Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Marine Technology

Kristoffer Rakstad Solberg

Manta v1: A Deliberative Agent Software Architecture for Autonomous Underwater Vehicles

Written in collaboration with Vortex NTNU

Master's thesis in Engineering and ICT
Supervisor: Asgeir Johan Sørensen, Thor Inge Fossen
January 2020

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF MARINE TECHNOLOGY

Manta v1: A Deliberative Agent Software Architecture for Autonomous Underwater Vehicles

Author
KRISTOFFER RAKSTAD
SOLBERG

Main-Supervisor
Prof. Asgeir Johan
SØRENSEN
Co-Supervisor
Prof. Thor Inge FOSSEN

February 9, 2020

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

MASTER DESCRIPTION SHEET

Name of the candidate: Kristoffer Rakstad Solberg
Field of study: ICT and Marine Technology
Thesis title (Norwegian): Manta v1 – En Deliberative Agent Software Arkitektur for Autonome Undervannsfarkoster
Thesis title (English): Manta v1 – A Deliberative Agent Software Architecture for Autonomous Underwater Vehicles

Background

Architecture design is one of the most important problems for an intelligent system. In Autonomous Unmanned Vehicles (AUVs) there is no human operator; thus, they function based on built-in machine intelligence and an on-board control system. The design of the mission control system and the underlying components is a major issue in the development of autonomous unmanned vehicles mostly due to computation-intensive processing, real-time execution constraints when dealing with a stochastic, dynamic and partially observable environment. System architectures for autonomous vehicles are extremely diverse, and no standardized solution has yet emerged. As of today there is hard to come across any end-to-end guide on how to design, implement and test software for this purpose, and even more so, its hard to come across any off-the-shelf software architecture that is open-sourced and available to the public.

Objective

This Master's Thesis aims to design, develop and implement a deliberative agent software architecture for high-level planning and execution of autonomous underwater vehicle missions, as well as low-level navigation, guidance and motion control. The system software architecture will be designed as a distributed system taking use of ROS open source robotic middleware for inter-process communication. The ease of programming and the flexibility in both high-level and low-level control are ensured by breaking down a complex system into small and functionally independent components, while the efficiency in execution is provided by a fast preemptive scheduler and event propagating kernel. Events are initiated by components and in turn trigger the scheduling of action procedures in other components. Finally, a report is presented, explaining the design and implementation of the proposed software architecture as proof of concept. The system capabilities will be implemented step-by-step through case studies, while the system capabilities will be demonstrated through 3-D simulation and field testing.

Work description

The Master's Thesis is written in collaboration with the student organization Vortex NTNU with the goal of completing a fully functional AUV to compete in the 2020 International RoboSub Competition, July 29 – August 4, 2020 | SSC Pacific TRANSDEC, San Diego, California, USA.

The work of this thesis should involve the following points:

- Software design philosophy and principles.
- Description of embedded electronics, hardware and proposed software architecture.
- Kinetic and kinematic modeling of vehicle and actuators.
- Create a high fidelity testbed / simulator for underwater vehicles.
  - Create a simulation world scenario of the Robosub competition ground.
  - Implement vehicle kinetics and kinematics with actuators.
  - Implement underwater vehicle cameras and sensors.
- Design, implement and test on-board navigation for an autonomous underwater vehicles.
  - Background theory.
  - Set up Extended Kalman Filtering for robot localization.
  - Set up Computer vision for underwater perception and object detection.
- Design, implement and test a guidance and motion control system for a underwater vehicle.
  - Background theory.
  - Create dynamic positioning controller for terminal operations.
  - Create a path following controller for transit operations.
- Design, implement and test mission control for autonomous missions.
  - Background theory.
  - Set up a finite state machine for scheduling and propagating events.
- Demonstrate the system in the field / testing pool.
  - Test robot localization.
  - Test controllers.
  - Test computer vision.

Specifications

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. 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 within 09.02.2020

Supervisor: Asgeir J. Sørensen **Co-advisor(s):** Thor I. Fossen

Trondheim, 30.01.2020, **Asgeir J. Sørensen,** _____

Abstract

Research in autonomous underwater vehicles (AUVs) has reached a level of maturity where robotic systems can be expected to efficiently perform complex missions involving intelligent agents in unstructured underwater environments without teleoperation. As the AUVs becomes more independent, the requirements for robust and reliability software increases to ensure safe operations, particularly for those missions that are expensive or risk-intensive.

The research of this thesis is motivated by the desire to design and create an open-sourced software architecture for AUVs that could support coordinated mission execution for many scenarios. First an review of the current state-of-the-art software architectures and explanation of design principles was conducted. After that, a software flow diagram of the Manta v1 was developed, showing the necessary components, their interdependence and the communication backbone. The implementation of each of the components in the architecture was done in a bottom-up fashion starting of with a mathematical formulation of the Manta AUV. Next, a high-fidelity testbed for 3-D simulation of the vehicle with actuators and sensors was created. Next, the necessary low-level navigation, guidance and motion control systems was designed, implemented and tested using the 3-D simulation testbed. Once all the low-level components was developed and thoroughly tested, a high-level mission control was develop to ensure a coordination and execution of mission.

Through comparison of results from testing the nonlinear PID controller and the nonlinear backstepping controller in simulation and in physical experiments, it can be concluded that the derived mathematical model of the Manta AUV is accurate. From physical experiments it can also be concluded that the implemented extended Kalman filter for localization, as well the nonlinear PID controller and nonlinear backstepping controller all achieves high performance much similar to what was seen in simulations. A lesser appealing result came from the computer vision object detection. It performance well in a simulated environment, however in physical experiments the object detection is more brittle and sensitive to noise. As recommendations for further work, it should be invested time in creating particle filters for mapping of the environment, once that is done it is possible to start investigating path planning and path generation. When it comes to object detection, it is suggested to shift from traditional computer vision techniques to using convolutional neural networks.

To this end, the Manta v1 software architecture has been developed as seen by Figure 1. It is an extensive software stack consisting of about 25.000 lines of code. Manta v1 supports the efficient execution of real-world missions involving multiple concurrent goals. The largest component of Manta v1 is its mission control, which continuously optimizes the execution of a mission as information about the world is acquired. The architecture is rich in functionality, distributes its computation, and performs efficient re-planning in an unknown, unstructured, and changing environment. This system has been demonstrated on the Manta AUV in an indoor experimental pool and extensively verified in simulation.

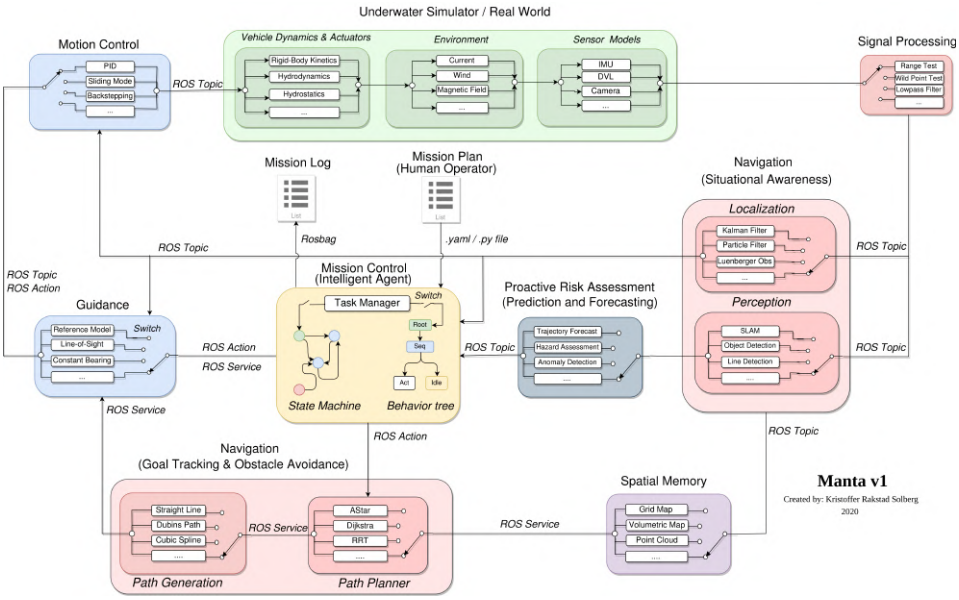


Figure 1: The Manta v1 software architecture.

Sammendrag

Forskning på autonome undervannsfarkoster (AUV-er) har nådd er modningsnivå der robotsystemer kan forventes å utføre komplekse oppdrag som involverer intelligent agenter i ustrukturerte undervannsmiljøer uten fjernstyring. Etterhvert som AUV-er blir mer uavhengige, stilles større krav til robust og pålitelige programvare for å sikre sikker drift. Dette gjelder spesielt for de oppdragene som er dyre og risikable.

Forskningen i denne oppgaven er motivert av ønsket om å designe og lage en open-source programvarearkitektur for AUV-er som kan støtte koordinert oppdragsutførelse for mange scenarier. Først ble det gjennomført en gjennomgang av de mest populære programvarearkitekturene og en forklaring vanlige designprinsipper. Etter dette ble det utviklet et flytskjema av den foreslåtte programvarearkitekturen, Manta v1. Denne viser de nødvendige komponentene, deres avhengighet av hverandre og det anvendte kommunikasjonsnettet. Implementeringen av de forskjellige komponentene i programvaren ble gjort i en 'bottom-up' stil. Det startet med en matematisk formulering av undervannsdronen Manta. Deretter ble det ble et 3-dimensjonalt simuleringsverktøy med sensorer og aktuatorer utviklet. Deretter ble de nødvendige navigasjonsystem og kontrollere designet, implementert og testet i dette simuleringsverktøyet. Når disse var ferdig utviklet og grundig testet, ble 'mission control' for høynivå-kontroll utviklet for å sikre koordinering og utførelse av oppdrag.

Gjennom sammenligning av resultater fra testing av den ulinære PID-kontrolleren og den ulinære backstepping-kontrolleren fra simulering og fysiske eksperimenter, kan det konkluderes at den formulerte matematiske modellen av underfarkosten Manta er nøyaktig. Fra fysiske eksperimenter man også konkludere med at det implementerte 'extended Kalman filteret' for lokalisering, så vel som den ulinære PID-kontrolleren og den ulinære backstepping-kontrolleren alle oppnår høy ytelse som ligner mye på det som ble observert i simuleringer. Et mindre tiltalende resultat kom fra datasyn og bildegjenkjenning. Datasyn yter godt i et simulert miljø, men i fysiske eksperimenter strekker det ikke til da resultatene blir for støyete. Som anbefaling for videre arbeid, bør det brukes tid på å lage partikkelfilter for 'mapping' av miljøet. Når dette er gjort kan man begynne å utforske metoder for baneplanlegging. Når det gjelder datasyn bør det investeres tid å implementere nevrale nett istendenfor tradisjonelle metoder for datasyn.

For dette formål er programvarearkitekturen Manta v1 utviklet. Det er en omfattende kode som består av omtrent 25.000 linjer med kode. Manta v1 støtter effektiv utførelse av oppgrad i den virkelige verden som involverer flere konkurrerende mål. Den mest signifikante delen av Manta v1 er dens 'mission control', som kontinuerlig optimaliserer utførelsen av et oppdrag etter hvert som ny informasjon om verdenen kommer inn. Arkitekturen har rik funksjonalitet, distribuert data-prosessering, og effektiv re-planlegging i ukjente, ustrukturerte og skiftende omgivelser. Dette systemet har blitt kvalitetsikret igjennom både simuleringer og fysiske tester.

Preface

This Master's thesis is written as the final part of my Ms.c degree in Engineering and ICT specializing in Marine Technology. The thesis is written at the Department of Marine Technology at the Norwegian University of Science and Technology (NTNU) in Trondheim during autumn of 2020. It is a continuation of the project thesis *A Simulation-Based Approach to Software Development for Autonomous Underwater Vehicles* written in spring 2019.

I would like to acknowledge the guidance of my main supervisor, Professor Asgeir Johan Sørensen, who shaped my ideas while allowing me freedom in my research endeavors. He has been an invaluable resource when it comes to topics within the fields of autonomy and high-level control and has also provided encouragement and support throughout the process of writing this Master's thesis. I would also like to acknowledge my co-supervisor, Thor Inge Fossen, for motivation, feedback and discussions on topics within fields of guidance, navigation and low-level control. He has been a valuable resource for refining and sparking new ideas for the work of this thesis.

I appreciate the camaraderie of my fellow teammates at Vortex NTNU, who made my Master's thesis an enjoyable and memorable learning experience. In particular, I want to thank Øyvind Denvik and Ambjørn Waldum for our numerous enlightening conversations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abbreviations

Abbrevation	=	Description
AUV	=	Autonomous Underwater Vehicle
UUV	=	Unmanned Underwater Vehicle
UAV	=	Unmanned Aerial Vehicle
ROV	=	Remotely Operated Vehicle
CO	=	Center Of Orientation
COB	=	Center Of Bouyancy
COG	=	Center Of Gravity
COA	=	Circle of Acceptance
SOA	=	Sphere of Acceptance
DP	=	Dynamic Positioning
DOF	=	Degree Of Freedom
DVL	=	Doppler Velocity Log
IMU	=	Inertial Measurement Unit
INS	=	Inertial Navigation System
NED	=	North-East-Down
ENU	=	East-North-Up
GUI	=	Graphic User Interface
FSM	=	Finite State Machine
BT	=	Behavior Tree
ROS	=	Robot Operating System

CHAPTER 1

INTRODUCTION

1.1 Motivation and Background

Autonomous Underwater Vehicles (AUVs) have a wide range of applications in marine geoscience, and are increasingly being used in the scientific, military, commercial and policy sectors. Their ability to operate autonomously and make decision independently of a host vessel makes them well suited to exploration of extreme environments; from the world's deepest hydrothermal vents to beneath polar ice sheets. AUVs can gather data with sufficient resolution in time and space to provide an understanding of nature's dynamics. They are well-suited to carry out swathe mapping (detailed 3-D mapping of the seabed using sonar), and large-area surveys [77]. They can carry multiple payloads, allowing synoptic coverage; for example an AUV may simultaneously image the seafloor while measuring water quality. Moreover AUVs reduce the risk or cost compared to other sampling methods such a SCUBA diver, towed platform, or ship. Figure 1.1 show current state of the art AUVs for subsea inspection.

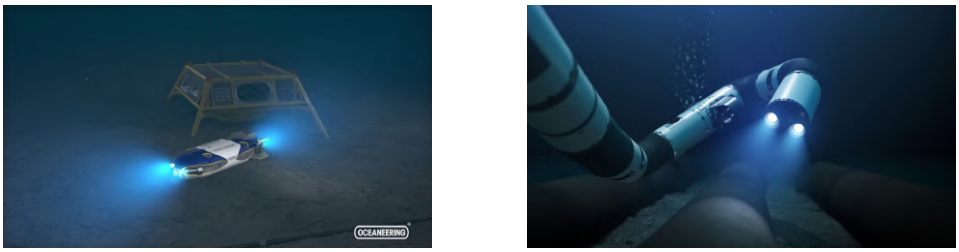


Figure 1.1: AUVs performing underwater inspections. *Courtesy: Oceaneering (left picture), Equinor: Eelume underwater robot (right)*

1.1.1 What is an AUV?

An AUV is an unmanned, untethered, sub-sea vehicle with sensors, actuators and on-board intelligence to carry out a mission without human operator interference or support from a ship [46]. The current generation of AUVs usually range in length from 1.5 – 5.5 meters and weigh in air from 20 to more than 1,400 kilograms. Most are torpedo-shaped to reduce drag. Subjected to difference in water density, AUVs are ballasted to ensure slight positive buoyancy at each operation. This ensures that the AUV will surface even if onboard control systems have failed.

Fully autonomous operations are supplied with on-board energy, which is used for thrust generation and sensor systems. Most AUVs use specialized batteries, although some AUVs have used fuel cells or rechargeable solar power. Certain AUVs, such as gliders, minimize energy demands by allowing gravity and buoyancy to propel them. Others can stop, hover, and move like blimps or helicopters do through the air. Solar-powered AUVs can spend a portion of their time at the surface, blurring the distinction between undersea and surface vehicles [62].

In comparison to an AUV you also have remotely operated vehicle (ROV) that are unoccupied underwater robot connected to a ship by a series of cables. These cables transmit command and control signals between the operator and the ROV, allowing remote navigation of the vehicle.

1.1.2 What is Autonomy?

Autonomy - in the context of a robotics - is a systems ability to [90]:

- *Perform complex tasks under significant uncertainties when operating in an unstructured environment.*
- *Be highly dependable and able to handle both external events and internal faults including reconfiguration, planning and re-planning.*
- *Adapt, learn and improve.*
- *Establish long-term goals, hence making deliberative choices.*

For missions involving unmanned underwater vehicles (UUVs), there is a spectrum of autonomy ranging from basic automation (mechanistic execution of action or response to stimuli) through to fully autonomous systems able to act independently in dynamic and uncertain environments. The goals the system is trying to accomplish are provided by another entity; thus, the system is autonomous from the entity on whose behalf the goals are being achieved. In contrast, an *automated system* follows a script. If an automated system encounters an un-planned event, it will halt operations and wait for human intervention [3, NASA, 2010].

1.1.3 When is Autonomy Needed?

Autonomous systems research seeks to improve performance with a reduced burden on crew and ground support personnel, achieving safe and efficient control, and enabling decisions in complex and dynamic environments.

The situations we most care about are the situations when the only way to do something is through adding more autonomy to a system. That is, for the times, those missions and those tasks, when autonomy is the enabling technology that allows something to actually be accomplished [25]. Some of these situations can be:

1. When the frequency of decision making exceeds **communication constraints** (delays bandwidth, and communication windows). Autonomy also continues to work when no communication is possible at all because of interference or physical obstacles.
2. When **time-critical decisions** (control, health, life-support, etc) must be made onboard the system vehicle. Even with negligible communication delays, autonomy can provide computer-speed reaction times in places where human response time would be too slow with respect to the environment, as for example in collision avoidance systems.
3. When **variability in training, proficiency**, etc, associated with manual control is unacceptable.
4. When decisions can be better made using **rich on-board data** compared to limited downlinked data when you have a surface-based operations center in the communication loop. On-board models have near-instant insight into the immediate status of the vehicle and its surroundings, these autonomous systems can make better decisions than the remote human operators when responding to anomalies.
5. When local decisions **improve robustness** and **reduce complexity** of a system architecture, often made possible by having decentralized control and recovery procedures at lower levels [2].
6. When autonomous decision making can **reduce system cost** and **improve performance** through constant learning and progression in the way tasks are performed.

1.1.4 Why Autonomy is Hard?

*"There are **known knowns**. There are things we know that we know. There are **known unknowns**. That is to say, there are things that we now know we don't know. But there are also **unknown unknowns**. There are things we don't know we don't know" - Donald Rumsfeld, 2002*

The quotation above from Donald Rumsfeld is a reminder that the real world is highly uncertain and ever changing. As we are trying to make systems more robust and resilient, and be able to operate independently, they have to deal with all these uncertain factors. They not only have to deal with the known knows and the unknown knowns, but also the unknown unknowns. A part of that is reflected by the fact that the world is not an exponential distribution, in fact it is a heavy tailed distribution [3] (see Figure 1.2. That is, the longer you operate and the more that you try to do things, there is still a probability that you will encounter unexpected events. That is things that you have never ever seen before, or things that you only see one time and never again. All the models that are being used to build control systems or in terms of state awareness or for understanding of what we are trying to accomplish. These are all approximations of the world. Modeling the unknowns and modeling uncertainty are difficult tasks still to this day, and the computation that supports all these models is - not yet - instantaneous and infinite because there are limitations on memory, data storage and processors.

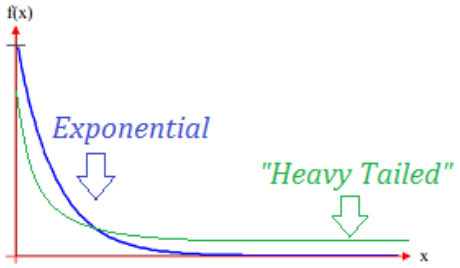


Figure 1.2: Exponential vs heavy tailed distribution. First axis represents operating time, while the second axis represents probability density function for a particular operating scenario

This is why arguably the top technical challenge in autonomy is **verification and validation** of autonomous systems [57]. Large software projects have such complex software that exhaustive and manual exploration of all possible cases is not feasible. Human rated autonomous systems are particularly challenging. Verification techniques are needed to more fully confirm system behavior in all conditions.

1.2 Problem Formulation & Research Questions

The objective of this Master's thesis has been to design, develop and implement a deliberative agent software architecture for high-level planning and execution, with an underlying low-level navigation, guidance and motion control. Software architectures for AUVs are extremely diverse, and a standardized solution has yet to emerge. There is hard to come across any end-to-end guide on how to do to that.

In the process of formulating a software architecture for the Manta AUV, there are three research questions that need answering:

1. *Traditionally, system identification of underwater vehicles are done through experimental study in marine labs. Will an empirical and analytical study suffice in identifying accurate hydrodynamic model representation for underwater vehicles?*
2. *Robust robotic software is paramount to high performance capabilities across variety of tasks. Intuitively, adding more capacity to an application should increase the components performance. What are the most effective strategies to ensuring a robust and high performing software for a growing code stack?*
3. *Intelligent agents cannot work just in a purely reactive capacity. How can we enable for a deliberate and long-term robot behavior to be established?*

1.3 Stakeholders and Collaborators

The master thesis is written in collaboration with the student organization Vortex NTNU with the goal of completing a fully functional AUV to compete in the 2020 International RoboSub Competition, July 29 - August 4, 2020 — SSC Pacific TRANSDEC, San Diego, California, USA.

Each year up until 2019 Vortex NTNU have conceived, designed, and built an ROV from scratch to compete in the MATE international ROV competition at the King County Aquatic Center, Washington, USA. From 2020 and onwards Vortex NTNU will take on the task of building its first AUV. This means that the team will be entering a new competition and face different challenges to push the boundaries even further.

1.3.1 Vortex NTNU

Vortex NTNU is an independent student organization at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway. The team is composed of students from different engineering disciplines from bachelor and master degree programs. The purpose of the organization is to provide an ideal opportunity for ambitious students to explore and develop their talents and skills in a collaborative undertaking. Through the project, the students demonstrate that they can deliver a complex and integrated product in the demanding environment of an underwater competition. The current underwater vehicle under development by the Vortex NTNU team is the Manta AUV, as seen in Figure 1.3.

Figure 1.3: The Manta AUV *Courtesy: Manta AUV - Vortex NTNU*

1.3.2 RoboSub

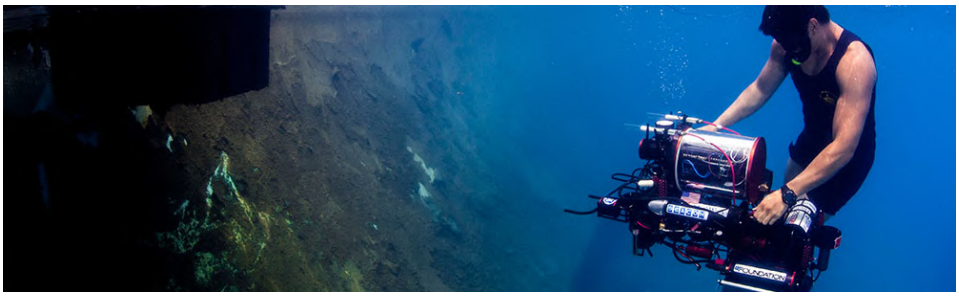


Figure 1.4: Cornell CUAUV at Robosub. *Courtesy: Cornell University — CUAUV*

RoboSub is an underwater robotics competition originally launched in 1997 co-sponsored by the Association for Unmanned Vehicle Systems International (AUVSI) Foundation and the Office of Naval Research (ONR) [73]. In later years the competition has also gained funding from companies such as SpaceX, Blue Origin, Northrop Grumman, NVIDIA, SolidWorks and several more. The goal of RoboSub is to advance the development of autonomous underwater vehicles (AUVs) by challenging a new generation of engineers to perform realistic missions in an underwater environment. This event also serves to foster ties between young engineers and the organizations developing AUV technologies. The competition is open to high school, college and university teams from around the world. Since about 2002, it has been held each summer at the U.S. Navy Space and Naval Warfare Systems Center Pacific's TRANSDEC Anechoic pool in San Diego, California.

In Figure 1.4 and 1.5 you can see the competition course and one of the underwater intervention tasks the robots will be facing.

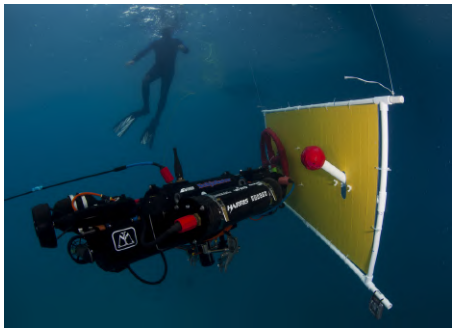



Figure 1.5: The TRANSDEC competition pool *Courtesy: flickr.com/Robonation (left picture) and flickr.com/US_NAVY (right picture)*

1.4 Contributions

The main contributions of this thesis are related to the analysis, design and implementation of the Manta v1 software architecture. The quality of the software architecture comes from synthesizing ideas and knowledge from various fields such as vehicle hydrodynamics, cybernetics & robotics, computer science, embedded systems and software engineering.

The main contributions of this thesis can be summarized as:

- A methodology and Matlab source code (see Appendix D.1 and D.2) for modeling rigid-body kinetics, hydrostatics, hydrodynamics and thruster dynamics for underwater vehicles based on empirical and analytical analysis.
- A complete implementation of a simulated Manta AUV with corresponding mathematical model in Gazebo using UUV Simulator with a 1:1 scale design of the Marine Cybernetics Laboratory and the Robosub Transdec experimental facility.
- A complete design of a deliberative agent software architecture - Manta v1 - for AUVs. The design comes with supporting case studies, simulations and physical experimentation for software verification.
- A complete software implementation of the Manta v1 software architecture using the robot operating system (ROS). As of today the software stack consist of about 25.000 lines of code written in C++, Python and C.

1.5 Thesis Outline

In an attempt to maintain agile and continuous development, the software has been developed in an bottom up fashion in the order of chapters. Starting of, chapter 2 will give some general insight into design principles and philosophy held by the author. Chapter 3 will introduce the Manta AUV, and also sketch out the flow diagram of Manta v1. The setup of the chapters from 4 to 8 is meant to allow for rapid prototyping and testing in accordance with agile and continuous development. Each chapter will start of with a introduction explaining the chapters role in the system architecture, and then proceed with background theory. Each chapter will finish up with one or several case studies explaining the design, development and implementation of that particular component in the Manta v1 software architecture. Each case study will end with an discussion and conclusion of the results.

A short summary of each chapter is presented below:

Chapter 2 - Software Design will go into details of common software design techniques and necessary system requirements to develop intelligent agents. This chapter will also discuss state-of-the-art software architectures for autonomy.

Chapter 3 - The Manta v1 Electronics and System Software will explain some of the underlying systems software, electronics and hardware of the Manta AUV. There will also be presented an overview of the proposed Manta v1 software architecture.

Chapter 4 - Modeling of Autonomous Underwater Vehicles will explain reference frames, transformations and hydrodynamics for AUVs. A case study is presented; finding rigid-body kinetics, hydrostatics and hydrodynamics for underwater vehicles using an empirical and analytical approach.

Chapter 5 - Underwater Simulation Environment presents a listing of 3-D simulation requirements, as well as a case study for creating a high-fidelity simulation environment with the Manta AUV using the open-source Gazebo robot simulator and the UUV simulator plugins.

Chapter 6 - Onboard Autonomous Underwater Navigation will go into details on how the AUV can compute its own location and immediate surroundings by fusing all sensor data in real-time. Two case studies are presented, one for implementation of extended Kalman filter and the other for implementation of computer vision object detection.

Chapter 7 - Guidance and Motion Control of Autonomous Underwater Vehicles will explain the concepts of guidance and motion control. After that, two case studies are presented. First the design of a nonlinear PID controller for low-speed waypoint tracking in a terminal state will be presented. And after that, the design of a nonlinear backstepping controller for path following in transit mode using line-of-sight guidance.

Chapter 8 - Mission Control for Autonomous Underwater Vehicles will explain the four components that make up the Manta v1 mission control. The chapter also goes into depth of the two most common task controllers, namely the finite state machine and behavior tree. Lastly a case study is presented, investigating mission control using a non-deterministic augmented hierarchical state machine for task control.

Chapter 9 - Physical Experiments Through a series of tests in the Marine Cybernetics Laboratory, this chapter will check whether the system is well-engineered, error-free, and so on. Verification will help to determine whether the software is of high quality.

Chapter 10 - Discussion and Recommendations for Further Work shows a general discussion in light of the research questions. The performance of the proposed software architecture is evaluated based on the results obtained in the case studies through simulations and physical experiments. This chapter also discusses the potential limitations and challenges of the proposed architecture.

As the source code of Manta v1 is 25.000 lines of code and to much to fit in the Appendix, there is provided a table with URLs directing the different source files in the GitHub repository. The table is found in Appendix A.

The whole Manta v1 repository can be found at:

https://github.com/Sollimann/Manta_v1

while the 3-D simulation testbed can found at:

https://github.com/vortexntnu/uuv-simulator

CHAPTER 2

SOFTWARE DESIGN

Software design is the most important phase of the software development cycle. Thinking about how to structure code before you start writing it is critical. Changes and updates will inevitably arise. Good software design plans and makes allowances for added features, algorithm changes, and new integration. The challenge about designing software from the beginning of a project is the future needs may not be clear at the outset. Software design best practices anticipate a variety of future needs. It implements best practices from the beginning, instead of hacking together a solution every time a new problem arises. Through this chapter you will get to know some common software design philosophies and principles. Finishing up, related work in software architectures will be presented.

2.1 Principles of Good Software Architecture Design

Software architecture is the representation of a system that describes its major components, the relationships between the components and the behaviors these components exhibit [42]. Its purpose is to give an overview of the software system in question and provide a basic high-level understanding of its operation. Think of it like the Google map of your hometown. It doesn't provide every detail of the town, rather, it provides sufficient detail to obtain a basic understanding and facilitate further discussion. For this reason, an architecture is commonly used in the area of system design.

When designing and implementing a software architecture, it is important to pick some design principles to stick with throughout the process. The design principles are important because they should be driving everything you do and every decision you make when writing code and structuring a project. Below are some of the more common design principles known to developers.

2.1.1 Reusable Code

Reusability principles must be followed when designing the software for any complex and long-term project. The ability to reuse designs relies in an essential way on the ability to build larger things from smaller parts, and on the independence of the input and output of those parts from their use in the project. Each functionality must interface the software architecture in a rigorous, universal and well-defined fashion.

2.1.2 Simplicity

The goal of software design is simplicity. Each class, method and module in your code should have a single purpose. Every new taks should get its own module that can be used and modified independently. This minimizes regressions and makes the code easier to use. Embrace simplicity, do not add complexity where simpler solutions will avail. Often, it is tempting to think you have a brilliant solution, but if there is a simpler way to accomplish the same task, you should go with that solution.

2.1.3 Modular Programming

A major concern of software development using traditional procedural techniques is when it comes to big projects where complexities surround the large software project to be designed. In projects that may involve tens of thousands of lines of codes, having a clear knowledge of what a segment of code does become more difficult. The foundation of good software design is separation of concerns [4]. This means that you divide your software into component parts and build each part once. Avoid code repetition. Always place code that you're likely to use again in a utility class and make it accessible throughout the application. When you need to update that code in the future, you only need to edit it in one place, instead of searching for the various locations where you repeated the code.

When you need a given component, you can call it and use it in an abstraction layer. This separation is called modularity, and it's a key to scalable, maintainable software architecture.

2.2 Picking a Network Design for your Software Architecture

Software design principles are concerned with providing means to handle the complexity of the design process effectively. Effectively managing the complexity will not only reduce the effort needed for design, but can also reduce the scope of introducing errors during design. In the design of our software architecture for our autonomous system there are three major design aspects that needs to be determined. What kind of network topology will the system have? What is the means of communication between nodes in the network?

and what type of agent architecture should the system have? In the next subsections you will get to know some of the options.

2.2.1 Network Topology

Network topology (in software context) refers to the layout of a network and how different nodes in a network are connected to each other and how they communicate. Topologies are logical and shows how the data passes through the network from one node to the next. When choosing a software architecture, it is important to pick an overall networking topology for how information is shared between nodes in our system. Picking a fitting topology is significant for the functioning of networks and plays a crucial role in performance.

There are broadly three different kinds of network as highlighted in Figure 2.1.

Centralized System

In case of a centralized system, there is a central network owner. The central network owner is a single point of contact for information sharing [72]. The biggest issue with a centralized network is with a single central owner it also becomes a single point of failure. Further, with a single copy stored with the owner, every instance of access to resource leads to an access issue with time.

Decentralized System

As for a decentralized system, we have multiple central owners that have the copy of the resources [72]. This eliminates the biggest problem of single point of failure with the centralized system. With multiple owners, if a particular central node fails, the information and services can still be accessed from other nodes. Further, with multiple owners the speed of access to the information is also reduced.

Distributed System

The distributed network is the decentralized network taken to the extreme [72]. It avoids the centralization completely. The main idea for the distributed network lies in the concept that everyone gets access, and everyone gets equal access.

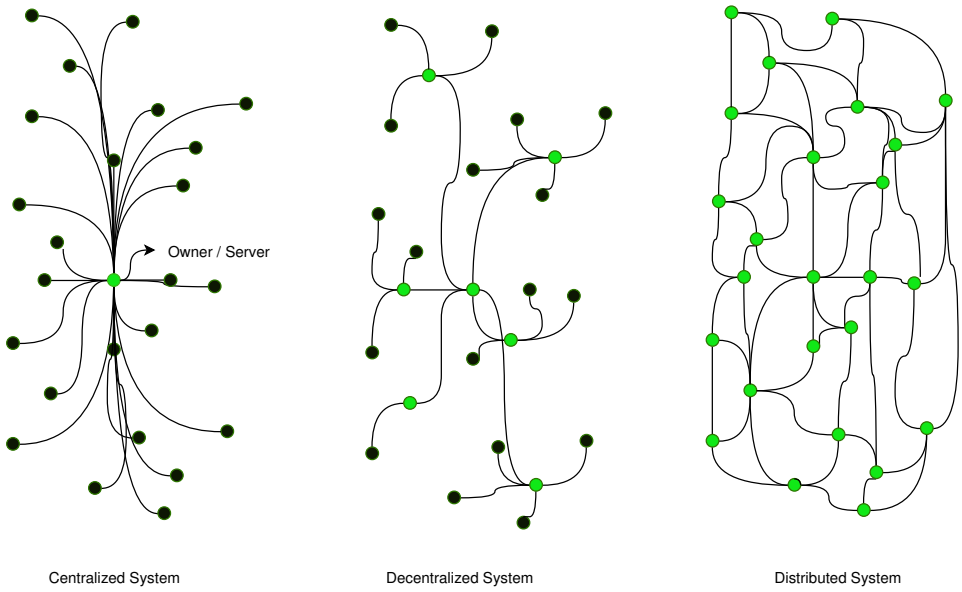


Figure 2.1: Illustration of the three different kinds of network topology

The following head-to-head comparison in the table below is meant to give an even better understanding of the differences of the three network topologies. The ratings ranges from low, moderate, to high.

	Fault tolerance	Maintenance	Scalability	Development	Evolution
Centralized Systems	*Low*	*Low*	*Low*	*High*	*Low*
Decentralized Systems	*Moderate*	*Moderate*	*Moderate*	*Moderate*	*High*
Distributed Systems	High	High	High	Moderate	High

If you want to grow fast and scale is not a concern, go for centralized. If you are solving a problem of scale and not in hurry, choose distributed. If in hurry, choose decentralized. These are three ”thumb rules” important to have in mind when picking a network topology [92].

2.2.2 Network Communication

Distributed Systems are composed of various hardware and software (collectively called components) that communicate with each other only by transfer of messages. These components are placed inside a single network. In the next section, four different means of communication pattern in a network is presented.

Layered Pattern

The layered architectural pattern - as seen by Figure 2.2 - works in layers that help subtasks which are conceptually different from each other to be implemented and worked on simultaneously but within two different layers [50]. In this pattern, a layer of higher abstraction can use lower abstraction services but it doesn't work the other way round.

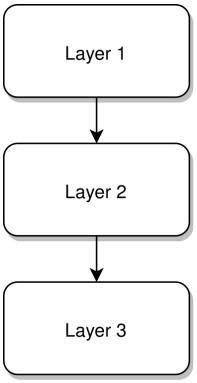


Figure 2.2: Layered Pattern. *Courtesy: [50, Distributed Systems]*

Client-Server Pattern

In the client-server architectural pattern - as seen by Figure 2.3 - the client component asks for services from the server component, which the server readily provides to the client [50]. Any server is always active for its clients. The client and the server may contain different processors.

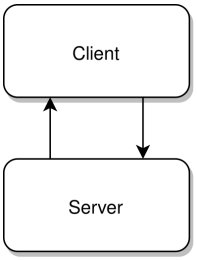


Figure 2.3: Client-Server Pattern. *Courtesy: [50, Distributed Systems]*

Master-Slave Pattern

In a master-slave architectural pattern - as seen by Figure 2.4 - the master component distributes the work among identical slave components, based on the divide and rule principle. This pattern supports parallel computing [50] and are often used in full tolerant systems.

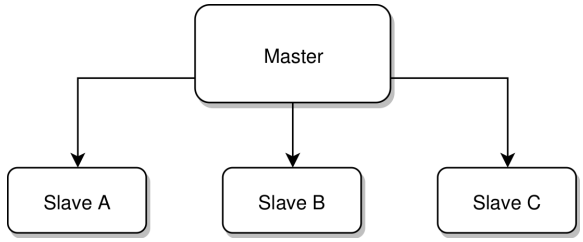


Figure 2.4: Master-slave Pattern. *Courtesy: [50, Distributed Systems]*

Peer-to-Peer Pattern

In the peer-to-peer architectural pattern - as seen by Figure 2.5 - the peer can act as a client asking for services to be provided as well as a server providing services to the client [50]. Thus, it acts as both and changes its role in a dynamic manner.

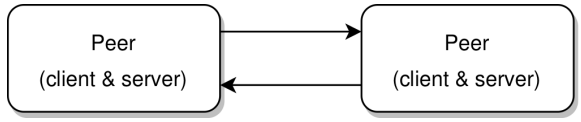


Figure 2.5: Peer-to-peer Pattern. *Courtesy: [50, Distributed Systems]*

2.2.3 Agent Architectures

What exactly is an agent? To date, there is no widely accepted definition of what an agent is. In this study, an agent is referred to as *an autonomous software entity that is situated in some environment where it can monitor and response to changes proactively or reactively by itself or through communication with other agents to persistently achieve certain goal/task on behalf of user or other agents* , [98, Wooldridge, 2009].

Agent architectures (also referred to as robot paradigm) in computer science is a blueprint for software agents and intelligent control systems, depicting the arrangement of components. There are different kinds of agent architectures, each with it's pros and cons. Here are some of the most common ones:

Reactive Architectures

Reactive architectures - as seen in Figure 2.6 - are based on the idea that intelligent rational behavior is innately linked to the environment and the idea that intelligent behavior emerges from the interaction of various simpler instances of sensing-acting couplings [63]. These couplings are concurrent processes, called behaviors, which take the local sensing data and compute the best action to take independently of what the other processes are doing.

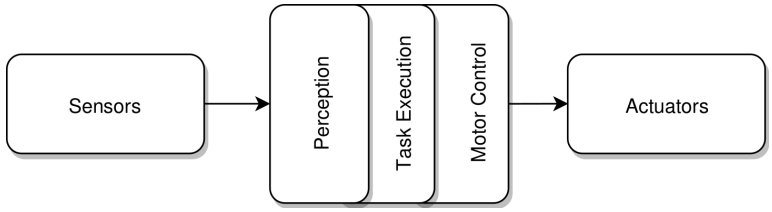


Figure 2.6: Reactive agent architecture. *Courtesy: [54, Agent Architectures]*

In rapidly changing environments, such as the real world, where the controller must react quickly to external changes, there may not be time available to perform many time-consuming actions such as planning and introspection. In this case an agent with a reactive, or behavioral architecture may be appropriate. Disadvantages of reactive architectures is that they have no long-term planning capabilities and have therefore limited applicability.

Deliberative (Hierarchical) Architectures

The core of this *deliberative architecture* is a planner - as seen by Figure 2.7 - which elaborates plans based on the knowledge of the problem domain [63]. Compared to reactive agent architectures, which are able to reach their goal only by reacting reflexively on external stimuli, a *deliberative* agent's internal processes are more complex. The difference lies in fact, that deliberative agent maintains a symbolic representation of the world it inhabits. In other words, it possesses internal image of the external environment and is thus capable to plan its actions. A plan defines a series of actions designed to accomplish a set of goals but not violate any resource limitations, temporal or state constraints, or other AUV operation rules. But any plan, no matter how it is generated, requires the help of an execution system to be useful for real-world execution.

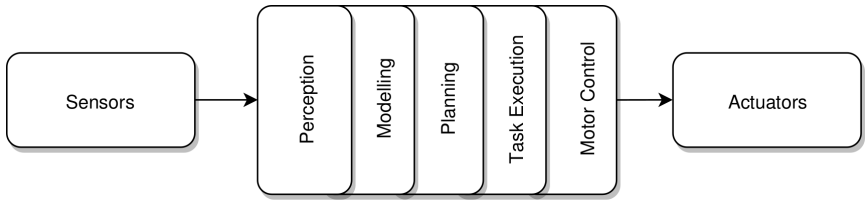


Figure 2.7: Deliberate agent architecture. *Courtesy: [54, Agent Architectures]*

A deliberative agent architecture is a sensible choice for applications where a robot needs to continually deliberate while acting, monitor the state of the world, monitor the action's execution, and abort or skips actions whenever needed.

Hybrid (Layered) Architectures

Many researchers have argued that neither a completely deliberative nor reactive approach is suitable for building agents. They have suggested using *hybrid* architectures, which attempt to marry classical and alternative approaches. In essence, robots needed to combine the planning capabilities of the classical architectures with the reactivity of the behavior-based architectures, attempting a compromise between bottom–up and top–down methodologies. This evolution was named layered architectures or hybrid architectures. As a result, many of today's architectures for robotics follow a hybrid pattern.

Usually, a hybrid agent architecture is structured in three layers [66]: the *reactive* layer, the *executive* layer, and the *deliberative* layer, see Figure 2.8. The reactive layer takes care of the real-time issues related to the interactions with the environment. It is composed of the robot primitives where the necessary sensors and/or actuators are directly connected. The executive layer acts as an interface between the numerical reactive layer and the symbolical deliberative layer interpreting high-level plan primitives' activation. The executive layer also monitors the primitives being executed and handles the events that these primitives may generate. The deliberative layer transforms the mission into a set of tasks which define a plan. It determines the long-range tasks of the robot based on high-level goals.

In the case of layered architectures we usually distinguish between a *horizontal* and *vertical* layering [54] as seen from Figure 2.8. In horizontal layering each layer is directly connected the sensory input and the action output. In effect, each layer acts like an agent itself, producing suggestions as to what action to perform. In vertical layering you will have interaction between layers at different levels of abstraction.

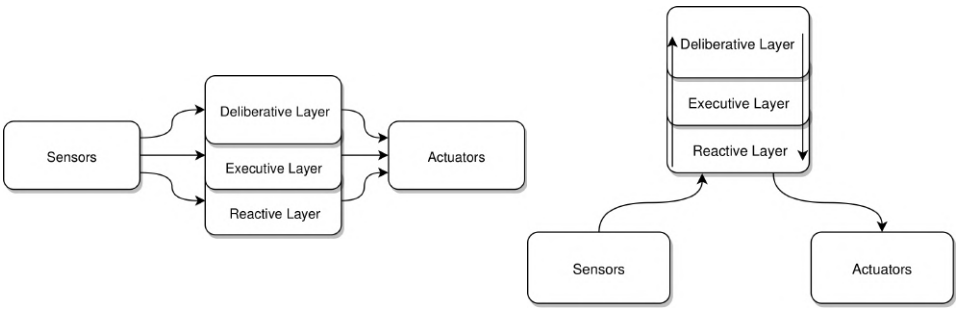


Figure 2.8: Horizontal and vertically layered architectures. *Courtesy: [54, Agent Architectures]*

A key problem in such architectures is what kind of control framework to embed the agent's subsystems in, to manage the interactions and the level of precedence between various layers. Other critiques to using hybrid architectures is that they lack general design guiding methodologies and are unsupported by formal theories. They are also very specific and application dependent [54]. It is however important to mention that most of the space autonomous systems have taken the 'three-layer' architecture [36].

2.3 Related Work in Software Architectures for AUVs

There have been many attempts to create scalable and robust software architectures for AUVs, some more successful than others. In the following subsections you will get to know some of the current state-of-the-art architectures still maintained and operational.

MOOS-IvP

MOOS-IvP (Mission Oriented Operating Suite Interval Programming) is a project situated at Massachusetts Institute of Technology (MIT) and aims to provide autonomy on robotic platforms, in particular autonomous marine vehicles [60]. MOOS is a set of open source C++ modules with a ranging functionality over low-level, multi-platform communications, dynamic control, high precision navigation and path planning, concurrent mission task arbitration and execution, mission logging and playback.

MOOS has a *centralized* topology as seen from Figure 2.9. Each application within a MOOS community (aMOOSApp) has a connection to a single "MOOS Database" (called MOOSDB) that lies at the heart of the software suite. All communication happens via a *client-server* pattern with no *peer-to-peer* communication [60]. This centralized system topology is obviously vulnerable to "bottle-necking" at the server regardless of how well written the server is. However the advantages of such a design are greater than its disadvantages. Firstly the network remains simple regardless of the number of participating clients. The server has complete knowledge of all active connections and can take responsibility for the allocation of communication resources. The clients operate independently with inter-connections. This prevents rogue clients (badly written or hung) from directly interfering with other clients

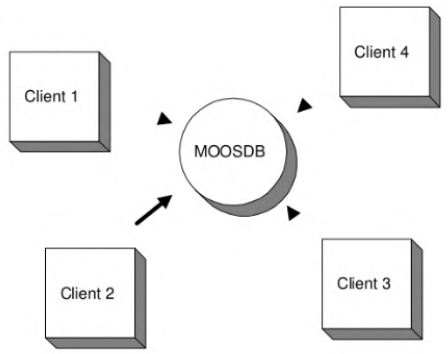


Figure 2.9: MOOS binds applications into a network with a star-shaped topology. Each client has a single communications channel to a server (MOOSDB). *Courtesy: [60, P. Newman]*

MOOS uses a TCP/IP as its communications backbone. This limits the use of MOOS to processors and operating systems that can support a TCP/IP stack [17] (e.g. Linux, Windows NT and Windows 2000).

DSAAV

DSAAV (Distributed Software Architecture for Autonomous Vehicles) is a project at the Acoustic Research Laboratory (ARL) of the National University of Singapore (NUS) [17]. In contrast to MOOS, it adopts a peer-to-peer communication architecture. By being distributed, DSAAV spreads the load and traffic across all processors in the AUV and avoids high load and dependency on single processors and databases. However a *centralized* system is easier to administer and monitor; for this reason, DSAAV provides a central configuration database and logging service. As the configuration database is required primarily during start-up and the logging service is not essential to the operation of the AUV, the advantages of a *distributed* architecture are not lost by having these centrally managed services.

DSAAV is a four-layer architecture as depicted in Figure 2.10. The bottom-most layer, *IComms*, provides an implementation of a unreliable messaging service over the communications backbone available. The next higher layer is the *RPC* layer which implements a remote procedure call (RPC) semantic using the IComms messaging service. The third layer consists of framework and sensor/actuator services implemented using the RPC framework. The top layer houses the *vehicle command & control* components which utilize the services provided by lower layers to achieve the mission of the vehicle.

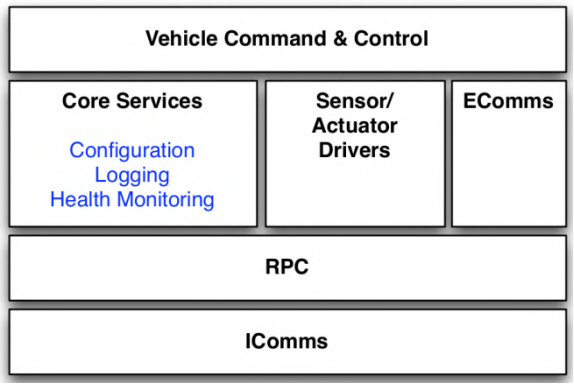


Figure 2.10: DSAAV's four layer software architecture *Courtesy: [17, M. Chitre]*

Unlike MOOS, DSAAV can operate on a variety of communication backbones including raw Ethernet; this makes it extremely lightweight and suitable for implementation on micro-controllers without a TCP/IP stack or multi-threading support as well as single-board computers running Linux or Windows [17].

T-Rex

The T-REX T-REX (Teleo-Reactive EXecutive) is another AUV software architecture that aims for enabling onboard automated planning to generate robust mission plans using system state and desired goals. It is developed by the Monterey Bay Aquarium Research Institute (Moss Landing, CA) and is a part of the remote agent architecture [48] developed by the National Aeronautics and Space Administration (NASA) Deep Space 1 spacecraft.

The system encapsulates the notion of a *deliberative* agent architecture. An agent is viewed as the coordinator of a set of concurrent control loops. Each control loop is embodied in a Teleo-Reactor (or reactors) that encapsulates all details of how to accomplish its control objectives. The mapping between reactors and timelines is the basis for sharing information. If a reactor owns a timeline it is declared *internal* to that reactor. If a reactor uses a timeline to observe values and/or express requirements it is declared *external* to that reactor. Figure 2.11 illustrates the flow of information in a system containing three reactors: The *Mission Manager* keeps track of science goals to give directives to the Navigator using the path external timeline. The *Navigator* manages the navigation of the AUV with one internal timeline and three external timelines. The navigation route is used to select the appropriate commands to send to the *Executive* as an internal timeline while position and attitude timelines capture AUV navigation data. Although T- REX's design leads to factoring of computation into layers, in practice a layered structure is not inherent [48].

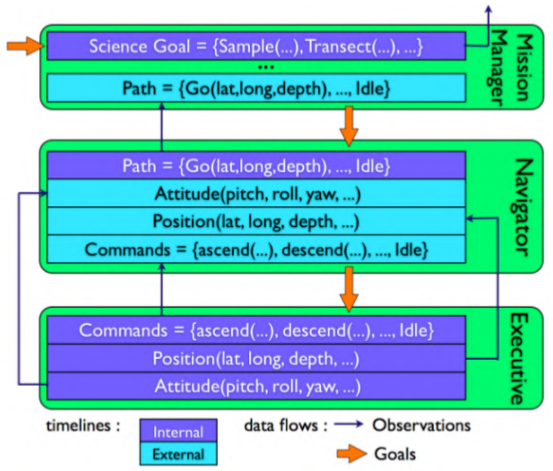


Figure 2.11: TREX's three layer software architecture. *Courtesy: [48, C. McGann]*

While T-REX was built for a specific underwater robotics application, the principles behind its design are applicable in any domain where deliberation and execution are intertwined.

COLA2

COLA2 (Component Oriented Layer-based Architecture) is a ROS-based software architecture running in the underwater vehicles.

Following the layer-based model, the components in the control architecture are distributed among three hierarchical layers, as seen by Figure 2.12. The *mission* layer obtains a mission plan by means of an onboard automatic planning algorithm (PN) or compiling a high-level mission description given by a human operator. The mission plan is described using a PN that is interpreted by the *execution* layer. This plan is executed by enabling/disabling available vehicle primitives contained in the *reactive* layer.

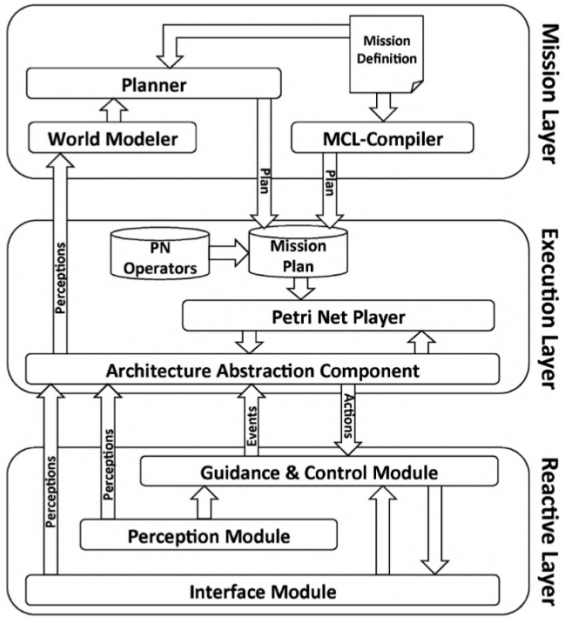


Figure 2.12: COLA2's three layer software architecture. *Courtesy: [66, C. McGann]*

2.4 Chapter Summary

The aim of this chapter was to give insight into important aspects of the software design process. This means pointing out design such as simple, reusable and modular code to help creating good and reliable software. When creating a software architecture it is important to determine how the 'skeleton' of your software, inter-dependencies between nodes and means of communication. Lastly, it is always a good practice to look up other software architectures for inspiration for your own architecture.

In the next chapter you will learn more about the embedded systems software, actuators, sensors and electronics of the Manta AUV. Bring especially attention to the concepts of distributed software, client/server communication pattern and deliberative hierarchical agent architectures. These concepts will be important when laying out the structure and information flow of the Manta v1 software architecture.

CHAPTER 3

THE MANTA V1 ELECTRONICS AND SYSTEM SOFTWARE

The complete *firmware* (permanent software programmed into a read-only memory) will be a collection of self-developed and open-source software. The ultimate success of an embedded system project depends both on its software and hardware. Therefore, both hardware and software skills are essential for developing embedded systems. Good software combined with average hardware will always outperform average software on good hardware. This chapter will show how the Manta v1 system software components and communication channels have been sketched out, compatible middleware and software frameworks are also layed out. The embedded electronics and hardware of Manta AUV will also be discussed in brief detail.

3.1 System Software

System software is a type of computer program that is designed to run a computers hardware and application programs. If thinking of the computer system as a layered model, the system software is the interface between the hardware and user applications. The *operating system* (OS) is the best-known example of system software. The OS manages all the other programs in a computer.

Operating System

The *operating system* is a type of system software kernel that sits between computer hardware and end user. It is installed first on a computer to allow devices and applications to be identified and therefore functional. System software is the first layer of software to

be loaded into memory every time a computer is powered up. Commonly used operating systems for software engineering are *Linux Ubuntu* and *Mac OS X*.

Embedded Linux

An *embedded system* is a controller programmed and controlled by a *real-time operating system* (RTOS) with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints. It is embedded as part of a complete device often including hardware and mechanical parts.

Embedded Linux is a type of Linux operating system that is designed to be installed and used within embedded devices and appliances. It is a compact version of Linux that offers features and services in line with the operating and application requirement of the embedded system. Embedded Linux, though utilizing the same Linux kernel, is quite different from the standard Linux OS. Embedded Linux is specifically customized for embedded systems [93]. Therefore it is has a much smaller size, requires less processing power and has minimal features. Based on the requirements of the underlying embedded system, the Linux kernel is modified and optimized as an embedded Linux version. Such an instance of Linux can only run device-specific purpose-built applications.

Device Drivers

Device drivers is a type of system software which brings computer devices and peripherals to life. Drivers make it possible for all connected components and external add-ons perform their intended tasks and as directed by the OS. Without drivers, the OS would not assign any duties. Device drivers are operating system-specific and hardware-dependent. In the case of the Manta v1 architecture there have been developed in-house device drivers for peripherals such as the *doppler velocity log* (DVL), *inertial measurement unit* (IMU), cameras and *electronic speed controllers* (ESC) to interface with the rest of the software.

3.2 Actuators & Sensors on the Manta AUV

The crux of many guidance, navigation, and control problems is the ability to estimate the state of the system (e.g., position and velocity) from a time history of sensor measurements (e.g., from a localization solution such as GPS). The actuator and sensor stack is an important property of the AUV. The choice of sensors and their placement at the vehicle body will determine the system observability, meaning whether or not the relevant system states for the control problem can be identified and to what degree each state can be observed. In a growing number of applications, however, the control task must be accomplished with limited sensing capabilities. Control system performance is highly conditioned on the quality of sensor information available, therefore the choice of sensors and there position as well their precision is important. Actuators are equally important for the vehicle, the

number of thrusters and their position ultimately determine what degrees of freedom one can control and not.

The Manta AUV is outfitted with a number of sensors and actuators to accommodate the challenges of underwater situational awareness and maneuvering. The most important ones can be seen in Figure 3.1.

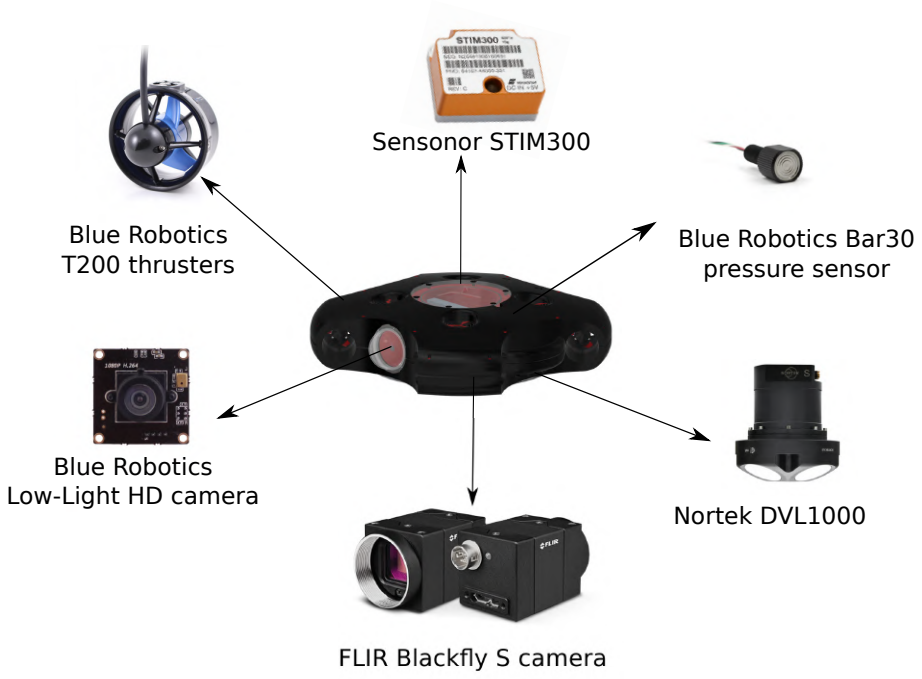


Figure 3.1: The Manta AUV hardware stack.

Thrusters

There are eight thrusters mounted on the platform. The thrusters are of the type T200 BlueROV2 from Blue Robotics. The core motor design is a three-phase brushless outrunner motor, similar to what you'd see on drones and RC airplanes, but optimized for underwater use [75]. Each thruster are optimized to run at a voltage of 16v (such as a 4s lithium-ion battery pack), but can run at a range of voltages.

Inertial Measurement Unit (IMU)

The IMU implemented is a STIM300 from Sensonor. This is a high performance non-GPS aided IMU. It contains 3 highly accurate microelectromechanical system (MEMS) gyros,

3 high stability accelerometers and 3 inclinometers [82]. Each axis is factory calibrated for bias, sensitivity and compensated for temperature effects to provide high-accuracy measurements.

Doppler Velocity Log (DVL)

To correct for integration errors caused by senor bias, misalignment and temperature variation, the Manta AUV is equipped with a Nortek DVL1000. The DVL is an acoustic sensor that estimates velocity relative to sea bottom. This is achieved by sending a long pulse along with the minimum of three acoustic beams (four in our case) in different directions as seen from Figure 3.2. This particular version has a maximum operational depth of 300 m and can perform bottom tracking from $0.2 - 75$ m range [61]. The DVL1000 is ideally suited for subsea navigation where size and weight are a concern.

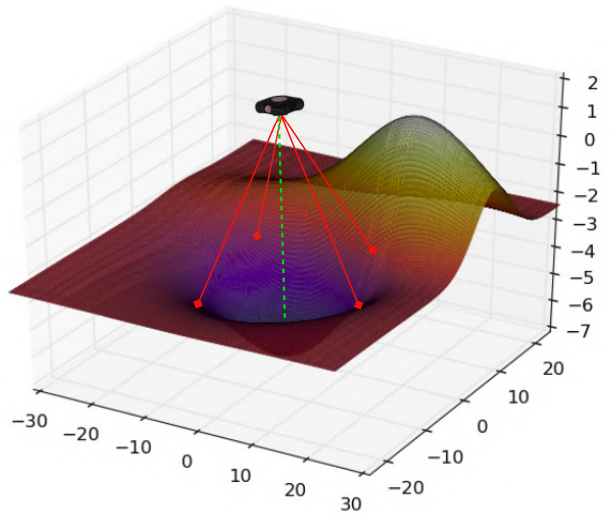


Figure 3.2: Illustration of the Manta AUV operating with a DVL.

Pressure Sensor

In addition to a pressure sensor within the Nortek DVL1000, Manta AUV is also equipped with a standard Blue Robotics Bar30 pressure sensor. The pressure sensor measures the static pressure at various depths in which we can use to calculate the absolute water depth of the vehicle.

Cameras

The Manta AUV is equipped with a forward facing low-light Blue Robotics HD camera. This Low-Light HD USB Camera is ideally suited to use underwater with excellent low-light performance, good color handling, and onboard video compression [74]. A specially-chosen wide-angle, low distortion lens provides excellent picture quality on the AUV.

