THESIS
DESIGN OF AN AUTONOMOUS DREDGE BOT CONTROLLER

The subject of this thesis (B. Eng) is the design of a controller capable of autonomous operating maritime operations; in particular those of an Archimedes-driven dredging crawler. The proposed controller is a generic and extendable framework which runs on a (cluster) Single-Board Computer. Promising short lead times and cost reduction for prototypes.

THESIS
AUTONOMOUS CRAWLER DESIGN
IHC MTI B.V.

Jelle Spijker

IHC MEDUSA B.V.

June 14, 2020

PUPPET MASTER *It can also be argued that DNA is nothing more than a program designed to preserve itself. Life has become more complex in the overwhelming sea of information. And life, when organized into species, relies upon genes to be its memory system. So, man is an individual only because of his intangible memory... and memory cannot be defined, but it defines mankind. The advent of computers, and the subsequent accumulation of incalculable data has given rise to a new system of memory and thought parallel to your own. Humanity has underestimated the consequences of computerization.*

Masamune Shirow - *Kôkaku Kidôtai*

Client	Royal IHC
External reference	HAN-666
Internal reference	JS01
Version	2
Status	final
Classification	none

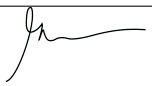
IHC MTI B.V.
P.O. Box 2, 2600 MB Delft
Delftechpark 13, 2628 XJ Delft
S: info@ihcmti.com
T: +31 88 015 2535
M: info@ihcmti.com

THE TECHNOLOGY INNOVATOR.

ROYALIHC.COM

QUALITY CONTROL

This report has been reviewed and approved in accordance with the policies of IHC MTI B.V.

	Name	Date	Signature
WRITTEN BY	Jelle Spijker	June 14, 2020	
REVIEWD BY	ir. Frits Hofstra		
REVIEWD BY	R. Kaandorp MSc. BASc.		
APPROVED BY	J. B. van Elburg MSc. BEng.		

DISCLAIMER

This document has been prepared by Jelle Spijker for Royal IHC. The opinions and information in this report are entirely those of Jelle Spijker, based on data and assumptions as reported throughout the text and upon information and data obtained from sources which Jelle Spijker believes to be reliable.
While Jelle Spijker has taken all reasonable care to ensure that the facts and opinions expressed in this document are accurate, it makes no guarantee to any person, organization or company, representation or warranty, express or implied as to fairness, accuracy and liability for any loss howsoever, arising directly or indirectly from its use or contents.

This document is dated June 14, 2020

CHAPTER SUMMARY

This thesis centers around desiging, implementating, and validating a controller used in autonomous maritime operations. In particular operations of an Archimedes driven dredging crawler. It includes an extensive review of the current state of technology with regards to underwater communications in Section 3.1. Describing the dampening effects of water on the wireless communication, and explaining why Global Positioning System (GPS) isn't suited for the work environment of a sub-merged crawler. Dead-reckoning is often used to overcome such limitations, but these bring their own challenges.

Dead-reckoning relies on state sensing sensors, which are described in Section 3.2. Commonly used Inertial Measurement Units (IMUs) are comprised of accelerometer, gyroscope, and magnetometer. It is not uncommon in underwater applications, to use a pressure sensor, to further limit the uncertainty of the depth estimate. Section 3.2.1 explains why this accuracy of might degrade during dredging operations, because these disturb the soil; creating a localized suspension with a different density, which results in an over estimation of the crawlers depth.

It is imperative for underwater vessels to have a correct estimation of its position and pose. Since it is a GPS deprived environment a literature review has been performed how to minimize this uncertainty. A common method is the use of a Linear Quadratic Estimation (LQE) also known as a Kalman filter. A tried and practice method. It is used to control a real-world dynamic system and uses a fairly faithful replication of the true state of dynamics. It works by estimating a state of the process, based on *a priori* state $\vec{x}_k$, which is transformed to *a post priori* state $\vec{x}_{k+1}$. It is able to filter out white noise with a normal probability distribution. A simple linear example of a falling ball is described in Section 3.3.2.

A literature review regarding different Coverage Path Planning (CPP) strategies, is written in Section 3.4. It describes three different categories: morse-based cellular decomposition, Landmark-based topological coverage and grid-based methods. These methods all use a "divide and conquer" strategy. Dividing an unknown, dynamically chancing environment in to sub-regions, based on certain distinct features. Section 4.1 compares the researched strategies and weighs them against each other. Settling on topological Boustrophedeon Cellular Decomposition (topBCD), a topological coverage algorithm that borrows a Boustrophedon Cellular Decomposition (BCD) aspect from the morse-base strategies. It divides an area into sub-regions based on topological landmark features, and it processes these sub-regions as an ox would plough the field.

Section 4.2 describes which pheripherals will be used, Section 4.3 explains the design of the used Kalman filter, which is a Unscented Kalman Filter (UKF), due to the non-linear behaviour of the physics governing the kinematic and dynamic behaviour. A state representation for the crawler is described in Section 4.3.1. The state vector $\vec{x}_k$ describes position, velocity and pose and its derivative, in 3-dimensions. Mathematical rotation are usually performed with a rotation matrix, this construct suffers from gimbal-lock and is computational "heavy". It is quite common to use quaternions for this kind of rotations. These are an extension to complex numbers. Instead of using one imaginary axis, they use three imaginary axes. The pose vector and its derivative are therefor represented as quaternions.

Section 4.3.2 and Section 4.3.3 describe how a control vector $\vec{u}_k$ will translate to *a post priori* state. It takes into account the characteristics of the drive-train, which is a hydraulic system, actuated with an electrical motor. This drive-train rotates two mirrored Archimedes screw, and transform this rotation in a trust forward. A novel proposal is made, estimating the slip between screw and soil in the terramechanic model. The vanes of an Archimedes screw will act as a bulldozer, if the translation forward is not proportional to the pitch of the vanes. A kinematic steering model of the crawler is based on a differential drive, and is described in Section4.3.5.

Travel of the dredger has two States: normal travel and dredging. In normal travel the maximum velocity is limited by the drive-train. During dredging the limiting factor is its ability of soil removal. Section 4.3.4 determines the maximum allowable speed, governed by the flow of slurry in the dredging system. Resulting in maximum speed of $155.0\,\mathrm{m/h}$ at an optimal production rate of $140.0\,\mathrm{m}^3/\mathrm{h}$.

A C++ controller framework named "ohCaptain" is specifically written for this project. The description of this framework is provide in Section 4.4. It's primarily intended for the control of autonomous maritime vessels. It is based on an actor-model design pattern, where each actor is responsible for its own tasks, which is executed asynchronous; either on the same controller or on a cluster of controllers.

The actor responsible for the big-picture, is called the Captain, he communicates with his Navigators, responsible for state estimation and path planning, and his First mates. These mates orchestrate and controller the Boatswains. Actors who perform low-level dedicated tasks. Such as reading out a sensor signal, or controlling the speed of an Archimedes screw. Because this framework is written in C++ it can be used on a multitude of different controllers, ranging from dedicated specialized hardware, to Single Board Computer (SBC) such as a Raspberry pi (Rpi) or Beagle Bone Black (BBB). This design decision has potential to greatly reduce the cost and shorten the lead time of future prototypes.

Validation of the controller is performed in a simulated environment. Chapter 5 describes this in detail. It uses Project Chrono, a multi-body physics engine with specialized modules for vehicle simulation and terramechanics. However, tt lacks a module for sensor simulation. Something that is necessary in validating the performance of the UKF. Section 5.1.1 describes a custom written extension to Project Chrono, called “chrono_sensors”. It allows for realistic sensor simulation, providing sensor signals to a controller, and is subject to noise, discretization, delay, and transformation. This extension was written for this project, but can also be integrated with other Project Chrono simulations, it is provided as an open-source project to the community under the MIT license.

A virtual scenario is created and described in Section 5.1.2. This scenario is based on a use-case in which the crawler needs ton perform a maintenance dredging task in a small recreational harbor near Bruinisse in the Netherlands. The controller “ohCaptain” and “chrono_sensors” are integrated in a simulation and the same scenario is run three times. Section 5.2 evaluates the different scenarios. The first scenario can be considered as an “analytical” run. This is shows the path generated by the Captain and his Navigator with an accurate location as basis. This scenario shows that the selected topBCD strategy does what its intended to do. The whole harbor is covered in a seemingly logical way.

The second and third run are executed to show the difference between a UKF based controller and a simple Proportional Integral Derivative (PID) based one. The difference is rather pronounced, were the PID has a localization error that keeps on growing, eventually letting the crawler think it is inside the body of water, while in reality it has crossed the boundary and is now dredging on land. In real life unacceptably. The UKF based controller, has however managed to cover the whole soil bed.
The findings of this thesis are discussed in the conclusion, Chapter 6.

CHAPTER CONTENTS

CHAPTER 1 INTRODUCTION

The last decenia has shown a spurt in the acceptance and adaptation of machine learning. Smart controllers capable of performing complex tasks without interference of human operator. These range from "smart" cars to home-owners whom integrate cheap Single Board Computers (SBCs), such as a Raspberry pi (Rpi), into their home. This general acceptance and an abundance of cheap available devices, has given a real boost to the development of machine learning and autonomous operating machines.

1.1 BACKGROUND

Royal IHC (IHC) is investigating on a strategic level how to best place this technology in their product portfolio. One of the key parameters is knowing the Technological Readiness Level (TRL). Multiple business units within IHC are developing prototypes of autonomous operating machines. Ranging from catamarans to deep sea mining crawlers. A lot of these explorations are usually small side projects. With short leads and limited budgets. All give valuable insights and are of import in developing a familiarization with this technology.

IHC MTI B.V. (MTI) is one of these business units exploring the technological challenges that of various autonomous devices. They're the research and development department within IHC. They adopted an Archimedes driven crawler from one of their sibling business units. This crawler was unfinished at the time and MTI saw it as a great opportunity to learn.

This thesis discusses the design of a controller, for autonomous maritime operations; First and foremost for the operations of an Archimedes driven dredging crawler; In addition, it will take into account low budget considerations and rapid prototype requirements. Such that it can serve as a stepping stone for future autonomous operating maritime vessels. The crawler in this document is a relative small dredging crawler, designed for maintenance tasks in basins, harbors and around dams.

1.2 USE CASES

Three use cases have been specified by ir. F. Hofstra, these cases are valid and realistic scenarios whilst keeping in mind their marketability. These cases will determine the needed functionality for a crawler and form the basis for the controller design.

1.2.1 ARBITRARY SHAPED SPACE

An crawler is placed in a predefined arbitrary shaped space, not too complex, with an area of $3500\mathrm{m}^2$. The shape of this space is set but the movement pattern is unrestricted. The crawler has to remove a layer with a depth of $5\mathrm{cm}$. The controller has to determine an optimal path with the least amount of time or the shortest path. This can be coupled with learning capabilities and an analyze capacity. At a later time, additional constrains can be added which keep in mind the deployment location of a flexible dredgeline and an umbilical.

1.2.2 MARINA AQUA DELTA

The crawler operates in a predefined space with obstacles; not every obstacle is known. The actual location is marina Aqua Delta located in Bruinisse, the Netherlands. The shape of this location is set but the movement pattern is unrestricted. An crawler has to remove a layer with a depth of $5\mathrm{cm}$. The controller has to determine an optimal path with the least amount of time or the shortest path. This can be coupled with learning capabilities and an analyze capacity. The marina has enough depth for the crawler to move underneath the scaffolding. No consideration has to be made for a flexible dredgeline and an umbilical. These conditions can be introduced at a later stage.

1.2.3 THREE GORGES DAM

An crawler operates in a predefined space with obstacles, not every location of those obstacles is known. The predefined space is located at the foot of the three Gorges Dam. Silt is deposited at the foot of this dam, due to natural occurring erosion and sedimentation. The accumulation of silt can be controlled by dredging localized pits which, in turn, create locations with a lower density. This induces a gravity driven density current towards those locations. The crawler has to maintain an average nominal depth with a certain silt deposit rate.

1.3 STRUCTURE OF THIS DOCUMENT

In Chapter 2 a short introduction is made regarding dredging principles and their applications. Readers familiar with this subject can skip this this chapter. Chapter 3 is a review of of the current state of technology with regards to autonomouse operations underwater. It investigates communication techniques, various sensing methods needed for localization, and describes how a position can be estimated in a Global Positioning System (GPS) deprived environment. The final research section looks into various Coverage Path Planning (CPP) algorithms, need for dredging operations.

With the gained knowledge from Chapter 4 a design for a controller is proposed. A choice is made between different CPP strategies based on usefulness for the crawler and the needed peripherals laid out. It then describes the models for the state vector $\vec{x}_k$, state transition matrix F and the control vector $\vec{u}_k$, needed for a Unscented Kalman Filter (UKF), which is used to determine the position under uncertainty. A framework “ohCaptain” is proposed, which is a C++ library, intended for controlling maritime autonomous vessel, with a focus on extensibility and rapid development.

The performance of the controller is validated using a multibody physics engine, called Project Chrono. The setup and results are described in Chapter 5. An important part of measuring the performance of the controller, is grasping its ability to process noisy and transformed sensor signals. A feature, which was not present in the physics engine’s. An extension of Project Chrono, was written, which allows for realistic sensor simulations. The works are explained in this chapter as well. Finally the result are discussed.

The final Chapter 6 looks back at the original goal and discusses how to proceed from here.

CHAPTER 2
DREDGING PRINCIPLES AND APPLICATIONS

This chapter describes the dredging task in some detail. Readers familiar with dredging and commonly used terminology can skip this chapter, since no new information will be provided. It first describes basic principles, applications and tools applicable by the used machinery for the use-cases.

2.1 BASIC DREDGING APPLICATION

Training Institute for Dredging [42] defines dredging as the underwater removal of soil and its transport from one place to another for the purpose of deepening or making profitable use of the removed soil. They make a distinction between nine types of operations: dredging for prosperity, dredging in ports and channels, exploitation of agricultural resources, mineral dredging, coastal protection, land reclamation, infrastructural projects, improvement of the environment and trenches for cables and pipelines.

