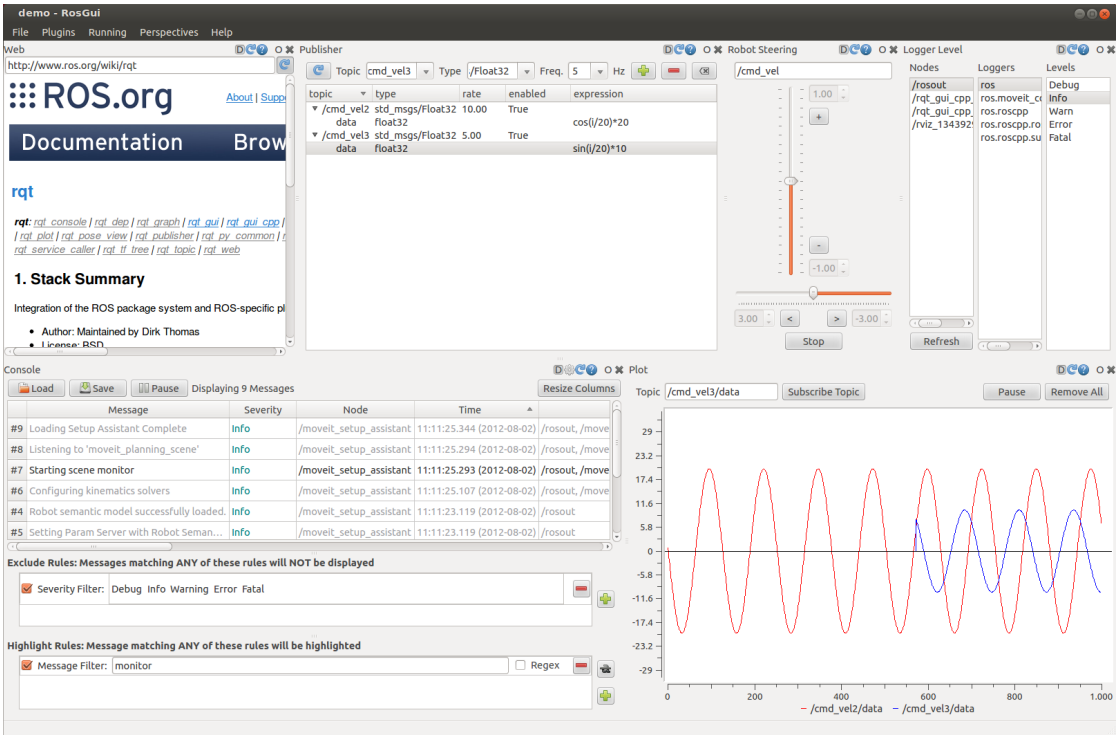


Figure 2.8: Custom interface made with `rqt`.

Figure 2.9: AUVs approaching a shipwreck in UWSim. Courtesy of IRS Lab (http://www.irs.uji.es/uwsim).

2.4 Portability

The most used and recommended operating system for ROS is the Ubuntu Linux distribution (Ubuntu, 2015) running on a x86 processor. There is experimental support for Arch Linux, OSX, and Ubuntu running ARM processors. Windows support is currently being developed, but is still considered unstable. Since ROS is comprised of many separate programs, cross compilation to Single Board Computers (SBCs) running ARM can be tedious and requires more effort. However, ROS have been used on embedded devices with success as done by Ma'sum et al. (2013) and Sa and Corke (2012).

It is the distributed nature of ROS that makes it less portable. Each node/program is compiled to a separate executable or library that may use any external resource they deem necessary. ROS packages will therefore have more requirements to external software than DUNE. Since most of the community evolves around Ubuntu as the main target, other Linux distributions, and especially Windows, have less focus. Small differences in the Linux distributions can lead to compilation errors.

DUNE on the other hand is compiled into a single executable that runs in a single operating system process. Because of this, cross compilation is only a matter of compiling the single binary to the new platform. This is in accordance with the goal that DUNE should work on embedded processors with limited capabilities.

2.5 Documentation and Real World Usage

ROS has a big community developing open source packages. At the time of writing, there are around 3500 community developed packages available for ROS. However, most of the focus goes towards robots in air and on ground. Few of the applications of ROS in underwater operations have published any source packages. Although few packages is focused directly on underwater vehicles, ROS has extensive support for kinematics and manipulators (Sucan and Chitta, 2014).

The community around DUNE is active, but considerable smaller.

2.5.1 ROS Used for AUV and ROV operations

Open-loop control of ROV

The first documentet usage of ROS for underwater applications is done by Johns Hopkins University Dynamical Systems and Control Laboratory (2014). They have published a video where they use ROS to do open loop control of an ROV. This was part of an exercise in using ROS. Based on the demonstration in the video, the exercise was successful.

No source code, or any documentation beyond this video, have been published.

Yellowfin

Yellowfin is an AUV developed by Georgia Tech Research Institute specifically for missions that require autonomous collaboration amongst multiple vehicles. (Melim and West, 2011; DeMarco et al., 2011). It has been developed from the ground up with common of the shelf hardware and a custom built body.

The Yellowfin computer control is separated into two sections called *front-seat* and *back-seat*. Each of the sections are a implemented on its own Single Board Computer (SBC). The front-seat board is responsible for the low-level control of the actuators and sensors. This will coincide with the actuator control level used by Sørensen (2013). The back-seat computer is responsible for the high-level control like mission planning and navigation. It is the back-seat computer that is running ROS. This is the most complex of the two and is therefore more prone to errors, which is the reason for the separation.

The control system in the back-seat is separated in two platforms. On earlier operations, they have used the MIT developed MOOS framework (Newman, 2008). To utilize the MOOS code, they made a bridge between the MOOS framework and ROS (Newman, 2014).

This shows some of the flexibility of ROS. Both ROS and MOOSE uses message passing for inter-process communication. Given a configuration file specifying the message formats, the bridge will translate the messages between the systems.

The source code for the ROS/MOOSE bridge have been published as a ROS package. However, the ROS source code for path planning and navigation has not.

Girona 500 and SPARUS II

The University of Girona in Spain have developed two AUVs that use ROS for the whole control chain: Girona 500 and SPARUS II (Carreras et al., 2013). The control system is developed mostly using the Python programming language. As opposed to the other AUVs, the code is open source and freely available (Cola2, 2015). They have used UWSim for the simulations.

2.5.2 DUNE Used for AUV Operations

DUNE and Neptus have been used on numerous missions. This section presents two missions of particular interest.

Autonomous operation

Faria et al. (2014) have used DUNE and Neptus to coordinate multiple vehicles. The mission that was carried out involved multiple AUVs and unmanned aerial vehicles (UAVs). The UAVs were equipped with cameras that were used to detect features of interest. When a feature of interest had been found, they messaged the AUVs that would do further inspections.

`ArduPilot` (ArduPilot, 2015) was used as the flight controller for the UAVs. ArduPilot is an open source low-cost autopilot suite consisting of both hardware and software. This autopilot was connected through a serial port to a Single-Board Computer running DUNE . To integrate it with DUNE, a specialized driver task was created which translates commands between DUNE and ArduPilot. The AUVs were controlled in full by DUNE.

`T-REX` is an open source on-board adaptive control system that integrates AI-based planning and state estimation (Faria et al., 2014). T-REX was responsible for the deliberation of the plans while the plan execution was carried out by DUNE.

To communicate with the AUVs when they were underwater, they employed a gateway buoy that would forward messages to the AUV via acoustic signals. When the AUVs were in the surface, it would forward messages via long range WiFi. The Network connectivity is shown in Figure 2.10.

Coordinated maneuvers with multiple vehicles

Marques et al. (2015) presents the NVL language used to coordinate multiple

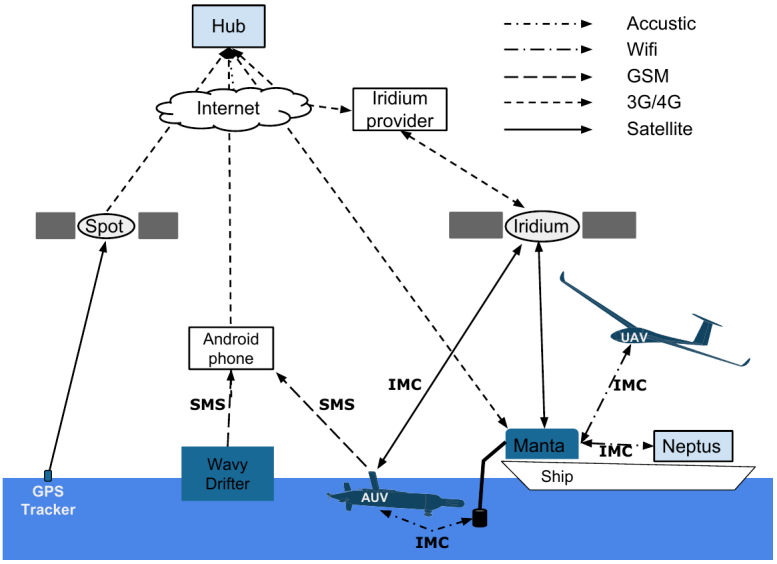


Figure 2.10: Network connectivity across platforms. Courtesy of Faria et al. (2014)

vehicles. Control of a network of vehicles is challenging, especially when vehicles need to interact with each other. The scenario included three AUVs surveying three areas and an UAV that was used as a data mule. That is, the UAV was collecting data from the AUVs after they had finished surveying. The NVL language was used to select vehicles to execute maneuvers and synchronize maneuvers involving multiple vehicles. The action in need of synchronization was the transmission of data between the AUVs and the UAV. During data transmission, the UAV would fly in circles above the AUV until the data had been transferred.

2.6 Summary

On the low level, DUNE and ROS supports the same functionality. Both systems are applicable for developing control software for underwater vehicles. On-board robotic software has been tested successfully using both platforms. However, DUNE has more ready to use software for underwater operations. Of the two uses of ROS in underwater operations, only one of the projects had published working code.

DUNE on the other hand has been successfully used with both AUVs and ROVs. It connects to the Neptus Ground Control System that is a ready to use graphical user interface for operators. The IMC protocol allows for joint missions with many different vehicles that can all be monitored and controlled through Neptus.

ROS has a strong open source history and an active and growing community. However, most of the focus goes towards robots on ground and in air. A common focus for ground robots and underwater vehicles is the use of manipulator arms. With the development of Gazebo, ROS has extensive support for kinematics and manipulator arms that is applicable for ROVs.

ROS also has the advantage of a 3D kinematic simulator of underwater vehicles, UWSim. Work is being made to integrate UWSim with Gazebo that may be of interest for the ROV community.

The PP computer aboard the REMUS 100 AUV is using the Windows 7 Embedded operating system. If a software framework should be used on the REMUS, it must therefore support Windows. As of 2015, Windows support on ROS is considered experimental. For this reason, and since DUNE works so well with the Neptus GCS, DUNE was chosen to be used on the REMUS 100 AUV.

A summary in pros and cons format is given below.

DUNE

Pros

- Available software and documentation.
- Tested in the field numerous times.
- Supports coordinated behavior of a network of vehicles.
- Neptus allows multiple operators to monitor and maneuver a network of vehicles.
- In active development by LSTS.
- Available on Windows.

Cons

- Not as flexible as ROS for communication between tasks.
- No concept of packages, making collaboration require more effort and coordination.

ROS

Pros

- Extensive support for manipulator kinematics.

- Big and active community.
- `rqt_graph` can visualize the communication which makes it easier to understand the control flow.
- 3D-visualization with UWSim.
- Easy collaboration with packages.

Cons

- Windows support is experimental.
- Crosscompilation requires more effort.
- Little available software for underwater applications.
- The community is mostly focused on ground and air vehicles.

Chapter 3

Modeling and Control System

During missions it is often convenient for the AUV to run on a constant altitude to be able to collect data near the seabed. Most AUVs are equipped with an altimeter that measures the altitude. The altitude can then be fed into a depth controller to perform altitude control. An alternative to using the altimeter sensor, is to use the range measurements from a Doppler velocity log (DVL). One advantage with using the DVL is that the DVL makes range measurements in multiple directions making it possible to compensate for the slope of the seabed.

This chapter will present the kinematics and kinetics for doing altitude control with REMUS, and will present two altitude controllers that use the range measurements from a DVL. Both altitude controllers have been implemented in DUNE.

3.1 Modeling

For the system equations, the SNAME notation (Society of Naval Architects and Marine Engineers) is used (see Table 3.1).

Table 3.1: The notation of SNAME(1950) for marine vessels.

DOF		Forces and moments	Linear and angular velocities	Position and Euler angles
1	motions in the x direction (surge)	X	u	x
2	motions in the y direction (sway)	Y	v	y
3	motions in the z direction (heave)	Z	w	z
4	rotation about the x axis (roll)	K	p	ϕ
5	rotation about the y axis (pitch)	M	q	θ
6	rotation about the z axis (yaw)	N	r	ψ

The time derivative of a variable $x(t)$ is denoted $\dot{x}$. Vectors are written in small letters in bold and matrices are written in capital letters in bold. The dimension of each variable will be defined. A variable in Euclidean space with dimension n is denoted $\mathbb{R}^n$, while matrices of dimension $n \times m$ are denoted $\mathbb{R}^{n \times m}$.

3.1.1 Reference Frames

Two reference frames are used to express the motions and dynamics of the vehicle. The BODY fixed reference frame has its origin in the center of the vehicle, usually coincided with the vehicles center of gravity. The North-East-Down (NED) reference frame has its origin relative to an earth-fixed position typically given by a latitude and longitude.

The vehicle's position $\boldsymbol{p}$ and attitude $\boldsymbol{\Theta}$ are given in the NED frame.

$$\boldsymbol{\eta} = \begin{bmatrix} \boldsymbol{p} & \boldsymbol{\Theta} \end{bmatrix}^T = \begin{bmatrix} x & y & z & \phi & \theta & \psi \end{bmatrix}^T \in \mathbb{R}^6 \tag{3.1}$$

The vehicle's linear velocity $\boldsymbol{v}$ and angular velocity $\boldsymbol{\omega}$ are given in the BODY frame.

$$\boldsymbol{\nu} = \begin{bmatrix} \boldsymbol{v} & \boldsymbol{\omega} \end{bmatrix}^T = \begin{bmatrix} u & v & w & p & q & r \end{bmatrix}^T \in \mathbb{R}^6 \tag{3.2}$$

The kinematic relationship between the velocity vector $\boldsymbol{\nu}$ in the BODY frame and the position vector $\boldsymbol{\eta}$ in the NED frame is expressed by

$$\dot{\boldsymbol{\eta}} = \boldsymbol{J}_\Theta(\boldsymbol{\eta})\boldsymbol{\nu} \tag{3.3}$$

where

$$\boldsymbol{J}_\Theta(\boldsymbol{\eta}) = \begin{bmatrix} \boldsymbol{R}_b^n(\boldsymbol{\Theta}) & 0_{3x3} \\ 0_{3x3} & \boldsymbol{T}_\Theta(\boldsymbol{\Theta}) \end{bmatrix}. \tag{3.4}$$

$\boldsymbol{R}_b^n$ denotes the rotation matrix from BODY (b) frame to NED (n) frame. Using $c\cdot = \cos(\cdot)$, $s\cdot = \sin(\cdot)$, and $t\cdot = \tan(\cdot)$, $\boldsymbol{R}_b^n(\boldsymbol{\Theta})$ is given by

$$\boldsymbol{R}_b^n(\boldsymbol{\Theta}) = \begin{bmatrix} c\psi c\theta & -s\psi c\phi + c\psi s\theta s\phi & s\psi s\phi + c\psi c\phi s\theta \\ s\psi c\theta & c\psi c\phi + s\phi s\theta s\psi & -c\psi s\phi + s\theta s\psi c\phi \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix} \tag{3.5}$$

and $\boldsymbol{T}_\Theta(\boldsymbol{\Theta})$ is given by

$$\boldsymbol{T}_\Theta(\boldsymbol{\Theta}) = \begin{bmatrix} 1 & s\phi t\theta & c\phi t\theta \\ 0 & c\phi & -s\phi \\ 0 & s\phi/c\theta & c\phi/c\theta \end{bmatrix}. \tag{3.6}$$

3.1.2 Equations of Motion

According to Fossen (2011), the marine craft equations of motion can be written in a vectorial setting.

$$\dot{\boldsymbol{\eta}} = \boldsymbol{J}_\Theta(\boldsymbol{\eta})\boldsymbol{\nu} \tag{3.7}$$

$$\boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{g}(\boldsymbol{\eta}) = \boldsymbol{\tau} \tag{3.8}$$

$\boldsymbol{M} = \boldsymbol{M}_{RB} + \boldsymbol{M}_A \in \mathbb{R}^{6\times6}$ is the rigid-body inertia and added mass of the AUV. $\boldsymbol{C}(\boldsymbol{\nu}) = \boldsymbol{C}_{RB} + \boldsymbol{C}_A \in \mathbb{R}^{6\times6}$ is the Coriolis and centripetal for the rigid-body and added mass, while $\boldsymbol{D}(\boldsymbol{\nu}) \in \mathbb{R}^{6\times6}$ represents the hydrodynamic damping effects. The restoring forces are given by $\boldsymbol{g}(\boldsymbol{\eta}) \in \mathbb{R}^6$. $\boldsymbol{\tau}$ is the forces and moments acting on the AUV from the propeller, rudder, and stern planes.

The forces and moments from the rudder and stern fins have been derived by Prestero (2001).

$$\begin{aligned} Y_r &= \frac{1}{2}\rho c_{L\alpha} S_{fin}[u^2\delta_r - uv - x_{fin}(ur)] \\ Z_s &= -\frac{1}{2}\rho c_{L\alpha} S_{fin}[u^2\delta_s - uw - x_{fin}(uq)] \\ M_s &= \frac{1}{2}\rho c_{L\alpha} S_{fin} x_{fin}[u^2\delta_s - uw - x_{fin}(uq)] \\ N_r &= \frac{1}{2}\rho c_{L\alpha} S_{fin} x_{fin}[u^2\delta_r - uv - x_{fin}(ur)] \end{aligned} \tag{3.9}$$

where ρ is the density of the water, S_{fin} is the fin area, x_{fin} is the distance from the BODY-fixed origin to the fin position, and $c_{L\alpha}$ is the lift coefficient for the fin. δ_s and δ_r are the stern and rudder angles relative to the body-fixed x-axis,

respectively. The coefficients listed above have been estimated for the REMUS 100 AUV by Prestero (2001).

The surge force and roll moment from the propeller have been derived by Carlton (2012).

$$X_p = K_T \rho D^4 n^2 \tag{3.10}$$
$$K_p = K_Q \rho D^5 n^2 \tag{3.11}$$

where K_T and K_Q is the thrust and torque coefficients, respectively. D is the diameter of the propeller, and n is the propeller shaft speed given in revolutions-per-second. The propeller coefficients for the REMUS 100 AUV have been estimated by Allen et al. (2000).

Finally, $\boldsymbol{\tau}$ can be written in terms of the forces and moments

$$\boldsymbol{\tau} = \begin{bmatrix} X_p & Y_z & Z_s & M_s & N_r & K_p \end{bmatrix}. \tag{3.12}$$

This AUV model has been implemented in the AUVSim simulator by Norgren and Skjetne (2015).

3.1.3 Altitude Kinematics

The altitude of the AUV is given by the following definition by Dukan and Sørensen (2012).

Definition 1. *The altitude is the length of the vector from the center of origin (CO) of the AUV to the point on the seabed with the same horizontal coordinates as the CO.*

Let

$$z_{sb} = f(x, y) \tag{3.13}$$

define the depth of the seabed at position (x, y). The altitude can then be expressed in a vectorial setting as

$$\boldsymbol{a} = \mathbf{r}_a - \mathbf{p} = \begin{bmatrix} x \\ y \\ f(x,y) \end{bmatrix} - \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ f(x,y) - z \end{bmatrix} \tag{3.14}$$
$$a = |\boldsymbol{a}| = f(x, y) - z \tag{3.15}$$

where $\mathbf{r}_a$ is the position vector to the seabed at the same horizontal position as the AUV.

Altitude Rate of Change

The altitude rate of change can be found by differentiating the altitude expression:

$$\dot{a} = \dot{f}(x, y) - \dot{z}. \tag{3.16}$$

By using partial derivation and the chain rule, this becomes

$$\dot{a} = \frac{\delta f}{\delta x}\frac{dx}{dt} + \frac{\delta f}{\delta y}\frac{dy}{dt} - \frac{dz}{dt}. \tag{3.17}$$

The expression can by simplified by introducing the seabed surface given by

$$F(x, y, z) = f(x, y) - z = 0. \tag{3.18}$$

F is generally an unknown function, but only the gradient of F, the slope of the seabed, is of interest. Evaluated at the AUVs position $\mathbf{p}$, this gradient is expressed by

$$\nabla F(\mathbf{p}) = \left[\tfrac{\delta f}{\delta x}|_{\mathbf{p}}, \tfrac{\delta f}{\delta y}|_{\mathbf{p}}, -1\right]. \tag{3.19}$$

The expression for $\dot{a}$ can then be written as

$$\dot{a} = \nabla F(\mathbf{p}) \cdot \dot{\mathbf{p}} \tag{3.20}$$

$$= \nabla F(\mathbf{p}) \boldsymbol{R}_b^n(\boldsymbol{\Theta}) \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{3.21}$$

$$= \left[\tfrac{\delta f}{\delta x}|_p, \tfrac{\delta f}{\delta y}|_p, -1\right] \boldsymbol{R}_b^n(\boldsymbol{\Theta}) \begin{bmatrix} u \\ v \\ w \end{bmatrix}. \tag{3.22}$$

(3.22) is a result found by Dukan and Sørensen (2012).

Sea Depth Rate of Change

Another use case for the bathymetry gradient is to estimate the sea depth rate of change $\dot{z}_{sb}$. Since the sea depth is given by $z_{sb} = f(x, y) = f(\mathbf{p})$, the gradient is expressed by

$$\nabla f(\mathbf{p}) = \left[\tfrac{\delta f}{\delta x}|_{\mathbf{p}}, \tfrac{\delta f}{\delta y}|_{\mathbf{p}}, 0\right]. \tag{3.23}$$

By using the same approach as with the altitude rate of change, the expression becomes

$$\dot{z}_{sb} = \nabla f(\mathbf{p}) \boldsymbol{R}_b^n(\boldsymbol{\Theta}) \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{3.24}$$

$$= \left[\frac{\delta f}{\delta x}|_{\mathbf{p}}, \frac{\delta f}{\delta y}|_{\mathbf{p}}, 0 \right] \boldsymbol{R}_b^n(\boldsymbol{\Theta}) \begin{bmatrix} u \\ v \\ w \end{bmatrix}. \tag{3.25}$$

The difference between the two rates is that the latter does not take the vertical motion of the AUV into account.

3.2 Sensors and Controllers Aboard REMUS 100

3.2.1 DVL Sensors

The REMUS 100 is equipped with two DVL sensors: one above and one below the vehicle. It is thus capable of measuring the distance both to the seabed and to the surface.

Each of the DVL beams has an angle of 20 degrees along the vehicles longitudinal and transversal axis illuminating a square as seen in Figure 3.1. See Figure 3.2 for the numbering of the DVL beams. The 1^{st} beam will be offset with 20 degrees in positive u direction and 20 degrees in positive v direction.

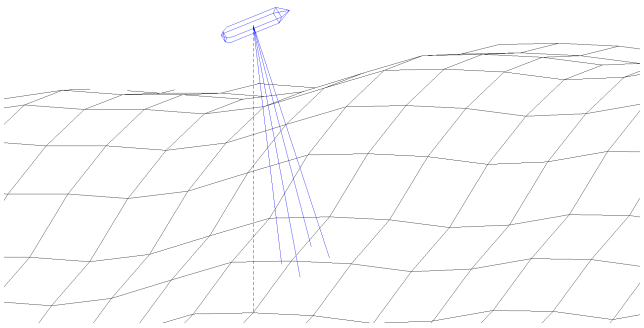


Figure 3.1: An AUV sending out four DVL beams. The dottet line represents altitude.

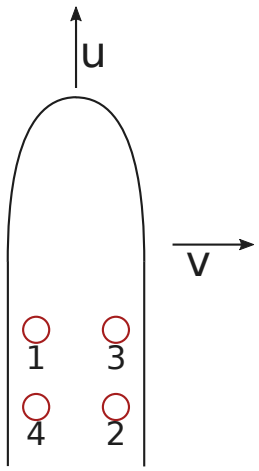


Figure 3.2: Numbering of the DVL beams. u points in the surge direction and v point in the sway direction

3.2.2 The REMUS Controllers

REMUS has two controllers that works in conjunction to control the vertical position: a pitch controller and a depth controller. The depth controller provides the set points for the pitch controller. The depth controller is a PI controller and the pitch controller is a PID controller.

The heading is controlled by a PID controller. The guidance algorithm that provides set points to the heading controller is not known. However, the results from the field testing discussed in Chapter 6 indicate that it is similar to the line-of-sight algorithm.

3.3 Altitude Estimation by use of DVL

The method of altitude estimation that is used in this section has been presented by Dukan and Sørensen (2012).

We assume that the direction of the DVL beams and the rotation of the vehicle is known. Let $\boldsymbol{R}^b_{DVL_i}$ be the rotation matrix from the i^{th} beam to the BODY frame. The direction of each beam can then be expressed by

$$\frac{\mathbf{r}_i}{|\mathbf{r}_i|} = \boldsymbol{R}^n_b \boldsymbol{R}^b_{DVL_i} \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T \tag{3.26}$$

where $\mathbf{r}_i$ is the position vector from the AUVs center of origin to the i^{th} beams end point. With the range measurements $|\mathbf{r}_i|$ from the DVL, the beams position vector in the NED frame is given by

$$\mathbf{r}_i = \boldsymbol{R}_b^n \boldsymbol{R}_{DVL_i}^b \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T \cdot |\mathbf{r}_i|. \tag{3.27}$$

The position vectors $\mathbf{r}_i$ is used to create a first order approximation of the seabed beneath the vehicle. The altitude estimate is then found by following the approximated plane to the horizontal position of the vehicle. This is shown in 2D in Figure 3.3.

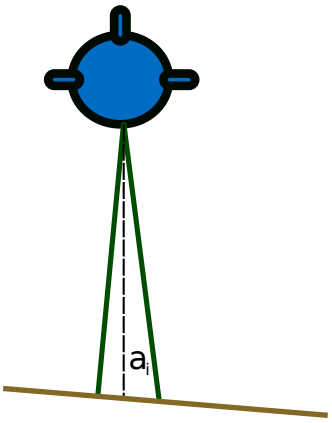


Figure 3.3: Altitude estimate in 2D.

Three points are needed to define a plane, and since the DVL sends out 4 beams, 4 planes can be defined. The normal vectors for the planes are given by

$$\begin{aligned} \mathbf{n}_0 &= (\mathbf{r}_0 - \mathbf{r}_1) \times (\mathbf{r}_3 - \mathbf{r}_0) \\ \mathbf{n}_1 &= (\mathbf{r}_1 - \mathbf{r}_2) \times (\mathbf{r}_0 - \mathbf{r}_1) \\ \mathbf{n}_2 &= (\mathbf{r}_2 - \mathbf{r}_3) \times (\mathbf{r}_1 - \mathbf{r}_2) \\ \mathbf{n}_3 &= (\mathbf{r}_3 - \mathbf{r}_0) \times (\mathbf{r}_2 - \mathbf{r}_3). \end{aligned} \tag{3.28}$$

This gives one altitude approximation a_i for each plane $\mathbf{n}_i$.

$$a_i = r_{i,z} + \frac{n_{i,x}}{n_{i,z}} r_{i,x} + \frac{n_{i,y}}{n_{i,z}} r_{i,y} \tag{3.29}$$

3.3.1 Altitude Rate of Change

The DVL ranges can be used to estimate the altitude rate of change by using the planes given in (3.28).

$$\nabla F(\mathbf{p}) \approx \begin{bmatrix} -\frac{n_{i,x}}{n_{i,z}} & -\frac{n_{i,y}}{n_{i,z}} & -1 \end{bmatrix} \tag{3.30}$$

3.4 Altitude Control

Since the REMUS vehicle has a working depth controller, an altitude controller can be developed to send a desired depth to this depth controller.