In addition the Manta AUV is also equipped with a downward facing FLIR Blackfly S camera. These cameras have a variety of features, making them ideal for integration into machine vision systems including precise control over exposure, gain, white balance, and color correction [88]. The downward facing camera is mainly used for Simultaneous Localization and Mapping (SLAM) research for the Manta AUV.

3.3 Embedded Electronics System for the Manta AUV

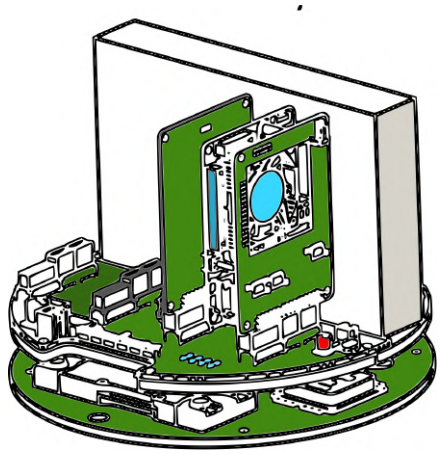


Figure 3.3: Drawing of Manta AUV electronics motherboard

Figure 3.3 gives an overview of the software employed on the Manta AUV. The core of the software system is implemented on the bottom side on an *Odroid XU4* (shown in green and blue standing up), which acts as Manta's On-board Computer (OBC). The OBC runs the control system and interfaces to a number of peripheral modules each with a specific responsibility. Each module is implemented on an designated computer. All of Mantas peripherals are standalone modules that have a dedicated NanoPi Neo plus 2 (a small sized Single Board Computer, SBC) for communications with the main computer. Having a dedicated computer for each peripheral module (cameras, manipulators, etc) is essential for making Manta modular. It enables us to do changes in the peripheral modules without having to change, or even open the main electronics enclosure.

The microcontroller unit (MCU) module utilizes an EFM32 Giant Gecko chip. The MCUs main function is communication with the two *electronic speed controllers* (ESC) that controls the eight thrusters. All control software on Manta is written in-house in C++ and Python using the Robot Operating System (ROS) framework. All of the computers peripheral computers communicates with the ROS framework over Ethernet. Other languages and tools that are used are *Gstreamer* for camera feed, bash scripting, and other OS dependent tools on the SBC's and a bare-metal C program on the MCU. Figure 3.4 shows the schematics of the embedded electronics on the Manta AUV.

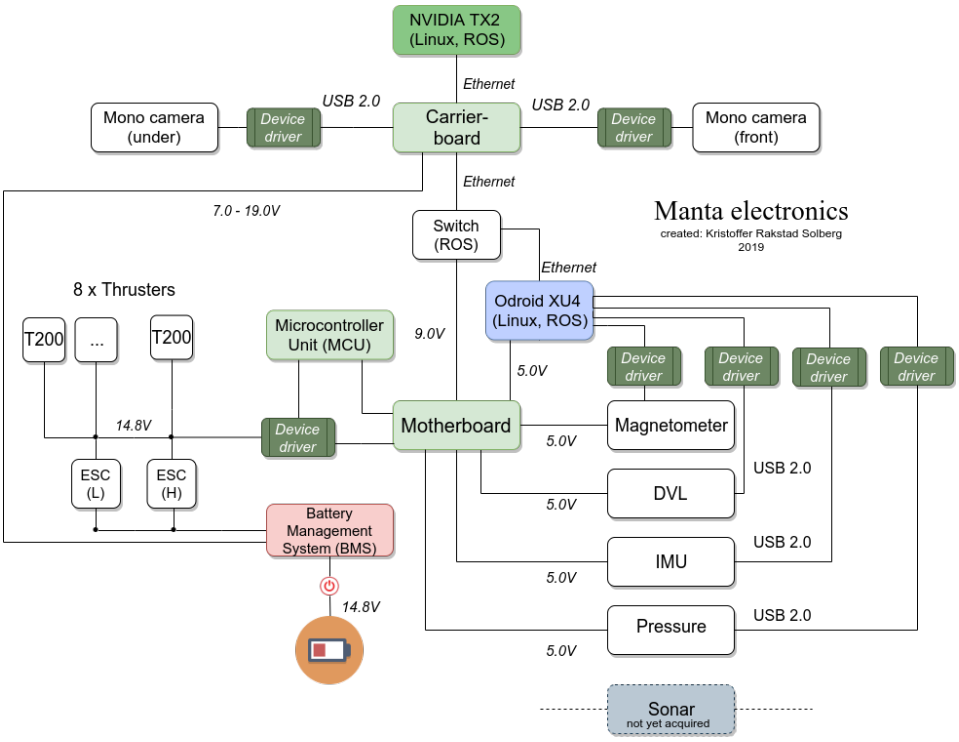


Figure 3.4: Manta AUV electronic system schematic

3.4 Manta v1 Software Architecture

System Requirements for the AUV

The organization of a robotic system for AUVs determines its capacities to achieve tasks and to react to events. The control structure of an autonomous robot must have both deliberative and reactive decision-making. Situations must be anticipated and the adequate actions decided by the robot accordingly. Tasks must be instantiated and refined at execution time according to the actual context. The robot must react in a timely fashion to

events. To meet these requirements, a general control structure for an AUV should have the following properties [2]:

1. **Programmability:** The AUV should be programmable across different environments and tasks. It should be possible to achieve multiple tasks described at an abstract level. Its functions should be easily combined according to the task to be executed.
2. **Autonomy and adaptability:** The AUV should be able to carry out its actions and to refine or modify the task and its own behavior according to the current goal and execution context as perceived.
3. **Reactivity:** The AUV has to take into account events with time bounds compatible with the correct and efficient achievement of its goals (including its own safety).
4. **Consistent behavior:** The reactions of the AUV to events must be guided by the objectives of its task.
5. **Robustness:** The AUV should be able to exploit the redundancy of the processing functions. Robustness will require the mission control to be decentralized to some extent.
6. **Extensibility:** Integration of new functions and definition of new tasks should be easy. Learning capabilities are important to consider here: the AUV should be able to gain more knowledge about tasks and environments over time for better decision-making.

3.4.1 Proposed Software Architecture

Many of the current state-of-the art software architectures displayed in Chapter 2 have a hybrid (layered) agent architectures, have a centralized or distributed network topology, and client/server communication pattern.

The proposed Manta v1 software architecture on the other hand, has a deliberative (hierarchical) agent architecture, a distributed network topology, and a client/server network communication pattern between objects or components. A component can be a complex piece of software such as a path planner, or something less complex such a battery monitoring. Objects or components can make requests to, and receive replies from, other objects or components located locally in the same process, in many different processes, or on different processors on the same or separate machines. At any time during the execution of a mission, a number of interacting concurrent processes at various levels of abstraction; ranging from high-level services such as path planners to low-level services such as execution of control laws, are being executed at various operational rates and latency.

Many of the functionalities which are part of the architecture can be viewed as clients or servers where the communication infrastructure is provided by robot operating system (ROS). Figure 3.5 depicts a complete schematic of the software components used in the architecture.

The distributed network topology of Manta v1 should provide robustness by not allowing single point of failures. The distributed computing will enable high reactivity by allowing decisions to be made locally in a node without having to consolidate with a decentralized control unit. The modular architectural design allows for adaptability, programmability and extensibility of the code, while consistent behavior is assured by the mission control.

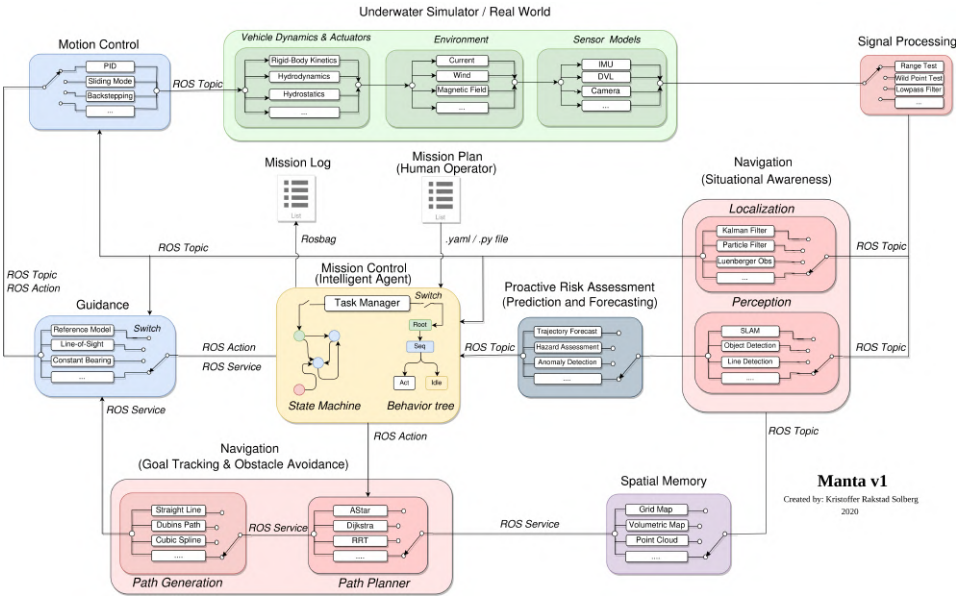


Figure 3.5: The Manta V1 Software Architecture

3.4.2 Manta v1 Information Flow

The flow of information in the Manta v1 software architecture travel clock-wise as denoted by the arrows in Figure 3.5. Starting of with the mission control. The mission plan is communicated by a human operator and provides aprori details of a mission such a maps, waypoints, objects of interest and emergency behaviors. On mission boot the mission plan provided as either a *.py* or *.yaml file* is interpreted by a task manager. The task manager operates as an interaction point between the high-level commands of the human operator and the low-level controls of the computer. Based on the mission plan, the task manager will create deliberate goals and task schedules. These are then compiled in the task controller as either a finite state machine or a behavior tree (these will be thoroughly explained in Chapter 8) depending on operators choice. The task controller manages the real-time agent behavior and task execution.

Once the task controller is compiled, the mission control will arm the actuators and start querying its sensors. All sensory information are pre-processed in the device driver before entering the navigation modules. Sensory information are sent to the perception and navigation modules for processing and sensor fusion. Risk assessment of both internal

and external events are processed by a 'proactive risk assessment' module before entering the mission control. If any anomalies or hazards are picked up, this will trigger an alert in the task controller and counter measures will be initiated. Otherwise, the mission control module interprets the data, updates the state of the task controller and appoint an appropriate next action.

Navigation loop-closure is continuously running in the background generating a mesh of the current robot's localization and surroundings in real-time. The mesh is stored in spatial memory as a look-up table for the path planner (i.e 2D occupancy grid map). Depending on the mission state and system deliberation, the mission control module will either command a set of target waypoints directly through the guidance module or ask the path planner construct a set of waypoints. Whenever the mission control sends a path request, the path planner will call for an updated map from the the spatial memory module. The path planner provides a coarse path that are fed to a path generation module for refining.

The guidance module appoints temporal and spatial constraints (i.e velocity, acceleration, jerk, orientation) to the geometric path provided by the path generation module. The motion control module will receive directives from the guidance system, as well as information about the position, velocity etc from the navigation module. When the controller has calculated desired forces in the AUVs body frame, this information is sent through a thrust allocator and converted to an PWM for each thruster. The motion of the vehicle are constrained by the vehicle dynamics and environmental forces and disturbances modelled by the underwater simulator / real world.

As seen in Figure 3.5 each node has multiple *switches*. These are included demonstrate the configurability of the Manta v1 software architecture. At all times, you have only one switch running in each module i.e you cannot have a DP controller and path following control running simultaneously in the motion controller. However, the ROS middleware allows for simultaneous (asynchronous execution) across modules (or nodes as they are called in ROS). Online reconfiguration of the data stream is handled in the mission control module.

At the time of writing, Manta v1 uses a version of ROS named Kinetic Kame on the Linux Ubuntu 16.04 operating system.

3.4.3 Robot Operating System

Figure 3.6: Robot Operating System (ROS)

The robot operating system (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms [30]. ROS is open-sourced, and its main goal is to make the multiple components of a robotics system easy to develop and share so they can work on other robots with minimal changes. This basically allows for code reuse, and improves the quality of the code by having it tested by a large number of users and platforms.

ROS is sometimes called a *meta operating system* because it performs many functions of an operating system, but it requires a computer's operating system such as Linux. One of its main purposes is to provide communication between the user, the computer's operating system, and equipment external to the computer. This equipment can include sensors, cameras, as well as robots. ROS has chosen a multilingual approach, which means several programming languages are compatible. Many software tasks are easier to accomplish in "high-productivity" scripting such as Python or MATLAB. However, there are times when performance requirements dictate the use of faster languages, such as C++. At the time of writing, there is support for C++, Python, MATLAB, Java and more.

Before we start writing software, let us take a moment to introduce some of the key concepts that underlie the framework. When implementing algorithms in ROS, they are implemented as *Nodes*. The *ROS Master* provides naming and registration services to the rest of the nodes in the ROS system. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer. The Master is most commonly run using the *roscore command*, which loads the ROS Master along with other essential components.

As mentioned above the ROS system consist of a number of independent nodes that comprises and *graph*. These nodes by themselves are typically not very useful. Things only get interesting when the nodes communicate with each other, exchanging information and data. The most common way to do that is through *topics*, *services* and *actions* which will now be described.

ROS Topics

ROS *Topics* implement a *publish/subscribe* mechanism, one of the more common ways to exchange data in a distributed system. Before nodes start to transmit data over topics, they must first announce (or advertise) both the topic name and the types of messages that are going to be sent. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic. By *registering* a request to *roscore*/*Master*; after subscribing, all messages on the topic are delivered to the node that made the request. All this can be seen in Figure 3.7. There can be multiple publishers and subscribers to a topic.

Topics are well suited for continuous data streams like sensor data, camera, robot state and so on.

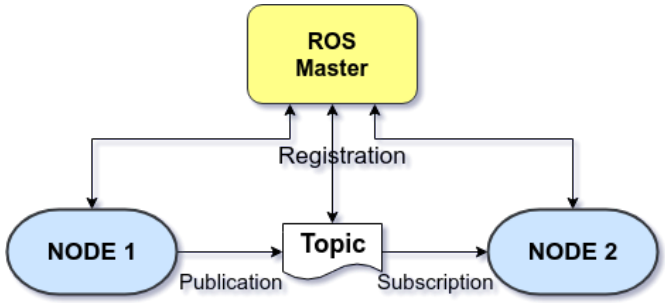


Figure 3.7: Illustration of ROS Topics

ROS Services

ROS *Services* are another way to pass data between nodes in ROS. Services are just **synchronous** meaning they execute tasks in *series*, like, ”go to position”, and **then after** you have completed that task you can start ”juggling” a ball. We define the inputs and outputs of this function similarly to the way we define new message types, and we also make a registration with the *roscore/Master*. The server (that provides the *service*) specifies a callback to deal with the service request, and advertises the service. The client (which calls the service) then accesses this service through a local proxy (third party matchmaker). This can be seen in Figure 3.8.

Service calls are well suited to things that you only need to do occasionally and that take a bounded amount of time to complete. There are discrete actions that the robot might do, such as turning on and off a light, changing control mode, or honk a horn.

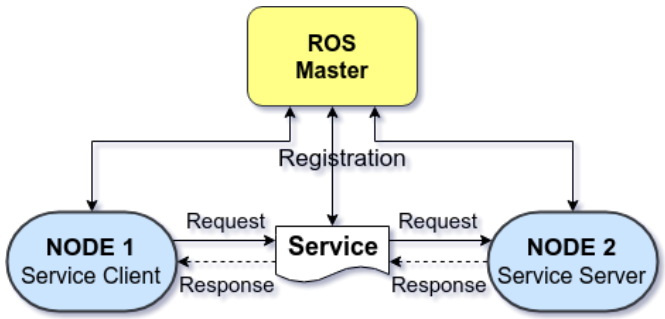


Figure 3.8: Illuestration of ROS Service

ROS Actions

While services are handy for simple get/set interactions like querying status and managing configuration, they don’t work well when you need to initiate long-running task. ROS *ac-*

tions are the best way to implement interfaces to time-extended, goal-oriented behaviors. While services are **synchronous**, actions are **asynchronous** meaning they can execute tasks in parallel, like, ”go to position” **while** ”juggling” this ball. Similar to the request and response of a service, an action uses a *goal* to initiate a behavior and sends a *result* when the behavior is complete. But the action further uses *feedback* to provide updates on the behavior’s progress toward the goal and also allows for goals to be canceled. Actions themselves are implemented using topics. An action is essentially higher-level protocol that specifies how a set of topics (goal, result, feedback, etc) should be used in combination. Figure 3.9 explains the architecture.

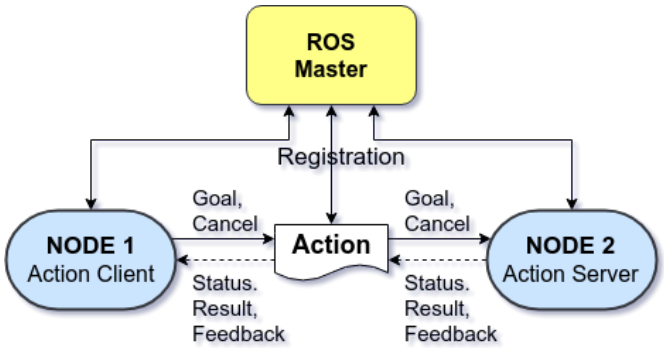


Figure 3.9: Illustration of ROS Action

3.5 Robot Simulation and Visualization

Robot simulation is an essential tool in every roboticist’s toolbox. A well-designed simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. We are going to use the *Gazebo Robot Simulator* simulator running on *Linux Ubuntu 16.04* to simulate the underwater environment and AUV.

3.5.1 Robot Creation with CAD Tools

Figure 3.10: AutoCAD and Blender logos. *Courtesy: Autodesk AutoCAD and Blender*

The first phase of robot manufacturing is its design and modeling. For this purpose it we'll be using AutoCAD and Blender (see Figure 3.10. AutoCAD is a commercial computer-aided design (CAD) and drafting software that will be both used for creating geometries and visualizations of the Manta AUV. The models created in AutoCAD will both be used for technical drawings for production purposes, and will also be exported to formats compatible with the Gazebo simulation environment. Blender is a computer graphics software for making 3D models and animations (among other things). This program will mainly be used to make the seabed heightsmaps and mesh files for the underwater simulation environment.

3.5.2 Robot Simulation with Gazebo Robot Simulator

Gazebo is an open source, multi robot simulator. We can simulate complex robots, robot sensors, and a variety of 3D objects. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. It provides a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces. Gazebo has a good interface in ROS, which exposes the whole control of Gazebo in ROS. A screen grab from the Gazebo Robot Simulator is shown in Figure 3.11.

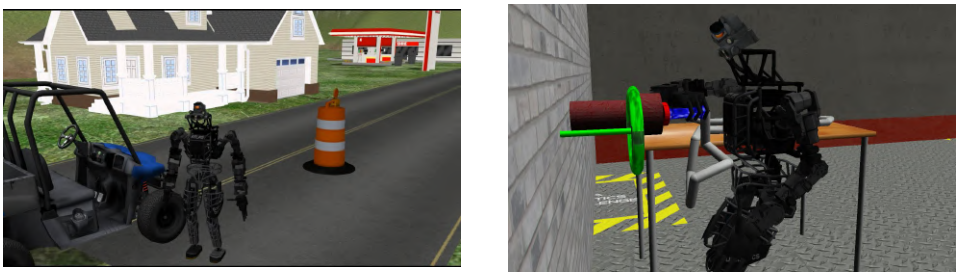


Figure 3.11: The Boston Dynamics Atlas robot performing intervention tasks in Gazebo Robot Simulator. *Courtesy: DARPA Virtual Robotics Challenge*.

3.6 Chapter Summary

The aim of this chapter was first of all to introduce the Manta AUV, its electronics, system software and the of course the Manta v1 software architecture. Manta v1 has a deliberative (hierarchical) agent architecture, a distributed network topology, and a client/server network communication pattern between objects or components. For hosting the software architecture on Linux Ubuntu, Manta v1 uses ROS middleware. For simulation of vehicle and software architecture a 3-D simulation testbed will be created using the Gazebo robot simulator. Before creating the simulator, one must derive a good mathematical model of the Manta AUV, which is the job of the next chapter.

CHAPTER 4

MODELING OF AUTONOMOUS UNDERWATER VEHICLES

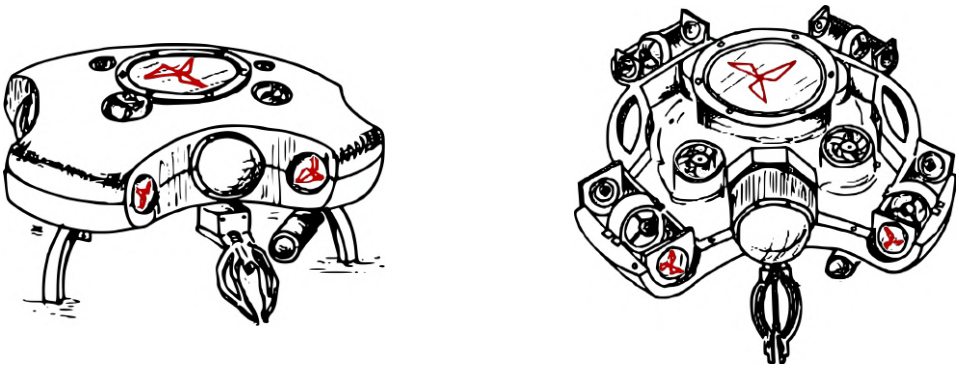


Figure 4.1: Sketching of Manta AUV with and without top cover

The goal of this chapter is to quantify the different rigid-body and hydrodynamic properties that dictates the behavior of a typical AUV - in our case the Manta AUV (see Figure 4.1). The modeling of marine vehicles involves the study of *statics* and *dynamics*. Statics is concerned with the equilibrium of bodies at rest or moving with constant velocity, whereas dynamics is concerned with bodies having accelerated motion. It is further common to divide the study of dynamics into two parts: *kinematics*, which treats only geometrical aspects of motion, and *kinetics*, which is the analysis of the forces causing the motion. The resulting marine craft equations of motion from this study is equal to the external forces such as thruster forces, current, waves and viscous forces with the rigid body mass (inertia) forces. It therefore follows that if the external forces can be correctly estimated,

then the corresponding motion of the AUV can be predicted accurately. Table 4.1 shows the SNAME-notation used for a marine vehicle.

DOF	Forces & Moments	Velocities	Positions & Euler Angles
Surge	X	u	x
Sway	Y	v	y
Heave	Z	w	z
Roll	K	p	ϕ
Pitch	M	q	θ
Yaw	N	r	ψ

Table 4.1: SNAME-notation for marine vehicles

4.1 Kinematics

A natural approach is to start with the kinematic equations. Kinematics is the science of describing motions themselves, leaving out the the details of how the motions was initiated in the first place [27].

4.1.1 Reference Frames

When Analyzing the motion of marine vehicles in 6 DOF it is convenient to define two coordinate frames as indicated in Figure 4.2: An Earth-fixed frame and a body-fixed frame.

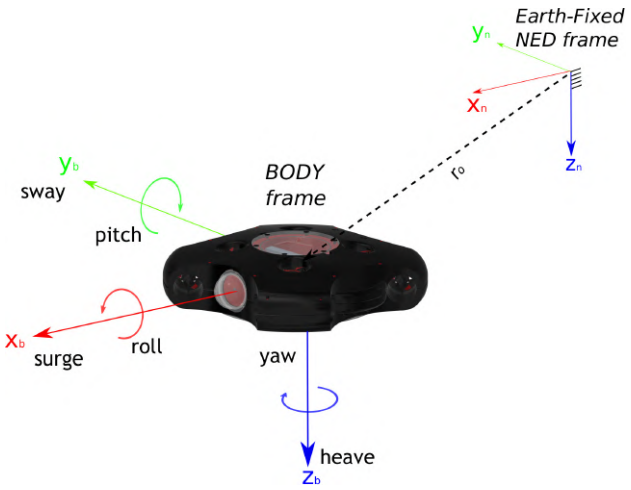


Figure 4.2: Body-fixed and earth-fixed reference frames

NED The *North-East-Down* frame (**NED**) is a reference frame that is usually defined as the tangent plane on the surface of the Earth moving with the craft. For slow moving ocean vehicles the motion of the earth has little effect on the vehicles itself. Hence the earth-fixed coordinate system can be considered to be *inertial*, with the *x-axis* pointing towards *North*, the *y-axis* pointing towards *East* and the *z-axis* pointing *Down* towards the Earth center.

BODY The *body-fixed* frame (**BODY**) is a moving reference frame that is fixed to the vehicle. The motion of the body-fixed frame is described relative to the inertial reference frame. Its axes coincide with the principal axes of inertia, where the *x-axis* is longitudinal (directed from aft to fore), the *y-axis* is transverse (directed to starboard), and *z-axis* is normal (directed from top to bottom).

Based on the SNAME notation mentioned in the beginning of this section, the general motion of a marine vehicle in 6 DOF can be described by the following set of vectors:

Parameter	Total	Linear	Angular
NED position	$\boldsymbol{\eta} = [\boldsymbol{p}^n_{b/n}, \boldsymbol{\Theta}_{nb}]^T$	$\boldsymbol{p}^n_{b/n} = [x, y, z]^T$	$\boldsymbol{\Theta}_{nb} = [\phi, \theta, \psi]^T$
BODY Velocity	$\boldsymbol{\nu} = [\boldsymbol{\nu}^b_{b/n}, \boldsymbol{\omega}^b_{b/n}]^T$	$\boldsymbol{\nu}^b_{b/n} = [u, v, w]^T$	$\boldsymbol{\omega}^b_{b/n} = [p, q, r]^T$
BODY force / moment	$\boldsymbol{\tau} = [\boldsymbol{f}^b_b, \boldsymbol{m}^b_b]^T$	$\boldsymbol{f}^b_b = [X, Y, Z]^T$	$\boldsymbol{m}^b_b = [K, M, N]^T$

Table 4.2: 6 DOF marine vehicle described on vector form

- $\boldsymbol{p}^n_{b/n}$ = position of the point o_b (body center) with respect to {n} expressed in {n}
- $\boldsymbol{\Theta}_{nb}$ = Euler angles between {n} and {b}
- $\boldsymbol{\nu}^b_{b/n}$ = linear velocity of the point o_b with respect to {n} expressed in {b}
- $\boldsymbol{\omega}^b_{b/n}$ = angular velocity of {b} with respect to {n} expressed in {b}
- $\boldsymbol{f}^b_b$ = force with line of action through the point o_b expressed in {n}
- $\boldsymbol{m}^b_b$ = moment about the point o_b expressed in {n}

4.1.2 Transformations Between BODY and NED

Vessel position and orientation are commonly expressed in earth-fixed NED, while its linear and angular velocities are expressed in BODY. These values are expressed using the SNAME notation, given in Table 4.2. The relationship between the two reference frame is described by a set of differential equations as seen in Equation 4.1.

Euler Angle Transformation

In [27, Euler Angle Transformation] it is shown that the transformation between the BODY and NED coordinate frames can be expressed in vector form by using a transformation matrix $\boldsymbol{J}_\Theta(\boldsymbol{\eta})$ giving us the kinematic equations of motion :

$$\dot{\boldsymbol{\eta}} = \boldsymbol{J}_{\Theta}(\boldsymbol{\eta})\boldsymbol{\nu} \qquad \Leftrightarrow \qquad \begin{bmatrix} \dot{\boldsymbol{p}}^{n}_{b/n} \\ \dot{\boldsymbol{\Theta}}_{nb} \end{bmatrix} = \begin{bmatrix} \boldsymbol{R}^{n}_{b}(\boldsymbol{\Theta}_{nb}) & \boldsymbol{0}_{3x3} \\ \boldsymbol{0}_{3x3} & \boldsymbol{T}_{\Theta}(\boldsymbol{\Theta}_{nb}) \end{bmatrix} \begin{bmatrix} \boldsymbol{\nu}^{b}_{b/n} \\ \boldsymbol{\omega}^{b}_{b/n} \end{bmatrix} \tag{4.1}$$

with the matrices for linear and angular velocity transformations:

$$\boldsymbol{R}^{n}_{b}(\boldsymbol{\Theta}_{nb}) = \begin{bmatrix} c\psi c\theta & -s\psi c\theta + c\psi s\theta s\phi & s\psi s\phi + c\psi c\phi s\theta \\ s\psi c\theta & c\psi c\phi + s\phi s\theta s\psi & -c\psi s\theta + s\theta s\psi c\phi \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix} \tag{4.2}$$

$$\boldsymbol{T}_{\Theta}(\boldsymbol{\Theta}_{nb}) = \begin{bmatrix} 1 & s\phi t\theta & c\phi t\theta \\ 0 & c\phi & -s\phi \\ 0 & s\phi/c\theta & c\phi/c\theta \end{bmatrix} \tag{4.3}$$

here you have that $s* = sin(*), c* = cos(*)$, and $t* = tan(*)$.

Unit Quaternions

An alternative to the Euler angle representation is a four-parameter method based on *unit quaternions* or *Euler parameters*. Unit quaternions is used in this project to represent the attitude of the vehicle when doing state estimation and control. Quaternions are a very useful mathematical tool for representing the rotation of a rigid body, they have great advantages over the more commonly used Euler angles' representation due to, e.g., lack of singularities/discontinuities and mathematical simplicity. Simultaneous rotation and translation (also called transformation) can be described using unit quaternions if the position is rotated to the inertial frame [31].

The quaternion $\boldsymbol{q} = [\eta, \boldsymbol{\varepsilon}^T]^T = [\eta, \varepsilon_1, \varepsilon_2, \varepsilon_3]^T$ can be interpreted as a complex number with η being the real part and ε the complex part. A visualization of the quaternion representation is shown in figure [4.3].

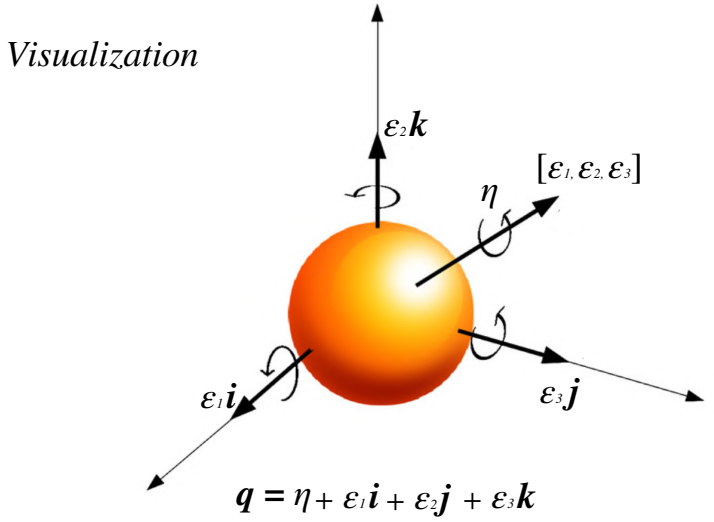


Figure 4.3: Axis–angle representation of a rigid body rotation.

In terms of unit quaternions the kinematic equations of motion are [27]:

$$\dot{\boldsymbol{\eta}} = \boldsymbol{J}_q(\boldsymbol{\eta})\boldsymbol{\nu} \qquad \Leftrightarrow \qquad \begin{bmatrix} \dot{\boldsymbol{p}}^n_{b/n} \\ \dot{\boldsymbol{q}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{R}^n_b(\boldsymbol{q}) & \boldsymbol{0}_{3x3} \\ \boldsymbol{0}_{4x3} & \boldsymbol{T}_q(\boldsymbol{q}) \end{bmatrix} \begin{bmatrix} \boldsymbol{\nu}^b_{b/n} \\ \boldsymbol{\omega}^b_{b/n} \end{bmatrix} \tag{4.4}$$

with the matrices for linear and angular velocity transformations:

$$\boldsymbol{R}^n_b(\boldsymbol{q}) = \begin{bmatrix} 1-2(\varepsilon_2^2+\varepsilon_3^2) & 2(\varepsilon_1\varepsilon_2-\varepsilon_3\eta) & 2(\varepsilon_1\varepsilon_3+\varepsilon_2\eta) \\ 2(\varepsilon_1\varepsilon_2+\varepsilon_3\eta) & 1-2(\varepsilon_1^2+\varepsilon_3^2) & 2(\varepsilon_2\varepsilon_3-\varepsilon_1\eta) \\ 2(\varepsilon_1\varepsilon_3-\varepsilon_2\eta) & 2(\varepsilon_2\varepsilon_3+\varepsilon_1\eta) & 1-2(\varepsilon_1^2+\varepsilon_2^2) \end{bmatrix} \tag{4.5}$$

$$\boldsymbol{T}_q(\boldsymbol{q}) = \frac{1}{2} \begin{bmatrix} -\varepsilon_1 & -\varepsilon_2 & -\varepsilon_3 \\ \eta & -\varepsilon_3 & \varepsilon_2 \\ \varepsilon_3 & \eta & -\varepsilon_1 \\ -\varepsilon_2 & \varepsilon_1 & \eta \end{bmatrix} \tag{4.6}$$

When integrating the $\dot{\boldsymbol{q}} = \boldsymbol{T}_q(\boldsymbol{q})\boldsymbol{\omega}^b_{b/n}$ term in Equation 4.4 a *normalization* procedure is necessary to ensure that the constraint:

$$\boldsymbol{q}^T\boldsymbol{q} = \eta^2 + \varepsilon_1^2 + \varepsilon_2^2 + \varepsilon_3^2 = 1 \tag{4.7}$$

is satisfied in the presence of measurement noise and numerical round-off errors. Several libraries in both C++ and Python have supporting functions for this purpose.

4.2 Kinetics

In order to derive the marine craft equations of motion, it is necessary to study the motion of rigid bodies, hydrostatics and hydrodynamics.

4.2.1 Rigid-Body Kinetics

The rigid-body equations of motion can be derived using the *Newton-Euler formulation*, *vectorial mechanics* and Lagrangian parametrization. From Newton's 2nd law, one knows that the force acting on a system is equal to the product of the mass and acceleration of the system ($\Sigma F = ma$). Euler's axioms also states that this is valid for linear momentum as well as angular momentum.

The rigid-body kinetics can be expressed in a vectorial setting according to [27]:

$$\boldsymbol{M}_{RB}\dot{\boldsymbol{\nu}} + \boldsymbol{C}_{RB}(\boldsymbol{\nu})\boldsymbol{\nu} = \boldsymbol{\tau}_{RB} \tag{4.8}$$

where $\boldsymbol{M}_{RB}$ is the rigid-body mass matrix and $\boldsymbol{C}_{RB}(\boldsymbol{\nu})$ is the rigid-body Coriolis and centripetal matrix.

Rigid-Body Inertia Matrix

The rigid-body mass matrix, $\boldsymbol{M}_{RB}$ is constant, symmetric and positive definite, which means that it satisfies the following criteria:

$$\boldsymbol{M}_{RB} = \boldsymbol{M}_{RB}^{T} > 0 \in \mathbb{R}^{6x6}, \qquad \dot{\boldsymbol{M}}_{RB} = \boldsymbol{0}_{6x6} \tag{4.9}$$

It's unique parametrization is on the form:

$$\begin{aligned} \boldsymbol{M}_{RB} &= \begin{bmatrix} m\boldsymbol{I}_{3x3} & -m\boldsymbol{S}(\boldsymbol{r}_g^b) \\ m\boldsymbol{S}(\boldsymbol{r}_g^b) & \boldsymbol{I}_b \end{bmatrix} \\ &= \begin{bmatrix} m & 0 & 0 & 0 & mz_G & -my_G \\ 0 & m & 0 & -mz_G & 0 & mx_G \\ 0 & 0 & m & my_G & -mx_G & 0 \\ 0 & -mz_G & my_G & I_x & -I_{xy} & -I_{xz} \\ mz_G & 0 & -mx_G & -I_{yx} & I_y & -I_{yz} \\ -my_G & mx_G & 0 & -I_{zx} & -I_{zy} & I_z \end{bmatrix} \end{aligned} \tag{4.10}$$

Here, m is the mass of the vehicle, $\boldsymbol{I}_{3x3}$ is the identity matrix, $\boldsymbol{I}_b$ is the inertia matrix, and $\boldsymbol{S}(\boldsymbol{r}_g^b)$ is the skew-symmetric martix where $\boldsymbol{r}_g^b$ is location of the centor of gravity (CG) with respect to center of origin (CO).

Rigid-Body Coriolis and Centripetal Matrix

The rigid-body Coriolis and centripetal term is due to the rotation of the body-fixed reference frame {b} with respect to the inertial reference frame {n}. The rigid-body Coriolis and centripetal matrix, $\boldsymbol{C}_{RB}(\boldsymbol{\nu})$, does not have a unique parametrization but can always be parametrized such that it is skew symmetrical under the assumption that the vehicle is moving in ideal fluid. According to [29, Fossen and Fjellstad 1995] the Lagrangian parametrization approach yields the following expression.

$$\begin{aligned}\boldsymbol{C}_{RB}(\nu) &= \begin{bmatrix} \mathbf{0}_{3x3} & -m\boldsymbol{S}(\boldsymbol{\nu_1}) - m\boldsymbol{S}(\boldsymbol{\nu}_2)\boldsymbol{S}(\boldsymbol{r}_g^b) \\ -m\boldsymbol{S}(\boldsymbol{\nu}_1) + m\boldsymbol{S}(\boldsymbol{r}_g^b)\boldsymbol{S}(\boldsymbol{\nu}_2) & -\boldsymbol{S}(\boldsymbol{I}_b\boldsymbol{\nu}_2) \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ -m(y_g q + z_g r) & m(y_g p + w) & m(z_g p - v) \\ m(x_g q - w) & -m(z_g r + x_g p) & m(z_g q + u) \\ m(x_g r + v) & m(y_g r - u) & -m(x_g p + y_g q) \end{bmatrix} \\ &\quad \begin{bmatrix} m(y_g q + z_g r) & -m(x_g q - w) & -m(x_g r + v) \\ -m(y_g p + w) & m(z_g r + x_g p) & -m(y_g r - u) \\ -m(z_g p - v) & -m(z_g q + u) & m(x_g p + y_g q) \\ 0 & -I_{yz}q - I_{xz}p + I_{zz}r & I_{yz}r + I_{xy}p - I_{yy}q \\ I_{yz}q + I_{xz}p - I_{zz}r & 0 & -I_{xz}r - I_{xy}q + I_{xx}p \\ -I_{yz}r - I_{xy}p + I_{yy}q & I_{xz}r + I_{xy}q - I_{xx}p & 0 \end{bmatrix}\end{aligned} \tag{4.11}$$

4.2.2 Hydrostatic Restoring Forces and Moments

In the hydrodynamic terminology, the gravitational and buoyant forces are called restoring forces. The gravitational force $\boldsymbol{f}_g^b$ will act through the center of gravity (CG) defined by vector $\boldsymbol{r}_g^b := [x_g, y_g, z_g]^T$ with respect to CO of the vehicle. Similarly, the bouyancy force $\boldsymbol{f}_b^b$ will act through the center of bouyancy (CB) defined by $\mathbf{r}_b^b := [x_b, y_b, z_b]^T$ with respect to CO of the vehicle. Hydrostatic restoring forces and moments follows from Archimedes principle where buoyancy force of a submerged body is equal to the volume of fluid displaced by the vehicle. These forces act in the vertical plane of n, hence the buoyancy force and gravitational force vectors can be written as:

$$\boldsymbol{f}_g^n = \begin{bmatrix} 0 \\ 0 \\ W \end{bmatrix}, \quad \boldsymbol{f}_b^n = - \begin{bmatrix} 0 \\ 0 \\ B \end{bmatrix} \tag{4.12}$$

where $W = mg$ and $B = pg\nabla$, with ∇ being volume displacement. Figure [4.4] shows the forces and moments acting on the vehicle.

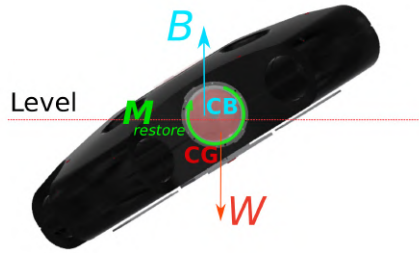


Figure 4.4: Hydrostatic restoring forces and moments acting on Manta

The restoring forces and moments can be transformed to the BODY-coordinate frame by using the matrix inverse of the Euler Angle (or unit quaternion) coordinate transformation matrix defined by Equation 4.2. The restoring vector then become [27]:

$$\begin{aligned} g(\eta) &= -\begin{bmatrix} \boldsymbol{R}_b^n(\boldsymbol{\Theta}_{nb})^{-1}(\boldsymbol{f}_g^n + \boldsymbol{f}_b^n) \\ \boldsymbol{r}_g^b \times \boldsymbol{R}_b^n(\boldsymbol{\Theta}_{nb})^{-1}\boldsymbol{f}_g^n + \boldsymbol{r}_b^b \times \boldsymbol{R}_b^n(\boldsymbol{\Theta}_{nb})^{-1}\boldsymbol{f}_b^n \end{bmatrix} \\ &= \begin{bmatrix} (W-B)sin(\theta) \\ -(W-B)cos(\theta)sin(\phi) \\ -(W-B)cos(\theta)cos(\phi) \\ -(y_gW - y_bB)cos(\theta)cos(\phi) + (z_gW - z_bB)cos(\theta)sin(\phi) \\ (z_gW - z_bB)sin(\theta) + (x_gW - x_bB)cos(\theta)cos(\phi) \\ -(x_gW - x_bB)cos(\theta)sin(\phi) - (y_gW - y_bB)sin(\theta) \end{bmatrix} \end{aligned} \tag{4.13}$$

4.2.3 Hydrodynamics

The equation of motion due to hydrodynamic forces is formulated as:

$$-\boldsymbol{M}_A\dot{\boldsymbol{\nu}}_r - \boldsymbol{C}_A(\boldsymbol{\nu}_r)\boldsymbol{\nu}_r - \boldsymbol{D}(\boldsymbol{\nu}_r)\boldsymbol{\nu}_r = \boldsymbol{\tau}_{hyd} \tag{4.14}$$

where $\boldsymbol{M}_A$ is the added mass matrix, $\mathbf{C}_A(\boldsymbol{\nu}_r)$ is the hydrodynamic Coriolis and centripetal matrix, $\mathbf{D}(\boldsymbol{\nu}_r)$ is the hydrodynamic damping matrix, and $\boldsymbol{\nu}_r$ is the relative velocity of the body due to currents. The relative velocity is expressed as:

$$\boldsymbol{\nu}_r = \boldsymbol{\nu} - \boldsymbol{\nu}_c \quad \textit{where} \quad \boldsymbol{\nu}_c = [u_c, v_c, w_c, 0, 0, 0]^T \tag{4.15}$$

Added Mass

Definition (added mass), [27, Fossen 2011]:

Hydrodynamic added mass can be seen as a virtual mass added to a system because an accelerating or decelerating body must move some volume of the surrounding fluid as it moves through it. Moreover, the object and fluid cannot occupy the same physical space simultaneously.

Due to the underwater vehicle operating outside of the wave zone, the added mass matrix $\boldsymbol{M}_A$ can be assumed approximately constant and thus independent of the wave circular frequency. With no incident waves, no currents and frequency independence the added mass matrix being positive definite has proved to be a reasonable assumption [26]. This in all means that the underwater vehicle (AUV in this case) satisfies the following criteria:

$$\boldsymbol{M}_A = \boldsymbol{M}_A^T > 0 \in \mathbb{R}^{6x6}, \qquad \dot{\boldsymbol{M}}_A = \boldsymbol{0}_{6x6} \tag{4.16}$$

$$\boldsymbol{M}_A = \begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \end{bmatrix} = - \begin{bmatrix} X_{\dot{u}} & X_{\dot{v}} & X_{\dot{w}} & X_{\dot{p}} & X_{\dot{q}} & X_{\dot{r}} \\ Y_{\dot{u}} & Y_{\dot{v}} & Y_{\dot{w}} & Y_{\dot{p}} & Y_{\dot{q}} & Y_{\dot{r}} \\ Z_{\dot{u}} & Z_{\dot{v}} & Z_{\dot{w}} & Z_{\dot{p}} & Z_{\dot{q}} & Z_{\dot{r}} \\ K_{\dot{u}} & K_{\dot{v}} & K_{\dot{w}} & K_{\dot{p}} & K_{\dot{q}} & K_{\dot{r}} \\ M_{\dot{u}} & M_{\dot{v}} & M_{\dot{w}} & M_{\dot{p}} & M_{\dot{q}} & M_{\dot{r}} \\ N_{\dot{u}} & N_{\dot{v}} & N_{\dot{w}} & N_{\dot{p}} & N_{\dot{q}} & N_{\dot{r}} \end{bmatrix} \tag{4.17}$$

Here, the SNAME notation Table 4.1 is used to determine the coefficients in the matrix. For instance $X_{\dot{w}}$ denotes the added mass force X along the x-axis due to fluid acceleration $\dot{w}$ in the z-direction.

The motion of a underwater vehicle moving in 6DOF at high speed will be highly non-linear and have coupled dynamics. However, the application of the Manta AUV does not involve any rapid motions and the vehicle will only be allowed to move at low speeds. The underwater vehicle in our case also has three planes of symmetry. Together, this suggests that we can neglect the contribution from the off-diagonal elements, and the added mass matrix $\boldsymbol{M}_A$ then become [27]:

$$\boldsymbol{M}_A = -diag[X_{\dot{u}}, Y_{\dot{v}}, Z_{\dot{w}}, K_{\dot{p}}, M_{\dot{q}}, N_{\dot{r}}] \tag{4.18}$$

Hydrodynamic Coriolis and Centripetal Matrix

The nonlinear Coriolis and centripetal matrix $\boldsymbol{C}_A(\boldsymbol{\nu})$ due to a rotation of $\{\mathrm{b}\}$ about the inertial frame $\{\mathrm{n}\}$ can be derived using an energy formulation based on the constant matrix $\boldsymbol{M}_A$.

$$\boldsymbol{C}_A(\boldsymbol{\nu}) = \begin{bmatrix} \mathbf{0}_{3x3} & -\boldsymbol{S}(\boldsymbol{A}_{11}\boldsymbol{\nu}_1 + \boldsymbol{A}_{12}\boldsymbol{\nu}_2) \\ -\boldsymbol{S}(\boldsymbol{A}_{11}\boldsymbol{\nu}_1 + \boldsymbol{A}_{12}\boldsymbol{\nu}_2) & -\boldsymbol{S}(\boldsymbol{A}_{21}\boldsymbol{\nu}_1 + \boldsymbol{A}_{22}\boldsymbol{\nu}_2) \end{bmatrix}$$
$$= \begin{bmatrix} 0 & 0 & 0 & 0 & -Z_{\dot{w}}w & Y_{\dot{v}}v \\ 0 & 0 & 0 & Z_{\dot{w}}w & 0 & -X_{\dot{u}}u \\ 0 & 0 & 0 & -Y_{\dot{v}}v & X_{\dot{u}}u & 0 \\ 0 & -Z_{\dot{w}}w & Y_{\dot{v}}v & 0 & -N_{\dot{r}}r & M_{\dot{q}}q \\ Z_{\dot{w}}w & 0 & -X_{\dot{u}}u & N_{\dot{r}}r & 0 & -K_{\dot{p}}p \\ -Y_{\dot{v}}v & X_{\dot{u}}u & 0 & -M_{\dot{q}}q & K_{\dot{p}}p & 0 \end{bmatrix} \tag{4.19}$$

4.2.4 Hydrodynamic Damping

Hydrodynamic damping on marine vehicles are generally caused by:

1. **Potential Damping**: Assuming potential flow theory it is possible to evaluate the forces acting on a body without the presence of friction. Potential damping is caused by oscillation of a body with the wave excitation frequency in the absence of incident waves. As the body moves in the water surface waves will be generated. With knowledge of the transport of wave energy away from the body it is possible to find the potential damping coefficients using energy relations [21]. The potential damping is a *linear frequency-dependent* phenomenon usually given as a 6x6 matrix.

$$\mathbf{D}_P(\omega) = \sum_{k=1}^{6} B_{jk_p}(\omega) \qquad (j = 1, ..., 6) \tag{4.20}$$

   where ω is the wave excitation frequency of a sinusoidal (regular) wave.

   By assuming that the AUV is far below the free surface the generation of surface waves will be less to none. This implies that the potential damping is negligible, hence $\boldsymbol{D}_P(\omega) = \sum_{k=1}^{6} B_{jk_p}(\omega) = 0$. When the AUV is close to the surface this approximation will not be valid, but the operational condition is usually not in this range.

2. **Skin Friction**: Skin friction is the damping force that arises when a boundary layer is created between the body and the water surrounding it. Due to the *no-slip condition*, the fluid that is close to the body will have zero velocity relative to the body. The no-slip condition arises due to the viscosity of the fluid and can be found on any stationary surface exerted to fluid flow [99]. This effect gradually decreases with the distance from the body surface. In Figure 4.5 you can see the Manta AUV portrayed with a boundary layer. The shape of Manta is geometrically simplified to a circular disk / oblate spheroid for convenience.

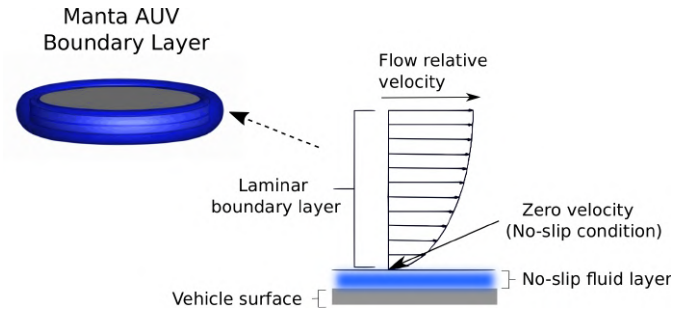


Figure 4.5: No-slip condition on Manta. The body of Manta is portrayed as a circular disk / oblate spheroid

Skin friction is a *linear frequency-dependent* phenomenon due to laminar boundary layer theory and is especially important when considering the low-frequency motion of marine craft [23, Faltinsen and Sortland, 1987]. In addition to linear skin friction, there will be a high-frequency contribution due to a turbulent boundary layer. This is usually referred to as a quadratic or nonlinear skin friction. The damping matrix due to skin friction can be written as:

$$\boldsymbol{D}_S(\omega) = \sum_{k=1}^{6} B_{jk_v}(\omega) \qquad (j = 1, ..., 6) \tag{4.21}$$

The off-diagonal terms of the skin friction will be assumed to be negligible compared to the diagonal terms. For a symmetric body such as Manta this is a reasonable approximation due to the diagonal terms usually being dominant for symmetric bodies [21]. The damping matrix due to skin friction can hence be rewritten:

$$\boldsymbol{D}_S(\omega) = diag[B_{11_v}, B_{22_v}, B_{33_v}, B_{44_v}, B_{55_v}, B_{66_v}] \tag{4.22}$$

3. **Wave Drift Damping**: Wave drift damping is a 2nd-order damping phenomenon. It is caused by the slow-drift motion of a surface vessel interacting with rapid oscillating behaviour of incident waves [22]. Wave drift damping is the most important damping contribution for surface vessels advancing in higher sea states. The wave drift damping can be presented as a 6x6 matrix:

$$\boldsymbol{D}_W = \sum_{k=1}^{6} B_{jk_w} \qquad (j = 1, ..., 6) \tag{4.23}$$

Usually wave drift damping is only important for large volume structures in high sea (upwards of wave period $T = 12s$, wave height $H = 8m$). Manta is designed to operate mostly in confined waters and below the wave zone, and we choose the neglect the damping contribution from incident waves. Matter of fact, wave drift

damping in surge, sway, yaw for a three plane symmetric body is small relative to the eddy-making damping (damping due to vortex shedding)[27].

4. **Damping Due to Vortex Shedding**:

   In a viscous fluid (not covered by potential theory), frictional forces are present such that the system is not conservative with respect to energy. This is commonly referred to as *inference drag*. It arises due to the shedding of discrete, swirling vortices at sharp edges where boundary layers separate from the vehicle surface as illustrated in Figure 4.6. The frequency at which the vortices are shed from a bluff section is proportional to the approach velocity U and iversely proportional to the section width D. The frequency in cycles per second (hertz) is [7]:

$$\mathrm{f} = \mathrm{SU/D} \quad [Hz] \tag{4.24}$$

   where S is the Strouhals number , a dimensionless proportionality constant generally between 0.2 - 0.3.

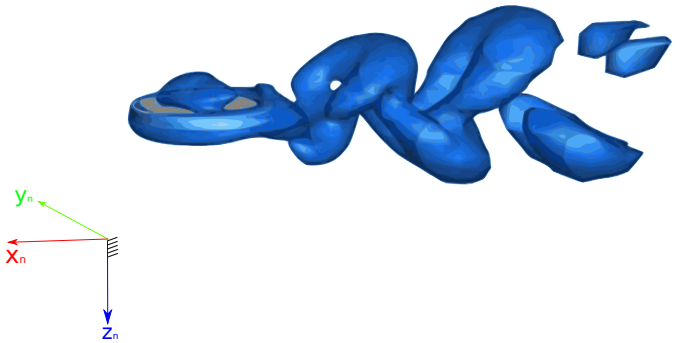


Figure 4.6: Vortex shedding on a Manta. The body of Manta is portrayed as a circular disk / oblate spheroid

   As the generation of vortices takes energy away from the vehicle motion it will create a viscous damping force in the opposite direction of motion. The viscous damping force can be modeled as:

$$f(u) = -\frac{1}{2}\rho C_D(R_n) A|u|u \tag{4.25}$$

   where u is the velocity of the craft, A is the projected cross-sectional area under water, $C_D(R_n)$ is the drag coefficient based on the representative area and ρ is the water density. This expression is recognized as one of the terms in *Morison's equation*. The drag coefficient $C_D(R_n)$ is a function of the *Reynolds Number*:

$$R_n = \frac{uD}{\nu} \tag{4.26}$$

The damping coefficients due to vortex shedding can be presented as a 6x6 matrix:

$$\mathbf{D}_M = \sum_{k=1}^{6} B_{jk_m} \qquad (j = 1, ..., 6) \tag{4.27}$$

5. **Damping Due to Viscous Pressure Drag**:

   The viscous pressure drag is a force caused by resultant pressure distribution over the surface of body. It can be thought of as the component of the pressure force parallel to the tangent to the travel path. For instance, consider a sphere (bluff body) in a moving fluid. Its drag consists of the friction between its surface and the moving fluid, and the difference in pressure along its surface. The force is the product of the pressure acting on a cross-sectional area of the body, normal to the travel path. The pressure distribution, and thus pressure drag, has several distinct causes: *Induced drag* (sometimes known as "drag due to lift" or "vortex drag") and *form drag* (sometimes known as boundary-layer pressure drag) [34].

   A simple interpretation of the basic drag is shown in Figure 4.7, which shows a sphere moving through a fluid forming a high-pressure region in front of it (upstream) and a low-pressure region behind it as the turbulent wake (downstream). The pressure differential across its cross-sectional area yields a drag force.

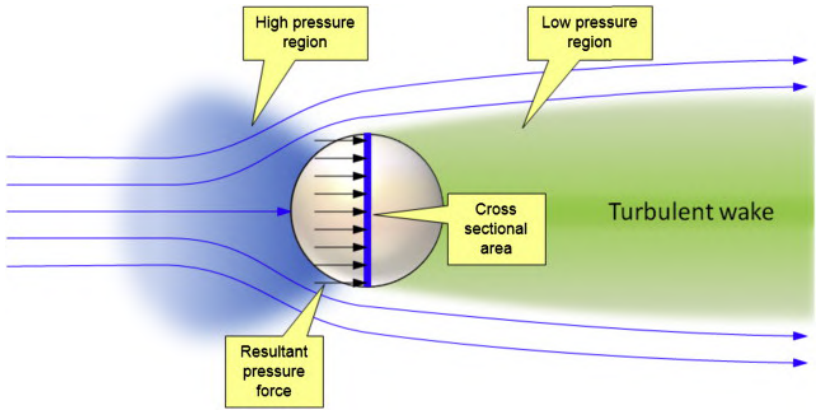


Figure 4.7: Viscous pressure drag across a rigid body. *Courtesy: General Aviation Aircraft Design, 2014*

The pressure drag contribute both to linear (laminar boundary layer) and nonlinear damping (turbulent boundary layer and drag due to lift). By assuming no coupled effects for slow dynamics, we collect the damping due to pressure drag into a diagonal damping matrix:

$$\boldsymbol{D}_{VP} = diag[B_{11_{VP}}, B_{22_{VP}}, B_{33_{VP}}, B_{44_{VP}}, B_{55_{VP}}, B_{66_{VP}}] \tag{4.28}$$

The Principle of Superposition

The different damping terms contribute to both linear and quadratic (nonlinear) damping. However, it is generally difficult to separate these effects. For convenience it is common to separate the total hydrodynamic damping into:

$$\begin{aligned}\boldsymbol{D}(\boldsymbol{\nu}_r) &= \boldsymbol{D}_P + \boldsymbol{D}_V(\boldsymbol{\nu}_r) \\ &= \underbrace{\mathbf{D}_P}_{\text{linear}} + \underbrace{\boldsymbol{D}_S(\boldsymbol{\nu}_r) + \boldsymbol{D}_W(\boldsymbol{\nu}_r) + \boldsymbol{D}_M(\boldsymbol{\nu}_r) + \boldsymbol{D}_{VP}(\boldsymbol{\nu}_r)}_{\text{linear + nonlinear}} \\ &= \boldsymbol{D}_L + \boldsymbol{D}_{NL}(\boldsymbol{\nu}_r)\end{aligned} \tag{4.29}$$

where $\boldsymbol{D}_L$ is the *linear damping matrix* due to potential damping and possible skin friction and $\boldsymbol{D}_{NL}(\boldsymbol{\nu}_r)$ is the *nonlinear damping matrix* due to quadratic damping and higher-order terms.

Linear Viscous Damping

The linear viscous damping are constituted by two matrices:

$$\begin{aligned}\mathbf{D}_L &= \mathbf{D}_P + \mathbf{D}_V \\ &= -\begin{bmatrix} X_u & 0 & 0 & 0 & 0 & 0 \\ 0 & Y_v & 0 & Y_p & 0 & Y_r \\ 0 & 0 & Z_w & 0 & Z_q & 0 \\ 0 & K_v & 0 & K_p & 0 & K_r \\ 0 & 0 & M_w & 0 & M_q & 0 \\ 0 & N_v & 0 & N_p & 0 & N_r \end{bmatrix}\end{aligned} \tag{4.30}$$

where the diagonal terms relate to seakeeping theory according to

$$\begin{aligned} -X_u &= B_{11v} \\ -Y_v &= B_{22v} \\ -Z_w &= B_{33v} + B_{33}(\omega_{heave}) \\ -K_p &= B_{44v} + B_{44}(\omega_{roll}) \\ -M_q &= B_{55v} + B_{55}(\omega_{pitch}) \\ -N_r &= B_{66v} \end{aligned} \tag{4.31}$$

Nonlinear Viscous Damping

Assuming no cross-coupling of terms, the non-linear / quadratic damping matrix will be:

$$\mathbf{D}_{NL} = \mathbf{D}_{S_{NL}}(\boldsymbol{\nu}_r) + \underbrace{\mathbf{D}_{W_{NL}(\boldsymbol{\nu}_r)} + \mathbf{D}_{M_{NL}}(\boldsymbol{\nu}_r)}_{=0} + \mathbf{D}_{VP_{NL}}(\boldsymbol{\nu}_r)$$

$$= -\begin{bmatrix} X_{|u|u}|u_r| & 0 & 0 & 0 & 0 & 0 \\ 0 & Y_{|v|v}|v_r| & 0 & 0 & 0 & Y_{|v|r}|v_r| \\ 0 & 0 & Z_{|w|w}|w_r| & 0 & 0 & 0 \\ 0 & K_v & 0 & K_{|p|p}|p| & 0 & K_r \\ 0 & 0 & M_w & 0 & M_{|q|q}|q| & 0 \\ 0 & N_{|v|v}|v_r| & 0 & 0 & 0 & N_{|v|r}|v_r| \end{bmatrix} \tag{4.32}$$

4.3 Equations of Motion

The vehicle kinematics and kinetics can all be constituted in a composed vectorized form. Taking into account the rigid-body inertia, hydrostatics, hydrodynamics and hydrodynamic damping we get the final equation of motion for 6DOF:

$$\dot{\boldsymbol{\eta}} = \boldsymbol{J}_{\Theta}(\boldsymbol{\eta})\boldsymbol{\nu} \tag{4.33}$$

$$\boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{g}(\boldsymbol{\eta}) = \underbrace{\boldsymbol{\tau}_{env}}_{\text{wind and wave forces}} + \underbrace{\boldsymbol{\tau}}_{\text{propulsion forces}} \tag{4.34}$$

$$\boldsymbol{M} \triangleq \boldsymbol{M}_{RB} + \boldsymbol{M}_A; \qquad \boldsymbol{C}(\boldsymbol{\nu}) \triangleq \boldsymbol{C}_{RB}(\boldsymbol{\nu}) + \boldsymbol{C}_A(\boldsymbol{\nu})$$

$$\boldsymbol{D}(\boldsymbol{\nu}) \triangleq \underbrace{\mathbf{D}_P}_{=0} + \boldsymbol{D}_S(\boldsymbol{\nu}_r) + \underbrace{\boldsymbol{D}_W(\boldsymbol{\nu}_r)}_{=0} + \boldsymbol{D}_M(\boldsymbol{\nu}_r) + \boldsymbol{D}_{VP}(\boldsymbol{\nu}_r)$$

4.4 *Case Study: Estimating Rigid-Body Kinetics, Hydrostatics and Hydrodynamics for the Manta AUV using Empirical and Analytical Estimates*

In this section we will perform a case study to investigate the forces and moments that acts on the body of Manta. Through knowledge in hydrodynamics, fluid mechanics and Newtonian mechanics we will try to determine some of the most significant rigid-body, hydrostatic and hydrodynamic coefficients for the vehicle. Knowledge into the properties of the vehicle will set a foundation for further development of simulation tools, model-based controllers and model-based observers.