All three described use-cases are of the maintenance type. Schrieck [72] states that, in order to maintain existing waterways and harbours, the depth of the bed must be preserved by regularly removing silt. In canals and ports basins, where currents are low, the sediment is mostly fine-grained silt and sludge. Where currents are stronger, as in access channels in tidal zones, or rivers, the sediment is sand. He further describes that a characteristic of this kind of work is the weak cohesion of the soil to be removed, since it consists of recently deposited sediment and no significant consolidation has taken place yet.

Sanitation dredging is a distinct form of maintenance dredging and is a process that has been specifically designed for contaminated sediment. Just in the way sediment settles in rivers, harbours and deltas so does heavy metal, inorganic and aromatic compounds, especially downstream of industrial areas. When these contaminated sediments become a risk towards public health and environment, they need to be removed with care and precision.

2.2 COMMONLY USED VESSELS AND EQUIPMENT

Common dredge tools used during maintenance work are listed below. Out of this list, backhoes and suction dredgers are mostly used during port maintenance. Vlasblom [83] states that dredgers can be divided into two categories: mechanical dredgers and hydraulic dredgers. The difference lies in the way the soil is excavated; either mechanically or hydraulically.

2.2.1 MECHANICAL DREDGERS

They work by removing soil and sediment from the submerged soil bed by mechanically excavating it and transporting it to a storage location, such as a hopper. The various types of mechanical dredgers won't be described in this section, since the crawler used in our use-cases will be of a hydraulic type.

2.2.2 HYDRAULIC DREDGERS

These types of dredgers work by removing and transporting soil from the seabed. They use a hydraulic system, were the necessary work needed for mass transportation is deliver by a pump. The soil is transported as a slurry which is a mixture that consists of both solid and fluid phases, and this is usually stored in a dedicated place such as a hopper.

PLAIN SUCTION DREDGERS

Vlasblom [83] describes a plain suction dredger as a stationary dredger, consisting of a pontoon anchored by one or more wires and with at least one sand pump that is connected to a suction pipe.

The discharge of the dredged material can take place via a pipeline or via a barge-loading installation. During sand dredging, the dredger is moved slowly forwards by a set of winches.

TRAILING SUCTION HOPPER DREDGERS

The Trailing Suction Hopper Dredger (TSHD) is a seagoing ship equipped with one or two suction tubes, a pump installation and a hopper with multiple bottom doors and one or more overflows. A draghead is attached to each suction tube and is trailed across the sea bed to loosen the soil before it's pumped up [72]. This soil is stored in a hopper which is periodically discharged, at a designated location, through dumping or pumping out.

AUGER SUCTION DREDGERS

According to VBKO Vereniging van waterbouwers in bagger-, kust- en oeverwerken [12] an Auger Suction Dredger (ASD) consists of a double symmetrical Archimedes screw, also called an auger, surrounded with a steel protective cover and a flexible rubber curtain. This auger is lowered, on a rigid arm, and positioned on the soil bed. Here, it cuts the material and actively transports it into the centre where it's sucked away by a dredge pump. Because the complete dredging process takes place behind a flexible rubber curtain and the auger guides all material towards the suction mouth, this type of dredger is well suited for sanitation maintenance.

CUTTER SUCTION DREDGERS

According to Vlasblom [83] a Cutter Suction Dredger (CSD) is a stationary dredger equipped with a cutter device (cutter head) which excavates the soil before it's sucked up by the flow dredge-pump. During this operation, the dredger moves around a spud pole by pulling and slacking on the two fore sideline wires. This type of dredger is accurate and can cut almost all types of sediment.

2.3 HYDRAULIC DREDGING PRINCIPALS

According to Van Den Berg [67] hydraulic systems are the de-facto industry of transportation for dredged sedimented or slurry; hydraulic systems consist of pipes, either flexible or rigid, combined with centrifugal pumps, a suction mouth and a discharge unit. The pump adds energy to a slurry, suchs that a required flowrate can be achieved, this energy is needed to overcome a system specific pressure drop. Which is the result of energy losses due to potential height differences, kinematic behavior of the fluid and friction, both from shearing of a fluid along a wall and internal shearing of the fluid itself.

The section below briefly describes the workings of two main components in this hydraulic system, namely a dredge-pump and a draghead.

NOTE 2.1: OUT-OFF SCOPE

Two of the use-cases mention that additional constraints, such as a flexible dredge line to shore, can be added to the assignment. It was however opted, to not applied these additional constraints, due to a time constraint on the assignment as a whole.

2.3.1 DREDGE PUMP

In order to transport slurry with a particular density and velocity through a pipeline, a pressure, equal to the sum of all the resistances and geodetic head must be generated. A pump supplies this pressure [67]. Assuming a steady flow, the pump basically increases the Bernoulli head of the flow between point 1, the eye, and point 2, the exit [56].

2.3.2 AUGER DREDGE HEAD

An auger dredge head excavate soil by employing a Archimedes screw transportation principle. This method ensures an extremely quiet cutting and mixing process with little spillage and turbidity in the surroundings. The large working width of the auger makes it extremely suited to dredge thin, possibly polluted, layers at a relatively high production rate [72].

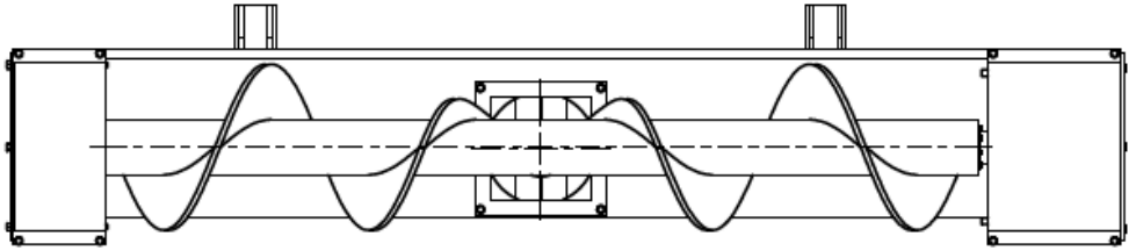


FIGURE 2.1: SCHEMATIC DRAWING OF AN AUGER DREDGE HEAD [91]

The auger is in effect a screw conveyor which guides the material towards the suction head. Green and Perry [35] states that the screw conveyor is one of the oldest and most versatile conveyor types there is. It consists of a helicoid flight mounted on a pipe which turns in a trough. Screw conveyors are well standardized, using International Standard ISO [4] empirical gathered factor values for filling rates and progress resistance.

NOTE 2.2: ASSUMPTION

The assumption is made that the hydraulic system, consisting of flexible pipes and a pump, is the limiting factor in the mass flow, and that the auger simply delivers what is needed.

3 CHAPTER RESEARCH

A crawler performs its tasks in an underwater environment. Its task consists of moving, mapping and dredging a certain basin or area. In order to fulfill tasks of its own accord, it has to be able to sense its surrounding environment and execute its task using a strategy. This ensures performance according to specification.

In the next sections, the key philosophies and processes are investigated; all of these are needed to fulfill its objective. Firstly, in Section 3.1, different ways of underwater communication are reviewed. This is after all the interface between man and machine. A secondly, a review regarding useful sensors, their workings and possible applications is made in Section 3.2.

Once the low-level tools (such as communication devices and sensors) are discussed, a detailed study is made into possible implementation and fusion of these sensors. Such that they can be used to estimate a location of a crawler, needed to operate it in a Global Positioning System (GPS) deprived environment. Section 3.3 describes the use of cooperative localization techniques and Kalman filtering.

Lastly, a survey is made for useful strategy at a higher abstraction level. Section 3.4 describes how a crawler could best perform its main task: covering and dredging a large basin, uniformly. These so called Coverage Path Planning (CPP) algorithms describe and propose different strategies that allow a crawler to perform its task in an unknown and changing environment.

3.1 UNDERWATER COMMUNICATION

This section describes various principles of underwater communication. It identifies two basic methods of transmitting data, namely: wired communication or wireless communication. Wired communication will be in the form of an umbilical which is an electronic cable connecting to an underwater vehicle using regular and industry standard communication protocols. Wireless communication can be performed through four basic principles: electromagnetic, electric current, acoustic or optical signals. Of these principles only electromagnetic and acoustic are explored, since an electrical current doesn't work in a fresh water reservoir and optical signals get sub-optimal performance in a dredging environment due to diffraction and scattering of light by floating sand particles.

The environment presented in the use-cases, described in Section 1.2, state that the crawler will operate in fresh water basins. It's also likely that it will be connected to the water surface with a floating dredgeline. The choice for wired communication is therefore easily made. There may, however, still be a need for wireless communication with external sensors (as the principles presented in Section 3.3 illustrate).

3.1.1 WIRED COMMUNICATION

With wired communication, data signals are transmitted through a wire which acts as a pathway where the information is transmitted as a digital bitstream (a sequential binary sequence). Transmission of information through this wire is limited by a certain bandwidth in Hz were the limiting factors are material properties such as: conductivity, permittivity and permeability. As well as processing of the signals at the end and start node, communication wires are made of a carrier medium, such as copper or glass fibre. This carrier medium facilitates transmission of electromagnetic waves or currents where electromagnetic waves, such as light, are transmitted through fiber optic cables and a modulated pulse of light propagates through a glass tube through the principle of Total Internal Reflection (TIR). Electromagnetic communication makes use of copper wires, where an electric charge propagates through the cable. Copper is the industry de-facto preference due to its high-conductivity, low electrical resistance and resistance to corrosion.

Babani, Bature, Faruk, *et al.* [79] made a comparative study between fibre-optic and copper cables in a context of modern network protocol. They identified the following properties for comparison: bandwidth, cost, dimensional properties (such as weight, size and flexibility), signal loss and safety and immunity. They illustrate that fiber optics cables, although more expensive, are the better choice by

stating that fibre-optic cables are smaller and lighter compared to metal cables, especially copper based. Optical fibre occupies less space in conduits than copper cabling, they weigh less and have a tighter bend radius than any copper cable. Furthermore, signals don't cross-talk with neighboring wires. The low signal attenuation performance and superior signal integrity found in fiber optical systems facilitates much longer runs for signal transmission. The attenuation loss experienced in fiber optic cables can be attributed to microscopic and macroscopic impurities in the fiber material and structure, which cause absorption and scattering of light signal. In figure 3.1 the attenuation loss of $1\mathrm{km}$ of cable is shown as a function of frequency. Both signals propagate with nearly the same speed through their corresponding wire, but when a high data throughput is wanted. It becomes evident from this figure that usage of fiber-optics are paramount.

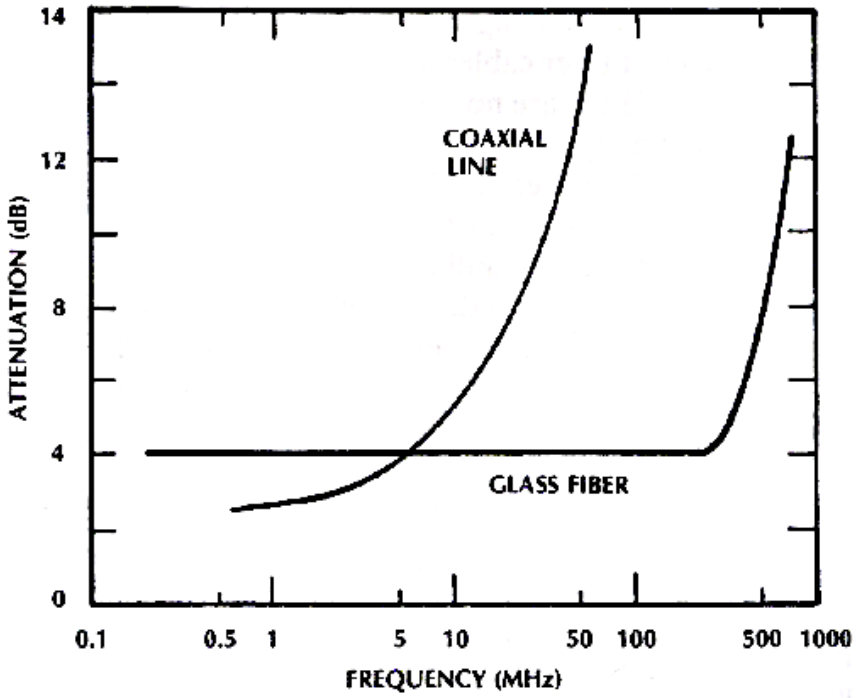


FIGURE 3.1: EFFECTIVE ATTENUATION FIBER VS COPPER CABLE 1 km [10]

Other important factors to consider, for an underwater wired-communication between a base station and a dredge bot, are the effects of the wire on the bot itself. Whitcomb [17] states that most present day vehicles are Remote Operated Vehicles (ROVs) — tele-operated vehicles employing an umbilical cable to carry both power and telemetry from a mother-ship to the vehicle. He further states that a growing number of research vehicles are Autonomous Underwater Vehicles (AUVs) — which operate without an umbilical tether. This statement is supported by Valavanis, Gracanin, Matijasevic, *et al.* [8], who describes that the ROV umbilical cable constrains the vehicle to operations in close proximity to the support ship. Because the crawler is tethered to a location above water level, due to it's floating dredgeline, and because this crawler is from its starting-point constructed as a ROV, it will, in all likelihood, be controlled through an umbilical.

Westneat, Blidberg, and Corell [5] describes that, as the range of operations becomes longer and the water deeper, the drag exerted by the tether becomes significant. The thrusters, and thus the vehicle itself, must become larger and the cable thicker, and the energy that goes into the cable maintenance becomes a major factor. This factor is illustrated by Fang, Hou, and Luo [34], who describes a mathematical model which allow the state representation of the dredge bot, as described in section 4.3.1, to be modified by the forces that are exerted on the cable. In these equations, mass and inertia of the cable play an important role. Because these are just a fraction of the properties for a dredgeline, it's assumed that these forces can be neglected. According to Feng and Allen [25], the effects of the cable can be reduced when it's deployed by a drum on the shore with negligible tension when it's pulled by the vehicle.