Another option would be to implement an altitude controller in DUNE that sets the fin angles directly. However this would require direct control of both the rudder fin and the propeller RPM, and it would thus be necessary to reimplement a complete control system for the REMUS AUV. It was therefore easier to develop the altitude controller using the depth controller on REMUS. How the REMUS AUV can be controlled is discussed in more detail in Chapter 4.

3.4.1 Auto Altitude Controller

Auto altitude is the simplest possible altitude controller. The control law is given by

$$z_{d,auto} = z_{sb} - a_d \tag{3.31}$$

where a_d is the desired altitude and $z_{d,auto}$ is the desired depth sent to the depth controller. Given an ideal depth controller that is able to follow any control input, the error

$$a_e = a_d - a \tag{3.32}$$

will always be zero.

Due to the integrator in the depth controller, the vehicle will have no steady state error when the seabed slope is constant.

3.4.2 Highpass Controller

The change in seabed depth $\dot{z}_{sb}$ can be exploited in a feedforward term to reduce the altitude error. Let

$$z_{d,hp} = z_{sb} - a_d + z_{ff} = z_{d,auto} + z_{ff} \tag{3.33}$$

be a new control law for altitude control where z_{ff} is the feedforward term. Since the REMUS depth controller have zero steady state error when receiving a linearly increasing control input, the feedforward term must converge to zero when the seabed slope is constant. To be added to the feedforward term, $\dot{z}_{sb}$ must therefore

be filtered such that only changes in the seabed slope are used. It is thus only the high frequency part of $\dot{z}_{sb}$ that can be used. This high frequency part can be found by running it through a highpass filter.

The highpass filter is given by

$$z_{ff}(t) = k_{hp}\Big(\dot{z}_{sb}(t) - \dot{z}_{sb}(t-1) + z_{ff}(t-1)\frac{\alpha_{hp}}{\alpha_{hp} + dt}\Big) \tag{3.34}$$

where α_{hp} controls the cut-off frequency of the filter, dt is the timestep, and k_{hp} controls the amplitude of the feedforward term. $\dot{z}_{sb}(t-1)$ and $z_{ff}(t-1)$ denotes that previous values for $\dot{z}_{sb}(t)$ and $z_{ff}(t)$, respectively.

A comparison to auto altitude is shown in Figure 3.4. The figure shows the ideal control output when the AUV moves above a bathymetry formed like a sinus wave. The output from the auto altitude controller would be a_r meters above the seabed. The output from the highpass controller should compensate for the seabed when the slope of the seabed changes. When the slope is constant, it should give no compensation. Since the AUV needs time to react when the control input is changing, the depth of the AUV following $z_{d,hp}$ should be closer to the desired depth $z_{d,auto}$.

3.5 Altitude Controller Implementation

The controllers are implemented in DUNE in the task `Control.REMUS.AltitudeFromDvl`.

The configuration parameters available for the task is shown in Listing 3.1.

```
1 [Control.REMUS.AltitudeFromDVL]
2 Altitude Control Method = Highpass
3 Altitude Reference = 10.0
4 Highpass -- Gain = 10
5 Highpass -- Alpha = 3.0
6 Depth Reference = 1
7 Max Distance To Auto = 2
```

Listing 3.1: Example of configuration parameters for the `AltitudeFromDVL` task.

This DUNE tasks supports three difference control methods that must be specified in the `Altitude Control Method` parameter. The three available methods are: `Depth`, `Auto`, and `Highpass`. The `Depth` method was implemented during field

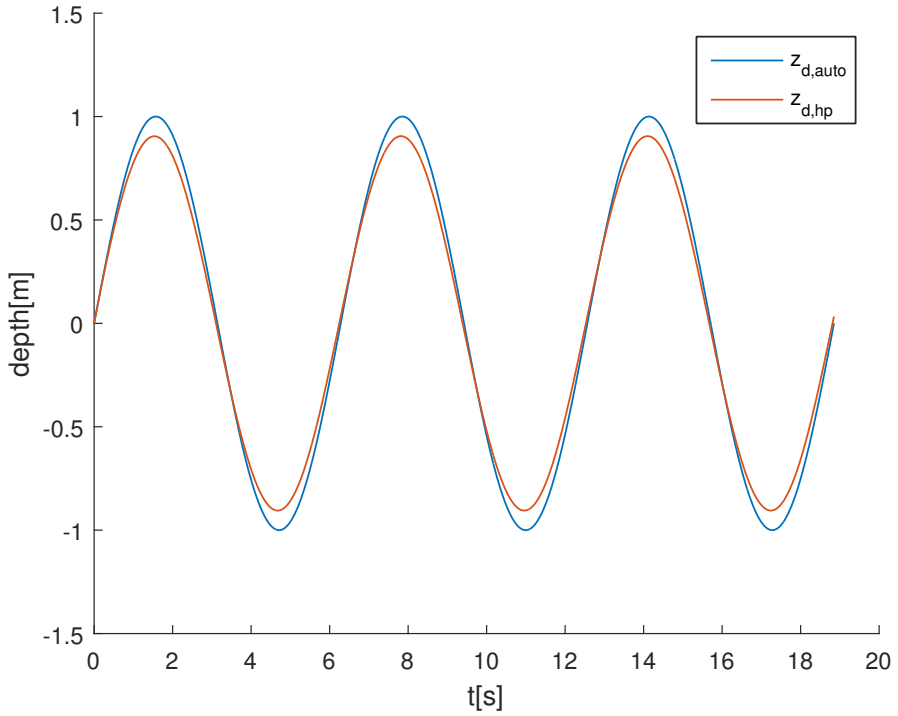


Figure 3.4: Comparison of ideal control output from the auto altitude controller and highpass controller.

testing at Hopavågen. When this method is enabled, it well send a constant depth command specified by the `Depth Reference` parameter.

The `Auto` control method uses the auto altitude controller described in Section 3.4.1. The `Highpass` control method uses the highpass controller described in Section 3.4.2. The highpass control method needs two additional parameters: `Highpass - Gain` and `Highpass - Alpha`. The alpha parameter determines the α_{hp} value in (3.33) and the gain parameter determines the gain k_{hp} in (3.33). For both altitude controllers, the desired altitude is given by the `Altitude Reference` parameter.

Chapter 4

Software

To be able to control the vehicle, DUNE must know the current states of the vehicle like the position $\mathbf{p}$ and the attitude $\mathbf{\Theta}$. Some of the states are only available inside HUGIN SDK and must therefore be made available to DUNE. It is thus necessary to implement software that interfaces HUGIN SDK with DUNE.

This chapter presents the software written to control the REMUS vehicle during the field tests. After the field tests were carried out, the programs were rewritten based on the experiences. The changes to the program design are presented in Chapter 7.

4.1 Mission Execution

The Vehicle Interface Program (VIP) is the graphical user interface used to control REMUS vehicles and to create missions (see Figure 4.1).

A mission is a pre-programmed list of objectives that the REMUS vehicle should follow. When the vehicle is on a mission, it will step through that list of objectives. Examples of objectives are navigating to a new position, updating GPS position, and going up to the surface. The index of the objectives in the mission is called the *Leg*.

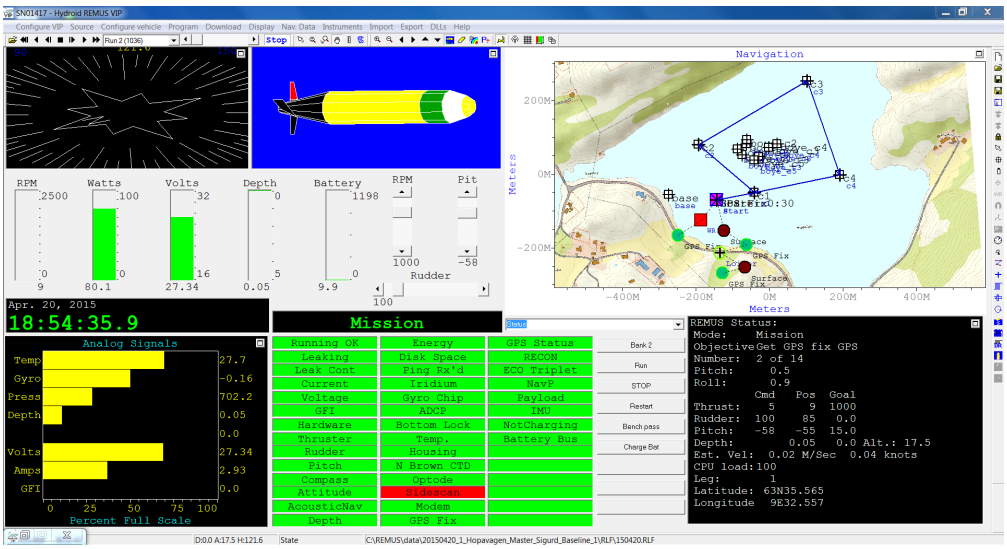


Figure 4.1: The Vehicle Interface Program for REMUS.

4.2 Interfacing with the Vehicle

There are two ways to communicate with the vehicle. The first way is to create a plugin in HUGIN SDK that is run on the PP computer. The plugin must by run through a special program named `PP.exe` (see Figure 4.2). For simulation purposes, there is also a `PP.exe` simulator that can run plugins on a local computer. The second way is to use the Remote Control Interface.

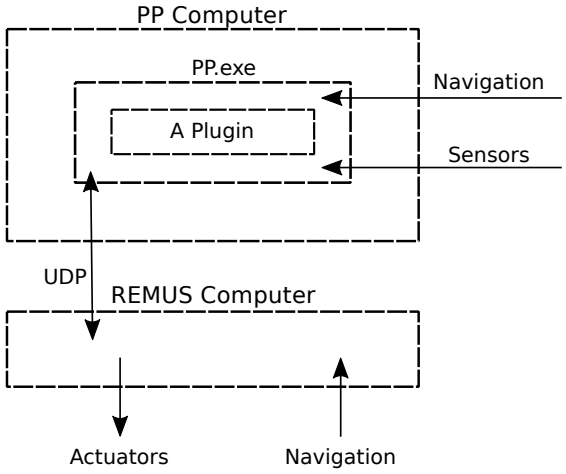


Figure 4.2: Example of a plugin running on the PP computer.

4.2.1 Interfacing with HUGIN SDK

HUGIN SDK enables plugins to listen to updates about the states of the vehicle. The data is grouped in C++ structs. The two most important data groups used in this thesis is `CDvlData` and `CNavSolutionData` that holds the data from the DVL and the navigation system, respectively. The `CPPDataBase` is used to listen to the updates. The code used to listen to navigation updates are given in Listing 4.1. This code will wait for 100 milliseconds for a navigation update. If an update is sent during this time, it will be written to the `navData` variable.

```
CNavSolutionData navData;
CPPDataBase &db = CPPDataBase::getInstance()
db.read(CPPDataBase::ePositionData, navData);
if(db.wait(CPPDataBase::ePositionData, 100))
{
    db.read(CPPDataBase::ePositionData, navData);
    // Do something with navData
}
```

Listing 4.1: Code for listening for new vehicle state updates.

4.2.2 Interfacing with RECON

The Remote Control Interface (RECON) is the only way for programs to send commands to the REMUS computer and change the vehicle's behavior. These commands include setting speed, depth, altitude, next waypoint, fin positions, and propeller RPM (revolutions per minute). After a command has been sent, the REMUS computer will respond with an acknowledge message. If the acknowledge message is not equal to the command message, the command will be rejected.

RECON uses the User Datagram Protocol (UDP) to send messages between the PP and REMUS computers. UDP is a minimal network protocol and a corner stone for network communication. Since UDP is a network protocol, it can be used to send packages between computers on the same network using the server-client model. To send a UDP package, both the server and the client must create a UDP *socket*. The server that is going to receive the packages must then *bind* to a network port. The network port, together with the IP of the computer, will function as an address to the server. The client can then send data packes to this address by using the UDP socket.

When RECON is connected, the REMUS computer will send periodic updates about the state of the vehicle. However, these updates are not equivalent to the updates from HUGIN SDK. The beam ranges from the DVL is only available through HUGIN SDK, and the mission state is only available through RECON. HUGIN SDK supplies a C++ implementation of the RECON interface in the class `CReconDriver`.

Since the REMUS computer is responsible for the control of the vehicle, it is of great importance that the computer is not stressed. Therefore, as a precaution, each call to `CReconDriver` will return after a timeout of 100 milliseconds. This prevents that messages sent to the REMUS computer will be sent too frequently and stress the computer.

If no commands have been sent in the last 5 seconds, the REMUS computer will take back control of the vehicle. This security measure ensures that loss of communication with a plugin does not cause the vehicle to continue using old or obsolete commands. When sending commands to the REMUS computer, it is thus important that the frequency of the commands is not so high that the computer is stressed, and not so low that the mission aborts.

4.2.3 RECON Control Modes

When a program wants to take control over the vehicle, the program must enable one of three control modes. The simplest mode is depth-only mode. In this mode, only depth commands will be transferred to the REMUS computer and the horizontal path will be controlled by REMUS's path controller following the mission plan. The depth may be controlled either by setting a depth or by setting an altitude.

To control heading and speed in addition to the depth, the full-override mode must be enabled. When this mode is enabled, all three control parameters must be set within the 5-second window, or else the vehicle exits RECON control and continues to the next mission objective. As in depth-only mode, the vehicle's depth must be set by a depth reference or by an altitude reference. The speed may be set in meters per second, knots, or by propeller RPM. The heading may be set by a heading angle, heading angular velocity, or by a latitude/longitude goal.

The final mode is the direct-control mode. This mode is for directly setting the propeller RPM and fin positions, and can be used to develop low-level controllers.

These modes cannot be mixed. It is thus not possible to control the depth by setting the stern fin position, and control heading and speed by following the

mission plan.

4.3 HuginDuneBridge

HuginDuneBridge is the plugin created to interface HUGIN SDK with DUNE, and the main contribution through this thesis. The plugin is intended to be a thin translation layer sending sensor and state data from HUGIN SDK to DUNE, receiving control signals from DUNE, and sending the appropriate RECON commands.

HuginDuneBridge is linked to the DUNE core library enabling it to understand the IMC message format and to use other DUNE constructs. It communicates with DUNE by sending and receiving IMC messages using UDP. The DUNE task `Transports.UDP` is responsible for this communication on DUNE.

HuginDuneBridge will be run inside `PP.exe` while DUNE will be run as a standalone program. During a mission, both programs will be run on the PP computer aboard the vehicle as seen in Figure 4.3. During a simulation with AUVSim, the programs will either be on the same computer as the AUVSim (Figure 4.4a) or on the PP computer (Figure 4.4b).

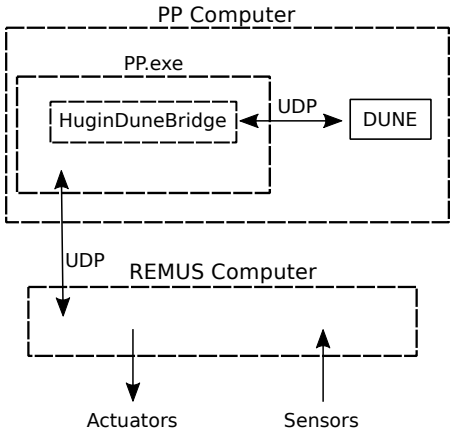
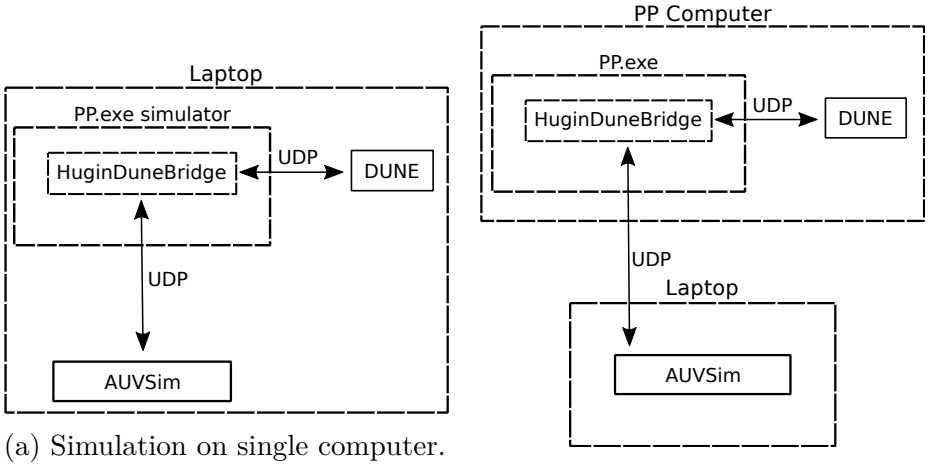


Figure 4.3: Communication between HuginDuneBridge, DUNE, and the REMUS computer.

A third option would be to install a new computer aboard the vehicle and use that computer to run DUNE. The computer will be connected to the same local area Ethernet, and because DUNE and HuginDuneBridge uses UDP, they will still be able to communicate. One advantage of running DUNE on a separate computer



(a) Simulation on single computer.

(b) Simulation with DUNE on the PP computer.

Figure 4.4: Communication flow for HuginDuneBridge running in simulation on a local laptop or on REMUS.

is that it gives a separation of concerns. The PP computer has a 1.6 GHz Intel Atom CPU with two cores, and due to the low clock rate, this makes it hard to use advanced control strategies like solving optimization problems in real-time. Using a new and faster computer will allow running such computational demanding tasks.

Another advantage is that the new computer can run the Linux operating system. Although DUNE compiles on Windows, it is more frequently tested on Linux. How the communication would be with the new computer is shown in Figure 4.5

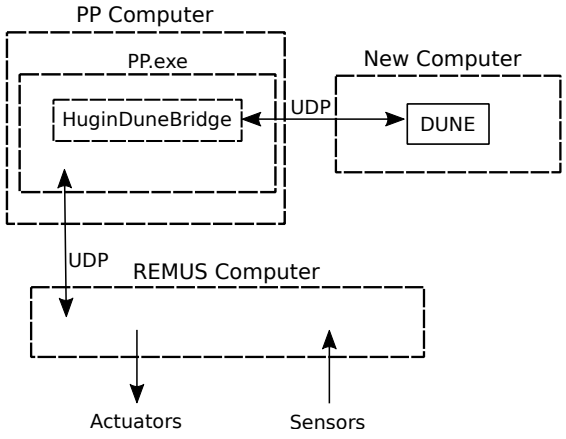


Figure 4.5: Running DUNE on a separate computer

4.3.1 Architecture

Since HuginDuneBridge must listen to data from DUNE, HUGIN SDK, and RECON, it is convenient to use multiple threads. Communication between these threads is done either by method calls guarded by synchronization locks, or by using a local IMC bus to send IMC messages. All threads in HuginDuneBridge are DUNE tasks using the *subscribe* and *broadcast* constructs described in Chapter 2.

The communication is split into a receiving part, and a sending part (see Figure 4.6). The control flow for the receiving part is shown in Figure 4.7.

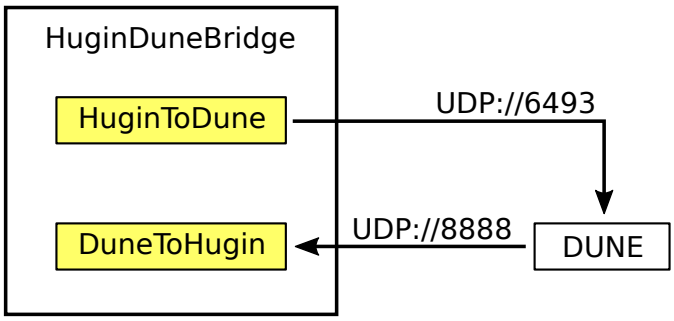


Figure 4.6: Network architecture for HuginDuneBridge and DUNE.

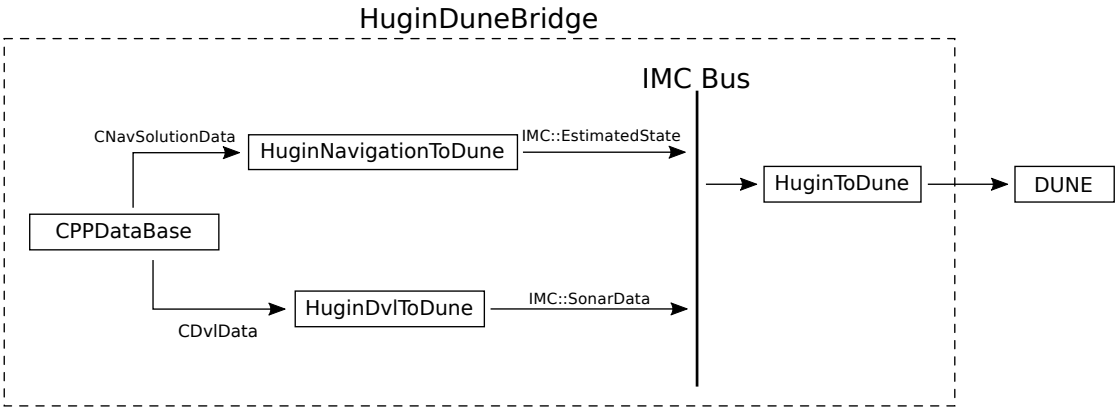


Figure 4.7: Control flow from CPPDataBase in HuginDuneBridge to DUNE.

When it comes to the receiving part, we have chosen four RECON commands to be controllable: speed, depth, and heading by setting an angle or a goal destination. For depth control, the task listens to `IMC::DesiredZ`. For speed control, it listens to `IMC::DesiredSpeed` and `IMC::ManeuverControlState`. The last one is necessary to detect when a maneuver is done, so that the speed can be set to zero. For heading control, it listens to `IMC::DesiredHeading` and for setting the next waypoint, it listens to `IMC::DesiredPath`.

Not all RECON commands should be available at all times. If a mission is programmed such that DUNE is controlling the depth only, then other commands should be rejected. A supervisor layer is thus necessary to control which RECON commands should be enabled and to enable them at the correct time. This has been implemented in the `ReconSupervisor` class. The current mission objective, the *leg*, is used to control when the commands should be enabled. When the *start* objective is reached, `ReconSupervisor` will enable control, and will have control until the *end* objective is reached. The *start* and *end* objectives are set in the configuration file for HuginDuneBridge (see Listing 4.4). For the field tests described in Chapter 6, the supervisor was configured to enable commands at objective 2 and disable commands at objective 7.

To control how often RECON commands are sent to REMUS, all commands are routed through the `ReconProxy` class. When a command is first sent to `ReconProxy`, it will be sent every second until REMUS takes back control. If a command is sent to frequently, it will act as a zero-order-hold and only send the latest received command at each iteration.

`ReconProxy` is the class that ensures that the REMUS computer is not stressed by receiving commands too frequently, and that it will not take back control if the time between commands exceeds 5 seconds. A drawback of this approach is that if DUNE should fail, old and outdated commands will still be sent to REMUS. This has been improved in the new architecture described in Chapter 7. A control flow diagram of the receiving part can be seen in Figure 4.8.

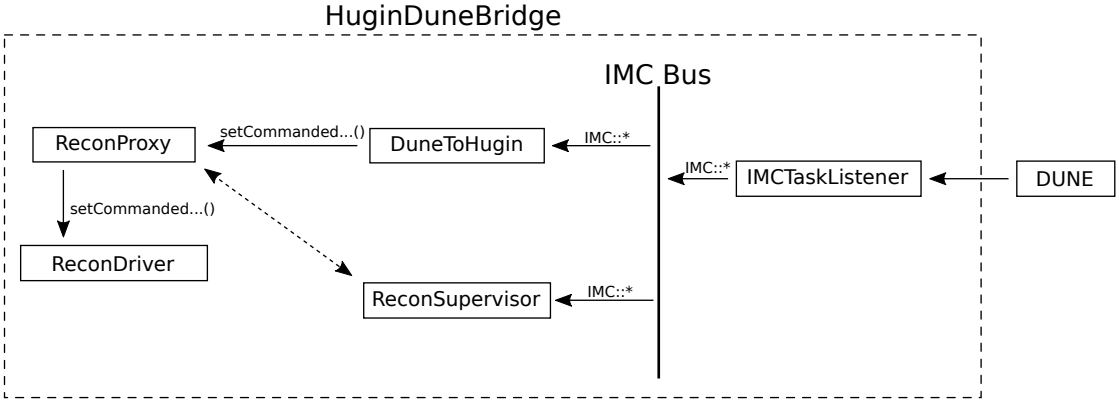


Figure 4.8: Control flow from DUNE to HUGIN

4.3.2 Connecting to AUVSim

This section documents how the HuginDuneBridge is connected to the AUVSim simulator described in Section 4.4.

AUVSim communicates with HuginDuneBridge by use of the UDP protocol (see Figure 4.9). The `AuvsimToHugin` class is responsible for receiving UDP packages from AUVSim. The format of the message is given by `FromAuvsimState` listed in Listing 4.2.

```
struct FromAuvsimState {
    double t, dt;
    double x, y, z, phi, theta, psi;
    double u, v, w, p, q, r;
    double dvlUpRange[4];
    double dvlDownRange[4];
};
```

Listing 4.2: Data format of packages sent from AUVSim to HuginDuneBridge.

After a UDP package has been received, the data is put into the `CPPDataBase`. Since `HuginNavigationToDune` and `HuginDvlToDune` is listening for changes in `CPPDateBase`, the data will be sent to DUNE immediately.

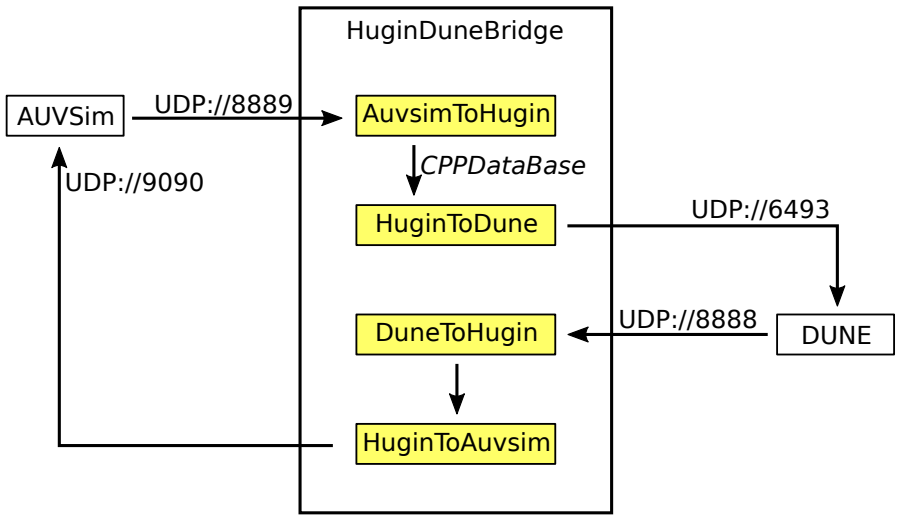


Figure 4.9: Network architecture for HuginDuneBridge, DUNE, and AUVSim

Ideally all data received from AUVSim would be written to the `CPPDataBase` such that all data flow through the HUGIN interface. However, some data do not have

any storage type in HUGIN. An example is the current simulation time `t` and the simulation timestep `dt` that is needed by DUNE to compute integrals. Therefore, `AuvsimToHugin` has a reference to `HuginToDune` to send additional data through method calls. These method calls are guarded by synchronization locks to avoid race conditions.