4.4.1 Geometry Simplifications and Anticipated Results for Added Mass and Damping

An automated Matlab script for estimating the vehicle data presented in this case study is found in Appendix D.1.

To be able to make use of empirical equations the shape of Manta has to be simplified. It is believed that most accurate representation of Manta when it comes to hydrodynamic coefficients such as added mass and damping is somewhere in between the geometry of a oblate spheroid and a circular disk with the same diameter and thickness as Manta, as seen in Figure 4.8. If we make calculations for both shapes and calculate the relative difference we can determine the margin of error. By then performing some educated guesses on parameters and dominating effects, we should be able to narrow down our solution space.

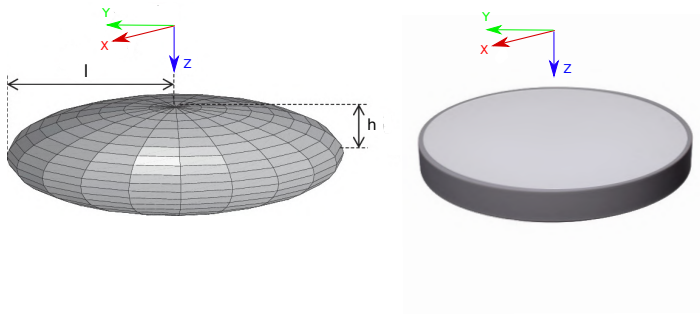


Figure 4.8: The body of Manta is portrayed as a circular disk / oblate spheroid

It can be seen in Figure 4.9 the body of Manta (portrayed as circular disk / oblate sphere) facing an undisturbed horizontal flow. The front facing the flow will experience the largest pressure force. The total pressure difference between the bow and aft (front and back) causes the pressure drag, which is often a dominant drag acting on bluff bodies like AUVs and ROVs for velocities inducing mostly turbulent wake [16]. However in our case we design the AUV for a maximum velocity of about $U = 0.7[m/s]$, which is considered a low velocity. For low velocities there will be less turbulence and vortex shedding which will make the linear damping the dominating term for the damping matrix. The linear part of the damping therefore consists of linear skin friction. The non-linear damping consists of all higher order terms such as turbulent skin friction, drag due to vortex shedding and viscous pressure drag.

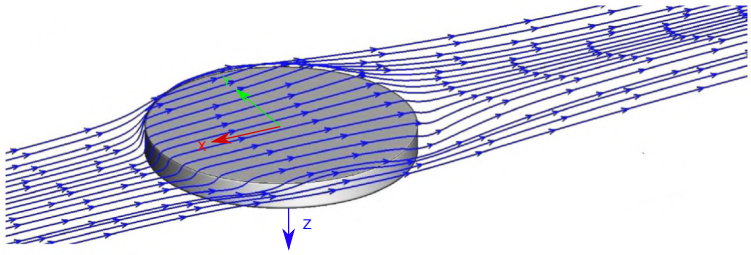


Figure 4.9: Streamline flow across disk

Due to the shape of Manta it is believed hydrodynamic coefficients in heave will be the dominating term in both the added mass and damping matrices due to its large displacement effect to that of surge and sway. It is also believed that the displacement effect in yaw will be small compared to that of roll and pitch for added mass and hydrodynamic damping coefficients.

4.4.2 Shape and Reynolds Number Effects

An important quantity when it comes to viscous damping is the Reynolds number, derived in Equation 4.26. For any boundary layer there exists a Reynolds number for which below the boundary layer is laminar and above the boundary layer is turbulent. For Reynolds number less than $R_n = 10^5$, based on free stream velocity and overall length, the boundary layer is laminar and so the skin friction coefficient decreases with increasing Reynolds number. In the transition regime $5 \times 10^4 < R_n < 5 \times 10^5$, the boundary layer is turbulent over the aft portions and there is an increase in drag coefficient with increasing Reynolds number as the transition point moves forward and the relatively high drag turbulent boundary layer makes it contribution to skin friction drag. For Reynolds number in excess of $R_n = 10^6$ the skin friction drag is largely due to the turbulent boundary layer and the drag coefficient decreases slowly with increasing Reynolds number [7, Blevins, 1984].

The Reynolds number is an important parameter for vortex shedding as it can indicate how the vortices are shed or if vortices will shed. The high drag on bluff bodies (spheres, boxes etc) is at least partially associated with vortex shedding.

4.4.3 Method, Calculations and Results

In this section we will use empirical equations and our collected knowledge in past sections to estimate rigid-body kinetics, hydrodynamics and hydrodynamic damping for the Manta AUV. All calculations are done in Matlab, see attached Listing (1.1) in Appendix D.1 for details of code.

Moment of Inertia, Geometry and Fluid Properties

Autodesk AutoCAD provides all the necessary information such as total mass, center of gravity, moment of inertia, material density, total surface area and total volume for the Manta AUV. Make notice that the CAD assembly of Manta shown in Figure 4.10 does not include embedded electronics and peripheral devices such as cameras, electronics, DVL, battery pack and sensor skid. They contribute about $14.0kg$ to the total weight of the vehicle, giving the total mass of Manta about, $m = 30.7[kg]$.

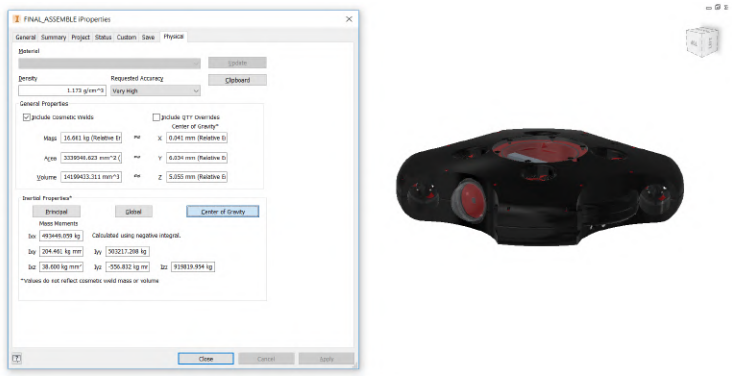


Figure 4.10: Properties of the Manta AUV in Autodesk AutoCAD

From AutoCAD we get the total surface area of Manta:

$$S = 3.3395[m^2] \tag{4.35}$$

For this project we design the vehicle for a presumed maximum transit velocity of $U = 0.7[m/s]$. Assuming that the Manta AUV will travel no faster than $0.3[m/s]$, the corresponding Reynolds number for the geometry of Manta will be:

$$R_n = \frac{UD}{\nu} = \frac{0.3[m/s] \times 0.72[m]}{1.0 \times 10^{-6}[m^2/s]} = 5.04 \times 10^5[-] \tag{4.36}$$

From section [4.4.2] a Reynolds number of $R_n = 5.04 \times 10^5[-]$ suggest that we are in a transition regime with both laminar and turbulent boundary layers.

Rigid-Body Inertia Matrix

The rigid-body inertia matrix is found using our knowledge of mass and inertia found in Autodesk AutoCAD inserted into Equation 4.37. The results are:

$$\mathbf{M}_{RB} = \begin{bmatrix} 30.7000 & 0 & 0 & 0 & 3.0700 & 0 \\ 0 & 30.7000 & 0 & -3.0700 & 0 & 0 \\ 0 & 0 & 30.7000 & 0 & 0 & 0 \\ 0 & -3.0700 & 0 & 0.5032 & -0.0002 & 0.0005 \\ 3.0700 & 0 & 0 & -0.0002 & 0.4934 & 0 \\ 0 & 0 & 0 & 0.0005 & 0 & 0.9198 \end{bmatrix} \tag{4.37}$$

Added Mass

Empirical values for the shapes of oblate spheroids and circular disks are depicted from [9, Values for added mass, Table B6 and B7]. Table 4.3 shows the range of added mass values for the geometry of Manta.

Added Mass, [M_A]

DOF	Oblate Spheroid, $l/h = 2.0$	Circular Disk, $l/h = 4.0$	Unit	Relative Difference
Surge	13.7997	7.7456	kg	78.2%
Sway	13.7997	7.7456	kg	78.2%
Heave	49.7679	105.9014	kg	-53.8 %
Roll	0.4080	1.6104	kgm^2	-74.6 %
Pitch	0.4080	1.6104	kgm^2	-74.6 %
Yaw	0.0	0.0	kgm^2	0 %

Table 4.3: Empirical Added Mass

Comparing with the results for added mass of the *Seabotix LBV600-6* in [21, Eidsvik, Master thesis 2015], the results for Manta seems very reasonable. It is believed that the added mass matrix of Manta is somewhere in between the added mass matrix of a identically sized oblate spheroid and a circular disk. To reduce the margin of error we choose to weight each matrix 50 % each. However the results for added mass in heave is somewhat inflated, due to the fact the projected area of manta onto the xy-plane is not a perfect circle. We do not want to overcompensate the added mass as this might make our controllers overcompensate the dynamics of Manta in simulation and deem our controllers aggressive and unstable when tested in the field. Therefore we choose to weight the added mass of the oblate spheroid in heave at 100 % making it less inflated. The results show the following:

$$\begin{aligned} \boldsymbol{M}_A &= \boldsymbol{M}_{A_{oblate}} \times \boldsymbol{w}_{6x6} + \boldsymbol{M}_{A_{disk}} \times (\boldsymbol{I}_{6x6} - \boldsymbol{w}_{6x6}) \\ &= \begin{bmatrix} 10.7727 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10.7727 & 0 & 0 & 0 & 0 \\ 0 & 0 & 49.7679 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.0092 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.0092 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{aligned} \tag{4.38}$$

where $\boldsymbol{w}_{6x6}$ is the weighting matrix.

This result correspond to our anticipated guess that the surge and sway terms would be smaller to that of heave. Also the empirical data for oblate spheroid and circular disk neglect the added mass effects in yaw due to the low displacement effect for these shapes in yaw motion, which is reasonable. And again, comparing with the results for added mass of the *Seabotix LBV600-6* in [21, Eidsvik, Master thesis 2015], the results for Manta seems very reasonable.

Linear Damping

The linear damping can be very hard to estimate analytically. A number for general approximations exists, though they are for surface vessels and make use of strip theory (only valid for slender body shapes). A possible alternative is to make use of empirical estimates for the linear skin friction damping of a floating vessel suggested in [27, Fossen, 2011, page 125]. This is in fact a good choice because it considers a sea keeping scenario with a PD-controller for dynamic positioning, which is the controller we plan to use. See Chapter 7 for details of the controller.

Consider a second order system stabilized by a PD-controller:

$$m\ddot{x} + (d + K_d)\dot{x} + K_p x = 0 \tag{4.39}$$

This system satisfies

$$2\zeta\omega_n = \frac{d + K_d}{m}, \qquad \omega_n^2 = \frac{K_p}{m} \tag{4.40}$$

In the uncontrolled case $K_p = K_d = 0$. Hence, the time constant becomes

$$T = \frac{m}{d} \tag{4.41}$$

In controlled closed loop, K_p and K_d are positive constants satisfying

$$\begin{aligned} \frac{1}{T} + \frac{K_d}{m} &= 2\zeta(\frac{2\pi}{T_n}) \\ &= \frac{4\pi\zeta}{T_n} \end{aligned} \tag{4.42}$$

if K_d is specified as $K_d/m = 1/T$, the bandwidth of the closed-loop system is approximately the double [27] and it follows that

$$\frac{2}{T} = \frac{4\pi\zeta}{T_n} \tag{4.43}$$

The relationship between the time constant T, relative damping ratio ζ and the natural period T_n under feedback control is

$$T = \frac{T_n}{2\pi\zeta} \tag{4.44}$$

The corresponding feedback gains are

$$K_p = m\omega_n^2, \qquad K_d = m/T. \tag{4.45}$$

It is fair to assume that the natural period in translation (surge and sway) for a underwater vehicle is way much lower than that for a large ship ($100s \leq T_n \geq 150s$); more likely in the range ($1s \leq T_n \geq 15s$). For this project we assume a natural period in surge, sway and yaw of $1.3s$ In [27, Fossen, 2011, page 125] it is suggested a relative damping factor of $\zeta = 0.1$ for surface vessels which is highly underdamped. With a relative damping factor of $\zeta_{11} = \zeta_{22} = \zeta_{66} = 0.1$ gives the time constants $T_{surge} = T_{sway} = T_{yaw} = 2.1s$. This seems like a reasonable choice for a small underwater vehicle. Using this together with Equations 4.45 and using the added mass information for two different shapes provided in Table 4.3, the linear viscous damping coefficients B_{11_v}, B_{22_v} and B_{66_v} are found by inserting into Equation 4.46:

$$\begin{aligned}
B_{11v} &= \frac{m + M_{A_{11}}}{T_{surge}} \\
B_{22v} &= \frac{m + M_{A_{22}}}{T_{sway}} \\
B_{33v} &= 2\Delta\zeta_{heave}\omega_{heave}[m + M_{A_{33}}(\omega_{heave})] \\
B_{44v} &= 2\Delta\zeta_{roll}\omega_{roll}[I_{xx} + M_{A_{44}}(\omega_{roll})] \\
B_{55v} &= 2\Delta\zeta_{pitch}\omega_{pitch}[I_{yy} + M_{A_{55}}(\omega_{pitch})] \\
B_{66v} &= \frac{8\pi\zeta_{yaw}[I_z z\ M_{A_{66}}]}{T_{n,yaw}}
\end{aligned} \tag{4.46}$$

where the natural (when the systems swings freely) frequencies ω_n in heave, roll and pitch can be found as:

$$\omega_n = \frac{2\pi}{T_n} \tag{4.47}$$

By looking at a *free decay model test* performed on the *Qtrencher ROV* model with similar geometry ($L, W, H = 0.542, 0.438, 0.442[m]$) and mass ($m = 24.2[kg]$), the results show a natural period of $1 - 2$ seconds for both heave, roll and pitch at 5 meters depth [5, Hydrodynamic Characteristics of ROVs].

Using this information we assume the same natural period of Manta in heave is a bit larger due to its large surface area in the xy-plane. By that, we set the natural period equal to

$T_{n,heave} = 4s$. The natural freq When using a PD-controller for station keeping in heave for a positively buoyant submersible, you will most likely experience an oscillatory motion without an integral term. The oscillation suggests that the system is underdamped. For this thesis the relative damping factor in heave is set equal to $\zeta_{3,heave} = 0.2$.

The natural periods in roll and pitch are mostly determined by the restoring moments caused by the location of buoyancy center and gravity center of the submersible. In [27, Fossen, 2011, page 125] it is suggested a relative damping factor of 0.4-0.6 for ships with anti-roll tanks. Manta is ballasted such that it is slightly underdamped with a rapid natural period compared to that in heave. For this reason the relative damping factor in roll and pitch are set to $\zeta_{roll} = \zeta_{pitch} = 0.8$ and the natural period $T_{n,roll} = T_{n,pitch} = 1.0s$.

All this combined we are able to find B_{33_v}, B_{44_v} and B_{55_v} and using the added mass information for two different shapes provided in Table 4.3. The results for two different matrices of linear damping are displayed in Table 4.4:

Linear Damping, [$\mathbf{D}_L$]

DOF	Oblate Spheroid, $l/h = 2.0$	Circular Disk, $l/h = 4.0$	Unit	Relative Difference
Surge	21.0211	18.1612	kg/s	15.7%
Sway	21.0211	18.1612	kg/s	17.7%
Heave	46.0197	85.7856	kg/s	-45.3 %
Roll	0.5726	8.0159	kgm/s	-46.3 %
Pitch	0.5726	7.9299	kgm/s	-46.3 %
Yaw	1.1559	0.0	kgm/s	0 %

Table 4.4: Linear Viscous Damping

And again, if our assumptions about natural periods and relative damping factors, combined with the our best estimate of added mass in Equation 4.38, we end up with the final result of linear damping equal to:

$$\mathbf{D}_L = \begin{bmatrix} 19.5912 & 0 & 0 & 0 & 0 & 0 \\ 0 & 19.5912 & 0 & 0 & 0 & 0 \\ 0 & 0 & 50.5595 & 0 & 0 & 0 \\ 0 & 0 & 0 & 13.3040 & 0 & 0 \\ 0 & 0 & 0 & 0 & 13.218 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.1559 \end{bmatrix} \tag{4.48}$$

Take note that the method used is primarily aimed at surface vehicles, and therefore our estimates may not be realistic. However, by comparing with with the hydrodynamic coefficients found from both analysis and experimental trials of other submersibles found in [21, Eidsvik, Masther thesis 2015], the results looks very promising.

Non-linear Viscous Damping

As stated earlier the non-linear viscous damping (often called quadratic damping) represent all higher order damping contributions mainly consisting of turbulent boundary layer skin friction, vortex shedding and viscous pressure drag. All three terms are tightly coupled and therefore it is easier to collect them into one coefficient. The non-linear viscous damping cannot be found using potential theory because it assumes that the fluid is irrotational, incompressible and inviscid.

To find the non-linear terms in translation (surge, sway and heave) we will take use of two methods. The first method is most concerned with the turbulent boundary layer skin friction in surge, whilst the other one is more concerned with the vortex shedding and viscous pressure drag in heave.

1. **Nonlinear viscous damping in surge and sway**
   The first method is a method for computing *nonlinear surge damping* for ships, but will will be applicable for sway as well due to the symmetrical body of Manta. The method is depicted from [27, Fossen, 2011, page 125] and it goes as follows:

$$X_{|u|u} = -\frac{1}{2}\rho S(1+k)C_f(u_r)|u_r| \tag{4.49}$$

$$C_f(u_r) = \frac{0.075}{(log_{10}R_n - 2)^2} + C_R \tag{4.50}$$

   Where $X_{|u|u}$ is a quadratic damping term, ρ is the water density, S is the wetted surface of the vehicle, $u_r = u - u_c$ is the design speed, k is the form factor giving a viscous correction, C_F is a flat plate friction for turbulent boundary layer using IITC'57 line which is the most common one to calculate resistance on ships. C_R is *residual friction* due to hull roughness, pressure resistance, wave-making and wave-breaking. We choose to neglect C_R as this coefficient is mostly concerned with surface vessels in wave-zone.

   When computing the turbulent boundary layer skin friction we neglect the pressure difference. The pressure distribution around the vehicle hull is associated with the flow velocity (Bernoulli's equation). The induced velocity at the top center and bottom center is due to the displacement effect of Manta's hull. The induced velocity will then again contribute to the skin friction resistance. We compensate for this by using the form factor, k [86]. For ships in transit k is typically 0.1, for ships in DP, k is typically 0.25 as water flows in from all directions. Due to the fact that Manta has rather little curvature on the top and bottom, it is believed that the induced velocities are rather small, and there is little contribution to the total skin friction resistance. Therefore the form factor is set equal to $k = 0.1$. Using the computed Reynolds number, wetted surface area, form factor and design speed we are able to estimate the nonlinear viscous damping in surge and sway.

2. **Nonlinear visous damping in heave**
   The second method is depicted from [21, Eidsvik, 2015] and takes use of Morison's equation to find the viscous pressure drag due to body geometry and vortex shedding for ROVs. The method takes use of empirical coefficients such as the drag coefficient to describe the viscous drag force on a specific geometry. The damping term in Morison's equation is:

$$Z_{|w|w} = -\frac{1}{2}\rho C_D \lambda A_p^{xy} |w| \tag{4.51}$$

   where C_D is the 3-dimensional drag coefficient, λ is a scaling factor, C_p^{xy} is the projected area in the xy-plane and w is the absolute velocity in heave.

   Note that the reference area in Morison equation [22, See Faltinsen, 1990 - Morison's equation] is scaled using the projected area. The difference between strip theory (2D) and the 3D drag coefficient then needs to be calculated and collected in the λ parameter. As the geometry of Manta is known, fitting geometries are found in [99, Cengel, 2010, Drag and Lift]. We pick a thin circular disk for 3-dimensional drag coefficient. For 2-dimensional drag coefficient the empirical data assume that the body shape is applicable for use of strip theory (integral over the length of the body), a circular shape like the projected area of Manta into the xy-plane is unfitted for strip theory. The closest resemblance is a thin rectangular plate. The corresponding drag coefficients are seen in Table 4.5.

Table 4.5: Table 11-1 amd 11-2 in Fluid Mechanics Fundamentals and Applications, Cengel 2010

L/D	Thin rectangular plate, $C_D(2D)$ $Rn \geq 1 \times 10^4$	Thin Circular disk, $C_D(3D)$ $Rn \geq 1 \times 10^4$
all ratios	1.9	1.1

   Using the drag coefficients and our knowledge of the project area in xy-plane we get the following parameters.

$$\lambda = \frac{C_D(3D)}{C_D(2D)} = \frac{1.1}{1.9} = 0.58[-], \qquad A_p^{xy} = \frac{\pi D^2}{4} = 0.40[m^2] \tag{4.52}$$

   From the Equation 4.51 and the parameters in Equation 4.52, together with a design speed in heave of $w = 0.2[m/s]$ we are able to find the nonlinear viscous damping coefficient.

The final results for non-linear damping matrix is:

$$\mathbf{D}_{NL}(\boldsymbol{\nu}_r) = \begin{bmatrix} 7.3486|u_r| & 0 & 0 & 0 & 0 & 0 \\ 0 & 7.3486|v_r| & 0 & 0 & 0 & 0 \\ 0 & 0 & 26.1105|w_r| & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.53}$$

The estimates for nonlinear damping in the rotational DOFs is quite challenging, mainly because good empirical methods were not found. However, it is believed that the nature of the operations conducted by Manta should not involve large rotational angular velocities, and we therefore choose to not include nonlinear angular damping in our model.

By comparing the results for linear damping in matrix in Equation 4.48 with the nonlinear damping matrix in Equation 4.53, we see that the linear damping is the dominant effect for our design speed. This is also an anticipated result. The magnitude of the nonlinear damping coefficients corresponds to similarly sized ROVs found in [21, Eidsvik, 2015]

4.4.4 Conclusion of Case Study

At current standing there is little ground to tell whether or not the derived estimates are somewhat close to reality. However, based on comparison to experimental studies on similar sized vehicles, the numerical estimates seem to be in the in range of values one could expect. Also, if the the assumptions made during the case study are realistic then the error margins will correspond to the relative difference columns in the in tables 4.3 and 4.4. It is expected that the behavior of the vehicle in simulation compared that to physical experiments will give a good indication of if the mathematical model is accurate.

4.5 Chapter Summary

The main goal of this chapter was to suggest an accurate and practical procedure for obtaining knowledge of the rigid-body kinetics and hydrodynamic properties of the Manta AUV. As the rigid-body inertia's can be extracted from AutoCAD, the main focus was the added mass forces and especially the hydrodynamic drag(damping) forces. As limited empirical or analytical methods existed for estimating these coefficients a substantial amount of effort was put into creating a method for complete estimate of the coefficients based on both analytical and empirical data from similar vehicles.

With a mathematical model representation of the Manta AUV it is natural to advance to creating a 3-D simulation testbed to deploy the model. The next chapter will explain details of designing vehicles and environment in Gazebo robot simulator.

CHAPTER 5

UNDERWATER SIMULATION TESTBED

This chapter is aimed at creating a high fidelity 3-D simulation tesbed using the Gazebo robot simulator with the UUV simulator plugin. The resulting simulator is an important piece in a agile and continuous development process, by allowing for rapid prototyping and testing.

5.1 Requirements Analysis for the Simulation Model

The simulation model should be the most accurate description of a system. It should include the marine craft dynamics, propulsion system, measurement system and the environmental forces due to wind, waves and ocean currents at best. It should also include other features not used for control and observer design that have a direct impact on model accuracy. The simulation model should be able to reconstruct the time responses of the real system and it should also be possible to trigger failure modes to simulate events such as accidents and erroneous signals. The need for a simulation environment that allows development of low-level controllers, observer design and high-level algorithms for complex behavior was identified as one of the challenges to overcome. During the requirements analysis for a simulation platform, the following basic requirements were defined:

1. **Communication with ROS**
   Interface for Robot Operating System (ROS) applications (the standard middleware to be used by the vehicles and systems involved in this project). For the sake of easy testing, tuning and deployment of new algorithms, it is really important that the same source code can be rendered onto the actual physical AUV with its corresponding hardware, without any reconfiguration and adaptation.
2. **3D visualization in real time**

To be able to test control algorithms and perception algorithms that react to the immediate environment, there is a need for a simulation testbed that presents itself in real time and that is identical or at least similar to what a camera, sonar etc will experience in the real world.

3. **Vehicle dynamics**
   The vehicle dynamics module must be able to estimate the thrust forces based on control input, and estimate the corresponding motion of the AUV. Basic specifications needed to simulate motion is the rigid-body kinetics such as moments and forces due to mass and inertia of the AUV. At best the simulator will also be able to compute hydrodynamic forces and hydrostatic restoring.

4. **Sensor models**
   The sensor models should simulate real sensor output based vehicle motions and its surroundings. In traditional simulators this is done with a *Hardware in the loop* (HIL) approach. We however, want to the sensors to be integrated as part of the simulation tool.

5. **Thruster dynamics**
   As thrusters usually have a nonlinear relationship between control input and actual thrust, there should possible to add information about thruster coefficients, thruster configuration, thruster inertia and thrust loss factor.

6. **Environmental disturbances**
   It should be possible to apply environmental disturbances. One of the steps to evaluate the vehicle's performance using a control strategy is to check how it would behave under different kinds of noise. For a underwater scenario disturbances can be underwater currents, sensor noise, loss of vision in water and more.

The focus of the search was to find a robotics simulation tool that was set for an open-source implementation with a permissive license. The first option was to use *MATLAB & Simulink*. Simulink offers a block diagram environment for model-based design, and even has available packages for communication with ROS. However, Simulink lacks the ability of 3D visualization of the robot in real-time and the source-code will need a re-configuration and adaptation to be rendered on the embedded systems of the Manta AUV. The second and third option was *Unity* and *Unreal Engine 4*. Unity and Unreal Engine are world leading real-time engines for game development. They produce by far the most realistic models of the world, and the graphics superior to other simulation tools. However, both Unity and Unreal Engine has poor documentation and support for Linux and ROS at the time of writing. The final and best choice was the *Gazebo* robot simulator. It fulfilled most criterias needed for our simulation model, but lacked the ability to simulate the effects due to hydrodynamic and hydrostatic forces. In that case an EU-funded project SWARMSs [1] (Smart and Networking Underwater robotics in Cooperation Meshes) has developed a underwater plugin for Gazebo called the *UUV Simulator*, which is essentially an extension to underwater scenarios. This ended up being the simulator tool of choice.

[1] http://www.swarms.eu/

5.2 Working with the UUV Simulator

Whereas several good open-source simulation environments exist for aerial as well for ground based robots, the scenario for underwater robots is less appealing. This is mostly due to the difficulty to realistically model not only hydrodynamic forces acting on the robot itself, but also the complex environments in which the operations take place. In contrast to other existing solutions, the UUV simulator reuses and extends a general-purpose robotics simulation platform Gazebo to underwater environments.

The simulation currently includes typical underwater sensor models (e.g sonar, pressure sensor and Doppler velocity Log), actuator dynamics models (e.g thruster and fins). Gazebo is already capable of rigid-body dynamics simulation due to its integration with four physics engines. In underwater environments, hydrodynamic and hydrostatic forces and moments must also be taken into account. This is achieved using additional plugins implemented as part of the UUV simulator. Plugins can access the simulator by using the Gazebo API. When communication is established the ROS system can transmit information via topics by using protocol buffer messages and apply torques and forces to objects in the scenario. All environment objects such as world heightmaps, buildings and so on must be initialized through a SDF file - an XML format that describes objects and environments for robot simulators, visualization, and control. Objects that are communicating with ROS such as robots, sensors and actuators must be initialized in Universal Robotic Description Format (URDF) - a similar XML-format as the SDF-files.

The robot description must also contain a geometry file. Visual geometries are rendered in COLLADA file format and the collision geometry as provided in standard CAD format such as STL.

5.3 *Case Study: Design and Implementation of Underwater Environment, Vehicle Dynamics and Thruster Dynamics for the Vortex NTNU, Manta AUV*

In this section we will design a underwater environment and a new vehicle for the Gazebo robot simulator by taking use of the UUV Simulator plugin and AutoCAD. Through our knowledge into the dynamics of marine vehicles and its surroundings we should be able to create a simulation testbed that is somewhat realistic to what the vehicle will experience in real-life. The simulation tool is highly important for the software development process, and allows for configuration and testing of software.

The case study will be for the Manta vehicle in the Transdec RoboSub pool environment in 1:1 scale. There will also be created a 1:1 scale simulation tesbed of the Marine Cybernetics Laboratory at the Department of Marine Technology, NTNU [2]. Both processes are done in similar matter, therefore the latter will not be described in a case study.

5.3.1 Creating the Environment

The competition ground is completed in AutoCAD according the dimensions of the Transdec International Robosub pool. The pool is provided as an inertial earth-fixed NED frame with origin in the center and magnetic north point along the y-axis of the model frame. UUV Simulator includes a model named *ocean box* consiting of texturizes water and light attenuation. The files are collected together in to a world-file named *robosub_2019.world*. The comparison between the real world photo and the CAD model is seen in Figure 5.1.

For more details about the implementation, see source code in Appendix D.9.

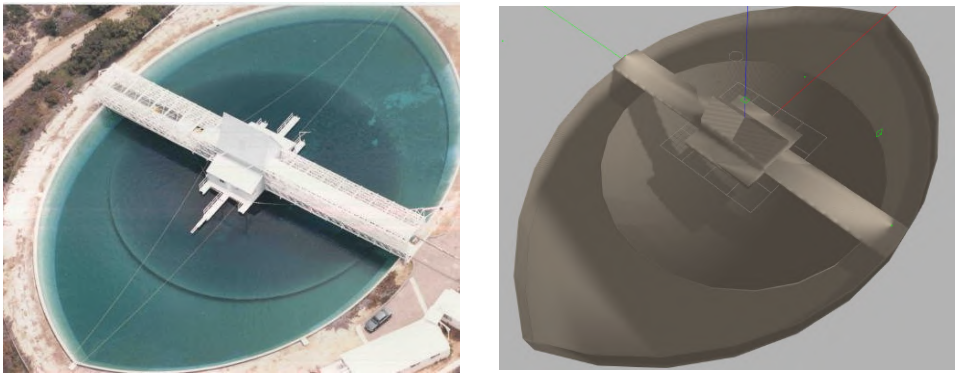


Figure 5.1: Transdec CAD

[2]Marine Cybernetics Lab website https://www.ntnu.edu/imt/lab/cybernetics

Creating the Gates

One of the main tasks of the Robosub competition is to pass through a submerged gate. The gate is created as a collection of visual geometries and collision geometries completed in *gate.sdf*-file. The gate is positioned at its respective coordinates and orientation in the inertial earth-fixed NED frame in the *robosub_2019.world* file. The illustration provided by Robosub and the corresponding gate created in Gazebo is shown in Figure 5.2.

For more details about the implementation, see source code in Appendix D.11.

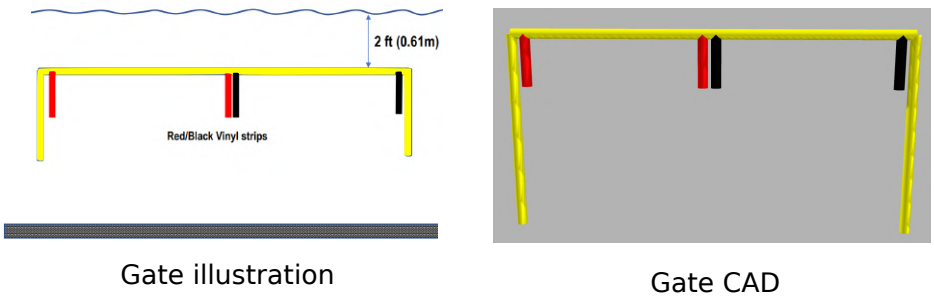


Figure 5.2: Gate CAD

Creating the Dices

Another task for the competition is to perform object detection and classification of dices. The dices are created as a collection of visual geometries and collision geometries completed in *gate.sdf*-file. The dices are positioned at their respective coordinates and orientation in the inertial earth-fixed NED frame in the *robosub_2019.world* file. For more details about the implementation, see source code in Appendix D.10.

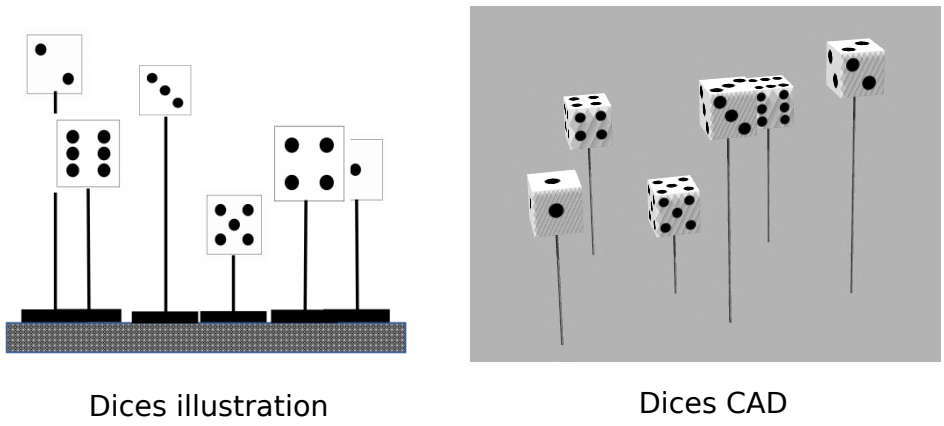


Figure 5.3: Dices CAD

5.3.2 Creating the Robot

In order to design an underwater vehicle, it is essential or compulsory to have strong fundamental knowledge about the processes and physical laws governing the underwater vehicle in its environment. For this part we will use our knowledge into the modelling and hydrodynamics from Chapter 4 to realize and the simulation model for the AUV in its operational environment.

First of all we include the visual geometry and the collision geometry of the vehicle. The visual mesh is exported from AutoCAD as a COLLADA file. The collision geometry has been largely simplified due to computational matters. The geometries are then added to the vehicle configuration file (see Appendix D.5). The visual geometry and the collision geometry serves as the vehicle body frame - named */manta/base_link* in ROS. All additional equipment such as sensors and actuators that are attached to the main body will be described as a static transformation (rotation and translation) relative to the body frame.

Adding Sensors and Thrusters

An UUV can be described in Gazebo as a set (model) of rigid bodies (links) fixed to a main body. The Manta vehicle consist of a total of eight thrusters, two cameras, one IMU, one DVL, one pressure sensor, a magnetometer (currently not in use) and it is also planned for a underwater sonar. In Gazebo the sensor and actuators are all added to the vehicle configuration file with their respective coordinate frame consisting of a position and orientation relative to the */manta/base_link* frame. The ROS topic of each module where the data streams are published, along with the Gazebo plugin used to integrate them into the ROS network are all shown in Figure 5.4.

Note that all sensors and actuators comes as default in Gazebo and UUV simulator. You only have to configure equipment parameters so that they correspond to respective equipment. Equipment information are found by looking up their data sheets.

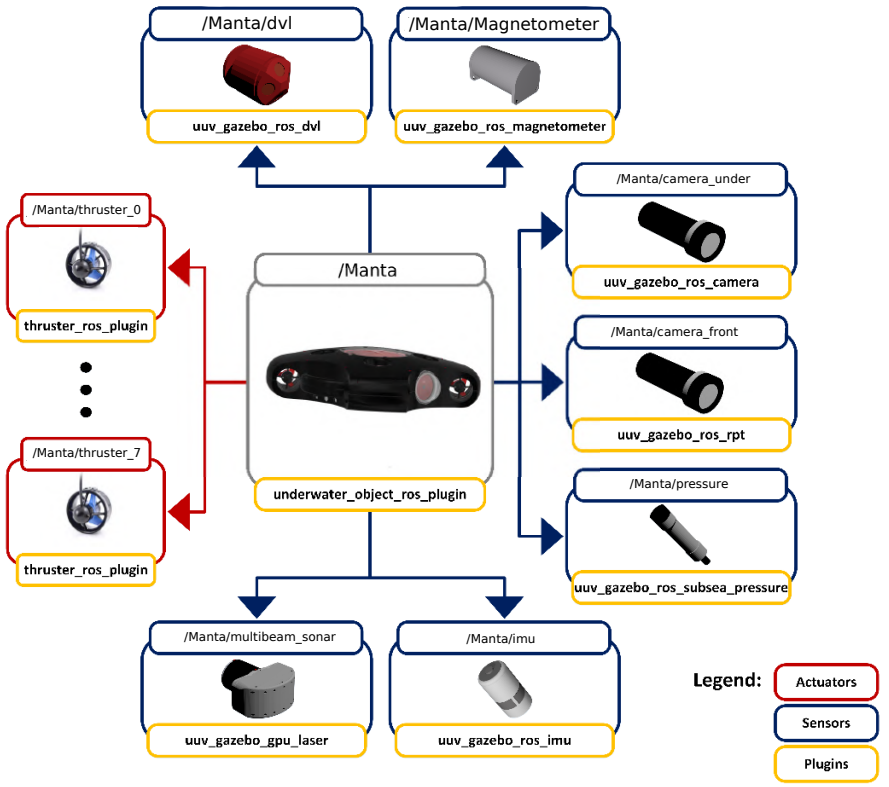


Figure 5.4: Manta AUV interfacing with the UUV Simulator plugin in Gazebo Robot Simulator. *Courtesy: [43, Morena, 2016] for original picture*

The static transformation between bodies are always published by the ROS network, and therefore available for controllers and state estimators to use when taking into account the relative transformation between coordinate frames of interest. When running ROS you can check e.g. the transformation between inertial and body frame, or body and thruster frame by running the following commands in the Linux terminal:

rosrun tf tf_echo /world_ned /manta/base_link
rosrun tf tf_echo /manta/base_link /thruster_0

for more details about the implementation of sensors and actuators, see source code D.8 and D.7 in the Appendix.

Setting up Rigid-Body Kinetics

For pure rigid-body kinetics simulation, Gazebo has to be provided a robot description with correct values for mass, inertial tensor, collision geometry and joint parameters.

These values are all depicted from the AutoCAD model, and then they are composed into a *manta_base.xacro* -file. All values for the rigid-body inertia matrix are provided by the M_{RB} matrix in Equation 4.37 from the previous case study.

For details on the implementation, see Appendix D.5.

Hydrodynamic and Hydrostatic Modelling of Vehicle

Gazebo performs dynamic simulation and takes as inputs the forces and torques that acts on a body. The dynamics model for underwater vehicles takes into account the relative velocity $\nu_r = \nu - \nu_c^b$ of the vehicle with respect to its surrounding fluid. Through plugins, it is possible to add any kind of effort and thus to consider an hydrodynamic model. Gazebo integrates the default rigid-body equations of motion with six degrees of freedom:

$$\tau_g = M_{RB}\dot{\nu} + C_{RB}(\nu)\nu + g_0 \tag{5.1}$$

(See Equation 4.8 for explanation of the terms.)

where τ_g are the external forces and torques. Through the UUV simulator, the hydrodynamic effects are applied by setting the external forces and torques to [43]:

$$-\mathbf{M}_A\dot{\nu}_r - \mathbf{C}_A(\nu_r)\nu_r - \mathbf{D}(\nu_r)\nu_r - \mathbf{g}(\eta) = \tau_g \tag{5.2}$$

See Equation 4.14 for explanation of the terms.

The Gazebo simulator subscribes to a ROS topic that gives the intensity and direction of the water current. For this project we have chosen to set the current velocity $\nu_c = 0$. It is thus possible to have the current vary if needed. The buoyancy direction is of course opposed to the gravity, and becomes null as the vehicle goes above the surface [37]. The external forces and torques applied in each link are directly transmitted to the main link, in such way that the resultant efforts are applied in the model's center of gravity (COG). A Gazebo world file is given and handles this plugin.

All the values are picked from the results of added mass, linear damping and nonlinear damping in Section 4.4.3 and added to the vehicle properties in Gazebo. For details about the implementation, see source code D.6 in the Appendix.

5.3.3 Thruster Configuration

The thruster configuration consist of eight thrusters are positioned as shown in Figure 5.5. Thruster number 0,3,4 and 5 are all oriented in a 45 degree angle and contribute to motions in surge, sway and yaw. Thruster number 1,2,5 and 6 have oriented upwards and contribute to motions in heave, roll and pitch. Their respective positions and orientations are included in the robot's thruster configuration file, see Appendix D.7.

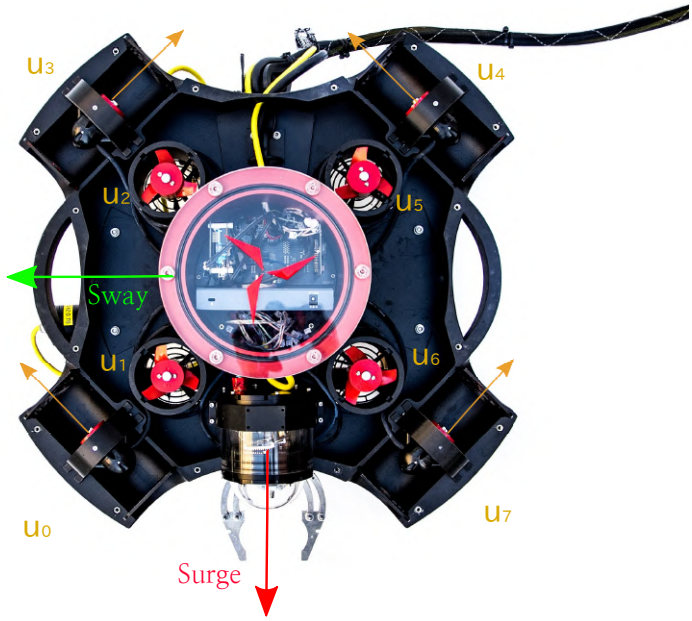


Figure 5.5: Thruster configuration, Manta AUV

5.3.4 Propeller Dynamics

Due to the fact that the thrusters themselves have an inertia and does not respond immediately to a desired control input it is necessary to add their dynamics into the simulation model. The thruster dynamics and their conversion curve are added through a *thruster plugin* in the UUV Simulator. The plugin is described in Figure 5.6.

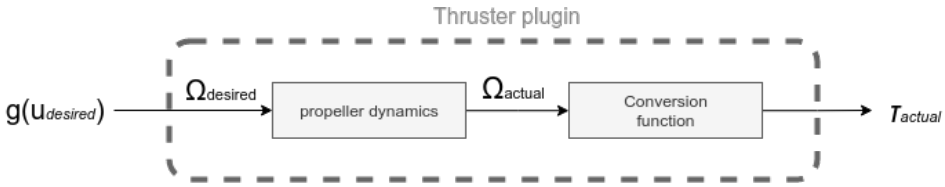


Figure 5.6: Thruster Plugin Model. *Figure inspired by [43, Morena, 2016]*

Where $u_{desired}$ is an instance of the control vector $\mathbf{u}_d$ saying what kind of thrust should be allocated to each of the thrusters. The thrust mapped from the conversion function $g(u_{desired})$ used by your controller to a corresponding angular velocity. The desired angular velocity $\Omega_{desired}$ is then picked up by the thruster plugin.

The propeller dynamics depend on whatever thrusters / propellers that you have for you vehicle. For our application we choose to describe the dynamics through a first order

transfer function, with time constant T

$$\begin{aligned} \Omega_{actual} &= \frac{\Omega_{desired}}{Ts+1} \\ &= \mathcal{L}^{-1}\{\frac{\Omega_{desired}}{Ts+1}\} \\ &= \Omega_{desired}(1-e^{\frac{t}{T}}) \end{aligned} \tag{5.3}$$

The time constant for the *Blue Robotics T200* [3] thrusters are not provided by the technical documentation, but we choose to set $T = 0.2s$ which means that the thrusters will achieve 90% of desired thrust within about 0.5s, and 100% thrust after about 1s. See Figure 5.7 for the step response for two different time constants.

The propeller dynamics are added to the thruster configuration file used by Gazebo. The implementation be seen in Appendix D.7.

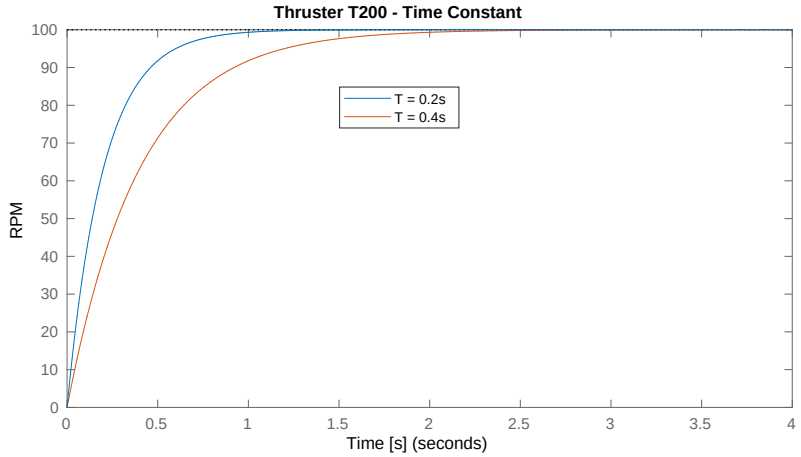


Figure 5.7: Step response of first order system

5.3.5 Conversion Function

By using available performance measurements for the T200 thrusters at their website, the conversion relationship between RPM and actual thrust can be plotted.

By assuming a second order relationship between RPM and produced thrust, we get the conversion function:

$$t_i = k_{rotor} \cdot |\Omega| \cdot \Omega \tag{5.4}$$

[3] https://www.bluerobotics.com/thruster/

where t_i is the thruster force, and Ω is the angular velocity of the rotor blade. By now performing a curve fit to the actual relationship graph, we get the best resemblance by having a rotor constant equal to, $k_{rotor} = 0.000004$. See Figure 5.8.

The rotor constant are added to the thruster configuration file used by Gazebo. The implementation be seen in Appendix D.7.

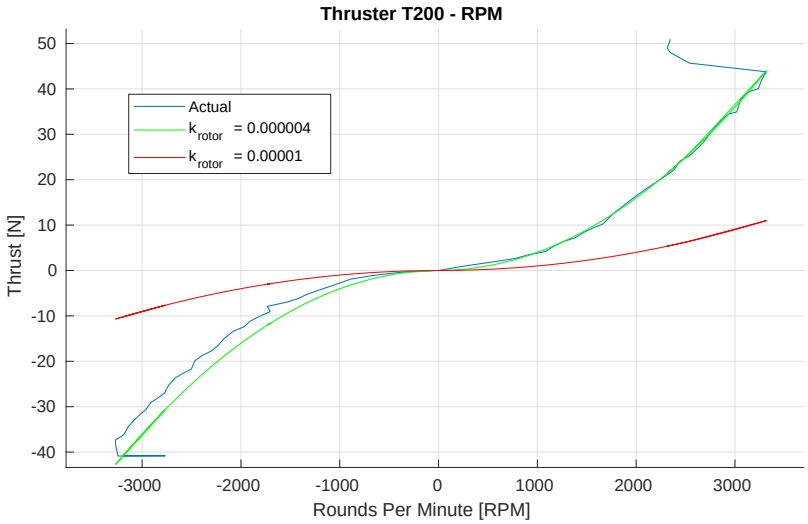


Figure 5.8: Thruster dynamics - Bluerobotics T200

5.3.6 Conclusion of Case Study

The completion of the Transdec Robosub competition pool along with the Manta AUV are shown in Figure 5.9. The resulting simulation testbed includes the hydrostatics and hydrodynamics forces and moments that act on a submerged body. The data stream from each sensor can be read through their respective ROS topics, while the actuators both take control input by publishing a control command on its respective ROS topic. The position and orientation of the Manta AUV vehicle as provided by the Gazebo simulator, and can serve as *ground truth* reference when testing control and navigation algorithms.

Figure 5.9: Screengrab of Manta in the Robosub Transdec pool.

5.4 Chapter Summary

The result of this chapter is a new vehicle and two new environments compatible with ROS and Gazebo. The source-code is made open-source through GitHub for anyone to use. Link to the project is found in Appendix A.

CHAPTER 6

ONBOARD AUTONOMOUS UNDERWATER NAVIGATION

With the emergence of inspection-class AUVs; the navigation and navigational accuracy are becoming increasingly important in order for the AUV to complete its task. Without an operator in the loop, the vehicle must use sensors to determine not only to estimate its own position, velocity and orientation; but also states of observed objects of interest. This chapter will cover in depth robot localization and robot perception. The chapter will also describe briefly how to implement path planning ones sonar or stereo vision is mounted on the Manta AUV.

6.1 What is Underwater Navigation?

Navigation attempts to provide a means for an autonomous robot to move safely from one location to another, where the current state of the vehicle is seen with respect to a specific domain object; e.g. when the vehicle targets a domain object in the environment. The general problem of navigation can be formulated in three questions [41],

- *Where am I?* The robot has to know where it is in order to make useful decisions. Finding out the whereabouts of the robot is commonly known as *robotic localization*.
- *What is around me?* In order to fulfill some task the robot must perceive and make sense of its immediate environment to know where it is going. It has to identify a goal and this problem is therefore known as *goal recognition*; a sub-field to *robot perception*.

- *Where is my goal and how do I get there?* Once the robot knows where it is and its surroundings, it has to decide on how to safely maneuver to its goal. Finding a way to get to the goal and avoiding hazards is known as *path planning*.

6.2 Requirements for AUV Navigation

In determining its location, the AUV has access to two kinds of information. First, it has some *a-priori information* gathered by the robot itself or supplied by an external source - such as human operator - in the *initialization* phase. Second, the robot gets information about the environment through every observation and action it makes during *navigation*.

6.2.1 A-priori Information

In general, the *a-priori* information supplied to the robot describes the environment in which it operates. It specifies certain features that are time-invariant and thus can be used to determine a location. The a-priori information can come in different forms. Examples of these are *maps* and *cause-effect relationships*.

Maps

The robot may have access to a map describing the spatial characteristics of the environment. Such a map can be *geometric* or *topological* [83]. Geometric maps describe the environment in metric terms, much like normal road maps. Topological maps describe the environment in terms of characteristic features at specific locations and instructions on how to get from one location to another. A map can given by an external source in an *initialization* phase, or it can be learned by the robot in advance through an *exploration* phase. A third option is that the robot learns the map of the environment in the midst of navigating through it. This third option - in the field of robotics - is known as *Simultaneous Localization and Mapping* (SLAM) [95].

Cause-effect Relationships

Another way of supplying a-priori information to the robot is in terms of *cause-effect* relationships [83]. Given the input from observations, these relationships tell the robot where it is. This will enable the robot to correct its localization estimates along the way.

6.2.2 Navigational Information

The second type of information to which a robot has access is *navigational information*, i.e., information that the robot gathers from its sensors while navigating through the environment [58]. A robot typically performs two alternating types of actions when navigating;

it *drives* around or *acts* in the environment on one hand, and it *senses* the environment on the other. These two types of actions give rise to two different kinds of position information.

Driving

For an AUV the guidance system will send instructions to the motion control system on what to do. Further the motion control system will compute the proper actions for the actuators on which will fulfill the instructions of the guidance system. Knowing the effects of actions executed by the motion control (driving) system, gives a direct indication of the location of the vehicle after execution of these actions. By monitoring what the motion control system actually does using sensors, the displacement of the robot vehicle can be estimated. This results in *relative position measurements*, or also sometimes referred to as *proprioceptive measurements* [76]. Relative position measurements are measurements that are made by looking at the robot itself only. No external information is used and these measurements can therefore only supply information relative to the point where the measurements were started. Acquiring relative measurements is also referred to as *dead reckoning* [58].

Sensing

The robot *senses* the environment by means of its *sensors*. These sensors give the momentary situation information, called *observations* or *measurements*. This information describes things about the environment of the robot at a certain moment. Observations made from the environment provide information about the location of the robot that is independent of any previous location estimates. They provide *absolute position measurements*, also referred to as *exteroceptive measurements* [76] to emphasize that the information of these measurement comes from looking at the environment instead of at the robot itself.

6.2.3 Multi-Sensor Fusion

Algorithms that solve the localization and perception problem combine initial information and relative and absolute position measurements to form estimates of the location of the robot at a certain time. If the measurements are considered to be reading from different sensors, the problem becomes how to combine the readings from different sensors to form a combined representation of the environment. Fusion of information from multiple sensors are important, in particular when not all sensors are able to sense the same. Some features may be occluded for some sensors, while being visible to others. Together the sensors can provide a more complete picture of a scene at a certain time. Multi-sensor fusion methods can rely on a *probabilistic approach* [58]. In the following section we will describe a probabilistic framework for multi-sensor fusion in the robot localization problem. This framework is a general framework describing the probabilistic foundations

of many existing, currently used, methods for solving the localization problem across domains, such as for underwater vehicles.

6.3 Robot Localization

For positioning, global positioning system (GPS) is commonly used in land or air vehicles, yet it cannot be used underwater [87] since the electromagnetic signals decay very quickly in water. Most AUVs employ an inertial navigation system (INS) as their main navigation sensor [68]. This is for many reasons; one of which is that the INS is a standalone system that can provide all of the required navigation data: position, velocity and orientation. However, even with a high-grade INS, the navigation solution drifts in time due to measurement errors of its inertial sensors. Therefore, INSs are usually aided by other external sensors or data such as the DVL for velocity, magnetometers for heading and depth/pressure sensor for altitude. This sort of solution is commonly referred to as Aided Inertial Navigation System (A-INS), and will be our approach to solve the robot localization problem.

6.3.1 AUV Localization Utilizing a Bayesian Framework

The general localization problem can be described as a *Bayesian estimation problem* [12]. We want to estimate the location of a robot given noisy measurements. In a sense, you can say that the robot has a *belief* about where it is located in terms of position and orientation. At any time, it does not consider just one possible location, but the whole space of locations. Based on all available information, the robot can believe to be at a certain location to a certain degree. The localization problem then becomes estimating the probability density of the space of all possible locations.

A Bayesian framework that estimates this probability density is the *Markov localization* framework originally derived by [96, Thrun et.al, 1999]. This framework captures the probabilistic foundations of many currently used localization methods. The Markov localization framework combines information from multiple senors in the form of relative and absolute measurements to form a combined belief of the location.

Beliefs

Belief. The robot has a *belief* about where it is. This belief is the probability density over all locations $\boldsymbol{x} = [x, y, z, \psi, \theta, \phi] \in \boldsymbol{D}$, where $\boldsymbol{D}$ is the set of all possible locations in your domain. We denote the belief by *Bel*

$$Bel(\boldsymbol{x}_k) = P(\boldsymbol{x}_k | d_0 ... d_k) \tag{6.1}$$

That is, the probability that the robot is at location $\boldsymbol{x}_k$ at time k given all information or data $d_0...d_k$ up to that time. This information also includes the a-priori information like for example a map of the environment. The location that given this probability distribution has the highest probability is the location at which the robot is most likely to be [58].

In Figure 6.1 you see a case of only two random variables in the 2D plane, with the probability density expressed normal to the plane, this is called a *bivariate distribution*. The concept however generalizes to any number of random variables. In the case of our 6-dimensional state vector $\boldsymbol{x}$, one can imagine the belief to be a *multivariate normal distribution* expressed in a 6-dimensional hyperplane. The goal of localization is to make this belief get as close as possible to the real distribution of the robot localization. The real distribution of the robot location has a single peak at the true location and is zero everywhere else. If the robot achieves this goal, then it will know exactly where it is located.

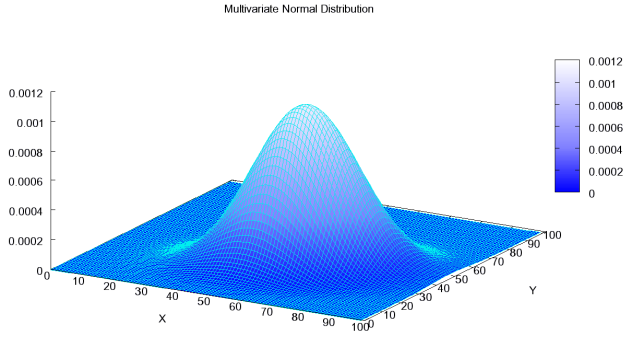


Figure 6.1: Bivariate Gaussian joint probability density function for random variables X and Y. *Courtesy: Multivariate normal distribution, https://www.wikipedia.org/*

Prior versus Posterior. During navigation, the robot has access to relative and absolute measurements. The robot incorporates these measurements into its belief to form a new belief about where it is. Commonly, a distinction is made between *prior* and *posterior* belief. The *prior* belief, denoted $Bel^-(\boldsymbol{x}_k)$, is the belief the robot has after incorporating all information up to time step k, also incorporating estimates derived from integrating a sequence of measurements. The *posterior* belief $Bel^+(\boldsymbol{x}_k)$ is the belief the robot has after it has also included the latest absolute measurement at time step k in its belief. The concept is seen in Figure 6.2.

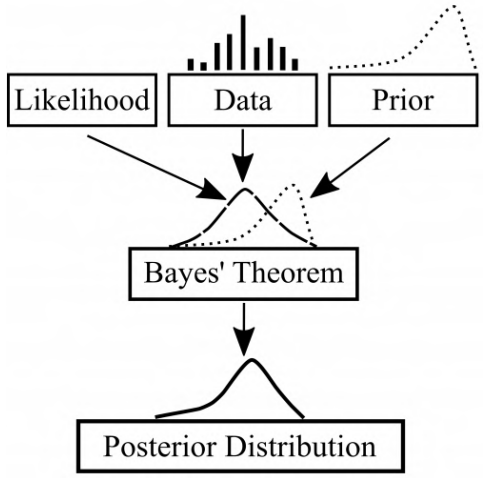


Figure 6.2: Prior vs posterior belief. *Courtesy: Hyperparameter Bayesian Optimization, https://medium.com/*

Probabilistic Acting and Sensing

To be able to update the beliefs with the latest measurement information, we need to express measurement information in probabilistic terms. It is necessary to define a probabilistic model for the acting, that is, the relative measurement, and a probabilistic model for the sensing, that is, the absolute measurements. These two models to derive a probabilistic model for AUV localization that computes the a collective belief by incorporating both types of information.

Acting. While performing actions in the environment, the AUV will change the position and orientation of the vehicle. Let $a_k \in A$, where a_k is an action from a set of possible actions A performed by the robot at time k. The way the location of the robot changes can be expressed probabilistically as a transition density [95]

$$P(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}, a_{k-1}) \tag{6.2}$$

This probability density gives probability that if at time step $k-1$ the robot was at location $\boldsymbol{x}_{k-1}$ and performed action a_{k-1}, then it ended up at location $\boldsymbol{x}_k$ at time step k. In other words, the transition density describes how the actions of the robot change its location. This density is therefore called the *action* or *motion model* [58]. In practice one can roughly approximate this motion model from from the kinematics and dynamics of the robot. Another option is to have the robot learn the model itself [94].

Sensing. Let S be the space of all possible measurements coming from a particular sensor, and let s_k denote an element in S observed at time k, such that $s_k \in S$ [96] [94]. The

probability that a sensor observes s_k from a certain location $\boldsymbol{x}_k$ at time k is thus described by

$$P(s_k|\boldsymbol{x}_k) \tag{6.3}$$

This is called the *sensor* or *perceptual model* [58]. Unlike the motion model of the robot, the perceptual model can be difficult to compute. The reason for this is sometimes high dimensionality of the measurements.

6.3.2 Localization Formula

In Subsection 6.3.1 we described the Bayesian approach to state estimation for robots such as the AUV. We now know that at every time step, the robot performs an action that ends at the next time step. This action changes the location of the robot according to the motion model from Equation 6.2. Besides this, the robot can also incorporate information from sensing its environment which is distributed according to the probability distribution from the perceptual model in Equation 6.3. In the following three subsections we will describe how to represent the initial belief probabilistically, and then update our beliefs on the premises of a Bayesian framework.

Initial Belief

Before the robot starts acting in the environment it has an *initial belief* of where it is and how it is oriented. This belief is model by prior belief at time step 0, $Bel^-(x_0)$. If the robot knows where it is initially, then $Bel^-(x_0)$ is a distribution with a peak at the location where the robot knows it is. The goal of the localization to compensate for currents, waves, drift and possible noise sources to keep track of the AUV location.

Updating Beliefs

Starting with the initial belief the robots start querying its sensors and performing actions in the environment. Each measurement and vehicle action has to be incorporated into the belief of robot to give it the most up-to-date state estimate.

The belief the robot has after it has incorporated the action a_{k-1} executed at step $k-1$, and before it gets a new measurement y_k, is the prior belief

$$Bel^-(\boldsymbol{x}_k) = P(\boldsymbol{x}_k|y_1, a_1, y_2, a_2, ..., y_{k-1}, a_{k-1}) \tag{6.4}$$

once it has received an absolute measurement y_k at step k, it incorporates this measurement to obtain the posterior belief

$$Bel^{+}(\boldsymbol{x}_k) = P(\boldsymbol{x}_k|y_1, a_1, y_2, a_2, ..., y_{k-1}, a_{k-1}, y_k) \tag{6.5}$$

The question now is - knowing that the computation of these probability densities would have to happen 20-40 times per second - how can it be done in the most efficient way possible?