PROTOCOLS

The signals which are transported through the wires need to adhere to certain rules and conventions. In other words, the transponder and receiver need to speak the same language and be aware of etiquette in order for a message to be received as intended. The Institute of Electrical and Electronics Engineers (IEEE), have dictated most of the widespread used norms today. The most common used norm in wired communication is *IEEE 802.3* or as it's more commonly known Ethernet. Ethernet consists of a multitude of protocols. In this IEEE norms are the physical layer, data link layers and the Media Access Control (MAC) for each protocol defined.

Shortly put, MAC is defined as the lower sub layer of the data link layer and provides addressing and channel access control mechanisms that allow for communication between several terminals, or nodes, within a multiple access network. This layer acts as an interface between the Logical Link Control (LLC) sub layer and the network's physical layer; the LLC makes it possible to let several network protocols coexist. According to Jolectra [80], the current dredge bot makes use of an *Allen Bradley ETHERNET/IP adapter* of type *1769-AENTR*, which allows the use of the Common Industrial Protocol (CIP), Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). CIP is used by EtherNet/IP, and is a familiar and widely used protocol for controllers.

3.1.2 WIRELESS COMMUNICATION

Freitas [69] tells us that wireless communications have been subject to enormous research and improvements in the near past. This effort is responsible for allowing multiple devices to securely communicate simultaneously with high availability, great distances and high data rates. While these improvements are applied and tested mainly in over-the-air communications, underwater communications suffer from a low applicability of radio frequency transmission systems due to a low attenuation of Electromagnetic Waves (EMW) in water.

He [69] further states that when using radio frequency, underwater communications do not fully benefit from the improvements achieved in air due to the fact that electromagnetic propagation in water causes a vast reduction in the range of communcation. Because of the limitations that water imposes, these communications are currently performed using acoustic waves and in some cases optical systems. This is further supported by Lloret, Sendra, Ardid, *et al.* [59] who remarks that underwater communication research is primarily focused on the use of optical signals, electromagnetic signals and the propagation of acoustic and ultrasonic signals. Each technique has its own characteristics, with its benefits and drawbacks, mainly due to the chemical characteristics [49] and physical constraints of the medium [40].

ELECTROMAGNETIC COMMUNICATION

A common method to transfer data via a wireless connection is to make use of EMW, this is a type of electromagnetic radiation with wavelengths in the electromagnetic spectrum (as is shown in figure 3.2). Waves in this spectrum can have frequencies between $3\mathrm{kHz}$ or $3\mathrm{GHz}$. These waves travel the speed of light and are transverse waves, because the amplitude is perpendicular to the direction of the wave travel. However, EMW are always waves of fields, not of matter, because they are fields, EMW can propagate in empty space [77].

Data is transferred between devices by either modulating the frequency or altering the amplitude of a signal to carry the data. Where a carrier frequency is modulated by superimposing a data signal. This is illustrated in figure 3.3.

Hagman, Elias [44] tells us that the reasons why EMW are used to transfer information in the classic wireless air channel lies in their fast propagation speed, their wide usable frequency spectrum and the small amount of environment noise created compared, for example with acoustics factors. This all leads to the possibility of high data rates. Furthermore, the EMW has the ability to propagate without a carrier medium and the electric-magnetic field conversion enables in general very large communication ranges.

But in water — especially in seawater — things are much different. This statement is supported by Ramakrishna and Nissen [60] when they convey that the ocean is almost impervious to EMW, which makes them useless for wireless underwater communication over distances greater than a hundred meters. Hagman, Elias [44] illustrated this by solving Maxwell's equation to predict the propagation

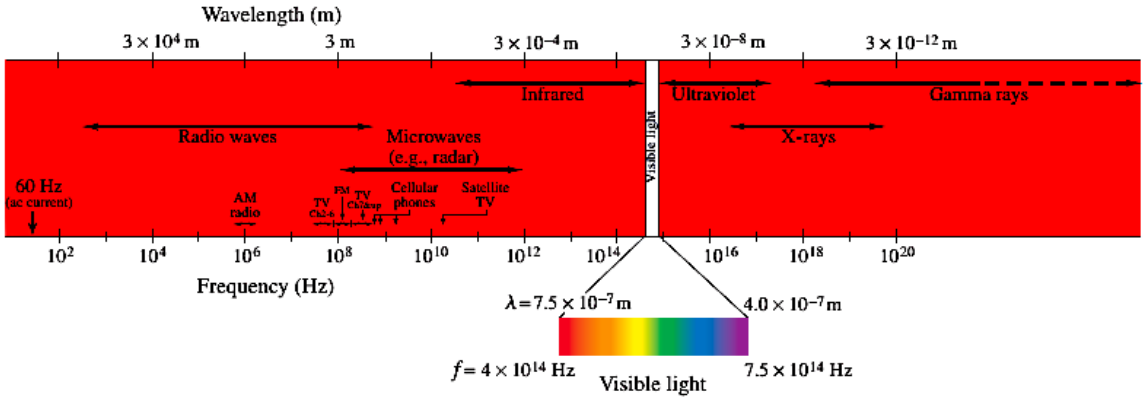


FIGURE 3.2: ELECTROMAGNETIC SPECTRUM [77]

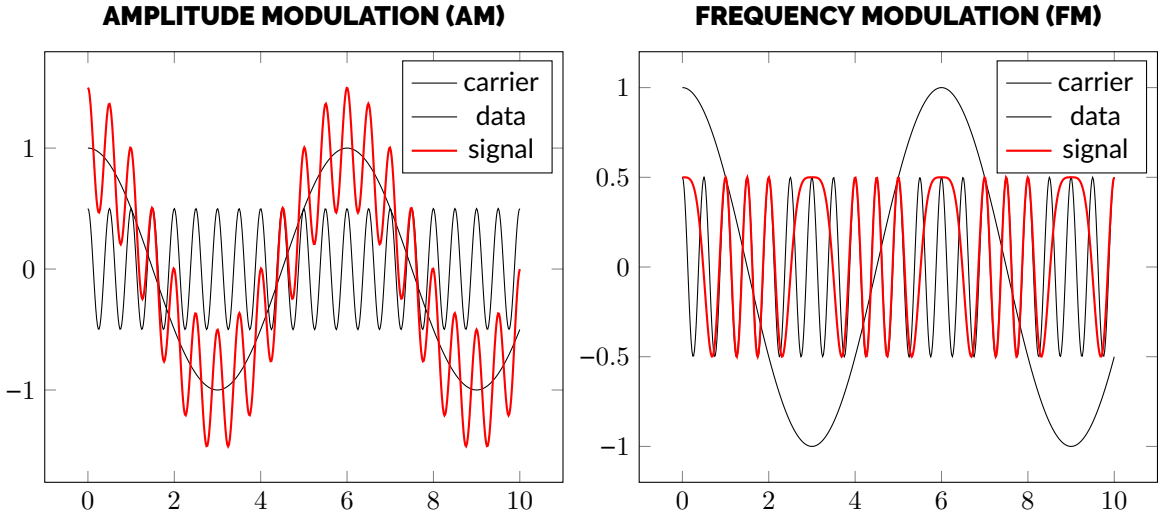


FIGURE 3.3: SIGNAL MODULATION

of EMW for the case of a linearly polarized plane travelling in z-direction, we get the electric field strength E_x and the magnetic field strength H_y [44].

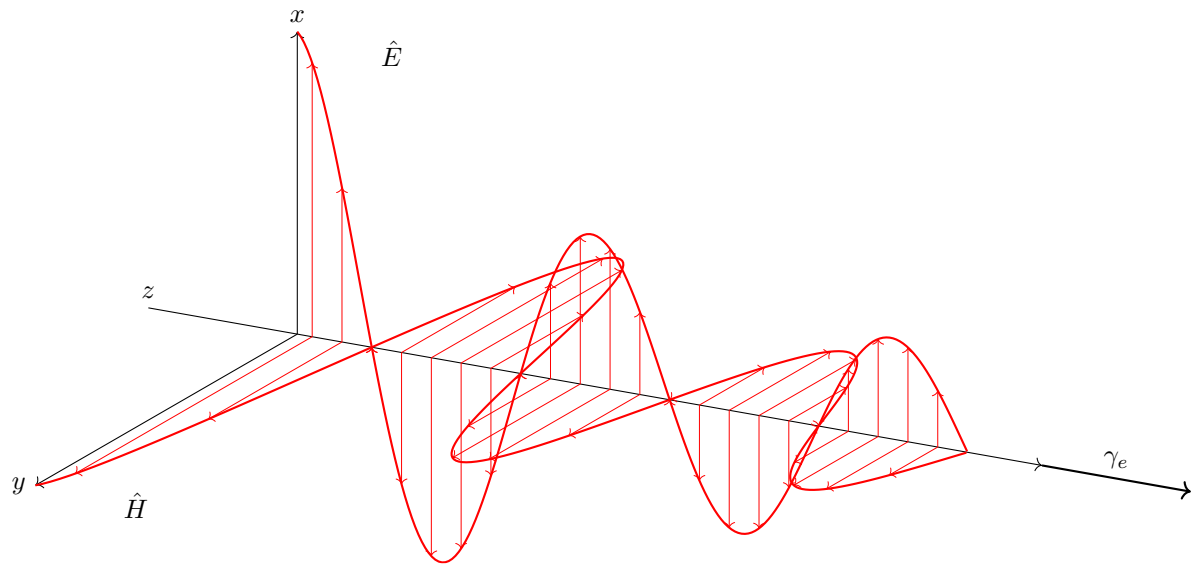


FIGURE 3.4: DAMPENING OF ELECTRIC AND MAGNETIC FIELD

$\hat{E}$ and $\hat{H}$ are the amplitudes of the electric and the magnetic field wave and γ_e is the propgation constant, expressed in ϵ_e, which is the permittivity, as shown in equation 3.3, μ_e is the electromagnetism permeability and σ_e is the electrical conductivity of a material. Here α_e is the attenuation and β the

phase factor of a wave.

$$E_x = \hat{E}\exp^{i\omega t-\gamma_e z} \tag{3.1}$$

$$H_y = \hat{H}\exp^{i\omega t-\gamma_e z} \tag{3.2}$$

$$\gamma_e = i\omega\sqrt{\epsilon_e\mu_e - \frac{i\sigma_e\mu_e}{\omega}} = \alpha + i\beta \tag{3.3}$$

$$\alpha_e \approx 0.0173\sqrt{f\sigma_e} \tag{3.4}$$

As is evident from equation 3.1 and 3.2, there is a logarithmic relationship; maximization of the propagation γ_e leads to a lower amplitude of the electric and magnetic fields. This propagation is mostly determined by the attenuation α_e, which varies at different frequencies and mediums. Claus [68] tells us that this attenuation factor is given as equation 3.4, which shows us that the attenuation is related to the square root of the frequency f multiplied by the conductivity of the water σ_e. Hattab, El-Tarhuni, Al-Ali, *et al.* [65] states that the loss of a signal travelling through water can be calculated using equation 3.5. They state that the knowing the real-part of γ_e is sufficient to calculate the loss for a given frequency. Since the only changing term, due to frequency in the complex-valued γ_e, is in its imaginary part, and due to the fact that each γ_e is multiplied with i, both outside of the root as inside, this value will be a constant throughout the frequency spectrum. This attenuation model will not be used for our calculations where $\Delta d_{1,2}$ is the separation distance between transmitting and receiving nodes and only the real part of the propagation constant σ_e is used.

$$L_{\alpha,\epsilon} = \mathrm{Re}(\gamma_e) = \frac{20}{\ln(10)}\Delta d_{1,2} \Rightarrow \Delta d_{1,2}\frac{L_{\alpha,\epsilon}}{\mathrm{Re}(\gamma_e)\frac{20}{\ln(10)}} = \frac{L_{\alpha,\epsilon}}{\alpha_e} \tag{3.5}$$

The maximum penetration depth of signal in (sea) water, will, for simplicity's sake be calculated with equation 3.5, where α_e is obtained using equation 3.4. Jiang and Georgakopoulos [50] tells us that seawater has a typically high conductivity of $4.0\,\mathrm{S/m}$, whilst freshwater has a typically conductivity of only $0.0\,\mathrm{S/m}$, 400.0 times less. He [50] further states that communication using electromagnetic waves in fresh water can be more efficient in fresh water. This statements are confirmed by Jiang and Georgakopoulos [50], Ainslie [47] and Bogie [3]. Figure 3.6 and 3.5, which shows the EMW propagation in fresh and seawater for commonly used frequencies, illustrate this phenomenon.

PROTOCOLS

Subsection 3.1.1 describes the need for a transceiver and receiver to speak the same language and adhere to the same etiquette. This holds true for wireless protocols as well. Most wireless protocols are described in the IEEE 802 standards. These are a family of standard network protocols describing networks carrying variable-size packets. These protocols are the de-facto industry standards. A short description for the most popular 802 standards are given below. These protocols map two layers, namely: a data link layer and physical layer. The data link layer is split into two sublayers: LLC and MAC. LLC provides the multiplexing mechanisms that enable the network protocols and provide flow control and automatic repeat requests whilst MAC provides addressing and channel access control

mechanisms that makes it possible for several nodes to communicate within a multiple access network.

IEEE 802.11 WLAN

The IEEE 802.11 standard is also known as WiFi. It encompasses wireless modulation techniques, designated as 802.11 (a, b, g, n and ac). The 802.11 standard makes use of the $2.4\,\mathrm{GHz}$ and $5.0\,\mathrm{GHz}$ bandwidth. Freitas [69] states that Wi-Fi frequencies maybe a challenge when used in underwater communications, because its attenuation drastically reduces the channel distance, as is shown in figure 3.5. A new standard 802.11af is being developed. This standard will make use of the $700.0\,\mathrm{MHz}$ $700[MHz]$ frequency which might give an extra couple of meters underwater.