For the same reasons, `DuneToHugin` will need to hold a reference to `HuginToAuvsim`. After `DuneToHugin` has received the messages from DUNE, it signals to `HuginToAuvsim` to send an update message to AUVSim. The format of the message is given by `ToAuvsimState` listed in Listing 4.3.

```
1 struct ToAuvsimState {
2     double t;
3     double z_ref;
4     double altitude;
5     double z;
6 };
```

Listing 4.3: Data format of packages sent from HuginDuneBridge to AUVSim.

4.3.3 Run Configurations

HuginDuneBridge is configured with a configuration file in ini format (see Listing 4.4). The configuration file is split in a network part and a supervisor part. The network part specifies what IP addresses and ports that should be used. The supervisor part specifies what modes should be enabled and at which objective they should be enabled.

The configuration file in Listing 4.4 enables sending of commands at objective 2, and disables sending at objective 7. The mode is set to full-override since both depth, speed and waypoint heading is enabled. Messages that set heading by an angle will be rejected, while messages that set heading by a waypoint goal will be allowed.

```
[Network]
DUNE Incoming Port = 8888
DUNE Outgoing Port = 6493
DUNE Outgoing IP = 192.168.1.50
AUVSim Incoming Port = 8889
AUVSim Outgoing Port = 9090
AUVSim Outgoing IP = 192.168.1.100
RECON IP = 192.168.1.44

[ReconSupervisor]
Enable At Objective = 2
Disable At Objective = 7
Depth Enabled = 1
Speed Enabled = 1
Heading Enabled = 0
Waypoint Enabled = 1
```

Listing 4.4: Configuration file for HuginDuneBridge.

4.3.4 Standby Mode

When DUNE is in control of heading and speed, and have not started a DUNE mission, the vehicle is set in standby mode. The standby mode is a state where the commanded depth, heading and speed is set to zero. The mode is necessary so that the REMUS computer does not take back control after 5 seconds of inactivity.

4.4 AUVSim

AUVSim is a simulator for REMUS 100 created by PhD candidate Petter Norgren (Norgren and Skjetne, 2015). The simulator is implemented in MATLAB and Simulink. The top level block diagram can be seen in Figure 4.10. The parameters for the simulator have been taken from a previous REMUS 100 model by Prestero (2001). The connection to HuginDuneBridge is implemented as a C++ sFunction inside the *Guidance system* block.

HuginDuneBridge translates the messages from AUVSim into multiple IMC-messages, and DUNE is responding by sending multiple IMC-messages back. As a result, the sFunction may receive more than one package in some iterations, and if the timing is unfortunate, no packages in other iterations.

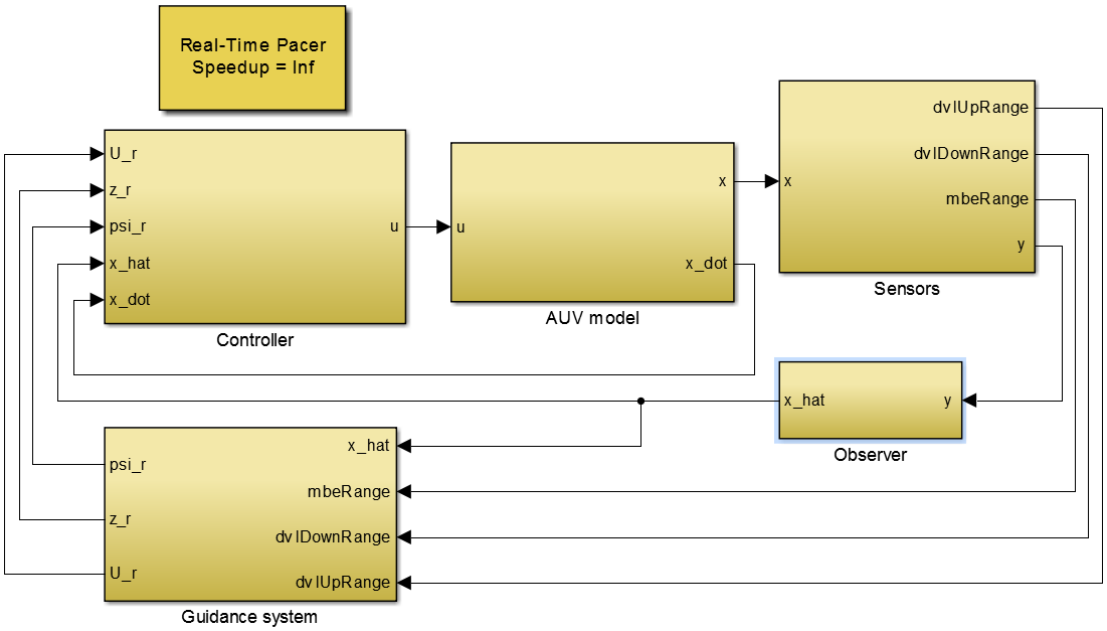


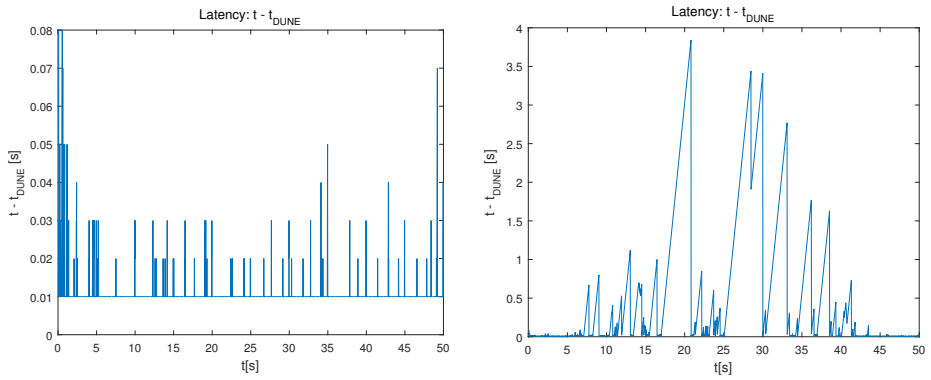
Figure 4.10: Block diagram of the AUVSim simulator.

The first implemenation of the sFunction had a single blocking receive call waiting for the next IMC message. Due to the uncertainty of how many packages will be received each iteration, the sFunction's performance was unpredictable. If no packages were received, it would go into a deadlock state where both HuginDuneBridge and the sFunction were listening and no one was sending. If more than one package were received, the queue of incoming packages would stack up, and introduce a growing time delay between the two applications.

This problem was solved by introducing a new thread in the sFunction that listens for new packages. On each iteration, the sFunction returns the last received package. This introduces a time delay since HuginDuneBridge and DUNE do not have time to perform their actions between a send and receive. However, since DUNE performs high-level control, this time delay is so small that it does not affect the simulation.

Since the sFunction only returns the last received message, it is prone to time delay problems if AUVSim runs faster than HuginDuneBridge and DUNE. To solve this problem a `Realtime Pacer` block (Gautam Vallabha, 2015) was introduces to the Simulink program. This block can slow down the simulation time so that the simulator is not speeding past HuginDuneBridge and DUNE. The amount of slowdown is controlled by a speedup parameter. If the speedup is set to 1, the simulator runs in real time.

To be able to measure the delay between AUVSim and DUNE, the simulation time t is sent through the control chain. The difference between the simulation time used by AUVSim and the simulation time returned from HuginDuneBridge will indicate if AUVSim is speeding past HuginDuneBridge and DUNE. An example of a good network delayed can be seen in Figure 4.11a. As seen in the figure the lowest network delay is 0.01 which is the same as the timestep in this simulation. An example of AUVSim running faster than the other components can be seen in Figure 4.11b.



(a) Good network delay.

(b) Bad network delay.

Figure 4.11: Network delay between AUVSim and DUNE. The timestep dt is 0.01 seconds.

4.5 Simulation of Beam Ranges

Section 3.4 presents two altitude controllers using the range measurements from a DVL as input. To perform simulations of these controllers, it is thus necessary to be able to simulate DVL range measurements. A beam range simulator has therefore been developed. This section is an excerpt from the preceding project thesis (Holsen, 2014).

4.5.1 Theory

By using the same beam model as in Section 3.3, the vector describing the direction of each beam is given by

$$\frac{\mathbf{r}_i}{|\mathbf{r}_i|} = \boldsymbol{R}_b^n \mathbf{R}_{DVL_i}^b \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T \tag{4.1}$$

where $\mathbf{r}_i$ is the position vector representing the i^{th} beam. We will assume that the position and attitude of the vehicle as well as the direction of the DVL beams are known.

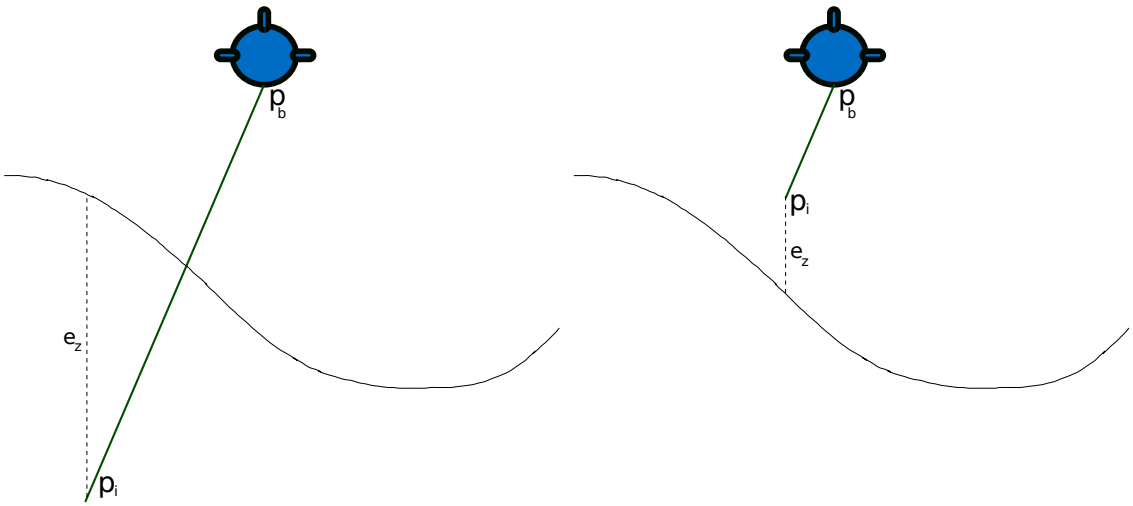
The next step is to determine the correct length of each beam. Let e_i be the estimated length of the i^{th} beam. Let $\mathbf{p}_b = [x, y, z]^T \in \mathbb{R}^3$ be the position of the vehicle in the NED frame. Finally, let $\mathbf{p}_i = [x_i, y_i, z_i]^T \in \mathbb{R}^3$ be the estimate of the end position of the i^{th} beam in the NED frame. This gives the following relationship

$$\mathbf{p}_i = \mathbf{p}_b + \frac{\mathbf{r}_i}{|\mathbf{r}_i|} \cdot e_i. \tag{4.2}$$

The relationship may be used to indicate whether the estimate of the beam length is too large or too small. Let the depth at position (x, y) be given by $Z(x, y)$. We can now find the distance between the beam end position $\mathbf{p}_i$ and the seabed. The distance is given by

$$e_z = z_i - Z(x_i, y_i) \tag{4.3}$$

If the error is positive, the beam range is too small, while a negative error tells us that the beam range is too large (see Figure 4.12). Binary search is used to reduce the error to a sufficient accuracy.



(a) Negative error (b) Positive error

Figure 4.12: Range error for the beam simulator.

The data structure chosen for the map data is a matrix where each cell represents the depth at a specific position. To get the depth $Z(x, y)$ there must be a mapping

from coordinates given in the NED frame to the cells in the map matrix. The size and resolution of the map is given by the `Bounds`-structure. The `Bounds`-structure consists of four values: `northMin`, `northMax`, `eastMin`, `eastMax`. The north and east index will then be given by

$$percentageDone = \frac{x - min}{max - min} \tag{4.4}$$

$$x_{index} = x_{size} \cdot percentageDone \tag{4.5}$$

This formula applies to both north and east where x is the position and min and max are the minimum and maximum values specified in the `Bounds`-structure. x_{size} is the number of rows or column in the map matrix in that direction.

The resulting north and east index will be a real number, that is $x_{index} \in \mathbb{R}$. To get the depth between grid points, bilinear interpolation is used between the four nearest neighbours.

4.5.2 Implementation

The class diagram for the beam range simulator can be seen in Figure 4.13.

The simulator has been separated into two classes: `Map` and `BeamRangeSimulator`. The main responsibility for the `Map` class is to calculate the depth $Z(x, y)$ given a coordinate. This is done through the `Map::getDepthAtCoordinates` method. `Map` implements the bilinear interpolation and conversion from coordinates to index values.

The `BeamRangeSimulator` class implements the binary search algorithm used to estimate the beam lengths. Another important feature is the possibility to use different beam configurations. This allows the user to simulate a Doppler Velocity Log (DVL) or a Multibeam Echosounder (MBE). The `BeamConfiguration` struct gives the spread of the beams and the number of beams.

The estimate is returned as a `BeamRangeResult`. The `beamRanges` vector contains the computed ranges and the `hitPositions` contain the corresponding $\mathbf{p}_i$ positions.

The Beam Range Simulator have been simulated to ensure that the implementation gives satisfactory results. The simulation results are shown in Section 5.2.

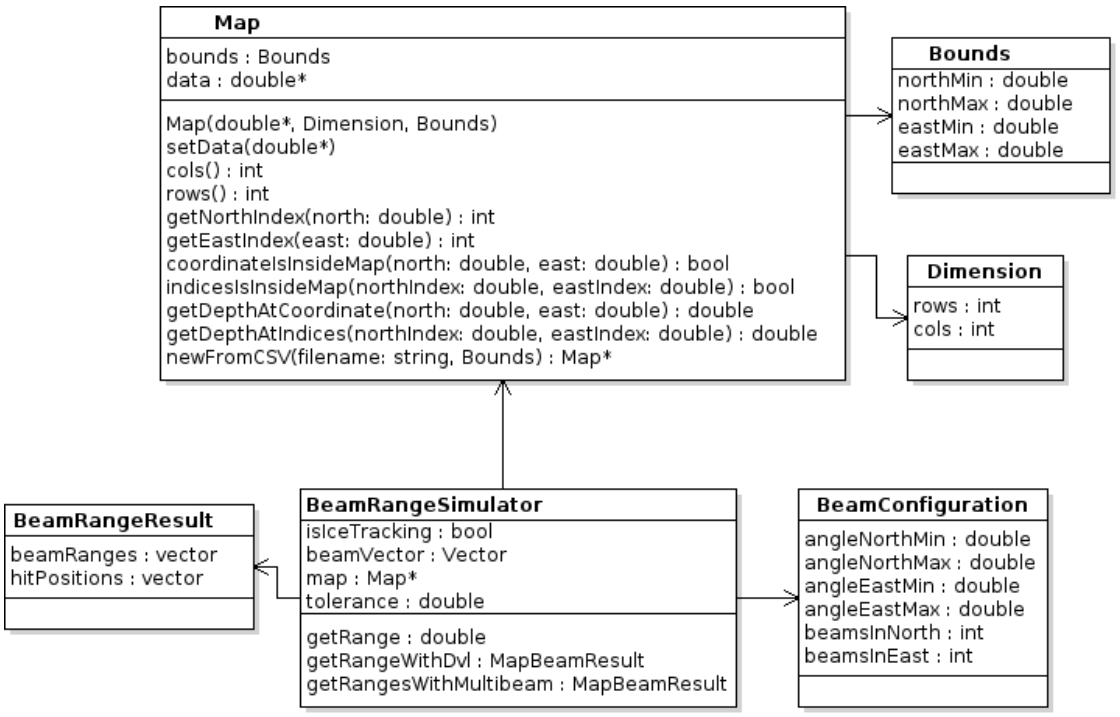


Figure 4.13: Class diagram for the Beam Range Simulator.

Chapter 5

Simulation

This chapter presents the simulations using the controllers presented in Chapter 3 and the software presented in Chapter 4. The altitude controllers that have been simulated are the auto altitude and highpass controllers. In addition, heading control has been simulated using the heading controllers in DUNE and AUVSim. All simulations have been run with the AUVSim simulator.

5.1 Simulation Maps

Three maps representing different bathymetric conditions were used for the simulations.

5.1.1 Constant Slope

The constant slope map is shown in Figure 5.1. The map is used to test the altitude controllers going up a constant slope.

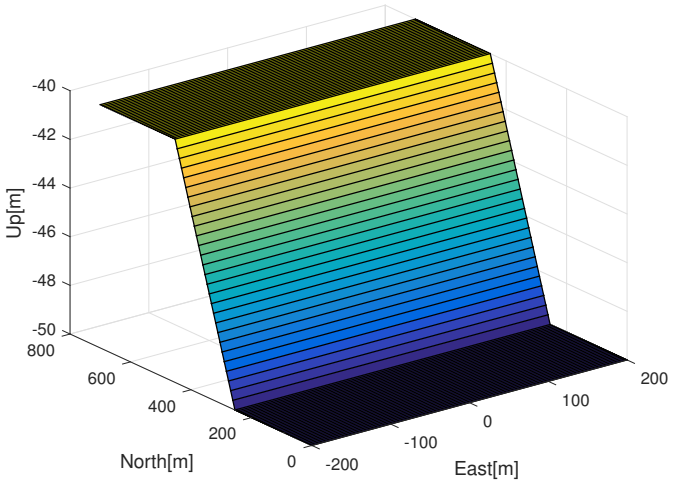


Figure 5.1: The constant slope map.

5.1.2 Sinus Curved

The sinus curved map shown in Figure 5.2 is a sinus curve in the north-south direction. The map is used to test the altitude controllers' performance when following a curved seabed.

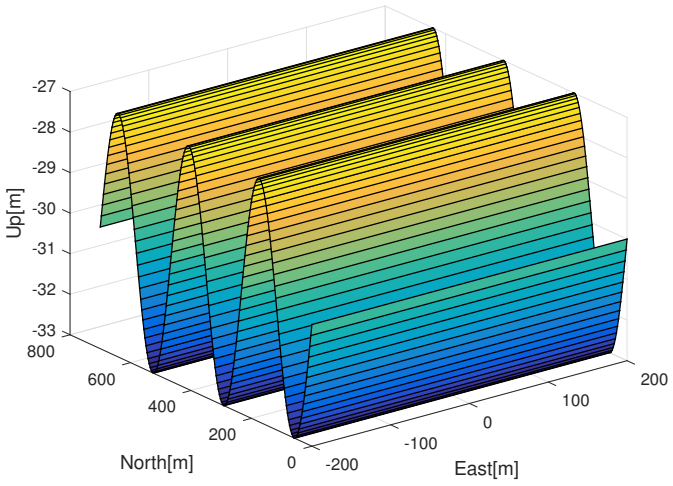


Figure 5.2: The sinus curved map.

5.1.3 Hopavågen

A map of Hopavågen has been created by PhD Candidate Pierre-Yves T Henry at AMOS NTNU and Mancheño (2014). The map is generated using datasets from previous missions at Hopavågen and gives a low fidelity estimate of the bathymetry. The bathymetry is shown in Figure 5.3 with the shore line represented by the thick blue line. Ideally the shore line would be aligned with the contour lines in the plot, but since it is a low fidelity approximation it does not. However, it gives a good approximation of the seabed slope.

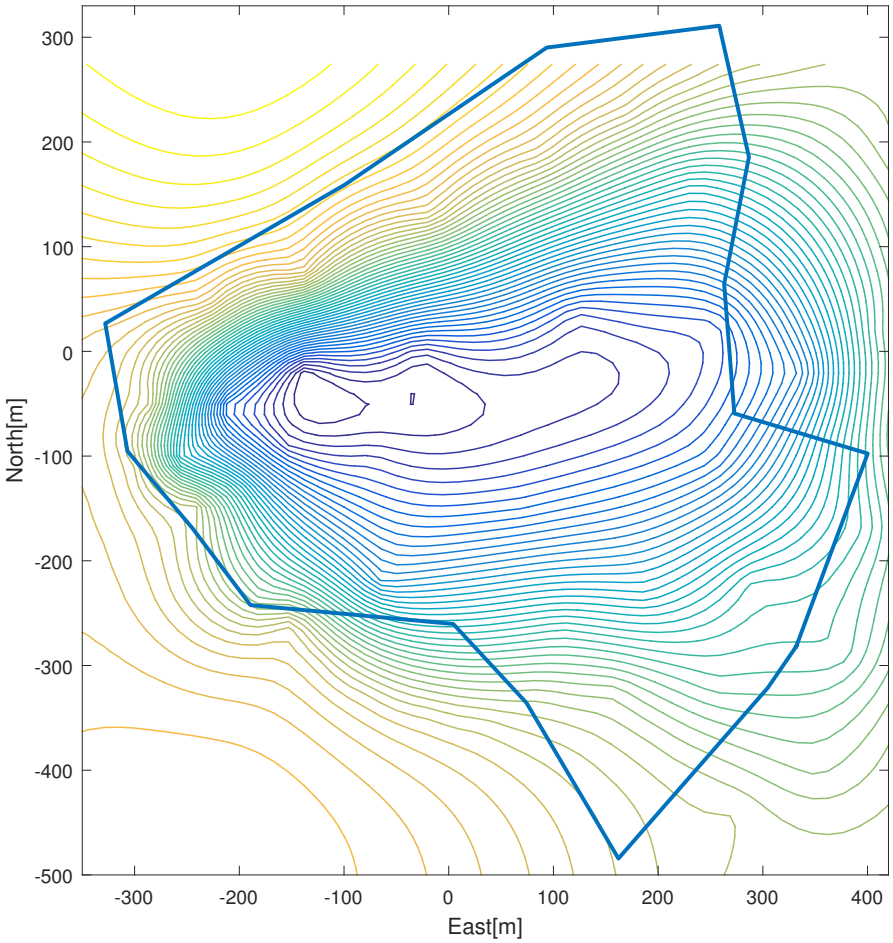


Figure 5.3: Contour plot of the bathymetry at Hopavågen. The lowest point is 28 meters deep. The thick blue line represents the shore line.

5.2 Beam Range Simulator

A number of simulations have been performed to ensure that the implementation of the Beam Range Simulator described in Section 4.5 is correct. This is a continuation of the simulations done in the project thesis (Holsen, 2014).

5.2.1 Visual Confirmation That the Beams Hit the Sea Bed

An AUV was positioned in the middle of the map with a nonzero pitch and roll. The beam range simulator was used to find the position where the DVL beams would hit. The beams are then plotted as seen in Figure 5.4. The plot can now be used to verify the length of the beams. Each beam should touch the seabed, but not go through it.

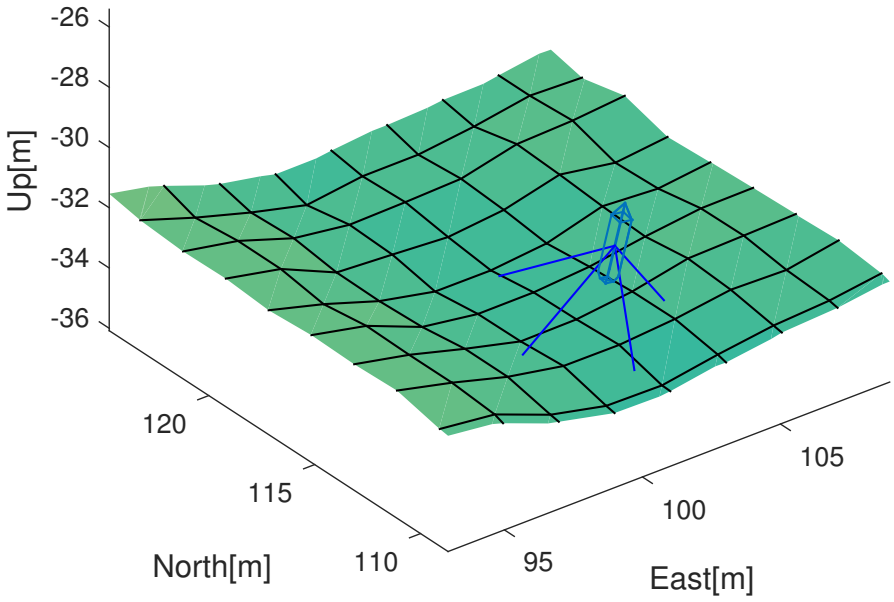


Figure 5.4: Visualizing the DVL beams from an AUV. The beams should barely touch the seabed.

5.2.2 Altitude Control of an AUV Across the Sea Bed

This simulation uses both the beam simulator and the altitude estimator described in Section 3.3.

An AUV was sent along a straight path over the Hopavågen seabed. The AUV was moved in a fixed path with a constant altitude of 3 meters. At each step, the beam simulator would estimate the range measurements. The range measurements was then sent to the altitude estimator. After the altitude was found, the sea depth could be estimated by adding the AUV depth to the estimated altitude. The sea depth estimated could then be compared to the real sea depth.

Estimated sea depth and true sea depth is plotted in Figure 5.5. The estimation error is plotted in Figure 5.6. As seen in the figures, the error is small. The reason for the deviation is that the seabed is nonlinear, and it can therefore not give a perfect estimate using the planes described in Section 3.3. Since the altitude estimator uses four range measurements around the vehicle, it is expected that the estimated altitude should be smoother than the real altitude. Looking closer at Figure 5.5, one can see that this is the case.

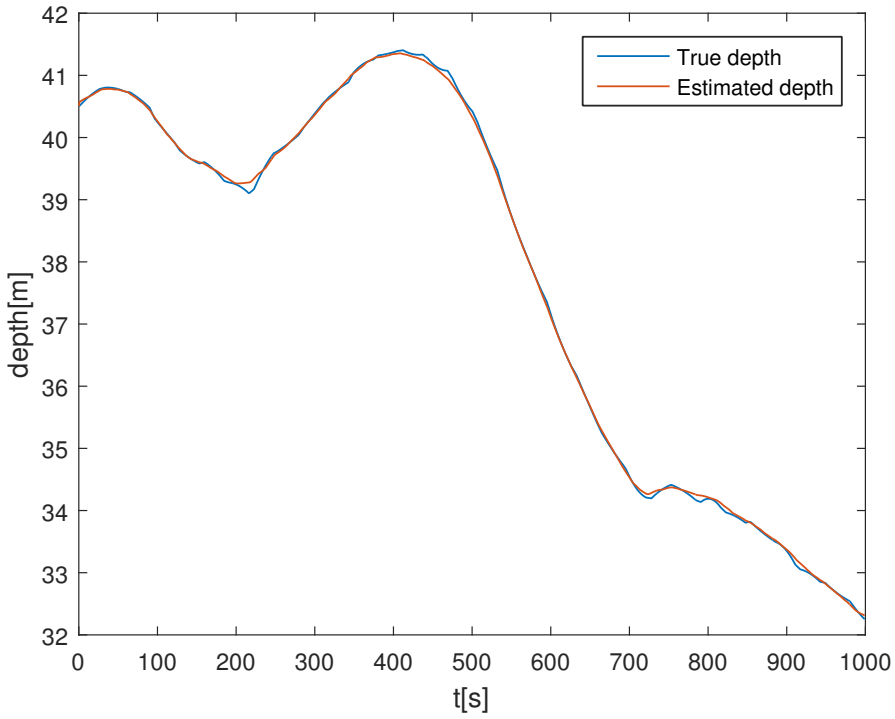


Figure 5.5: Estimated depth and true depth of the seabed recorded along an AUV trajectory.