Incorporating Acting

Assume that robot has performed an action and wants to include the relative position measurement monitoring this recent action into its belief. In Equation 6.4 we defined the belief in which the latest action information is incorporated, the prior belief $Bel^{-}(\boldsymbol{x}_k)$. By the means of *theorem of total probability* and then use the *Markov property* to rewrite the original definition into a computationally efficient formula [58]. The theorem of total probability states that the probability of an outcome is equal to the sum of the probabilities of each of its dependent, partial, outcomes. Using this theorem, the definition of the prior belief in Equation 6.4 become

$$\begin{aligned} Bel^{-}(\boldsymbol{x}_k) = \int_D & P(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}, y_1, a_1, ..., y_{k-1}, a_{k-1}) \\ & \times P(x_{k-1}|y_1, a_1, ..., y_{k-1}, a_{k-1}) d\boldsymbol{x}_{k-1} \end{aligned} \tag{6.6}$$

This equation expresses that the prior belief of being in state $\boldsymbol{x}_k$ is the sum of the probabilities of coming from state $\boldsymbol{x}_{k-1}$ to state $\boldsymbol{x}_k$ given all earlier actions and measurements multiplied with the probability of being at location $\boldsymbol{x}_{k-1}$ given all information up to step $k-1$. However, we are not concerned with the action in step $k-1$, a_{k-1}, only the physical location itself. Thus, Equation 6.6 can be rewritten as

$$\begin{aligned} Bel^{-}(\boldsymbol{x}_k) = \int_D & P(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}, y_1, a_1, ..., y_{k-1}, a_{k-1}) \\ & \times Bel^{+}(x_{k-1}) d\boldsymbol{x}_{k-1} \end{aligned} \tag{6.7}$$

For the sake of simplicity and computational efficiency, the Bayesian framework applies the *Markov assumption*, for which we only consider the current state. With the knowledge of the previous location x_{k-1}, it is no importance how the robot ended up at that location or what it sensed. With that, the probability density simplifies to

$$Bel^{-}(\boldsymbol{x}_k) = \int_D P(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}, a_{k-1}) \times Bel^{+}(x_{k-1}) d\boldsymbol{x}_{k-1} \tag{6.8}$$

That is, the *multivariate cumulative normal distribution* function or the result of integrating over all last locations x_{k-1}, the probability that the performed action a_{k-1} brought the robot from location x_{k-1} to x_k times the posterior belief that the robot had it being at x_{k-1} at the last time step [58].

Incorporating Sensing

We want to incorporate the collected measurements y_k into the prior belief $Bel^-(x_k)$ to form a posterior belief as we defined Equation 6.5. With *Bayes' theorem* and *Markov assumption* we can rewrite this posterior belief into a computationally efficient form.

In the Bayesian interpretation, probability measures a "degree of belief". Bayes' theorem [56] then links the degree of belief in a proposition before and after accounting for evidence. For a proposition A and evidence B, then $P(A|B)$; is the conditional posterior probability of event A occurring given that B is true. The Bayes' theorem is stated mathematically as

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{6.9}$$

With $P(A)$, the *prior*, is the initial degree of belief in A. $P(A|B)$, the *posterior* is the degree of belief having accounted for B. Using the Bayes' theorem in Equation 6.9 we can rewrite the definition of the posterior belief in Equation 6.5,

$$Bel^+(\boldsymbol{x}_k) = \frac{P(y_k|\boldsymbol{x}_k, y_1, a_1, ..., y_{k-1}, a_{k-1})Bel^-(\boldsymbol{x}_k)}{P(y_k|y_1, a_1, ..., y_{k-1}, a_{k-1})} \tag{6.10}$$

that is, the conditional probability of observing y_k, $P(y_k|\boldsymbol{x}_k, y_1, ..., a_{k-1})$, times the prior belief of being in state $\boldsymbol{x}_k$, $Bel^-(x_k)$, divided by the probability of observing measurement y_k conditioned on all information so far, $P(y_k|y_1, ..., a_{k-1})$.

To make the computations easier, we again make the Markov assumption. That is, the probability of reading a sensor measurement is independent of the actions and observations that were made before the AUV arrived in its current state. By substituting this into Equation 6.10, we obtain

$$Bel^+(\boldsymbol{x}_k) = \frac{P(y_k|\boldsymbol{x}_k)Bel^-(\boldsymbol{x}_k)}{P(y_k|y_1, a_1, ..., y_{k-1}, a_{k-1})} \tag{6.11}$$

The denominator of the equation is a normalizing constant ensuring that the probability density integrates to 1. This constant is calculated by integrating the numerator over all possible locations $\boldsymbol{x}_k$ [56] [94]

$$P(y_k|y_1, a_1, ..., y_{k-1}, a_{k-1}) = \int_D P(y_k|x_k)Bel^-(\boldsymbol{x}_k)d\boldsymbol{x}_k \tag{6.12}$$

A Summarized Localization Formula

By combining the derived results into a single localization formula that recursively computes the posterior belief in the location of a robot taking into account sensing and action

information. Substituting the poserior belief in Equation 6.8 into Equation 6.11, the posterior belief becomes

$$\begin{aligned} Bel^{+}(\boldsymbol{x}_k) &= \frac{P(y_k|\boldsymbol{x}_k)Bel^{-}(\boldsymbol{x}_k)}{P(y_k|y_1,...,a_{k-1})} \\ &= \frac{P(y_k|\boldsymbol{x}_k)\int_D P(\boldsymbol{x}_k|\boldsymbol{x}_{k-1},a_{k-1}) \times Bel^{+}(\boldsymbol{x}_{k-1})d\boldsymbol{x}_{k-1}}{\int_D P(y_k|x_k)Bel^{-}(\boldsymbol{x}_k)d\boldsymbol{x}_k} \end{aligned} \tag{6.13}$$

To implement the derived formula in Equation 6.13, one need to specify the three probability densities: the *action model*, $P(\boldsymbol{x}_k|\boldsymbol{x}_{k-1},a_{k-1})$, the *perceptual model*, $P(y_k|\boldsymbol{x}_k)$ and the *initial belief*, $Bel^{-}(x_0)$.

6.3.3 Kalman Filters

As discussed in subsection 6.3.1 and 6.3.2, the representation of the belief has a significant effect on the computational efficiency in calculating the belief. One way to deal with the computational complexity of beliefs over continuous spaces is by representing the belief as a parametrized continuous function. In this subsection the investigation on how Kalman Filters work and how they use Gaussian functions to update beliefs will be presented.

Filter Applications and Drawbacks

A *Kalman Filter* (KF) is a *state estimator* that works on the *prediction-correction* basis [8]. This means that it computes a belief in a certain state estimate by first making a prediction (prior belief) based on the dynamics of the system and later correcting this prediction (posterior belief) using measurements of the system. KFs assume that the action and sensor models are subject to *Gaussian noise*. In practice, this might not always be the case, but it does allow the KF to efficiently make its calculations [96]. The fact that the variables of the state might be noisy and not directly observable makes the state estimation difficult. To estimate the state a KF has access to measurements of the system. Those measurements are *linearly* related to the state and are corrupted by noise. If the *Gaussian assumption* holds, then the KF estimator is an optimal state estimator for virtually any kind of meaningful criterion of optimality [33].

There is a disadvantage of choosing the Gaussian representation for the belief. Choosing this representation is a restrictive way of representing the location space. A Gaussian function is a *uni-modal* density, i.e, it has only one peak. Thus, it does not allow multiple ideas about locations. There is only one best location estimate, corresponding to the mean of the belief. Therefore, representing the state space as a Gaussian distribution can only be done when the initial location of the robot is known [58].

Noise Characteristics

As mentioned earlier, the system and sensors are subject to noise. The KF assumes that the system noise denoted $\boldsymbol{w}_k$ and the measurement noise denoted $\boldsymbol{v}_k$ are random variables that are *independent*, *white*, *zero-mean Gaussian* probability distributions. Besides this, the KF assumes that the initial state of the system $\boldsymbol{x}_k$ at time $k = 0$ is independent and Gaussian distributed.

The *independence* assumption makes it so that the noise in the system and measurements are independent. For example, the noise intensities in camera images does not influence noise in the guidance system of a robot.

The *white noise* assumption also greatly simplifies the mathematics involved in the filter. White noise is the noise that has power at all frequencies in the spectrum and that is completely uncorrelated with itself at any time except the present [47]. This assumption implies that errors are not correlated through time.

The *zero-mean* assumption implies that the errors in the system and measurements are random. Noise can be classified into *systematic* noise and *non-systematic* or *random* noise [91]. Systematic noise is noise that constantly corrupts the system state or measurements in a certain way. It is *biased* noise, often caused by inaccurate parameters. For example, if the diameter of the wheels of a guidance system is not accurately determined, there will be a systematic error in the position estimation. Random noise is noise that is not systematic in that way. Random noise is sometimes positive, sometimes negative, but, in the case of zero-mean, on average zero [58].

The *Gaussian* assumption deals with the amplitude of the noise. It states that the amount of noise involved can be modeled as a bell-shaped curve. Gaussians are fully characterized by their mean and variance and therefore capture all available noise information. With the zero-mean and Gaussian distribution assumptions for the noises we can write down how they are distributed. If we let $Q_k = E[(\boldsymbol{w}_k)(\boldsymbol{w}_k)^T]$ be the *process noise covariance* at time step k and $R_k = E[(\boldsymbol{v}_k)(\boldsymbol{v}_k)^T]$ be the *measurement noise covariance* at time step k, we can express the system noise w_k and the measurement noise $\boldsymbol{v}_k$ as

$$\boldsymbol{w}_k = \mathcal{N}(0, Q_k) \tag{6.14}$$

$$\boldsymbol{v}_k = \mathcal{N}(0, R_k) \tag{6.15}$$

Where $\mathcal{N}(\mu, \Sigma)$ denotes the Gaussian function with mean μ and covariance Σ. The main diagional of the covariance matrices Q_k and R_k contains the variance in the state and measurement vector variables respectively. The off-diagonal elements are zero, since we assume that the noises are independent.

Extending the Linear Assumption of Kalman Filters

The *Linear Kalman Filter* (LKF) assumes that the system state and measurements can be described as a *linear dynamic system* [33]. This is a set of linear equations that models

the evolution of the state of the system over time and describes how the measurements are related to the state. A motivation for assuming a linear model is that it simplifies the computations and since often a linear approach is adequate for the problem to be modeled. When the problem is not linear, we can then use linearizing techniques to transform a non-linear problem into a linear one. That is exactly the idea of the *Extended Kalman Filter* (EKF) [47] [33], originally called *Kalman-Schmidt Filter* [47]. The most recent state estimates are used to update the nominal trajectory; that being a particular solution of a noisy system in which the random variables take on their expected values. If the system state is sufficiently observable, that is, if the measurements provide information about the state at a high enough frequency then the deviations between estimated trajectory (solution) and the actual trajectory will stay small [58].

Unlike its linear counterpart, the EKF in general is not an optimal estimator. If the initial estimate of the state is wrong, or if the process is modeled incorrectly, the filter may quickly diverge. Having stated this, the EKF can give reasonable performance, and is arguably the standard of implementation in navigation systems and GPS.

6.3.4 Extended Kalman Filter Algorithm

Nonlinear system model. The goal of the Extended Kalman Filter (EKF) is to estimate the full 3D (6DOF) pose and velocity of a moving robot over time. The process can be described as a nonlinear dynamic system [52], with

$$\boldsymbol{x}_k = \boldsymbol{f}(\boldsymbol{x}_{k-1}) + \boldsymbol{w}_{k-1} \tag{6.16}$$

where $\boldsymbol{x}_k$ is the robot's system state (i.e, 3D pose) at time k, $\boldsymbol{f}(\boldsymbol{x}_{k-1})$ is a nonlinear state transition function relating the state of the previous time step to the current state, and $\boldsymbol{w}_{k-1}$ is the process noise corrupting the system. The noise is assumed to be independent, white, zero-mean, and Gaussian distributed [58]. A 17-dimensional state vector, $\boldsymbol{x}$, comprises the vehicle's 3D pose, 3D orientation (attitude), and their respective velocities. The rotational values are expressed in unit quaternions.

Nonlinear measurement model. Additionally, it is no longer assumed that the measurements are governed by a linear equation. Instead, measurements are received of the form

$$\boldsymbol{y}_k = \boldsymbol{H}_k \boldsymbol{x}_k + \boldsymbol{v}_k \tag{6.17}$$

where $\boldsymbol{y}_k$ is a measurement at time k, $\boldsymbol{H}_k$ is a observation matrix that maps the state into measurement space, and $\boldsymbol{v}_k$ is the noise corrupting the measurement. This noise is also assumed to be independent, white, zero-mean and Gaussian distributed [58].

The observation matrix is computed as

$$\boldsymbol{H}_k = \left.\frac{\partial \boldsymbol{h}(\boldsymbol{x})}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\hat{\boldsymbol{x}}_k^-} \tag{6.18}$$

Initialization step. The EKF is initialized with the posterior state estimate $\hat{\boldsymbol{x}}_0^+$ and uncertainty $\boldsymbol{P}_0^+$ at step $k = 0$.

$$\hat{\boldsymbol{x}}_k^- = \boldsymbol{x}_0 \tag{6.19}$$

$$\boldsymbol{P}_k^- = \boldsymbol{P}_0 \tag{6.20}$$

Prediction step. At every time step, the EKF carries out a prediction step that projects the current state estimate and error covariance forward in time

$$\hat{\boldsymbol{x}}_k^- = \boldsymbol{f}(\hat{\boldsymbol{x}}_{k-1}^+) \tag{6.21}$$

$$\boldsymbol{P}_k^- = \boldsymbol{F}_k \hat{\boldsymbol{P}}_{k-1} \mathbf{F}_k^T + \boldsymbol{Q}_{k-1} \tag{6.22}$$

For an AUV, $\boldsymbol{f}$ is a standard 3D kinematic model derived from Newtonian mechanics. The estimate error covariance, $\hat{\boldsymbol{P}}_{k-1}$ from the previous time step, is projected via $\boldsymbol{F}_k$, the Jacobian matrix of $\boldsymbol{f}$, and then perturbed by $\boldsymbol{Q_{k-1}}$ the process noise covariance from the previous time step.

The Jacobian matrix is computed as

$$\boldsymbol{F}_k = \left.\frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\hat{\boldsymbol{x}}_{k-1}^+} \tag{6.23}$$

Correction step. The EKF corrects the prior state estimate with a full measurement $\mathbf{z}_k$ by means of the correction equations

$$\boldsymbol{K}_k = \boldsymbol{P}_k^- \boldsymbol{H}_k^T (\boldsymbol{H}_k \boldsymbol{P}_k^- \boldsymbol{H}_k^T + \boldsymbol{R}_k)^{-1} \tag{6.24}$$

$$\hat{\boldsymbol{x}}_k^+ = \hat{\boldsymbol{x}}_k^- + \boldsymbol{K}_k (\boldsymbol{y}_k - \boldsymbol{H}_k \boldsymbol{x}_k^-) \tag{6.25}$$

$$\boldsymbol{P}_k^+ = (\boldsymbol{I} - \boldsymbol{K}_k \boldsymbol{H}_k) \boldsymbol{P}_k^- (\boldsymbol{I} - \boldsymbol{K}_k \boldsymbol{H}_k)^T + \boldsymbol{K} \boldsymbol{R}_k \boldsymbol{K}_k^T \tag{6.26}$$

The Kalman gain, $\boldsymbol{K}_k$ is calculated using the observation matrix, $\boldsymbol{H}_k$, the measurement covariance, $\boldsymbol{R}_k$, and $\boldsymbol{P}_k^-$. The gain is used to update the state vector and covariance. In Equation 6.26 the Joseph form covariance update equation [52] to promote filter stability by ensuring that $\boldsymbol{P}_k$ remains positive semi-definite.

6.4 *Case Study: Vehicle State Estimation for the Vortex NTNU, Manta AUV, using Extended Kalman Filter Design*

For this case study we'll be using our understanding of the Bayesian estimation problem and our knowledge of the extended Kalman filter to put in place a vehicle state estimator for the Manta AUV.

6.4.1 Prediction and Correction Models

The EKF models follow a standard predict/correct cycle. During prediction, if there is no acceleration reference, the velocity at timestep $k+1$ is simply predicted to be the same as the velocity at timestep k. During correction, this predicted value is fused with the measured value to produce the new velocity estimate. This can be problematic, as the final velocity will effectively be a weighted average of the old velocity and the new one. When this velocity is the integrated into a new pose, the result can be sluggish convergence. This might be the case for prediction of the angular velocities, in which we have no sensors measuring the angular acceleration measurements and can cause some slight drift over time depending on quality of the hardware.

Nonlinear System Model

The nonlinear system model is expressed as a generic omnidirectional motion model derived from Newtonian mechanics. Each variable has a non-linear transition function, and we encode those transition functions into a nominal state vector. Naturally, this will not match every robot perfectly, but it works pretty well for a broad range of robots. You can always increase the process noise covariance matrix, $\boldsymbol{Q}_k$ if the model does not match your robot.

The nominal state vector of the nonlinear system $\boldsymbol{x}_k^-$ at time k is written as:

$$
\begin{aligned}
\hat{\boldsymbol{x}}_k^- &= \boldsymbol{f}(\hat{\boldsymbol{x}}_{k-1}^+) \\
&= [\hat{\boldsymbol{p}}_k^-, \hat{\boldsymbol{q}}_k^-, \hat{\boldsymbol{v}}_k^-, \hat{\boldsymbol{\omega}}_k^-, \hat{\boldsymbol{a}}_k^-]^T \\
&= \begin{bmatrix}
\hat{\boldsymbol{p}}_{k-1}^+ + \boldsymbol{R}(\hat{\boldsymbol{q}}_{k-1}^+)\hat{\boldsymbol{v}}_{k-1}^+\Delta t + \frac{1}{2}\boldsymbol{R}(\hat{\boldsymbol{q}}_{k-1}^+)\hat{\boldsymbol{a}}_{k-1}^+\Delta t^2 \\
\hat{\boldsymbol{q}}_{k-1}^+ + \boldsymbol{T}(\hat{\boldsymbol{q}}_{k-1}^+)\hat{\boldsymbol{\omega}}_{k-1}^+\Delta t \\
\hat{\boldsymbol{v}}_{k-1}^+ + \hat{\boldsymbol{a}}_{k-1}^+\Delta t \\
\hat{\boldsymbol{\omega}}_{k-1}^+ \\
\boldsymbol{a}_{k-1}^+ - \boldsymbol{R}(\hat{\boldsymbol{q}}_{k-1}^+)\boldsymbol{g}_{k-1}^b
\end{bmatrix} \\
&= [\hat{\boldsymbol{p}}_{(b/n)_k}^n, \hat{\boldsymbol{q}}_{(nb)_k}, \hat{\boldsymbol{v}}_{(b/n)_k}^b, \hat{\boldsymbol{\omega}}_{(b/n)_k}^b, \hat{\boldsymbol{a}}_{(b/n)_k}^b]^T
\end{aligned}
\tag{6.27}
$$

where, $\hat{\boldsymbol{p}}^n_{(b/n)_k}$ is the prior estimate distance from world-fixed inertial frame NED {n} to BODY {b} expressed in NED {n} at time step k. $\hat{\boldsymbol{q}}_{(nb)_k}$ is the prior estimate unit quaternions between {n} and {b} at time step k. $\hat{\boldsymbol{v}}^b_{(b/n)_k}$ is the prior estimate linear velocity of {b} with respect to {n} in {b} at time step k. $\hat{\boldsymbol{\omega}}^b_{(b/n)_k}$ is the prior estimate angular velocity of {b} with respect to {n} expressed in {b} at time step k, and finally $\hat{\boldsymbol{a}}^b_{(b/n)_k}$ is the prior estimate linear acceleration of {b} with respect to {n} expressed in {b} at time step k.

Nonlinear Measurement Model

The inertial sensors are mounted onboard the vehicle in a body-fixed coordinate system {m} located at $\boldsymbol{r}^b_m = \boldsymbol{r}^b_{m/b} = [x_m, y_m, z_m]^T$ with respect to the {b}-frame coordinate origin CO. This is referred to as a strapdown system because the sensors are strapped to the craft and a lightweight digital computer is used to perform computations [27]. There are three strapdown systems that we need to consider; the IMU, DVL and the pressure sensor.

IMU:

The IMU is mounted close to the origin CO so no static translation is needed. However, the IMU is rotated 180 degrees and thus the measurements need a static rotation $\boldsymbol{q}_{bm}$ denoting the rotation between {b} and {m}, such that

$$\boldsymbol{a}^b_{imu} = \boldsymbol{R}^b_m(\boldsymbol{q}_{bm})\boldsymbol{a}^m_{imu} \tag{6.28}$$
$$\boldsymbol{\omega}^b_{imu} = \boldsymbol{R}^b_m(\boldsymbol{q}_{bm})\boldsymbol{\omega}^m_{imu} \tag{6.29}$$

Here we have that,

$$\boldsymbol{a}^m_{imu} = \boldsymbol{R}^m_b(\boldsymbol{q}_{bm})\boldsymbol{R}^b_n(\boldsymbol{q}_{bn})(\dot{\boldsymbol{v}}^n_{m/n} - \boldsymbol{g}^n) + \boldsymbol{b}^m_{acc} + \boldsymbol{w}^m_{acc} \tag{6.30}$$
$$\boldsymbol{\omega}^m_{imu} = \boldsymbol{R}^m_b(\boldsymbol{q}_{bm})\boldsymbol{\omega}^b_{m/n} + \boldsymbol{b}^m_{gyro} + \boldsymbol{w}^m_{gyro} \tag{6.31}$$

Where the gravity of the earth in $\boldsymbol{g}^n$ {n}-frame is subtracted from the accelerometer measurement. The accelerometer and gyro biases are denoted as $\boldsymbol{b}^m_{acc}, \boldsymbol{b}^m_{gyro}$. Additive zero-mean sensor measurement noises are modeled by $\boldsymbol{w}^m_{acc}, \boldsymbol{w}^m_{gyro}$

DVL:

The DVL has both static rotation and translation relative to {b}-frame that need to be accounted for as seen by Figure 6.3. The velocity measurements can be written as follows

$$\boldsymbol{v}^b_{dvl} = \boldsymbol{R}^b_m(\boldsymbol{q}_{bm})\boldsymbol{v}^m_{dvl} \tag{6.32}$$

Here we have that,

$$\boldsymbol{v}_{dvl}^{m} = \boldsymbol{R}_{b}^{m}(\boldsymbol{q}_{bm})\boldsymbol{R}_{n}^{b}(\boldsymbol{q}_{bn})\underbrace{(\boldsymbol{v}_{dvl}^{n} - \boldsymbol{R}_{b}^{n}(\boldsymbol{q}_{nb})\boldsymbol{S}(\boldsymbol{\omega}_{b/n}^{b})\boldsymbol{r}_{dvl}^{b})}_{\boldsymbol{v}_{m/n}^{n}} + \boldsymbol{w}_{dvl}^{m} \tag{6.33}$$

where, $\boldsymbol{S}(\boldsymbol{\omega}_{b/n}^{b})$ is the skew symmetric matrix and $\boldsymbol{r}_{dvl}^{b}$ is the lever arm from the {dvl}-frame to the {b}-frame. $\boldsymbol{R}_{b}^{n}(\boldsymbol{q}_{nb})\boldsymbol{S}(\boldsymbol{\omega}_{b/n}^{b})\boldsymbol{r}_{dvl}^{b}$ is the velocity contribution from the gyro measurements that need to be subtracted from the DVL measurements to achieve linear velocity $\boldsymbol{v}_{m/n}^{n}$ of the vehicle [27, Fossen, p. 340].

Pressure Sensor:

The pressure sensor is used to depth relative to world-fixed inertial NED frame at the water surface. The gauge pressure at a certain depth can be expressed as:

$$p_{pg} = p_{atm} + \rho g z_{pg/n}^{n} + w_{pg} \tag{6.34}$$

where p_{atm} is the atmospheric pressure at the water surface, ρ is the water density and w is the measurement noise. Re-arranging the equation and taking into account the orientation and strapdown reference frame of the sensor, we get can express the vehicle depth as:

$$z_{pg/n}^{n} = z_{m/n}^{n} - [0,0,1]\boldsymbol{R}_{b}^{n}(\boldsymbol{q}_{nb})\boldsymbol{r}_{pg/n}^{b} \tag{6.35}$$

where $\boldsymbol{p}_{pg/n}^{n} = [0,0,z_{pg/n}^{n}]$

6.4.2 Initial Estimate and Process Noise Covariance

The *process noise covariance* matrix, $\boldsymbol{Q}_k$, can be difficult to tune, and can vary for each application, so it is exposed as a configuration parameter. This matrix represents the noise we add to the total error after each prediction step. The better the omnidirectional motion model matches your system, the smaller these values can be. However, if one find that a given variable is slow to converge, one approach is to increase the *process noise covariance* diagonal value for the variable in question, which will cause the filter's predicted error to be larger, and thus the filter will trust the incoming measurement more during correction. The initial process noise covariance, $\boldsymbol{Q}_0$, for our application can be found in Appendix B.1.

The $\boldsymbol{P}_k$ represents the value for the *state estimate error covariance* matrix. Setting a diagonal value (variance) to a large value will result in rapid convergence for initial measurements of the variable in question. One should be careful not to use large values for variables that will not be measured directly. The initial state estimate error covariance, $\boldsymbol{P}_0$, for our application can be found in Appendix B.1.

6.4.3 Software Implementation

The the link to the source code for this case study can be found in Appendix A

Open-Source Packages, Libraries and Standards

For this particular case study we will be using a software package, *robot_localization* ROS [53]. The package currently contains an implementation of an extended Kalman filter (EKF). It can support an unlimited number of inputs from multiple sensor types, and allows users to customize which sensor data fields are fused with the current state estimate. The package is generally made for wheeled, non-holonomic robot that works in a planar environment; however, adaptations were made in the configuration files and in the device driver of the Doppler Velocity Log (DVL) to extend it's use for AUVs in 3D space.

For adherence of ROS standards on coordinate frames, we'll be using the *tf2* standard transform library for ROS. tf2 is the second generation of the standard ROS transform library, which lets the user keep track of multiple coordinate frames over time. tf2 maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time.

Dynamic Transformations

A robotic system typically has many 3D coordinate frames that change over time (dynamic), such as a world frame, base frame, gripper frame, camera frame, etc. tf2 keeps track of all these frames over time, and allows you to ask questions like:

- Where was the camera frame relative to the world frame, 5 seconds ago?
- What is the pose of the object in my gripper relative to my base?
- What is the current pose of the base frame in the map frame?

tf2 can operate in a distributed system. This means all the information about the coordinate frames of a robot is available to all ROS components on any computer in the system. The EKF is concerned with our principal coordinate frames and the dynamic transform between them: *base_link*, *odom*, *map*, and *earth*. *base_link* is the coordinate frame that is affixed to the robot. Both odom and map are world-fixed frames. The robot's position in the odom frame will drift over time, but is accurate in the short term and should be continuous. The odom frame is therefore the best frame for executing local motion plans. The map frame, like the odom frame, is a world-fixed coordinate frame, and while it contains the most globally accurate position estimate for your robot, it is subject to discrete jumps, e.g., due to the fusion of GPS data or a correction from a map-based localization node. The earth frame is used to relate multiple map frames by giving them a common reference frame. The EKF for our use are not concerned with the earth frame, nor a map frame since there is no prior knowledge of the surroundings. For our purpose we'll

be using the odom frame that is the world-fixed frame whose origin is aligned with the robot's start position as seen from Figure 6.3.

Static Transformations

The goal of static transforms was to remove the need for re-communicating things that don't change. The ability to update the values was implemented in case they are subject to uncertainty and might be re-estimated later with improved values. But importantly those updated values are expected to be true at all times. For the Manta AUV static transforms will typically apply to sensors and actuators that are strapped to the body of the vehicle, where there relative translation and rotation to the *base_link* (body frame) does not change over time. The EKF needs to perform three static transforms from body frame to sensor frame in order to make the state estimation work. This is the static transform from *base_link* to *dvl_link*, from *base_link* to *imu_link* and from *base_link* to *pressure_link*. The corresponding translation and rotation can be seen from Figure 6.3.

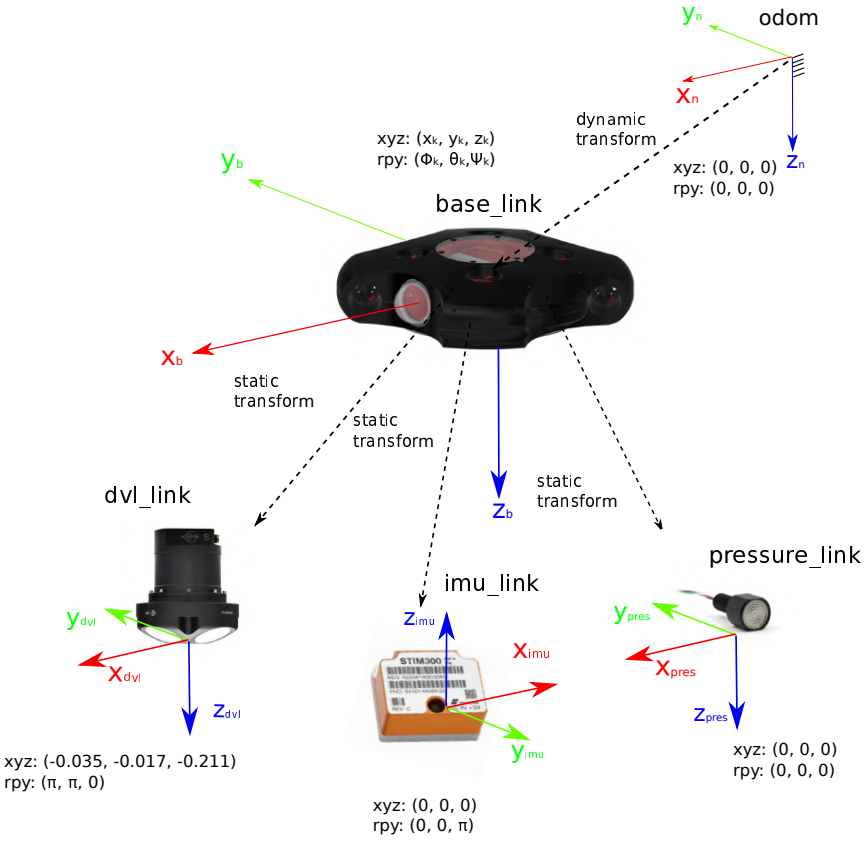


Figure 6.3: Static and dynamic transformations of the Manta AUV sensor stack.

Message Types

The state estimation nodes of *robot_localization* produce a state estimate whose pose is given in the map or odom frame and whose velocity is given in the *base_link* frame. All incoming data is transformed into one of these coordinate frames before being fused with the state. The data in each message type is transformed as follows:

nav_msgs/Odometry - All pose data (position and orientation) is transformed from the message header's *frame_id* into the coordinate frame specified by the world frame parameter (in this case odom). In the message itself, this specifically refers to everything contained within the pose property. All twist data (linear and angular velocity) is transformed from the *child_frame_id* of the message into the coordinate frame specified by the *base_link_frame* parameter (typically *base_link*).

geometry_msgs/PoseWithCovarianceStamped - Handled in the same fashion as the pose data in the Odometry message.

geometry_msgs/TwistWithCovarianceStamped - Handled in the same fashion as the twist data in the Odometry message.

sensor_msgs/Imu - The IMU message is currently subject to some ambiguity, though this is being addressed by the ROS community. Most IMUs natively report orientation data in a world-fixed frame whose X and Z axes are defined by the vectors pointing to magnetic north and the center of the earth, respectively, with the Y axis facing east (90 degrees offset from the magnetic north vector). This frame in which we call NED (North, East, Down) as mentioned earlier. However, REP-103 [1] specifies an ENU (East, North, Up) coordinate frame for outdoor navigation. As of this writing, robot_localization assumes an ENU frame for all IMU data, and does not work with NED frame data. This may change in the future, but for now, users should ensure that data is transformed to the ENU frame before using it with any node in robot_localization [51].

6.4.4 Simulation Results and Discussion

The simulation is performed in the Gazebo simulation environment created in the case study from Chapter 5. To test the performance of the extended Kalman filter we track the results in comparison to the ground truth pose and velocity estimates provided by the UUV simulator in Gazebo.

In Figure 6.4 you can see the results of the Manta AUV performing a diving spiral move with a radius of $0.5m$. From the estimated position plot (in red) we see that there is barely any difference from the ground truth position (in green). We also see that over the approximately ten minutes that it took to record the run, there is almost no accumulated

[1] https://www.ros.org/reps/rep-0103.html

bias. This is largely due to the fact the applied Gaussian white noise are set low. Once you have some real sensor recordings from an actual test run, one can measure the Gaussian noise mean and standard deviation, and then apply that to the simulated sensors in Gazebo.

P.S. All the figures and plots have vector resolution.

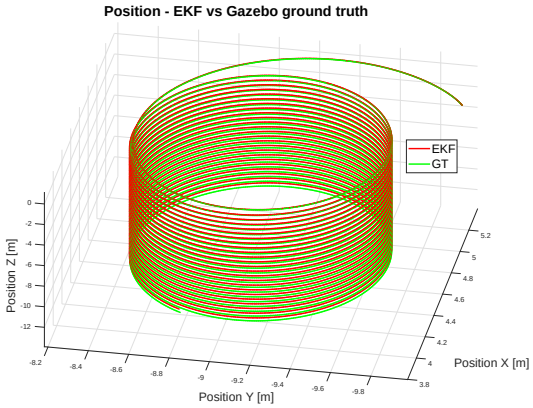


Figure 6.4: A 3D position plot comparing EKF with ground truth. Scan the QR code to see a live demo of the recorded run.

On the left in Figure 6.5 you'll see the position - both EKF and ground truth - plotted against time. As mentioned earlier, there is barely any difference between the two, which confirms the high accuracy of the EKF. On the right side in Figure 6.5, you can see the attitude - both EKF and ground truth - plotted against time. As you can see from the two plots, there is less to none estimation error. The wrapping of the yaw angle about -180 to 180 degrees is due to the constant turning of the vehicle.

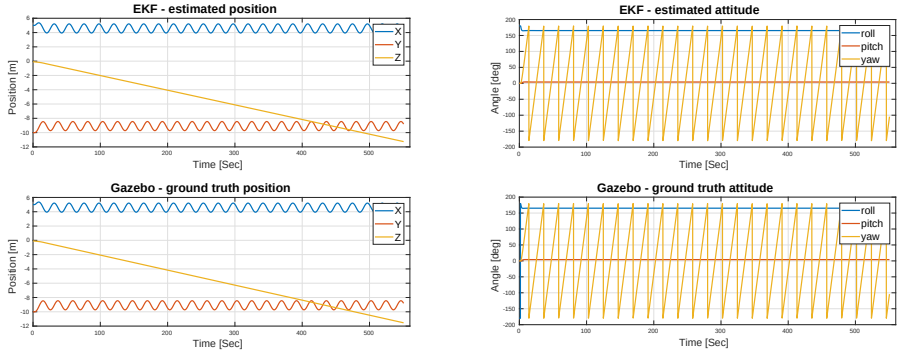


Figure 6.5: Estimated vs ground truth position and attitude

The plots in Figure 6.6 can be a little confusing. As you can see, the two bottom plots on the left and right are showing ground truth angular and linear velocities with respect to the **world-fixed frame**, while the upper two plots on the left and right are showing estimated angular and linear velocities with respect to a **body-fixed frame**. This is a weakness with the UUV Simulator, because the ground truth velocities should also be computed in body-fixed frame which makes the most sense.

However, even though the bottom and top plots are done in different coordinate frames, they are fundamentally the same. In Figure 6.6, you can see that the ground truth angular velocity in yaw is $16[deg/s]$ and this is also what the EKF estimates show. By looking at the estimated linear velocity in X-direction (surge) in the top right plot, you can see that the vehicle has a constant velocity in body-fixed frame with respect to world-fixed frame at $0.17[m/s]$. By looking at the ground truth linear velocity in world-fixed frame with respect to world-fixed frame in the bottom right plot, you'll see that the velocity in X-direction (surge) and Y-direction (sway) both oscillate between $-0.17[m/s]$ and $0.17[m/s]$ due to the constant rotation in yaw. This confirms the actual ground truth linear velocity and estimated linear velocities match.

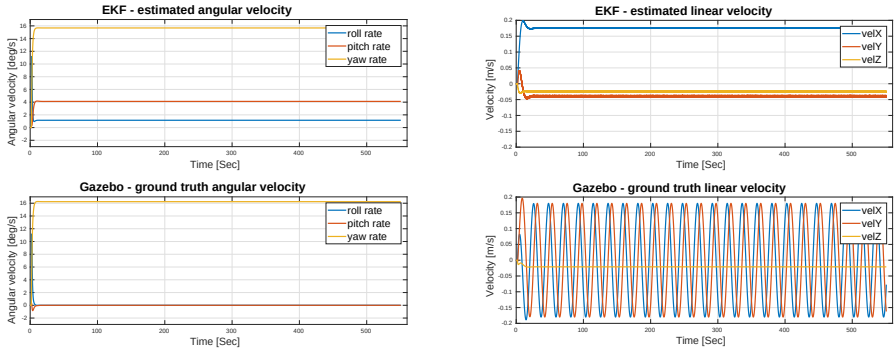


Figure 6.6: Estimated vs ground truth linear and angular velocity

6.4.5 Conclusion of Case Study

The results of this case study suggest the EKF is very precise in its estimates of pose, attitude, linear velocities and angular velocities. The simulated test lasted for 10 minutes and yet there were barely any bias in the estimates compared to the ground truth. The high performance of the EKF is mostly due miniscule noise in the simulated sensors. The noise minuscule in comparison to the noise you will experience in the real world.

the results of the EKF will play a big role in the quality of the controllers implemented later.

6.5 Robot Perception

Robot perception is crucial for a robot to make decisions, plan and operate in real-world environments, by means of numerous functionalities and operations ranging from environment occupancy grid mapping to computer vision. Some examples of robot perception subareas for autonomous vehicles in general, are obstacle detection, object recognition, semantic place classification, 3D environment representation, gesture and voice recognition, activity classification, terrain classification, road detection, vehicle detection, pedestrian detection, object tracking, human detection, and environment change detection [13].

Nowadays, a growing number of robot perception systems are using machine learning (ML) approach, with applications ranging from classical to deep-learning techniques. Machine learning for robot perception can be in the form of supervised classifiers with hand-labeled features, unsupervised learning, or deep-learning neural networks (e.g, convolutional neural networks (CNN)). Based on the sensors implemented, the robot perception task for AUVs can be tackled by using sonars, cameras and various thermal optics, or a sensor fusion with several kinds of devices.

6.5.1 Digital Image Representation and Processing

A digital image is represented through a matrix cell array, where each pixel cell takes a 3-channel RGB (red-green-blue) value, each with brightness intensities between 0 and 255. In this manner a broad array of colors can be produced. The HSV (hue-saturation-value) model is an alternative representation to the RGB colors model, designed to be more representative to the way humans perceive colors. Hue determines the main colors, saturation the intensity and value the brightness. A visualization of the RGB representation is given in Figure 6.7

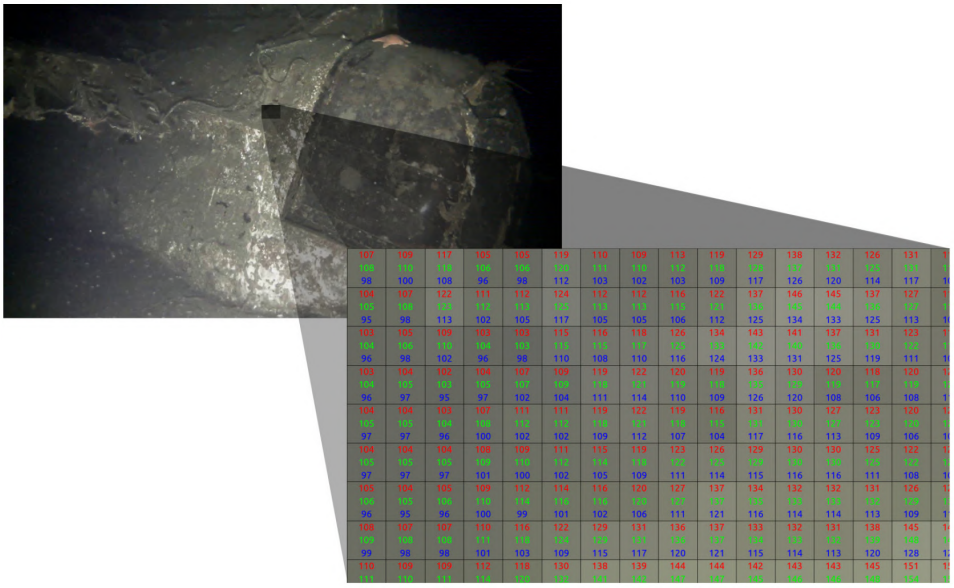


Figure 6.7: A digital image representation. *Courtesy: [40, Kvalberg, 2019]*

High resolution videos introduces a large computational expense, and optimized algorithms for downsizing and cropping the images are therefore a natural part of computer vision. Spatial operations such as convolution kernels are the workhorse of image processing and the kernel, $\boldsymbol{K}$, can be chosen to perform functions such as smoothing, gradient calculation or edge detection. A convolution kernel is a very small matrix and in this matrix, each cell has a number and also an anchor point. The anchor point is used to know the position of the kernel with respect to the image. Convolution is the process in which each element of the image is added to its local neighbors, and then it is weighted by the kernel. It is related to a form of mathematical convolution. For example, consider a convolution kernel which is a square 21×21 matrix containing equal elements:

$$\boldsymbol{K} = \frac{\boldsymbol{I}_{21x21}}{21^2} \tag{6.36}$$

The result of convolving an image with this kernel is an image where each output pixel is the mean of the pixels in a corresponding 21×21 neighborhood in the input image. This results in smoothing, blurring or defocus as seen by Figure 6.8.

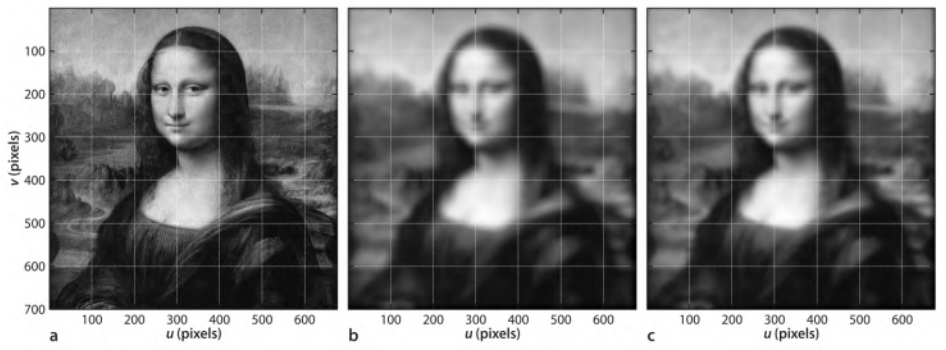


Figure 6.8: Smoothing. a) Original image; b) smoothed with a 21 × 21 averaging kernel; c) smoothed with a 31 × 31 Gaussian $G(\sigma = 5)$ kernel. *Courtesy: [19, Corke, 2016]*

Blurring is a counter-intuitive image processing operation since we typically go to a lot of effort to obtain a clear and crisp image. To deliberately ruin it seems, at face value, somewhat reckless. However as we will see later, Gaussian smoothing turns out to be extremely useful.

6.5.2 Enable Computer Vision in a Robot Application

Computer vision (CV) is a field of study that works on enabling computers to see, identify and process images in the same way that human vision does, and then provide appropriate output. It is a multidisciplinary field that could broadly be called a subfield of artificial intelligence and machine learning, which may involve the use of specialized methods and make use of general learning algorithms [10].

Pinhole Camera Model

The process of image formation, whether it is in a camera or an eye, involves a projection from the 3-D world onto a 2-D surface. Camera models is used to describe the mathematical relationship between 3-D coordinates in space and its projection onto a 2-D image plane [40]. One of the widely used mathematical models in computer vision which describes the perspective transformation is the *pinhole camera model*.

The pinhole camera model is the simplest device that captures accurately the geometry of perspective projection. The model assumes that rays of light travel in straight lines from the object through the pinhole/focal point (depicted as C in Figure 6.9), resulting in inverted image is projected onto the image (sensor) plane at $z = -f$. This mapping from the three dimensions onto two dimensions is called *perspective projection*.

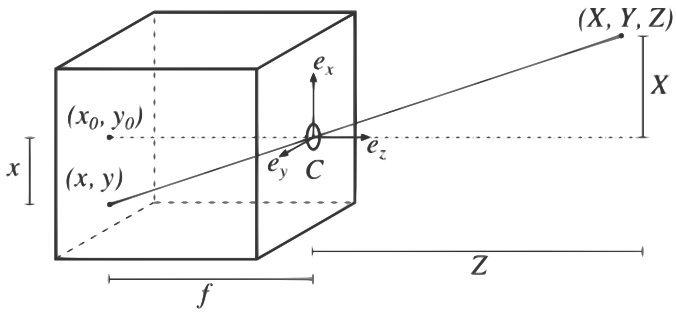


Figure 6.9: Pinhole Camera Model. *Courtesy: [40, Kvalberg, 2019]*

The camera is modelled as a box, where the distance from the focal point C to the image plane is called the *focal length*, f. Geometrically, f can be expressed as follows

$$\frac{x}{f_x} = \frac{X}{Z} \qquad \textit{and} \qquad \frac{y}{f_y} = \frac{Y}{Z} \tag{6.37}$$

we can write the camera projection in general form that maps the 3-D world scene into the image plane:

$$\underbrace{w}_{\text{Scale factor}} \underbrace{\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}}_{\text{Image points}} = \underbrace{\boldsymbol{P}}_{\text{Camera matrix}} \underbrace{\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}}_{\text{World points}} \tag{6.38}$$

The pinhole camera parameters are represented in a 4-by-3 matrix, $\boldsymbol{P}$, called the *camera matrix*:

$$\begin{aligned} \boldsymbol{P} &= \underbrace{\begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{t} \end{bmatrix}}_{\text{Extrinsics}} \underbrace{\boldsymbol{K}}_{\text{Intrinsics}} \\ &= \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{t} \end{bmatrix} \begin{bmatrix} f & sf & c_x \\ 0 & \alpha f & c_y \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{6.39}$$

where f is the focal length, α is the aspect ratio, s is the skew parameter, and c_x and c_y is the principle point coordinates. For most cameras, the skew parameter is close to zero ($s \approx 0$) and the aspect ratio is close to one ($\alpha \approx 1$) [40].

The matrix of intrinsic parameters, $\boldsymbol{K}$, does not depend on the scene viewed. So, once estimated, it can be re-used as long as the focal length is fixed (in case of zoom lens).

The joint rotation-translation matrix $[\boldsymbol{R}, \boldsymbol{t}]^T$ is called a matrix of extrinsic parameters. It is used to describe the camera motion around a static scene, or vice versa, rigid motion of an object in front of a still camera. That is, $[\boldsymbol{R}, \boldsymbol{t}]^T$ translates coordinates of a point $[X, Y, Z]$ to a coordinate system, fixed with respect to the camera.

Camera Calibration

The process of correcting image imperfections that occur due to distorting elements of the camera's interior orientation is called *camera calibration*. In robot applications, we are usually most concerned with the geometric distortions to the image. Real lenses usually have some geometrical distortion, mostly radial distortion and slight tangential distortion [65]. Radial distortions causes straight lines to appear curved, and the distortion becomes larger the farther points are from the centre of the image. As depicted in Figure 6.10, we usually distinguish positive and negative distortions, also known as *barrel distortion* and *pincushion distortion*. Barrel distortion causes the lines at the edge of the image to curve outwards, an occuring effect when magnification decreases with distance from the focal point, C. The opposite effect, when magnification increases with distance from the principle point, causes pincushion distortion, which results in that straight lines curves inward.

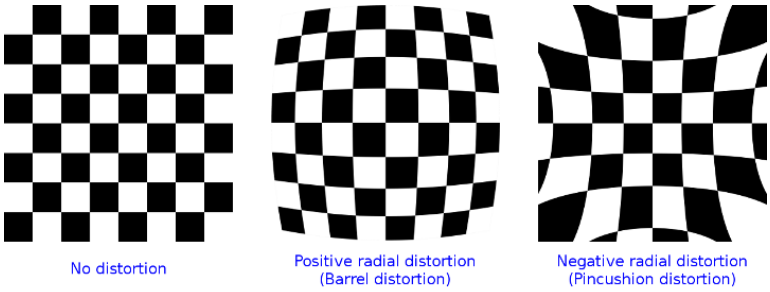


Figure 6.10: Examples of positive and negative radial distortion. *Courtesy: [65, OpenCV Documentation, 2014]*

In the process of calibrating the camera, we try to estimate the the camera parameters. Camera parameters include intrinsics, extrinsics, and distortion coefficients. To estimate the camera parameters, you need to have 3-D world points and their corresponding 2-D image points. You can get these correspondences using multiple images of a calibration pattern, such as a checkerboard as depicted in Figure 6.10.

After you calibrate a camera, to evaluate the accuracy of the estimated parameters, you can use these parameters to correct for lens distortion, measure the size of an object in world units, or determine the location of the camera in the scene [45]. These tasks are used in applications such as computer vision to detect and measure objects. They are also used in robotics, for navigation systems, and 3-D scene reconstruction.

6.5.3 Image Segmentation and Feature Extraction

In the last two subsections we learned that images are simply large arrays of pixel values but for robotic applications images have too much data and not enough information. We need to be able to answer questions such as what is the pose of the object? what type of object is it? how fast is it moving? how fast am I moving? and so on. The answers to such questions are measurements obtained from the image and which we call *image features*. Features are the essence of the scene and the raw material that we need for robot control [19]. Figure 6.11 depicts the traditional approach to computer vision where a raw image enter through a series of image operations that will ultimately extract useful information telling the robot what it is "actually" looking at (in other words, robot vision / computer vision).

Figure 6.11: A traditional approach to computer vision / robot vision

Image Feature Extraction

Where image processing operates on one or more input images and return another image, *feature extraction* operates on an image and return one or more *image features*. Features are typically scalars (for example area or aspect ratio) or short vectors (for example the coordinate of an object or parameters of a line). Image feature extraction is a necessary first step in using image data to control a robot. It is an information concentration step that reduces the data rate from $10^6 - 10^8$ bytes s^{-1} at the output of a camera to something of the order of tens of features per frame that can be used as input to a robot's control system [19]. In this section we will discuss features and how to extract them from images.

Image Segmentation

Image segmentation is a critical process in computer vision. It involves dividing a visual input into segments to simplify image analysis. Segments represent objects or parts of objects, and comprise sets of pixels, or "super-pixels". Image segmentation sorts pixels into larger components, eliminating the need to consider individual pixels as units of observation.

Image segmentation is considered as three subproblems [19]. The first step is *classification* which is a decision process applied to each pixel that assigns the pixel to one of C classes $c \in \{0...C-1\}$. We commonly use $C = 2$ which is referred to as binary classification. The pixels have been classified as as object ($c = 1$) or non-object ($c = 0$) displayed as either white or black pixels in the image. We usually classify pixels based on some homogeneous characteristic of the object such as intensity or color.

The second step in the image segmentation process is *representation* where adjacent pixels of the same class c are connected to form a spatial set $S_1 ... S_m$. The sets are represented by assigning a list of pixel coordinates that defines the boundary of the connected set.

In the third step of the image segmentation, the sets S_i are *described* in a compact form such as scalar or vector-valued *image features* such as shape, size and position. The simplest representation of size and shape is the bounding box - the smallest rectangle with sides parallel to the u- and v-axes that encloses the region.

A depiction of segmentation and detection of fish with bounding boxes is seen in Figure 6.12.

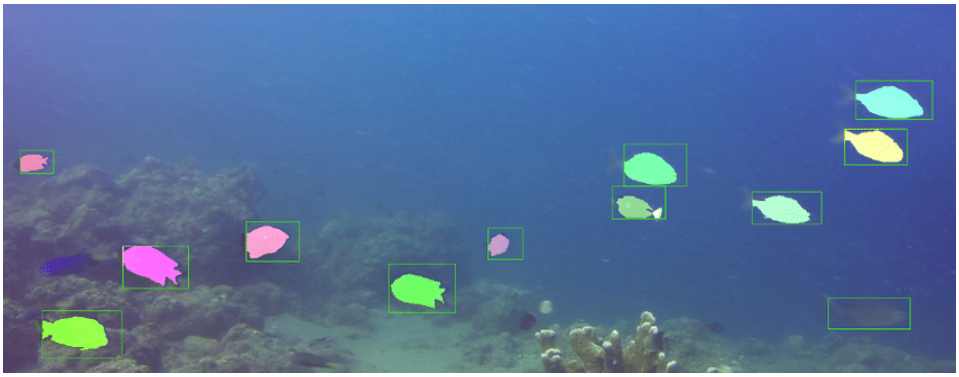


Figure 6.12: Object detection and image segmentation of fish. *Courtesy: [1, Labao, 2019]*

6.6 *Case Study: Underwater Object Detection using Thresholding Based Techniques for Image Feature Extraction*

Our approach in the field of robot perception encompasses numerous research goals and pursuits. For instance, we investigate lifelong mapping and environment reconstruction. Current robot mapping systems take one sweep through an environment, construct a map off-line, and then use it until the environment changes enough to require another sweep. Our goal is to continuously update a detailed geometrical map as the robot goes about its tasks, using both vision and laser sensors to recognize places, and label spaces and objects therein.

6.6.1 The Pole and Gate Detection Algorithm

1. **RGB to HSV Conversion:** We want to convert the image to HSV (hue-saturation-value) because working with HSV values is much easier to isolate colors. In the HSV representation of color, hue determines the color you want, saturation determines how intense the color is and value determines the lightness of the image.

2. **Use a Gaussian Blurring Kernel:** Gaussian Blurring is an example of applying a low-pass filter to an image. In computer vision, the term "low-pass filter" applies to removing noise from an image while leaving the majority of the image intact. When blurring a image, we make the color transition from one side of an edge in the image to another smooth rather than sudden. In effect there will be less rapid changes in pixel intensity. A blur is a very common operation we need to perform before other tasks such as edge detection.

3. **Color Thresholding for Binary Classification:** Image thresholding is a simple, yet effective, way of partitioning an image into a foreground and background. To isolate the colors, we have to apply multiple masks. A low threshold and high threshold mask for hue, saturation and value. Anything pixel within these thresholds will be set to 1 and the remaining pixels will be zero. For this algorithm the applied mask will get all of the red and yellow hues in the image to be able to mask the red pole and the yellow gate.

4. **Canny Detector Algorithm for Edge Detection:** For edge detection we'll be using a common algorithm called the *Canny edge detector* after it's creator John F. Canny (1986) [2]. It is a multi-stage algorithm consiting of four steps: Noise reduction, finding the intensity gradient of the image, non-maximum suppression to remove everything that does not constitute an edge, and finally hysteresis thresholding to reveal every edge in the image. The edge detection step is necessary for finding the silhuettes of the pole and the gate.

5. **Apply Bounding Boxes** The bounding boxes of the detected object - in this case a green pole and a yellow gate - are computed by applying the homography of the detected edges. To avoid picking up noise and unwanted elements in the image, a bounding box will appear if the detected element has a large enough size and proper aspect ratio to be recognized as either a gate or a pole.

	Min hue	***Max hue***	***Min sat***	***Max sat***	***Min val***	***Max val***
Gate	*90*	*151*	*6*	*250*	*0*	*255*
Pole	*98*	*178*	*9*	*134*	*17*	*255*

Table 6.1: Choice of hsv thresholding limits.

6.6.2 Software Implementation

The the link to the source code for this case study can be found in Appendix A

Open-Source Packages, Libraries and Standards

We will be using a common library for real-time computer vision called *OpenCV*. The library is cross-platform and free for use under the open-source BSD license. It also pro-

[2] https://docs.opencv.org/3.3.1/da/d5c/tutorial_canny_detector.html

vides very simple integration with ROS.

OpenCV has support for a lot of areas within computer vision ranging from classical image operations such as thresholding and blurring to statistical machine learning operations like Naive Bayes Classifiers (NBC) and Deep Learning Neural Networks (DNN). For this particular case study we will be using a selection of classical image operations to perform image segmentation and object detection. The reason for not using a popularized image segmentation technique such as DNN is because they are computationally much more expensive to run and also put additional requirements to graphics cards (GPU) for training the network and to run the inference model (the inference model applies knowledge from a trained neural network model and a uses it to infer a result). DNN also requires data collection and a labor intensive process of labeling images. Even though classical methods are more brittle and require human expertise to make, they are easily adapted to handle new objects and can do not need a dedicated GPU module (can run on CPU only).

In relation to choice of programming language, we will be using C++. Because Python is an interpreted language, while C++ is compiled down to machine code, generally speaking, you can obtain performance advantages using C++. In Python, all of the OpenCV functions return new copies of the image matrices. Whenever you capture an image, or if you resize it - in C++ you can re-use existing memory. There are workarounds in Python to obtain faster real-time performance by pre-allocating memory, but you will not be able to reach the exact same performance as with C++.

Simulation Results and Discussion

The simulation is performed in the Gazebo simulation environment created in the case study from Chapter 5. For this particular simulation we will be using a 1:1 scale version of the MC-lab as mentioned earlier. This simulation environment currently has two objects placed about 20m apart: A yellow gate and a green pole for demonstration of the computer vision capabilities. Luminance, light direction, shading, graining and fading has been applied to the environment in an attempt to recreate the poor and shifting visibility that a underwater camera will experience.

In Figure 6.13 - 6.15 you see the object detection algorithm searching for a yellow gate in the image. On the left side you see a binary image after blurring and color thresholding has been applied. On the right side you have the applied bounding box when the gate has been detected. Even though the contours of the gate is visible at a distance, the algorithm will not apply a bounding box unless the gate is close enough. This is done to avoid that the gate start picking up to much unwanted noise from the surroundings. Another criteria for applying a bounding box (seen in blue) is that the contours that make of the gate should be two vertical contours connected by a horizontal contour. The smallest possible rectangular enclosing of the contours should at least 1.5 wider than its height to ensure that we will not recognize the gate unless it's oriented facing straight at us. If this condition was not applied we might end up tracking a gate that is oriented at a degree such that the vehicle would not be able to enter through the opening.

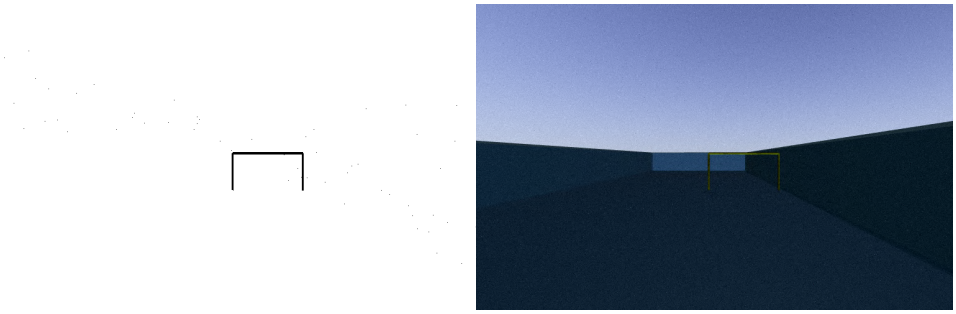


Figure 6.13: Gate detection picture 1.

As you can see in both Figure 6.14 and 6.15, the algorithm is not prone to noise. When the bounding boxes are applied, there is also some small bounding boxes appearing elsewhere in the image. These noise particles can be an issue when we later try to track the gate in the image, however they become less of a problem when the vehicle is close to the gate. All things considered, the computer vision algorithm is able to accurately track the gate for a real-time video stream.

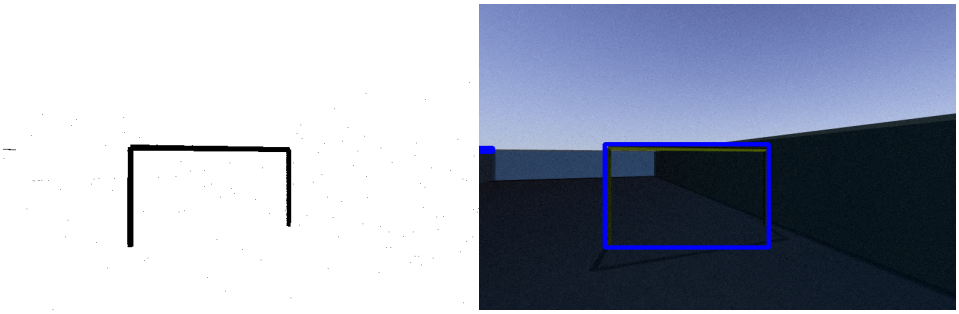


Figure 6.14: Gate detection picture 2.

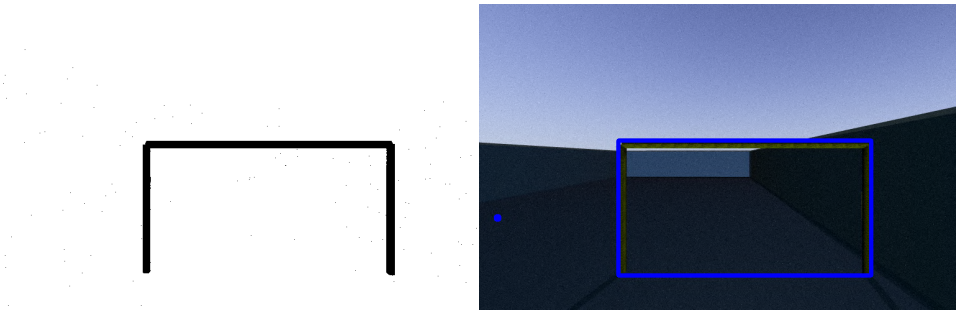


Figure 6.15: Gate detection picture 3.