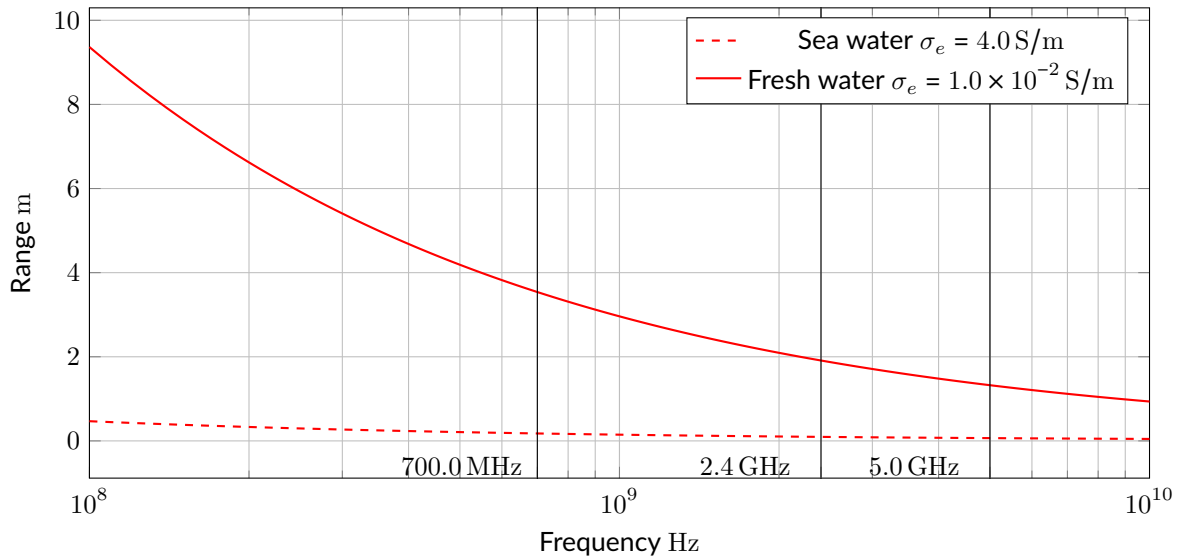


FIGURE 3.5: PROPAGATION RANGE OF WI-FI IN WATER.

IEEE 802.15.4 LO-FI

From all different protocols described in the IEEE 802.15 special consideration is made into the IEEE 802.15.14 or LoRa which is an upcoming communication protocol for Internet of Things (IoT) devices. It operates in $433.0\,\mathrm{MHz}$ and $(863.0$ to $870.0)\,\mathrm{MHz}$. The protocols are opensource and the modules are very cheap. This protocol is developed for robust, long range communication which can reach $22.0\,\mathrm{km}$ on land. Akyildiz, Pompili, and Melodia [28] tell us that the electromagnetic waves at $433.0\,\mathrm{MHz}$ have been reported to have a transmission range of $120.0\,\mathrm{cm}$ in an underwater environment. These experiments have been performed at the Robotic Embedded Systems Laboratory (RESL) at the University of Southern California.

Because of the use of lower frequencies, LoRa shows a three-fold increase in range compared with normal WiFi. The propagation of LoRa signal in (sea)water is shown in figure 3.6. When this is compared with figure 3.5, an increase in range is found.

ELECTRIC CURRENT

Another way to communicate is through the use of electric current. Hagman, Elias [44] describes that seawater, as a conductive medium, can be subject to a modulated signal generated by a pair of transmitting electrodes that launch a current field in the channel. If this current field is strong enough, the receiver — that also uses a pair of electrodes — could measure a potential difference and therefore receive the signal. Since electric current noise is extremely low in seawater, small current field amplitudes are sufficient to receive information and a large data rate is achievable [44]. Since this type of transmission only works in a conductive medium, and the use case only specifies that a dredge bot will be deployed in fresh water basins, electric current communication is not deemed a viable candidate.

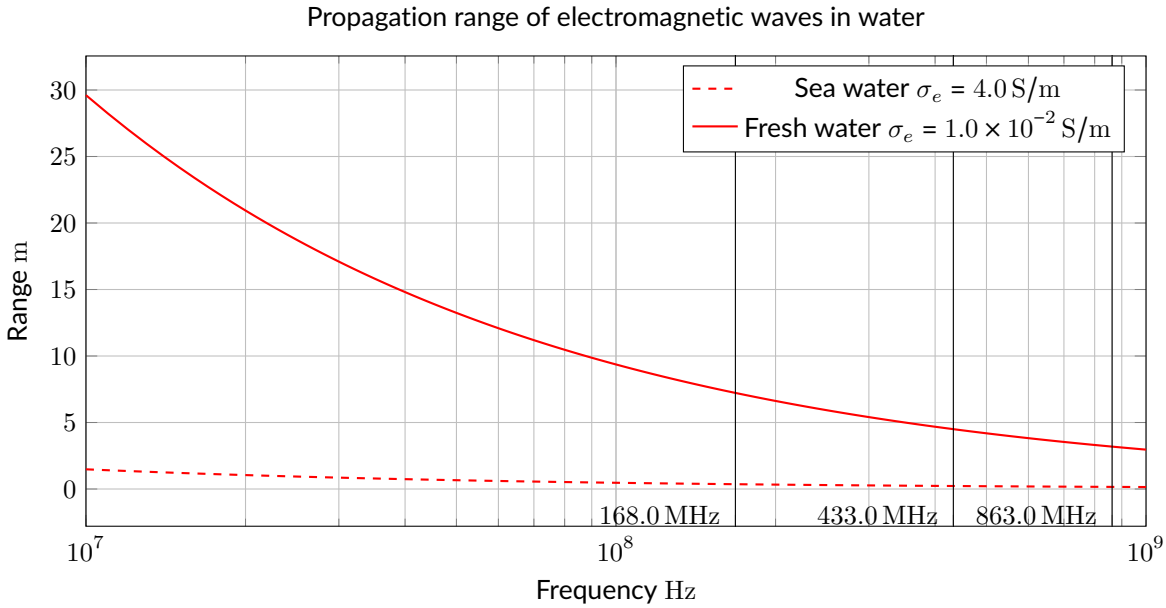


FIGURE 3.6: PROPAGATION RANGE OF LO-FI IN WATER

ACOUSTIC COMMUNICATION

As is shown in Section 3.1.2, EMW have a very limited range in (sea) water, due to a high attenuation. Multiple sources such as Hagman, Elias [44], Claus [68] and Domingo [57] state that acoustic communication is therefore the preferred way. This type of communication makes use of Sound Waves (SW), or Acoustic Waves (AW), which are often described as the vibration of molecules of the medium in which it travels — that is, in terms of the motion or displacement of the molecules. SW can also be analysed from the point of view of pressure. Indeed, longitudinal waves are often called pressure waves. The pressure variation is usually easier to measure than the displacement [77]. This principle is used by hydrophones; these are, in effect, microphones designed to be used underwater using piezoelectric transducers to convert pressure waves into electricity. Although acoustic communication is the preferred method, there are a lot of challenges to overcome. According to Tetley and Calcutt [37] transmitting and receiving acoustic energy in seawater is affected by the often unpredictable ocean environment. Lanbo, Shengli, and Jun-Hong [40] and Edward Tucholski [30] both state that the speed of sound in the sea is not constant, but a function of temperature, pressure and salinity $v(T, P, S)$. Because the speed is not constant sound does not travel in a straight line. Acoustic communication can can be summarized as follows:

PARAMETER	VALUE
Attenuation	A variable factor related to the transmitted power, the frequency of transmission, salinity of the seawater and the reflective consistency of the ocean floor.
Salinity of seawater	A variable factor affecting both the velocity of the AW and its attenuation.
Velocity of sound in salt water	This is another variable parameter. Acoustic wave velocity is precisely $1505.0\,\mathrm{m/s}$ at $15.0\,^{\circ}\mathrm{C}$ and atmospheric pressure, but most echo-sounding equipment is calibrated at $1500.0\,\mathrm{m/s}$
Reflective surface of the seabed	The amplitude of the reflected energy varies with the consistency of the ocean floor.
Noise	Either inherent noise or that produced by one's own transmission causes the signal-to-noise ratio to degrade, and thus weak echo signals may be lost in noise.
Frequency of transmission	This will vary with the system, i.e. depth sounding or Doppler speed log.
Angle of incidence of the propagated beam	The closer the angle to vertical the greater will be the energy reflected by the seabed.

3.2 SENSORS

In the following section a variety of sensor types, their workings and useful applications, are presented. A selection is made for sensor types that can be used underwater, in an environment which is deprived of a GPS coverage.

The shortcomings and strength of the different sensor are often fused together with a complementary filter, where a mathematical filter is used to mix and merge the two values, or by use of a Kalman filter which is an algoritm, that uses a series of measurements observed over time, containing statistical noise and other inaccuracies and produces estimates of unknown variables that tend to be more precise than those based on a signle measurement alone. The following sections briefly describe the workings of an accelerometer, gyroscope magnetometer and a pressure sensor, which will be used in Section 3.3.1, where these sensors will be fused together with a Kalman filter to obtain an accurate heading and positioning system.

The sensors described in Section 3.2.1 determine the state of a dredge bot; namely its orientation and position. The sensors described in Section 3.2.2 describes a variety of sensors which are needed to gauge the environment.

3.2.1 STATE SENSING

In order for a dredge bot to perform its tasks it has to be aware of its state. As described in Section 4.3.1, the state vector $\vec{x}_k$ describes the position in a global reference frame and the orientation of the dredge bot itself. This state vector can be obtained using a fusion of multiple sensors, which are described below.

INERTIAL MEASUREMENT UNIT

Leccadito, Bakker, Niu, *et al.* [66] describes Inertial Measurement Unit (IMU) as a platform of sensors which output measurements of the vehicle state, such as angular rates and accelerations. The sensors usually consist of a gyroscope, which outputs angular rates about the three vehicle axes, and accelerometer, which output acceleration also along each of the three axes. These sensors are sometimes complemented with a magnetometer, which measures the strength of a magnetic field, like the one generated by the earth, along three axes.

ACCELEROMETER

There are many different types of Micro Electro Mechanical System (MEMS) based accelerometers. The more expensive MEMS are laser and optical based, whilst cheaper models are piezoresistive, capacitive sensing and piezoelectric. Leccadito, Bakker, Niu, *et al.* [66] describes the working of an accelerometer as follows; the sensor can be thought of as a ball in a box. If the accelerometer meter is still and there are no forces present, the sensor will measure $0.0\,\mathrm{m/s^2}$ on all three axes; the ball is suspended in air. If the sensor is suddenly moved, the ball will hit the wall with an opposing force compared to the movement. An acceleration can be measured because of Newton's second law $F = ma$.

In the scenario where there is no external forces present, the accelerometer would only measure the acceleration of the opposite direction of movement, however, on earth there is the external force of gravity pulling on the sensor. If the sensor is positioned on a flat surface with the z-axis aligned as up and down, x-axis left and right and y-axis forward and back, gravity will always be in the negative z direction.

Due to the gravitational pull, an accelerometer can be used to calculate the heading because the sensed acceleration is divided amongst the walls that the ball is in contact with, as is shown in figure 3.7. These measurements can be directly computed into position or Euler angles roll ϕ_{IMU} and pitch θ_{IMU} using trigonometry which is shown in equation 3.6. This allows the magnetometer to calculate a heading angle, which will be described in section 3.2.1.

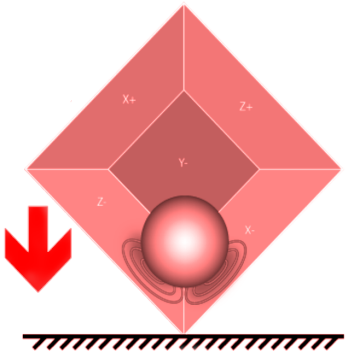


FIGURE 3.7: GRAVITATIONAL PULL ON MULTIPLE AXES [66]

$$\begin{bmatrix} \psi_{IMU} \\ \theta_{IMU} \\ \phi_{IMU} \end{bmatrix} = \begin{bmatrix} \arctan\left(-\frac{a_y}{a_z}\right) \\ \arcsin -\frac{a_x}{\sqrt{{a_x}^2+{a_y}^2+{a_z}^2}} \\ \text{Magnetometer Heading} \end{bmatrix} \tag{3.6}$$

Since acceleration can be integrated over time as velocity, which in turn can be integrated over time as a distance travelled. accelerometers can be used as a dead-reckoning device determining a location, with respect to a starting position, in a GPS deprived environment. Abyarjoo, Barreto, Cofino, *et al.* [74] states that the problem with accelerometers is that they measure both acceleration, due to the device's linear movement, and acceleration due to the earth's gravity, which is pointing toward earth. Since it cannot distinguish between these two accelerations, there is a need to separate gravity and motion acceleration by filtering. This is also described by Nistler and Selekwa [54], who further states that it should be clear that the measurement for a robotic vehicle on an irregular terrain needs to be processed further if they are to be used in the robot odometry system.

Possible sources of error with MEMS accelerometers are identified as effects of temperature and discretization of an analog signal to its digital representation. Abyarjoo, Barreto, Cofino, *et al.* [74] observed no drift of the signal but established that it contains a lot of noise. Kownacki [51] describes that a Kalman filter is a good candidate to filter the noise, using a gyroscope where the Analog Digital Conversion (ADC) stores an obtained analog value as a digital representation. This is usually done with a resolution between $2^{10}[bit]$ and $2^{16}[bit]$, resulting in a resolution of 1024, 2048 till 65536 but discretization of a continuous signal inherently degrades it.