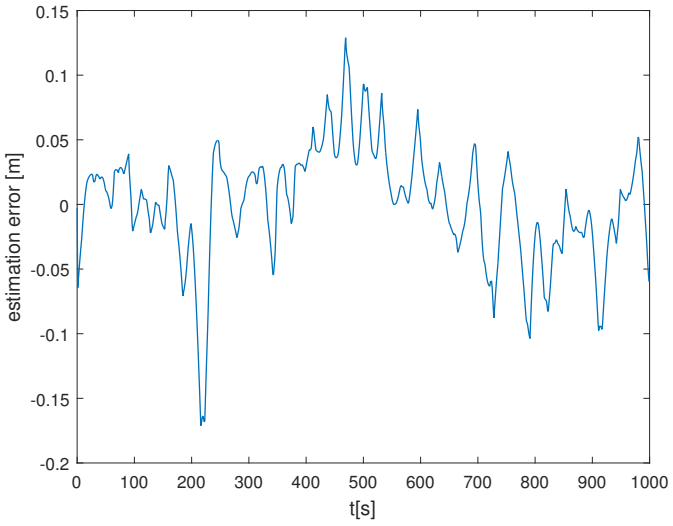


Figure 5.6: Depth estimation error of the seabed recorded along an AUV trajectory.

5.3 Tuning the Depth Controller

The altitude controllers developed in Chapter 3 uses the depth controller on REMUS for the low-level control. It is therefore important that the depth controller in AUVSim is similar to the depth controller on REMUS. Using mission data from previous missions as input, the AUVSim depth controller has been tuned so that the simulation results should resemble real results. The desired depth z_d from the mission data was sent as input to the depth controller in AUVSim. The real mission depth z_{remus} could then be compared to the simulated depth z_{auvsim}.

The mission that was used for the tuning was performed near Ny-Ålesund, Spitsbergen January 20, 2015. Figure 5.7 shows a plot of the depth z_{remus} and desired depth z_d for the mission as well as the simulated depth z_{auvim}. It shows two depth control challenges: slope-following and big steps in z_d.

Figure 5.8 shows the response from the REMUS depth controller when it received a step in z_d. The figure indicates that there is a maximum diving and surfacing velocity, and that the surfacing velocity (see $t = 8400$) is greater than the diving velocity (see $t = 7500$). The maximum diving velocity was found to be around $0.21^m/_s$ and the maximum surfacing velocity was found to be around $0.68^m/_s$.

Based on this information, a guidance block was added in front of the AUVSim depth controller to guarantee that the velocity will not exceed this maximum. The block receives the depth reference z_r and outputs the desired depth z_d. If the input would result in a step bigger than what the maximum velocity would allow, the

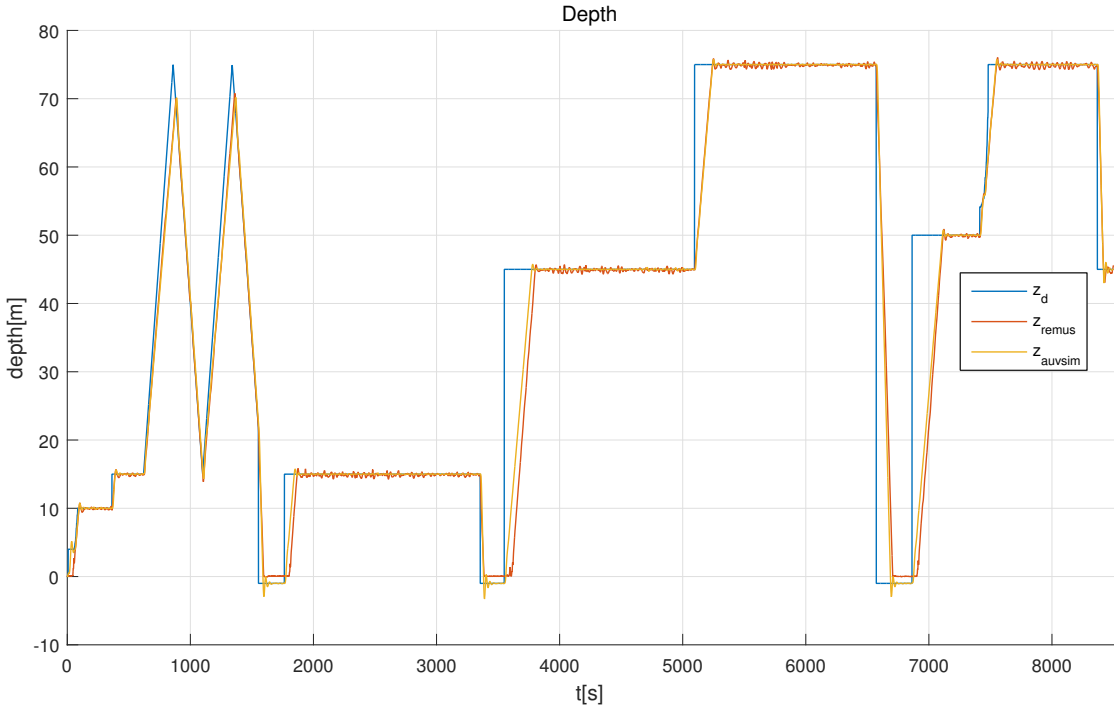


Figure 5.7: Depth reference z_d, depth z_{remus}, and simulated depth z_{auvsim} for a mission in Ny-Ålesund, Spitsbergen January 20, 2015.

output would change linearly towards the input with the maximum velocity. The behavior of the guidance block can be seen in Figure 5.9.

A second or third order reference model could also have been used to solve this problem (Fossen, 2011, pp. 249-251). A benefit of using a reference model is that it can give a smooth transition between the point where it is able to follow the signal and the point where the signal jumps. However, a reference model will introduce a delay between the z_r and z_d, and no such transitions can be observed in the previous missions (see Figure 5.8). It was therefore concluded that REMUS does not employ a reference model, and it could therefore not be added to the simulator.

Table 5.1: Simulation coefficients for the AUVSim PI depth controller.

Parameter	Value
k_p	0.12
k_i	0.005
$i_{max,z}$	10.0

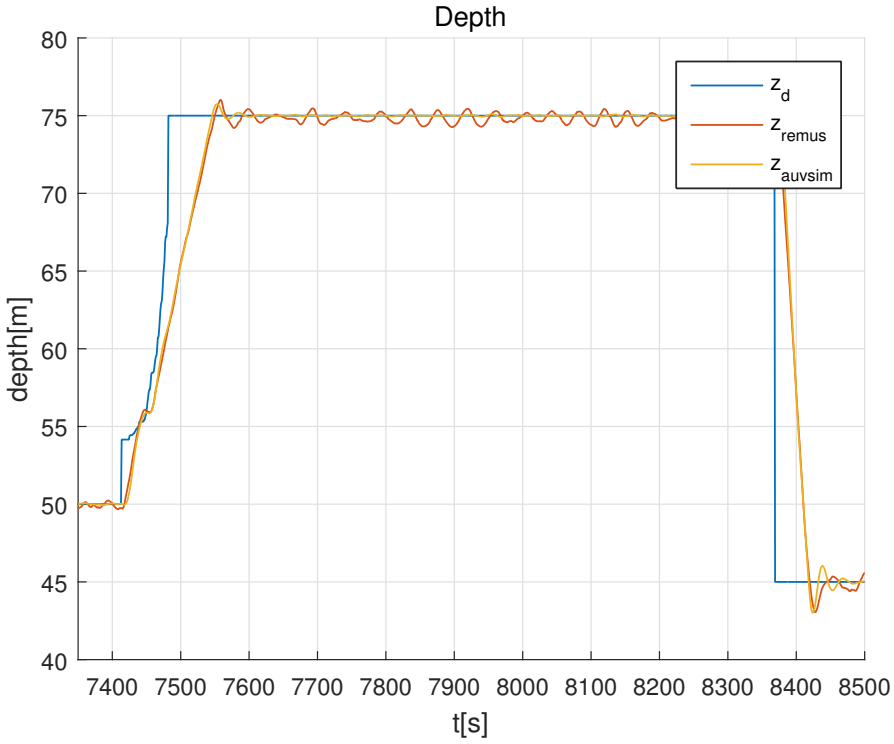


Figure 5.8: Depth reference z_d, depth z_{remus}, and simulated depth z_{auvsim} for a mission in Ny-Ålesund, Spitsbergen January 20, 2015.

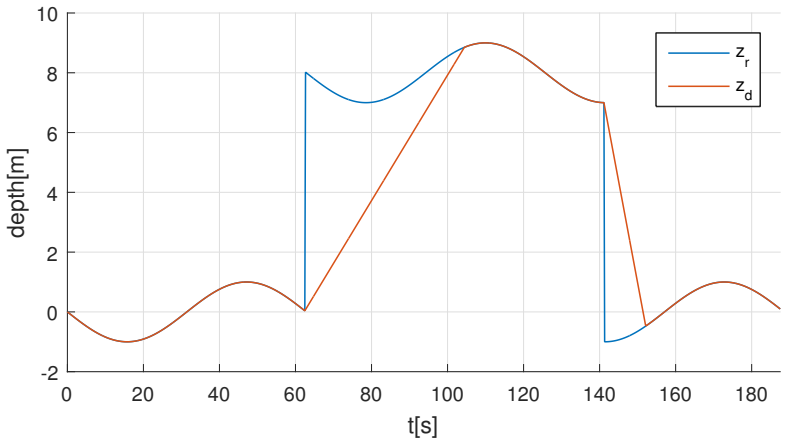


Figure 5.9: Example of input z_r and response z_d for the depth guidance block.

5.4 Altitude Control

Three scenarios for altitude control have been tested, first by using the auto altitude controller, and then by using the highpass controller. The first scenario uses the linear slope map with a path that moves in a direct line from south to north. The second scenario uses the sinus curved map and follows the same path. The third scenario uses the Hopavågen map with the path shown in Figure 5.10. All scenarios have set the altitude reference a_r to 10 meters.

The simulation results presented in this section uses the depth controller tuned in Section 5.3. Using a depth controller that resembles the one used on the REMUS AUV should give results that are similar to that of the real system. However, to best show the effect of the highpass controller, the depth controller should be as good as possible. Appendix C shows simulations using a PID controller instead of a PI controller to control the depth.

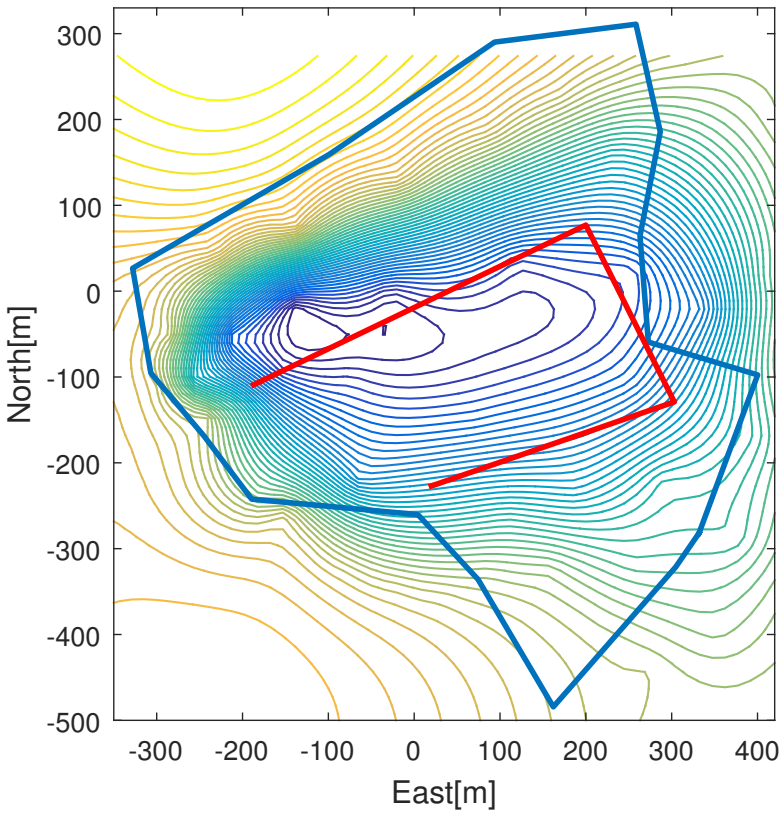


Figure 5.10: AUV path for the Hopavaagen simulation scenario. The thick blue line represents the shore line while the thick red line represents the AUV path. The start position is west of the deepest point.

The figures that will be presented in this section includes three plots: z shows the vehicles depth, z_r shows the desired depth sent to the AUVSim depth controller, and $z_{perfect}$ shows the depth 10 meters above the seabed.

5.4.1 Scenario: Constant Slope Map

The first scenario uses the constant sloped map. A simulation using the auto altitude controller is shown in Figure 5.11, and a simulation using the highpass controller is shown in Figure 5.12. The auto altitude controller gives a maximum altitude error of around 0.5 meters while the highpass controller gives a maximum altitude error of around 0.3 meters.

The reduction is caused by the increased error $z_d - z$ at $t = 120$ s and $t = 225$ s. The greater error increases the proportional gain in the depth controller resulting in a faster response. As seen in the error plot, there are two consecutive peaks after the slope changes. The second peak is caused by the sudden reduction in the altitude error after the highpass controller compensation has converged. The proportional gain from the controller is then reduced, and the integral gain needs time to increase resulting in the second peak.

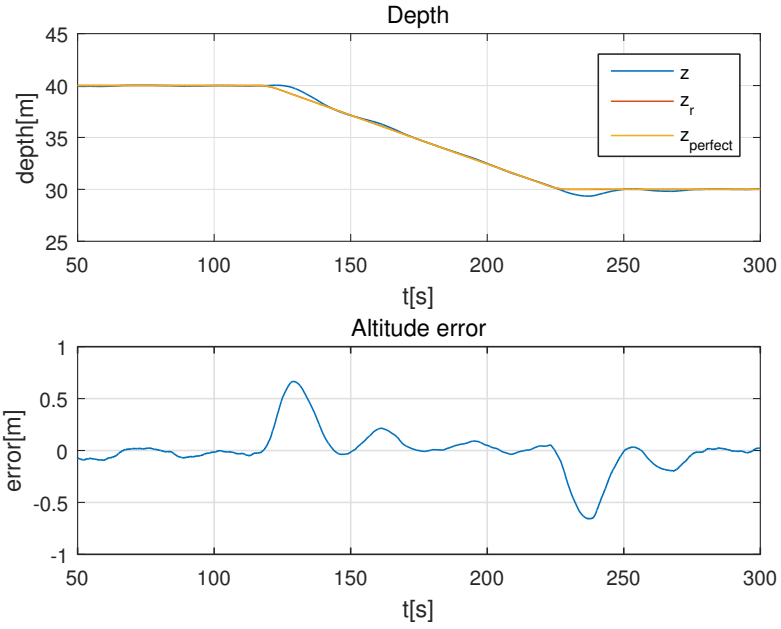


Figure 5.11: Simulation using the auto altitude controller and the constant slope map.

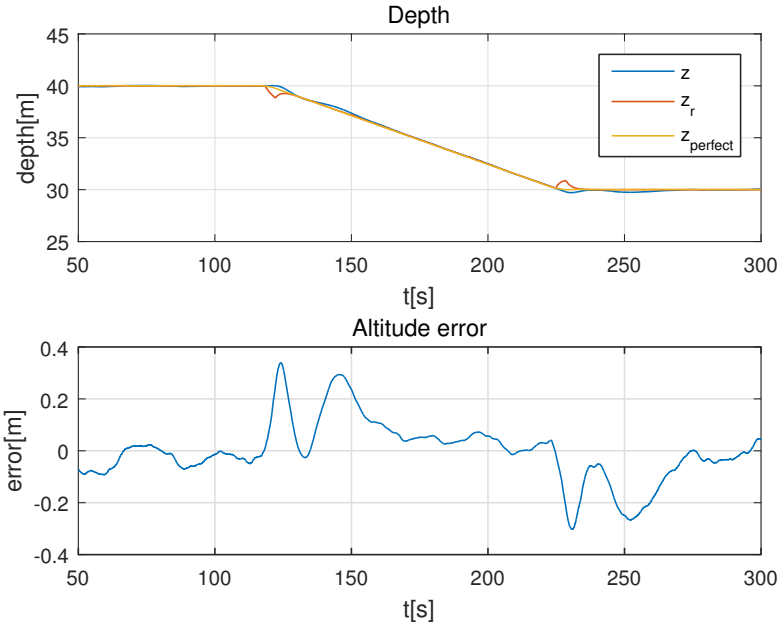


Figure 5.12: Simulation using the highpass controller and the constant slope map.

5.4.2 Scenario: Sinus Curved Map

The second scenario uses the sinus curved map to see how the AUV behaves when going over a curved seabed. Figure 5.13 shows a simulation using the auto altitude controller while Figure 5.14 shows a simulation using the highpass controller. As can be seen in the figures, the maximum error is reduced from around to 0.6 meters to 0.4 meters when using the highpass controller. This error could have been further improved by using a PID depth controller (see Appendix C).

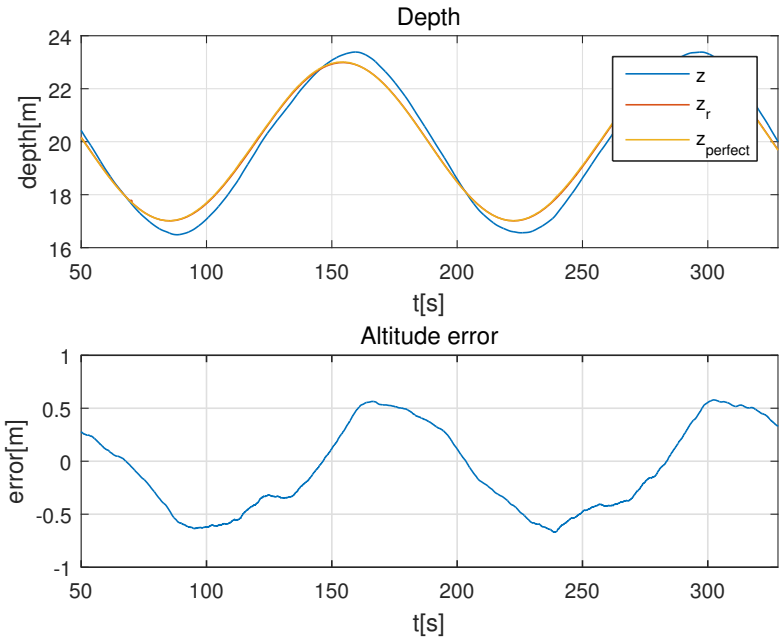


Figure 5.13: Simulation using the auto altitude controller and the sinus map.

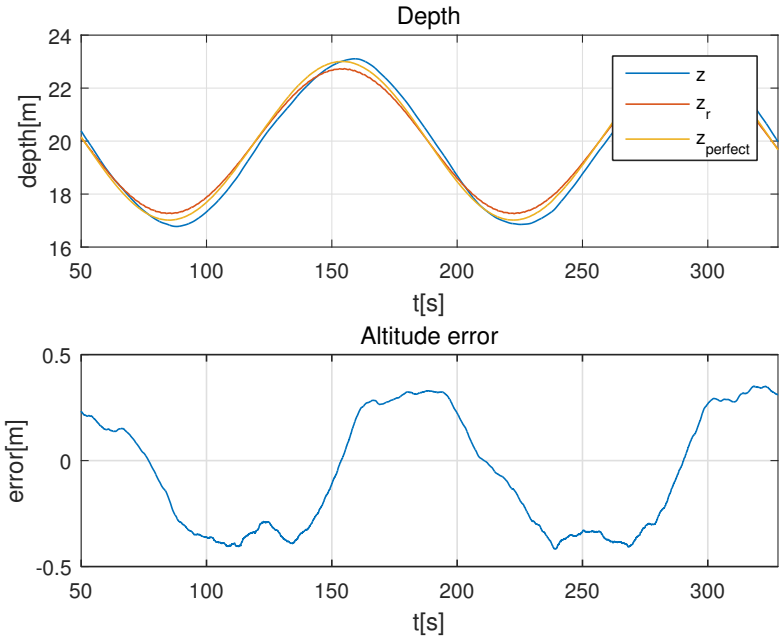


Figure 5.14: Simulation using the highpass controller and the sinus map.

5.4.3 Scenario: Hopavågen Map

The final scenario uses the Hopavågen map. The simulation using the auto altitude controller can be seen in Figure 5.15 while the simulation using the highpass controller can be seen in Figure 5.16. This is the scenario with the greatest difference between the two controllers. The maximum error is reduced from 1.5 to 0.8 by using the highpass controller.

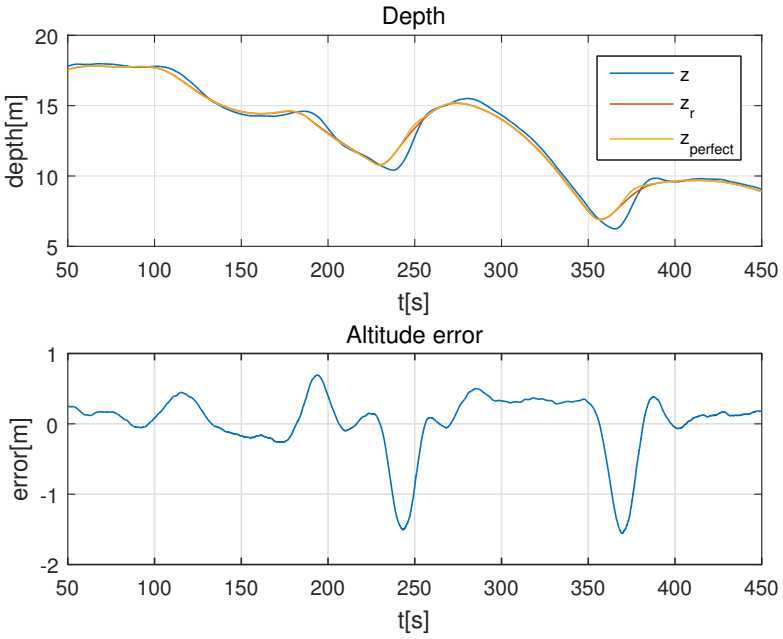


Figure 5.15: Simulation using the auto altitude controller and the Hopavågen map.

A downside with using the highpass controller is that it introduces two new parameters that are dependent both on the depth controller of the vehicle and on the underlying seabed. If a new section is added to the front of the AUV such that the depth controller need to be re-tuned, the parameters for the highpass controller must be re-tuned as well.

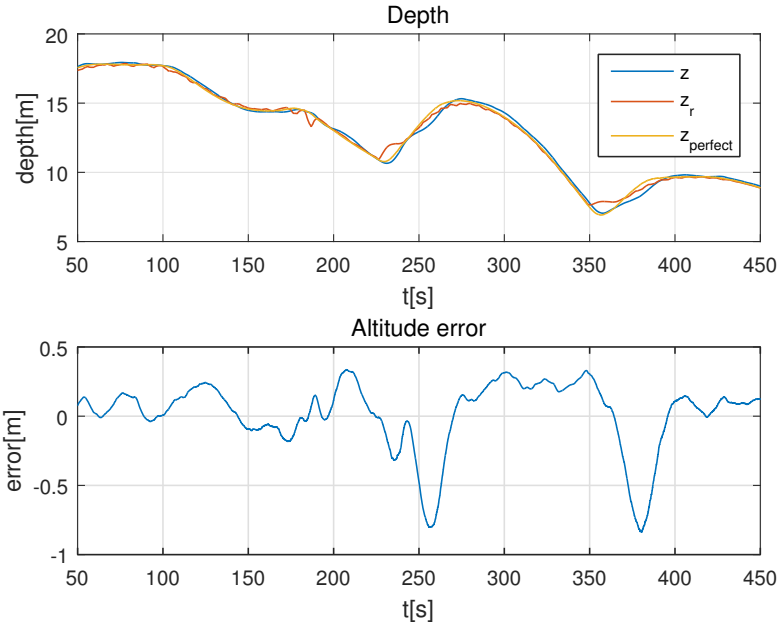


Figure 5.16: Simulation using the highpass controller and the Hopavågen map.

5.5 Heading Control

Two simulations have been run using heading control; both have been run with a constant depth so that the bathymetry would not influence the simulations. The first scenario controls the heading by sending an `IMC::DesiredPath` message from DUNE to AUVSim. This message includes the position of the next waypoint. That waypoint is then sent to the line-of-sight guidance in AUVSim that in turn sends a desired heading Ψ_d to a heading controller. A simulation of the first scenario is shown in Figure 5.17.

The second scenario uses the vector field guidance (Lim et al., 2014) implemented in the DUNE task `Control.Path.VectorField` and sends the the heading angle Ψ_d to AUVSim directly. The configuration parameters used by the vectorfield DUNE task is given in Listing 5.1. The two parameters that have been tuned are the corridor width and the corridor entry angle. The other parameters are the same as for the other LSTS AUVs. It is the ratio between the corridor entry angle and the corridor width that determines how fast the path converges to the desired path. A large entry angle corresponds to a high gain.

How DUNE performs path planning is discussed in more detail in Appendix B. A simulation of the second scenario is shown in Figure 5.18. In both scenarios, the AUV is able to follow the path with good accuracy.

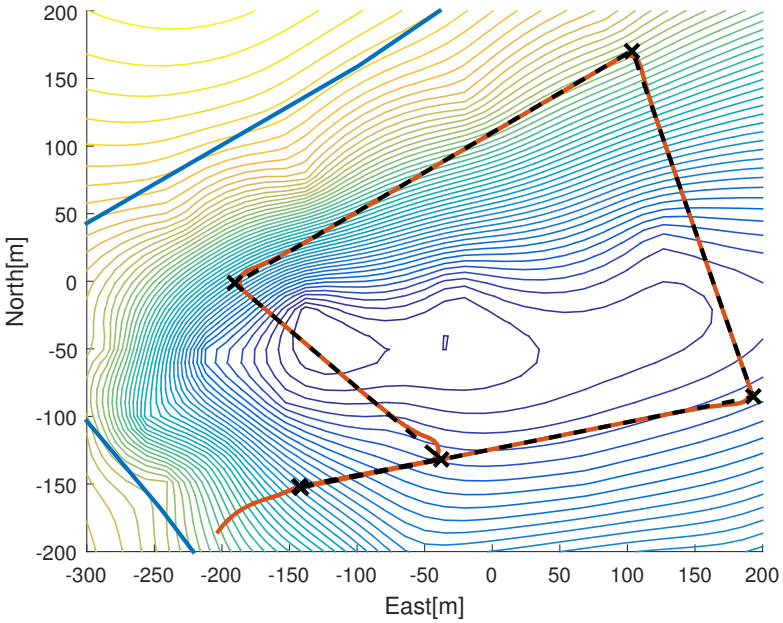


Figure 5.17: Simulation of Heading using DUNE to set the destination goal and AUVSim to set the heading. The thick red line represents the AUV path while the thick dotted black line represents the desired path.