In Figure 6.16 - 6.18 you see the object detection algorithm searching for a green pole in the image. The object detection algorithm for the pole is similar to that of the gate, but the binary thresholding is such that the background becomes dark ($c = 0$) and the object becomes white ($c = 1$). A criteria for applying a bounding box (seen in green) is that the object we are viewing is close enough (meaning the contours take up enough pixels in the binary image). We also have the criteria that the height of the contour is at least larger than it's width. For a single circular pole orientation does not mean anything because will always appear the same from any angle due to symmetry.

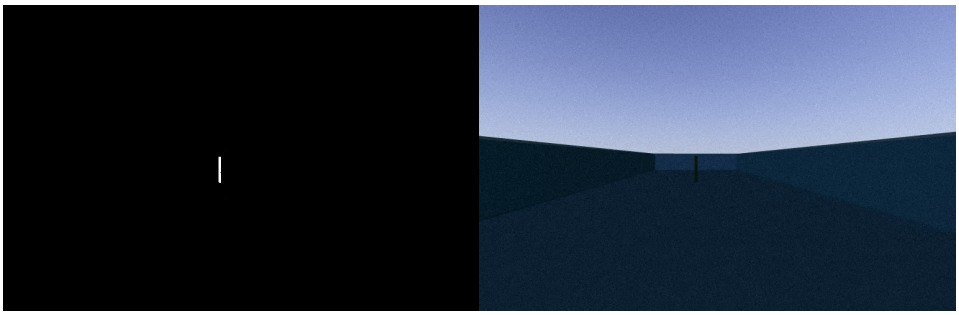


Figure 6.16: Pole detection picture 1.

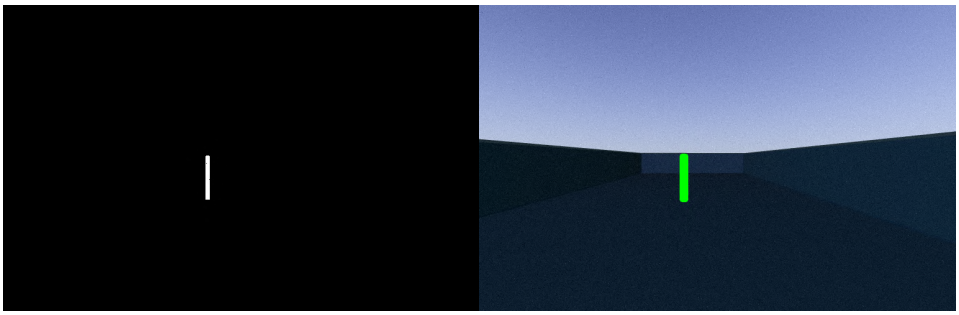


Figure 6.17: Pole detection picture 2.

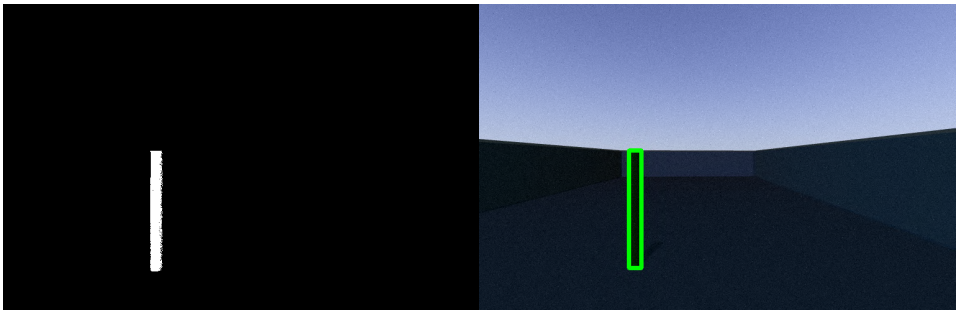


Figure 6.18: Pole detection picture 3.

6.6.3 Conclusion of Case Study

It should be noted that thresholding-based techniques are notoriously brittle - a slight change in illumination of the scene means that the thresholds that we have chosen in advance will no longer be appropriate. Distinguishing an object from the background remains a hard computer vision problem.

Improvement of the computer vision object detection through the use of convolutional neural networks instead of traditional computer vision algorithms would most likely improve robot perception across both simulation and in physical experiments.

6.7 Chapter Summary

The aim of the chapter was to design and implemented both localization and perception capabilities for the Manta v1 software architecture. Two case studies were conducted: The first one designed and implemented an extended Kalman filter for vehicle state estimation. The EKF prove good results in simulations and is suited for further use by the guidance and motion control. The second case study investigated robot perception by implementing computer vision object detection for detecting gates and poles. The results are usable for simulation purposes, but there should be invested time in implementing CNNs instead.

On a further note, since the Manta AUV is neither equipped with a sonar nor stereo vision it has not been possible create any form of path planning.

However, once any of the two senors are acquired one can start producing spatial memory represented through occupancy grid, volumetric maps, point clouds and graphs. The process of mapping can be done through particle filters for SLAM.

Once capable of constructing and updating maps through spatial memory, there should be invested considerable time in adding both local path planning (obstacle avoidance) and global path planning (route selection). Once implemented the vehicle could function of a local planner voting on the quality of arcs in front of the vehicle, a path-following

module votes likewise based on which arcs will take the AUV along the global path, and an arbitrator determines which direction is actually driven.

CHAPTER 7

GUIDANCE & MOTION CONTROL OF AUTONOMOUS UNDERWATER VEHICLES

The goal of this chapter is to design and implement motion controllers capable of both dynamic positioning and path following. The testing of the controllers will be performed in the Gazebo simulator using the derived EKF as part of the vehicle state feedback loop.

P.S, this chapter assumes the the reader has some knowledge in nonlinear control theory.

7.1 Guidance System

Guidance is the action or the system that continuously computes the reference (desired) position, velocity and acceleration of a marine craft to be used by the motion control system [27]. The data on which the guidance system computes its reference are usually provided by a human operator or the navigation system (typically a path planner). A the guidance system will work in close interaction with the motion control system and the navigation system. Based on the current localization and instructions by the path planner, the guidance system computes the instructions for the control system, which comprises the vehicle's actuators.

The different motion control scenarios are often classified according to:

- *Setpoint regulation* is a special case where the desired position and attitude are chosen to be constant.
- *Path following* is following a predefined path independent of time. There are no temporal constraints along the path.

- *Trajectory tracking*, where the objective is to force the system output $y(t) \in \mathbb{R}^m$ to track a desired output $y_d(t) \in \mathbb{R}^m$.

Depending the motion control scenario, one must choose a proper guidance method. For this project we'll be using a mass-spring-damper for setpoint regulation and line-of-sight guidance for path following of straight-line paths.

7.1.1 Reference Model for MIMO Setpoint Regulation

In order to provide high-performance dynamics positioning (DP) operations with transfer between station-keeping and marked position operations, a *reference model* must be introduced to calculate feasible and smooth trajectories for the vehicle to follow [89].

In its simplest form a reference model is obtained by using a low-pass filter (LP). But for marine vehicles it is more convenient to use reference models by dynamics of *mass-damper-spring systems* to generate feasible trajectories, for instance

$$\frac{x_d}{x_{ref}}(s) = \frac{\omega_{n_i}^2}{s^2 + 2\zeta_i\omega_{n_i}s + \omega_{n_i}^2} \tag{7.1}$$

where x_d is the desired state and x_{ref} denotes the input reference specified by the guidance system.

If applied in several degrees of freedom, the reference model is constituted in matrix form using:

$$\begin{aligned} \boldsymbol{\Omega} &= diag\{\omega_1, \omega_2, ...\} \\ \boldsymbol{\Delta} &= diag\{\zeta_1, \zeta_2, ...\} \end{aligned} \tag{7.2}$$

7.1.2 Line-of-Sight Guidance for Straight-Line Paths

$$\alpha_k := atan2(y_{k+1} - y_k, x_{k+1} - x_k) \in \mathbb{S} \tag{7.3}$$

Hence, the coordinates of the AUV in the path-fixed reference frame can be computed by:

$$\boldsymbol{\varepsilon}(t) = \boldsymbol{R}_p(\alpha_k)^T(\boldsymbol{p}^n(t) - \boldsymbol{p}_k^n) = [s(t), e(t)]^T \in \mathbb{R}^2 \tag{7.4}$$

where $s(t)$ represents the along-track distance (tangential to the path), and $e(t)$ cross-track error (normal to the path).The rotation matrix is defined as:

$$\boldsymbol{R}_p(\alpha_k) := \begin{bmatrix} cos(\alpha_k) & -sin(\alpha_k) \\ sin(\alpha_k) & cos(\alpha_k) \end{bmatrix} \tag{7.5}$$

The associated control objective for straight-line path following becomes:

$$\lim_{t\to\infty} e(t) = 0 \tag{7.6}$$

For path-following purposes, only the cross-track error is relevant since $e(t) = 0$ means that the AUV has converged to the straight line. See Figure 7.1 for a better context.

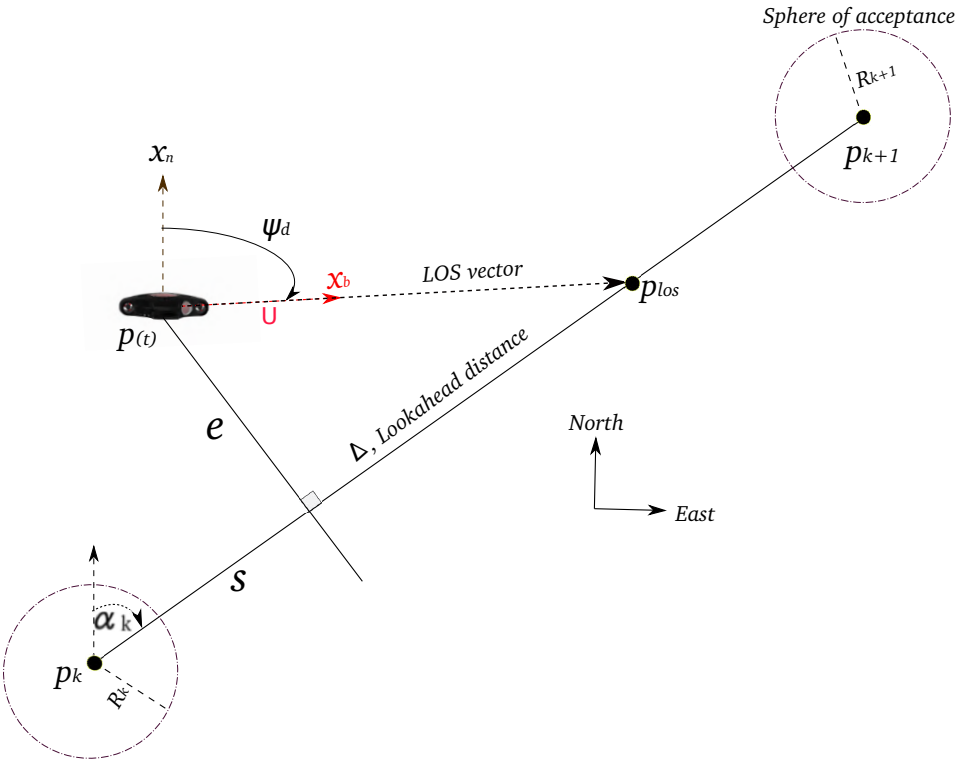


Figure 7.1: LOS guidance where the desired course angle $\psi_d = \chi_d$ (angle between x_n and the desired velocity vector) is chosen to point toward the LOS intersection point $p_{los} = (x_{los}, y_{los})$

The LOS position p_{los} is located somewhere along the straight line segment connecting the current p_k and the next p_{k+1} waypoints. Let the ship's current horizontal position $p(t)$. To be able to guide they vehicle towards its next goal, the LOS algorithm will compute a LOS vector pointing towards the coordinate p_{los} along the target path. Whenever the vehicle intersect the *sphere of acceptance* for waypoint p_{k+1} a new waypoint will be assigned.

There are two different guidance principles that can be used to steer along the LOS vector and at the same time stabilize $e(t)$ to the orgin. The two are *enclosure-based steering* and *lookahead-based steering*. The two steering methods essentially operate by the same principle, but the lookahead-based steering scheme is less computationally intensive than the enclosure-based approach.

Loohahead-Based Steering

For lookahead-based steering, the course angle assignment is separated into two parts:

$$\chi_d(e) = \chi_p + \chi_r(e) \tag{7.7}$$

where

$$\chi_p = \alpha_k \tag{7.8}$$

is the path-tangential angle, while

$$\chi_r(e) := arctan(\frac{-e}{\triangle}) \tag{7.9}$$

is a velocity-path relative angle, which ensures that the velocity is directed toward a point on the path that is located a lookahead distance $\triangle(t) > 0$ ahead of the direct projection of $\boldsymbol{p}^n(t)$ on to the path [27].

The control objective $\chi \rightarrow \chi_d$ is satisfied by transforming the course angle command χ_d to a heading angle command ψ_d using the fact that $\psi_d = \chi_d - \beta$. We do not operate under any large disturbance of wind, current or waves, so we assume the sideslip angle $\beta \approx 0$, resulting in:

$$\psi_d(e) = \alpha_k + arctan(\frac{-e}{\triangle}) \tag{7.10}$$

7.2 Thrust Allocation

The purpose of the thrust allocation is to distribute among the thrusters the desired generalized forces computed by the motion controller. The output from the thrust allocator is the desired thruster forces and directions.

For the Manta AUV the angle of the thrusters are fixed, but the thrusters themselves can produce thrust in both directions depending of the sign ($\pm$) of the rpm-input signal. The vehicle have a total of eight thrusters, see Figure 7.2 for thruster setup.

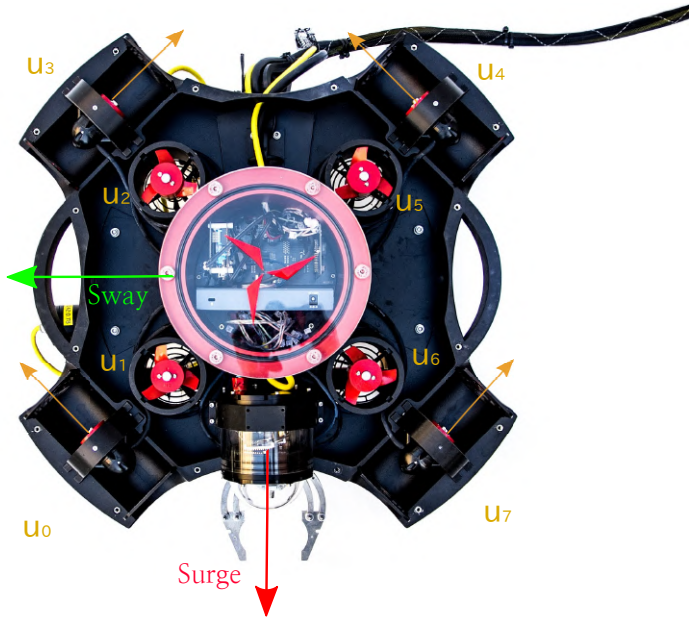


Figure 7.2: Thruster configuration, Manta AUV

The actuator forces and moments relate to the control forces and moments by

$$\boldsymbol{\tau} = \mathbf{T}\mathbf{K}\mathbf{u}_d \tag{7.11}$$

For the Manta AUV, the thruster configuration matrix become:

$$\mathbf{T} = \tag{7.12}$$

$$\begin{bmatrix} 0.7071 & 0.0000 & 0.0000 & -0.7071 & -0.7071 & 0.0000 & 0.0000 & 0.7071 \\ -0.7071 & 0.0000 & 0.0000 & -0.7071 & 0.7071 & 0.0000 & 0.0000 & 0.7071 \\ 0.0000 & 1.0000 & 1.0000 & 0.0000 & 0.0000 & 1.0000 & 1.0000 & 0.0000 \\ 0.0000 & 0.1200 & 0.1200 & 0.0000 & 0.0000 & -0.1200 & -0.1200 & 0.0000 \\ 0.0000 & -0.1200 & 0.1200 & 0.0000 & 0.0000 & 0.1200 & -0.1200 & 0.0000 \\ -0.2900 & 0.0000 & 0.0000 & 0.2900 & 0.2900 & 0.0000 & 0.0000 & 0.2900 \end{bmatrix}$$

where $\boldsymbol{\tau}$ is the generalized force vector, $\mathbf{T}$ is the thruster configuration matrix, $\mathbf{u}_d$ is the control input vector and $\mathbf{K} = diag\{K_i\}$ is the diagonal thruster coefficient. For this vehicle $\mathbf{K}$ is equal to the indentity matrix.

For a underwater vehicle such as Manta that is self-stabilized in roll and pitch, the *working space* is $m = 4$. Since the number of thrusters $r = 8$ this system is referred to as a overactuated system with $r > m$ [89] making the control allocation to an optimization problem with essentially infinite many solutions.

An explicit solution to this overdetermined solution space can be found using least-squares optimum [27]. The *generalized inverse* is found as

$$\mathbf{T}_w^{\dagger} = \mathbf{W}^{-1}\mathbf{T}^T(\mathbf{T}\mathbf{W}^{-1}\mathbf{T}^T)^{-1} \tag{7.13}$$

For our case $\mathbf{W} = \mathbf{I}$, that is equally weighted control forces and the Equation 7.13 reduces to the right *Moore-Penrose pseudo-inverse*

$$\mathbf{T}^{\dagger} = \mathbf{T}^T(\mathbf{T}\mathbf{T}^T)^{-1} \tag{7.14}$$

resulting in a control input vector specifying force on each thruster:

$$\mathbf{u}_d = \mathbf{T}^{\dagger}\boldsymbol{\tau} \tag{7.15}$$

The url link to the C++ and ROS implementation of the thrust allocation can be found in the Appendix A.

7.3 Nonlinear Motion Controllers

Nonlinear control theory can often yield a more intuitive design than linear theory. The downside of linearization is that the control effort is often not very efficient and often destroy the model properties of systems [27]. *Multiple-input multiple-output* (MIMO) systems are often controlled with *model-based* techniques, relying on dynamic models, most often linear empirical models obtained by system identification. The nonlinear design methods in this chapter are based on the robot-like model of Fossen (1991):

$$\begin{aligned} \dot{\boldsymbol{\eta}} &= \boldsymbol{J}_{\Theta}(\boldsymbol{\eta})\boldsymbol{\nu} \\ \boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{g}(\boldsymbol{\eta}) &= \boldsymbol{\tau} + \boldsymbol{\omega} \end{aligned} \tag{7.16}$$

It is important to understand the physical properties of the model in order to know which terms in the model can be omitted when deriving a model-based nonlinear controller. This is an important question since model inaccuracies can destabilize a feedback control system [27]. For this MIMO model class, the nonlinear controllers can designed by exploiting the fact that the mass matrix is positive definite and constant ($\boldsymbol{M} = \boldsymbol{M}^T > 0, \dot{\boldsymbol{M}} = 0$), the Coriolis and centripetal matrix $\boldsymbol{C}(\boldsymbol{\nu}) = -\boldsymbol{C}^T(\boldsymbol{\nu})$ is skew-symmetrical and the damping matrix $\boldsymbol{D}(\boldsymbol{\nu}) > 0$ is strictly positive. Controller design based on *Lyapunov stability theory* provides a means of stabilizing unstable nonlinear systems using feedback control. The idea is that if one can select a suitable function and force it to decrease along the trajectories of the system, resulting in a system converging to its equilibrium. In addition, the control can be chosen to speed up the *rate of convergence* to the origin by forcing the Lyapunov function to decrease to zero faster [55].

7.3.1 MIMO Nonlinear PID Control

Conventional proportional-integral-derivative (PID) control systems have their origin from SISO linear systems theory. However, it is possible to generalize this to **MIMO nonlinear systems** by expressing the marine vehicle equations of motion in a vectorial setting as in Equation 7.16. From Lyapunov stability proven that the control law in Equation 7.17 will suffice in controlling the vehicle if $\boldsymbol{J}_{\Theta}(\eta)$ is defined for all $\boldsymbol{\eta}$ (no representation singularity).

$$\boldsymbol{\tau} = -\boldsymbol{K}_p\tilde{\boldsymbol{\eta}} - \boldsymbol{K}_d\dot{\boldsymbol{\eta}} - \boldsymbol{K}_i \int_0^t \tilde{\boldsymbol{\eta}}(\tau)d\tau \tag{7.17}$$

A large advantage with the MIMO control strategy is that the interactions between the surge, sway and yaw modes could be dealt with. This is not possible with three decoupled PID controllers.

Pros and Cons

The PID controller is arguably the most popular control strategy used in industry to this date, here are some of common advantages and disadvantages of the controller:

Pros	Cons
Intuitive and simple controllers	May not assure the desired performance for changing operating points.
Easy to tune	
It is efficient and robust against some common uncertainties if properly tuned (around an operating region)	

7.3.2 MIMO Nonlinear Backstepping Control

Backstepping is a recursive procedure that interlaces the choice of a Lyapunov function with the design of feedback control. It breaks a design problem for the full system into a sequence of design problems for lower order (even scalar) subsystems. By exploiting the extra flexibility that exists with lower order and scalar subsystems, backstepping can often solve stabilization, tracking, and robust control problems under conditions less restrictive than those encountered in other methods [38, Hassan K. Khalil, 2002].

In general the backstepping design approach requires that the system can be expressed on the form:

$$\begin{aligned} \dot{\eta} &= f(\eta) + g(\eta)\xi \\ \dot{\xi} &= \tau \end{aligned} \tag{7.18}$$

Since the nonlinear system consists of two states η and ξ, this will be a recursive design in two steps and the system in Equation 7.18 is therefore treated as two cascaded systems,

each with a single input and output (SISO system). As you will see later, we can extend this to MIMO systems by introducing the *two vectorial steps* (first introduced by Fossen and Berge [28]) allowing us to exploit the structural properties of nonlinear MIMO systems. This will simplify the design and analysis significantly.

Backstepping design introduces a *stabilizing function* $\phi(\eta)$ and a new state variable $z = \xi - \phi(\eta)$. Through variable change the system can be re-formulated as:

$$\begin{aligned} \dot{\eta} &= [f(\eta) + g(\eta)\phi(\eta)] + g(\eta)z \\ \dot{z} &= \tau - \dot{\phi} \end{aligned} \tag{7.19}$$

The design objective is to render the equilibrium point *globally asymptotically stable* (GAS) or *globally exponentially stable* (GES). By introducing a *Lyapunov function candidate* (LFC) of type:

$$V_c(\eta, \xi) = V(\eta) + \frac{1}{2}z^2 \tag{7.20}$$

and assuring that *globally* V_c remains positive definite and radially unbounded, and that $\dot{V}_c$ is negative definite (in other words, the origin (η, z) is GAS), we will be able to derive the state feedback control law on the form:

$$\tau = \frac{\partial \phi}{\partial \eta}[f(\eta) + g(\eta)\xi] - \frac{\partial V}{\partial \eta}g(\eta) - k[\xi - \phi(\eta)] \tag{7.21}$$

Having chosen the stabilizing function $\phi(\eta)$ such that the backstepping transformation is a *global diffeomorphism* (meaning $\eta = \phi^{-1}(z)$ and $\phi(0) = 0$), then the origin of the original system (η, ξ) will also be GAS.

The price for exploiting the so-called good nonlinearities in the design is that the error dynamics becomes *nonautonomous* (meaning time-dependent).

Pros and Cons

Unlike the PID controller, the backstepping design approach is less common control strategy used in industry solutions. At face value it can seem rather tedious

Pros	Cons
Does not cancel stabilizing terms (good nonlinearities)	The controllers can be rather complex
Simple proofs	Need a good approximation of the system parameters
Achieves a cascaded system structure	

You may use the method to achieve a cascaded systems structure with an asymptotically stable nominal system. Then you can utilize cascaded control systems theory to derive the control law and prove stability of the total system.

7.4 *Case Study: Stationkeeping and Low-speed Setpoint Regulation using Quaternion Feedback Regulation for the Manta AUV*

This case study will focus on designing and implementing a position and attitude setpoint regulator for the Manta AUV in 6 degrees of freedom. A nonlinear PID-control law for position and attitude regulation is presented using Euler rotation for attitude feedback where the attitude is represented as unit quaternions. The controller is derived using Lyapunov functions for stability properties and then later implemented in $C++$ and ROS. The works of this case study is based on the paper *Quaternion Feedback Regulation of Underwater Vehicles* [24, Fjellstad & Fossen, 1994].

7.4.1 Problem Statement

For 6 DOF control problems like underwater vehicles there are significant couplings between rotational and translational motion. For instance, hydrodynamic damping will be strongly coupled and the hydrodynamic added mass will introduce additional couplings due to Coriolis and centrifugal forces.

To increase the applicability of an AUV, it should be able to operate at any global attitude. There are some obvious disadvantages in terms of Euler angle representation in a control system as mentioned in Section 4.1.2. Consequently, it cannot be claimed that Euler angles are better suited than other attitude represented than other attitude representations in control applications.

7.4.2 Control Design

Attitude Error Dynamics

The rotation matrix $\boldsymbol{R}_b^n(\boldsymbol{q})$ in Equation 4.5, represents the actual attitude of the vehicle in a inertial frame. Let $\boldsymbol{R}_b^n(\boldsymbol{q}_d)$ denote the desired attitude of the vehicle in body rotated to an inertial frame. The control objective is to make the body-frame coincide with a desired-frame such that $\boldsymbol{R}_b^n(\boldsymbol{q}) = \boldsymbol{R}_b^n(\boldsymbol{q}_d)$. The attitude error is defined as $\boldsymbol{R}_b^n(\tilde{\boldsymbol{q}}) = \boldsymbol{R}_b^{nT}(\boldsymbol{q}_d)\boldsymbol{R}_b^n(\boldsymbol{q})$, where

$$\tilde{\boldsymbol{q}} = \bar{\boldsymbol{q}}_d\boldsymbol{q} = \begin{bmatrix} \eta_d & \boldsymbol{\varepsilon}_d^T \\ -\boldsymbol{\varepsilon}_d & \eta_d\mathbf{I}_{3x3} - \mathbf{S}(\boldsymbol{\varepsilon}_d) \end{bmatrix} \begin{bmatrix} \eta \\ \boldsymbol{\varepsilon} \end{bmatrix} \tag{7.22}$$

and $\bar{\boldsymbol{q}}$ is the complex conjugate of quaternion $\boldsymbol{q}$.

The perfect setpoint regulation is expressed in quaternion notation as:

$$\boldsymbol{q} = \boldsymbol{q}_d \Leftrightarrow \tilde{\boldsymbol{q}} = \begin{bmatrix} \pm 1 \\ \boldsymbol{0} \end{bmatrix} \tag{7.23}$$

The attitude error differential equations follows as

$$\dot{\tilde{\boldsymbol{q}}} = \boldsymbol{T}_q(\tilde{\boldsymbol{q}})\tilde{\boldsymbol{\omega}}^b_{b/n} \tag{7.24}$$

where desired angular velocity $\boldsymbol{\omega}^b_{b/n_d} = \boldsymbol{0}$. Hence, $\tilde{\boldsymbol{\omega}}^b_{b/n} = \boldsymbol{\omega}^b_{b/n} - \boldsymbol{\omega}^b_{b/n_d} = \boldsymbol{\omega}^b_{b/n}$

Euler Rotation Feedback Control

Euler rotation feedback is obtained by substituting the vector quaternion ε with the Euler rotation $2\boldsymbol{\eta\varepsilon}$ in the control law. Assuming that $\omega = 0$, the system we want to control is expressed as:

$$\boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{g}(\boldsymbol{q}) = \boldsymbol{\tau} \tag{7.25}$$

The design objective is to render the equilibrium point (GAS) or (GES). To do that we have to choose a proper $\boldsymbol{\tau}$ that ensure this stability property. Before we dive into the Lyapunov proof, we are going to introduce a few new variables to make the computation easier:

$$\boldsymbol{z}_1 = \begin{bmatrix} \tilde{\boldsymbol{x}} \\ \tilde{\boldsymbol{\varepsilon}} \end{bmatrix}, \qquad \boldsymbol{K}_p(\boldsymbol{q}) = \begin{bmatrix} \boldsymbol{R}_b^{n\,T}(\boldsymbol{q})\boldsymbol{K}_x & \boldsymbol{0}_{3x3} \\ \boldsymbol{0}_{3x3} & \frac{c}{2}\boldsymbol{I}_{3x3} \end{bmatrix} \tag{7.26}$$

Having that $\boldsymbol{K}_x = \boldsymbol{K}_x{}^T > \boldsymbol{0}_{3x3}$, with $c > 0$. With $\boldsymbol{K}_1$ defined as:

$$\boldsymbol{K}_1 = \begin{bmatrix} \boldsymbol{K}_x & \boldsymbol{0}_{3x3} \\ \boldsymbol{0}_{3x3} & c\boldsymbol{I}_{3x3} \end{bmatrix} = \boldsymbol{K}_1^T > \boldsymbol{0}_{6x6} \tag{7.27}$$

Since $\boldsymbol{M} = \boldsymbol{M}^T > \boldsymbol{0}_{6x6}$ and $\dot{\boldsymbol{M}} = \boldsymbol{0}$ (this was described in Chapter 4). By introducing a *Lyapunov function candidate* (LFC) of type:

$$V_1 = \frac{1}{2}(\boldsymbol{\nu}^T\boldsymbol{M}\boldsymbol{\nu} + \boldsymbol{z}_1^T\boldsymbol{K}_1\boldsymbol{z}_1) \tag{7.28}$$

time differentiation of V_1 gives:

$$\begin{aligned} \dot{V}_1 &= \boldsymbol{\nu}^T\boldsymbol{M}\dot{\boldsymbol{\nu}} + \dot{\boldsymbol{z}}_1^T\boldsymbol{K}_1\boldsymbol{z}_1 \\ &= \boldsymbol{\nu}^T\boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{\nu}^T\boldsymbol{J}_q{}^T(\boldsymbol{q})\boldsymbol{K}_1\boldsymbol{z}_1 \end{aligned} \tag{7.29}$$

$$\begin{aligned}\dot{V}_1 &= \boldsymbol{\nu}^T[\boldsymbol{\tau} - \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} - \boldsymbol{g}(\boldsymbol{q})] + \boldsymbol{\nu}^T \begin{bmatrix} \boldsymbol{R}_b^{n\,T}(\boldsymbol{q}) & \boldsymbol{0}_{3x3} \\ \boldsymbol{0}_{3x3} & \frac{1}{2}\boldsymbol{T}_q{}^T(\tilde{\boldsymbol{q}}) \end{bmatrix} \boldsymbol{K}_1 \dot{\boldsymbol{z}}_1 \\ &= \boldsymbol{\nu}^T[\boldsymbol{\tau} - \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} - \boldsymbol{g}(\boldsymbol{q})] + \boldsymbol{\nu}^T \boldsymbol{K}_p(\boldsymbol{q}) \boldsymbol{z}_2 \end{aligned} \tag{7.30}$$

with $\boldsymbol{T}_q(\boldsymbol{q})$ defined by Equation 4.6, $\boldsymbol{K}_p(\boldsymbol{q})$ defined by Equation 7.26 and

$$\boldsymbol{z}_2 = \begin{bmatrix} \tilde{\boldsymbol{x}} \\ \tilde{\eta}\tilde{\boldsymbol{\varepsilon}} \end{bmatrix} \tag{7.31}$$

Since $\boldsymbol{\nu}^T \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} \equiv \boldsymbol{0}$ and $\boldsymbol{\nu}^T \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} > \boldsymbol{0}$ we can choose the control law of a PD type according to

$$\boldsymbol{\tau} = -\boldsymbol{K}_d \boldsymbol{\nu} - \boldsymbol{K}_p(\boldsymbol{q})\boldsymbol{z}_2 + \boldsymbol{g}(\boldsymbol{q}) \tag{7.32}$$

with $\boldsymbol{K}_d = \boldsymbol{K}_d{}^T > \boldsymbol{0}_{6x6}$ the Lyapunov function time derivative becomes negative definite, that is:

$$\dot{V}_1 = -\boldsymbol{\nu}^T[\boldsymbol{K}_d + \boldsymbol{D}(\boldsymbol{\nu})]\boldsymbol{\nu} \leq \boldsymbol{0} \tag{7.33}$$

7.4.3 Investigating Stability

We evaluate the closed-loop system in relation to the Lyapunov function candidate (LFC) in Equation 7.30. Evaluating the system with respect to the error variable in Equation 7.31, we can determine that the system has three closed-loop equilibrium points.

There are two equilibrium at the error quaternions $\tilde{\varepsilon} = 0 \Rightarrow \tilde{\eta} = \pm 1$ and $\tilde{\eta} = 0$. Both equilibrium points are *stable* and represent the vehicle achieving a desired attitude (see Equation 7.23). If $\boldsymbol{q}$ represent one certain attitude, then $-\boldsymbol{q}$ is the same attitude after a $\pm 2\pi$ rotaion about an axis. Physically these two points are indistinguishable, but mathematically they are distinct.

The latter equilibrium point $\tilde{\eta} = 0$, is however, *unstable*. Suppose that $\tilde{\eta} = 0$ and $\tilde{\boldsymbol{x}} = \boldsymbol{0}$. The steady-state of the Lyapunov function will be

$$V_{1s} = \frac{1}{2}(\boldsymbol{\nu}^T \boldsymbol{M} \boldsymbol{\nu} + 0) = \frac{c}{2} \tag{7.34}$$

Where c is some scalar value. If the system advancing towards $\tilde{\eta} = \pm\varepsilon$, the Lyapunov function is changed:

$$V_1 = \frac{c}{2}(1 - \varepsilon^2) < V_{1s} \tag{7.35}$$

Consequently, application of *LaSalle's invariant set theorem* [27], implies (almost) global asymptotic stability. From Equation (7.33) it is seen that *local asymptotic stability* (LAS) is achieved for a proportional feedback control law, that is $\boldsymbol{K}_d = \boldsymbol{0}_{6x6}$.

Adding Integral Action

Unmodelled external forces and moments due to mostly currents, but sometimes also wind and waves (if operating in the wave-zone) are lumped into a body-fixed disturbance vector $\boldsymbol{w} \in \mathbb{R}^3$ and added to the low-frequency low-speed dynamic model.

$$\boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{g}(\boldsymbol{q}) = \boldsymbol{\tau} + \boldsymbol{w} \tag{7.36}$$

If the disturbance vector $\boldsymbol{w}$ have a non-zero mean, this will result in a steady-state offset when using the nonlinear PD-controller [89, Sørensen, 2014]. To counteract the steady-state offset we include integral action so that

$$\boldsymbol{\tau}_{PID} = -\boldsymbol{K}_d\boldsymbol{\nu} - \boldsymbol{K}_p(\boldsymbol{q})\boldsymbol{z}_2 + \boldsymbol{g}(\boldsymbol{q}) - \boldsymbol{K}_i(\boldsymbol{q}) \int_0^t \tilde{\boldsymbol{z}}_2(\tau)d\tau \tag{7.37}$$

with

$$\boldsymbol{K}_i(\boldsymbol{q}) = \begin{bmatrix} \boldsymbol{R}_b^{nT}(\boldsymbol{q})\boldsymbol{K}_{ix} & \boldsymbol{0}_{3x3} \\ \boldsymbol{0}_{3x3} & \frac{c_i}{2}\boldsymbol{I}_{3x3} \end{bmatrix} \tag{7.38}$$

If the integral action is included with $\boldsymbol{K}_i > \boldsymbol{0}$, it is possible to prove local asymptotic stability (LAS) also for the case $\boldsymbol{w} \neq \boldsymbol{0}$ [27, Fossen, 2011, page 377].

7.4.4 Guidance System Design

Waypoint Switching using Sphere of Acceptance

For this controller we chose to include setpoint regulation with a static desired position and attitude. A new setpoint is released whenever the vehicle is within the boundaries of the *sphere of acceptance* given by

$$[x_{k+1}]^2 + [y_{k+1}]^2 + [x_{k+1}]^2 \leq R_{k+1}^2 \tag{7.39}$$

Reference Model using Tustin's Method

Tustin's Method is used in digital signal processing and discrete-time control theory to transform continuous-time system representations to discrete-time and vice versa [89]. When designing a digital control system, we first need to find the discrete equivalent of the continuous portion of the system.

It is possible to relate the coefficients of a continuous-time, analog filter with those of a similar discrete-time digital filter created through the biquad transform process. Transforming a general, second-order continuous-time filter with the given transfer function of a *mass-damper-spring system* given in Equation 7.2 using Tustin's approximation

$$s = \frac{2(1-z^{-1})}{T(1+z^{-1})} \tag{7.40}$$

where T is the numerical integration step size of the trapezoidal rule, we get that

$$\frac{x_d}{x_{ref}}(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}} \tag{7.41}$$

by doing a inverse z-transform and normalizing the constant term in the denominator you'll end up with

$$x_d[k] = b_0 x_{ref}[k] + b_1 x_{ref}[k-1] + b_2 x_{ref}[k-2] - a_1 x_d[k-1] - a_2 x_d[k-2] \tag{7.42}$$

Here the b_0, b_1 and b_2 coefficients determine zeros, and a_1 and a_2 determine position of the poles.

$x_d[k]$ = the current filtered (output) value
$x_d[k-1]$ = the last filtered (output) value
$x_d[k-2]$ = the 2nd-to-last filtered (output) value
$x_{ref}[k]$ = the current raw input value
$x_{ref}[k-1]$ = the last raw input value
$x_{ref}[k-2]$ = the 2nd-to-last raw input value

For our reference model we would like to choose the relative damping ratio $\zeta = 1.0$ and $\omega_n = 0.1$ for a critically damped and rapid reference model. Inserting the values into Equation (7.1) the transfer function is

$$\frac{x_d}{x_{ref}}(s) = \frac{0.01}{s^2 + 0.2s + 0.01} \tag{7.43}$$

Performing a z-transform using Tustin's method in Matlab the discrete transfer function become

$$\frac{x_d}{x_{ref}}(z) = \frac{6.219 \times 10^{-6} z^2 + 1.244 \times 10^{-5} z + 6.219 \times 10^{-6}}{z^2 - 1.99z + 0.99} \tag{7.44}$$

giving us the following coefficients

$$\begin{aligned} b_0 &= 6.219 \times 10^{-6} \\ b_1 &= 1.244 \times 10^{-5} \\ b_2 &= 6.219 \times 10^{-6} \\ a_0 &= 1.0 \\ a_1 &= 1.99 \\ a_2 &= 0.99 \end{aligned} \tag{7.45}$$

Inserting the coefficients the discrete filter, $x_d[k]$ from Equation 7.42 the desired position and velocity for a reference point becomes $(N, E, D) = (5, 10, 3)$. See Appendix C.1.1 for plots of step response.

The plots show that the reference model produces a feasible path converging to the referenced positions in time. The velocities are also stable and will decrease to zero in time.

The Matlab source-code for computing poles and coefficients can be found in Appendix D.3.

Investigating Stability

This system has two zeros and two poles that are all on the unit circle, and thus maintaining the stability properties of the discrete system as seen by the root locus plot in Figure 7.3.

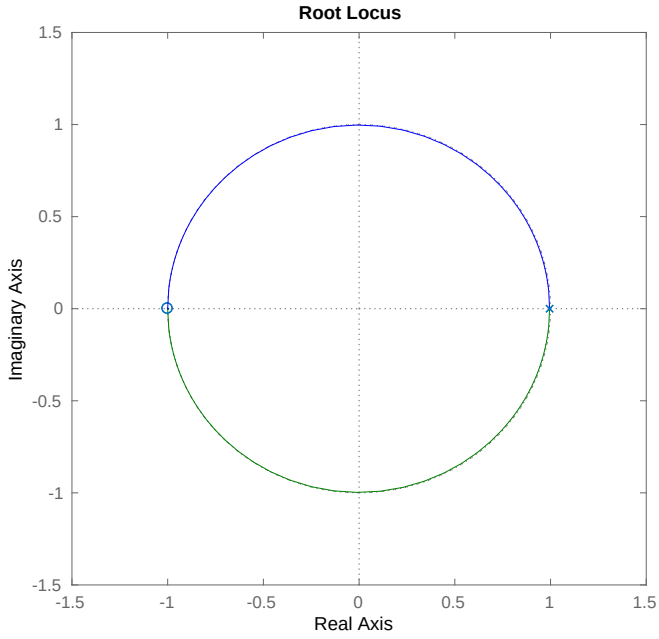


Figure 7.3: Root locus in z-domain

7.4.5 Software Implementation

The urls linking to the source code can be found in Appendix A.

Open-Source Packages, Libraries and Standards

The controller is a nonlinear PID controller with Euler rotation feedback control as derived in Section 7.4.1. It is implemented in C++ and ROS, and consist of about 2000 lines of code. The code is written using *Eigenlib* - a C++ template library for linear algebra: matrices, vectors, numerical solvers. Eigenlib is preferable over other libraries such as *Armadillo* due to its extensive support for use of quaternions in cooperation with ROS.

Setpoint Regulation

Figure 7.4 shows a communication tree of the guidance system and the controller node. The setpoint generator are provided by a guidance module called */static_waypoint_client* implemented in Python using ROS. The */static_waypoint_client* node as seen in Figure 7.4

is an action client that sends out an action goal */move_base/goal* consisting of a desired position provided in inertial cartesian coordinates and a desired attitude in unit quaternions. The action goal is picked up by the controller. The controller will continuously send feedback and status composed in the messages */move_base/feedback* and */move_base/status* back to the guidance module */static_waypoint_client*. Whenever the controller has reached its goal it will send a message */move_base/result* back to the guidance module saying it has finished. The guidance system will then proceed to send out a new setpoint goal. The action goal can always be cancelled by the guidance module by sending a message */move_base/cancel*.

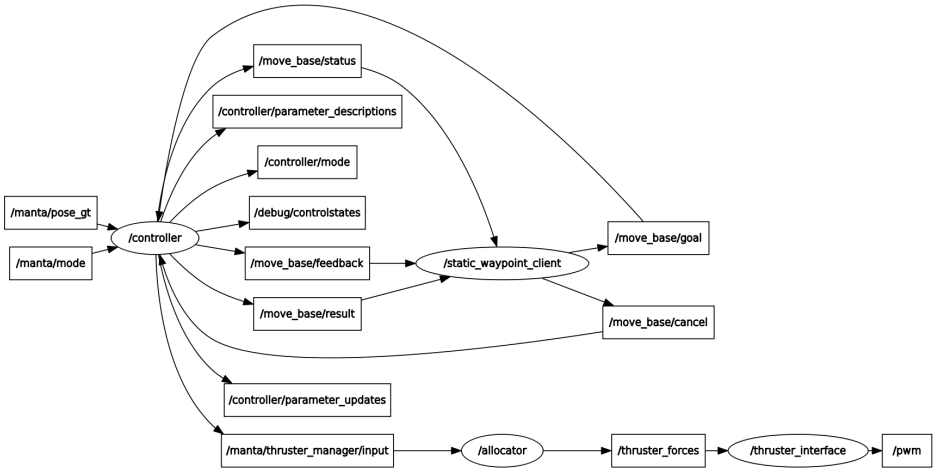


Figure 7.4: A communication tree showing all participating nodes and messages

Reference Model

Whenever the controller gets a new setpoint it will first be handled by the reference model. This is done to avoid that the actuators are saturated and the system becoming unstable. The reference model is the second-order *mass-damper-spring* system using Tustin's discretization derived in Section 7.4.4.

Controller Modes

The controller is implemented as an action server, meaning it will take goals from an action client such as the guidance module */static_waypoint_client*. The state of the vehicle is provided by Gazebo as a ROS topic */manta/pose_gt* and consist of 15 different measurements including absolute position and attitude, linear and angular velocity and linear accelerations. Whenever the controller has computed its desired generalized forces in body-frame, it is published as message on a rostopic */manta/thruster_manager/input*. The

message consisting of a generalized forces vector τ is handled by an allocator node and the control input u_d is then sent to their respective thrusters. The thrusters control input is then mapped on from force [N] to PWM-signal on a conversion curve and then published on the topic */pwm*, see Figure 7.4.

The controller is implemented with 7 different operational modes that can later can easily switched using supervisory control. The different control modes are:

1. **OPEN LOOP** The controller has no feedback loop and can be controlled using a remote joystick
2. **POSE_HOLD** The controller will track setpoints in (x,y,z), while (roll,pitch,yaw) remain open loop.
3. **HEADING_HOLD** The controller will only track setpoint in (yaw), while (x,y,z, roll,pitch) remain open loop.
4. **DEPTH_HEADING_HOLD** The controller will track setpoints in (z,yaw), while (x,y,roll,pitch) remain open loop.
5. **DEPTH_HOLD** The controller will track setpoints in (z), while (x,y,roll,pitch, yaw) remain open loop.
6. **POSE_HEADING_HOLD** The controller will track setpoints in (x,y,z,yaw), while (roll,pitch) remain open loop.
7. **CONTROL_MODE_END** The controller will end all control action.

Models for dynamic positioning are derived under the assumption of low speed. The DP models are valied for stationkeeping and low-speed manueuvering up to approximately $2m/s$ [27]. For this particular project we design our AUV for a maximum transit velocity of about $U = 0.7m/s$. It was decided not to include the velocities generated by the reference model, but rather setting the desired translational and angular veloctites $\nu_d = 0$.

Controller Tuning

The tuning process is done by performing parameter corrections while monitoring the step response of each degree that we want to control through a plotting tool (*rqt_plot* in this case). To tune a PID, use the following steps:

1. Set all gains to zero.
2. Increase the P gain until the response to a disturbance is steady oscillation.
3. Increase the D gain until the the oscillations go away (i.e. it's critically damped).
4. Repeat steps 2 and 3 until increasing the D gain does not stop the oscillations.
5. Set P and D to the last stable values.

6. Increase the I gain until it brings you to the setpoint with the number of oscillations desired (normally zero but a quicker response can be had if you don't mind a couple oscillations of overshoot)

The final tuning parameters can be found in Appendix C.2.1.

7.4.6 Simulation Results and Discussion

The simulation is performed in the Gazebo simulation environment created in the case study from Chapter 5, which is a 1:1 scale version of the marine cybernetics lab at the Department of Marine Technology, NTNU. For vehicle localization the AUV take use of the extended Kalman filter (EKF) created in the case study from Section 6.4.

In Figure 7.5 you see the 3-D plot of the recorded run. The red graph shows the estimated location of the vehicle, the green graph shows the ground truth position of the vehicle, the large blue spheres are the sphere of acceptance (SOA) of the waypoints and the small blue dots are the actual target waypoints themselves.

As you can see, the DP-controller is successfully able to dive from the surface down to the first waypoint located at $-0.5m$ depth. The sphere of acceptance for the each waypoint is $5cm$, which really demonstrates how precise the nonlinear PID controller is in tracking the square made up of 5 waypoints. Scan the QR-code in Figure 7.5 with a QR-code scanner app to see the real-time simulation in Gazebo.

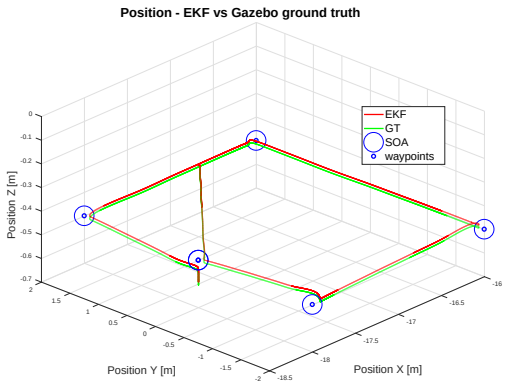


Figure 7.5: Dynamic positioning in Gazebo. EKF vs GT with plotted waypoints and sphere of acceptance (SOA). Scan the QR code to see the real-time simulation on youtube

On the left side in Figure 7.6 you see the estimated position of the vehicle on the top, while on the bottom you have the actual ground truth position. But more interesting are the two plots on the right showing the estimated and ground truth attitude of the vehicle.

The desired roll, pitch and yaw angle for all five setpoints are $0[deg]$, and as you can see the controller is successful in doing that with a maximum deviation of $-1.3[deg]$ at most.

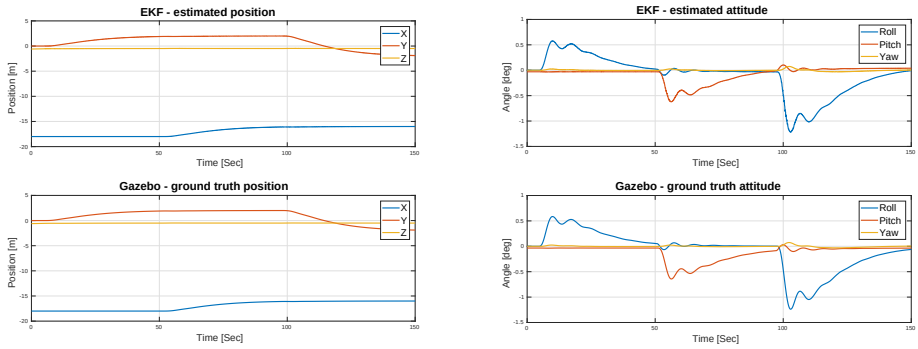


Figure 7.6: EKF vs GT position and attitude

On the left side in Figure 7.13 you see the desired forces [N] in body frame on the bottom plot and the actual produced linear velocities [m/s]in body frame on the top. What is interesting to see is that, even though the units are different for the two plots, they are more or less proportional to each other and has almost no phase delay. We designed the dp-controller such that the velocities will drive towards zero once we close to the target waypoint. As you can see in the plots, both the thrust and velocities drives towards zero showing us that the controller in fact has a stable behavior where the energy in the controllers gradually escape the system.

The same goes with the two plots on the right in Figure 7.13. On the bottom you have the desired torques [Nm] in body frame, while on the top you have the actual angular velocities [deg/s]. And again, you can see the even though the units are different for the two plots, the torques and angular velocities are more or less proportional to each other and there is barely any phase delay. The the torques and angular velocities will adjust to accommodate the desired goal of having zero roll, pitch and yaw angle. As you can see, both the torques and angular velocities swing towards zero and again demonstrating behavior of a stable controller.

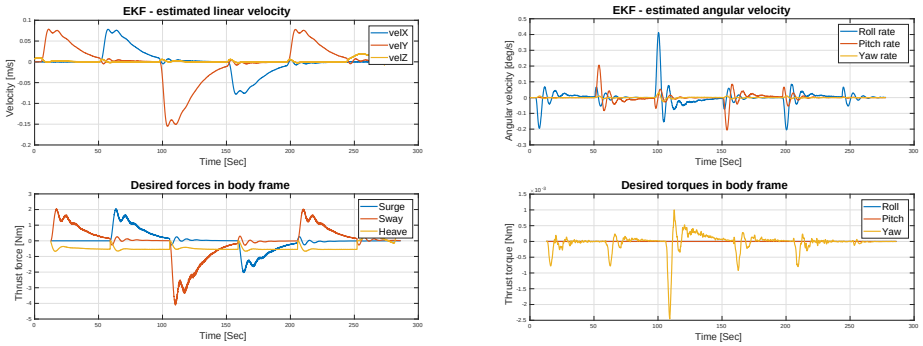


Figure 7.7: Desired torques and forces in body frame vs actual linear and angular velocities

7.4.7 Conclusion of Case Study

To conclude the case study, one can tell that the nonlinear PID controller is successful in regulating in 6 DOF with a really high precision. The precision of the controller is not only ensured by the careful tuning of the controller itself, but also the guidance module that help guide the vehicle towards the setpoint with a pace that matches the dynamics of the vehicle. The implemented mass-spring-damper discretized with Tustins' method is successful in doing that.

7.5 *Case Study: Path-Following using Line-of-Sight Guidance with 3 DOF Backstepping Controller for the Manta AUV*

This case study presents a maneuvering controller involving an LOS guidance system and a nonlinear feedback trajectory-controller. The design of this controller is based on the paper: *Path-following Controller for Underactuated Marine Craft* [84, Skjetne, Fossen, Breivik, 2003].

First, a 3DOF tracking controller is derived using the backstepping technique. Three stabilizing functions $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \alpha_3]^T$ are defined where α_1 and α_3 are specified to satisfy the tracking objectives in surge and yaw modes. The stabilizing function α_2 in sway mode is left a free design variable. By assigning dynamics to α_2, the resulting controller becomes a a dynamic feedback controller so that $\alpha_2(t) \rightarrow v(t)$

7.5.1 Problem Statement

In the previous case study we developed a nonlinear PID controller for setpoint regulation. This controller is particularly good for low-speed waypoint tracking and dynamic positioning in a terminal state. However, the dp-controller will not suffice for long distance transit while tracking a path. For this purpose we need a path-following controller.

The maneuvering problem is divided into two tasks as suggested by [85, Skjetne et al., 2004]. The first, called the *geometric task*, is to force the system output to converge to a desired path parametrized by a continous scalar variable of your choice. The second task, called the *dynamic task*, is to satisfy a desired dynamics along the path. This dynamic behavior is further specified via time, speed, or acceleration assignment. While the main concern is to satisfy the geometric task, the dynamic task ensures that the system follows the path with desired speed.

7.5.2 Control Design

Consider the 3 DOF nonlinear maneuvering model assuming there is no current ($\boldsymbol{\nu_r} = \boldsymbol{\nu}$) in the following form:

$$\begin{aligned} \dot{\boldsymbol{\eta}} &= \boldsymbol{R}(\psi)\boldsymbol{\nu} \\ \boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} = \boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{N}(\boldsymbol{\nu})\boldsymbol{\nu} &= \begin{bmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \end{bmatrix} \end{aligned} \tag{7.46}$$

where $\boldsymbol{\eta} = [N, E, \psi]^T$, $\boldsymbol{\nu} = [u, v, r]$ and

$$\boldsymbol{R}(\psi) = \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{7.47}$$

The matrices $\boldsymbol{M}$ and $\boldsymbol{N}(\boldsymbol{\nu})$ take the following form:

$$\boldsymbol{M} = \begin{bmatrix} m_{11} & 0 & 0 \\ 0 & m_{22} & m_{23} \\ 0 & m_{32} & m_{33} \end{bmatrix} = \begin{bmatrix} m - X_{\dot{u}} & 0 & 0 \\ 0 & m - Y_{\dot{v}} & mx_g - Y_{\dot{r}} \\ 0 & mx_g - N_{\dot{v}} & I_z - N_{\dot{r}} \end{bmatrix} \tag{7.48}$$

$$\boldsymbol{N}(\boldsymbol{\nu}) = \begin{bmatrix} n_{11} & 0 & 0 \\ 0 & n_{22} & n_{23} \\ 0 & n_{32} & n_{33} \end{bmatrix} = \begin{bmatrix} -X_u & 0 & 0 \\ 0 & -Y_v & mu - Y_r \\ 0 & -N_v & mx_g u - N_r \end{bmatrix} \tag{7.49}$$

where $\boldsymbol{M} = \boldsymbol{M}^T > 0$. Define the error signals as two new states, a scalar variable z_1 and a vector $\boldsymbol{z}_2$:

$$\begin{aligned} z_1 &= \chi - \chi_d = \psi - \psi_d \\ \boldsymbol{z}_2 &= [z_{2,1}, z_{2,2}, z_{2,3}]^T = [u - u_d, v - \alpha_2, r - \alpha_3]^T = \boldsymbol{\nu} - \boldsymbol{\alpha} \end{aligned} \tag{7.50}$$

where χ_d and its derivatives r_d and $\dot{r}_d$ is provided by filtering the LOS angle through a mass-spring-damper reference model. The desired speed $u_d \in \mathscr{L}_\infty$ and $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \alpha_3]^T \in \mathbb{R}$ is a vector of stabilizing functions to be specified later. Next, let

$$\boldsymbol{h} = [0, 0, 1]^T \tag{7.51}$$

such that

$$\begin{aligned} \dot{z}_1 &= r - r_d = \boldsymbol{h}^T \boldsymbol{\nu} - r_d \\ &= \alpha_3 + \boldsymbol{h}^T \boldsymbol{z}_2 - r_d \end{aligned} \tag{7.52}$$

where $r_d = \dot{\psi}_d$

$$\begin{aligned} \boldsymbol{M}\dot{\boldsymbol{z}}_2 &= \boldsymbol{M}\boldsymbol{\nu} - \boldsymbol{M}\dot{\boldsymbol{\alpha}} \\ &= \boldsymbol{\tau} - \boldsymbol{N}\boldsymbol{\nu} - \boldsymbol{M}\dot{\boldsymbol{\alpha}} \end{aligned} \tag{7.53}$$

We want to choose a *Lyapunov function candidate* (LFC):

$$V(z_1, \boldsymbol{z}_2) = \frac{1}{2} z_1^2 + \frac{1}{2} \boldsymbol{z}_2^T \boldsymbol{M} \boldsymbol{z}_2 \tag{7.54}$$

We see that the LFC in Equation 7.54 has that $V(0,0) = 0$, $V(z_1, \boldsymbol{z}_2) > 0, \forall \boldsymbol{z} \neq 0$ and $\|\boldsymbol{z}\| \to \infty$. Differentiating V along the trajectories of z_1 and $\boldsymbol{z}_2$ yields

$$\begin{aligned} \dot{V} &= z_1 \dot{z}_1 + \boldsymbol{z}_2^T \boldsymbol{M} \dot{\boldsymbol{z}}_2 \\ &= z_1(\alpha_3 + \boldsymbol{h}^T \boldsymbol{z}_2 - r_d) + \boldsymbol{z}_2^T(\boldsymbol{\tau} - \boldsymbol{N}\boldsymbol{\nu} - \boldsymbol{M}\dot{\boldsymbol{\alpha}}) \end{aligned} \tag{7.55}$$

The design objective is to render the equilibrium point *uniformly globally asymptotically stable* (UGAS). To do that we have to choose a proper $\boldsymbol{\tau}$ that ensure this stability property. While α_1 and α_2 are yet to be defined, choosing the stabilizing function (virtual control function) α_3 as

$$\alpha_3 = -c z_1 + r_d \tag{7.56}$$

Where c is a fixed heading gain. inserting α_3 gives:

$$\begin{aligned} \dot{V} &= -c z_1^2 + z_1 \boldsymbol{h}^T \boldsymbol{z}_2 + \boldsymbol{z}_2^T(\boldsymbol{\tau} - \boldsymbol{N}\boldsymbol{\nu} - \boldsymbol{M}\dot{\boldsymbol{\alpha}}) \\ &= -c z_1^2 + \boldsymbol{z}_2^T(\boldsymbol{h} z_1 + \tau - \boldsymbol{N}\boldsymbol{\nu} - \boldsymbol{M}\dot{\alpha}) \end{aligned} \tag{7.57}$$

Suppose we can assign

$$\boldsymbol{\tau} = \begin{bmatrix} \tau_1 \\ \tau_2 \\ \tau_3 \end{bmatrix} = \boldsymbol{M}\dot{\boldsymbol{\alpha}} + \boldsymbol{N}\boldsymbol{\nu} - \boldsymbol{K}\boldsymbol{z}_2 - \boldsymbol{h}z_1 \tag{7.58}$$

where $\boldsymbol{K} = diag[k_1, k_2, k_3] > 0$. This will result in

$$\dot{V} = -cz_1^2 - \boldsymbol{z}_2^T \boldsymbol{K}\boldsymbol{z}_2 < 0, \forall \neq 0, \boldsymbol{z}_2 \neq 0 \tag{7.59}$$

so, we also have that the LFC is negative definite, and the state variables (error signals) $(z_1, \boldsymbol{z}_2)$ is bounded and converges to zero.

Investigating Stability

The closed-loop equations become:

$$\begin{aligned} \Sigma_2 : \begin{bmatrix} \dot{z}_1 \\ \dot{\boldsymbol{z}}_2 \end{bmatrix} &= \begin{bmatrix} -c & h^T \\ -\boldsymbol{M}^{-1}h & -\boldsymbol{M}^{-1}\boldsymbol{K} \end{bmatrix} \begin{bmatrix} z_1 \\ \boldsymbol{z}_2 \end{bmatrix} \\ \Sigma_1 : m_{22}\dot{\alpha}_2 &= -n_{22}\alpha_2 + \gamma(z_1, \boldsymbol{z}_2, r_d, \dot{r}_d) \end{aligned} \tag{7.60}$$

Since the trajectory tracking problem does depend on t explicitly, the nonlinear nonautonomous system will based on the Lyapunov stability arguments in Equation 7.54 and 7.59, will deem the equilibrium point $(z_1, \boldsymbol{z}_2) = (0, 0)$ of the z-subsystem *uniformly globally asymptotically stable* (UGAS).

Moreover, the unforced system α_2 - subsystem ($\gamma = 0$) is clearly exponentially stable. Since $(z_1, \boldsymbol{z}_2) \in \mathscr{L}_\infty$ and $(r_d, \dot{r}_d) \in \mathscr{L}_\infty$:

$$\begin{aligned} (z_1, \boldsymbol{z}_2) \in \mathscr{L}_\infty \Leftrightarrow \|\boldsymbol{z}\|_{\mathscr{L}_\infty} = \sup_{t\geq 0} \|\boldsymbol{z}(t)\| < \infty \\ (r_d, \dot{r}_d) \in \mathscr{L}_\infty \Leftrightarrow \|(r_d, \dot{r}_d)\|_{\mathscr{L}_\infty} = \sup_{t\geq 0} \|(r_d, \dot{r}_d)(t)\| < \infty \end{aligned} \tag{7.61}$$

then $\gamma(z_1, \boldsymbol{z}_2, r_d, \dot{r}_d) \in \mathscr{L}_\infty$. This implies that the α_2-subsystem is input-to-state stable (ISS) from γ to α_2. Since Σ_2 is UGAS and Σ_1 is ISS, then by Lemma 4.7 in [38, Khalil, 2002] the cascaded system consisting of the two is UGAS as well. The system stability properties of the cascaded system can be seen by Figure 7.8.