GYROSCOPE

gyroscope has been used for many years in navigation. They usually involve a spinning object, which is tilted perpendicular to the spin, allowing the angle of the reference surface to be measured. The angle is affected by tilting or rotating. Gyroscope which are usually used in electronics, are also classed as MEMS . They are based on other principles such as a laser ring, which observes a phase shift between two laser being sent in a circular path. However, these sensor are expensive and the cheaper alternative is a gyroscope which uses a piezoelectric sensor that works because of a Coriolis effect coupled with vibrations.

Leccadito, Bakker, Niu, *et al.* [66] tells that most MEMS gyroscopes are based on the tuning fork structure, where the Coriolis effect is used to measure ω, this is accomplished by two masses oscillating in opposite directions. When a rotation is applied, the masses are affected by the Coriolis force and the displacement is measured by a change in capacitance, as is shown in figure 3.8. The heading, at a certain axis, can be calculated using the Trapezoidal rule which is a technique for approximating the definite integral. Equation 3.7 illustrates how to obtain the current heading from a discrete sample set.

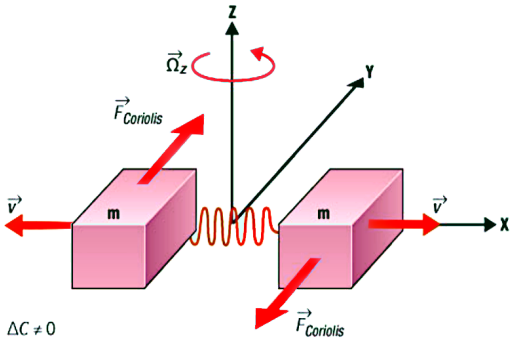


FIGURE 3.8: GYROSCOPE USING CORIALIS EFFECT [66]

$$\theta_n = \int_{t_{n-1}}^{t_n} \omega \mathrm{d}x = \sum_{t_{n-1}}^{t_n} \omega \mathrm{d}t \approx \theta_{n-1} + (t_n - t_{n-1}) \left[\frac{\omega(t_{n-1}) + \omega(t_n)}{2} \right] \tag{3.7}$$

Abyarjoo, Barreto, Cofino, *et al.* [74] observed that the computed results drift over time. The explanation for this phenomenon is that the integration accumulates the noise over time and turns noise into the drift, which yields unacceptable results. Another source of drift is temperature related, Feng, Li, and Zhang [76] state that a gyroscope is sensitive to temperature variations, so the surrounding temperature variations lead to a bias drift of the gyroscope. Then, as an error of the angular velocity, the drift causes error accumulation in the orientations where this drift is not linear with temperature. Equation 3.8 shows the model of a MEMS gyroscope drift, ω_t is the true angular velocity, but unknown and B_d is a slow changing component of the signal; this is the gyroscope drift. n_s is the stochastic component of a signal.

$$\omega = \omega_t + B_d + n_s \tag{3.8}$$

Abyarjoo, Barreto, Cofino, *et al.* [74] further states that the slow chancing component of the gyroscope is not only related to the measured temperature of the MEMS, but also related to the temperature gradient of the surroundings. Because the temperature gradient and the rate of temperature variation have a linear relationship, the slow-chancing component B_d can be modelled, as shown in equation 3.9 a, b, and c are the parameters of the model [31] and T is the measured temperature of the gyroscope in K and $T^{'}$ is the rate of temperature variation in time.

$$B_d = aT + bT^{'} + c \tag{3.9}$$

Other sources of errors are the conversion from the generated analog signal to a digital representation. The ADC in a MEMS stores the obtained analog value as a discrete digital representation with a certain sequence of bits. This is usually done in word with a resolution between $2^{10}[bit]$ and $2^{16}[bit]$, resulting in a resolution of 1024, 2048 till 65536 which should be stored in two registries. Discretization of a continuous signal inherently degrades it.

MAGNETOMETER

A magnetometer measures the strength of a magnetic field. A MEMS magnetometer operates by detecting the effects of the Lorentz force resulting in a change in voltage or resonant frequency which can be measured electronically. Leccadito, Bakker, Niu, *et al.* [66] who explains that a magnetometer coupled with an accelerometer can effectively calculate a heading angle. This is further explained by

Konvalin [39] whom explain that raw magnetometer measurements cannot be used to calculate the heading angle due to the decrease in sensitivity as elevation and bank angles increase, introducing error. In order to obtain the correct heading, a rotation must first be applied removing the bank angle, after which removes the pitch angle. This can be obtained by equation 3.6 where the heading, or yaw ψ_{IMU} can be calculated following equations 3.10 through 3.12 where x_m, y_m and z_m are the raw magnetometer values.

$$x_h = x_m \cos\theta_{IMU} + z_m \sin\theta_{IMU} \tag{3.10}$$

$$y_h = x_m \sin\phi_{IMU} \sin\theta_{IMU} + y_m \cos\phi_{IMU} - z_m \sin\phi_{IMU} \cos\theta_{IMU} \tag{3.11}$$

$$\phi_{IMU}(y_h, x_h) = \begin{bmatrix} \arctan\left(\frac{y_h}{x_h}\right) & \text{if } x_h > 0 \\ \arctan\left(\frac{y_h}{x_h}\right) + \pi & \text{if } x_h < 0, y_h \geq 0 \\ \arctan\left(\frac{y_h}{x_h}\right) - \pi & \text{if } x_h < 0, y_h < 0 \\ +\frac{1}{2}\pi & \text{if } x_h = 0, y_h > 0 \\ -\frac{1}{2}\pi & \text{if } x_h = 0, y_h < 0 \\ \text{undef} & \text{if } x_h = 0, y_h = 0 \end{bmatrix} \tag{3.12}$$

The main sources of error using a magnetometer are distortions of the earth's magnetic field, which can be classified in two categories: soft and hard iron. Hard iron distortions can be described as a constant additive disturbance in the magnetic field of the magnetometer which can be created by ferrous materials around the sensors such as the construction of a crawler and the casing of the electronics and hydraulics. This can create its own magnetic field and adds to the sensor's magnetic fields and is in constant position relative to the sensor. According to Leccadito, Bakker, Niu, *et al.* [66], such a distortion is constant and can be eliminated by a constant offset or bias. To eliminate the offset equation 3.14 can be used. Where $\vec{m}$ is the raw magnetometer vector and $\vec{m}_{hi}$ is the hard iron adjusted vector. This is offset from centre obtained by averaging the minimum and maximum value in n calibration values obtained by rotating the sensor in the iron casing. Since this value will be constant, it can be stored in memory.

$$\vec{m} = \begin{bmatrix} x_m \\ y_m \\ z_m \end{bmatrix} \tag{3.13}$$

$$\vec{m}_{hi} = \vec{m} - \frac{\min(\vec{m})_n + \max(\vec{m})_n}{2} \tag{3.14}$$

Soft iron distortions are different from hard iron disturbances since they don't necessarily generate their own magnetic field. Leccadito, Bakker, Niu, *et al.* [66] describes that soft iron effects on the sensor are determined by the orientation of the materials, and it's usually a perturbation of a circular magnetic field to an ellipse. Calculating the soft iron distortion is computationally more expensive than the hard iron elimination.

it's assumed that tilt compensation (eq. 3.12) and hard iron offset (eq. 3.14) are already performed at this stage and that the centre of the ellipse is positioned at point $(0, 0)$, which is drawn in figure 3.9. The first step is to calculate the magnitude of each point on the ellipse and find the smallest and greatest

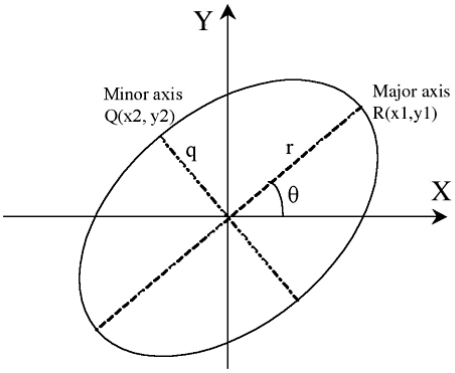


FIGURE 3.9: SOFT IRON DISTORTION [39]

value, using equation 3.15 and 3.16. The y-index of the greatest magnitude should be stored in y_1, which together, can be used to calculated θ_{si}, as is shown in equation 3.17. By scaling and rotating the hard iron $\vec{m}_{hi}$ vector, a correct heading $\vec{m}_{si}$ can be calculated, which is shown in equation 3.18.

$$r_{si} = \max\left(\sqrt{x_n^2 + y_n^2}\right) \tag{3.15}$$

$$q_{si} = \min\left(\sqrt{x_n^2 + y_n^2}\right) \tag{3.16}$$

$$\theta_{si} = \arcsin\left(\frac{y_1}{r_{si}}\right) \tag{3.17}$$

$$\vec{m}_{si} = \frac{q_{si}}{r_{si}} \begin{bmatrix} \cos\theta_{si} & \sin\theta_{si} & 0 \\ -\sin\theta_{si} & \cos\theta_{si} & 0 \\ 0 & 0 & 1 \end{bmatrix} \vec{m}_{hi} \tag{3.18}$$

PRESSURE SENSOR

White [56] describes a fluid pressure p as the normal shear stress on any plane through a fluid element at rest is an infinitesimal point property, which is taken positive for compression, by convention. This can be described by equation 3.19. Here p is the pressure at a certain depth, which is comprised of the specific weight of water $\gamma_w(T)$ as a function of temperature, and the total mass of water on top of that point z.

$$p = p_a - \gamma_w(T)z \tag{3.19}$$

Since pressure is a function of γ_w special consideration regarding the impact of soil disturbance due to dredging activities in water, has to be made. The specific weight of the water column above the sensor changes when sediment is mixed with water above the pressure sensor. IHC MTI B.V. (MTI) dredging specialists Dr. ir. van Wijk and ir. Hoftsra both estimate that the disturbed sediment won't drift higher than $2.0\,\mathrm{m}$ for a sediment with an *in situ* specific weight of $\gamma_{sw} = 1400.0\,\mathrm{N/m^3}$. That mixture will in all likelihood have a specific weight of $\gamma_m = 1200.0\,\mathrm{N/m^3}$ because the specific weight of water is $\gamma_w = 1000.0\,\mathrm{N/m^3}$. The error when calculating depth with a pressure sensor is dependent on the

position of the sensor with regards to the bottom.

Using equation 3.20, Δp is the pressure difference between the specific weight of a column of water γ_w compared with a column of water and sediment γ_m of a certain height z_p. The specific weight consists of the density of a fluid ρ_w for water or ρ_m for mixture multiplied with a gravitational acceleration vector g. When the allowed z_ϵ is known. A height for the pressure sensor, with regards to the top fluid column can be obtained. it's estimated that an acceptable error in depth readings is $200.0\,\mathrm{mm}$, when using equation 3.20 gives a minimum sensor height of $1.9\,\mathrm{m}$ from the soil bed. This indicates that the sensor should be placed at the top of a dredge bot, away from the disturbance source.

$$\left.\begin{aligned} \Delta p &= (\gamma_w - \gamma_m)\, z_p \\ \gamma_w &= \rho_w g \\ \gamma_m &= \rho_m g \\ z_\epsilon &= \frac{\Delta p}{\gamma_w} \end{aligned}\right\} z_\epsilon = \frac{-(\rho_m - \rho_w)\, z_p}{\rho_w} \Longrightarrow z_p = \frac{z_\epsilon \rho_w}{\rho_m - \rho_w} \tag{3.20}$$

Three types of pressure measurements are usually performed, according to Webster [14]:

TYPE	DESCRIPTION
Absolute pressure	Where the pressure is measured against an perfect vacuum where pressure is zero.
Gage pressure	Is the pressure difference between the point of measurement and the ambient.
Differential pressure	Is the pressure difference between two points, one of which is chosen to be the reference. In reality, both pressures can vary, but only the pressure difference is of interest here.

Since pressure is defined as the force per unit area, the most direct way of measuring pressure is to isolate an area on an elastic mechanical element for the force to act on. The deformation of the sensing element produces displacements and strains that can be precisely sensed to give a calibrated measurement of the pressure [14].

NOTE 3.1: SCOPE

Although there are a multitude of pressure sensing techniques, such as: seals, snubbers, bellow, bourdon, helical, diaphragm, differential, electronic and manometers. This research will focus on diaphragm and electronic pressure sensors, since these are commonly used throughout the industry and easily integrated in a crawler. Where the focus lies on behavior and characteristics.

Pressure sensors that depend on deflection of a diaphragm have been used for centuries, the last few decades the elastic hysteresis, friction and drift effects have been reduced to $\pm 0.1\,\%$. This is mainly due to the use of a microprocessor, which applies memorized non-linearity corrections [24].

Detection methods are usually capacitive pressure sensors, which are highly accurate (better than $0.1\,\%$) and can cover a high pressure range, from nearly vacuum $(1.0 \times 10^{-1}$ to $1.0 \times 10^{7})\,\mathrm{Pa}$. These sensors rest on the principle, where a metal or silicon diaphragm serves as the pressure sensing element and is regarded as one electrode of a capacitor. The other electrode is stationary and usually consists of a deposited metal layer on a ceramic or glass substrate. When a pressure is applied the diaphragm deforms and the changes in between electrodes is changed which results in a change in capacitance.

Where an alternative are the piezoresistive pressure sensors, which are the most common type of pressure sensor in use. These sensors measure the pressure by measuring the change in electric resistance of a material when stresses or strains are applied. Both Webster [14] and Liptak [24] state that semiconductor, such as silicon and germanium, are by far the most appealing sensing elements in this type of sensor. The most attractive characteristic of semiconductors is their sensitivity, which is close to 100 times greater than that of metallic wires.

3.2.2 EXTERNAL SENSOR

A crawler needs to be aware of its surroundings; It needs to sense how far objects and landmarks are in respect to its own position and orientation. Awareness of its environment can be used to minimize dead-reckoning drift, defined the work area and help avoid collisions.