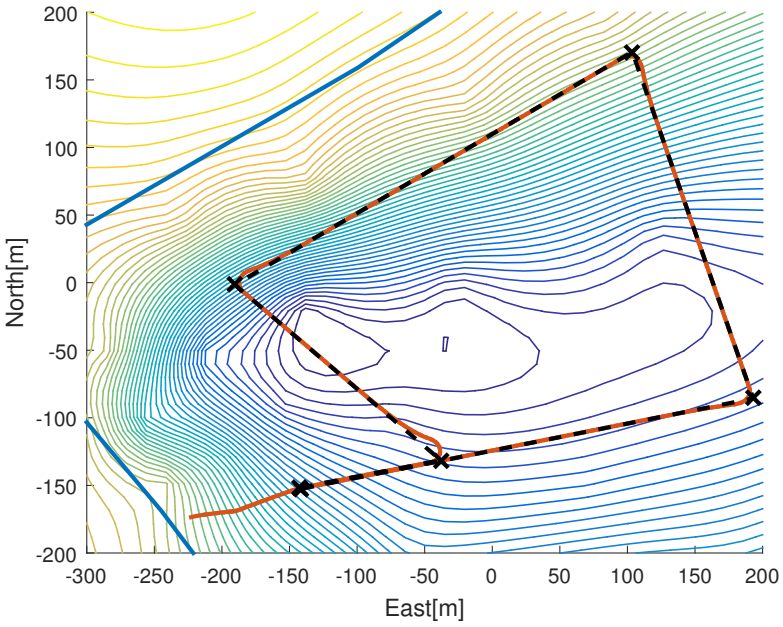


Figure 5.18: Simulation of Heading using the vectorfield task in DUNE to set the heading angle. The thick red line represents the AUV path while the thick dotted black line represents the desired path.

```
[Control.Path.VectorField]
Enabled                                 = Always
Entity Label                            = Path Control
Control Frequency                       = -1
Along-track -- Check Period             = 20
Along-track -- Minimum Speed            = 0.05
Along-track -- Minimum Yaw              = 2
Cross-track -- Monitor                  = false
Cross-track -- Nav. Unc. Factor         = 1
Cross-track -- Distance Limit           = 25
Cross-track -- Time Limit               = 20
Position Jump Threshold                 = 10.0
Position Jump Time Factor               = 0.5
ETA Minimum Speed                       = 0.1
New Reference Timeout                   = 5.0
Course Control                          = false
Corridor -- Width                       = 1.5
Corridor -- Entry Angle                 = 4
```

Listing 5.1: Configuration parameters used for the Vectorfield task.

Chapter 6

Field Testing

The field testing was carried out between the April 20, and April 23, 2015 at Hopavågen in Sør-Trønderlag by the author and PhD candidate Petter Norgren. The goal of the field tests was to see how HuginDuneBridge and DUNE worked together in a real application and to see if the simulation results were valid.

Good tuning of the depth controller was crucial for the performance of the altitude controllers developed in Chapter 3. The REMUS depth controller was therefore tuned before performing missions using the altitude controllers.

6.1 Organization

To launch and retrieve the AUV, we had packed a small inflatable boat with an outboard engine. With us in the boat we had two laptops and a portable WiFi router. The first laptop was used for the Vehicle Interface Program (VIP) and the second was used for Neptus. When the vehicle was underwater, it would communicate via an acoustic transponder developed by Hydroid, and when the vehicle was at the surface it would communicate via WiFi.

During the field testing there were two configuration files that had to be changed between missions: one for DUNE and one for HuginDuneBridge. To make it easier, we made all configuration files the day before and organized and grouped them by mission. After each mission, we would lift the AUV back into the boat. Then we exited the `PP.exe` and `dune.exe` programs, copied over the new configuration files and restarted `PP.exe` and `dune.exe`. Finally, we put the AUV back into the water and started the mission from the VIP.

6.2 Scenarios

6.2.1 Altitude Control

All altitude control scenarios followed the path shown in Figure 6.1. Two altitude controllers were tested: the auto altitude controller and the highpass controller. The altitude reference was set to 10 meters for both scenarios.

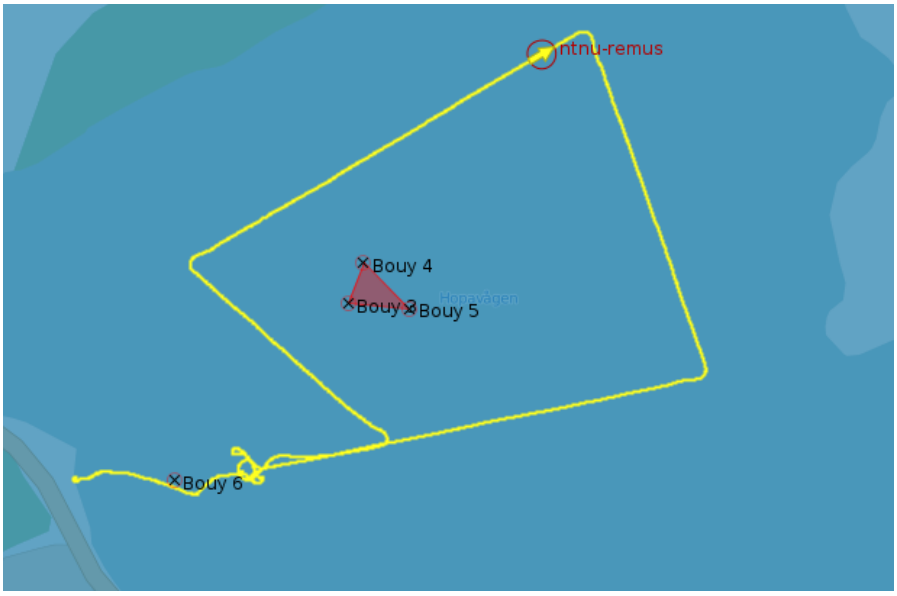


Figure 6.1: Path of the depth control mission in Hopavågen. Plotted using Neptus.

6.2.2 Heading Control

In the heading control scenarios, heading and speed are controlled in addition to the depth. Control of the vehicle's heading has a greater risk than only controlling the vehicles depth. If the vehicle should be moving too close to the seabed, thus risking a bottom crash, the control system aboard REMUS would move towards the surface until it reached a safe altitude. Since the REMUS AUV does not have any forward looking sonars, such fail safe detection is not possible for heading control.

For heading control we used Neptus to create a mission plan. That mission plan was sent to DUNE before starting the mission. The REMUS mission plan was to wait for a position update from a GPS and then hand over control to DUNE that would initiate the standby mode. We would then start the DUNE mission plan

from Neptus. After the DUNE mission had completed, we would abort the mission via VIP and retrieve the vehicle.

The path planned for the heading control missions is shown in Figure 6.2. The plan is to start the mission at Buoy 6, perform the mission and then stop at Goto4.

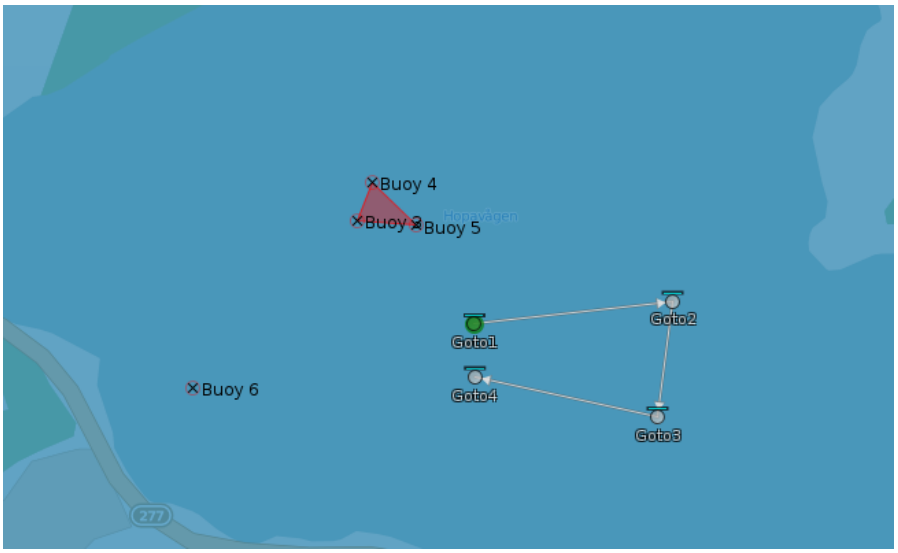


Figure 6.2: Path of the heading control mission in Hopavågen. Plotted using Neptus.

Two scenarios were planned for the heading control. The RECON interface supports setting heading both by setting a destination goal and by setting a heading angle. In the first scenario, we set the heading by setting a latitude/longitude goal. In the second scenario, we set the heading using the path controller in DUNE.

6.3 Tuning of the REMUS Depth Controller

To be able to see a measurable difference between the two altitude control methods, it is important that the depth controller is accurate. In the first altitude control missions, the vehicle oscillated around the reference point. It was therefore deemed necessary to tune the depth controller.

The depth controller on REMUS is a PI controller. The output of the controller is sent to a pitch controller that is a PID controller. Before we started tuning, the parameters for the depth controller were set to $k_p = 3.5$ and $k_i = 0.273$.

For the tuning we created a short mission with a duration of about 10 minutes. The mission was to go to a waypoint and back again with a constant depth of 6 meters. The horizontal path can be seen in Figure 6.3.

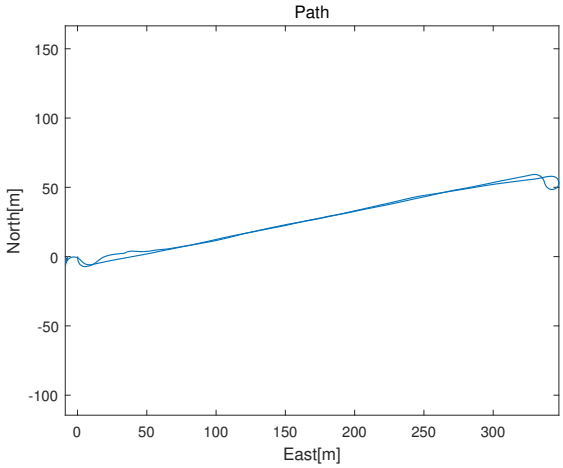


Figure 6.3: Horizontal path for the depth controller tuning mission.

The first tuning test was performed without changing the control parameters. The result can be seen in Figure 6.4.

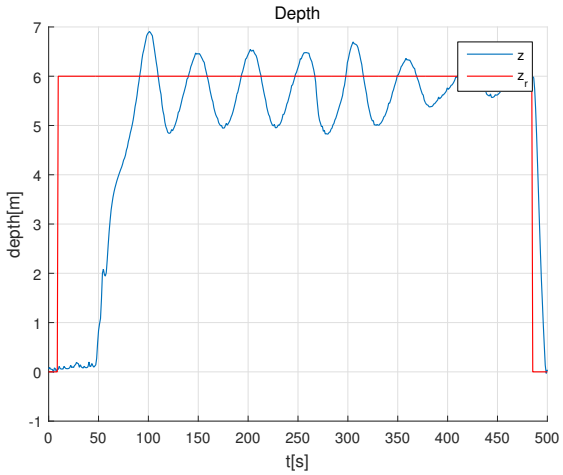


Figure 6.4: Result of the tuning mission with $k_p = 3.0$ and $k_i = 0.273$.

Due to the buoyancy being greater than the weight of the vehicle, the vehicle will rise to the surface when the pitch is zero. Since the vehicle's overshoot was greater under than over the reference point, we first thought that this was due to a low k_p value. A mission with $k_p = 3.5$ is shown in Figure 6.5.

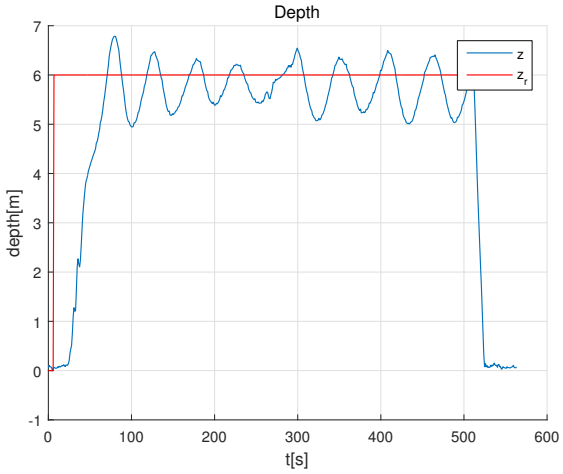


Figure 6.5: Result of the tuning mission with $k_p = 3.5$ and $k_i = 0.273$.

This tuning reduced the control error. To investigate the impact of k_p further, we ran another mission with k_p set to 2.5. The performance is shown in Figure 6.6.

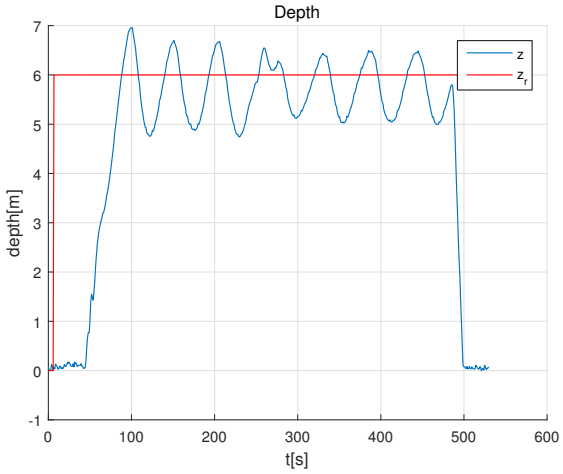


Figure 6.6: Result of the tuning mission with $k_p = 2.5$ and $k_i = 0.273$.

As expected, this increased the control error. The big bigger control error also indicated that the value of k_i was too big. A mission was run with $k_p = 2.0$ and $k_i = 2.223$, reducing k_i by 0.5. The performance of the tuning is shown in Figure 6.7.

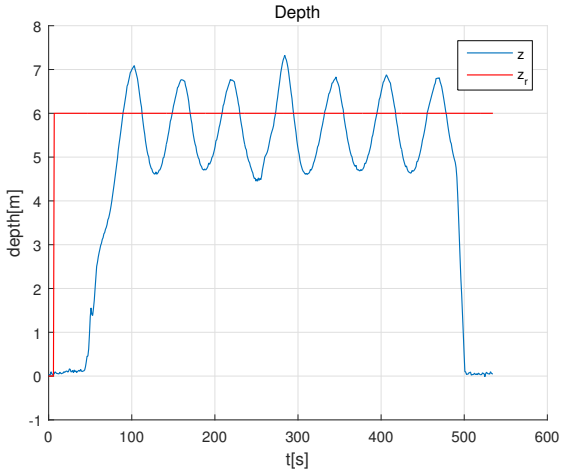


Figure 6.7: Result of the tuning mission with $k_p = 2.0$ and $k_i = 0.223$.

We then tried to turn the value of k_p up again. Figure 6.8 shows a depth plot with $k_p = 4.0$ and $k_i = 0.223$. The overshoot at $t = 300$ corresponds with the turning at the first waypoint. We found the performance of this tuning to be satisfactory.

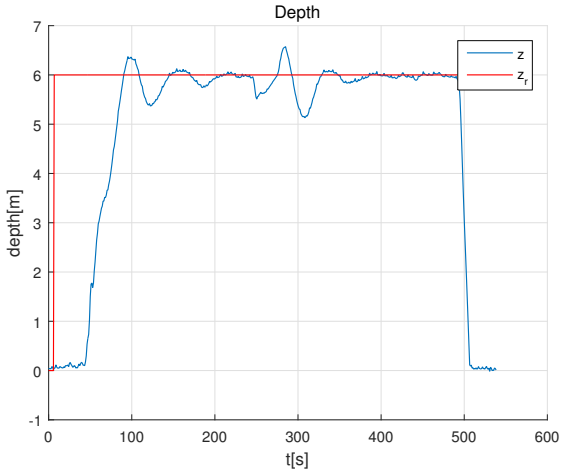


Figure 6.8: Result of the tuning mission with $k_p = 4.0$ and $k_i = 0.223$.

6.4 Altitude Control

The figures showing the results in this section include three plot lines. z represents the depth of the vehicle, z_r represents the depth reference, and $z_{perfect}$ represents the depth 10 meters above the sea bed.

The altitude control missions used the beam ranges from the DVL to find the altitude. After the field tests were carried out, it was found that the range measurements returned from the DVL was the altitude of the beam and not the range (see Figure 6.9). Since the altitude a was shorter than the beam range $|\mathbf{r}|$, the estimated altitude was always shorter than the real altitude.

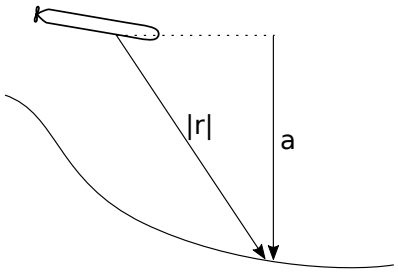


Figure 6.9: The range measurement retrieved from the DVL during the field testing was the altitude a and not the range $|\mathbf{r}|$.

The difference between the altitude estimated by REMUS and the computed altitude is shown in Figure 6.10. The figure shows the difference for the auto altitude mission where the mean error is 0.31 meters. To make it easier to compare the results from the field testing, the mean error was added to $z_{perfect}$ so that the lines would align. The original values for $z_{perfect}$ can be seen in Appendix D.

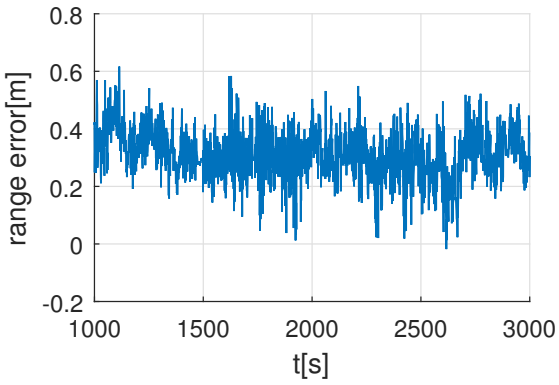


Figure 6.10: The difference between the altitude estimated by REMUS and the altitude estimated from the wrong beam ranges.

6.4.1 REMUS Altitude Controller

The altitude scenario was first run using the altitude controller implemented on REMUS. This mission would function as a baseline that could be compared to the

controllers implemented in DUNE. The depth and desired depth of the vehicle is given in Figure 6.11. As expected the vehicle overshoots when the slope of the seabed changes (see $t = 470$ and $t = 650$).

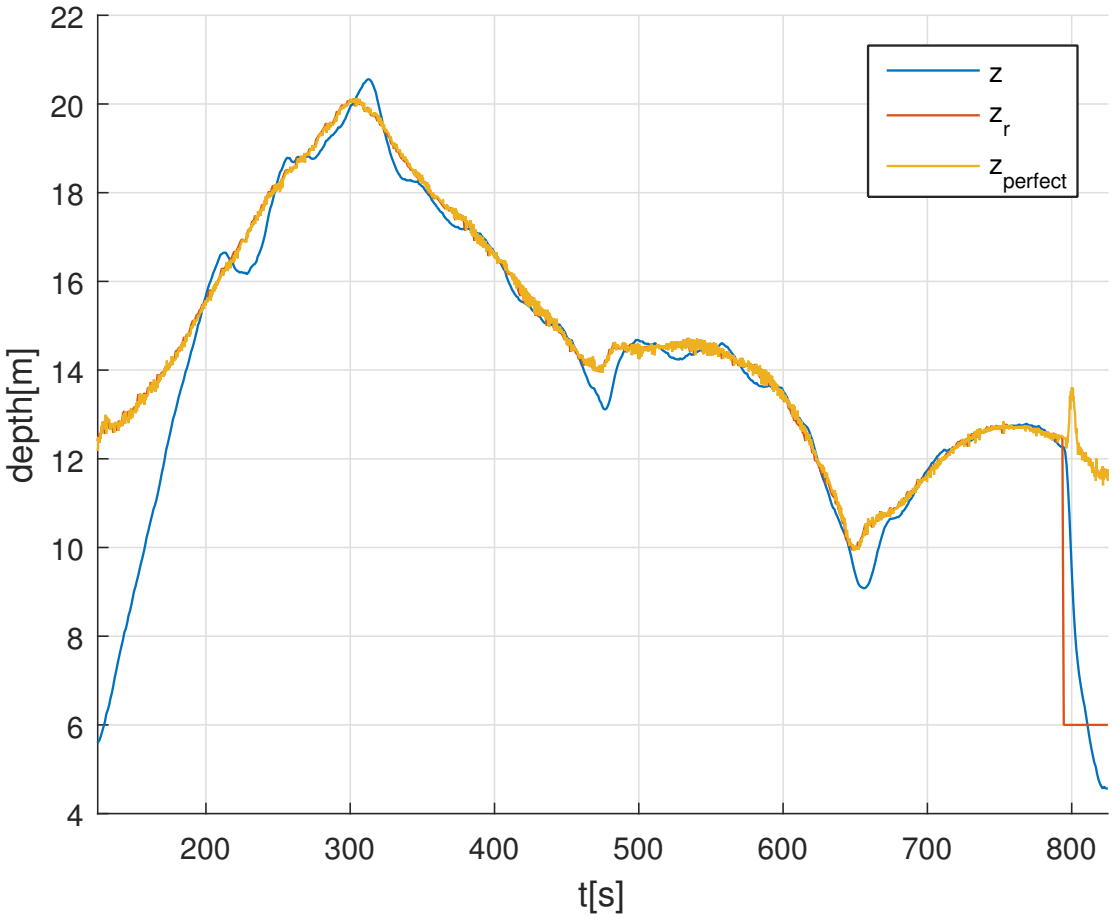


Figure 6.11: Depth z and desired depth z_r for the baseline mission using the REMUS altitude controller.

6.4.2 Auto Altitude Controller

The second mission was performed with the auto altitude controller implemented in DUNE. The result is shown in Figure 6.12. Compared to z_d set by the REMUS computer shown in Figure 6.11, the auto altitude controller gives a similar output. The oscillations observed in the vehicles depth indicate that more effort should be put into tuning of the depth controller, but due to time constraints, this was not done.

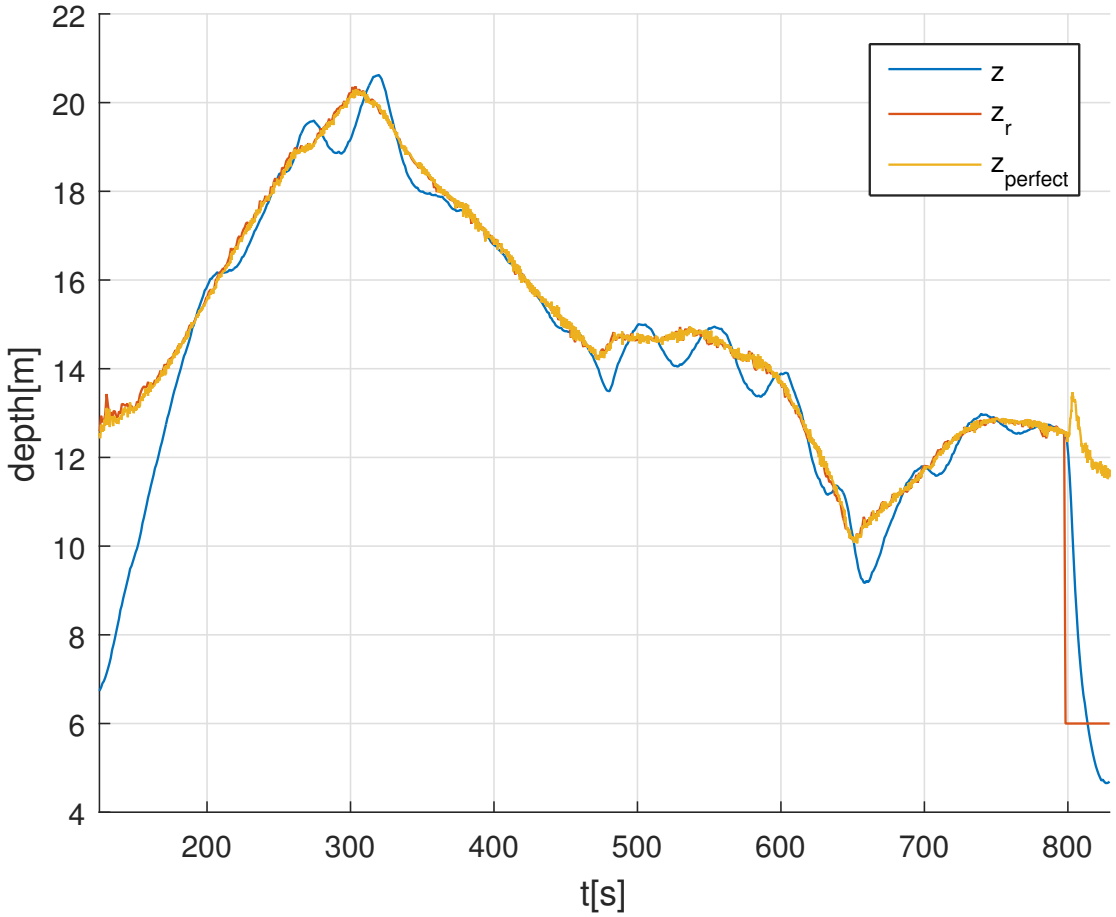


Figure 6.12: Depth z and desired depth z_r for the auto altitude mission.

6.4.3 Highpass Controller

The rest of the altitude control missions were performed with the highpass controller. The first mission with this method is shown in Figure 6.13. The highpass gain parameter k_{hp} was set to 2.0 and the highpass alpha α_{hp} was set to 3.0.

Compared to the desired depth $z_{d,auto}$ from the auto altitude controller, the desired depth $z_{d,hp}$ from the highpass controller had more noise. Although the signal is noisy, it can be seen that the controller is compensating for the slope of the seabed at $t = 470$ and $t = 660$. However, this effect is small.