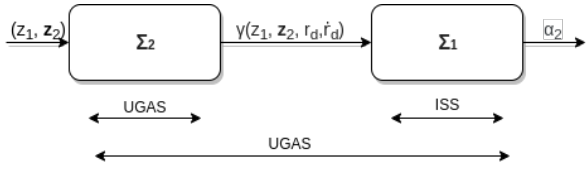


Figure 7.8: Stability properties for cascaded system

Choice of Stabilizing Functions

Writing out the control laws, we'll have:

$$\begin{aligned}\tau_1 &= m_{11}\dot{\alpha}_1 + n_{11}u - k_1(u-\alpha_1)\\ \tau_2 &= m_{22}\dot{\alpha}_2 + m_{23}\dot{\alpha}_3 + n_{22}v + n_{23}r - k_2(v-\alpha_2)\\ \tau_3 &= m_{32}\dot{\alpha}_2 + m_{33}\dot{\alpha}_3 + n_{32}v + n_{33}r - k_3(r-\alpha_3) - z_1\end{aligned} \tag{7.62}$$

Choosing $\alpha_1 = u_d$ solves the dynamic task, so that the closed-loop surge dynamics becomes:

$$m_{11}(\dot{u} - \dot{u}_d) + k_1(u - u_d) = 0 \tag{7.63}$$

The remaining unforced system sway dynamics ($\tau_2 = 0$) results in a dynamic equality constraint:

$$m_{22}\dot{\alpha}_2 + m_{23}\dot{\alpha}_3 + n_{22}v + n_{23}r - k_2(v-\alpha_2) = 0 \tag{7.64}$$

This equation is recognized as the *internal dynamics* and we need to check if the closed loop system in 3DOF based only two controls is stable.

Substituting $\dot{\alpha}_3 = c^2 z_1 - c_{2,3} + \dot{r}_d$, $v = \alpha_2 + z_{2,2}$, $r = \alpha_3(z_1, r_d) + z_{2,3}$, into Equation 7.64, gives:

$$m_{22}\dot{\alpha}_2 = -n_{22}\alpha_2 + \gamma(z_1, z_2, r_d, \dot{r}_d) \tag{7.65}$$

where:

$$\gamma = (n_{23}c - m_{23}c^2)z_1 + (k_2 - n_{22})z_{2,2} + (m_{23}c - n_{23})z_{2,3} - m_{23}\dot{r}_d - n_{23}r_d \tag{7.66}$$

The variable α_2 becomes a dynamic state of the controller according to Equation 7.65. Furthermore, $n_{22} > 0$ implies that Equation 7.65 is a stable differential equation driven by the converging error signal (z_1, z_2) and the bounded reference signal $(r_d, \dot{r}_d)$. Since $z_{2,2}(t) \to 0$, we get that $|\alpha_2(t) - v(t)| \to 0$ as $t \to \infty$. The reference signals $u_d, \dot{u}_d, \psi_d, r_d, \dot{r}_d$ are provided by the LOS guidance system combined with a reference model.

While the stabilizing function α_2 can be found through numerical integration of the internal dynamics in Equation 7.64, we choose it to be $\alpha_2 = 0$. This essentially means that we want the error signal $z_{2,2} = v - \alpha_2 = v$ to track the sway velocity towards zero.

So, summed up this implies that the stabilizing functions and there derivatives are set to be:

$$\begin{aligned}\alpha_1 &= u_d \\ \alpha_2 &= 0 \\ \alpha_3 &= -cz_1 + r_d\end{aligned} \tag{7.67}$$

$$\begin{aligned}\dot{\alpha}_1 &= \dot{u}_d \\ \dot{\alpha}_2 &= 0 \\ \dot{\alpha}_3 &= -c\dot{z}_1 + \dot{r}_d \\ &= -c(r - r_d) + \dot{r}_d\end{aligned} \tag{7.68}$$

and this will then result in the control laws:

$$\begin{aligned}\tau_1 &= m_{11}\dot{\alpha}_1 + n_{11}u - k_1(u - \alpha_1) \\ \tau_2 &= m_{23}\dot{\alpha}_3 + n_{22}v + n_{23}r - k_2 v \\ \tau_3 &= m_{33}\dot{\alpha}_3 + n_{32}v + n_{33}r - k_3(r - \alpha_3) - z_1\end{aligned} \tag{7.69}$$

Depth Hold

The path-following controller for this case study tracks paths projected in the XY plane. The depth tracking is uncoupled from the rest of the controller and uses a 1-D PID controller. The tuning gains for this controller is found in Appendix C.2.2.

7.5.3 Guidance System Design

LOS Geometric Task

We want to force the AUV to converge to a desired path with time. This is done by forcing the course angle, χ, to converge to:

$$\chi_d = atan2(y_{los} - y, x_{los} - x) \tag{7.70}$$

where the LOS position $\boldsymbol{p}_{los} = [x_{los}, y_{los}]^T$ is the point along the path to which the craft should be pointed. Notice that $\psi = \chi - \beta$ and $\psi_d = \chi_d - \beta$. The sideslip angle, β, is necessary for compensating for currents. In this project the influence of currents will be minuscule, so $\psi = \chi$ and $\tilde{\psi} = \tilde{\chi}$.

LOS Dynamic Task

We want to force the speed u to converge to a desired speed u_d according to:

$$\lim_{t \to \infty} [u(t) - u_d(t)] = 0 \tag{7.71}$$

where u_d is the desired speed composed along the body-fixed x axis.

Waypoint Switching using Sphere of Acceptance

For this controller we chose to include setpoint switching logic . A new setpoint is released whenever the vehicle is within the boundaries of the *sphere of acceptance* given by

$$[x_{k+1}]^2 + [y_{k+1}]^2 + [x_{k+1}]^2 \leq R_{k+1}^2 \tag{7.72}$$

The geometrical interpretation of the sphere of acceptance can also be seen from Figure 7.1.

Reference Model using Tustin's Method

For solving the geometric task, the signals ψ_d, r_d and $\dot{r}_d$ is required by the controller. To solve the dynamic task the signals u_d and $\dot{u}_d$ are required. To provide these signals, a mass-spring-damper reference model is implemented using Tustin's method. The implementation follows the same procedure as from subsection 7.4.4 in the previous case study, but where the MIMO reference model take the input (u_r, ψ_r) and then outputs $(u_d, \dot{u}_d, \psi_d, r_d, \dot{r}_d)$.

This will generate the necessary signals as well as smoothing the discontinuous waypoint switching to prevent rapid changes in the desired yaw that is sent to the controller. However, since the *atan2* is discontinous at the $(-\pi, \pi)$-junction, the reference model cannot be applied directly. This means that we have to wrap the heading first, input the reference model, and then un-wrap the heading thereafter. The wrapping procedure can be seen in Figure 7.9. This design reroutes the heading, making sure that the vehicle always will turn the shortest distance to achieve its desired heading. This design will also make sure that the controller will not stall at the discontinuous junction.

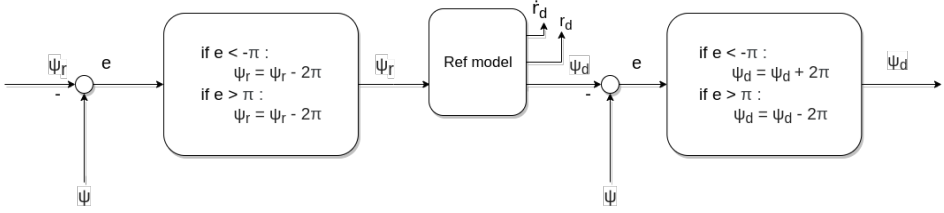


Figure 7.9: Mass spring damper reference model with wrapping of heading

The issue of discontinuous orientation is a nontrivial task that is subject to topological disruptions, i.e splitting angles evolving on a compact manifold into a defined range (as disrupting $\mathcal{S}^1$ to $[-\pi, \pi]$). The issue can be handled with wrapping and un-wrapping as we did in this case, or it can handled using a quaternion representation. The quaternion elements vary continuously over the unit sphere in $\mathbb{R}^4$, (denoted by $\mathbb{S}^3$) as the orientation changes, avoiding discontinuous jumps (inherent to three-dimensional parameterizations).

The issue can also handled by representing the angles as points on the unit circle and control these points towards the desired point as presented in [79, Haug, 2019].

7.5.4 Software Implementation

The urls linking to the source code can be found in Appendix A.

Open-Source Packages, Libraries and Standards

The backstepping controller is implemented using Python and ROS. The only python library that has been used for implementation is *NumPy*. NumPy is a fundamental package for scientific computing with Python, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Controller Tuning

The tuning of the backstepping controller is mainly based on the identified system model from the case study: *Case Study: Estimating Rigid-Body Kinetics, Hydrostatics and Hydrodynamics for the Manta AUV using Empirical Equation Estimates* from Chapter 4.

The final values of the tuning can be found in Appendix C.2.2.

7.5.5 Simulation Results and Discussion

The simulation is performed in the Gazebo simulation environment created in the case study from Chapter 5, which is a 1:1 scale version of the marine cybernetics lab at the Department of Marine Technology, NTNU. For vehicle localization the AUV take use of the extended Kalman filter (EKF) created in the case study from Section 6.4.

In Figure 7.10 you see a 3-plot of the recorded run. The stippled black lines show the desired path the vehicle tries to follow, while the red solid line shows the estimated position from the extended Kalman Filter. The large blue sphere is the sphere of acceptance, while the solid blue dots are the actual target waypoints.

The sphere of acceptance for each waypoint is $0.2m$. As you can see, the 1-D PID controller is succesful in tracking the desired depth of $-0.5m$, while the backstepping controller in surge, sway and yaw is almost perfect in tracking the desired path.

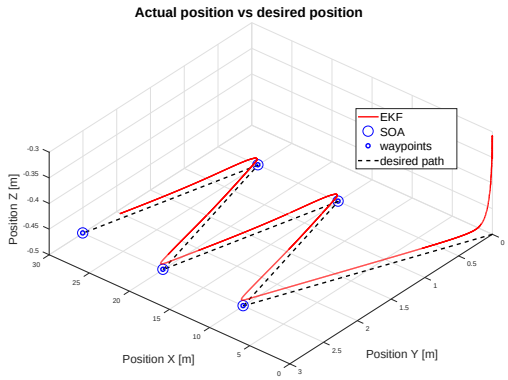


Figure 7.10: Path-following in Gazebo with line-of-sight and backstepping controller. EKF vs desired states with plotted waypoints and sphere of acceptance (SOA). Scan the QR code to see the real-time simulation on youtube

In the upper plot of Figure 7.11 you have the estimated attitude and in the bottom plot you have the desired attitude provided by the LOS guidance node. As you can see the backstepping controller is successful in achieving the tracking the attitude with high precision.

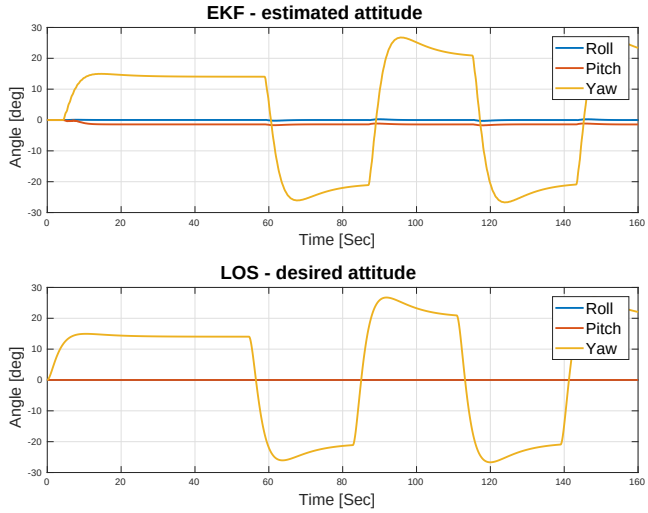


Figure 7.11: Line-of-sight guidance desired attitude

On the left side in Figure 7.12 you can see the desired linear velocities from the guidance module in the bottom plot and the estimates of the actual estimates of the produced linear velocities on the top. The controller is able to track the desired surge velocity of $0.2[m/s]$

with high precision. The controller is also able to have the velocities in sway and heave remain zero as desired by the reference model and LOS guidance node.

From the plots on the right side in Figure 7.12, you see the desired angular velocities plotted on the bottom and the estimates of the actual produced angular velocities in the top plot. The plots show that controller achieves the desired angular velocities with only a few seconds delay due to the body inertia.

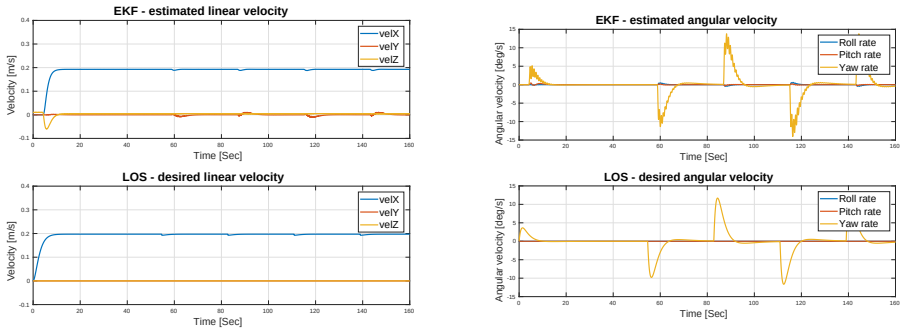


Figure 7.12: Desired linear and angular velocities vs actual estimates

Lastly, on the left in Figure 7.13 you have the desired forces in body frame in the bottom plot and the estimates of actual linear velocities produced in the top plot. In comparison, it can be seen that a desired velocity in surge (blue line) of $0.2[m/s]$ is maintained by applying a constant force in surge equal of about $4.8[N]$. Also, the sudden changes in the desire yaw force (red line) in the bottom plot is due to the vehicle applying force in sway to make sharp turns.

On the right side in Figure 7.13 you have the desired torques in body frame in the bottom plot and the estimates of actual angular velocities produced in the top plot. As you can see, the plots suggest that the applied torques and actual produced angular velocities are proportional with little phase delay.

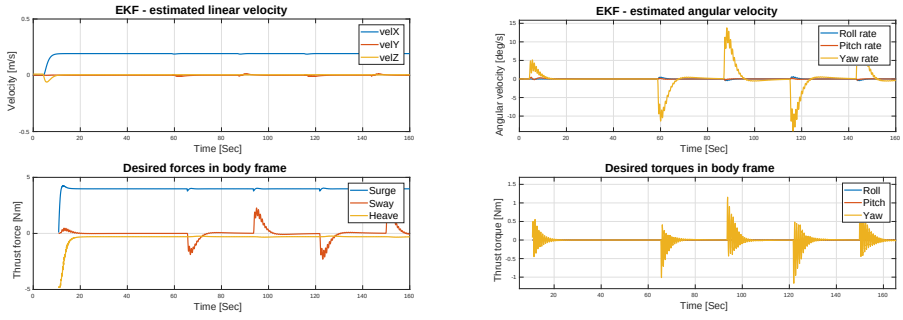


Figure 7.13: Desired torques and forces in body frame vs actual linear and angular velocities

7.5.6 Conclusion of Case Study

To conclude the case study, it is clear that the path-following backstepping controller is successful in tracking a desired path by controlling the velocities in surge, sway and yaw. As earlier mentioned the backstepping controller is a model-based type of control, and thus it relies on an accurate mathematical representation of the vehicle dynamics. The results has thus also made it clear that backstepping controller is accurate in its mathematical representation of the vehicle, as the performance of the controller rely heavily on this representation being accurate.

7.6 Chapter Summary

The aim of this chapter was taking upon the aspect of designing and implementing necessary guidance and motion control algorithms for enabling both long-distance transit using path-following controllers, as well as dynamic positioning and accurate low-speed set-point regulation smaller sectors. The chapter started of explaining some basic theory into guidance and control strategies, and then some of the advantages and disadvantages of common nonlinear controllers. The chapter ended of with two case studies that explain the design and implementation of both a nonlinear PID controller and a nonlinear backstepping controller. The results in simulations are encouraging and demonstrate robustness and precision in both controllers.

The next chapter will attempt to incorporate the two derived controllers into the Manta v1 software architecture. The controllers will ultimately play an important part of the low-level control capabilities of the vehicle.

CHAPTER 8

MISSION CONTROL FOR AUTONOMOUS UNDERWATER VEHICLES

The goal of this chapter is to ensure an software architecture that is capable of long-term and deliberative agent behaviors through careful coordination and execution of mission goals derived at various layers of abstraction. The mission goals must advance from a high-level abstraction communicated by a human operator down to the lower levels of abstraction through navigation, guidance and motion controllers. Constructing such behaviors are not a trivial task and depends not only on the mission control itself, but also software architecture as whole and the flow of information between the distributed components that make of the architecture. This chapter will describe how one go about designing and implementing the mission control. The chapter will finish up with a case study to demonstrating mission control in a simulated environment.

8.1 History of Mission Control

In the past 20 years, a lot of different approaches to intelligent mission planning and control have been proposed in the literature. Some works are based on the definition and design of vehicles primitives (basic vehicle maneuvers) or object-oriented frameworks such as in [14, Champeau (2000)], [59, Newman (2002)] or *distributed networked systems* as [49, McPhail and Pebody (1997)] and [69, Perrett and Pebody (1997)]. Hierarchical and intelligent architectures are described in [44, Marco (1996)] and [97, Turner (1995)]. Another approach for mission control system development is based on the design of *finite state machines* and *Petri nets* able to describe and manage in a simple way parallel execution of actions (asynchronous) inside a mission plan, examples are given in [64, Oliveira (1998)], [20, Poole (2000)] and [15, Chang (2004)]. Other works regarding development

of mission plan languages have been investigated in [39, Yuh (2003)] and [67, Palomeras (2007)].

8.2 Basic Components of Mission Control

As mentioned in Chapter 3, the agent displays the core decision-making and intelligent capabilities of the system. This section details the essential components of the Manta v1 mission control. The mission control takes as input a mission plan stated through a *.yaml* or *.py* file (optional). The *task manager* module will interpret the syntax and semenatics of the mission plan in which mission statements are made into to task schedules and goals. These are then processed through either of the two *task controllers* that again interpret and render the mission into states and transitions for the state machine or into a tree of nodes for the behavior tree. The mission control for Manta v1 is seen in Figure 8.1.

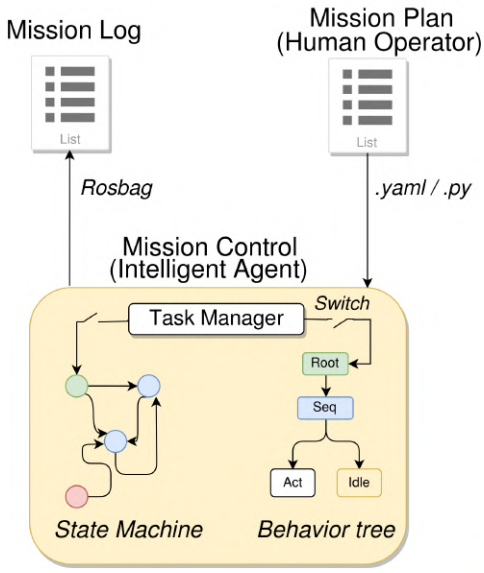


Figure 8.1: The Manta v1 mission control

In the next subsections we will go through the three most important components of the mission control.

Mission Plan: Human-Robot Interaction

The mission plan is the process of determining what a single (or a collection of) robot(s) should do to achieve the goals of the mission. The mission plan is communicated by a human operator and provides aprori details of a mission such a maps, waypoints, objects

of interest and emergency behaviors. On mission boot the mission plan provided as either a *.py* or *.yaml file*. The data used for creating i.e maps, may be retrieved from a geographic information system (GIS), operations databases, logs, aerial photos, or alternatively input directly by the user.

If possible, one should be abstracting the mission plan away from the details of the path planners, map sharing modules, local navigators, motion controllers etc. This often allows for multi-functional mission plans that extend over a wide variety of missions [11].

Task Manager: Scheduling and Goals

The use of a *task manager* (TM) in the mission control interface serves as a task-based communication and cooperation point between the human operators mission plan and the intelligent agent. The TM tracks, organizes and manipulates the cluster of high-level task statements provided by the mission plan (see Figure 8.1 into a 'plannable form' in terms of *schedules* and *goals*. Schedules glue together tasks in meaningful ways, often resulting in macros of intelligent behavior. Sometimes the agent might require multiple schedules to be executed in sequence in order to achieve an even more long-term behavior and that's where goals come in handy. In each case when a goal is active it dictates that upon completing a schedule, another one needs to be selected that will help that goal to be realized.

The human operator must trust that the intelligent agent will (a) pick up its tasks as established in the working agreement, and (b) can execute them well enough to fulfill mission requirements [35]. For the majority of real robotic applications, the operational environment is not static or structured. A priori information may be incorrect, commands may not be executed correctly, and perception information may contain noise; in general, unexpected events may occur frequently. To be successful and robust in real world operations, robotic systems must accommodate the fact that the world is dynamic and changing, and may not be amenable to a simple environmental model. Therefore, whatever plan a robot may be executing must be allowed to change regularly as new information about the world and the task arrives. In the context of a TM, it means the real-time manipulation of the task schedules and goals through task operations such as: *update, delete, append*.

Task Controller: Coordination and Execution

The *task controller* (TC) in the mission control interface serves as a coordinator between robot percept, mission plan, and task execution. There are a number of ways to program a so-called task controller for robots, with two of the most common ones originating from the game industry. Traditionally, most task controllers make use of *finite state machines* (FSM), while a relatively new approach take use of *behavior tree* (BT) [71]. Perhaps not surprisingly, programming an animated *non-player characters* (NPCs) in a game to behave in a realistic manner is not unlike programming a robot to carry out a series of tasks. Both must handle a set of unpredictable inputs and choose from a variety of behaviors appropriate to the task at hand as well as the current situation. We will get into the details of task controllers in Section 8.4.

Mission Log: Logging and Debugging

At last the mission log records desirable data points that one would like to collect for data about the environment, mission and internal sensors that can later be used for mission debugging and planning of future missions.

8.3 Basic Properties of your Intelligent Agent

8.3.1 Properties of Task Environment

The range of task environments that might arise in robotics is obviously vast. We can, however, identify a fairly small number of dimensions along which task environments can be categorized. These dimensions determine, to a large extent, the appropriate agent design and the applicability of each of the principal families of techniques for agent implementation [78].

Fully Observable vs. Partially Observable vs. Unobservable

If an agent's sensors gives access to the complete state space of the environment at each point in time, then the task environment is characterized as *fully observable*. A fully observable environment is convenient because the agent can detect all aspects to the next choice of action. An environment is *partially observable* if it doesn't have the sensors to measure all states of the environment relevant for the next course of action. An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data. If an agent has no sensors at all then the environment is *unobservable*.

Deterministic vs. Stochastic vs. Non-Deterministic

If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is *deterministic*; otherwise, it is *stochastic*. In principle, an agent need not worry about uncertainty in a fully observable, deterministic environment. The word "stochastic" generally implies that uncertainty about outcomes is quantified in terms of probabilities; a *non-deterministic* environment is one in which actions are characterized by their possible outcomes, but no probabilities are attached to them.

Static vs Dynamic

If the environment can change while an agent is deliberating, then we say that the environment is dynamic for that agent; otherwise it's static. Static environments are easy to deal

with because the agent does not need to look at the world while deciding its next action, nor need it worry about the passage of time.

8.3.2 Properties of your Task Controller in the Task Environment

An advanced mission control system has to fulfill some functionalities for the aim of increasing the level of autonomy and safety the overall system [6]. The overall task controller is often called the task executive and most such executives are expected to include at least the following key features [32]:

Task Failure: Repeated Attempts

Let us say the you have a robot gripper arm and a stereo vision camera. Your *goal* is to pick-up up the ball a drop it in a bin. The *task schedule* contains the tasks: (detect ball, open gripper, plan arm trajectory, extend arm, close gripper, detect bin, plan arm trajectory, extend arm, open gripper). The task controller executes all the tasks in a sequential order, but when the robot tries to close it gripper it misses and the ball rolls away. The task controller should be able to re-try tasks or complete task schedules if it fails. In the event of the robot never being able to accomplish its goal, we usually put an time or attempt limit, to avoid the task controller getting stuck in a loop.

Task Priorities: Pause, Resume and Abort proceedures

A lower priority task should be **preempted** (meaning, one task taking over in place of another) if a higher-priority task requires the same resources (e.g. drive motors). It is often desirable to pause a currently running task (or tasks) when a higher-priority task is given control and then **resume** the preempted task(s) when the executive yields back control. In the presence of hazardous events the AUV task controller should have the opportunity to **abort** the whole mission and re-surface.

Task Hierarchy: Sub-Mission Management

From a mission plan perspective it is very important to have the possibility of breaking complex mission executions into smaller so-called sub-mission modules [6]. For instance, a high level task called recharge might consist of three sub-tasks: *navigate to the docking station, dock the robot, and charge the battery*. Similarly, the docking task could be broken down into: *align with beacon; drive forward; stop when docked*.

This functionality is also called a *modular approach* in programming, and allows for the definition of emergency procedures that can be used by different missions.

Task Concurrency: Simultaneous Task Execution

Multiple tasks can run in parallel. For example, both the navigation task (getting to the next waypoint), collision avoidance and battery monitoring at the same time.

Task Tracking: Robot Working Memory

Working memory is a cognitive system with a limited capacity that is responsible for temporarily holding information available for processing. Working memory is important for reasoning and the guidance of decision-making and behavior, but is then discarded once it has served its purpose [70]. Drawing a parallel to human biology, working memory is what allows you to remember what phone number the operator told you, just long enough to dial it.

The simplest forms of working memory that we utilize in programming are *boolean* parameters to tell if an event has happened or not, counters to tell how many times an event has happened and internal clocks (timers) to keep track of occurrence of events through time. Working memory is a particular popular research field within *cognitive architectures* [78].

8.4 Current State of the Art Task Controller for Agent Deliberation

Many software systems are event-driven, meaning that they continuously wait for the occurrence of some external or internal event such as a mouse click, a button press, a time tick, or an arrival of a data packet. After recognizing the event, such systems react by performing the appropriate computation that may include manipulating the hardware or generating ”soft” events that trigger other internal software components. Once the event handling is complete, the system goes back to waiting for the next event.

The following methods presented; *finite state machine* and *behavior tree* are both considered the current state of the art agent deliberation methods. There are many other methods that also tackles the idea of agent deliberation such as *decision trees*, *subsumption architectures* and *Teleo-reactive programs* [18]. These methods will not be considered, as they are not considered current state of the art nor industry standard.

8.4.1 Finite State Machine

A *finite state machine* (FSM) or *finite-state automaton* (FSA, plural: automata) or simply a state machine - is a model commonly used to simulate simple sequential logic. It’s largely derived from two bodies of work by George H. Mealy and Professor Edward F. Moore in 1955 and 1956 respectively. The system is a collection of one or more pre-defined states. When modelling AI behavior in a game a state will represent a specific behavior

that a character or other system in the game should execute, this can be standing idle, attacking the player, moving to a point in the world, interacting with an object, whatever the designer sees fit. Often this means we're handling various aspects of gameplay systems such as animation, sound and decision making for whatever system the finite state machine is controlling. The state machine will continue to hold the current state as active until it receives an input that it recognizes, after which point it will then transition to another state within the system (See Figure 8.2).

Figure 8.2: A state transition triggered by an event

As a designer you can decide what inputs a state receives are valid for a transition to occur, as well as what states it will transition to based on this information. We say state(s) in plural, because you can decide to have a state transition to one or more states in the event it reads a given input. This results in either a *deterministic* FSM, which is where a state reads an input and can only transition to one other state, or a *non-deterministic* FSM, meaning that if that input occurs, the system could transition to a number of different states. The benefit of this approach, is that it means you can define multiple states that when implemented can lace into a much more nuanced behavior. The two definitions are seen in Figure 8.3

Figure 8.3: Non-deterministic vs deterministic finite state machine.

A good example of the non-deterministic FSM is the six-legged (hexapod) insect robot, shown in Figure 8.4, designed for walking through rough terrain. The robot's sensors are inadequate to obtain models of the terrain for path planning. Moreover, even if we added sufficiently accurate sensors, the twelve degrees of freedom (two for each leg) would render the resulting path planning problem computationally intractable [78]. For the gait (movement pattern of limbs), a statically stable approach is to first move the right front, right rear, and left center legs forward (keeping the other three fixed), and then move the other three. This gait works well on flat terrain. On rugged terrain, obstacles may prevent a leg from swinging forward. This problem can be overcome by a remarkably simple control rule for each of the legs: *when a leg's forward motion is blocked, simply*

retract it, lift it higher, and try again. The resulting controller is shown in Figure 8.4 as a *non-deterministic* finite state machine with four internal states (indexed s_1 through s_4). This FSM constitutes what is also referred to as a *non-deterministic* and *augmented finite state machine* (AFSM) [78], where the augmentation refers to the use of internal timers (clocks) that control the time it takes to traverse an arc (the line connecting the states). In other words, these are finite state machines augmented with timers which can be set to initiate a state change after some fixed time period has passed.

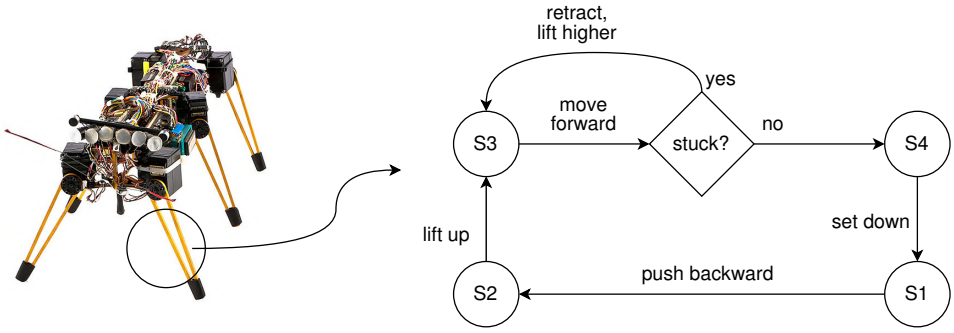
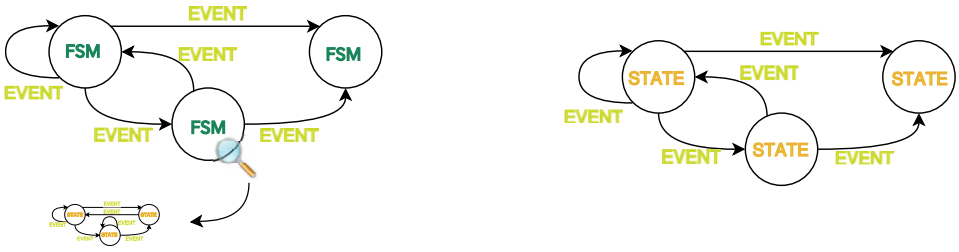


Figure 8.4: Genghis, a hexapod insect robot. Each leg has an augmented finite state machine (AFSM) for the control of a single leg. Notice that this AFSM reacts to sensor feedback.: if a leg is stuck during the forward swinging phase, it will be lifted increasingly higher. *Courtesy: [78, Russel & Norvig, 2016]*

In our example, we might begin with AFSMs for individual legs, followed by an AFSM for coordinating multiple legs. On top of this, we might implement higher-level behaviors such as collision avoidance, which might involve backing up and turning. In practice, the intricate interplay between dozens of interacting AFSMs (and the environment) is beyond what most human programmers can comprehend. Now one approach to resolving this is to use *hierarchical finite state machine* (HFSM) or just referred to as *hierarchical state machine* (HSM). In this instance you can group states together such that a transition can either go to a specific state as usual, but it can also go to a collection of states that have been built to transition in specific ways. In essence, you could go so far as to build a state machine that effectively transitions between more modular state machines. This will allow for more carefully managing the operation of specific behaviors and how entire subsets of behavior move between one another. The concept can be seen in Figure 8.5.



hierarchical state machine (HSM) *finite state machine (FSM)*

Figure 8.5: Hierichal state machine vs finite state machine

But the things is that these intelligent agents cannot work just in a purely reactive capacity. If there are so many tasks that can be executed, the systems needs to know what transitions it can make in the state machine, as well as which tasks make sense to run at a given point. So the intelligent agent architecture becomes more deliberative: meaning that it needs to work through the finite state machine and transition from state to state in practical and interesting ways that enables for a more deliberate and long-term behavior to be established. There are two ways that we can do this, through *schedules* and through *goals*, which are both handled through the task manager described earlier on.

Pros and Cons

Finite state machines help create intelligent agents that can not only respond to their own internal memory - given they make decisions about states to transition to based on information stored internally - but can also react to events happening in the world and be versatile to change often driven by the player. As a result, for many years FSM were the de-facto standard of how to build AI in game until arguably the mid to late 2000's

The concept of a FSM is important in event-driven programming because it makes the event handling explicitly dependent on both the event-type and on the state of the system. When used correctly, the introduction of a state machine can significantly cut down the number of execution paths through the code, simplify the conditions tested at each branching point, and simplify the switching between different modes of execution [80]. On the contrary, using event-driven programming without an underlying FSM model can lead programmers to produce error prone, difficult to extend and excessively complex application code (meaning deeply nested if-else or switch-case statements) [81].

In short, the benefits and drawbacks of the FSMs can be condensed into the table:

Pros	Cons
Modular and intuitive structure	Does not scale well as the number of behaviors and transitions increases
Ease of implementation	Highly susceptible to human design errors
Behavior inheritance allows sub-states to inherit behaviors from the superstate	

8.4.2 Behavior Tree

In this section, you are presented one of the core AI technologies that is now pervasive throughout much of AAA video games industry: the *behavior tree* (BT). While the wider world is caught in the thrall of machine learning, the games industry is heavily reliant on BT techniques. BT is a data structure which we set the rules of how certain behaviors can occur and the order in which they would execute. The term 'tree' is denoted given that is follows the traditional rules of the tree data structure in computer science: meaning as a series of nodes that start at the root - which is one at the top - which is then linked to other nodes by connections. In computer science terms this is a *directed acyclic graph*, meaning it's a collection of nodes that are connected to each other but in each case there is a fixed direction and that it doesn't loop back or repeat itself. The logic of the tree works its way down from the top, to a specific set of one or more nodes where a specific behavior is going to be executed. A node that has been linked to by another one above it calls that node it's parent, and any nodes it links to below are called its children as seen by Figure 8.6.

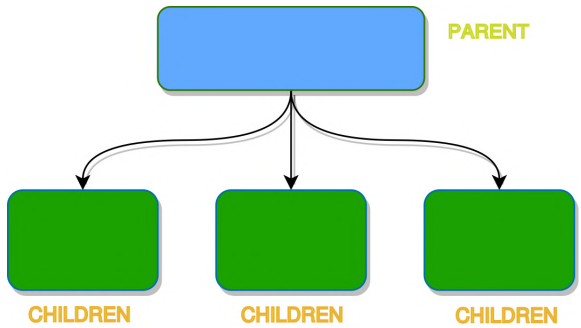


Figure 8.6: Direct acyclic graph parent and child.

So, with the basics down, let's explore how they work and some of the cool features they carry. So traditionally, when behavior trees were first devised, the idea was that every frame of a given game, the system would run an update from the root called a *tick*. To guarantee that every node is visited exactly once, the ticks traverse the tree in a *depth first traverse* (explores as far as possible along each traversal before backtracking) [71]. The

system then traverses down the tree to find the node that is active, rechecking whether other nodes should be active along the way down. However, that's a bit expensive - especially when the tree starts to get bigger. So nowadays the tree will retain a reference to the active node and have that process the tick. As soon a node indicates that it is OK to follow its sub-tree, that tree is explored downward until a problem occurs or a leaf node is reached. If execution is blocked at a particular node in the sub-tree, we back track up the tree and move laterally until a new sub-tree can be executed.

Now how a behavior is selected is based on how it moves down into the tree and the nodes it executes. Nodes are typically found to be one of four types [18]:

Root node: First up there is the *root node*: That's the one at the top, it has no parent, only ever has children and it's where the execution starts from.

Control flow nodes: Next up we have *control flow nodes (or composite nodes)*: these are special nodes that control the flow of how we move through the tree. There are two commonly found control flow node types:

A *selector*, where it decides which child to execute based on some logic in the world. A selector node executes each of its child behaviors in turn until one of them *SUCCEED* or it runs out of sub-tasks. So for example, if a grasping object is too far away, it won't execute the child node that does a the grasping action, but it might decide to advance forward until the object is within grasping reach.

Secondly, there is the *sequence node*: this allows us to execute several child nodes in sequence one after the other. A sequence node runs each of its child tasks in turn until one of them fails or it reaches the last sub-task. This is great if your vehicle is on collision course and you want it to stop, turn away, and take an alternate path. Given, that's arguably three distinct child nodes you're going to execute in sequence.

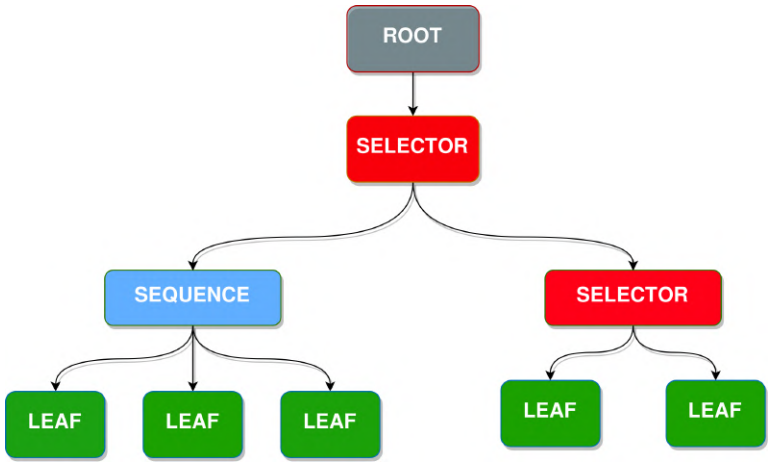


Figure 8.7: Behavior tree with four different kinds of nodes

Decorator node: Thirdly, there are decorator nodes. These take existing logic of a child and manipulate it. This is great for making us repeat actions a certain number of times, or let a selector make a decision based on the inverse of the logic we have given to it and so on.

Leaf node: Lastly, there is the basic node of a tree called a *lead node*: this is where we actually agent behaviors, such as navigating to a fixed position in the world, searching for objects or interacting with an object. It's the opposite of the root, it can only have parents and no children.

There are two commonly known lead node types: the *condition node* and the *action node*. When a depth traversal reaches a condition node it will check a proposition. It returns *SUCCESS* depending if the proposition holds or not. Note that a condition node never returns a status of *RUNNING*. An action node executes a command. It returns *SUCCESS* if the action is correctly completed or *FAILURE* if the action has failed. While the action is ongoing it returns *RUNNING* for that traversal.

So already with this setup you can come up with trees that mash up multiple behaviors. The easiest way to understand the execution of behavior trees is to dive into an example.

Let us pick a scenario where the Manta AUV would patrol an area looking for a particular object of interest that it needs to collect and bring back. The scenario could look something like in.

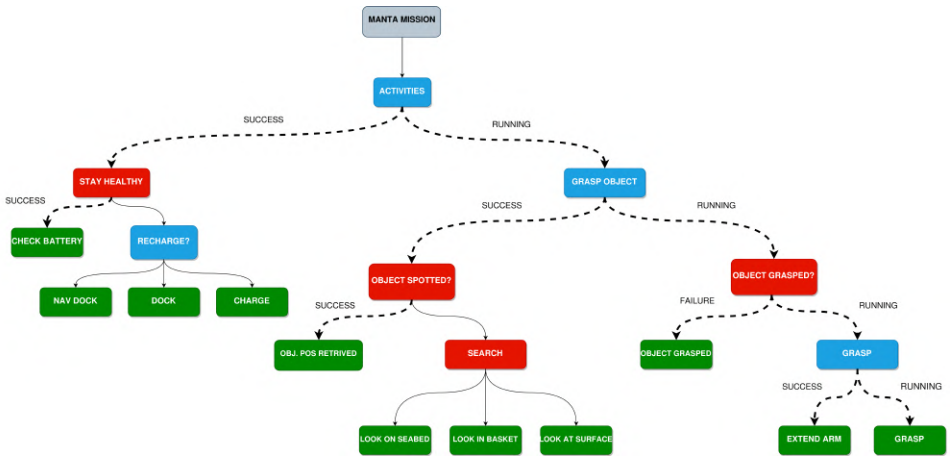


Figure 8.8: A Manta AUV behavior tree for patrolling area and looking for a particular object to pick up.

The nodes in our behavior tree will represent conditions and actions. An way up in the hierarchy can be something relatively abstract as "ACTIVITIES" or more specific like "NAVIGATE TO DOCK". A condition could be "THE BATTERY IS RUNNING LOW" or "THERE IS AN OBSTACLE IN THE WAY". Conditions area really just an action in disguise since the process of assessing the condition can be considered to be an action in

itself. For example, the condition ”THE BATTERY IS RUNNING LOW” can be turned into the task ”CHECK BATTERY”. For this reason, we will be able to use the same programming machinery for both tasks and conditions.

As mentioned, the ticks traversal works in a depth first traversal. We build our behavior tree from the top down beginning with the root node. From Figure 8.8 the thick stippled lines show the current tick traversals running. On the left traversal extending from the ”ACTIVITIES” sequence, we have the most important conditions for us to ensure, which is the safety of the vehicle: ”STAY HEALTHY”. The ”STAY HEALTHY” behavior will include checking the battery level. It could also include checking servos for overheating, watching for excessive current to the drive motors (perhaps the robot is stuck on something) or keeping an eye out for external hazards. The main idea is that before actually engaging in the ”GRASP OBJECT” task we want to make sure that the vehicle is safe. Now once we know the vehicle is safe, we can advance from left to the right in the tree and traverse down the ”GRASP OBJECT” traversal. The particular ticks' traversal shown in Figure 8.8 shows the vehicle having sufficient battery level, that has successfully detect an object of interest, that is currently not in possession of the object and is in the process of extending its grasping arm to reach for the object. This is of course a simplified model of how it would look, but it captures the idea of the behavior tree logic.

Pros and Cons

Now a common question arises is why would you use behavior trees versus the likes of say a finite state machine? So let discuss some common issues that can arise and why many developers adopt the behavior trees as their method of choice.

First up, let's consider how streamlined the flow is. Users can clearly watch the decision-making happen in a top-down fashion from root to leaf. This makes for a more scalable system, even as the tree gets bigger. We can still follow the train of logic that gets us from the root down to a specific behavior. This visual readability is so important as the number of unique paths increases.

Conversely a FSM becomes a nightmare to follow the more complex it becomes and more connections are made between states. For BTs, by streamlining the logic, they're not only more designer friendly, but perhaps more critically - they're easier to debug! The combination of simplified flow and modular behaviors reduces the capacity for error and it also resolves so many of the major issues a designer would have in building behaviors. By comparison, FSMs are typically very code intensive - given the conditions of their execution are often tied to the code of that specific behavior in that state. The third valuable aspect is reusability: we typically build individual behaviors in these trees such that they can be used in different contexts and the logic of the tree will allow us to reuse the same node whenever we need it. Conversely, with the FSMs, many of the states are typically tied to that specific context. So it gets a little uglier the more you try to maintain them. Lastly there's the big issue that impacts all agent systems regardless of scale: the CPU power and memory required to execute (computational overhead). The optimizations of tick processing and event-driven overrides have enabled behavior trees to become more

optimized over time, whereas a FSM still needs to update every typically every frame just to check if it is in the correct state and conduct state transitions accordingly.

While this section is dedicated to explaining how effective behavior trees are, this isn't to say that they're the one and only option for crafting behaviors in your games. If your behavior logic is relatively small scale and simple, finite state machines are more than sufficient for your needs. Ultimately, it comes down to the project you are working on.

In short, the benefits and drawbacks of the BTs can be condensed into the table:

Pros	Cons
Modular and intuitive structure	Computational expense
Streamlined logic	Yet to mature for robotics purposes
Ease of reusability	

8.5 *Case Study: Manta v1 Mission Control using a Non-Deterministic Augmented Hierarchical State Machine for Task Control*

This case study will focus on gluing together the robot localization, robot perception and controllers using an non-deterministic augmented hierarchical state machine for as the task controller in our mission control.

8.5.1 Problem Statement

The design scenario that the robot is set to solve can be seen in Figure 8.9. The simulation scenario is set up in the Gazebo robot simulator and starts of with the AUV spawning in the center of a $40x6x1.5m$ pool. For the robot to successfully complete its mission, it must be able to navigate to the terminal sector of the yellow gate, track the gate and pass through it. After that the AUV must navigate to the green terminal sector and locate the green pole. If the vehicle is able to perform these sets of tasks before draining its battery, then the mission has succeeded.

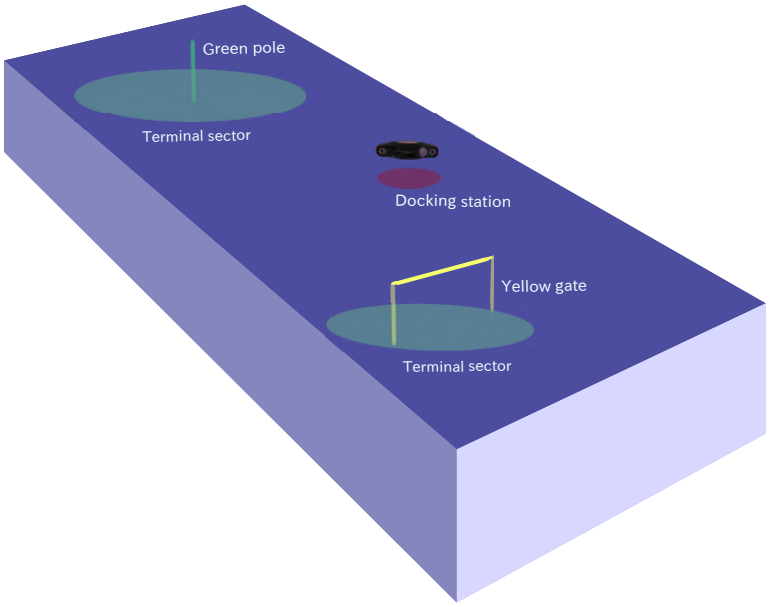


Figure 8.9: A descriptive figure of the mission at task in the Marine Cybernetics Lab (MC-lab) NTNU.

To challenge the the robot a little bit, the vehicle has not been provided with the exact location of either the yellow gate or the green pole. All it has been given are two sectors (denoted terminal sector in the image) with a radius of 4m where the objects of interest should be located. The AUV is neither equipped with a sonar or a stereo camera (only mono camera) and will have to use it object detection capabilities to track down the targets. On top of this there has also been added a battery simulator that will gradually drain the battery from 100% to 0% within 5 minutes. The design scenario is set up such that the AUV will not actually be able to complete the task unless it returns to the docking station and recharges its batteries along the way. However, this information is not provided in advance and the AUV will have to re-plan its mission to achieve its goals whenever the battery level becomes to low.

Other than that, the current mission control assumes that the local navigator on a robot is capable of preventing the vehicle from endangering itself. The workspace is assumed to be relatively open, so that a the robot need not concern itself overmuch with resolving collisions. This is due to the task environment being partially observable and the vehicle should have a sonar or stereo cameras to be able to predict collisions.

8.5.2 Mission Plan

The mission plan is provided through a *mission_plan.py* script. The information provided is:

- The actual depth of the pool: $1.5m$
- The transit speed when navigating: $0.3m/s$
- The location and size of the terminal sectors (search areas): (x_k,y_k) $\pm 4m$ radius.
- A navigation timeout if vehicle is not able to reach its target area: $60seconds$
- The searching speed (when searching for objects): $0.2m/s$
- A *low battery level* threshold: 30% battery capacity
- The exact location and orientation of the docking station

8.5.3 Task Manager

To achieve a particular goal the task manager will assemble a task schedule specifying in what order the tasks will have to be completed. Examples of goals are: *DETECT GATE, GO THROUGH GATE, CHARGE BATTERIES, PICK UP OBJECT*. The *schedule* is a data structure that contains a very simple goal-driven system. It consists of several levels of tasks that can be procedurally combined. *Tasks* are short atomized behaviors that have specific purpose. For example, a typical task for an AUV would be: *IDLE, GO TO SURFACE, GO TO WAYPOINT TURN LEFT, GO FORWARD, TURN ON LIGHTS, OPEN GRIPPER*. The tasks are defined as enumerations in an *ordered dictionary* container data type. A dictionary is a key-value pair, optimized to retrieve values when the key is known.

Each schedule has its own set of conditions that have to stay true in order for it not only to be selected, but also continue to operate. In the case of either ensuring the current schedule is still valid or selecting a new one is needed, that's where 'state' and 'conditions' mentioned in Subsection 8.2 comes in handy. The states are important given that the agent is unable to make any decisions. Meanwhile, the conditions - which are how the agent sees the world - are updated based upon the execution of the schedule as well as the new data received from vision, sonar, hydrophones, pressure sensor and more.

While a task manager is likely to be unable to determine a proper sequence of actions to stack blocks on top of each other, it is expected that problems of this sort will be solved in advance by the human operator before passing a mission plan along to the task manager.

8.5.4 Task Controller

For the task controller we will be using a *non-deterministic augmented hierarchical state machine*. There are two important reasons for choosing a finite state machine type of task controller for this mission. The entry level for starting to apply finite state machines for task control is low and there are a few really good libraries for this state machines in Python. The second reason for using a FSM is that the number of states that the task controller will contain for this mission is manageable and relatively easy to configure.

As described just earlier, non-deterministic means that a state can transition to not only one, but several other states. Augmented means that the transition arcs between the states has timers connected to them and can even bring information output information from one state along as input for the next state. Hierarchical state machine is introduced for the ease of programming and re-use of smaller finite state machines.

8.5.5 Software Implementation

The urls linking to the source code can be found in Appendix A.

Open-Source Packages, Libraries and Standards

For implementing the task controller we'll be taking use of a library called *SMACH*. SMACH is a task-level architecture for rapidly creating complex robot behavior. At its core, SMACH is a ROS-independent Python library that allows you to design, maintain and debug large, complex hierarchical state machines.

Before advancing into the use of SMACH it is important to know some of the limitations of the library. The library is not meant to be used as a state machine for low-level systems that require high efficiency as it is a task-level architecture. SMACH will also fall short as the scheduling of your task becomes less structured.

To create a SMACH state machine, you first create a number of states, and then add those states to a state machine *container*. There are a number of predefined SMACH containers that can also save you a lot of programming. The *Concurrence* container returns an outcome that depends on more than one state and allows one state to preempt another. The *Sequence* container automatically generates sequential transitions among the states that are added to it. And the *Iterator* container allows you to loop through one or more states until some condition is met.

The actual computations performed by the state can be essentially anything you want, but there are a number of predefined state types that can save a lot of unnecessary code. In particular, the *SimpleActionState* class turns a regular ROS *action* into a SMACH state. Similarly, the *MonitorState* wraps a ROS topic and the *ServiceState* handles ROS services.

Gate Search and Docking Finite State Machine

The first HSM to create is we will name PATROL. The PATROL HSM will handle the states and transitions that are only connected to the actual mission AUV is meant to solve; which is to track down the yellow gate first and then the green pole there after. The PATROL HSM consist of two smaller FSM's: the GATE_SEARCH FSM and the DOCKING FSM, as well as a few stand-alone states connecting the two FSM's together in various ways.

The PATROL HSM can be seen in Figure 8.10 which is a screen-grab of the HSM running in real-time. SMACH gives you full introspection in your state machines, state transitions and data flow through a plugin called the *smach_viewer*. If a state is colored green it means that the particular state is active. The large grey rectangles are FSMs on there own. The red rectangles are not states, but transitions in and out of the larger FSMs; where as the arrows dark arrows are transitions between states.

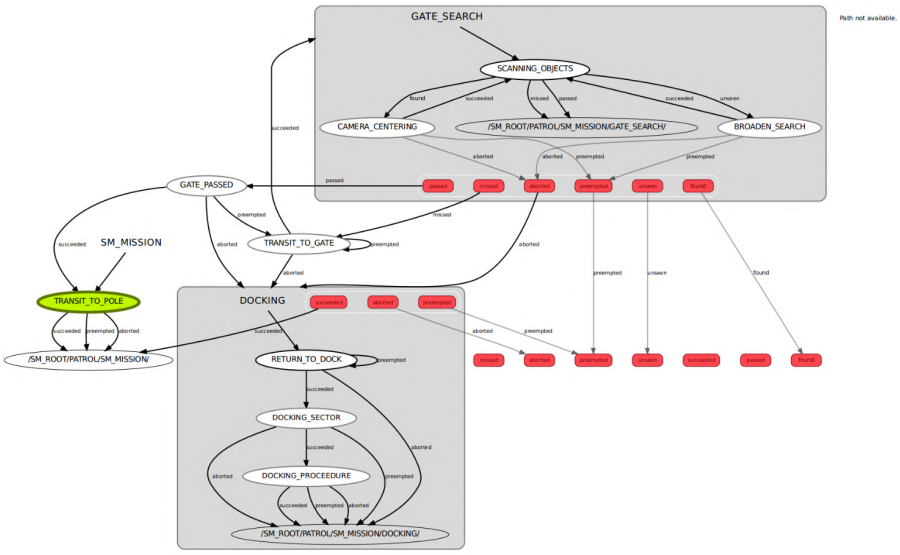


Figure 8.10: Overview of the state machine.

The PATROL HSM has a really simple control flow. The mission will start in the TRANSIT_TO_GATE state where the AUV will be in transit mode using the path-following controller with line-of-sight guidance made in the case study from Section 7.5. If the AUV succeed in transitting to the terminal sector, the HSM will transition into the GATE _SEARCH FSM. If the AUV is not able to track down the yellow gate terminal sector within 60 seconds, it will restart and try again. If some internal error along the way, the state will be aborted and the state machine will transition to the DOCKING FSM. In the DOCKING FSM the AUV will switch from the path-following controller over to the DP-controller made in the case study from Section 7.4 and go into docking an abort the whole mission.

When the state machine enters the GATE_SEARCH FSM, it will switch from the path-following controller into a camera-centering PID-controller in surge, sway, yaw and heave. The tuning gains for the PID camera centering controller can be found in Appendix C.2.3). The GATE_SEARCH FSM has three key states: (SCANNING_OBJECTS, CAMERA _CENTERING, BROADEN_SEARCH). When in the SCANNING_OBJECTS state, the

AUV will activate its computer vision object detection nodes that was created in the case study from Section 6.6. Whenever the yellow gate it detected in the image, the state machine will transition into the CAMERA_CENTERING state. In this state the camera-centering controller will pick up the location of the bounding box in the image and start centering it in the image while driving towards it. When in the CAMERA_CENTERING state, if the object detection node loses sight of the gate it will transition into the state BROADEN_SEARCH. In the BROADEN_SEARCH state, the vehicle will enter into an open-loop surge control and drive forwards. If the object detection node is able to pick up on the gate again, the state machine will go back into CAMERA_CENTERING, otherwise it will drive forwards until a timeout trigger aborts the GATE_SEARCH FSM and continues on the next task in the mission.

The logic of telling the state machine that the vehicle has successfully passed through the gate is challenging without having a sonar or stereo vision. To do this with a mono-camera, we combine several efforts of a working memory. Whenever the vehicle recognizes the yellow gate in the image and is able to create a bounding box, it will set a boolean parameter $FOUND = True$ and trigger an internal clock. If the vehicle has been able to track the object of interest over time, and there is a sudden period where the bounding box disappears out of the edges of the image, which is registered as a sign that the vehicle has successfully entered through the gate.

Recharge Battery Finite State Machine

To make the example more realistic, a *battery_simulator.py* node has been added to simulate a battery by publishing a decreasing value on a battery level ROS topic. Other nodes can then subscribe to this topic and respond accordingly when the battery level falls to low.

When the battery simulator node is running, one can access a dynamic reconfigure user interface (seen in Figure 8.11) to manually use the sliders or text boxes to change the battery runtime and battery level. The node also defines a ROS service called *set_battery_level* that takes a floating point value as an argument and sets the battery charge to that level. This will be use to simulate a recharge of the battery by setting the level to 100 or to simulate a sudden depletion of the battery by setting a low value.

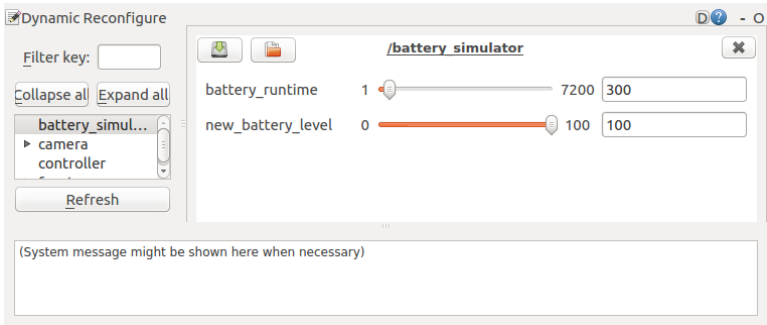


Figure 8.11: Fake battery simulator.

Suppose we want to monitor the AUV's battery level and if it falls below a certain threshold, the AUV should pause or abort what it is doing, navigate to the docking station and recharge, then continue the previous task where it left off. The battery monitoring task will subscribe to a diagnostics topic that includes the current battery level. When a low battery level is detected, a check battery condition should fire and the recharging task will begin execution while other tasks are paused or aborted.

The finite state machine for monitoring the battery will be called RECHARGE and can be seen in Figure 8.12. As earlier mentioned, SMACH provides the pre-defined states *MonitorState* and *ServiceState* to interact with ROS topics and services within a state machine. The state machine will use a *MonitorState* to track the simulated battery level and a *Servicestate* to simulate a recharge.

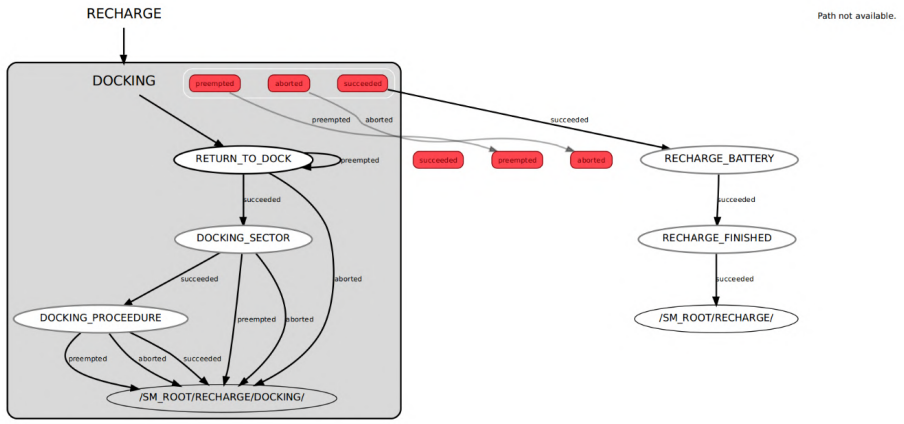


Figure 8.12: Overview of the state machine

The control flow of the RECHARGE state machine is really simple. When the battery level falls below the 30% as provided by the mission plan, the state machine will preempt all current tasks, change into a DP-controller and start navigating towards the docking sector. Once the vehicle is within the sphere of acceptance of the docking sector and also have the right orientation, the state machine will transition to the RECHARGE_BATTERY state. Once the battery is recharged (set to happen immediately).

Overview of the Augmented Hierarchical State Machine

With the two different FSMs created (patrolling and check battery), it is time to put the two together. We want a low battery signal to take top priority so that the AUV stops its patrol and navigates to the docking station. Once, recharged, the AUV will continue its patrol beginning with the last waypoint it was heading for before being interrupted. As

mentioned, SMACH provides the *Concurrence* container for running tasks in parallel and enabling one task to preempt the other when a condition is met.

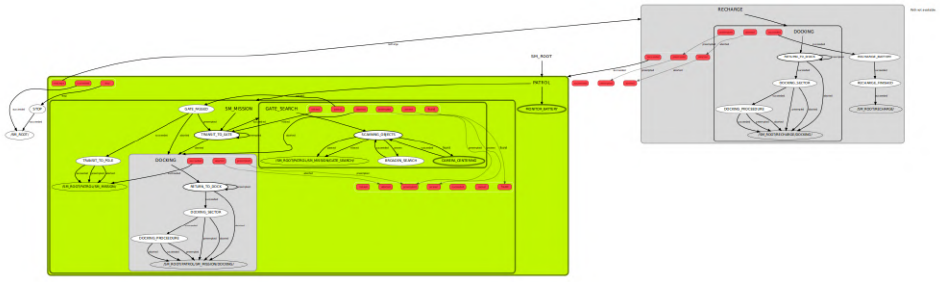


Figure 8.13: Overview of the state machine

The larger green box on the left represents the concurrence container thay contains the monitor state MONITOR_STATE that subscribes to the battery topic. The smaller grey box on the right represents the RECHARGE state machine and contains the DOCKING and RECHARGE_BATTERY states.

8.5.6 Simulation Results and Discussion

The simulation is performed in the Gazebo simulation environment created in the case study from Chapter 5, which is a 1:1 scale version of the marine cybernetics lab at the Department of Marine Technology, NTNU. For vehicle localization the AUV take use of the extended Kalman filter (EKF) created in the case study from Section 6.4. For vehicle perception the AUV take use of the computer vision object detection capabilities implemented in the case study from Section 6.6. For vehicle path-following the AUV take use of the backstepping controller made in the case study from Section 7.5 and for vehicle dynamic positioning and setpoint regulation the AUV take use of the nonlinear PID controller made in case study from Section 7.4. For high-level task control the AUV will use the constructed SMACH non-deterministic augmented hierarchical state machine depicted in Figure 8.13.