NOTE 3.2: SCOPE

Although there are multiple examples of AUVs that make use of EMW, light or computer vision to help them sense its environment, these are deemed not usable for a crawler. EMW has a limited range in salt water environments (discussed in Section 3.1.2), light will scatter due to diffraction created by floating floating sand particles and computer vision can be problematic due to limited light sources and low contrast conditions. Which is also a result of floating sand particles. The focus of this research will therefor lie on acoustic sensing.

3.3 LOCATION UNDER UNCERTAINTY

Due to the absence of an ubiquitous global localization system such as GPS in underwater environments, crawler navigation is confined to these three primary methods: (1) dead-reckoning, (2) time of flight acoustic navigation, and (3) geophysical navigation.

The most obvious and longest established technique is dead-reckoning, which consist in integrating vehicle velocity measurements from sensors such as accelerometers and gyroscopes to obtain new position estimates. The problem with exclusive reliance on dead-reckoning is that the position error increases without bound as the distance travelled by the crawler [70]. It will be illustrated in section 3.3.1 that the position error can be limited by making use of sensor fusion. But this won't be enough. The effects of sporadically position updating using stationary Long Base Line (LBL) and Ultra Short Base Line (USBL) is also shown. Both LBL as USBL make use of acoustic energy, which is described in more detail in section 3.1.2. Because acoustic energy is known for its excellent travel characteristics underwater, it is common practice to deploy those transponders as beacons, such that they can update the position and bound the dead-reckoning error.

Geophysical navigation such as Terrain Relative navigation (TRN) and Simultaneous Localization And Mapping (SLAM) are up and coming methods which show potential. These use the characteristic of a terrain, perceived through their sensors, to obtain their position. These methods are not discussed in this paper.

Other methods for navigation under uncertainty are based on probabilities taken into account *a priori* known characteristics of sensors and actuators such as Linear-Quadratic Gaussian Motion Planning (LQG-MP) and Rapidly exploring Random Trees (RRT). According to Galceran, Nagappa, Carreras, *et al.* [64], these methods are theoretically satisfactory but they require discretization of the environment and will, as a result, suffer from scalability problems. They propose the use of *a priori* known bathymetric map which is a submerged equivelant of an above-water topographic map. It classifies this method as off-line and therefore unsuitable to be employed for an autonomous operating crawler. These will therefore not be described in this thesis.

Recent studies have been focused on minimizing uncertainties using multiple robots, such as the leap-frog strategy proposed by Tully, Kantor, and Choset [48], which uses a team of three robots where two alternating robots act as stationary beacons. Others like Wei Gao, Yalong Liu, and Bo Xu [73] use a single surface which act as a communication and navigation aid. it's quite common to filter sensor readings and state vectors from the multiple robots using a Kalman filter.

This chapter begins with a dive into the Kalman filter. The sections below describe how the state representation $\vec{x}_k$ of a crawler can be obtained using a Kalman filter, which fuses multiple sensors together. It will then explore how the growth of errors can bound, using a sporadically obtained position estimate from an alternative source, such as moving or stationary beacons.

3.3.1 LOCALIZATION REFINEMENT USING KALMAN FILTERS

A crawler has multiple sensors on-board to establish were and what its orientation is; these will in a likelihood, be a gyroscope, accelerometer, magnetometer and a pressure sensor. It was established in section 3.2.1 that each of these sensors have their own limitations and strengths. it's common practice to fuse multiple sensors together to counteract these limitations with strengths of the other sensors. Kalman filters or, as they are also known Linear Quadratic Estimations (LQEs) are a tried and practiced method to achieve this.

Section 3.3.1 explains the filter using a simple example of a falling ball with only gravity working on the ball.

3.3.2 BASIC KALMAN FILTERING

Before a Kalman filter can be designed it's important that the basics are explained. This section will feature a short description of the background and workings of a Kalman filter and quaternion, which is a number system that extends the complex numbers. They were first described by William Rowan Hamilton in 1843. Hamilton defined a quaternion as the quotient of two directed lines in a 3-dimensional space or equivalently as the quatient of two vectors.

In 1960, R.E. Kalman published his paper Kalman [2] — *"A new approach to linear filtering and prediction problems"*. In this paper he described a recursive solution to the discrete-data linear filter problem. Welch and Bishop [32], d'Andréa-Novel and Lara [62], Chui and Chen [13], Grewal and Andrews [78] all describe how Kalman filters have had a huge impact on control theory and have been subject to extensive research and application. The paragraphs below are based on the theory proposed by Kalman [2].

A Kalman filter can be used to control a dynamic model, especially those represented by systems of linear differential equations. These generally come from the laws of physics. The real-world dynamics are used to model the state dynamics which should contain a fairly faithful replication of the true system dynamics. The state of a falling object in one dimension can be described with the state $\vec{x}_k = [s_z, v_z]^T$. Here s_z and v_z.

A Kalman filter works by estimating the state of a process based on *a priori* states. it's in effect an optimal estimator based on a prediction made from the previous input and current input. The Kalman filter addresses the general problem of trying to estimate the state $\hat{x}_k \in \mathrm{Re}^n$ of a discrete-time controlled process that is governed by the linear stochastic difference equation 3.21. Here $\mathbf{F}$ is a state transition model which is applied to the previous state $\vec{x}_{k-1}$ to estimate the current state. $\mathbf{B}$ is the control-input model which is applied to the control vector $\vec{u}_k$. The process noise $\vec{w}_k$ is assumed to be white with a normal probability distribution. The $\mathbf{Q}_k$ is the process noise covariance. Each predicted step is updated with a measurement, which is shown in equation 3.22. Here the measurement $\vec{z}_k \in \mathrm{Re}^m$, $\mathbf{H}$ is the observation model, which maps the true state space $\vec{x}_k$ into the observed space, whilst taking into account the observation noise $\vec{v}_k$, which is assumed to be unrelated to $\vec{w}_k$, and is white with a normal probability distribution. Where $\mathbf{R}$ is the measurement noise covariance.

$$\hat{x}_k = \mathbf{F}\vec{x}_{k-1} + \mathbf{B}\vec{u}_k + \vec{w}_k, \qquad p(\vec{w}_k) \sim N(0, \mathbf{Q}_k) \tag{3.21}$$

$$\vec{z}_k = \mathbf{H}\vec{x}_k + \vec{v}_k, \qquad p(\vec{v}_k) \sim N(0, \mathbf{R}) \tag{3.22}$$

Figure 3.10 shows the algorithm as a flow diagram. It starts with an initial assumption of the state $\vec{x}_0$ and $\mathbf{P}_0$, which is the initial state of the error covariance matrix. This can be described as a measure of the estimated accuracy of the state estimate.

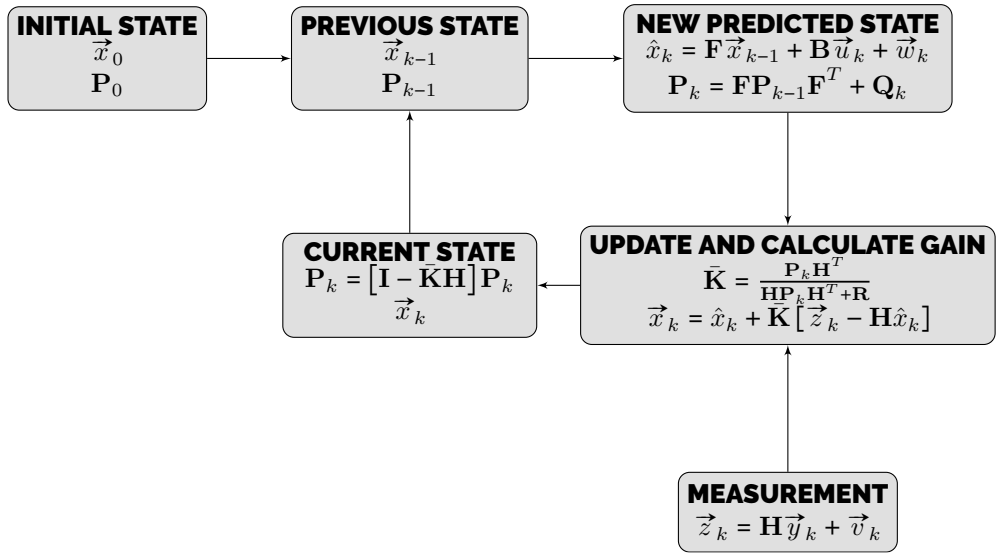


FIGURE 3.10: FLOW OF A KALMAN FILTER

These initialized values are fed in the loop as a **previous state** with which a **new predicted state** is estimated using equation 3.21. If the previous example of a falling object in one dimensions is used, with the state $\vec{x}_k = [s_z, v_z]^T$. A prediction can be made of the position in the next time step. This will follow the equation $s_{z,k} = s_{z,k-1} + v_{z,k-1}\Delta t + \frac{1}{2}a_{z,k-1}\Delta t^2$ and the new velocity $v_{z,k} = v_{z,k-1} + a_{z,k}\Delta t$.

For simplicity sake, the process noise $\vec{w}_k$ is set to zero. The matrix $\mathbf{F}$ is used to map the previous state to the new state. Whereas the matrix $\mathbf{B}$ is used to map the control variable $\vec{u}_k$ to the new state. These variables dictate the change; in the case of our example it will be an acceleration due to gravity a_z = -g. Equation 3.23 illustrates the new estimation of the state of our example. Here Δt is an incremental time step.

$$\hat{x}_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} s_{z,k-1} \\ v_{z,k-1} \end{bmatrix} + \begin{bmatrix} \frac{1}{2}\Delta t^2 \\ \Delta t \end{bmatrix} [a_{z,k}] + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} s_{z,k-1} + v_{z,k-1}\Delta t + \frac{1}{2}a_{z,k}\Delta t^2 \\ v_{z,k-1} + a_{z,k}\Delta t \end{bmatrix} \tag{3.23}$$

The new predicted error of the estimate, known as the error covariance matrix $\mathbf{P}_k$, is used to map the covariance between the i^{th} and j^{th} elements of the state vector $\vec{x}_k$. In this example it's initially assumed that error between the state of its position s_z and the velocity are unrelated. The assumption is also made that the position has an error of σs_z and the velocity σv_z. From this, a simple error covariance matrix can be constructed which can be used in equation 3.24 where a new error covariance matrix can be calculated, as is shown in equation 3.25. The noise matrix $\mathbf{Q}_k$ is set to zero.

$$\mathbf{P}_k = \mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^T + \mathbf{Q}_k \tag{3.24}$$

$$\mathbf{P}_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \sigma_{s_z}^2 & 0 \\ 0 & \sigma_{v_z}^2 \end{bmatrix} \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}^T + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} \Delta t^2\sigma_{v_z}^2 + \sigma_{s_z}^2 & \Delta t\sigma_{s_z}^2 \\ \Delta t\sigma_{s_z}^2 & \sigma_{s_z}^2 \end{bmatrix} \approx \begin{bmatrix} \Delta t^2\sigma_{v_z}^2 + \sigma_{s_z}^2 & 0 \\ 0 & \sigma_{s_z}^2 \end{bmatrix} \tag{3.25}$$

Once the prediction of a new state and error covariance matrix is made, the **measurements** can be calculated. It is important to note that only the inputs and outputs of the system can be measured. Equation 3.26 shows the measured values, mapped to the state space $\vec{x}_k$, where $\mathbf{H}$ is the measurement sensitivity matrix defining the linear relationship between the state of a dynamic system and the measurements that can be made, which for now is set equal to a 2×2 identity matrix. Let us assume that for our example only the position can be measured $s_{z,m,k}$ and that the measurement noise is assumed to be zero.

$$\vec{z}_k = \mathbf{H}\vec{y}_k + \vec{v}_k \tag{3.26}$$

$$\vec{z}_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} s_{z,m,k} \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} s_{z,m} \\ 0 \end{bmatrix} \tag{3.27}$$

With the predicted state and the obtained measurements a new state can be estimated. During this **update** phase, we determine how much weight the Kalman filter needs to put in to its measurements compared to its predicted state. This can be done by calculating the $\bar{\mathbf{K}}$ Kalman gain, which is the relative weight given to the measurement and current state estimate, and can be "tuned" to achieve a particular performance. With a heigh gain, the filter places more weight on the most recent measurements, and thus follows them more responsively. With a low gain, the filter follows the predictions more closely. At the extremes, a high gain close to one will result in a more jumpy estimated trajectory, whilst low

gain close to zero will smooth out noise but decrease the responsiveness. It can be calculated with equation 3.28 where $\mathbf{R}$ is the covariance matrix of observational (measurement) uncertainty.

$$\bar{\mathbf{K}} = \frac{\mathbf{P}_k \mathbf{H}^T}{\mathbf{H}\mathbf{P}_k \mathbf{H}^T + \mathbf{R}} \tag{3.28}$$

$$\bar{\mathbf{K}} = \frac{\begin{bmatrix} \Delta t^2 {\sigma_v}^2 + {\sigma_s}^2 & 0 \\ 0 & {\sigma_s}^2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}^T}{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \Delta t^2 {\sigma_v}^2 + {\sigma_s}^2 & 0 \\ 0 & {\sigma_s}^2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}^T \begin{bmatrix} {\sigma_{s,m}}^2 & 0 \\ 0 & {\sigma_{v,m}}^2 \end{bmatrix}} = \begin{bmatrix} \frac{\Delta t^2 {\sigma_v}^2 + {\sigma_s}^2}{\Delta t^2 {\sigma_v}^2 + {\sigma_s}^2 + {\sigma_{s,m}}^2} & 0 \\ 0 & \frac{{\sigma_v}^2}{{\sigma_v}^2 + {\sigma_{v,m}}^2} \end{bmatrix} \tag{3.29}$$

The Kalman gain obtained in equation 3.28, is used in equation 3.30 where a new state is calculated by taking the predicted state $\vec{x}_k$ calculated with equation 3.23 and adding the innovation multiplied with the Kalman gain $\vec{z}_k - \mathbf{H}\hat{x}_k$. Innovations are the differences between observed and predicted measurements. Grewal and Andrews [78] states that they are the carotid artery of a Kalman filter. They provide an easily accessible point for monitoring vital health status without disrupting normal operations, and the statistical and temporal properties of its pulses can tell us much about what might be right or wrong with a Kalman filter implementation.