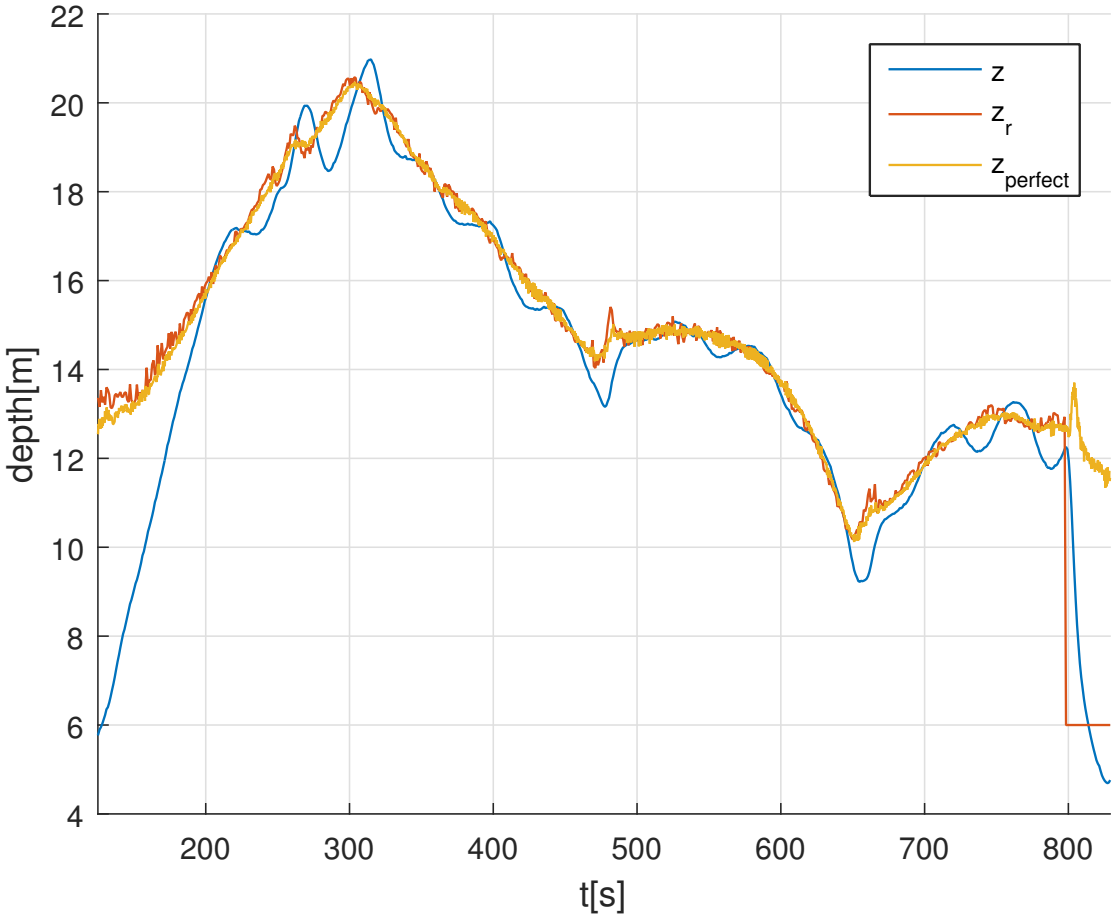


Figure 6.13: Depth z and desired depth z_r for the first highpass altitude mission.

For the last altitude mission we increased the highpass parameters to see if the effect would be more present. The result is shown in Figure 6.14 with the highpass gain k_{hp} set to 3.0 and the highpass alpha α_{hp} set to 5.0. This tuning lead to big oscillations for the vehicle's depth and confirms that good tuning of the highpass controller is crucial.

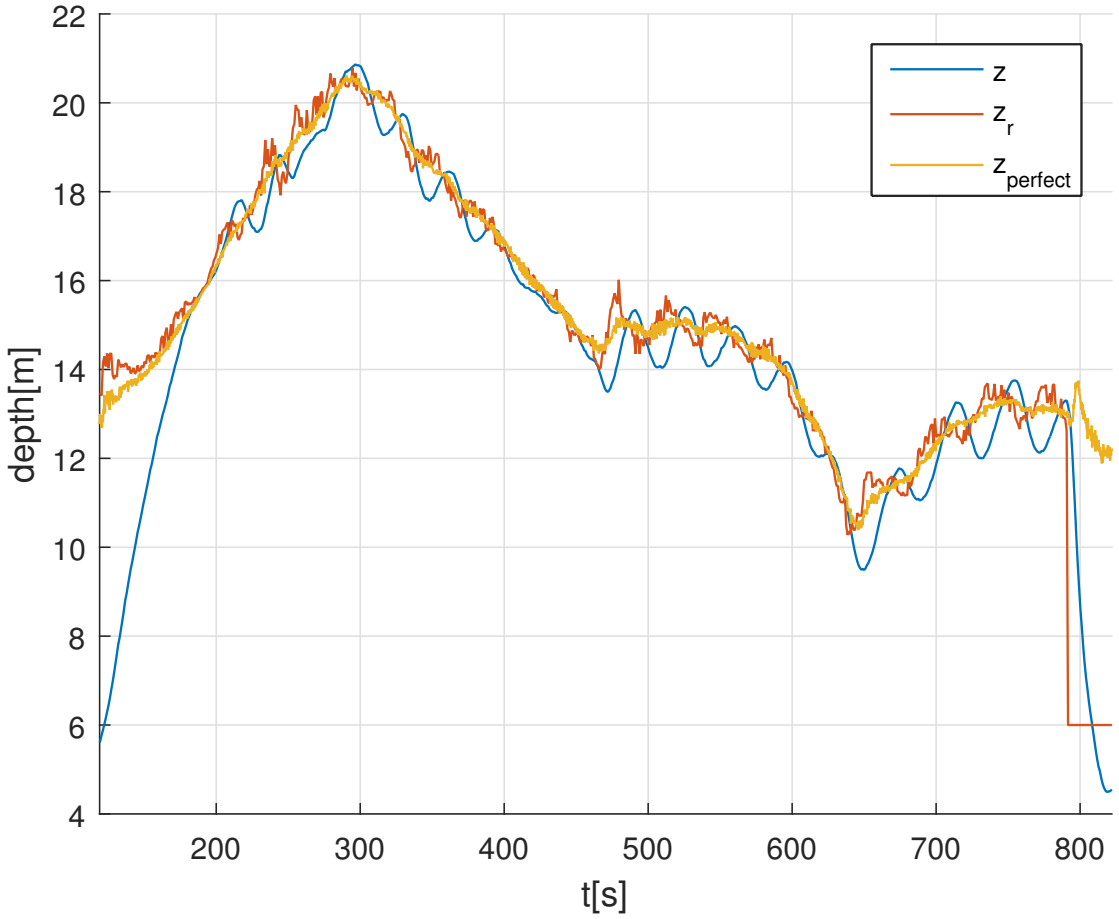


Figure 6.14: Depth z and desired depth z_r for the second highpass altitude mission.

6.4.4 Noise in the DVL range measurements

The high level of noise in the results with the highpass controller, may be explained by noisy input from the DVL. The DVL range measurements is shown in Figure 6.15 and Figure 6.16. In the latter figure it looks like the range measurements have periodic oscillations. Because the DVL is an acoustic system, it is expected that the range measurements should include noise, but due to the periodic oscillation, it looks more like digital noise.

Figure 6.17 shows the range measurements from a mission in the Trondheimsfjord October, 2014 with the same REMUS AUV. In this mission, the range measurements from the DVL had significantly less noise indicating that the DVL may have been faulty during the field tests in Hopavågen.

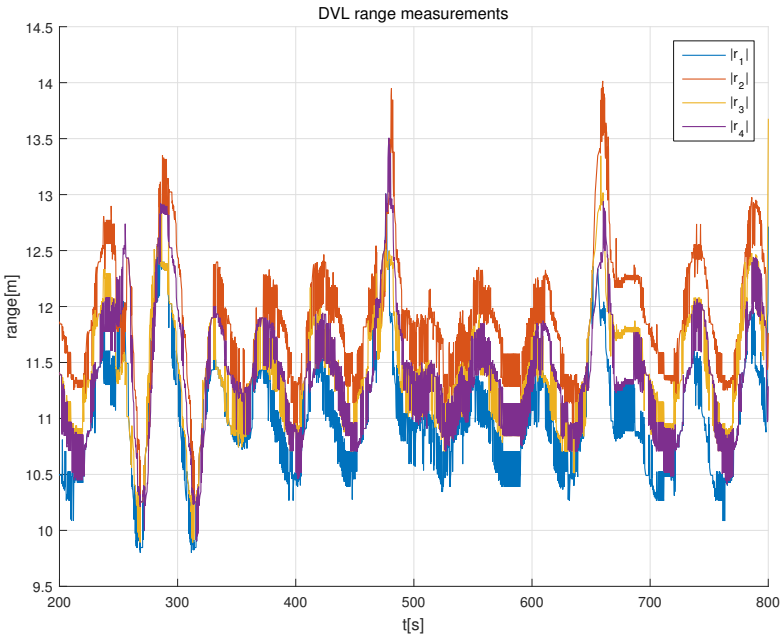


Figure 6.15: Range measurements from the DVL during the first highpass altitude mission.

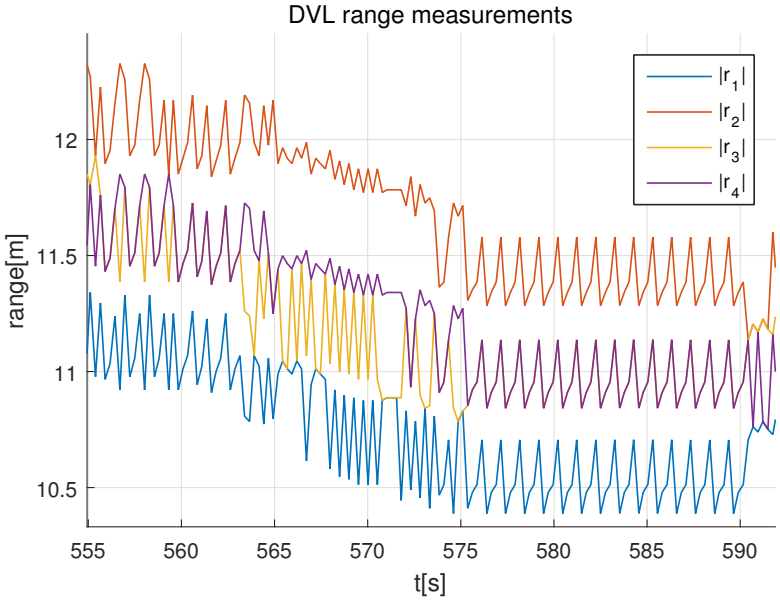


Figure 6.16: Range measurements from the DVL during the first highpass altitude mission.

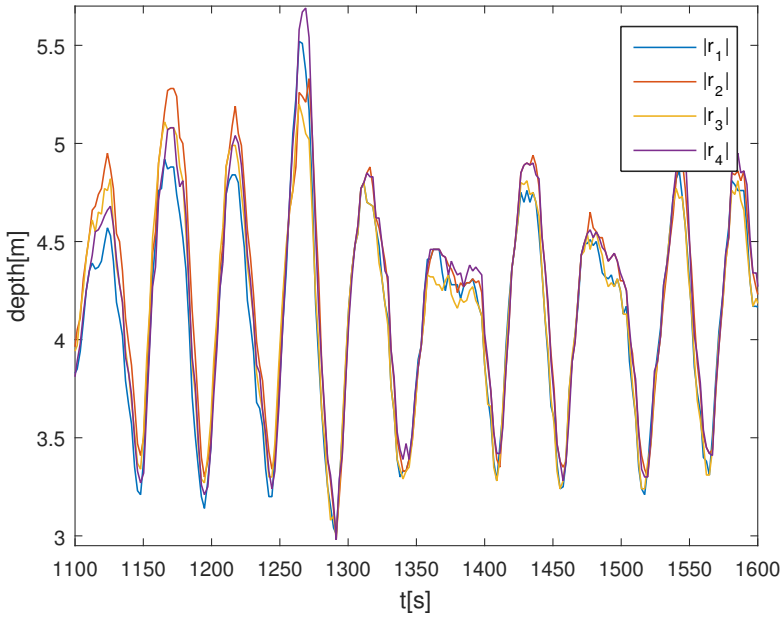


Figure 6.17: Range measurements from the DVL during a mission in the Trondheimsfjord October 2014. The range measurements from this mission had significantly less noise.

6.5 Heading Control

When we tried to perform the heading control we had some problems with the RECON interface. The first scenario was to test heading control by setting a waypoint destination the AUV should move towards. The log file showed that the depth, heading, and speed were sent repeatedly in 1 second intervals, but the REMUS VIP stated that it had not received any heading commands. The REMUS computer would therefore take back control after 5 seconds. Since we did not know why RECON rejected the heading commands, and because of the larger risk of performing heading control, we did not run more heading control tests.

During testing of the new HuginDuneBridge architecture discussed in Chapter 7 we found that the message format implemented in `CReconDriver` was wrong. The command for setting the heading via a waypoint destination in `CReconDriver` separated the latitude and longitude with a comma. The correct separation character was the space character. After fixing the error, RECON accepted the heading goal command.

6.6 Experiences

Using both Neptus and the REMUS VIP to control the mission execution proved to be more troublesome than first thought. Since the DUNE mission plan was initiated by Neptus, it was necessary for REMUS to be in the surface close to the boat so that it had WiFi connection. In the time from when the REMUS mission had started to the time before the DUNE mission was started from Neptus, one of us had to keep REMUS fixed and make sure that it did not slip under the boat due to the current. When we tried to run the heading missions, the boat was anchored to Buoy 6 shown in Figure 6.2. If the DUNE mission had started automatically when RECON control was enabled, this would not have been a problem. This problem has been addressed in the new architecture described in Chapter 7.

Chapter 7

Software Changes Based on Experiences from Field Testing

This chapter presents a new architecture for the HuginDuneBridge and for the interface between DUNE and REMUS. The new architecture has been implemented to address the problems encountered during the field testing. The implementation has been tested on the REMUS computer May 8, 2015 to ensure that the new architecture worked.

7.1 Problems Encountered During Field Testing

7.1.1 Changing Configurations Between Mission

One of the inconvenient moments during the field tests was changing the configuration files between missions. When we were out in the boat, and were getting ready for the next mission, we would have to turn off `PP.exe` and `dune.exe`, change the two configuration files, and restart both programs.

7.1.2 Safety if DUNE Stops Communicating

Another issue that could arise was if `dune.exe` crashed or if the communication suddenly stopped working. Since HuginDuneBridge would repeat the latest received commands at least every second, it would not stop transmitting messages even if DUNE stopped sending messages.

7.1.3 Complexity in HuginDuneBridge

A third problem is that the HuginDuneBridge complexity grew due to the RECON constraints. This is in conflict with the goal of the bridge to be a thin translation layer. This complexity made it more difficult to reason about the code base.

7.2 Architectural Overview

DUNE, being a complete framework for vehicle control, is a better fit for the complexities in HuginDuneBridge. A first goal was therefore to move some of the complexities from HuginDuneBridge to DUNE. Since the RECON interface uses the UDP protocol, it would be possible to move the RECON procedure calls from HuginDuneBridge to DUNE. This would solve or improve upon all the problems described above.

By sending RECON commands from DUNE, the HuginDuneBridge plugin would only be responsible for transmitting the vehicle state from HUGIN to DUNE, which will simplify its design. The HuginDuneBridge configuration would be reduced to the network configurations. Since these parameters rarely change, the DUNE configuration will be the only one that needs to be updated between missions, and the `PP.exe` program does not need to be restarted.

If `dune.exe` would crash, or otherwise be unable to send commands, the safety mechanisms on the REMUS computer would abort the mission after 5 seconds.

Another point that was important during the field testing was logging. If a mission failed to meet the desired performance, having a log file with debugging information was crucial. When HuginDuneBridge grew in complexity, it would also need logging, leading to multiple log files from both DUNE and HuginDuneBridge. Logging is easier in DUNE due to the logging task `Transports.Logging` that saves all IMC messages in a compressed file. After mission completion, the compressed log file can be viewed in Neptus and plotted.

7.3 Implementation

Two new DUNE tasks have been implemented to handle the communication with AUVSim and RECON, `Control.REMUS.AUVSim` and `Control.REMUS.RECON` respectively. In addition, 8 new IMC messages types were created. The code

is available at `https://www.bitbucket.org/sighol/dune-remus` and `https://www.github.com/sighol/imc`.

7.3.1 New IMC Messages

The new IMC messages were created to ease the communication between the two tasks and so that the state of the two tasks could be observed in the log files.

The `IMC::SimulatedTime` message includes the current simulation time and the timestep used by AUVSim. This message is useful for tasks that perform integrations or other time dependent operations. The `Control.REMUS.AltitudeFromDVL` task has been changed to subscribe to this message. If a `IMC::SimulatedTime` have been received, it uses that timestep for the propagation of the highpass filter.

For use by the new `Control.REMUS.RECON` task in DUNE, seven different IMC messages have been created. `IMC::ReconState` mirrors the data that is returned through the RECON interface. It is this message that includes the current mission objective number (or *leg*) used by REMUS.

`IMC::ReconMessage` is used for logging of all packages sent between DUNE and the REMUS computer over the RECON interface. The message includes the command string and the direction of the message: vehicle to external or external to vehicle. With this message, it was easier to debug why RECON rejected the heading commands as experienced during the field testing.

When the RECON task changes between being enabled and disabled, an `IMC::ReconControlState` message is sent out on the IMC bus. This message is used for logging.

Finally, four new messages have been created to mirror the desired state messages used by DUNE: `IMC::DesiredZ`, `IMC::DesiredSpeed`, `IMC::DesiredPath`, and `IMC::DesiredHeading`. The new messages are: `IMC::ReconDesiredZ`, `IMC::ReconDesiredSpeed`, `IMC::ReconDesiredHeadingWaypoint`, and `IMC::ReconDesiredHeadingAngle`. These messages are sent from the RECON task every time the corresponding RECON commands are sent to the REMUS computer. The messages are used by the AUVSim task so that AUVSim can use the commands sent to the REMUS computer instead of the more frequently updated `IMC::Desired*` messages.

The new IMC messages makes it easier to test the logic in the `RECON` task that involves enabling and disabling of the connection to the RECON interface.

7.3.2 HuginDuneBridge

HuginDuneBridge has been greatly simplified since it no longer is in charge of sending RECON messages to the REMUS computer. The communication is flowing in only one direction: from HuginDuneBridge to DUNE. The code in the `CReconDriver`, `ReconSupervisor`, and `ReconProxy` classes has been moved to the `Control.REMUS.RECON` task.

7.3.3 Control.REMUS.RECON

The `RECON` task is in charge of communicating with REMUS over the RECON interface to send commands to the vehicle. The communication can be split into three objectives: deciding which commands should be sent, when they should be sent, and the actual sending. The control flow is shown in Figure 7.1.

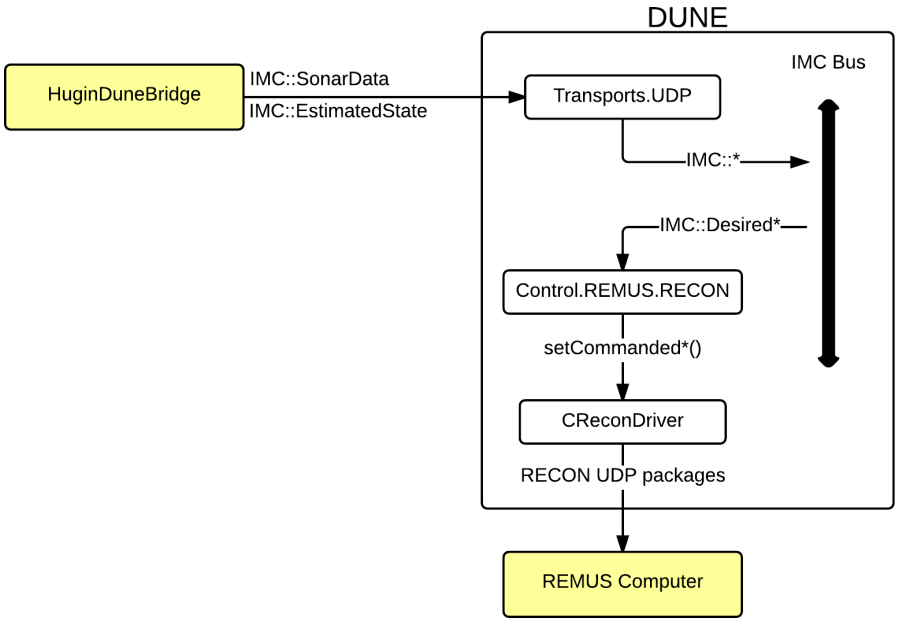


Figure 7.1: Control flow for the `RECON` task.

The configuration parameters for the `RECON` task is given in Listings 7.1.

```
Recon Address = 192.168.1.44
Recon Port = 23456
Control Mode = HeadingWaypoint
Disable After Maneuver Completion = false
Recon Enable -- Method = TimeDelay
Recon Enable -- Leg Start = 1
Recon Enable -- Leg End = 4
Recon Enable -- Time Delay = 20
Recon Enable -- Start Maneuver = HopavaagenMission
```

Listing 7.1: Configuration parameters for the `Control.REMUS.RECON` task.

Which Commands Should be Sent

Which commands that should be allowed to be sent to the REMUS computer is determined by the `Control Mode` parameter. The different control modes are given in Table 7.1.

Table 7.1: The available methods of control.

Method	Enabled Commands
DepthOnly	Depth
HeadingAngle	Depth, Speed, Heading by Angle
HeadingWaypoint	Depth, Speed, Heading by Waypoint
None	

When the Commands Should be Sent

As the `DuneToHugin` block in HuginDuneBridge did before the rewrite, the `RECON` task need to control the frequency of messages that are sent to REMUS. If the time between two messages exceeds 5 seconds, the REMUS computer will take back control, and if the messages are sent too frequently, the REMUS computer can be stressed. This is handled by the `CommandLoop` class that resembles the `ReconProxy` class in HuginDuneBridge. The `CommandLoop` class keeps track of the latest given command and sends this to the REMUS computer every 100 millisecond. If no commands have been given to `CommandLoop`, no commands are sent to REMUS.

The RECON interface will not always be enabled during the execution of a mission. A typical mission will use the REMUS controllers to move the vehicle to a starting

position, and then enable RECON and let DUNE take over control. This switching logic is determined by the *enable* and *disable* conditions. The conditions used in Hopavågen were to check that the current REMUS mission objective was inside a given interval. To use this condition, `Recon Enable - Method` must be set to `Leg` and the interval is determined by the `Recon Enable - Leg Start` and `Recon Enable - Leg End` parameters.

Two new conditions have been added to the `RECON` task. The `TimeDelay` method enables RECON control after a given time delay after the DUNE program has started. This time delay is specified by the `Recon Enable - Time Delay`.

During the field tests, we found that the mission file included an option to allow or deny RECON commands at each mission objective. This condition is sent back via the RECON status messages `RemoteControlAllowed`. Setting the `Recon Enable - Method` to `ReconAllowed` will enable this behavior.

The last switching logic method is the `None` condition. In this mode, RECON will never be enabled. This is useful if DUNE should be used for logging, and not take control.

Start Maneuver When RECON is Enabled

The parameter `Recon Enable - Start Maneuver` can be used to specify a DUNE maneuver that should be run when RECON is enabled. This was implemented due to the difficulties experienced during heading control in Hopavågen. Not being required to use Neptus to start the DUNE mission simplifies the mission execution. If this parameter is not present, DUNE will go into standby mode and wait for commands from Neptus as was done during the field tests.

7.3.4 Control.REMUS.AUVSim

The `Control.REMUS.AUVSim` tasks has taken over the responsibility of HuginDuneBridge for the interaction with the AUVSim simulator. Instead of routing the AUVSim messages through HuginDuneBridge, they are now sent directly from AUVSim to the `Control.REMUS.AUVSim` task. This makes simulations easier since HuginDuneBridge does not need be started. With less communication between the different programs, AUVSim will also be able to run simulations faster and with less delay.

The `AUVSim` task can be used in two different modes: RECON mode and automatic mode. When the `AUVSim` task is in RECON mode it will listen to the `IMC::Recon-`

`Desired*` messages instead of the `IMC::Desired*` messages. This is useful for testing the configuration of the `RECON` task to see that the correct messages are sent to REMUS. Note that this will add a delay to the messages sent to AUVSim since the `RECON` task sends messages at a fixed frequency. AUVSim should therefore be run at a low speedup when the `AUVSim` task is in RECON mode. In short the automatic mode is more convenient for quick simulations while the RECON mode tests a bigger part of the system. The control flow is shown in Figure 7.2.

The configuration for the `AUVSim` DUNE task is given in Listings 7.2.

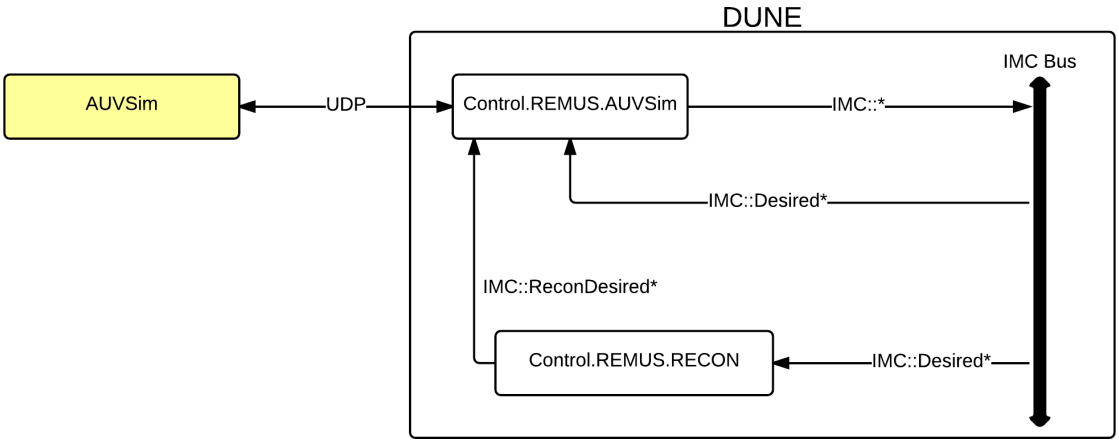


Figure 7.2: Control flow for the `AUVSim` task.

```
Input Port = 8889
Output Port = 9090
Output Address = 127.0.0.1
Only Use Recon Commands = false
```

Listing 7.2: Configuration parameters for the `Control.REMUS.AUVSim` task.

7.4 Testing

Testing of the new architecture was conducted on May 8, 2015. Since `CReconDriver` had to be rewritten to not depend on HUGIN SDK, it was necessary to test that it functioned as before. We ran both DUNE and the plugin on the PP computer and started the do-nothing mission that we used for heading testing in Hopavågen. The log files showed that the RECON connection worked as before, and that messages were sent and received.

We had some problems with the the `RemoteControlAllowed` status that we wanted to use to determine when the `RECON` task should be enabled. The status was set to true regardless of the ReconAllowed setting in the mission file. However, using the mission objective number, as we did during the field testing, still worked.

Chapter 8

Conclusions and Further Work

This master thesis has presented a new software system for developing control systems on the REMUS 100 AUV using the DUNE framework. Two altitude controllers have been developed using the range measurements from a Doppler velocity log (DVL) aboard REMUS. Both controllers used REMUS' depth controller as control output. The auto altitude controller used the DVL ranges to estimate the altitude. The second altitude controller, the highpass controller, used the DVL ranges to approximate the slope of the seabed, and used that estimate in a feedforward term.

The implementation has been tested both by simulations using the AUVSim simulator and in a field test at Hopavågen. The simulations have shown that the highpass controller can reduce the altitude error on large bathymetric variations. Using a PD depth controller, as used by the REMUS vehicle, the maximum altitude error reduction ranged from 30% to 45%. Using a PID depth controller the simulations show that the maximum altitude error was reduced by around 60%.

The field testing of the highpass controller gave varying results due to large amounts of noise in the DVL range measurements. The highpass controller proved to be sensitive to the noise, and did therefore give a noisy control output. However, the auto altitude controller gave good results that are similar to the control output used by the altitude controller on the REMUS AUV.

The DUNE integration proved to be a robust strategy for software development on REMUS. The NTNU REMUS 100 can now use DUNE as a development platform, allowing reuse of code from related research activities.

8.1 Recommendations for Further Work

Heading control was not tested due to an error in the `CReconDriver` interface. The error was identified after the field testing had been carried out, but it has not been tested in the field yet. Using DUNE to control the heading will make it possible to perform most of the different maneuvers supported by DUNE and make the DUNE interface more complete.

The highpass filter in the highpass controller used the difference between the two last measurements. This approach is sensitive to large steps between two measurements, and therefore also to noise. A new approach could be to increase the timespan for the filter such that more than two measurements are used.