This case study is not meant to investigate the performance of the localization, perception and controllers as these have already been proven robust and reliable in previous case studies. For this particular reason, the plots will show only the xy-positioning, battery levels and a live video demonstration.

In Figure 8.14 is the xy-plot of the vehicle. The AUV starts of in positioning $(0, 0)$ in the middle of the image. At mission start the vehicle switches into path-following control and starts transitting along the stippled path from $(0, 0)$ to $(10, -2)$. Once the vehicle has successfully entered the terminal sector (denoted by a large blue circle in $(10, -2)$)

it transitions into the GATE_SEARCH state machine and starts looking for the the gate. Once the gate is found, the vehicle switches into camera centering control, tracks down the gate, and then passes successfully passes through it (scan QR to in Figure 8.16 to see video).

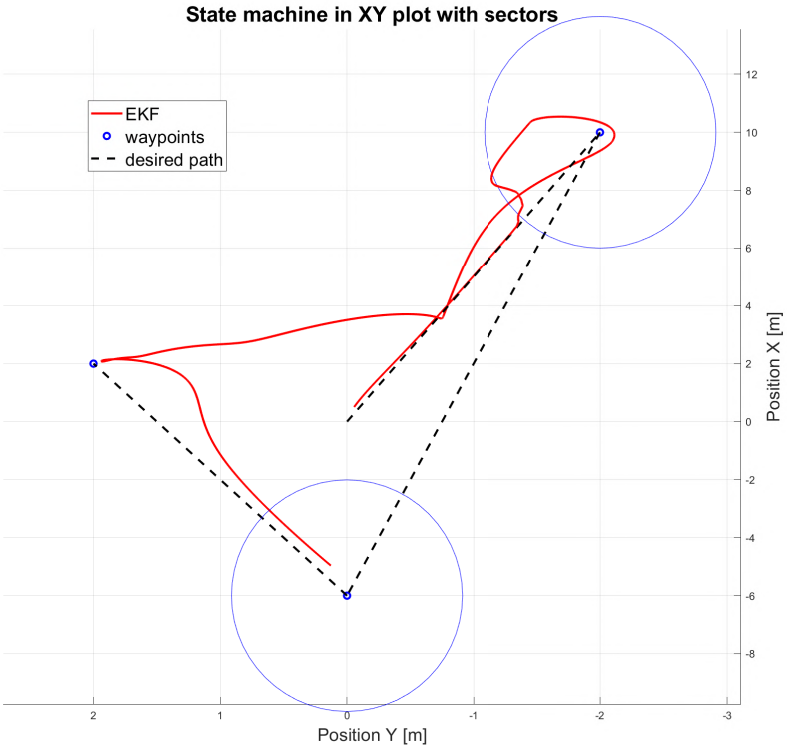


Figure 8.14: Caption

Once the vehicle has successfully passed the gate it turns of the camera centering controllers and switches back into path-following control, rotates 180 degrees and starts transitting towards the stippled line from coordinate $(10, -2)$ to $(-6, 0)$ towards the terminal sector of the green pole (denoted by the large blue circle in $(-6, 0)$). As you can see by the plot, the vehicle slowly starts guiding itself towards the line, but approximately in coordinate $(4, -0.75)$ the vehicle changes direction. At this point the state machine has found that the battery level is below 30% of capacity and it preempts the navigation towards the green pole, changes into dp-controller and starts navigating itself towards the docking station found in coordinate $(2, 2)$. Once the vehicle reaches the docking station and has successfully completed a "simulated recharge", then the state machine resumes the navigation task, switches to the path-following controller and start tracking the stip-

pled line from coordinate $(2, 2)$ to $(-6, 0)$ instead. The vehicle completes its mission once it has reached the terminal sector of the green pole in coordinate $(-6, 0)$ and successfully detects the object with a boundary box in the image.

From the Figure 8.15 is a plot of the battery level throughout the simulation. After about 105 seconds when the battery level is approximately at 8 % capacity, the vehicle is successfully able to dock and the charges the battery back up for 100 % capacity so that the vehicle can complete the mission.

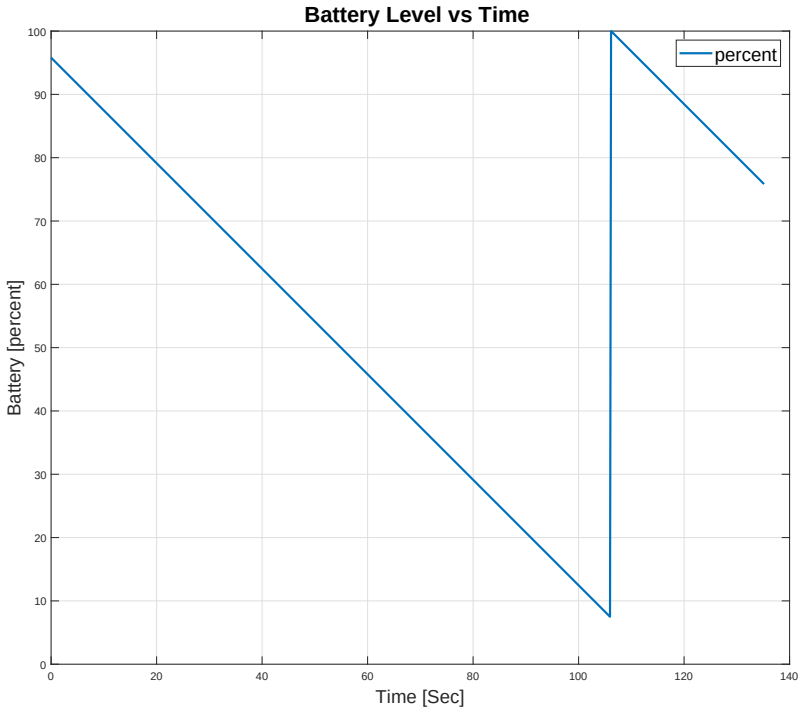


Figure 8.15: Caption

From Figure 8.16 is a QR code directing to a live video demonstration of the simulated run.

Figure 8.16: Scan the QR with a QR-code app to see a live demonstration of the simulation

8.5.7 Conclusion of Case Study

To conclude the case study, one can tell that the state machine has been successful in completing the mission. Not only is the high-level task controller able to control the flow of states and transitions during the mission, but it able connect the high-level execution of the mission plan with the low-level execution of the localization, perception, guidance and control.

One of the disadvantages of a FSM task controller mentioned earlier, was that they could become unmanagable when the number of states and transitions become large. For the number of states and transitions in this mission, the flow of logic and ease of implementation has been easy to manage. However, one could imagine how complex the state machine in Figure 8.13 would be with a 100 more states. In that case, it would be a good consideration to switch to a behavior tree rather than a finite state machine.

8.6 Chapter Summary

The aim of this chapter was taking upon the aspect of explaining, designing and implementing a mission control that is able to execute an AUV mission communicated by a human operator. The mission is communicated at a high level of abstraction through a mission plan. The mission plan is decoded into actual tasks through the task manager that deliberates long-term goals and task schedules. When the tasks are sent to the task controller, they are broken down to an even lower level of abstraction through states, transitions, nodes, leaves depending on your controller. The task controller ensures a careful control flow, coordination and execution of vehicle perception tasks and motion primitives. These elements all together ensure the long-term and deliberative behavior intended by the

mission controller. At last the mission logger records desirable data points that one would like to collect for data about the environment, mission and internal sensors that can later be used for mission debugging and planning of future missions.

When designing an mission control it is important to choose an appropriate agent design that is applicable in your task environment (deciding properties such as partially observable, stochastic, static, etc.). Once you have figured out what properties of your task environment, you can choose a task controller that has the appropriate tasking properties (properties such as task hierarchies, task priorities, task concurrency, etc.) for handling the task environment at hand.

The case study *Manta v1 Mission Control using a Non-Deterministic Augmented Hierarchical State Machine for Task Control* proves that Manta v1 software architecture is successful in connecting the high-level mission control and low-level navigation, guidance and motion control.

CHAPTER 9

PHYSICAL EXPERIMENTS

Before a finishing up the development process for the Manta v1 software architecture, there must be process to check if system meet basic requirements for the system. The experimental testing in this chapter is aimed at checking whether the system is well-engineered, error-free and detect potential software bugs. It also has the objective of help finding out whether the software is working according to the intended purpose. The experimental process will evidently help determine whether the software is of high quality.

9.1 Physical Experiments at the Marine Cybernetics Laboratory

The Marine Cybernetics laboratory depicted in Figure 9.1 is a small ocean basin laboratory at the Department of Marine Technology, NTNU. It is relatively small, but suitable for tests of motion control of model-scale surface vessel and small underwater vehicles. It is equipped with a moveable bridge with positioning motion capturing cameras capable of measuring 6DOF movements of target objects, as well as a wave maker. The basin measures $40[m]x6.45[m]x1.5[m]$ in length, width and depth, respectively.

Figure 9.1: The Marine Cybernetics Laboratory

9.1.1 Laboratory Equipment

The laboratory is equipped with a *Qualisys* real-time motion capture system. The key components are the Qqus cameras and the Qualisys track manager (QTM) software. The Qqus system has three high-speed infrared (IR) cameras mounted on a towing carrage, which tracks the IR reflector orbs fitted on the vehicle body. The experiments can be supervised from the control room with a computer dedicated for the QTM system and a TV connected to the two cameras in the lab. The experimental captured from the QTM system is recorded and stored in *.mat* files that can later be plotted and compared with logged *rosbag* data from the vehicle.

9.1.2 Experimental Setup and Calibration

To receive 3D data the system must be calibrated. The most commonly used calibration is the Wand calibration that uses two calibration objects to calibrate the system: the L-shaped reference structure and the calibration wand.

Once calibrated, the QTM software can calculated the orientation and attitude of the vehicle by tracking and performing trigonometrical calculations with mounted orbs on top of the vehicle body depicted in Figure 9.2.

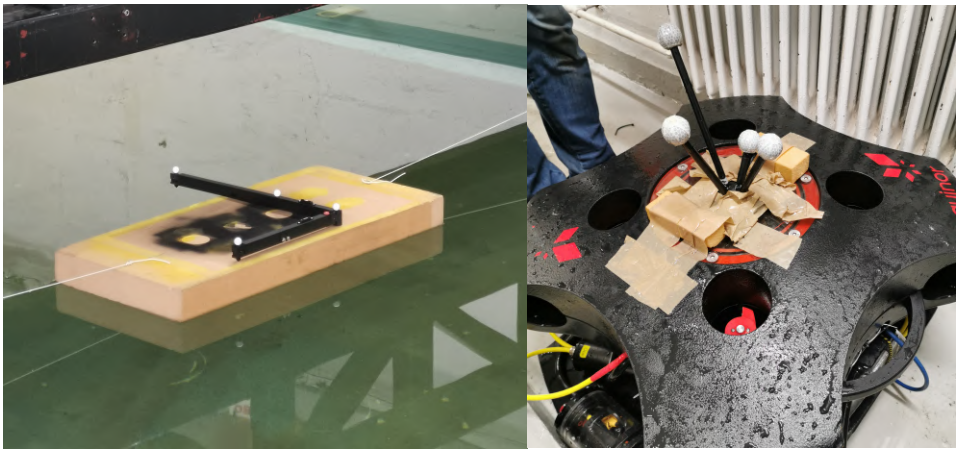


Figure 9.2: Qualisys calibration procedure

The motion capture system is calibrated before each measurement session to make sure that the captured data has high quality. Figure 9.3 shows the calibration mass defining the region in 3-D region where the motion capture system can provide valid measurements.

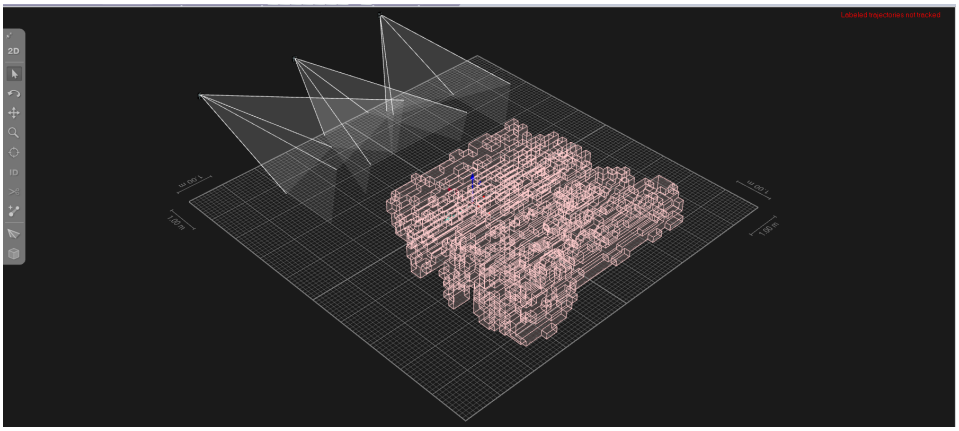


Figure 9.3: Qualisys available tracking volume after calibration procedure

9.2 Test Results and Discussion

The physical experiments were completed over the time-span of 12 days. Approximately 70% of the time were spent configuring the Qualisys system and performing calibration, where as remaining 30% of the time were spent tuning and recording the experiments.

A total of six tests where conducted during the physical experiments, all of them listed in below:

- **Test 1:** Multiple rounds of waypoint tracking in a square formation at the water surface using extended Kalman filter and nonlinear PID controller.
- **Test 2:** Multiple rounds of waypoint tracking in a hexagon formation at the water surface using extended Kalman filter and nonlinear PID controller.
- **Test 3:** Single round of waypoint tracking in square formation when submerged to $0.5[m]$ using extended Kalman filter and nonlinear PID controller.
- **Test 4:** Path following of parallel line segments at the water surface using extended Kalman filter, line-of-sight guidance and nonlinear backstepping control.
- **Test 5:** Path following of crossing line segments at the water surface using extended Kalman filter, line-of-sight guidance and nonlinear backstepping control.
- **Test 6:** Underwater object detection of yellow gate and red pole

Before diving into the results and discussions from the tests, it is worth mentioning that the actual produced thrusts from the thrusters have not been logged because the message type used for sending thrust messages ($geometry_msgs/Wrench$) does not have time stamps and can therefore not be plotted as a time series as is. Thrust information has therefore not been included in the plots.

Test 1: Square Tracking using EKF and Nonlinear PID Controller

The first test is aimed at testing whether or not the extended Kalman filter is successful in estimating the position and attitude of vehicle, if there are any accumulated biases over time in the estimates, and if the nonlinear PID controller is able to track down multiple waypoints. Due to the fact that the Qualisys motion detection cameras are unable to track the mounted orbs whilst the vehicle is submerged, the tests had to be conducted at the water surface to get ground truth position and attitude.

The result on the left side in Figure 9.4 show the vehicle positioning in the XY-plane. The solid green line shows the estimated position from the EKF and the solid red line show the ground truth position from Qualisys. The large blue circles show the circle of acceptance, and the blue points show the target waypoints.

First and foremost, one can notice from the plot that the vehicle is successful in tracking all the waypoints from start to end. The vehicle starts up in position $(x, y) = (-2, 0)$ and tracks the waypoints clockwise. Notice at waypoint number 3 in $(x, y) = (-1, 1)$ that the vehicle always takes a large turn. This is due to the attached tether cable - when stretched - providing extra resistance such that the vehicle is overcompensated when the waypoint switching happens.

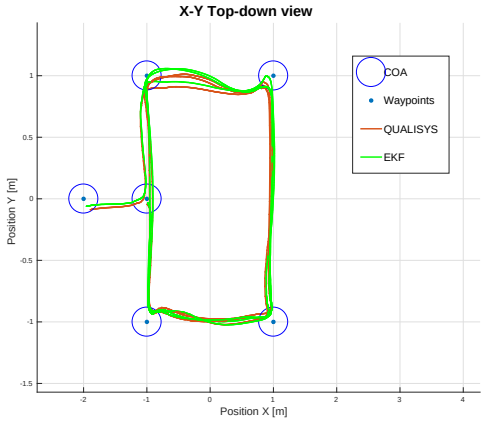


Figure 9.4: Left figure: Low-speed waypoints tracking of square configuration seen in XY-plane with EKF and Qualisys comparison. Right figure: A QR-code directing to video of from the test.

From Figure 9.5, in comparison to the ground truth reference of Qualisys, one can tell that the EKF is undoubtedly accurate in estimating the positioning of vehicle. The purpose of repetitive tracking of the same square formation was to check if there is any accumulated bias that could affect the position and attitude estimation from the EKF. As you can see there is no significant signs of any bias, the reason being that the static transformations of the sensors are correct and also that the sensors themselves are good. However, one should take note that even though there are not signs of any bias during the 4 minutes of the recorded run, that does mean there would not be any bias after 20 minutes. The EKF does not have any bias estimation, so there would be no way of accounting for the bias unless you were to implement an error state Kalman filter (ESKF) instead.

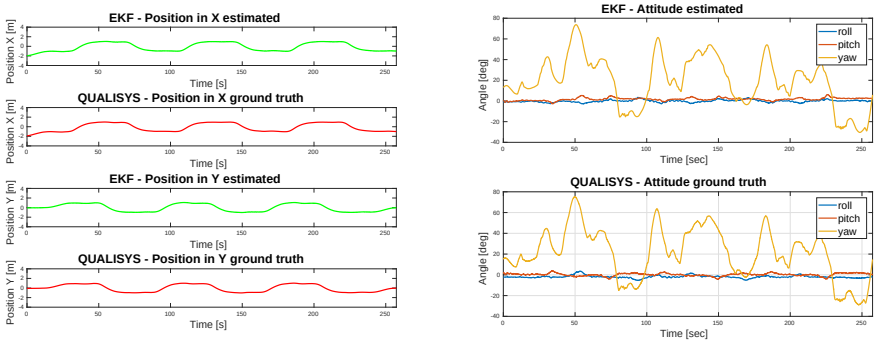


Figure 9.5: EKF vs Qualisys position and attitude comparison

The plots in Figure 9.6 show the ground truth and estimated linear velocities on the left

and the angular velocities on the right. The first thing to mention is that Qualisys does not actually measure the velocities, only absolute position and orientation. In post processing the ground truth velocities were aquired by differentiating the timeseries, and as you one see they pick up a lot of noise in the process. Even though the ground truth plots are noisy, they still prove the accuracy of the EKF in estimating both the linear and angular velocities. It is fair to mention that the controllers use the velocities from the EKF in the feedback loop, and not the Qualisys. If you would use Qualisys instead of the EKF, the signals would have to go through extensive signal processing to avoid ruining the actuators of the vehicle.

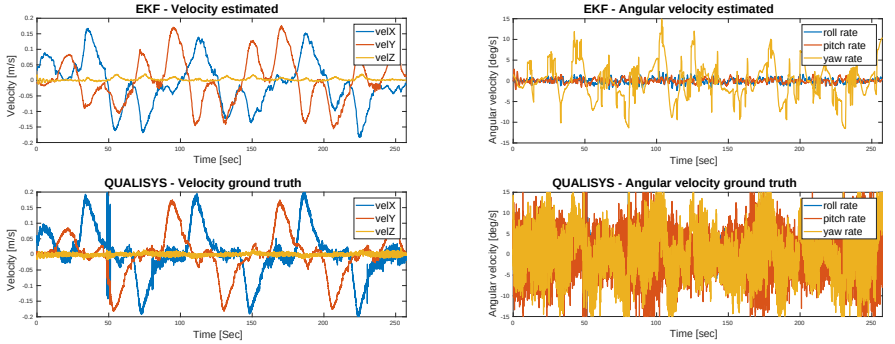


Figure 9.6: EKF vs Qualisys linear and angular velocity comparison

Apart from both the EKF and nonlinear PID controller proving themselves accurate, the largest take-away is probably that the tuning of initial state estimates, initial process noise covariance (Appendix B.1), and the PID gains (Appendix C.2.1) are exactly the same in simulations and the physical experimentation's, proving that the formulated mathematical modeling of the vehicle seem to be very much accurate.

Test 2: Hexagon Tracking using EKF and Nonlinear PID Controller

For **test 2** the intention was to re-do **test 1**, only this time with a hexagon formation of the waypoints rather than a square formation. The reason was to check if the state estimation or the nonlinear PID controller would behave differently if the tracking operation was different.

As one can see from the Figure 9.7 through to Figure 9.9, apart from the fact that the geometrical tracking operation being different, the EKF and controller behave just the as well as before. The statements from the last test run remain, and there is no reason to comment any further on **test 2**.

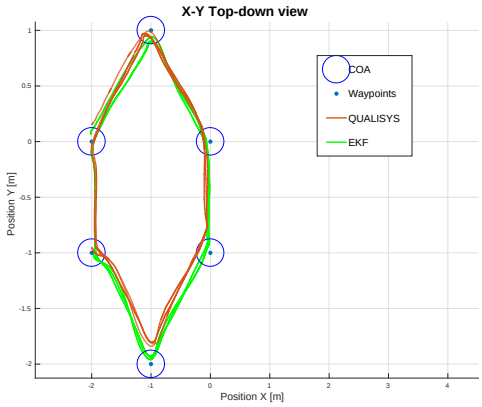


Figure 9.7: Left figure: Low-speed waypoints tracking of hexagon configuration seen in XY-plane with EKF and Qualisys comparison. Right figure: QR-code directing to video of the recorded run.

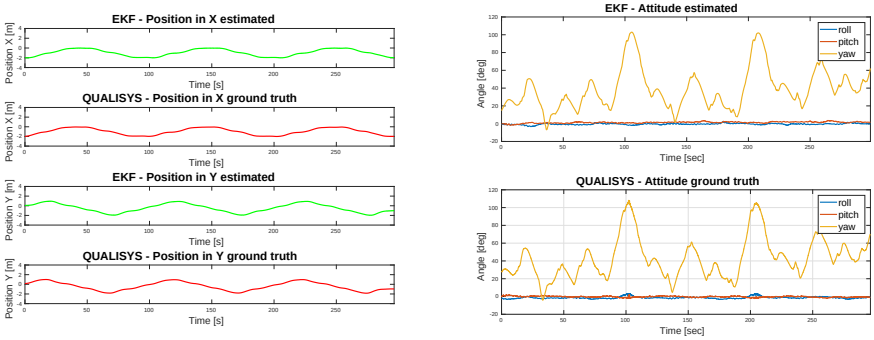


Figure 9.8: EKF vs QUALISYS position and attitude comparison.

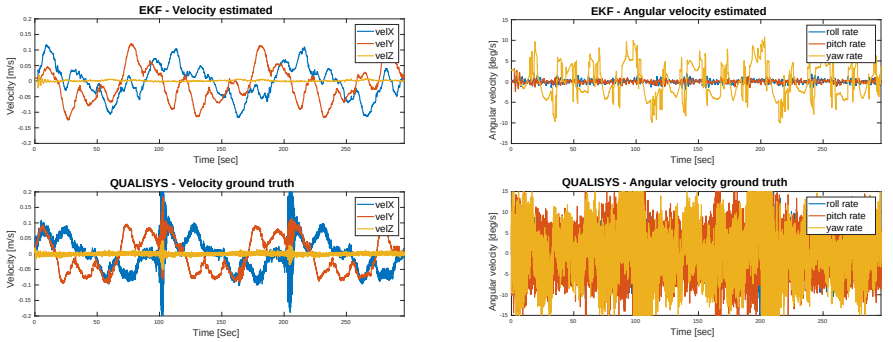


Figure 9.9: EKF vs QUALISYS linear and angular velocity comparison.

Test 3: Square Tracking using EKF and Nonlinear PID Controller while Submerged.

For **test 3** the goal was to check whether or not the nonlinear PID controller were able to track waypoints while the vehicle being submerged. Since the Qualisys is not able to provide ground truth while the mounted orbs are submerged, this test does not how any ground truth position and attitude to compare with. However, from **test 1** and **test 2** the EKF has proved itself highly accurate and there are every reason to believe that the estimates of vehicle position and attitude remain accurate when submerged.

First, from the left plot in Figure 9.10 it is clear that the controller is able to track the target waypoints located at $0.5[m]$ depth. You can see that every time the setpoint switch happens, there is a slight inconsistency in the depth tracking. This is because each time a new setpoint is acquired by the PID controller, the integral term is reset to avoid the vehicle being to aggressive when approaching the next waypoint. The controller is however able to reach target depth again after a short time when the integral term has had time to build up again.

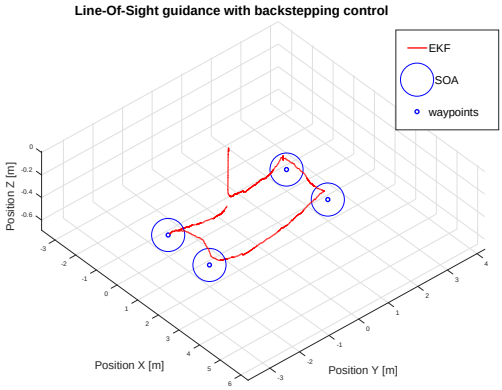


Figure 9.10: Left figure: EKF position with target waypoints. Right figure: QR-code directing to video of the recorded run.

All the plots in Figure 9.11 are produced from the EKF. There are two things worth noting in the plots: first, the plot of estimated attitude in the bottom left corner shows that the yaw changes a lot during the recorded run. That is because in this particular run the controller were set to operate in POSE_HOLD which tracks position in x,y and z, but leaves roll, pitch and yaw as open loop. So the yaw angle can be an changes freely. As for the roll and pitch angle, they remain more or less zero because the vehicle is self-stabilizing. The second thing worth noticing is that the estimated linear and angular velocities in the plots on the right sight always tend towards zero. As mentioned in the simulation results from Chapter 7, this is because the nonlinear PID controller is intended for dynamic positioning, and therefore regulate the velocities towards zero.

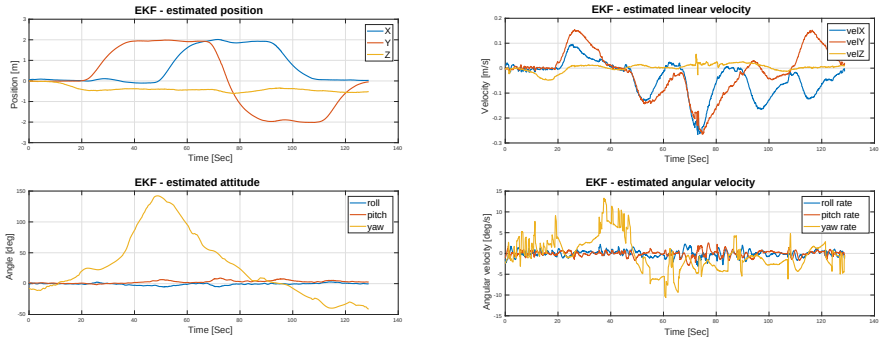


Figure 9.11: EKF position and attitude estimates

Test 4: Path Following of Parallel Lines using EKF, LOS Guidance and Backstepping Control

For **test 4** the goal were to check how the backstepping controller will perform when having to switch between two parallel line segments.

From Figure 9.14 the vehicle starts of in $(x, y) = (0, 0)$ and starts tracking the stippled line segment from $(0, 1)$ to $(3, 1)$. When the vehicle reaches the circle of acceptance (which is 0.2[m]) of the first waypoint the controller switches setpoint and start tracking the towards the next waypoint in $(7, -2)$. As one can see, the controller is successful in tracking the setpoints, however it seems that the lookahead distance for the line-of-sight guidance could to be a bit smaller so that vehicle will merge faster with the path.

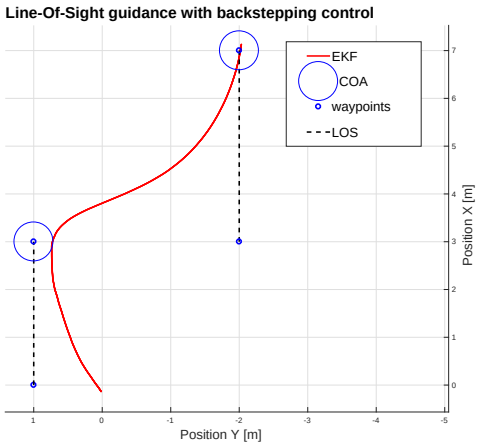


Figure 9.12: Tracking XY-plane path segments with line-of-sight guidance and backstepping control

All the plots in Figure 9.13 are produced from the EKF. There are two things worth noting in the plots: first the yaw angle shown in the bottom left plot shows that the vehicle start of pointing in a $30[deg]$ angle, and as it comes towards the first setpoint the yaw angle slowly tend towards $0[deg]$ angle. Once the waypoint switching happens, the yaw angle has a rapid change towards $-50[deg]$, and again the yaw angle slowly tends towards zero once the vehicle is closing in on the waypoint. Looking at the top right plot in Figure 9.13, the second thing worth noticing is that while the target surge speed is $0.2[m/s]$ the vehicle is only able to reach $0.12[m/s]$. There are several reasons for this, first of all the controller could be a little bit more aggressive in surge. However, a noticeable difference between simulation and physical experimenting is that the connected tether (only used for logging and monitoring) provide both spring resistance and drag. It is believed that the vehicle would be closer to the target speed $0.2[m/s]$ when the tether is not connected.

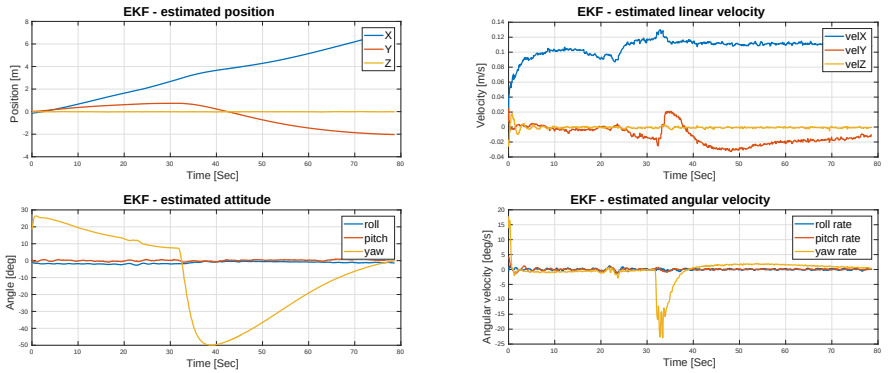


Figure 9.13: EKF position, attitude, linear velocity and angular velocity estimates.

Similarly with the EKF and the nonlinear PID control, the largest take-away from this test is the fact that the tuning of the backstepping controller gains are exactly the same in simulations and the physical experimentation's (see Appendix C.2.2), proving again that the formulated mathematical modeling of the vehicle seem to be very much accurate.

Test 5: Path Following of Crossing Lines using EKF, LOS Guidance and Backstepping Control

For **test 5** the intention was to re-do **test 4**, only this time with two crossing line segments rather than a parallel ones. The reason was to check if the nonlinear backstepping controller would behave differently if the tracking operation was different.

As one can see from the Figure 9.14 through to Figure 9.15, apart from the fact that the geometrical tracking operation being different, the controller behave just the as well as before. The statements from the last test run remain, and there is no reason to comment any further on **test 5**.

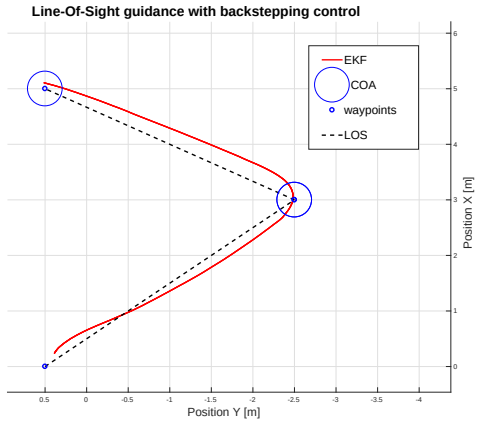


Figure 9.14: Tracking XY-plane path segments with circle of acceptance around goal waypoints using EKF position and attitude estimates.

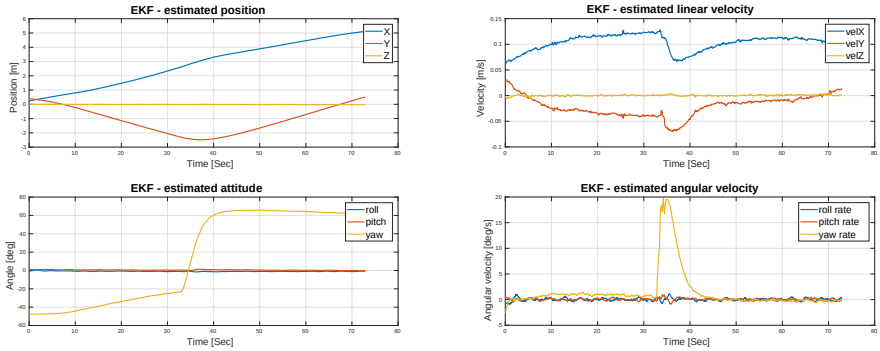


Figure 9.15: EKF position, attitude, linear velocity and angular velocity estimates

Test 6: Underwater Object Detection of Yellow Gate and Red Pole

The the yellow gate and red pole was built by the mechanical team at Vortex NTNU using plastic tubes and spray paint depicted in Figure 9.16. Camera calibration is done with the vehicle in the water looking at a chessboard. For camera calibration a ROS package named *camera_calibration* is used. Once the calibration is completed the barrel distortion and pinchusion distortion can be removed from the image.

Figure 9.16: Gate and pole for testing. Chess board for calibration of forward facing camera

On the left in Figure 9.17 is a screen grab of the RGB video stream from the computer vision object detection node with a bounding box capturing one leg of the gate. There was a lot of issues when deploying the computer vision capabilities in a physical test pool. The first thing you will notice is that there is a lot of particles in the water that effects the quality of the image due to the colors becoming vague. Another issue that occurred is the fact that there is a lot of objects in the pool that has colors in the yellow part of the color spectrum. This can be seen by looking at the binary image in Figure 9.17. You will see there is a lot of small white particles making in hard for the computer vision node to complete a bounding box encapsulating the gate. A third issue that occured, is that when the vehicle is close to the surface you get reflection in the water surface ultimately giving two mirrored gates in image making it hard to create a bounding box.

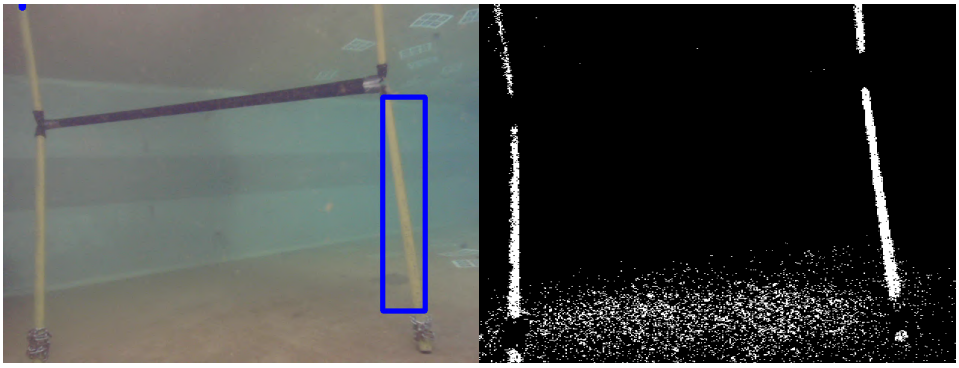


Figure 9.17: Gate detection with bounding boxes

In Figure 9.18 is a screen grab of the RGB video stream from the computer vision object detection node capturing the red pole. Capturing the red pole was easier due to the fact that there was not as much other red colored particles and objects in the water. However, it was still the issue with the red pole having a mirrored reflection in the water surface.

Figure 9.18: Pole detection with bounding boxes

All things considered, the results of both the yellow gate and red pole object detection was OK. However, for later the computer vision object detection should be done using convolutional neural networks instead to get better and more robust performance. The results of using traditional computer vision techniques is that they are quick to implement, tune and do not require any training of a neural network; however, the resulting object detection capabilities are to brittle for actually using it in a complete software architecture.

9.3 Conclusion of Physical Experiments

The physical experimenting has been a really important part for verification and testing of the software architecture. One thing is to have everything working in a simulation, but once the performance show also in physical experiments you can tell that the software carry high quality. The results show that both the EKF and the controllers have almost equal performance both in simulation and physical experiments. And even better, they have the exact same tuning parameters. The fact that the tuning parameters are the same ultimately means that the mathematical model of the Manta AUV derived in Chapter 4 is very much accurate.

9.4 Chapter Summary

The aim of this chapter was to check wether the system is well-engineered, error-free, has software bugs and is working for its intended purpose. A total of six tests were conducted. The testing of the EKF, nonlinear PID controller and nonlinear backstepping controller show very good performance. The testing of computer vision object detection show OK performance, but should be re-done using convolutional neural networks.

CHAPTER 10

CONCLUSIONS AND RECOMMENDATIONS FOR FURTHER WORK

The objective of this Master's thesis has been to design, develop and implement a deliberative agent software architecture for high-level planning and execution, with an underlying low-level navigation, guidance and motion control. There is hard to come across any end-to-end guide on how to do to that, so for that reason three important research questions were formulated:

1. *Traditionally, system identification of underwater vehicles are done through experimental study in marine labs. Will an empirical and analytical study suffice in identifying accurate hydrodynamic model representation for underwater vehicles?*
2. *Robust robotic software is paramount to high performance capabilities across variety of tasks. Intuitively, adding more capacity to an application should increase the components performance. What are the most effective strategies to ensuring a robust and high performing software for a growing code stack?*
3. *Intelligent agents cannot work just in a purely reactive capacity. How can we enable for a deliberate and long-term robot behavior to be established?*

The following two sections concludes the Master's thesis and provides recommendations for further work.

10.1 Conclusions

In the light of the first research question, a major concern for agile and continuous development of the software architecture was to have an accurate representation of the vehicle

in its underwater environment. The mathematical model was acquired through a careful and iterative system identification process. Even though verifying the model would require hydrodynamic experimentation, comparison of results from testing the nonlinear PID controller and the nonlinear backstepping controller in simulation and in physical experiments suggest that the model is indeed very much accurate. This is mostly confirmed by the fact that the controllers have the same tuning parameters both in simulation and in physical experiments. And not only that, the performance of a model-based controller such as the nonlinear backstepping controller is tied to accurate model representation. By all means, an empirical and analytical approach to hydrodynamic modeling will suffice, however not to all geometrical shapes. The Manta AUV body has the benefit of being relatively uniform and blunt, and geometrical simplifications are liable. If the body were of a slender type, strip theory would apply and could give even better results. On the other hand, if the vehicle has a lot of irregularities in the shape, methods such as strip theory that rely on the superposition principle would not apply and an experimental lab testing would be a favorable approach.

As for research questions number two, it becomes very obvious through the software development process that the code become more maintainable and easier to debug by always putting large emphasis on modular programming and reuse of code. First of all this allows for a smaller uncoupled code stacks that are both easier to debug, but ultimately also lets the program execute faster as the computational cost is at least proportional to the size of the code. Another thing that also become evident is that as these two concepts enforces rapid evolution of the software. If you were to create a new improved controller, update the computer vision capabilities or change of the mission controller you can easily take out the old one and insert the new as long as the endpoints remain the same. Lastly, a important point to make is how much you get in return for investing a lot of time into a careful system identification process. This has allowed for creation of a high-fidelity 3-D test environment in Gazebo and ultimately led to rapid prototyping and testing. It also gives invaluable insight into software weaknesses and points of failure without actually having to deploy the software on the vehicle each time.

Lastly, for research question number three, it has become abundantly clear that the performance of software architecture all together is heavily tied to the performance of the low-level navigation, guidance and control. Referring to the term 'a chain is only as strong as its weakest link', the performance of intelligent agent (being the mission control) is only as good as the tools of its disposal. That being said, the deliberative agent capabilities was mostly assured by separating the mission control into four parts (the mission plan, task manager, task controller and mission log). This allowed four several layers of abstraction in the controller where altered behaviors (or tasks) are consistent through the layers. There is also large benefits to using ROS as a communication backbone with i.e a task controller such as the finite state machine through SMACH. This allows for distributed and asynchronous processing through ROS, with task hierarchies and task concurrence provided through SMACH.

To this end, the Manta v1 software architecture supports the efficient execution of real-world missions involving multiple concurrent goals. The largest component of Manta v1 is its mission control, which continuously optimizes the execution of a mission as infor-

mation about the world is acquired. The architecture is rich in functionality, distributes its computation, and performs efficient re-planning in an unknown, unstructured, and changing environment. This system has been demonstrated on the Manta AUV in an indoor experimental pool and extensively verified in simulation.

10.2 Recommendations for Further Work

The software architecture present encouraging results and performance both in simulations and in physical experiments, but it also comes with limitations. The system is at current standing is only suitable for a single-agent systems, it does not possess any deep nested learning capabilities, mapping abilities, nor path planning.

The list below describe the most important recommendations for further work:

1. Representation of spatial memory through occupancy grid, volumetric maps, point clouds and graphs. This can be done through particle filters for SLAM once a sonar or underwater stereo vision is acquired.
2. Once capable of constructing and updating maps through spatial memory, there should be invested considerable time in adding both local path planning (obstacle avoidance) and global path planning (route selection). Once implemented the vehicle could function of a local planner voting on the quality of arcs in front of the vehicle, a path-following module votes likewise based on which arcs will take the AUV along the global path, and an arbitrator determines which direction is actually driven. This sort of architecture can be used to seek navigation goals in unknown waters without a pre-planned route.
3. In the interest of spending longer periods of time navigating underwater (30 minutes or more), there should be a consideration into changing from an extended Kalman filter to error state Kalman filter (ESKF). The ESKF is capable of online estimation of biases, and thus can provide extended time of missions while maintaining accuracy in terms of state estimation.
4. Improvement of the computer vision object detection through the use of convolutional neural networks instead of traditional computer vision algorithms would most likely improve robot perception across both simulation and in physical experiments.
5. In the interest of creating large and complex robot behaviors, it should be considered to take a step away from using finite state machines for task control, and invest time into implementing behavior trees. This approach is still mature for robot purposes, but will most likely surpass FSMs on most aspects in the future.
6. The Manta v1 has been created using python 2.7, C++ 11, ROS Kinetic and Ubuntu 16.04. These versions are soon to be deprecated and there could be a good idea in upgrading to Python 3.7, C++ 17, ROS Melodic and Ubuntu 18.04 to benefit of future software updates, bug fixes and support.

7. For easier mission logging, state monitoring and mission planning there should be invested time in creating a user friendly graphical user interface (GUI) connecting with the endpoints of the mission control.

BIBLIOGRAPHY

[1] A. LABAO, P. N. Simultaneous localization and segmentation of fish objects using multi-task cnn and dense crf. *In book: Intelligent Information and Database Systems* (2019).

[2] ALAMI R., C. R. An architecture for autonomy. *The International Journal of Robotics Research, SAGE Publications 17* (1998), 315–337.

[3] AMBROSE R., W. B. Draft robotics, tele-robotics and autonomous systems roadmap. *Article: NASA, National Aeronautics and Space Administration 1* (2010), 1–27.

[4] ASPECT. Importance of modularity in programming. Available at http://aosd.net/importance-of-modularity-in-programming/, 2018. Accessed: 2019-04-24.

[5] BASHIR M., B. S. Hydrodynamic characteristics of rovs during deployment through wave-affected zone. *34th International Conference on Ocean, Offshore and Arctic Engineering* (2015).

[6] BIBULI M., BRUZZONE G., C. M. Mission control for unmanned underwater vehicles: Functional requirements and basic system design. *IFAC Proceedings Volumes 31, Issue 1* (2008), 131–136.

[7] BLEVINS, R. *Applied Fluid dynamics Handbook*. Krieger, 1984.

[8] BORENSTEIN J., T. P. Hybrid mobile robot localization using switching state-space models. *Proceedings of the 2002 IEEE International Conference on Robotics and Automation, Washington D.C, USA* (2002), 366–373.

[9] BRENNEN C.E., C. An internet book on fluid dynamics. values of the added mass, section b6. http://brennen.caltech.edu/fluidbook/basicfluiddynamics/unsteadyflows/addedmass/valuesoftheaddedmass.pdf, 2019. Accessed: 2019-05-12.

[10] BROWNLEE, J. A gentle introduction to computer vision. https://machinelearningmastery.com/what-is-computer-vision/, 2019. Accessed: 2019-30-12.

[11] BRUMITT, B. L. A mission planning system for multiple mobile robots in unknown, unstructured, and changing environments, doctoral thesis in robotics. *The Robotics Institute, Carnegie Mellon University* (1998).

[12] BRUYNINKX, H. Bayesian probability. *Dept. of Mechanical Engineering, K.U.Leuven, Belgium. Available at http://people.mech.kuleuven.ac.be/~bruyninc* (2002).

[13] C. PREMEBIDA, R. AMBRUS, Z. M. *Chapter 6: Intelligent Robotic Perception Systems. In book: Applications of Mobile Robots*. IntechOpen, 2018.

[14] CHAMPEAU J., D. P., AND L., L. Mission control with the uml and sdl formalisms. *In Proc. of Oceans 2000, Providence, RI, USA. 1* (2000).

[15] CHANG Z., B. X., AND X., S. Autonomous underwater vehicle: Petri net based hybrid control of mission and motion. *In Proc. of the 3rd International Conference on Machine Learning and Cybernetics, Shangai, China 1* (2004).

[16] CHIN, C. *Computer-Aided Control Systems Design*. CRC Press, 2017.

[17] CHITRE, M. Dsaav - a distributed software architecture for autonomous vehicles. *Article from Acoustic Research Laboratory, Tropical Marine Science Institute, National University of Singapore* (2008).

[18] COLLEDANCHISE, M. Behavior trees in robotics. *Doctoral Thesis, KTH, Stockholm* (2017).

[19] CORKE, P. *Robotics, Vision and Control Fundamental Algorithms in MATLAB*. Springer, 2016.

[20] D., P. Finite state modeling as an automation technology. *IEEE Aerospace Conference Proceedings. 1* (2000).

[21] EIDSVIK, O. Identification of hydrodynamic parameters for remotely operated vehicles. *Master Thesis, NTNU* (2015).

[22] FALTINSEN, O. *Sea loads on ships and offshore structures*. Cambridge University Press, 1998.

[23] FALTINSEN O.M, S. B. Slow drift eddy making damping of a ship. *Applied Ocean Research 9, Issue 1* (January 1987), 37–46.

[24] FJELLSTAD O., F. T. Quaternion feedback regulation of underwater vehicles. *Proceedings of the 3rd IEEE Conference on Control Applications, Glasgow, August 24-26 1* (1994).

[25] FONG, T. 2018 workshop on autonomy for future nasa science missions : Day 1, afternoon. In *NASA* (October 10-11, 2018), P. Carnegie Mellon University Pittsburgh, Ed., NASA Science Mission Directorate.

[26] FOSSEN, T. *Guidance and Control of Ocean vehicles*. John Wiley & Sons Ltd, 1994.

[27] FOSSEN, T. *Marine Craft Hydrodynamics and Motion Control*. John Wiley & Sons Ltd, 2011.

[28] FOSSEN T.I., B. S. Nonlinear vectorial backstepping design for global exponential tracking of marine vessels in the presence of actuator dynamics. *Proceedings of the 36th IEEE Conference on Decision and Control* (1997).

[29] FOSSEN T.I, F. O. Nonlinear modelling of marine vehicles in 6 degrees of freedom. *Journal of Mathematical Model ling of Systems 1*, 1 (1995).

[30] FOUNDATION, O. S. R. Ros documentation. http://wiki.ros.org/Documentation, 2019. Accessed: 2019-05-07.

[31] GARCIA P.C, H. L. *Indoor navigation strategies for aerial autonomous systems*. Elsevier, 2017.

[32] GIEBEL, R. P. *ROS by example: Volume 2*. A Pi Robot Production, 2014.

[33] GREWAL M., A. A. *Kalman Filtering: Theory and Practice Using MATLAB, Second Edition*. John Wiley & Sons, Inc, 2001.

[34] GUDMUNDSSON, S. *General Aviation Aircraft Design*. Elsevier, 2014.

[35] GUTZWILLER, R. S. Tasking teams: Supervisory control and task management of autonomous unmanned systems. *from book Virtual, Augmented and Mixed Reality: 8th International Conference, VAMR 2016, Held as Part of HCI International, Toronto, Canada* (2016), pp 397–405.

[36] HINCHEY M., V. E. *Autonomy Requirements Engineering for Space Missions*. Springer, 2014.

[37] KERMORGANT, O. A dynamic simulator for underwater vehicle-manipulators. *International Conference on Simulation, Modeling, and Programming for Autonomous Robots Simpar 8810* (2014), 25–36.

[38] KHALIL, H. *Nonlinear Systems, Third Edition*. Prentice-Hall, Inc, 2002.

[39] KIM T.W., Y. J. Task description language for underwater robots. *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)* (2003).

[40] KVALBERG, J. Monocular visual odometry for a mini rov. *Master's Thesis, NTNU* (2019).

[41] LEONARD J., D.-W. H. Mobile robot localization by tracking geometric beacons. *IEEE Transactions on Robotics and Automation 1*, 7 (1991), 376–82.

[42] MALTSEVA, D. 5 concepts of software distributed systems. benefits of distributed architectures. https://study.com/academy/course/computer-science-307-software-engineering.html, 2018. Accessed: 2019-05-11.

[43] MANHÃES M. M, SEBASTIAN A.S, V. M. D. L., AND T., R. UUV simulator: A gazebo-based package for underwater intervention and multi-robot simulation. In *OCEANS 2016 MTS/IEEE Monterey* (sep 2016), IEEE.

[44] MARCO D.B., H. A., AND R.B., M. Autonomous underwater vehicles: Hybrid control of mission and motion. *In Proc. Autonomous Robots. 3* (1996), 169–186.

[45] MATHWORKS. What is camera calibration? https://www.mathworks.com/help/vision/ug/camera-calibration.html, 2019. Accessed: 2019-30-12.

[46] MATSEBE, O. A review of virtual simulators for autonomous underwater vehicles (auvs), article. *NGCUV 2008, Lakeside Hotel, Killaloe, Ireland 1* (2008), 1–17.

[47] MAYBECK, P. Stochastic models, estimation and control. *Academic Press, Inc., New Tork, USA* (1979).

[48] MCGANN C., PY F., R. K. T-rex: A model-based architecture for auv control. *In Proceedings from 2008 IEEE International Conference on Robotics and Automation*, 2008. Accessed: 2019-30-11.

[49] MCPHAIL S.D., P. M. Autosub-1. a distributed approach to navigation and control of autonomous underwater vehicle. *International Conference on Electronic Engineering in Oceanography 1* (1997).

[50] MEGHALEE, G. Architectural patterns for distributed systems definition importance. Available at https://study.com/academy/course/computer-science-307-software-engineering.html, 2018. Accessed: 2019-29-11.

[51] MOORE, T. Robot localization wiki. Available at http://docs.ros.org/melodic/api/robot_localization/html/index.html, 2019. Accessed: 2019-11-21.

[52] MOORE T., S. D. A generalized extended kalman filter implementation for the robot operating system. *IAS13* (2013).

[53] MOORE T., S. D. A generalized extended kalman filter implementation for the robot operating system. In *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)* (July 2014), Springer.

[54] MORENO, T. Lecture notes from artificiall intelligence —— - multi-agent system. lecture 2: Agent architectures. Available at https://www.slideshare.net/ToniMorenoURV/agent-architectures-3181662, 2010. Accessed: 2019-30-11.

[55] M.S. FADALI, A. V. *Ch 11 - Elements of Nonlinear Digital Control Systems. In book Digital Control Engineering*. Academic press, 2009.

[56] MURPHY, K. A brief introduction to bayes' rule. Available at https://www.cs.ubc.ca/~murphyk/Bayes/bayesrule.html, 2003. Accessed: 2019-31-10.

[57] NASA. Nasa technology roadmaps. *TA 4: Robotics and Autonomous Systems 1*, TA4 (2015).

[58] NEGENBORN, R. *Robot Localization and Kalman Filters. On finding your position in a noisy world*. Thesis, Utrecht University, 2003.

[59] NEWMAN, P. Moos - a mission oriented operating suite. technical report. *Massachusetts Institute of Technology 1* (2002).

[60] NEWMAN, P. Moos - mission orientated operating suite. *Article from Department of Ocean Engineering Massachusetts Institute of Technology* (2002).

[61] NORTEK. Nortek dvl1000. Product catalog https://www.nortekgroup.com/products/dvl-1000-300m, 2019. Accessed: 2019-30-11.

[62] OCEANIC, N., AND ADMINISTRATION, A. What is an auv? https://oceanexplorer.noaa.gov/facts/auv.html, 2008. Accessed: 2019-03-20.

[63] OCÓN, J. Autonomous frameworks architectures. Available at http://bit.do/autonomous-frameworks-architectures, 2010. Accessed: 2019-30-11.

[64] OLIVEIRA P., PASCOAL A., S. V., AND C., S. The mission control system of marius auv: System design, implementation, and tests at sea. *Int. J. on Sys. Science - Special Issue on Underwater Robotics, 29. 1* (1995), 1065–1080.

[65] OPENCV. Camera calibration and 3d reconstruction. https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html, 2014. Accessed: 2019-30-12.

[66] PALOMERAS N., EL-FAKDI A., C. M. Cola2: A control architecture for auvs. *IEEE Journal of Oceanic Engineering 37*, 4 (2012).

[67] PALOMERAS N., RIDAO P., C. M., AND J., B. a mission control language for auvs. *IFAC Proceedings Volumes 40, Issue 17* (2007), 123–128.

[68] PANISH R., T. M. Achieving high navigation accuracy using inertial navigation systems in autonomous underwater vehicles. *Proceedings of the 2011 IEEE OCEANS; Santander, Spain.* (2011), 6–9.

[69] PERRETT J.R., P. M. Autosub-1. implications of using distribuited system architectures in auv development. *In Proc. of the 7th International Conference on Electronic Engineering in Oceanography. 1* (1997).

[70] PHILIPS J.L, N. D. A biologically inspired working memory framework for robots. *IEEE International Workshop on Robot and Human Interactive Communication.* (2005).

[71] PIROBOTS. Behavior trees: Simple yet powerful ai for your robot. Article: https://www.pirobot.org/blog/0030/, 2020. Accessed: 2020-01-05.

[72] PORT, T. Centralized vs decentralized vs distributed networks + blockchain. Available at http://bit.do/networks-blockchain, 2018. Accessed: 2019-29-11.

[73] ROBOSUB. International robosub competition. Available at https://www.robonation.org/competition/robosub, 2018. Accessed: 2019-03-24.

[74] ROBOTICS, B. Blue robotics low-light hd usb camera. Product catalog https://bluerobotics.com/store/sensors-sonars-cameras/cameras/cam-usb-low-light-r1/, 2019. Accessed: 2019-30-11.

[75] ROBOTICS, B. Blue robotics t200 thruster. Product catalog https://bluerobotics.com/store/thrusters/t100-t200-thrusters/t200-thruster/, 2019. Accessed: 2019-30-11.

[76] ROUMELIOTIS, S. Reliable mobile robot localization. *Phd Thesis, Electrical Engineering, faculty of the graduate school university of Southern California* (2000).

[77] RUSSELL B.W., V. A. Autonomous underwater vehicles (auvs): Their past, present and future contributions to the advancement of marine geoscience. *Marine Geology 352* (2014), 451–468.

[78] RUSSELL S.J, N. P. *Artificial Intelligence: A Modern Approach (2nd ed.)*. Pearson, 2016.

[79] S., H. Robust hybrid heading control of autonomous ships. *NTNU Master's Thesis* (2019).

[80] SAMEK, M. *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems.* Elsevier Inc, 2009.

[81] SAMEK, M. "who moved my state?". *C/C++ Users Journal, The Embedded Angle column.* (April 2003).

[82] SENSONOR. Sensonor stim300 imu. Product catalog https://www.sensonor.com/products/inertial-measurement-units/stim300/, 2019. Accessed: 2019-30-11.

[83] SINGHAL, A. Issues in autonomous mobile robot navigation. *Article, Computer Science Department University of Rochester* (1997).

[84] SKJETNE R., FOSSEN T.I., B. M. Line-of-sight path following of underactuated marine craft. *Conference: Proceedings of the 6th IFAC MCMC, At Girona, Spain* (2003).

[85] SKJETNE R., FOSSEN T.I., K. P. Robust output maneuvering for a class of nonlinear systems. *Article in Automatica 40(3):373-383* (2004).

[86] STEEN, S. *Marin Teknikk 3 - Hydrodynamikk. Motstand og propulsjon, page 9.* Akademika, 20104.

[87] STURRERS L., LIU H., T. C. B. D. Navigation technologies for autonomous underwater vehicles. *IEEE Transactions on Systems, Man, and Cybernetics 38* (2008), 581 – 589.

[88] SYSTEMS, F. Flir blackfly s camera. Product catalog https://www.flir.com/products/blackfly-s-usb3/, 2019. Accessed: 2019-30-11.

[89] SØRENSEN, A. *Lecture Notes, TMR4240, Marine Control Systems. Propulsion and Motion Control of Ships and Ocean Structures*. Akademika Forlag, 2014.

[90] SØRENSEN, A. Automatic versus autonomous. *PowerPoint Presentation, Centre for Autonomous Marine Operations and Systems, NTNU AMOS* (2019).

[91] TAYLOR, J. An introduction to error analysis. *University Science Books, Sausalito, CA* (1997).

[92] TECHNOLOGIES, B. Centralized vs decentralized vs distributed systems. Available at https://berty.tech/blog/decentralized-distributed-centralized, 2019. Accessed: 2019-29-11.

[93] TECHOPEDIA. Embedded linux. Available at https://www.techopedia.com/definition/29946/embedded-linux, 2019. Accessed: 2019-05-06.

[94] THRUN, S. Bayesian landmark learning for mobile robot localization. *Machine Learning*, 33 (1998), 41–76.

[95] THRUN, S. Simultaneous localization and mapping. *Robotics and Cognitive Approaches to Spatial Mapping 38* (2007), 13–41.

[96] THRUN S., FOX D., B. W. Markov localization for mobile robots in dynamic environments. *Journal of Artificial Intelligence Research*, 11 (1999), 391–427.

[97] TURNER, R. Intelligent adaptive reasoning for autonomous underwater vehicle control. *In Proc. of the FLAIRS95 International Workshop on Intelligent Adaptive Systems, 1* (1995).

[98] WOOLDRIDGE, M. *An Introduction to Multiagent Systems. Second Edition*. John Wiley & Sons, Inc., New York, NY, USA, 2009.

[99] ÇENGEL Y.A, C. J. *Fluid mechanics : fundamentals and applications*. McGraw-Hill, 2010.

Appendices

APPENDIX A
GITHUB PROJECT

A.1 URLs

Code	URLs
Gazebo robot simulator UUV Simulator	https://github.com/vortexntnu/uuv-simulator
Extended Kalman filter robot localization	https://github.com/Sollimann/Manta_v1/tree/master/manta-auv/localization/robot_localization
Computer vision object detection	https://github.com/Sollimann/Manta_v1/tree/master/manta-auv/perception
Vortex thrust allocation	https://github.com/Sollimann/Manta_v1/tree/master/manta-auv/control/vortex_allocator
Nonlinear PID controller	https://github.com/vortexntnu/manta-auv/tree/master/control/dp_controller
Nonlinear backstepping controller	https://github.com/vortexntnu/manta-auv/tree/master/control/autopilot/src/backstepping
Battery simulator	https://github.com/vortexntnu/manta-auv/tree/master/risk/anomaly_detection/scripts
Finite state machine SMACH	https://github.com/vortexntnu/manta-auv/tree/master/mission/finite_state_machine/agents
Computer vision reference model	https://github.com/vortexntnu/manta-auv/tree/master/guidance/reference_model
Line-of-sight guidance	https://github.com/vortexntnu/manta-auv/tree/master/guidance/los_guidance/scripts

Table A.1: URLs to the Github project.

APPENDIX B

UNDERWATER NAVIGATION

B.1 Extended Kalman Filter Initial Values

As recommended by [51, Robot Localization, Tom Moore] the initial state estimate covariance, P_0, is given below. The values are ordered as $x, y, z, \phi, \theta, \phi, u, v, z, p, q, r, \dot{u}, \dot{v}, \dot{w}$.

$$\boldsymbol{P}_0 = \begin{bmatrix} 1e-9 & 0 & .. & .. & .. & .. & .. & 0 \\ 0 & 1e-9 & .. & .. & .. & .. & .. & : \\ : & : & 1e-9 & .. & .. & .. & .. & : \\ : & : & : & .. & .. & .. & .. & : \\ : & : & : & .. & 1e-9 & .. & .. & : \\ : & : & : & .. & .. & 1e-9 & .. & : \\ : & : & : & .. & .. & .. & 1e-9 & 0 \\ 0 & : & : & .. & .. & .. & 0 & 1e-9 \end{bmatrix}$$

As recommended by [51, Robot Localization, Tom Moore] the initial process noise covariance, Q_0, is given below. The values are ordered as $x, y, z, \phi, \theta, \phi, u, v, z, p, q, r, \dot{u}, \dot{v}, \dot{w}$.

$$\boldsymbol{Q}_0 =$$

$$\begin{bmatrix}
.05 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & .05 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & .06 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & .03 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & .03 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & .06 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .025 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & .025 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .04 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .01 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .01 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .02 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .01 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .01 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .015
\end{bmatrix}$$

APPENDIX C

GUIDANCE AND MOTION CONTROL

C.1 Guidance System

C.1.1 Step Response of PID Controller Reference Model

All Figures C.1 to C.3 are plotted with $\omega_n = 0.1$, $\zeta = 1.0$ and a sampling time of $h = 0.05[s]$.