From the worked out 1-dimensional example it becomes apparent that the state variable, calculated in equation 3.31, is a weighted average between the measurements and the prediction, normalized against the error of the covariance, between the state variables.

$$\vec{x}_k = \hat{x}_k + \bar{\mathbf{K}}\left[\vec{z}_k - \mathbf{H}\hat{x}_k\right] \tag{3.30}$$

$$\begin{aligned} \vec{x}_k = &\begin{bmatrix} s_{z,k-1} + v_{z,k-1}\Delta t + \frac{1}{2}a_{z,k}\Delta t^2 \\ v_{z,k-1} + a_{z,k}\Delta t \end{bmatrix} + \begin{bmatrix} \frac{\Delta t^2 {\sigma_v}^2 + {\sigma_s}^2}{\Delta t^2 {\sigma_v}^2 + {\sigma_s}^2 + {\sigma_{s,m}}^2} & 0 \\ 0 & \frac{{\sigma_v}^2}{{\sigma_v}^2 + {\sigma_{v,m}}^2} \end{bmatrix} \cdots \\ &\left[\begin{bmatrix} s_{z,m,k} \\ 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} s_{z,k-1} + v_{z,k-1}\Delta t + \frac{1}{2}a_{z,k}\Delta t^2 \\ v_{z,k-1} + a_{z,k}\Delta t \end{bmatrix}\right] = \cdots \\ &\begin{bmatrix} \frac{s_{z,m}\Delta t^2 {\sigma_v}^2 + 2a_{z,k}\Delta t^2 + 2v_{z,k-1}\Delta t {\sigma_{s,m}}^2 + s_{z,m,k}{\sigma_s}^2 + s_{k-1}{\sigma_{s,m}}^2}{\Delta t^2 {\sigma_v}^2 + {\sigma_s}^2 + {\sigma_{s,m}}^2} \\ \frac{{\sigma_{v,m}}^2 (v_{z,k-1} + 2a_{z,k}\Delta t)}{{\sigma_v}^2 + {\sigma_{v,m}}^2} \end{bmatrix} \end{aligned} \tag{3.31}$$

This newly obtained state, or **current state** $\vec{x}_k$ can be used in the next iteration. During this phase a new process covariance matrix $\mathbf{P}_k$ is calculated with equation 3.32. Where the matrix $\mathbf{I}$ is a 2×2 identity matrix. Both the new state, obtained from equation 3.30 and the newly obtained process covariance matrix are set as the previous iteration.

$$\mathbf{P}_k = \left[\mathbf{I} - \bar{\mathbf{K}}\mathbf{H}\right]\mathbf{P}_k \tag{3.32}$$

$$\mathbf{P}_k = \left[\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} \frac{\Delta t^2 {\sigma_v}^2 + {\sigma_s}^2}{\Delta t^2 {\sigma_v}^2 + {\sigma_s}^2 + {\sigma_{s,m}}^2} & 0 \\ 0 & \frac{{\sigma_v}^2}{{\sigma_v}^2 + {\sigma_{v,m}}^2} \end{bmatrix}\right]\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\left[\begin{bmatrix} \Delta t^2 {\sigma_v}^2 + {\sigma_s}^2 & 0 \\ 0 & {\sigma_s}^2 \end{bmatrix} = \dots\right.$$
$$\left.\begin{bmatrix} \frac{{\sigma_{s,m}}^2\left(\Delta t^2 {\sigma_v}^2 + {\sigma_s}^2\right)}{\Delta t^2 {\sigma_v}^2 + {\sigma_s}^2 + {\sigma_{s,m}}^2} & 0 \\ 0 & \frac{{\sigma_v}^2 {\sigma_{v,m}}^2}{{\sigma_v}^2 + {\sigma_{v,m}}^2} \end{bmatrix}\right] \tag{3.33}$$

The above described example of a falling ball can be simulated with a Python script. Such a script can be found in appendix D. Results from such a simulation are shown in figure 3.11. Were it is clearly evident that the estimated Kalman value, of the position, is a better estimate then the measured values. According to Roger R Labbe jr [85] an effective way to measure the results of a simulated Kalman filters, is the Normalized Estimated Error Squared (NEES). Which can be calculated with equation 3.35, where $\tilde{x}_k$ is the error, or difference, between the ground truth state vector $\vec{x}_{g,k}$ and the estimated filter value $\hat{x}_k$, squared and multiplied with the inverse of the process covariance matrix $\mathbf{P}_k$; All evaluated at time k.

$$\tilde{x}_k = \vec{x}_{g,k} - \hat{x}_k \tag{3.34}$$

$$\epsilon_{N,k} = \tilde{x}_k {\mathbf{P}_k}^{1} \tilde{x}_k \tag{3.35}$$

$$\bar{\epsilon}_N = \frac{1}{k}\sum_{1}^{k} \epsilon_{N,k} \leq n_x \tag{3.36}$$

This means that if the covariance matrix gets smaller, NEES gets larger for the same error. A covariance matrix is the filter's estimate of it's error, so if it's small relative to the estimation error then it's performing worse than if it's large relative to the same estimation error. Equation 3.35 gives a scalar for each time step, which is said to be *chi-squared distributed with n degrees of freedom*. The average NEES value $\bar{\epsilon}_N$ should be less then number of elements in the state space vector n_x, as is shown in equation 3.36. The performance of our example or the $\bar{\epsilon}_N = 1.3$ is shown in figure 3.13.

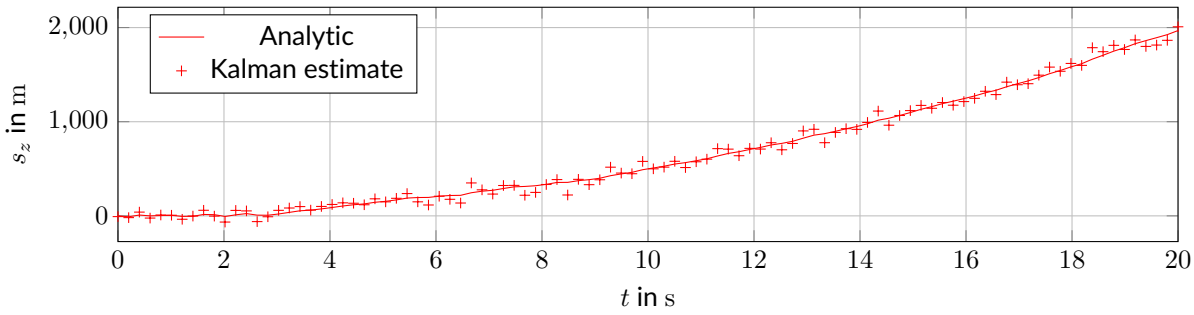


FIGURE 3.11: COMPARISON OF ESTIMATED, MEASURED AND REAL POSITION

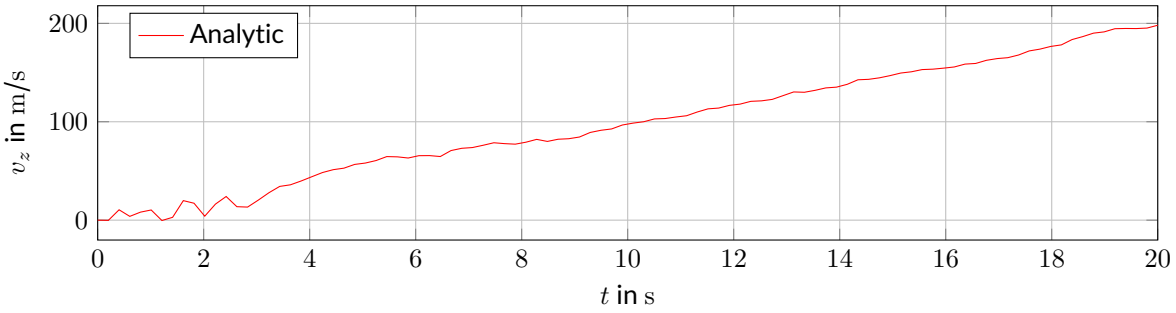


FIGURE 3.12: COMPARISON OF ESTIMATED AND REAL SPEED

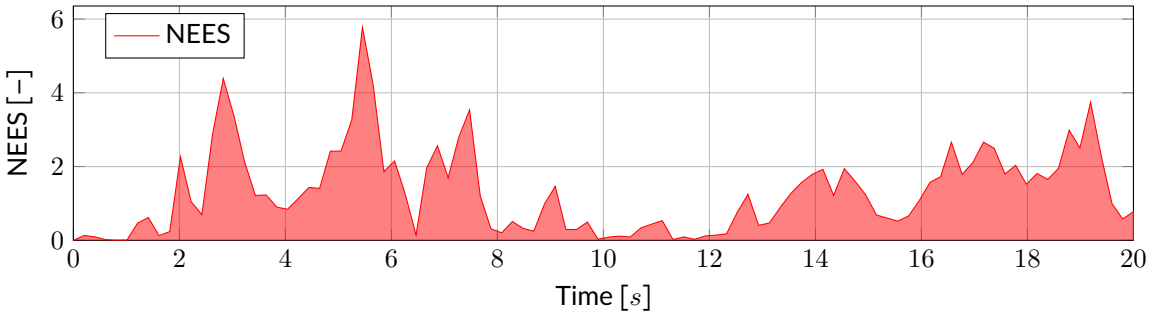


FIGURE 3.13: NEES for Kalman filter

3.4 COVERAGE PATH PLANNING

Two of the use cases described in the introduction dictate that the crawler has to cover the whole area preferably with an optimal path. This type of path planning problem differs from the in Chapter 3.4 described problem. Choset describes the task of determining a path that passes an effector (e.g., a robot, a detector, etc.) over all points in a free space as a coverage path problem [29], hence that this task is called a CPP. This type of task can be found as an integral part of many robotic applications such as vacuum cleaning robots, painter robots, autonomous underwater vehicles creating image mosaic, demining robots, lawn mowers, automated harvesters, window cleaners and inspection of complex underwater structures [63].

All these type of robots need to cover a complete region in order to perform their tasks. According to Cao, Huang, and Hall [6] such a mobile robot should use the following criteria, for a region filling operation

1. The mobile robot must move through an entire area, i.e., the overall travel must cover a whole region.
2. The mobile robot must fill the region without overlapping paths.
3. Continuous and sequential operations without any repetition of paths is required of the robot.
4. The robot must avoid all obstacles in a region.
5. Simple motion trajectories (e.g., straight lines or circles) should be used for simplicity in control.
6. An "optimal" path is desired under the available conditions. it's not always possible to satisfy all these criteria for a complex environment. Sometimes a priority consideration is required.

Galceran [71] describes that these types coverage algorithms can be classified as *heuristic* or *complete* depending on whether or not the provable guarantee complete coverage of the free space. At the same time they can be classified as off-line or on-line. off-line algorithm rely on only on stationary information, and the environment is assumed to be known. Usually on-line algorithms are needed if some kind of adaptivity to the requirement is required. On-line algorithms usually utilize real-time sensor measurements. Thus these algorithms can also be called *sensor-based coverage algorithms*. on-line coverage algorithms are in effect "*divide and conquer*" strategies, which Wong and MacDonald [27] describes as a powerful technique used to solve many problems and many mapping procedures carry out a process of space decomposition, where a complex space is repeatedly divided until simple sub-regions of a particular type are created. The problem at hand is then solved by applying a simpler algorithm to the simpler sub-regions.

Since an autonomous operating crawler can be stationed in different environments with multiple unknown obstacles, the focus of this chapter lies on on-line or sensor-based coverage algorithms from which the following are identified:

- morse-based cellular decomposition
  - On-line Morse-based boustrophedon decomposition
  - Morse-based cellular decomposition combined with generalized Voronoi diagram
- Landmark-based topological coverage
  - Slice decomposition
  - On-line topological coverage algorithm
- Grid-based methods
  - Grid-based coverage using spanning trees
  - Neural network-based coverage on grid maps
- Coverage under uncertainty
- Multi-robot methods

3.4.1 MORSE-BASED CELLULAR DECOMPOSITION

Morse-based cellular decomposition is mostly based upon the following method exact or approximate cellular decomposition Acar, Choset, Rizzi, *et al.* [22] State that exact cellular decompositions represent the free space of a robot by dividing it into non-overlapping region sub-level cells such that the union of the cells fills the free space. Complete coverage is then reduced to ensuring that the robot visits each cell. These cells are constructed using Morse function, a function for which all critical point are non-degenerate and all critical levels are different.

Morse-functions are visualized by Nicolaescu [36] as follows: Suppose M is a smooth, compact manifold which is assumed to be embedded in a Euclidean space $\mathscr{E}$, and from which we would like to understand some basic topological invariants. This is done with a "slicing" technique.

Were a unit vector $\vec{u}$ is fixed in $\mathscr{E}$ and which start slicing M with the family of hyperplanes perpendicular to $\vec{u}$. Such a hyperplane will in general intersect M along a submanifold (slice). The manifold can be recovered by continuously stacking the slices on top of each other in the same order as they were cut out of M.

If this collection of slices is visualized as a deck of cards with various shapes, which are piled up in the order that they were produced, there will be an increasing stack of slices. As this stack grows, it can be observed that at certain moments in time the shape suffers a qualitative change. The theory proposed by Morse extracts quantifiable information, through studying the evolution of this growing stack of slices.