The depth controller aboard REMUS is difficult to tune since it does not have a derivative gain. New control strategies could be investigated that controls the fin positions instead of using the depth controller on REMUS. This would require a re-implementation of the whole control system with heading controllers and path planning. However, the AUVs developed by LSTS use DUNE for the whole control system. Using these tasks will reduce the implementation time, but the tasks would still need extensive tuning.

Chapter 9

Bibliography

Allen, B., Vorus, W. S., and Prestero, T. (2000). Propulsion system performance enhancements on remus auvs. In *Oceans 2000 MTS/IEEE Conference and Exhibition*, volume 3, pages 1869–1873. IEEE.

ArduPilot (2015). Autopilot suite. [Online] http://ardupilot.com/.

Bruyninckx, H. (2001). Open robot control software: the ORocos project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528.

Candeloro, M., Sørensen, A., Longhi, S., and Dukan, F. (2012). Observers for dynamic positioning of rovs with experimental results. volume 9, pages 85–90.

Carlton, J. (2012). *Marine propellers and propulsion*. Butterworth-Heinemann.

Carreras, M., Candela, C., Ribas, D., Mallios, A., Magí, L., Vidal, E., Palomeras, N., and Ridao, P. (2013). Sparus ii, design of a lightweight hovering auv. In *Martech 2013 5th International Workshop on Marine Technology*. SARTI.

Cola2 (2015). The ros source code for the sparus ii auv. [Online] Available: `https://bitbucket.org/udg_cirs/cola2`.

DeMarco, K., West, M. E., and Collins, T. R. (2011). An implementation of ros on the yellowfin autonomous underwater vehicle (auv). In *OCEANS 2011*, pages 1–7. IEEE.

Dias, P., Fraga, S., Gomes, R., Goncalves, G., Pereira, F., Pinto, J., and Sousa, J. (2005). Neptus - a framework to support multiple vehicle operation. In *Oceans 2005 - Europe*, volume 2, pages 963–968.

Dias, P., Goncalves, R., Pinto, J., Sousa, J., Gongalves, R., and Pereira, F. (2006). Mission review and analysis. In *Information Fusion, 2006 9th International Conference on*, pages 1–6.

Dukan, F., Ludvigsen, M., and Sorensen, A. (2011). Dynamic positioning system for a small size rov with experimental results. In *OCEANS, 2011 IEEE - Spain*, pages 1–10.

Dukan, F. and Sørensen, A. J. (2012). Altitude Estimation and Control of ROV by use of DVL. volume 9, pages 79–84.

Faria, M., Pinto, J., Py, F., Fortuna, J., Dias, H., Martins, R., Leira, F., Johansen, T., Sousa, J., and Rajan, K. (2014). Coordinating UAVs and AUVs for oceanographic field experiments: Challenges and lessons learned. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 6606–6611.

Fossen, T. I. (2011). *Handbook of marine craft hydrodynamics and motion control*. John Wiley & Sons.

Gautam Vallabha (2015). Real-Time Pacer for Simulink. [Online] Available: `http://www.mathworks.com/matlabcentral/fileexchange/29107-real-time-pacer-for-simulink`. (April, 2015).

Holsen, S. A. (2014). *Evaluation of Software Platforms and Development of Terrain Relative Navigation for UUV*. Project thesis, NTNU.

Johns Hopkins University Dynamical Systems and Control Laboratory (2014). Video of open loop control of ROV using ROS. [Online] Available: `http://vimeo.com/26510562`. (December, 2014).

Kermorgant, O. (2014). A dynamic simulator for underwater vehicle-manipulators. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 25–36. Springer.

Koenig, N. and Howard, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154.

Lim, S., Jung, W., and Bang, H. (2014). Vector field guidance for path following and arrival angle control. In *Unmanned Aircraft Systems (ICUAS), 2014 International Conference on*, pages 329–338.

LSTS (2014). LSTS homepage. [Online] Available: `http://lsts.fe.up.pt/`. (December, 2014).

Magyar, G., Sinčák, P., and Krizsán, Z. (2015). Comparison study of robotic middleware for robotic applications. In Sinčák, P., Hartono, P., Virčíková, M., Vaščák, J., and Jakša, R., editors, *Emergent Trends in Robotics and Intelligent Systems*, volume 316 of *Advances in Intelligent Systems and Computing*, pages 121–128. Springer International Publishing.

Mancheño, A. G. (2014). Numerical model of currents and tides in an inlet using mike 3. Student project at the Dept. of Civil and Transport Engineering NTNU.

Marques, E. R., Ribeiro, M., Pinto, J., Sousa, J. B., and Martins, F. (2015). NVL: a coordination language for unmanned vehicle networks. In *ACM Symposium on Applied Computing (SAC'15)*. ACM, ACM.

Martins, R., Dias, P., Marques, E., Pinto, J., Sousa, J., and Pereira, F. (2009). Imc: A communication protocol for networked vehicles and sensors. In *OCEANS 2009 - EUROPE*, pages 1–6.

Ma'sum, M., Jati, G., Arrofi, M., Wibowo, A., Mursanto, P., and Jatmiko, W. (2013). Autonomous quadcopter swarm robots for object localization and tracking. In *Micro-NanoMechatronics and Human Science (MHS), 2013 International Symposium on*, pages 1–6.

Meinecke, G., Albiez, J., Joyeux, S., Ratmeyer, V., and Renken, J. (2013). ORocos based control software of the new developed marum hybrid-rov for under-ice applications. In *Oceans - San Diego, 2013*, pages 1–6.

Melim, A. and West, M. (2011). Towards autonomous navigation with the yellowfin auv. In *OCEANS 2011*, pages 1–5. IEEE.

Natarajan, S., Gaudig, C., and Hildebrandt, M. (2012). Offline experimental parameter identification using on-board sensors for an autonomous underwater vehicle. In *Oceans, 2012*, pages 1–8.

Newman, P. M. (2008). Moos-mission orientated operating suite. *Massachusetts Institute of Technology, Tech. Rep*, 2299(08).

Newman, P. M. (2014). Moos ROS Bridge. [Online] Available: `https://github.com/SyllogismRXS/moos-ros-bridge`.

Norgren, P. and Skjetne, R. (2015). Line-of-sight iceberg edge-following using an AUV equipped with multibeam sonar. Submitted.

OpenSceneGraph (2014). OpenSceneGraph. [Online] Available: `http://openscenegraph.org`. (December, 2014).

osgOcean (2014). oscOcean source code repository. [Online] Available: `http://code.google.com/p/osgocean`. (December, 2014).

Pinto, J., Calado, P., Braga, J., Dias, P., Martins, R., Marques, E., and Sousa, J. (2012). Implementation of a control architecture for networked vehicle systems. In *Navigation, Guidance and Control of Underwater Vehicles*, volume 3, pages 100–105.

Pinto, J., Dias, P., Martins, R., Fortuna, J., Marques, E., and Sousa, J. (2013). The LSTS toolchain for networked vehicle systems. In *OCEANS - Bergen, 2013 MTS/IEEE*, pages 1–9.

Prats, M., Pérez, J., Fernández, J. J., and Sanz, P. J. (2012). An open source tool for simulation and supervision of underwater intervention missions. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 2577–2582. IEEE.

Prestero, T. T. J. (2001). *Verification of a six-degree of freedom simulation model for the REMUS autonomous underwater vehicle*. PhD thesis, Massachusetts institute of technology.

Qt-Project (2015). Qt project. [Online] Available: `http://qt-project.org/`. (June, 2015).

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5.

ROS (2014). ROS wiki introduction. [Online] Available: `http://wiki.ros.org/ROS/Introduction`. (December, 2014).

Sa, I. and Corke, P. (2012). System identification, estimation and control for a cost effective open-source quadcopter. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 2202–2209.

Sørensen, A. J. (2013). *Marine Control Systems Propulsion and Motion Control of Ships and Ocean Structures*.

SQLite (2015). Sqlite3. [Online] Available: `https://www.sqlite.org/`. (June, 2015).

Sucan, I. A. and Chitta, S. (2014). Moveit! [Online] Available: `http://moveit.ros.org`. (December, 2014).

Teledyne RD Instruments (2014). [Online] Available: http://www.rdinstruments.com/datasheets/workhorse_monitor_ds_lr.pdf. (December, 2014).

Thomas, D., Scholz, D., and Blasdel, A. (2014). rqt. [Online] Available: http://wiki.ros.org/rqt. (December, 2014).

Ubuntu (2015). Ubuntu. [Online] Available: http://www.ubuntu.com/. (June, 2015).

Wadoo, S. and Kachroo, P. (2010). *Autonomous Underwater Vehicles: Modeling, Control Design and Simulation.* CRC Press.

Appendices

Appendix A

Guidelines for DUNE Development on REMUS

A.1 Replay Missions

It is possible to replay log files from previous missions in DUNE. Since all input data is stored in the log file, the whole mission may be run in replay and it should produce the same results.

This is of interest if additional data should have been logged during a mission, but was omitted. Logging can be added after the mission has completed, and DUNE can run the mission log file in replay to recover the omitted data. This was done with the first mission using the highpass controller in Hopavågen to retrieve intermediate values used in the algorithm. These values made it easier to tune the gain k_{hp} and alpha α_{hp} paremeters prior to last field tests.

The MATLAB script `replayDuneLog.m`, that is included in the git repository of AUVSim, can be used to replay missions from MATLAB. To use the command, DUNE must be running with the `Transports.Replay` task enabled. The script will send an IMC message to DUNE and return when the replay has finished.

A.2 Compiling Programs for the PP computer

To run programs on the PP computer, they have to be compiled with a compiler version supported by the PP computer. If the compiler used to compile the program has a newer version than the one supported by the PP computer, the program will

crash with a `side-by-side` configuration error. To use the currently supported compiler version, Visual Studio 2005 (VS2005) must be installed with Service Pack 1 (SP1). VS2005 must be installed withouth any service packs or updates and SP1 should be installed through Windows Update.

It is important that no other updates are installed from Windows Update. If VS2005 is updated, PP will no longer be able to run the compiled program. It is recommended to disable additional updates for VS2005 in Windows Update.

If VS2005 is accidentally updated, it must be reinstalled. It is therefore important to have the installation files available before doing field testing.

The HuginDuneBridge plugin must be compiled with the correct version of VS2005 to be able to run inside `PP.exe`. However, it should be possible to compile DUNE with more recent versions of Visual Studio. If the correct side-by-side files are copied to the folder containing `dune.exe`, it may work, but this has not been tested.

A.2.1 Debugging side-by-side Configuration Errors on the PP Computer

If a program is compiled with a compiler version not supported by the PP computer, it will crash with a `side-by-side` configuration error when run on the PP computer. This is caused by the program trying to refer to side-by-side libraries that are not present on the computer.

The error can be debugged using the `SxSTrace` command line tool. Before launching the program, run

```
SxsTrace Trace -logfile:debugSxS.etl
```

on the Windows command line. Then run the program that should be debugged. After the program has crashed, select the command line window and press *Enter*. To parse the sxstrace logfile, run

```
SxsTrace Parse -logfile:debugSxS.etl -outfile:debugSxS.txt
```

on the Windows command line. Open `debugSxS.txt` to see what caused the error.

The problem is usually solved by reinstalling Visual Studio.

A.3 Recompile IMC

A bash script has been made to automate recompilation caused by updates to `IMC.xml`. The script updates the IMC source code used by DUNE and compiles the IMC library used by Neptus. After the script has finished, DUNE and Neptus must be recompiled. The script is shown in Listing A.1.

```
#!/bin/bash

DUNE_DIR='path/to/dune'
IMC_DIR='path/to/imc'
IMCJAVA_DIR='path/to/imcjava'
NEPTUS_DIR='path/to/neptus'

cd "$DUNE_DIR" &&
python programs/generators/imc_code.py -x \
  "$IMC_DIR"/IMC.xml "$DUNE_DIR"/src/DUNE/IMC &&
python programs/generators/imc_blob.py -x \
  "$IMC_DIR"/IMC.xml "$DUNE_DIR"/src/DUNE/IMC &&
python programs/generators/imc_tests.py -x \
  "$IMC_DIR"/IMC.xml "$DUNE_DIR"/programs/tests &&
python programs/generators/imc_addresses.py -x \
  "$IMC_DIR"/IMC_Addresses.xml etc/common/imc-addresses.ini &&

cd "$IMCJAVA_DIR" &&
# ant will ask for location of the
# imc directory (default = ../imc).
# echo "" confirms the location.
echo "" | ant && ant lsf2llf &&

cp dist/libimc.jar "$NEPTUS_DIR"/lib &&

echo "Updated DUNE and recompiled imcjava."
echo "Please recompile DUNE and Neptus."
```

Listing A.1: Script to recompile IMC dependencies.

A.4 Logging

The easiest way to log new data is to use the IMC protocol and add new message types for the data that should be logged. The new messages can then be filled with the intended data and be dispatched to the IMC bus. The message must be added to the `Transports.Logging` configuration to be present in the log file. The Mission Review and Analysis tool inside Neptus may be used to plot the data after the mission or simulation has completed.

MATLAB can also be used to plot the data. First, the `lsf` file must be converted to the `llf` format using the `lsf2llf` tool that is included in the IMCJava repository at `github.com/lsts/imcjava`. The `llf` files can be loaded to matlab using the `llfload` tool that is included in the DUNE repository.

See `https://github.com/LSTS/dune/wiki/Working-with-IMC` for how to add new messages.

A.5 Mission Preparation

Some measures can be done to organize the code before a mission. Each mission should have its own git branch named mission/"*mission name*". A good reason to use a different branch for the mission is so that the state of the code and the configuration can be found later.

If DUNE or HuginDuneBridge needs to be recompiled it is convenient to have the installation files for Visual Studio 2005 SP1 available. Then, Visual Studio can be reinstalled if Windows Update have been run by accident.

A.6 Pre-Mission Checklist

- Stop the RecExtrCtrl program on the PP computer. This program uses RECON and will therefore interfere with DUNE.
- Enable the HuginDuneBridge plugin on the PP computer.
- Restart the `PP.exe` program on the PP computer. Check that the plugin is loaded.
- Run `dune.exe` on the PP computer.
- Run `Neptus` on a local computer and open the `lauv` console.

- Check that the `ntnu-remus` vehicle is connected and verify that the estimated position is near the actual position. If the vehicle is indoors, the estimated position can be far off, even multiple kilometers. When the vehicle is outdoors, the estimated error should be small.

A.7 Post-Mission Checklist

- Download the DUNE log files.
- Disable the HuginDuneBridge plugin on the PP computer.
- Stop the DUNE program.
- Start the RecExtrCtrl program.

Appendix B

Path Control in DUNE

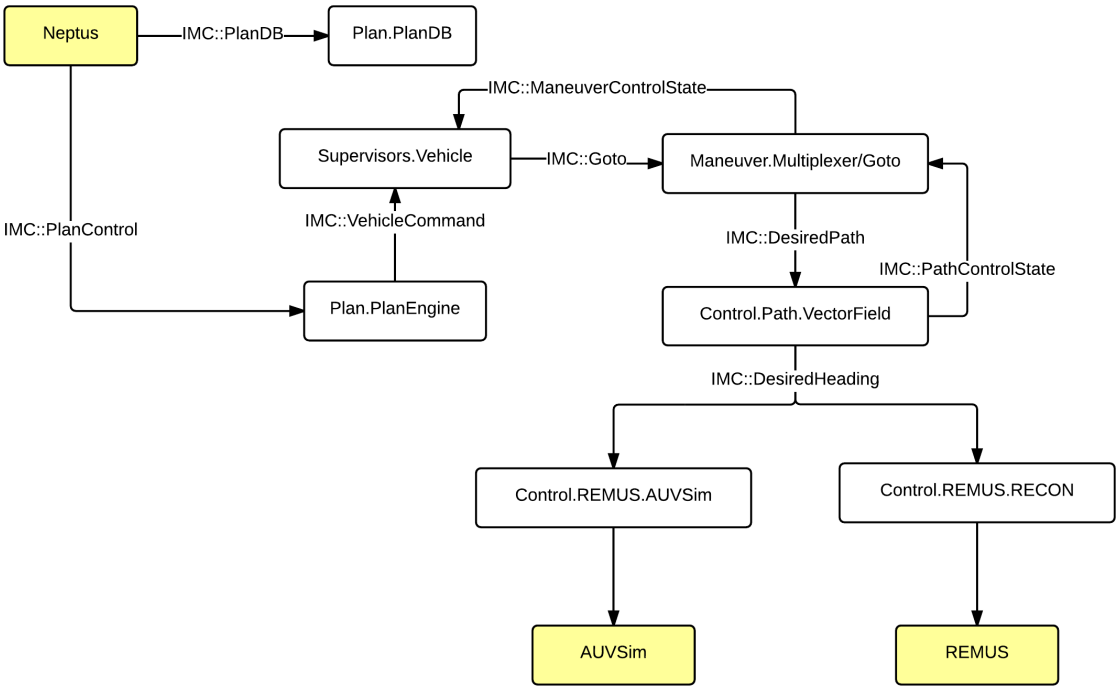


Figure B.1: Control flow for path planning in DUNE.

The control flow for path planning in DUNE is shown in Figure B.1. Missions plans are generated using the Neptus Ground Control System (GCS).

A mission plan consists of multiple *maneuvers* that are linked together into a plan. Examples of maneuvers are the `Goto` maneuver that commands the vehicle to go to a waypoint, the `Loiter` maneuver that commands the vehicle to run in a circle

around a waypoint, and the `PopUp` maneuver that commands underwater vehicles to surface at a specified waypoint. Mission plans can be sent to DUNE in an `IMC::PlanDB` message. The mission plan is then saved into an sqlite3 database (SQLite, 2015).

Launching of missions are initiated by Neptus. The start command is encoded in an `IMC::PlanControl` message that includes the mission name and the operation that should be run (start, stop, etc.). The message is received by the `Plan.PlanEngine` task in DUNE and translated into an `IMC::VehicleCommand` message that i sent to the `Supervisors.Vehicle` task. It is this supervisor task that is in charge of the plan execution. The supervisor sends the mission maneuvers one by one to the `Maneuver.Multiplexer` task. When the supervisor receives an `IMC::ManeverControlState` stating that the maneuver has finished, the next maneuver command is sent to the multiplexer.

The multiplexer task consists of multiple components representing each of the possible maneuvers. If an `IMC::Goto` message is received by the multiplexer, it is then handled by the `Maneuver.Multiplexer/Goto` component. In the case for the `Goto` command, the message is translated into an `IMC::DesiredPath` command. When the path controller has completed the path, it sends an `IMC::PathControlState` back to the multiplexer. There are multiple path controllers in DUNE, including pure pursuit and line-of-sight, but the one used in this thesis is the vector field path controller. When the path controller receives a desired path, it will send the `IMC::DesiredHeading` and `IMC::DesiredSpeed` commands that can be received by the `Control.REMUS.AUVSim` and `Control.REMUS.RECON` tasks.

Appendix C

Simulations Using a PID Depth Controller

The simulation results presented in Chapter 5 used a depth controller that resembles the one used on the REMUS 100 AUV. This depth controller was a PI controller that was difficult to tune since it had no derivative gain. This appendix includes simulation results using a PID depth controller. The derivative gain made it easier to get the vehicle to follow the desired path and made the difference between the two controllers more visible. The simulations scenarios presented in this appendix is the same as the ones used in Chapter 5.

The coefficients used by the depth controller are shown in Table C.1. The highpass gain k_{hp} was set to 20.0 and the highpass alpha α_{hp} was set to 3.0.

Table C.1: Simulation coefficients for the AUVSim PID depth controller.

Parameter	Value
$k_{p,z}$	0.20
$k_{i,z}$	0.02
$k_{d,z}$	0.80
$i_{max,z}$	10.00

C.1 Scenario: Constant Sloped Map

The first scenario uses the constant sloped map. With the auto altitude controller showed in Figure C.1, the maximum error is around 0.5 meters. With the highpass

controller showed in Figure C.2, the maximum error is around 0.2 meters. The highpass controller gives a faster response due to the increased error when the slope of the seabed changes.

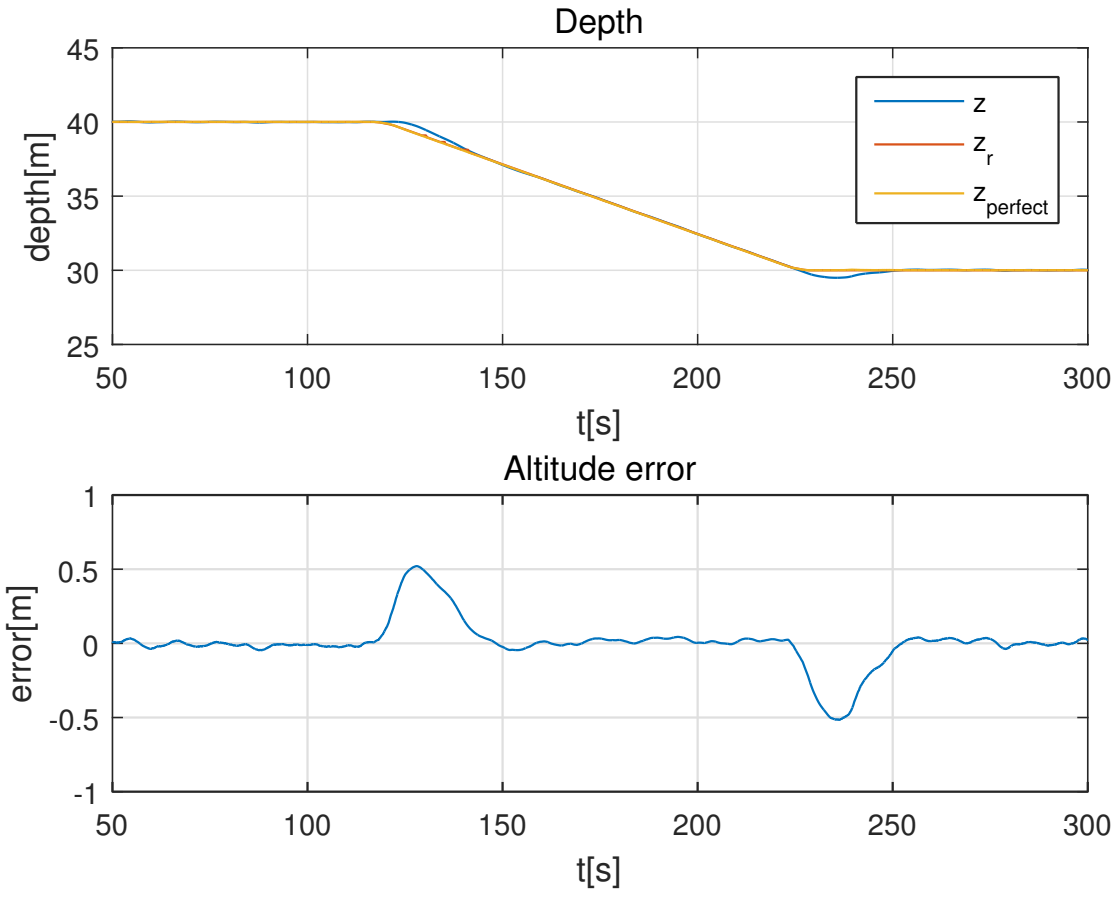


Figure C.1: Simulation using auto altitude and the constant slope map.

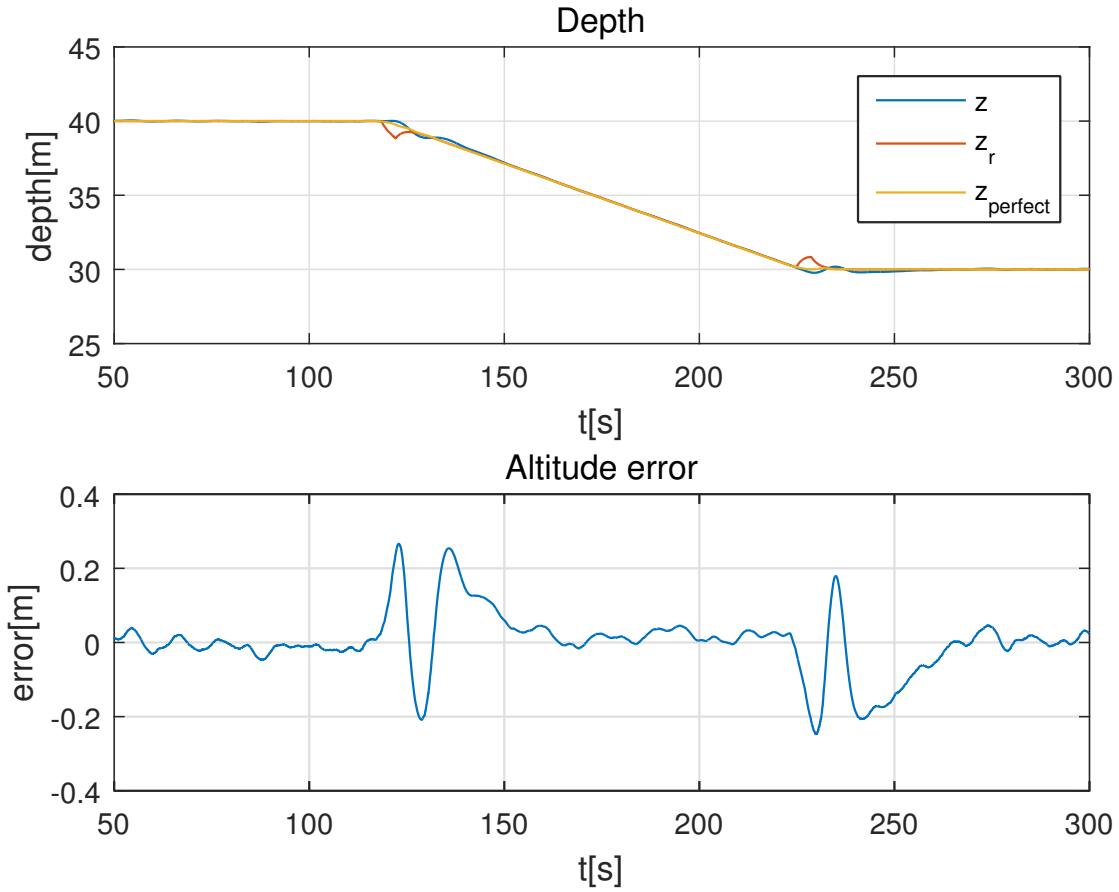


Figure C.2: Simulation using the highpass controller and the constant slope map.

C.2 Scenario: Sinus Curved Map

The second scenario using the sinus curved map shows the same effect as in the first scenario. Using auto altitude, as shown in Figure C.4, the vehicle overshoots by approximately 0.5 meters. Using the highpass controller, as shown in Figure C.4, the vehicle overshoots by approximately 0.2 meters.