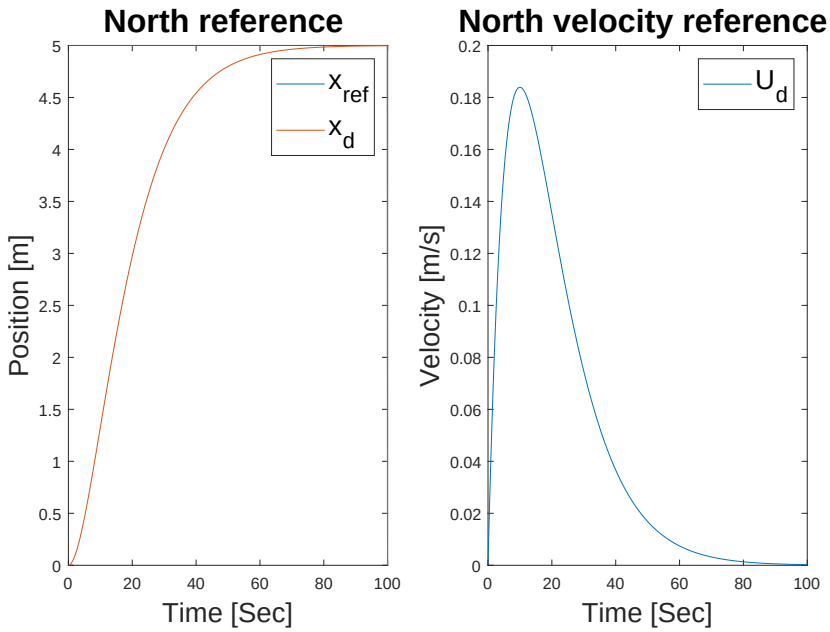


Figure C.1: Reference model in North

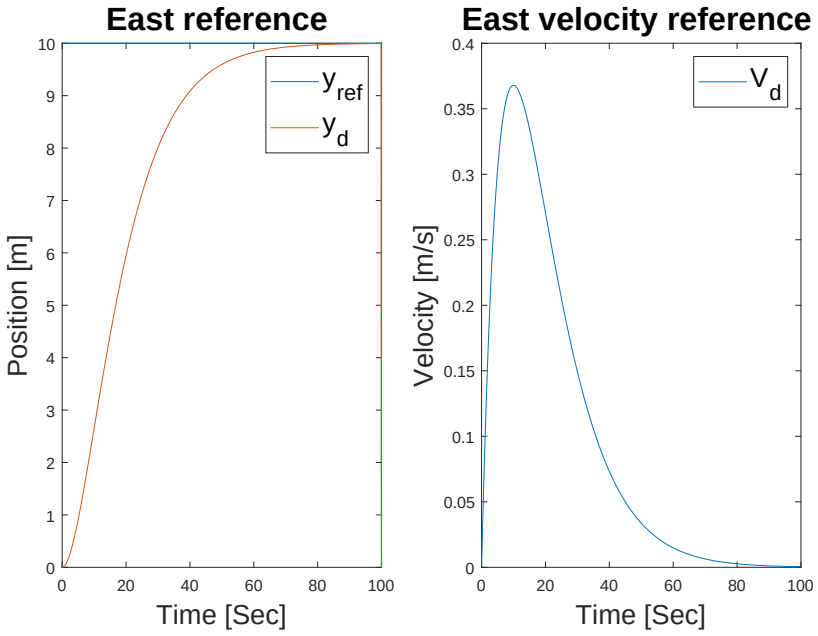


Figure C.2: Reference model in East

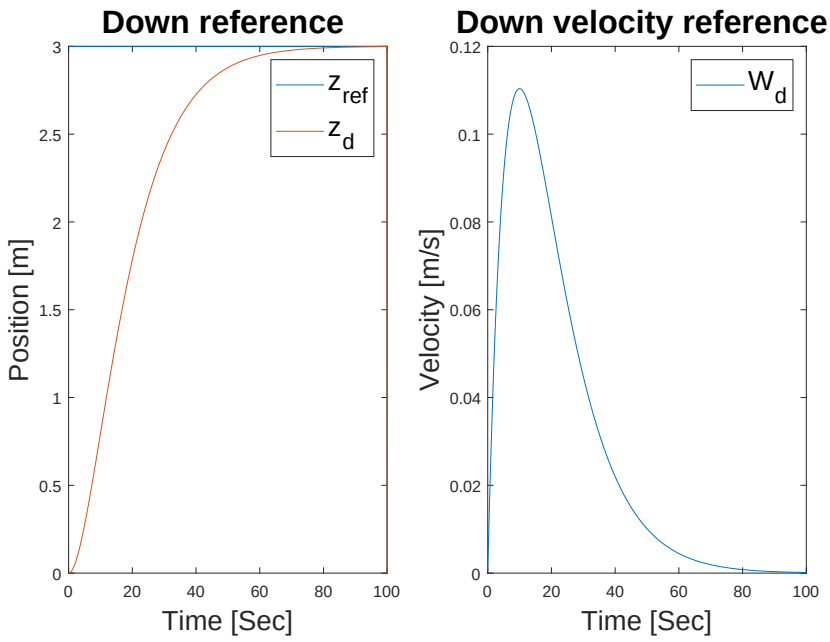


Figure C.3: Reference model in Down

C.1.2 Step Response of Backstepping Controller Reference Model

Both Figures C.4 and C.5 are plotted with $\omega_n = 0.7$, $\zeta = 1.0$ and a sampling time of $h = 0.05[s]$.

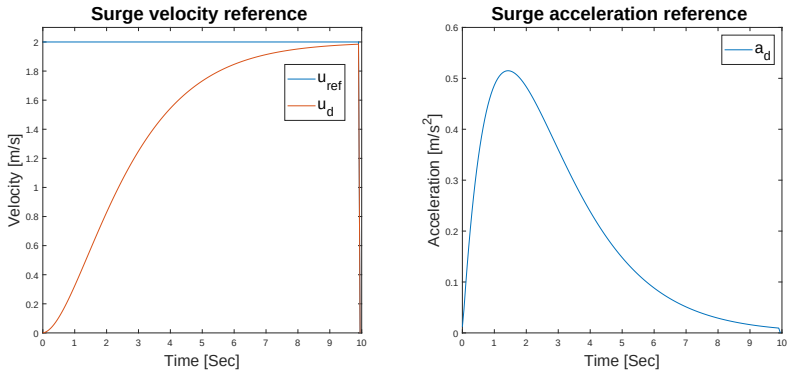


Figure C.4: Reference model in Surge

$\omega_{Yaw} = 0.7$ **and** $\zeta_{Yaw} = 1.0$

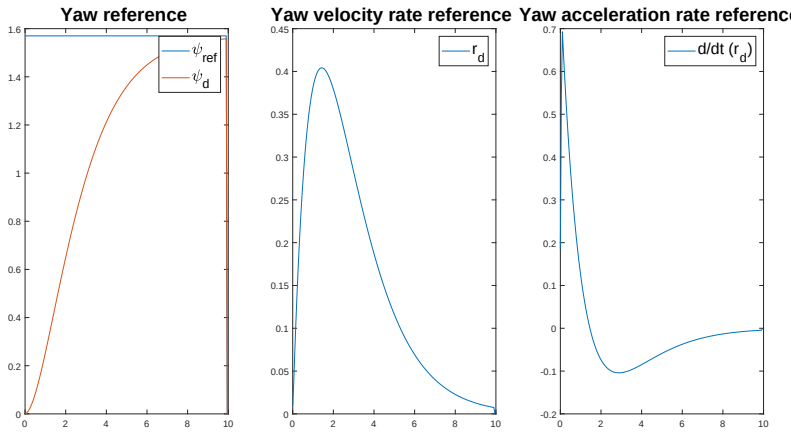


Figure C.5: Reference model in Yaw

C.2 Motion Control

C.2.1 Nonlinear PID Controller Tuning Parameteres

	P	***I***	***D***
Surge	30.5	0.024	3.5
Sway	30.5	0.024	3.5
Heave	30.5	0.024	3.5
Roll	1.3	0.13	0.0
Pitch	1.3	0.13	0.0
Yaw	1.3	0.13	0.0

Table C.1: PID gains

C.2.2 Nonlinear Backstepping Controller Tuning Parameters

Tuning in Surge, Sway and Yaw

$$\boldsymbol{M} = \begin{bmatrix} 70.7727 & 0.0 & 0.0 \\ 0.0 & 70.7727 & 0.0 \\ 0.0 & 0.0 & 1.9198 \end{bmatrix}, \qquad \boldsymbol{N}(\boldsymbol{\nu}) = \begin{bmatrix} 19.5909 & 0.0 & 0.0 \\ 0.0 & 19.5909 & 0.0 \\ 0.0 & 0.0 & 1.1559 \end{bmatrix}$$

$$c = 3.75, \quad k1 = 45.0, \quad k2 = 28.0, \quad k3 = 10.5$$

Tuning in Heave

$$P = 25.0, \quad I = 0.024, \quad D = 3.5, \quad Sat = 5.0$$

C.2.3 Camera PID Controller Tuning Parameters

In Table C.2 p_x denotes the desired pixel x-direction in the image, p_y denotes the desired pixel y-direction in the image, z denotes the desired depth and u denotes the desired speed.

	P	***I***	***D***	***Sat***
p_x	0.01	0.0001	0.0	7.5
p_y	0.15	0.002	0.0	0.25
z	25.0	0.024	3.5	5.0
u	25	0.024	3.5	5.0

Table C.2: Camera PID gains

APPENDIX D

CODE

D.1 Matlab - Hydrodynamics

Listing D.1: Computation of Manta AUV Hydrodynamics

```
clear all;
clc;

% ------------------------------------------------------------------
% Author:     Kristoffer Rakstad Solberg
% Date:       2019-05-26
% Revisions:
% ------------------------------------------------------------------
%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% Geometry, mass and intertia %%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%l = 0.44800;
%b = 0.4480;
%l = 0.50800;
%b = 0.40800;
D = 0.720;
H = 0.160;
xG = 0;
```

```
yG = 0;
zG = 0.1;

% mass [kg]
m = 18.0;

% Total surface area [m2]
S = 3.4;

% moment of inertia
% [ kg m2 ]
ixx =  0.503217;
iyy =  0.493449;
izz =  0.919819;
ixy =  0.000204;
iyx = ixy;
ixz = -0.000526;
izx = ixz;
iyz =  0.000038;
izy = iyz;

I0 = [ixx, -ixy, -ixz;
     -iyx,  iyy, -iyz;
     -izx, -izy,  izz];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% Rigid-Body Inertia %%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Fossen, 2011, Page 52

mI3x3 = m*eye(3);

mSkew = [0 -m*zG m*yG;
         m*zG 0 -m*xG;
         -m*yG m*xG 0];

M_RB = [mI3x3 -mSkew;
        mSkew     I0]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% Added Mass %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% MIT, Values for added mass, table B6 and B7

A_oblate = added_mass_oblate_spheroid(D/2, H/2, 2.0);
```

```
A_disk = added_mass_oblate_spheroid(D/2, H/2, 4.0);
relative_difference(A_oblate,A_disk);

wx = 0.5;
wy = 0.5;
wz = 1.0;
wp = 0.5;
wr = 0.5;
wh = 0.5;

weights = [wx, 0, 0, 0, 0, 0;
           0, wy, 0, 0, 0, 0;
           0, 0, wz, 0, 0, 0;
           0, 0, 0, wp, 0, 0;
           0, 0, 0, 0, wr, 0;
           0, 0, 0, 0, 0, wh];


I = eye(6);

A = A_oblate * weights + A_disk * (I - weights)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%% Hydrodynamic Damping %%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Quadratic Damping  %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Fossen, 2011, page 125-126

u = 0.7;
v = u;
w = 0.3;
k = 0.1;
%% Surge and Sway
Xuu = nonlinear_viscous_surge_damping(u,S,D,k);
Yvv = Xuu;


%% Heave
%% Eidsvik, 2011, Master Thesis
%% Faltinsen, 1990, Morison's equation
```

```
112
113 Cd_2D = 1.9;
114 Cd_3D = 1.1;
115 lambda = Cd_3D / Cd_2D;
116 Ap = 0.40;
117
118 Zww = -0.5*1025*Cd_3D*lambda*Ap*w;
119
120
121
122 DNL = -[Xuu*u, 0, 0, 0, 0, 0;
123         0, Yvv*v, 0, 0, 0, 0;
124         0, 0, Zww*w, 0, 0, 0;
125         0, 0, 0, 0, 0, 0;
126         0, 0, 0, 0, 0, 0;
127         0, 0, 0, 0, 0, 0]
128
129 %%%%%%%%%%%%%%%%%%%%%%%%%
130 %%%  Linear Damping  %%%
131 %%%%%%%%%%%%%%%%%%%%%%%%%
132 %% Fossen, 2011, page 124-125
133
134 DL_oblate = linear_viscous_damping(A_oblate, M_RB);
135 DL_disk = linear_viscous_damping(A_disk, M_RB);
136 relative_difference(DL_oblate,DL_disk)
137 DL = linear_viscous_damping(A, M_RB)
138
139
140 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
141 %%%%%%%%%%%%%%%%%%%%%%% FUNCTIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
142 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
143
144 %% General functions %%
145
146 function Re = reynolds_number(U, D)
147
148     % Kinematic viscosity 20 deg
149     % sea water
150     Re = (U * D)/(1.00 * 10^(-6));
151
152 %     if (Re < 2300)
153 %         disp('Laminar flow')
154 %     elseif (Re > 4000)
155 %         disp('Turbulent flow')
156 %     else
```

```
%          disp('Transient flow')
%      end

end

%% ADDED MASS %%

function A = added_mass_sphere( r )

rho = 1025;

A = [(2/3) * rho * pi * (r^3), 0, 0, 0, 0, 0;
     0, (2/3) * rho * pi * (r^3), 0, 0, 0, 0;
     0, 0, (2/3) * rho * pi * (r^3), 0, 0, 0;
     0, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0;];

end

function A = added_mass_prolate_ellipsoid(a, b, c, mass)

% eccentricity
e = 1-(b/a)^2;

% constants / ln(x) == log(x) in matlab
alpha0 = ( 2*(1-e^2)/(e^3) ) * ( (1/2)*log((1+e)/(1-e))-e );
beta0 = 1/(e^2) - (1-e^2)/(2*(e^3))*log((1+e)/(1-e));

% Diagonal added mass derivatives (cross-coupling terms
% will be zero due to body symmetry about three planes

Xud = - ( (alpha0)/(2-alpha0) ) * mass;
Yvd = - ( (beta0)/(2-beta0) ) * mass;
Zwd = Yvd;
Kpd = 0;
Nrd = -(1/5)*( (((b^2 - a^2)^2)*(alpha0-beta0))/(2*(b^2-a^2)
    + (b^2+a^2)*(beta0-alpha0)) )*mass;
Mqd = Nrd;

A = -[Xud, 0, 0, 0, 0, 0;
      0, Yvd, 0, 0, 0, 0;
      0, 0, Zwd, 0, 0, 0;
      0, 0, 0, Kpd, 0, 0;
      0, 0, 0, 0, Mqd, 0;
```

```
      0, 0, 0, 0, 0, Nrd];

end


% Assuming length > height
% table B6
% 2.0 < r/h < 4.0
function A = added_mass_oblate_spheroid(r, h, ratio)

% water density [kg/m3]
rho = 1025;

% Lamb's k-factor
% symmetry

if (ratio == 2.0)
% r/h = 2.0
    kx = 0.310;
    kz = 1.118;
    kr = 0.337;

else
% r/h = 4.0
    kx = 0.174;
    kz = 2.379;
    kr = 1.330;

end

Xud = -kx*(4/3)*rho*pi*h*(r^2);
Yvd = Xud;
Zwd = -kz*(4/3)*rho*pi*h*(r^2);
Kpd = -kr*(4/15)*rho*pi*h*(r^2)*(h^2 + r^2);
Mqd = Kpd;
Nrd = 0;

A = -[Xud, 0, 0, 0, 0, 0;
      0, Yvd, 0, 0, 0, 0;
      0, 0, Zwd, 0, 0, 0;
      0, 0, 0, Kpd, 0, 0;
      0, 0, 0, 0, Mqd, 0;
      0, 0, 0, 0, 0, Nrd];

end
```

```
247
248
249 %% DAMPIMG %%
250
251 function DL = linear_viscous_damping(A, M_RB)
252
253 Kp = 12;
254 Kd = 6;
255 Zeta33 = 0.2;
256 Zeta44 = 0.7;
257 Zeta55 = Zeta44;
258 Zeta66 = 0.1;
259
260 % Time constants and natural periods
261 T11 = M_RB(1,1) / Kd;
262 T22 = M_RB(2,2) / Kd;
263 T33n = 4.0;
264 T44n = 1.0;
265 T66n = 2.0;
266
267 % Natural frequency and damping ratio
268 w33n = (2*pi) / T33n;
269 w44n = (2*pi) / T44n;
270 w55n = w44n;
271
272 % Damping
273 B11v = (M_RB(1,1) + A(1,1)) / (T11);
274 B22v = (M_RB(1,1) + A(1,1)) / (T22);
275 B33v = 2*Zeta33*w33n*(M_RB(3,3) + A(3,3));
276 B44v = 2*Zeta44*w44n*(M_RB(4,4) + A(4,4));
277 B55v = 2*Zeta55*w55n*(M_RB(5,5) + A(5,5));
278 B66v = ( 8*pi*Zeta66*( M_RB(6,6) + A(6,6) ) )/(T66n);
279
280 DL =  [B11v, 0, 0, 0, 0, 0;
281        0, B22v, 0, 0, 0, 0;
282        0, 0, B33v, 0, 0, 0;
283        0, 0, 0, B44v, 0, 0;
284        0, 0, 0, 0, B55v, 0;
285        0, 0, 0, 0, 0, B66v];
286
287 end
288
289 function Xuu = nonlinear_viscous_surge_damping(u,S,D,k)
290
291 rho = 1025;
```

```
292 Rn = reynolds_number(u,D);
293
294 Cf = (0.075) / (log10(Rn)-2)^2;
295 Xuu = -0.5 * rho * S*(1+k)*Cf*abs(u);
296
297 end
298
299 function Cd = drag_coeff_sphere(Re)
300
301 Table = [10^2 10^3 10^4 10^5 10^6 5*10^6;
302          1.0  0.41 0.39 0.52 0.12 0.18];
303
304 for i = 1:6
305
306     if Re >= Table(1,i)
307        Cd = Table(2,i);
308     end
309
310 end
311
312 end
313
314 function Cd = drag_coeff_ellipsoid(L, D, Re)
315
316 % Assuming Re < 2 x 10^5
317 % Assuming 1 < L/D < 10
318
319 if (Re <= 2 * 10^5) && (1 < L/D < 10)
320     Cd = 0.44*(D/L) + 0.016*(L/D) + 0.016*sqrt(D/L);
321 elseif (Re > 2 * 10^5)
322     disp('Cannot calculate drag coeff due to large Re')
323 elseif ~(1 < L/D < 10)
324     disp('Cannot calculate drag coeff due to aspect ratio')
325 end
326
327 end
328
329 function Kuu = quad_damping_sphere(Cd, r)
330 rho = 1025;
331 xuu = -0.5*rho*Cd*pi*r^2;
332 nul = 0;
333
334 Kuu = -[xuu 0 0 0 0 0;
335        0 xuu 0 0 0 0;
336        0 0 xuu 0 0 0;
```

```
        0 0 0 nul 0 0;
        0 0 0 0 nul 0;
        0 0 0 0 0 nul];
end

%% Relative Difference %%
function diff = relative_difference(M1,M2)
diff = [0,0,0,0,0,0]';

    for i = 1:6
        diff(i) = (M1(i,i) - M2(i,i)) / M2(i,i);
    end

end
```

D.2 Matlab - Thruster Dynamics

Listing D.2: Computation of Manta AUV thruster dynamics

```
clear all;
clc;

% ------------------------------------------------------------------
% Author:     Kristoffer Rakstad Solberg
% Date:       2019-05-31
% Revisions:
% ------------------------------------------------------------------
%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% Conversion function %%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


load T200.mat
x = T200(:,1);
y = T200(:,2);
xRad = x * (2*pi/60);
yFit = conv_func(x, 0.000004);
yFit2 = conv_func(x,0.000001);
yFit3 = conv_func(x,0.000007);

figure(1)
```

```
hold on;
plot(xRad,y);
plot(xRad,yFit, 'g');
plot(xRad,yFit2, 'r');
title('Thruster T200 - rad/s ')
ylabel('Thrust [N]')
xlabel('Radians pr Second [rad/s]')
legend('Actual','k_{rotor} = 0.000004', ...
    'k_{rotor} = 0.00001')
grid on


figure(2)
hold on;
plot(x,y);
plot(x,yFit,'g');
plot(x,yFit2,'r');
title('Thruster T200 - RPM ')
ylabel('Thrust [N]')
xlabel('Radians pr Second [rad/s]')
legend('Actual','k_{rotor} = 0.000004', ...
    'k_{rotor} = 0.00001' )
ylabel('Thrust [N]')
xlabel('Rounds Per Minute [RPM]')
grid on
hold off;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%% Rotor Time constant %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

T1 = 0.4;
T2 = 0.2;
RPMd = 100;
dyn1 = tf([RPMd],[T1 1]);
dyn2 = tf([RPMd],[T2 1]);

figure(3)
hold on;
step(dyn1);
step(dyn2);
title('Thruster T200 - Time Constant ')
ylabel('RPM')
xlabel('Time [s]')
legend('T = 4s', 'T = 2s')
```

```
70
71
72  function yFit = conv_func(x, rotor_constant)
73
74  yFit = [];
75
76      for i = 1:length(x)
77          yFit(i) = rotor_constant * x(i) * abs(x(i));
78      end
79
80  end
```

D.3 Matlab - Reference Model

Listing D.3: Discrete reference model using Tustin's method

```
1   clear all;
2   clc;
3
4   % ____________________________________________________________
5   % Author:      Sondre Haug
6   % Date:        2019-06-05
7   % Revisions:   Kristoffer Rakstad Solberg
8   % ____________________________________________________________
9   %
10
11  T = 100;     % Time interval
12  h = 0.05;    % Sampling time
13  N = T/h;     % Number of samples
14
15  % North Reference
16  y_ref_vec = zeros(1, N);
17  y_d_vec = zeros(1,N);
18  y_d_dot_vec = zeros(1,N);
19
20  % East Reference
21  x_ref_vec = zeros(1,N);
22  x_d_vec = zeros(1,N);
23  x_d_dot_vec = zeros(1,N);
24  time_vec = 0:h:(T-h);
25
26  % Down Reference
27  z_ref_vec = zeros(1,N);
```

```
z_d_vec = zeros(1,N);
z_d_dot_vec = zeros(1,N);
time_vec = 0:h:(T-h);

%% Reference model
x_ref = 0;          % Initial command
x_ref_prev = 0;
x_ref_prev_prev = 0;
x_ref_0 = 5.0;

x_d = 0;
x_d_prev = 0;
x_d_prev_prev = 0;
omega_n_x = 0.1;
zeta_x = 1.0;
[a_sf_x, b_sf_x] = ...
    reference_model_coefficients(omega_n_x, zeta_x, h);

y_ref = 0;          % Initial command
y_ref_prev = 0;
y_ref_prev_prev = 0;
y_ref_0 = 10;

y_d = 0;
y_d_prev = 0;
y_d_prev_prev = 0;
omega_n_y = 0.1;
zeta_y = 1.0;
[a_sf_y, b_sf_y] = ...
    reference_model_coefficients(omega_n_y, zeta_y, h);

z_ref = 0;          % Initial command
z_ref_prev = 0;
z_ref_prev_prev = 0;
z_ref_0 = 3;

z_d = 0;
z_d_prev = 0;
z_d_prev_prev = 0;
omega_n_z = 0.1;
zeta_z = 1.0;
[a_sf_z, b_sf_z] = ...
    reference_model_coefficients(omega_n_z, zeta_z, h);

for i = 1:(N-1)
```

```
% Inside loop

    y_ref_prev_prev = y_ref_prev;
    y_ref_prev = y_ref;
    y_ref = y_ref_0;

    % Filtering the signal that is fed into the controller:
    y_d_prev_prev = y_d_prev;
    y_d_prev = y_d;
    y_d = b_sf_y(1) * y_ref + b_sf_y(2) * y_ref_prev + ...
        b_sf_y(3) * y_ref_prev_prev - ...
        a_sf_y(2) * y_d_prev - a_sf_y(3) * y_d_prev_prev;
    r_d_dot = (y_d - y_d_prev) / h;


    x_ref_prev_prev = x_ref_prev;
    x_ref_prev = x_ref;
    x_ref = x_ref_0;

    % Filtering the signal that is fed into the controller:
    x_d_prev_prev = x_d_prev;
    x_d_prev = x_d;
    x_d = b_sf_x(1) * x_ref + b_sf_x(2) * x_ref_prev ...
        + b_sf_x(3) * x_ref_prev_prev - ...
        a_sf_x(2) * x_d_prev - a_sf_x(3) * x_d_prev_prev;
    x_d_dot = (x_d - x_d_prev) / h;

    z_ref_prev_prev = z_ref_prev;
    z_ref_prev = z_ref;
    z_ref = z_ref_0;

    % Filtering the signal that is fed into the controller:
    z_d_prev_prev = z_d_prev;
    z_d_prev = z_d;
    z_d = b_sf_z(1) * z_ref + b_sf_z(2) * z_ref_prev ...
        + b_sf_z(3) * z_ref_prev_prev - ...
        a_sf_z(2) * z_d_prev - a_sf_z(3) * z_d_prev_prev;
    z_d_dot = (z_d - z_d_prev) / h;


    z_d_vec(i) = z_d;
    z_d_dot_vec(i) = z_d_dot;
    z_ref_vec(i) = z_ref;

    y_d_vec(i) = y_d;
```

```
    y_d_dot_vec(i) = r_d_dot;
    y_ref_vec(i) = y_ref;

    x_d_vec(i) = x_d;
    x_d_dot_vec(i) = x_d_dot;
    x_ref_vec(i) = x_ref;

end

figure(1)
subplot(1,2,1);
plot(time_vec, x_ref_vec, time_vec, x_d_vec)
title('North reference')
legend('x_{ref}','x_d')

subplot(1,2,2);
plot(time_vec, x_d_dot_vec)
title('North velocity reference')
legend('U_d')

figure(2)
subplot(1,2,1);
plot(time_vec, y_ref_vec, time_vec, y_d_vec)
title('East reference')
legend('y_{ref}','y_{d}')

subplot(1,2,2);
plot(time_vec, y_d_dot_vec)
title('East velocity reference')
legend('V_d')


figure(3)
subplot(1,2,1);
plot(time_vec, z_ref_vec, time_vec, z_d_vec)
title('Down reference')
legend('z_{ref}','z_{d}')

subplot(1,2,2);
plot(time_vec, z_d_dot_vec)
title('Down velocity reference')
legend('W_d')

function [a, b] = ...
    reference_model_coefficients(omega_n, zeta, Ts)
```

```
163
164 system_cont = tf([0, 0, omega_n^2], ...
165     [1, 2 * zeta * omega_n, omega_n^2]);
166 system_disc = c2d(system_cont, Ts, 'tustin');
167
168
169 % Which is the digial version of the TF
170 % omega_n^2 / (s^2 + 2*zeta*omega_n +
171 % omega_n^2) with sampling time h with tustin
172 %         b1 * z^2 + b2 * z + b3
173 %    ---------------------------------
174 %           z^2 + a1 * z + a2
175 %
176 % Sample time: 0.01 seconds
177 % Discrete-time transfer function.
178
179 a = system_disc.Denominator{1};
180 b = system_disc.Numerator{1};
181
182 end
```

D.4 Matlab - Rosbag Record

Listing D.4: A script for interpreting and plotting rosbag

```
1  clear all;
2  close all;
3
4  % ____________________________________________________________
5  % Author:     Kristoffer Rakstad Solberg
6  % Date:       2019-05-26
7  % Revisions:
8  % ____________________________________________________________
9  %
10
11 % Target points
12 x_d = [5.0, 5.0, 8.0, 8.0, 5.0];
13 y_d = [-10.0, -3.0, -3.0, -10.0, -10.0];
14 z_d = [0.0, -3.0, -3.0, -3.0, 0.0];
15
16 % import rosbag
17 bag = rosbag('pdcontroller3.bag');
18 bagselect1 = select(bag, 'Topic', '/manta/pose_gt');
```

```
bagselect2 = select(bag, 'Topic', ...
    '/manta/thruster_manager/input');

% topics
% /manta/pose_gt
pose_gt = timeseries(bagselect1, 'Pose.Pose.Position.X', ...
    'Pose.Pose.Position.Y', 'Pose.Pose.Position.Z');


attitude_quat = timeseries(bagselect1, ...
    'Pose.Pose.Orientation.X','Pose.Pose.Orientation.Y',...
    'Pose.Pose.Orientation.Z', 'Pose.Pose.Orientation.W');
attitude_euler = (180/pi)*quat2eul([attitude_quat.Data(:,1)
    attitude_quat.Data(:,2) ...
    attitude_quat.Data(:,3) attitude_quat.Data(:,4)]);

vel_gt = timeseries(bagselect1, 'Twist.Twist.Linear.X', ...
    'Twist.Twist.Linear.Y', 'Twist.Twist.Linear.Z');
ang_vel_gt = timeseries(bagselect1, ...
    'Twist.Twist.Angular.X', 'Twist.Twist.Angular.Y', ...
    'Twist.Twist.Angular.Z');
force_d = timeseries(bagselect2, 'Force.X', 'Force.Y', ...
    'Force.Z');
torque_d = timeseries(bagselect2, 'Torque.X', ...
    'Torque.Y', 'Torque.Z');
% thrust = timeseries(bagselect3, 'Data');

% Position and attitude

figure();
hold on
scatter3(x_d(1,:), y_d(1,:), z_d(1,:), '*');
plot3(pose_gt.Data(:,1), pose_gt.Data(:,2),...
    pose_gt.Data(:,3),'g');
title('Position ground truth')
ylabel('Position Y [m]')
xlabel('Position X [m]')
zlabel('Position Z [m]')
grid on
hold off

% Velocities

figure()
subplot(2,1,1);
```

```
plot(vel_gt);
title('Velocity ground truth')
ylabel('Velocity [m/s]')
xlabel('Time [sec]')
legend('velX','velY','velZ')
grid on

subplot(2,1,2);
plot(force_d);
title('Desired force in body frame')
ylabel('Force [N]')
xlabel('Time [sec]')
legend('Surge','Sway','Heave')
grid on

% Forces and torques


figure()
subplot(3,1,1);
plot(pose_gt.Time(:,1),attitude_euler);
title('Attitude ground truth')
ylabel('Angle [deg]')
xlabel('Time [sec]')
legend('yaw','pitch','roll')
grid on

subplot(3,1,2);
plot(ang_vel_gt);
title('Angular velocity ground truth')
ylabel('Angular velocity [rad/s]')
xlabel('Time [sec]')
legend('roll','pitch','yaw')
grid on

subplot(3,1,3);
plot(torque_d);
title('Desired moments in body frame')
ylabel('Torque [Nm]')
xlabel('Time [sec]')
legend('Roll','Pitch','Yaw')
grid on
```

D.5 UUV Simulator - Manta AUV Rigid-Body Kinetics Configuration File

```
<!-- Copyright (c) 2016 The UUV Simulator Authors.
     All rights reserved.

     Contributed by Kristoffer Rakstad Solberg,
     Student2019 Manta AUV, Vortex NTNU. -->

<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
  <!-- Includes -->
  <xacro:include filename="$(find manta_description)/urdf/common.urdf.xacro"/>
  <xacro:include filename="$(find uuv_sensor_ros_plugins)/urdf/sensor_snippets.xacro"/>
  <xacro:include filename="$(find uuv_gazebo_ros_plugins)/urdf/snippets.xacro"/>
<!--
    Vehicle's parameters (remember to enter the model parameters below)
  -->
  <!-- Mass -->
  <xacro:property name="mass" value="16.6"/>
  <!-- Describing the dimensions of the vehicle's bounding box: width, length, height -->
  <xacro:property name="x_size" value="0.448"/>
  <xacro:property name="y_size" value="0.444"/>
  <xacro:property name="z_size" value="0.28066"/>
  <!-- minion_usv_height is not really! It's just for run... We need first calculate the Fossen parameters -->
  <!-- Volume -->
  <xacro:property name="volume" value="0.01854"/>
  <!-- Center of gravity -->
  <xacro:property name="cog" value="0 0 -0.08"/>
  <!--
  Center of buoyancy -->
  <xacro:property name="cob" value="0 0 0.05"/>
  <!-- Fluid density -->
  <xacro:property name="rho" value="1028"/>
  <!-- Parameters -->
  <xacro:property name="namespace" value="manta"/>
  <xacro:property name="visual_mesh_file" value="file://$(find manta_description)/mesh/manta_shell.dae"/>

  <xacro:macro name="manta_base" params="namespace *gazebo inertial_reference_frame">
   <!-- Rigid body description of the base link -->
    <link name="${namespace}/base_link">
        <!--
          Be careful to setup the coefficients for the inertial tensor,
          otherwise your model will become unstable on Gazebo
        -->
        <inertial>
          <mass value="${mass}" />
          <origin xyz="${cog}" rpy="0 0 0"/>
          <inertia
            ixx="${0.503217}" ixy="${0.000204}" ixz="${-0.000526}"
            iyy="${0.493449}" iyz="${0.000038}" izz="${0.919819}" />
        </inertial>
        <!-- This visual geometry representation can be used when running
             tasks in which you need Gazebo to start quickly
        -->
        <!-- <xacro:if value="${use_simplified_mesh}"> -->
        <!-- Code in rexrov2_base.xacro if use the flag use_simplified_mesh -->
        <visual>
          <origin xyz="0 0 0" rpy="0 0 0"/>
          <geometry>
            <mesh filename="${visual_mesh_file}" scale="1 1 1" />
          </geometry>
        </visual>
        <!-- In manta_base i've made collision planes to minize the GPU load -->
    <collision>
      <origin xyz="0 0 ${0}" rpy="0 0 0"/>
      <geometry>
        <sphere radius="${0.25}"/>
      </geometry>
    </collision>
  </link>
    <gazebo reference="${namespace}/base_link">
      <selfCollide>false</selfCollide>
    </gazebo>
    <!-- Set up hydrodynamic plugin given as input parameter -->
    <xacro:insert_block name="gazebo"/>

   <!-- Attach actuators -->
  <xacro:include filename="$(find manta_description)/urdf/manta_actuators.xacro"/>

   <!-- Attach sensors -->
  <xacro:include filename="$(find manta_description)/urdf/manta_sensors.xacro"/>
```

Figure D.1: Implementation of Manta AUV thrusters

D.6 UUV Simulator - Manta AUV Hydrodynamics Configuration File

```
<!-- Copyright (c) 2016 The UUV Simulator Authors.
     All rights reserved.

     Contributed by Kristoffer Rakstad Solberg,
     Student2019 Manta AUV, Vortex NTNU. -->

<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
  <xacro:macro name="manta_hydro_model" params="namespace">
    <!-- List of hydrodynamic models this robot's links -->
    <link name="${namespace}/base_link">
      <!-- This flag will make the link neutrally buoyant -->
      <neutrally_buoyant>0</neutrally_buoyant>
      <!-- Link's volume -->
      <volume>${volume}</volume>
      <!-- Link's bounding box, it is used to recalculate the immersed
        volume when close to the surface.
        This is a workaround the invalid bounding box given by Gazebo-->
      <box>
        <width>${x_size}</width>
        <length>${y_size}</length>
        <height>${z_size}</height>
      </box>
      <!-- Center of buoyancy -->
      <center_of_buoyancy>${cob}</center_of_buoyancy>
      <!-- Fossen's equation of motion for underwater vehicles
      Reference:
          [1] Fossen, Thor I. Handbook of marine craft hydrodynamics and motion
          control. John Wiley & Sons, 2011.
     -->
      <hydrodynamic_model>
          <type>fossen</type>
          <!-- Added mass -->
          <added_mass>
            10.7727 0 0 0 0 0
            0 10.7727 0 0 0 0
            0 0 49.7679 0 0 0
            0 0 0 1.0092 0 0
            0 0 0 0 1.0092 0
            0  0  0  0  0  0
          </added_mass>
          <!--
            The linear damping coefficients can be provided as a diagonal (6 elements)
            or a full matrix (36 coefficients), like the added-mass coefficients above
          -->
          <linear_damping>
            -9.5909 -9.5909 -42.5798 -13.3040 -13.2181 -1.1559
          </linear_damping>
          <!--
            The quadratic damping coefficients can be provided as a diagonal (6 elements)
            or a full matrix (36 coefficients), like the added-mass coefficients above
          -->
          <quadratic_damping>
            -5.1386 -5.1386 -26.1105 0 0 0
          </quadratic_damping>
      </hydrodynamic_model>
    </link>
  </xacro:macro>
</robot>
```

Figure D.2: Implementation of Manta AUV hydrodynamics

D.7 UUV Simulator - Manta AUV Thruster Allocation and Dynamics Configuration File

```
<!-- Copyright (c) 2016 The UUV Simulator Authors.
     All rights reserved.

     Contributed by Kristoffer Rakstad Solberg,
     Student2019 Manta AUV, Vortex NTNU. -->

<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

<!-- Here you can put in whatever kind of model you want and it will
  still function as a propeller                                    -->
  <xacro:property name="prop_mesh_file_cw" value="file://$(find manta_description)/mesh/t200_propccw.dae"/>
  <xacro:property name="prop_mesh_file_ccw" value="file://$(find manta_description)/mesh/t200_propcw.dae"/>
  <!-- <xacro:property name="prop_mesh_file" value="file://$(find manta_description)/mesh/manipulator_claw.dae"/> -->
  <!-- Thruster joint and link snippet -->
<!-- Time constant for thruster dynamics: rpm = K / (Ts + 1)
    <rotorConstant> T200, 12V for a 2nd order curve fit: tau = gain * abs(x)*x -->
  <xacro:macro name="thruster_macro"
    params="namespace thruster_id *origin">
    <xacro:thruster_module_first_order_basic_fcn_macro
      namespace="${namespace}"
      thruster_id="${thruster_id}"
      mesh_filename="${prop_mesh_file_cw}"
      dyn_time_constant="0.2"
      rotor_constant="0.000004">
      <xacro:insert_block name="origin"/>
    </xacro:thruster_module_first_order_basic_fcn_macro>
  </xacro:macro>

<!-- Top thrusters -->

  <xacro:thruster_macro namespace="${namespace}" thruster_id="5"> <!-- heave back left prop up-->
  <origin xyz="-0.12070 0.12070 0" rpy="0 -1.5707963268 0"/>
  </xacro:thruster_macro>
  <xacro:thruster_macro namespace="${namespace}" thruster_id="2"> <!-- heave back right prop up-->
    <origin xyz="-0.12070 -0.12070 0" rpy="0 -1.5707963268 0"/>
  </xacro:thruster_macro>
  <xacro:thruster_macro namespace="${namespace}" thruster_id="6"> <!-- heave front left prop up-->
    <origin xyz="0.12070 0.12070 0" rpy="0 -1.5707963268 0"/>
  </xacro:thruster_macro>
  <xacro:thruster_macro namespace="${namespace}" thruster_id="1"> <!-- heave front right prop up-->
    <origin xyz="0.12070 -0.12070 0" rpy="0 -1.5707963268 0"/>
  </xacro:thruster_macro>

 <!-- Front thrusters -->

  <xacro:thruster_macro namespace="${namespace}" thruster_id="7"> <!-- lat front left-->
    <origin xyz="0.20506 0.20506 0" rpy="${0*d2r} ${0*d2r} ${135*d2r}"/>
  </xacro:thruster_macro>
  <xacro:thruster_macro namespace="${namespace}" thruster_id="0"> <!-- lat front right -->
    <origin xyz="0.20506 -0.20506 0" rpy="${0*d2r} ${0*d2r} ${-135*d2r}"/>
  </xacro:thruster_macro>

 <!-- Back thrusters -->

  <xacro:thruster_macro namespace="${namespace}" thruster_id="4"> <!-- lat back left-->
    <origin xyz="-0.20506 0.20506 0" rpy="${0*d2r} ${0*d2r} ${45*d2r}"/>
  </xacro:thruster_macro>
  <xacro:thruster_macro namespace="${namespace}" thruster_id="3"> <!-- lat back right -->
    <origin xyz="-0.20506 -0.20506 0" rpy="${0*d2r} ${0*d2r} ${-45*d2r}"/>
  </xacro:thruster_macro>
</robot>
```

Figure D.3: Implementation of Manta AUV thrusters

D.8 UUV Simulator - Manta AUV Sensors Configuration File

```
<!-- Copyright (c) 2016 The UUV Simulator Authors.
     All rights reserved.

     Contributed by Kristoffer Rakstad Solberg,
     Student2019 Manta AUV, Vortex NTNU. -->

<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
  <!-- DVL -->
  <xacro:default_dvl_macro
    namespace="${namespace}"
    parent_link="${namespace}/base_link"
    inertial_reference_frame="${inertial_reference_frame}">
    <origin xyz="0 0 0" rpy="0 ${0.5*pi} 0"/>
  </xacro:default_dvl_macro>

  <!-- Magnetometer  -->
  <xacro:default_magnetometer namespace="${namespace}" parent_link="${namespace}/base_link"/>
<!--
  <!- RPT   ->
  <xacro:default_rpt namespace="${namespace}" parent_link="${namespace}/base_link">
    <origin xyz="-1.32 0 0.8" rpy="0 0 0"/>
  </xacro:default_rpt> -->

  <!-- Pressure  -->
  <xacro:default_pressure_macro namespace="${namespace}" parent_link="${namespace}/base_link">
    <origin xyz="0 0 0" rpy="0 0 0"/>
  </xacro:default_pressure_macro>

  <!-- IMU, ENU -->
  <xacro:default_imu_macro
    namespace="${namespace}"
    parent_link="${namespace}/base_link"
    inertial_reference_frame="${inertial_reference_frame}">
    <origin xyz="0 0 0" rpy="0 0 0"/>
  </xacro:default_imu_macro>

  <!-- Mount a camera -->
  <xacro:default_camera namespace="${namespace}" parent_link="${namespace}/base_link" suffix="front">
    <origin xyz="0.29 0.0 0." rpy="0 0 0"/>
  </xacro:default_camera>

  <xacro:default_camera namespace="${namespace}" parent_link="${namespace}/base_link" suffix="under">
    <origin xyz="0.15 0.0 -0.06" rpy="0 ${0.5*pi} 0"/>
  </xacro:default_camera>
<!--
  <xacro:default_camera namespace="${namespace}" parent_link="${namespace}/base_link" suffix="left">
    <origin xyz="1.15 0.63 0.4" rpy="0 0.6 -0.4"/>
  </xacro:default_camera>   -->

  <!-- Mount a GPS.
  <xacro:default_gps namespace="${namespace}" parent_link="${namespace}/base_link" /> -->

  <!-- Mount a Pose 3D sensor. This will give 3D pose -->
  <xacro:default_pose_3d_macro
    namespace="${namespace}"
    parent_link="${namespace}/base_link"
    inertial_reference_frame="${inertial_reference_frame}" />
</robot>
```

Figure D.4: Implementation of Manta AUV thrusters

D.9 Gazebo - robosub_2019.world

Listing D.5: The description file for the robosub competition pool

```
<?xml version="1.0" ?>
<!--
     Written by Kristoffer Rakstad Solberg, Student
     Copyright (c) 2019 Manta AUV, Vortex NTNU.
     All rights reserved.
-->

<!-- Scenery specifics -->

<sdf version="1.5">
  <world name="empty_underwater">
  <physics name="default_physics" default="true" type="ode">
      <max_step_size>0.002</max_step_size>
      <real_time_factor>1</real_time_factor>
      <real_time_update_rate>500</real_time_update_rate>
      <ode>
        <solver>
          <type>quick</type>
          <iters>50</iters>
          <sor>0.5</sor>
        </solver>
      </ode>
    </physics>

<!-- Sky and underwater scenery -->

    <scene>
      <ambient>0.01 0.01 0.01 1.0</ambient>
      <sky>
        <clouds>
          <speed>12</speed>
        </clouds>
      </sky>
      <shadows>1</shadows>
      <fog>
        <color>0.0039 0.26 0.415 1.0</color>
        <type>linear</type>
        <density>0.2</density>
        <start>10</start>
        <end>40</end>
      </fog>
    </scene>

    <!-- Origin placed somewhere in the
           middle of the North Sea  -->
```

```
    <spherical_coordinates>
      <latitude_deg>56.71897669633431</latitude_deg>
      <longitude_deg>3.515625</longitude_deg>
    </spherical_coordinates>

    <!-- Global light source -->
    <include>
      <uri>model://sun</uri>
    </include>

    <!-- Bounding box with sea surface -->
    <include>
      <uri>model://ocean_box</uri>
      <pose>0 0 -0.1 0 0 0</pose>
    </include>

    <!-- Bounding box with sea surface -->

    <include>
      <uri>model://octagon</uri>
      <pose>15 15 -0.1 0 0 0</pose>
    </include>

    <include>
      <uri>model://pool</uri>
    </include>

    <include>
      <name>gate_side_a</name>
      <pose>12.4 -23.6 -0.9 0 0 -0.9</pose>
      <uri>model://2018/gate</uri>
    </include>

    <include>
      <name>dice_side_a</name>
      <pose>19.0 -20.2 -5.0 0 0 -0.6</pose>
      <uri>model://2018/dice</uri>
    </include>

    <include>
      <name>path_marker1_side_a</name>
      <pose>14.2 -22.6 -5.0 0 0 -1.13</pose>
      <uri>model://2018/path_markers</uri>
    </include>

```

```
    <include>
      <name>path_marker2_side_a</name>
      <pose>21.5 -18.4 -5.0 0 0 -0.85</pose>
      <uri>model://2018/path_markers</uri>
    </include>

    <include>
      <name>roulette</name>
      <pose>27.0 -4.1 -5.0 0 0 0</pose>
      <uri>model://2018/roulette</uri>
    </include> -->


   <!-- Lightning -->


    <light type="directional" name="sun4">
      <cast_shadows>1</cast_shadows>
      <pose>8 -19 10 1.75 0 2.9</pose>
      <diffuse>0.8 0.8 0.8 1</diffuse>
      <specular>0.2 0.2 0.2 1</specular>
      <attenuation>
        <range>1000</range>
        <constant>0.9</constant>
        <linear>0.01</linear>
        <quadratic>0.001</quadratic>
      </attenuation>
      <direction>-0.5 0.1 -0.9</direction>
    </light>


    <plugin name="sc_interface"
   filename="libuuv_sc_ros_interface_plugin.so"/>

  </world>
</sdf>
```

D.10 Gazebo - dice.sdf

Listing D.6: The Configuration File for the RoboSub Underwater Dices

```
<?xml version='1.0'?>
<sdf version='1.5'>
```

```
<!--
     Written by Kristoffer Rakstad Solberg, Student
     Copyright (c) 2019 Manta AUV, Vortex NTNU.
     All rights reserved.
-->

  <model name='dice'>
    <pose>0 0 0 0 0 0</pose>
    <self_collide> 0 </self_collide>
    <static> 1</static>

    <link name='one_dice'>
      <pose>0.80 0.5 0.75 0 0 0</pose>
      <visual name='one_dice_vis'>
        <geometry>
          <box>
            <size>0.23 0.23 0.23</size>
          </box>
        </geometry>
        <material>
          <script>
         <uri>model://2018/dice/materials/scripts</uri>
         <uri>model://2018/dice/materials/textures</uri>
         <name>OneFace</name>
          </script>
        </material>
      </visual>

      <visual name='one_dice_cable'>
        <pose> 0 0 -0.375 0 0 0</pose>
        <geometry>
          <cylinder>
            <radius> 0.006 </radius>
            <length> 0.75 </length>
          </cylinder>
        </geometry>
      <material>
       <script>
<uri> file://media/materials/scripts/gazebo.material</uri>
       <name> Gazebo/Grey </name>
       </script>
       </material>
      </visual>

```

```
48      <collision name='one_dice_collide'>
49        <geometry>
50          <box>
51            <size>0.23 0.23 0.23</size>
52          </box>
53        </geometry>
54      </collision>
55    </link>
56
57    <link name='two_dice'>
58      <pose>-0.75 0.35 1.25 0 0 0</pose>
59      <visual name='two_dice_vis'>
60        <geometry>
61          <box>
62            <size>0.23 0.23 0.23</size>
63          </box>
64        </geometry>
65        <material>
66          <script>
67            <uri>model://2018/dice/materials/scripts</uri>
68            <uri>model://2018/dice/materials/textures</uri>
69            <name>TwoFace</name>
70          </script>
71        </material>
72      </visual>
73
74      <visual name='two_dice_cable'>
75        <pose> 0 0 -0.625 0 0 0</pose>
76        <geometry>
77          <cylinder>
78            <radius> 0.006 </radius>
79            <length> 1.25 </length>
80          </cylinder>
81        </geometry>
82        <material>
83         <script>
84  <uri> file://media/materials/scripts/gazebo.material</uri>
85  <name> Gazebo/Grey </name>
86      </script>
87      </material>
88      </visual>
89
90      <collision name='two_dice_collide'>
91        <geometry>
92          <box>
```

```
            <size>0.23 0.23 0.23</size>
          </box>
        </geometry>
      </collision>
    </link>

    <link name='three_dice'>
      <pose>-0.05 0.50 1.15 0 0 0</pose>
      <visual name='three_dice_vis'>
        <geometry>
          <box>
            <size>0.23 0.23 0.23</size>
          </box>
        </geometry>
        <material>
          <script>
            <uri>model://2018/dice/materials/scripts</uri>
            <uri>model://2018/dice/materials/textures</uri>
            <name>ThreeFace</name>
          </script>
        </material>
      </visual>

      <visual name='three_dice_cable'>
        <pose> 0 0 -0.575 0 0 0</pose>
        <geometry>
         <cylinder>
           <radius> 0.006 </radius>
           <length> 1.15 </length>
         </cylinder>
        </geometry>
        <material>
        <script>
<uri> file://media/materials/scripts/gazebo.material</uri>
<name> Gazebo/Grey </name>
   </script>
    </material>
      </visual>

      <collision name='three_dice_collide'>
        <geometry>
          <box>
            <size>0.23 0.23 0.23</size>
          </box>
        </geometry>
```

```
      </collision>
    </link>

    <link name='four_dice'>
      <pose>0.5 -0.5 0.8 0 0 0</pose>
      <visual name='four_dice_vis'>
        <geometry>
          <box>
            <size>0.23 0.23 0.23</size>
          </box>
        </geometry>
        <material>
          <script>
            <uri>model://2018/dice/materials/scripts</uri>
            <uri>model://2018/dice/materials/textures</uri>
            <name>FourFace</name>
          </script>
        </material>
      </visual>

      <visual name='four_dice_cable'>
        <pose> 0 0 -0.4 0 0 0</pose>
        <geometry>
          <cylinder>
           <radius> 0.006 </radius>
           <length> 0.8 </length>
         </cylinder>
       </geometry>
      <material>
       <script>
<uri> file://media/materials/scripts/gazebo.material</uri>
       <name> Gazebo/Grey </name>
       </script>
       </material>
      </visual>

      <collision name='four_dice_collide'>
        <geometry>
          <box>
            <size>0.23 0.23 0.23</size>
          </box>
        </geometry>
      </collision>
    </link>
```

```
183     <link name='five_dice'>
184       <pose>0.15 0.10 0.5 0 0 0</pose>
185       <visual name='five_dice_vis'>
186         <geometry>
187           <box>
188             <size>0.23 0.23 0.23</size>
189           </box>
190         </geometry>
191         <material>
192           <script>
193             <uri>model://2018/dice/materials/scripts</uri>
194             <uri>model://2018/dice/materials/textures</uri>
195             <name>FiveFace</name>
196           </script>
197         </material>
198       </visual>
199
200       <visual name='five_dice_cable'>
201         <pose> 0 0 -0.25 0 0 0</pose>
202         <geometry>
203           <cylinder>
204             <radius> 0.006 </radius>
205             <length> 0.5 </length>
206           </cylinder>
207         </geometry>
208         <material>
209         <script>
210   <uri> file://media/materials/scripts/gazebo.material</uri>
211         <name> Gazebo/Grey </name>
212         </script>
213         </material>
214       </visual>
215
216       <collision name='five_dice_collide'>
217         <geometry>
218           <box>
219             <size>0.23 0.23 0.23</size>
220           </box>
221         </geometry>
222       </collision>
223     </link>
224
225     <link name='six_dice'>
226       <pose>-0.5 -0.4 0.85 0 0 0</pose>
227       <visual name='six_dice_vis'>
```

```
228         <geometry>
229           <box>
230             <size>0.23 0.23 0.23</size>
231           </box>
232         </geometry>
233         <material>
234           <script>
235             <uri>model://2018/dice/materials/scripts</uri>
236             <uri>model://2018/dice/materials/textures</uri>
237             <name>SixFace</name>
238           </script>
239         </material>
240       </visual>
241
242       <visual name='six_dice_cable'>
243         <pose> 0 0 -0.425 0 0 0</pose>
244         <geometry>
245          <cylinder>
246          <radius> 0.006 </radius>
247          <length> 0.85 </length>
248          </cylinder>
249         </geometry>
250         <material>
251         <script>
252 <uri> file://media/materials/scripts/gazebo.material</uri>
253         <name> Gazebo/Grey </name>
254         </script>
255         </material>
256       </visual>
257
258       <collision name='six_dice_collide'>
259         <geometry>
260           <box>
261             <size>0.23 0.23 0.23</size>
262           </box>
263         </geometry>
264       </collision>
265     </link>
266
267   </model>
268
269 </sdf>
```

D.11 Gazebo - gate.sdf

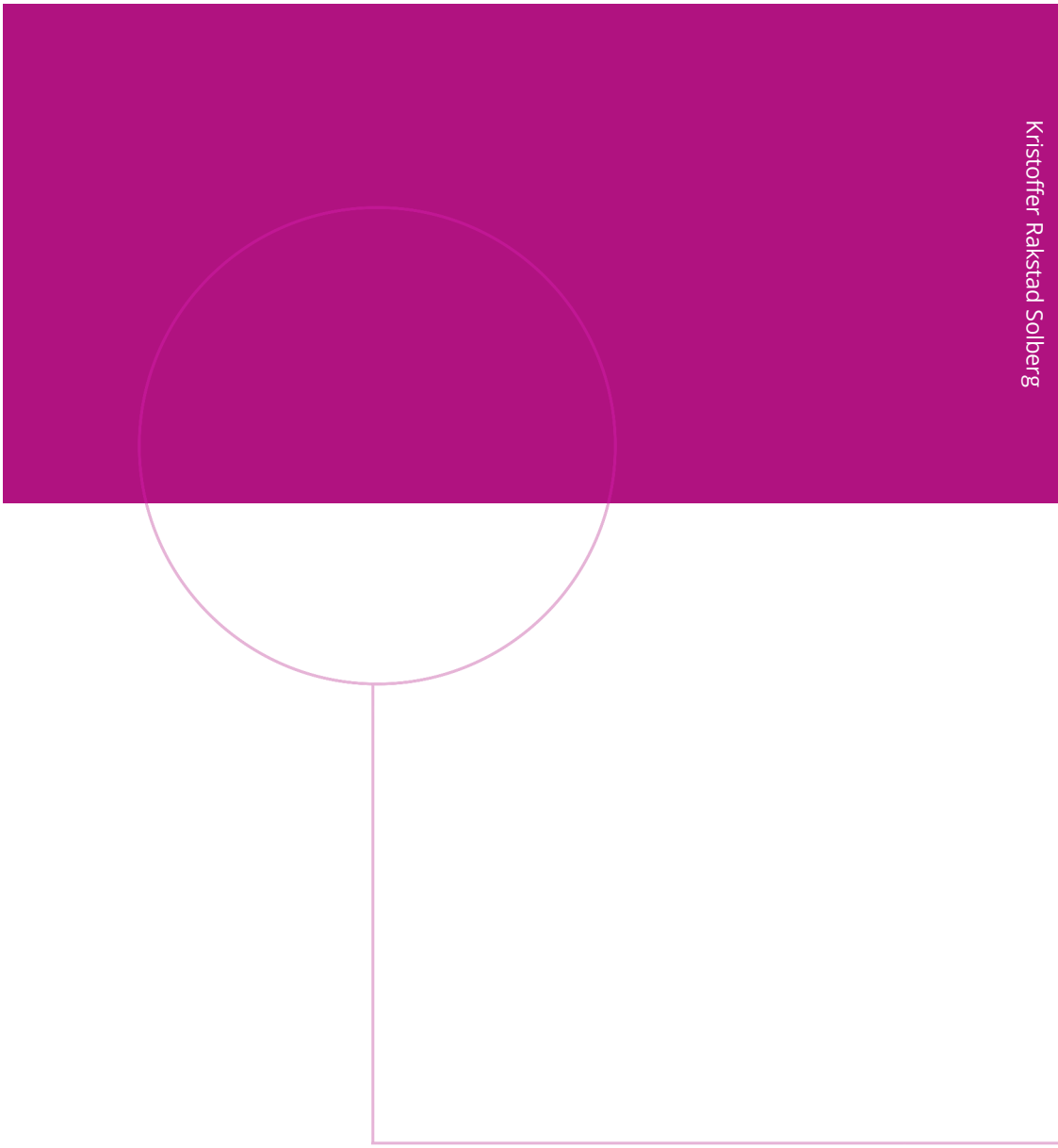
Listing D.7: The Configuration File for the RoboSub Underwater Gate

```
<?xml version='1.0'?>
<sdf version='1.5'>
<!--
     Written by Kristoffer Rakstad Solberg, Student
     Copyright (c) 2019 Manta AUV, Vortex NTNU.
     All rights reserved.
-->

  <model name='start_gate'>
    <pose> 0 0 0 0 0 0</pose>
    <self_collide> 0 </self_collide>
    <static>1</static>

    <link name='gate_top'>
      <pose> 0 0 0 0 1.5707 0</pose>
      <visual name='gate_top_vis'>
        <geometry>
          <cylinder>
            <radius>0.0381</radius>
            <length>3.0</length>
          </cylinder>
        </geometry>
        <material>
          <script>
<uri>file://media/materials/scripts/gazebo.material</uri>
            <name>Gazebo/Yellow</name>
          </script>
        </material>
      </visual>
      <collision name='gate_top_collide'>
        <geometry>
          <cylinder>
            <radius>0.0381</radius>
            <length>3.0</length>
          </cylinder>
        </geometry>
      </collision>
    </link>

    <link name='gate_left'>
      <pose>-1.5 0 -0.75 0 0 0</pose>
```

```
      <visual name='gate_left_top'>
        <geometry>
          <cylinder>
            <radius>0.0381</radius>
            <length>1.5</length>
          </cylinder>
        </geometry>
        <material>
          <script>
<uri>file://media/materials/scripts/gazebo.material</uri>
            <name>Gazebo/Yellow</name>
          </script>
        </material>
      </visual>
      <collision name='gate_left_collide'>
        <geometry>
          <cylinder>
            <radius>0.0381</radius>
            <length>1.5</length>
          </cylinder>
        </geometry>
      </collision>
    </link>

    <link name='gate_right'>
      <pose>1.5 0 -0.75 0 0 0</pose>
      <visual name='gate_right_top'>
        <geometry>
          <cylinder>
            <radius>0.0381</radius>
            <length>1.5</length>
          </cylinder>
        </geometry>
        <material>
          <script>
<uri>file://media/materials/scripts/gazebo.material</uri>
            <name>Gazebo/Yellow</name>
          </script>
        </material>
      </visual>
      <collision name='gate_right_collide'>
        <geometry>
          <cylinder>
            <radius>0.0381</radius>
            <length>1.5</length>
```

```
87            </cylinder>
88          </geometry>
89        </collision>
90      </link>
91
92      <link name='red_flag_left'>
93        <pose>1.4 0 -0.2 0 0 0</pose>
94        <visual name='red_flag_left_vis'>
95          <geometry>
96            <cylinder>
97              <radius>0.0381</radius>
98              <length>0.4</length>
99            </cylinder>
100         </geometry>
101         <material>
102           <script>
103 <uri>file://media/materials/scripts/gazebo.material</uri>
104             <name>Gazebo/Red</name>
105           </script>
106         </material>
107       </visual>
108     </link>
109
110     <link name='red_flag_right'>
111       <pose>0.05 0 -0.2 0 0 0</pose>
112       <visual name='red_flag_right_vis'>
113         <geometry>
114           <cylinder>
115             <radius>0.0381</radius>
116             <length>0.4</length>
117           </cylinder>
118         </geometry>
119         <material>
120           <script>
121 <uri>file://media/materials/scripts/gazebo.material</uri>
122             <name>Gazebo/Red</name>
123           </script>
124         </material>
125       </visual>
126     </link>
127
128     <link name='black_flag_right'>
129       <pose>-1.4 0 -0.2 0 0 0</pose>
130       <visual name='black_flag_right_vis'>
131         <geometry>
```

```
          <cylinder>
            <radius>0.0381</radius>
            <length>0.4</length>
          </cylinder>
        </geometry>
        <material>
          <script>
<uri>file://media/materials/scripts/gazebo.material</uri>
            <name>Gazebo/Black</name>
          </script>
        </material>
      </visual>
    </link>

    <link name='black_flag_left'>
      <pose>-0.05 0 -0.2 0 0 0</pose>
      <visual name='black_flag_left_vis'>
        <geometry>
          <cylinder>
            <radius>0.0381</radius>
            <length>0.4</length>
          </cylinder>
        </geometry>
        <material>
          <script>
<uri>file://media/materials/scripts/gazebo.material</uri>
            <name>Gazebo/Black</name>
          </script>
        </material>
      </visual>
    </link>

  </model>
</sdf>
```



Kristoffer Rakstad Solberg

NTNU
Norwegian University of
Science and Technology