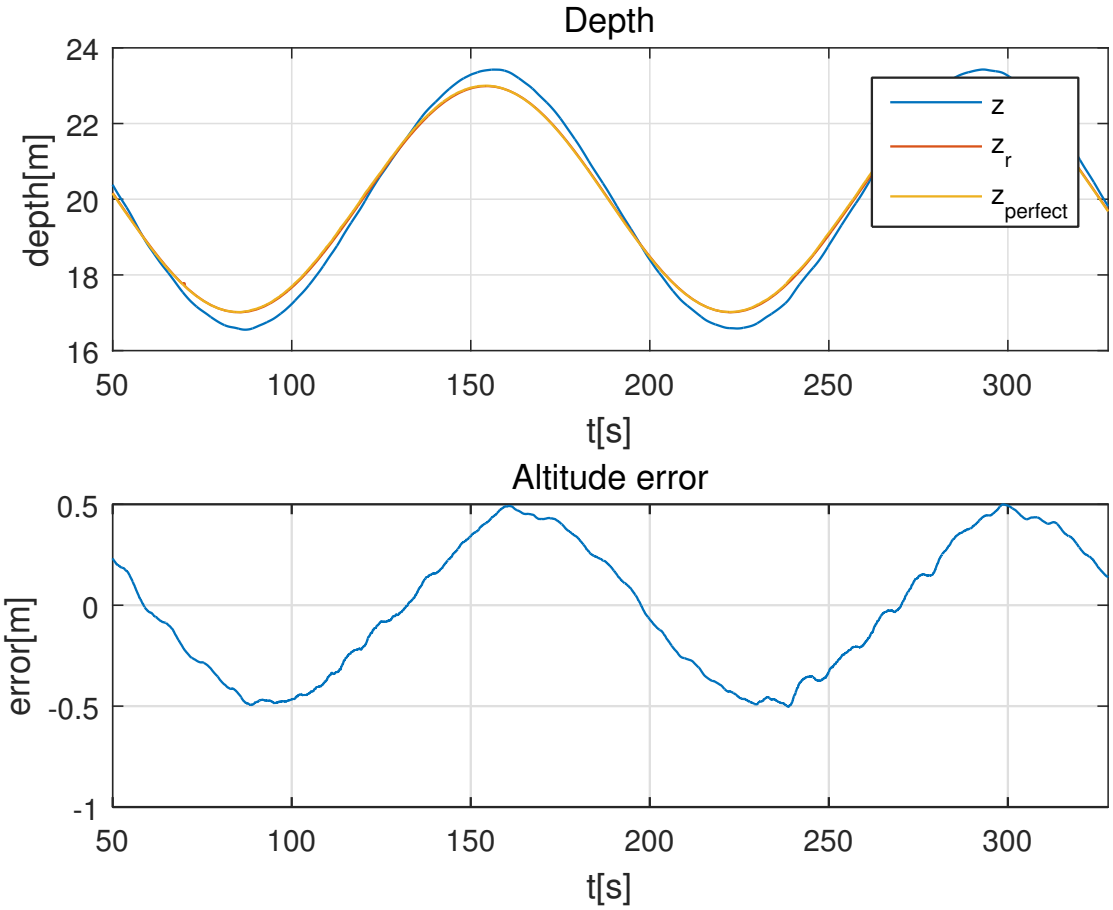


Figure C.3: Simulation using auto altitude and the sinus map.

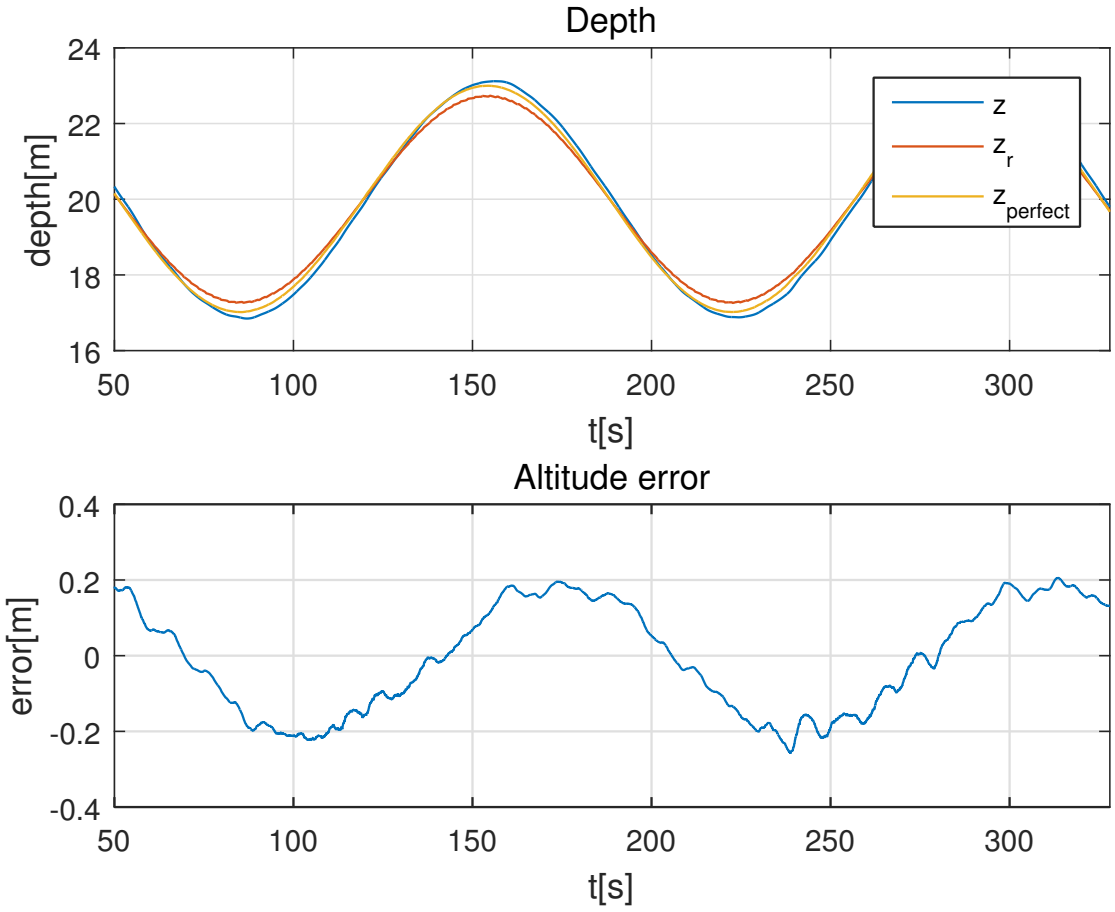


Figure C.4: Simulation using the highpass controller and the sinus map.

C.3 Scenario: Hopavågen Map

The third scenario uses the Hopavågen map. This is the scenario that shows the biggest difference between the two controllers. A simulation using auto altitude controller can be seen in Figure C.5. A simulation using the highpass controller can be seen in Figure C.6. The maximum error is reduced from around 1.4 meters to 0.6 meters by using the highpass controller.

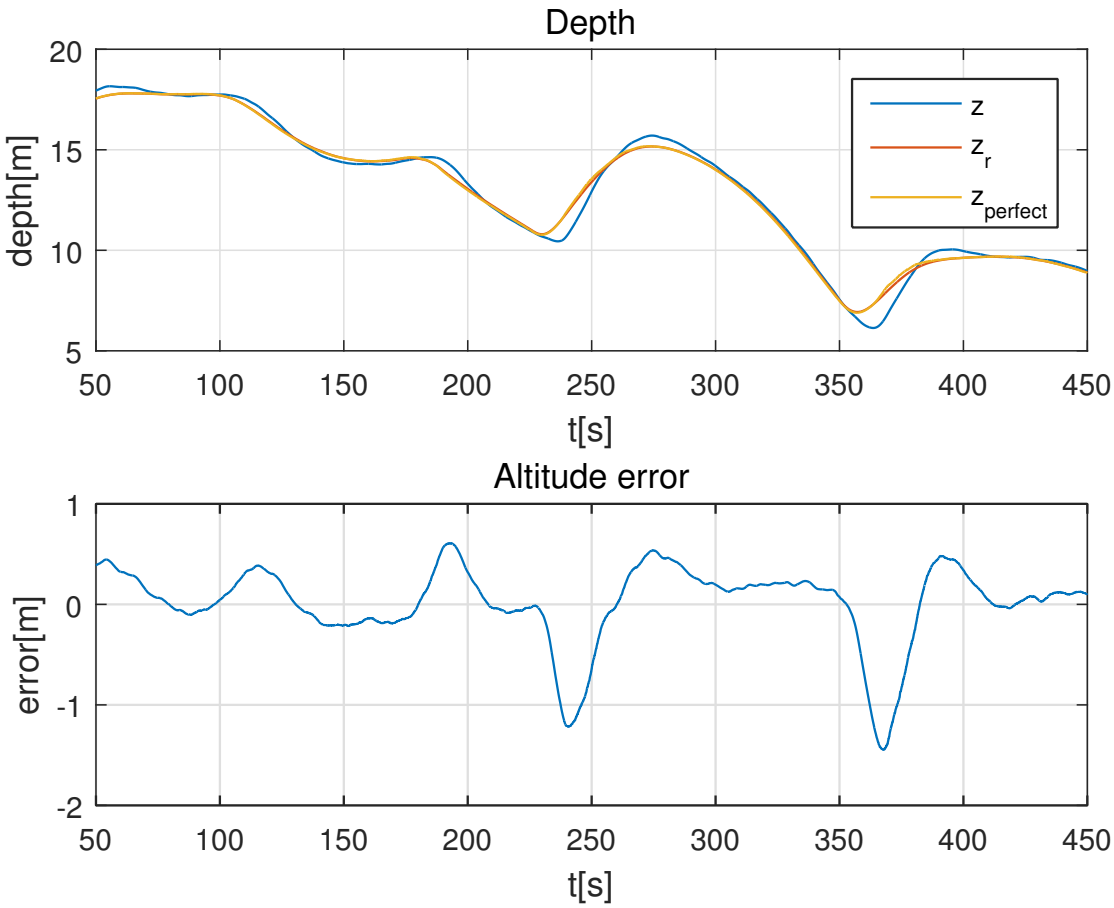


Figure C.5: Simulation using auto altitude and the Hopavågen map.

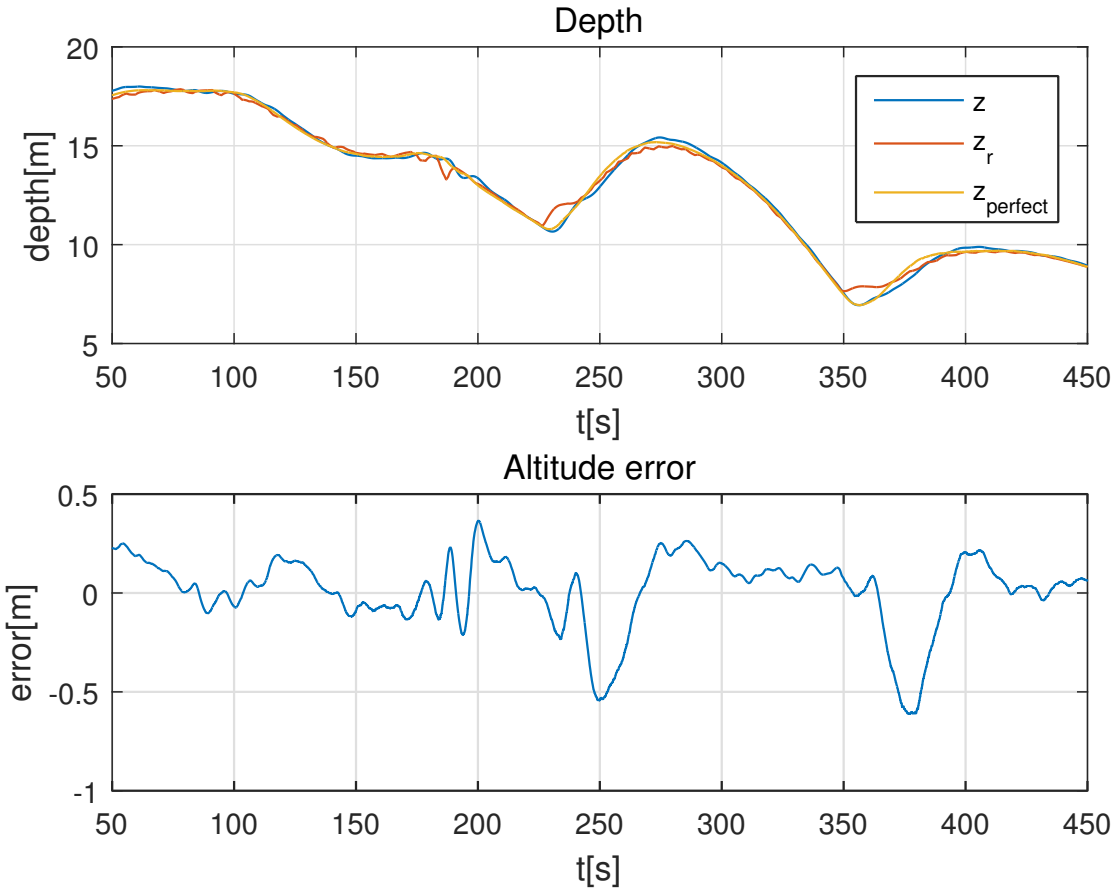


Figure C.6: Simulation using the highpass controller and the Hopavågen map.

Appendix D

Original Plots From the Field Tests

The altitude control missions used the beam ranges from the DVL to find the altitude. After the field tests were carried out, it was found that the range measurements returned from the DVL was the altitude of the beam and not the range (see Figure D.1). Since the altitude a was shorter than the beam range $|\mathbf{r}|$, the estimated altitude was always shorter than the real altitude.

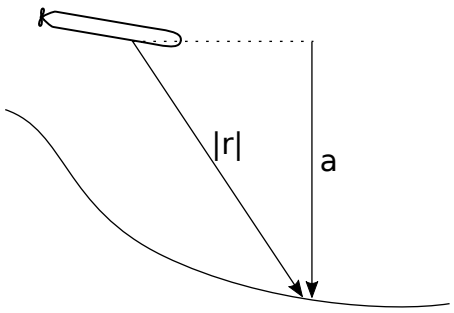


Figure D.1: The range measurement retrieved from the DVL during the field testing was the altitude a and not the range $|\mathbf{r}|$.

However, the difference is mostly constant. By adding 0.31 meters to the value of $z_{perfect}$, the error is reduced.

The results presented in Chapter 6 adds this difference to the value of $z_{perfect}$ so that the results are comparable. The original results are shown in this appendix.

D.1 REMUS Altitude Controller

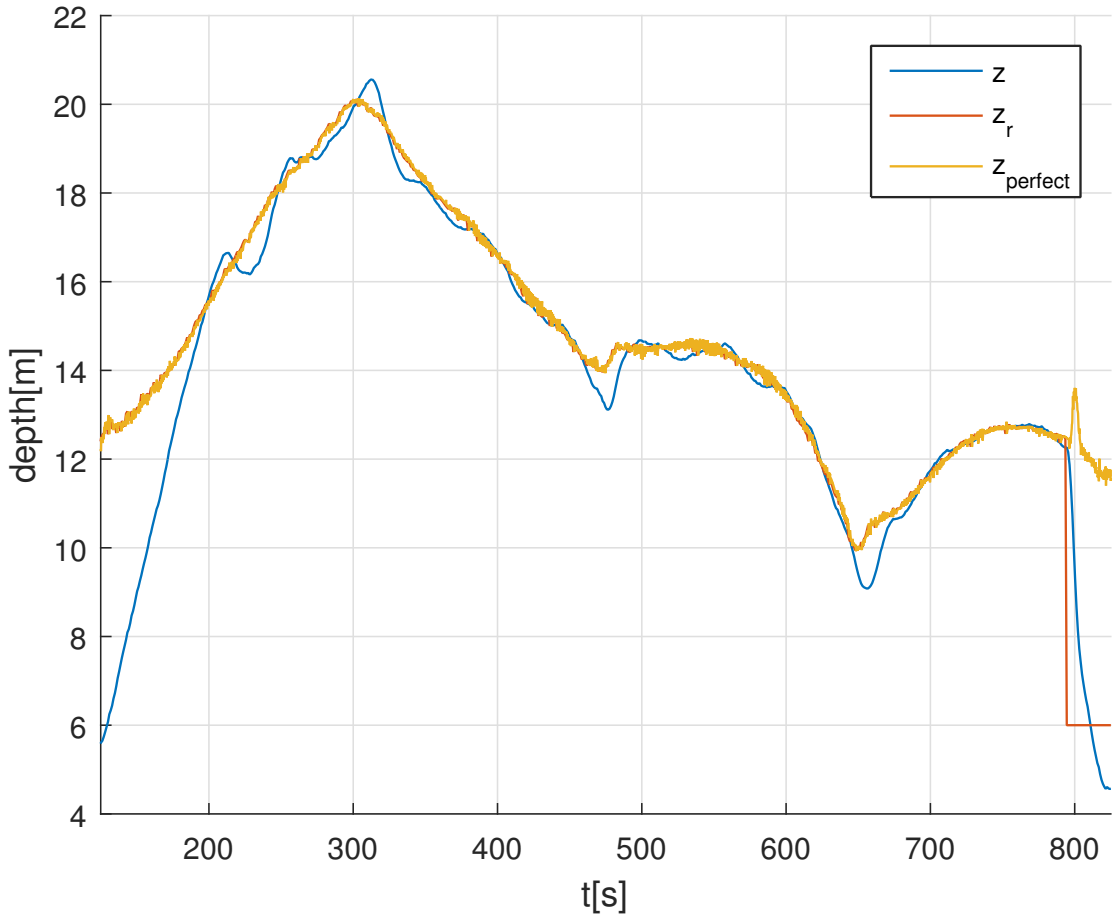


Figure D.2: Depth z and desired depth z_r for baseline mission using the REMUS altitude controller.

D.2 Auto Altitude Controller

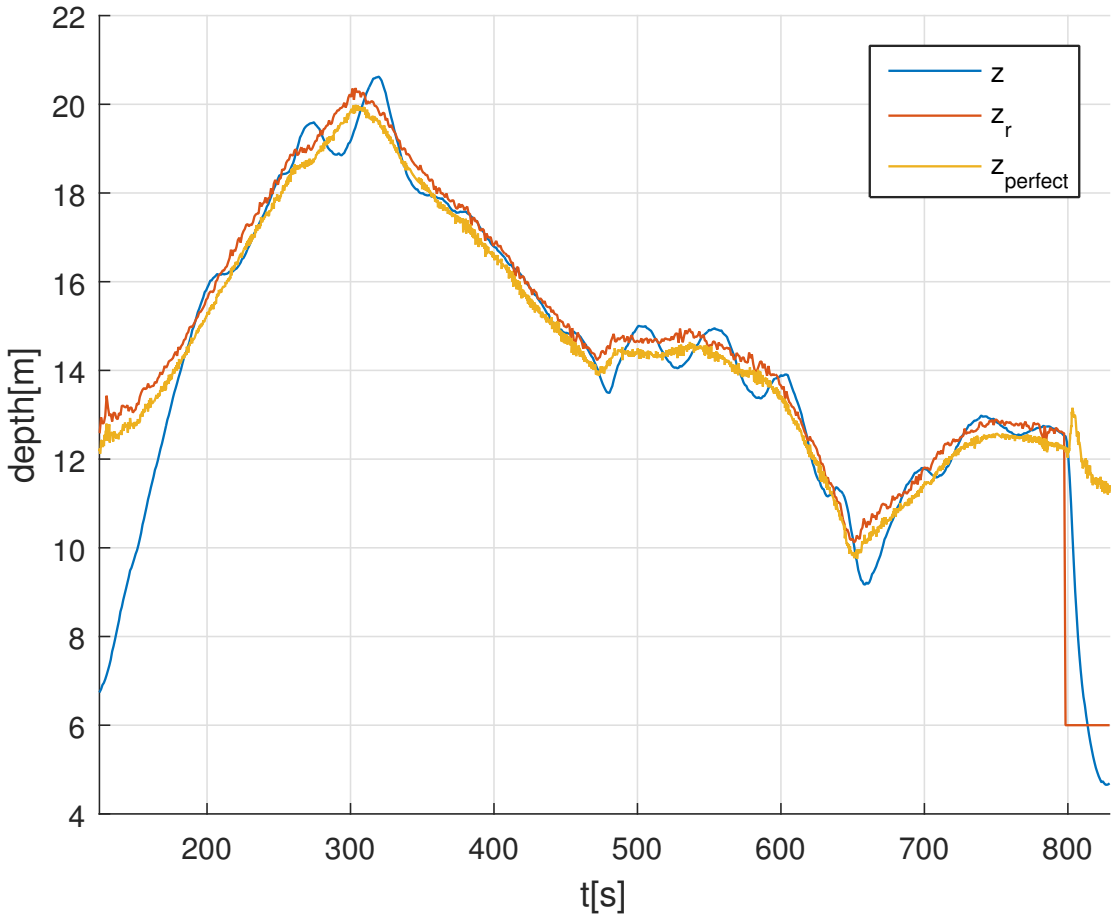


Figure D.3: Depth z and desired depth z_r for the auto altitude mission.

D.3 Highpass Controller

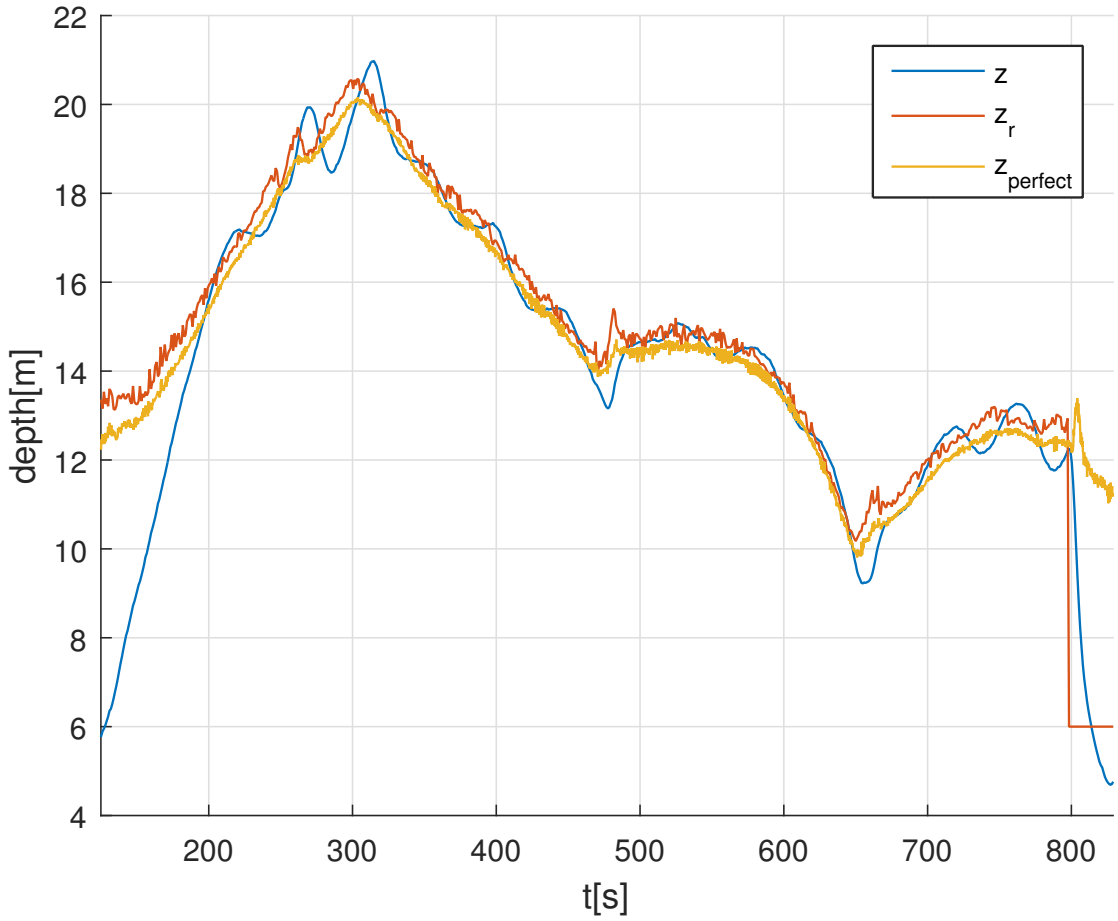


Figure D.4: Depth z and desired depth z_r for the first highpass altitude mission.

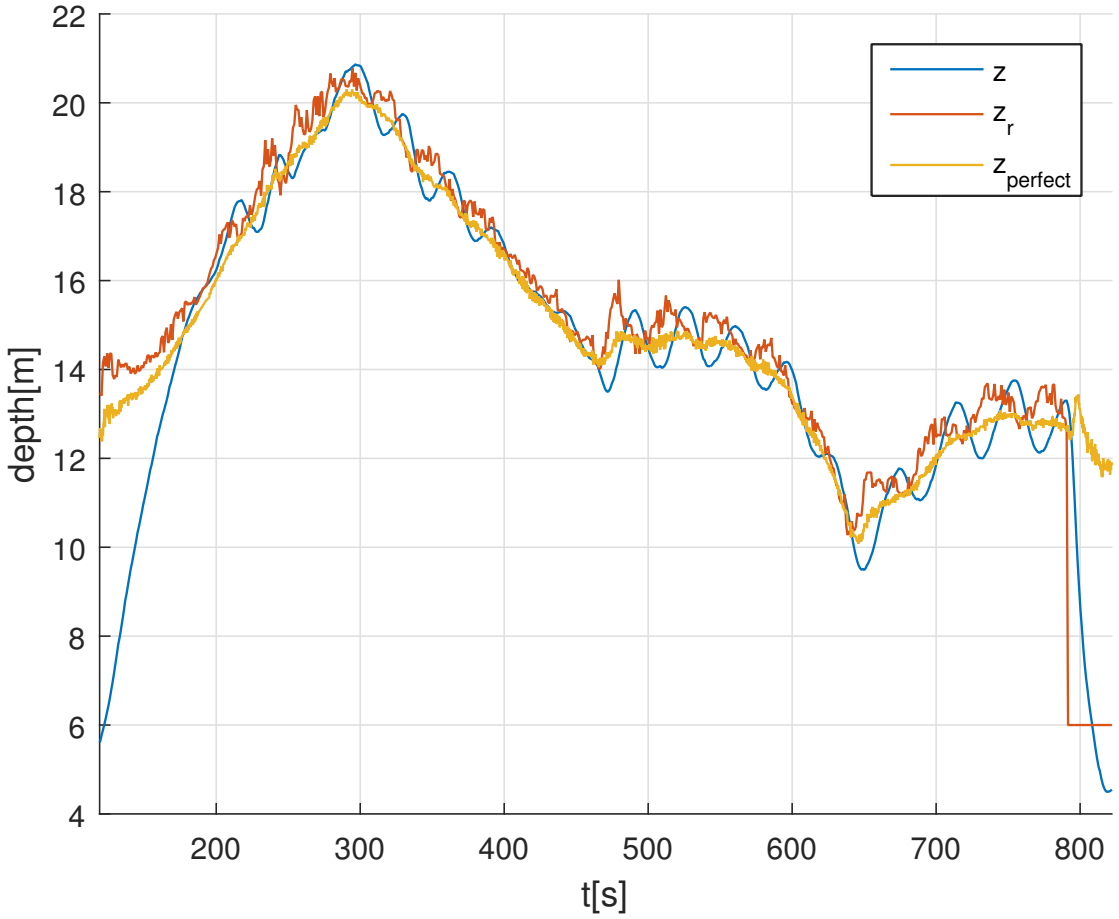


Figure D.5: Depth z and desired depth z_r for the second highpass altitude mission.

Appendix E

Attachments

This appendix lists the attachments to this thesis. The attachment is a zip file that includes the following folders.

AUVSim

This folder includes all source code written for AUVSim. AUVSim has additional dependencies on the Eigen3 and BOOST frameworks. These frameworks have not been attached due to their big size. To be able to run AUVSim, these frameworks must be downloaded into the vendor-folder.

dune-remus

This folder includes the DUNE source code that has been developed during the work on this thesis.

HuginDuneBridge

This folder includes the HuginDuneBridge.

imc

New IMC messages were created during the work on the new architecture described in Chapter 7. This folder includes the IMC repository.