Each moment in time that this pile changes is called a critical value, which correspond to moments in time when the hyperplane intersect tangentially. These points marks the boundary of a cell. Acar, Choset, Rizzi, *et al.* [22] states that Morse theory assures that between those critical point "merging" and "severing" of slices does not occur and that a robot can trivially perform simple motions, such as back and forth motions between critical points and thus guarantee complete coverage of a cell. Hence this method is duped Morse-based cellular decomposition

The above described method is depicted in figure 3.14 (a). Such an environment can be represented with a graph such as shown in figure 3.14 (b). Each critical value corresponds with a node while a cell is represented by an edge.

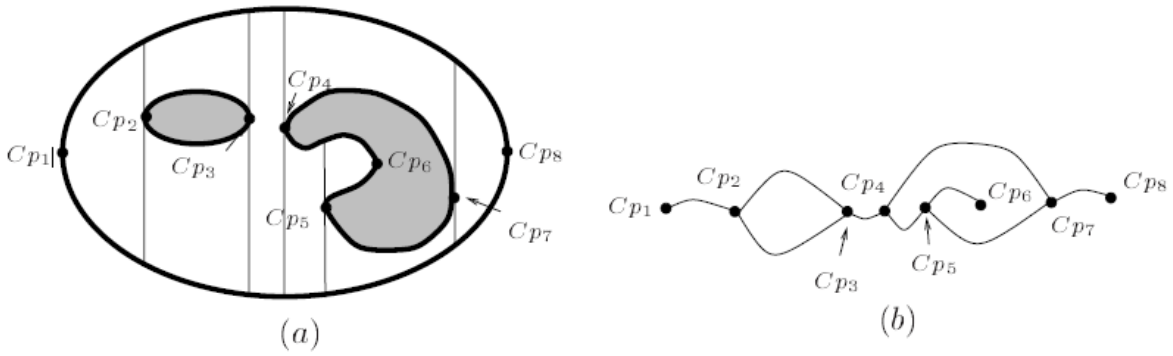


FIGURE 3.14: (a) Exact cellular decomposition, (b) Graph representation

The above describe technique has a minor short-coming, Choset, Acar, Rizzi, *et al.* [16] states that this method may result in many small cell, such as cell 9 shown in figure 3.15, which can seemingly be "clumped" into neighbouring cells. Reorganizing the cells can result in a shorter (more efficient) path to cover the same area. To address this issue, the Boustrophedon Cellular Decomposition (BCD) approach was introduced.

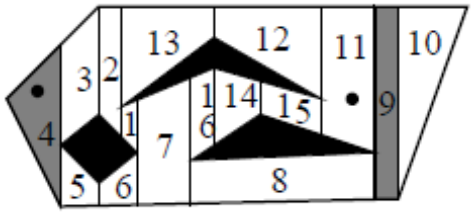


FIGURE 3.15: TRAPEZOIDAL DECOMPOSITION OF BOUND FREE SPACE[16].

Boustrophedon which literally means "*the way of the ox*" merges these cells, such that a more optimal

path can be found. This can be done by using different Morse function, which results in different slice shapes and therefore different cell decompositions, as is shown in figure 3.16 [63][15][22]. Such as spiral, spike or squarel.

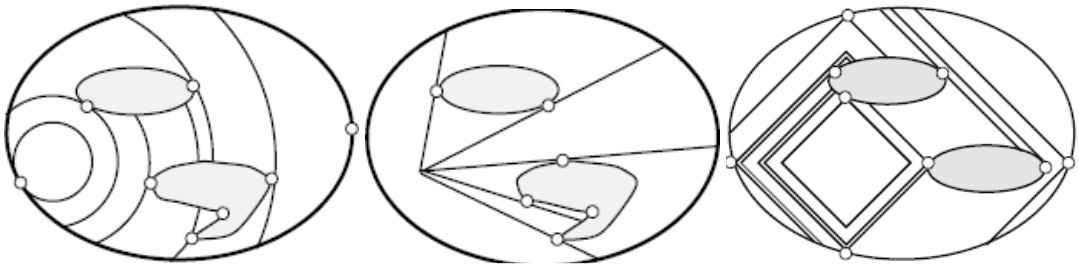


FIGURE 3.16: SPIRAL, SPIKE AND SQUAREL [22]

Once each cell is identified a strategy for the infill is executed, which is described by Huang [20] as coverage paths. Each region [or cell red.] is decomposed into sub-regions, a traveling salesman algorithm, is applied to generate a sequence of sub-regions to visit, and a coverage path is generated from this sequence that covers each subregion in turn. Huang [20] claims that turns take a significant amount of time: the robot must slow down, make the turn and accelerate. Thus by minimizing the number of turns, which are proportional to the altitude of a subregion, an optimal path can be found.

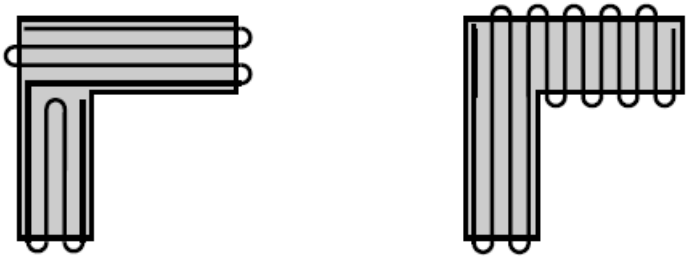


FIGURE 3.17: DIFFERENT SWEEP DIRECTIONS [20]

This is done by first creating an adjacency graph $\mathbf{G}$, generated with the Morse function, see figure 3.14 (b). This graph is split in two $\mathbf{G}_1$ and $\mathbf{G}_2$ sub-graphs. These contain all edges from $\mathbf{G}$ except those that connect a node from $\mathbf{G}_1$ to $\mathbf{G}_2$. With this definition the minimum sum of altitudes can be stated as equation 3.37. Where i iterates over all possibles ways to split graph $\mathbf{G}$ and $C(\mathbf{G})$ returns the cost of covering all cells corresponding to nodes in $\mathbf{G}$, once an optimum is found, movements of the sub-regions are implemented.

Let $\mathbf{G}_1$ and $\mathbf{G}_2$ be a subset of graph $\mathbf{G}$ which consist of all the nodes that needs to be visited and let C be a function that calculates the cost in movement S, which iterative over i for all possible combinations of $\mathbf{G}_1$ and $\mathbf{G}_2$.

$$S(\mathbf{G}) = min\left\{C(\mathbf{G}), \min_i S\left(\mathbf{G}_1{}^i\right) + S\left(\mathbf{G}_2{}^1\right)\right\} \tag{3.37}$$

So far it assumed that the environment is known *a priori* which labels this method as an off-line method. While the use case described in Chapter 1, dictates that the crawler encounters unknown obstacles.

ON-LINE MORSE-BASED BOUSTROPHEDON DECOMPOSITION

Acar and Choset [21] describe a method which allows the use of above portrayed Morse-based cellular decomposition in an unknown dynamically chancing environment. Critical point sensing, is a way to determine critical points based on a sweep direction and an omnidirectional range sensor. These can be detected when the sweep direction and the surface normal $\nabla_m(x)$ of an obstacle are parallel.

On-line region coverage is depicted in figure 3.18 which shows an incremental sweep as part of an on-line BCD (a) The robot starts to cover the space at the critical point Cp_1 and instantiates an edge

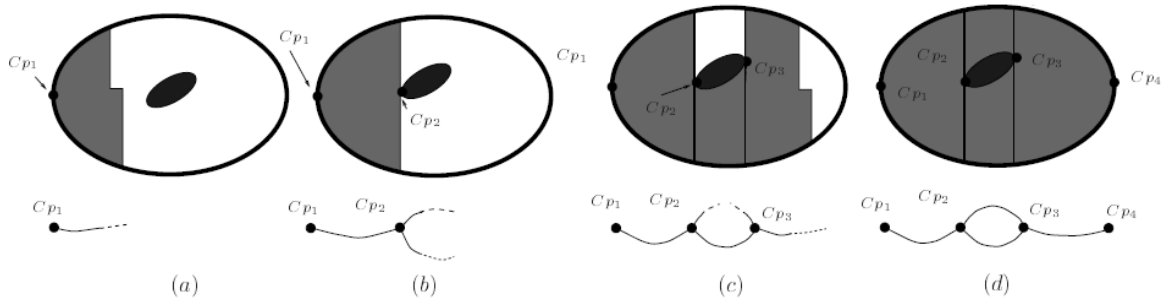


FIGURE 3.18: INCREMENTAL GRAPH CONSTRUCTION [21]

in the graph. (b) When the robot is done covering the cell between Cp_1 and Cp_2, it joins the nodes in the graph that correspond to Cp_1 and Cp_2 with an edge, and start two new edges. (c) The robot covers the cells below the obstacle and to the right of Cp_3. (d) While covering the cell above the obstacle, the robot encounters Cp_2 again. Since all the critical point have explored edges, i.e., covered cells, the robot concludes that it has completely covered the space [21].

On-line detection of critical points is illustrated by Galceran and Carreras [63]. They tell how a robot detects a surface normal which is the same as a gradient. Given a robot located at point x, let Cp_0 be the closest point to x on the surface of obstacle Cp_i:

$$Cp_0 = \underset{x \in Cp_i}{argmin} ||x - Cp||, \tag{3.38}$$

and let $d_i(x)$ be the distance between point x and the obstacle Cp_i. Now, the gradient of $d_i(x)$, $\nabla d_i(x)$ can be calculated as

$$\nabla d_i(x) = \frac{x - Cp_0}{||x - Cp_0||}. \tag{3.39}$$

Since a gradient is a unit vector normal to a surface at a given point and since Cp_0 is a point laying on the surface Cp_i, $x - Cp_0$ is a vector pointing outwards, from Cp_0 to x, by dividing it by its norm $||x - Cp_0||$ it becomes a unit vector. This leads to the conclusion that a critical point occurs when $\nabla d_i(x)$ is parallel to the sweep direction, as illustrated in figure 3.19.

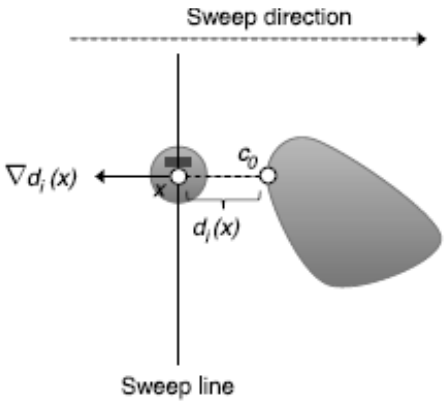


FIGURE 3.19: CRITICAL POINT DETECTING [63]

Points are only detected from the side view. A normal sweeping motion will miss critical point that lay parallel to the sweep direction. This can be counteracted with a Cyclic path. It's important to note that such a path will be longer then a normal zig-zag sweep, since it includes backtracking. A cyclic path starts forward

Cyclic path start by moving the robot in a forward phase, when it hits the boundary, it begins moves

downwards. When an obstacles is encounter during its travels it changes it state into a wall following unit, until reaching the next boundary in front of him or detects an critical point. If the later one is the case, it starts moving upward again. If an obstacles detected during this cycle it will follow that wall until a critical point is detected. It then marks it as a new next strip or cell boundary and moves back, towards the point where it ended its initial upward movement. It now starts filling the cell with general zig-zag sweeps. This incremental construct of the Morse decomposition on-line is stored as Reeb graph. Such a graph has the same functionality as a adjacency graph

The BCD sweep in given in figure 3.18 suggest that the robot will know that it has covered the whole region when it moves from Cp_3 to Cp_2. But this will require an absolute coördination system that tells the robot that Cp_2 is the same node as the one it encountered when moving from Cp_1 to Cp_2; This is error prone because of the accumulated error during dead-reckoning navigation.

MORSE-BASED CELLULAR DECOMPOSITION COMBINED WITH VORONOI-DIAGRAM

The above described on-line Morse-based BCD handles unknown vast environments pretty well. It does so by making use of it sensors. Most work that describe BCD either assume the detector range of the sensor is infinite in size or the same size as the robot. Acar, Choset, and Atkar [18] shows how a detector range of an sensor can be utilized which is $r < \delta_s < \infty$ here r and δ_s.

Morse-based cellular decomposition combined with Generalized Voronoi Diagram (GVD) describe how this can be exploited to find an optimum coverage pad for a space which consists of VAST-cell and NARROW-cell. Such that the robot handles both vast and cluttered regions well. See figure 3.20, for such an environment.

The robot has two modus operandi that perform coverage of an unknown space, consisting of vast and cluttered regions. In a vast open space the robot scans an unknown environment for critical points as described in sub section 3.4.1. In such an environment it uses a zig-zag motion with an offset of $2\delta_s$. it's important to note that the coverage of a suction head from a dredge bot will in all likelihood be less than $2\delta_s$. During coverage of this VAST-cell it will construct an adjacency graph from every critical point. Once it encounters a cusp point, it builds a GVD with corresponding nodes. Such a point is an indication of the presence of a NARROW-cell.

This newly placed node on the GVD represents a new NARROW-cell with a width less the $2\delta_s$. It continues to traverse in the NARROW-cell till it encounters additional cusp point, constructing a GVD. During this stage the sensor can be seen as a sensor with ∞ range. Once it tracks the wall of the second VAST-cell it knows it's in a vast environment, since no second boundary is detected and slips into its initial modus.

Figure 3.20 depicts the stages of the incremental construction of the hierarchical decomposition while the robot is covering the space. The graphs depicted in the gray ellipses depict the VAST-cells that contain VAST-subcells represented as solid edges. Each VAST-subcell has to associated critical points represented as black dots. NARROW-cells is represented by the white ellipse and it contains the NARROW-subcells depicted as dashed edges. Hollow dots correspond to cusp points and gray dots represent the meet points. The double arrows show the links between NARROW-cells and their neighboring VAST-cells.

3.4.2 LANDMARK-BASED TOPOLOGICAL COVERAGE

The above described Morse-based algorithms create cell boundaries based on the detection of critical points, these points are detected via side faced range sensors, with the use of wall following. Morse-based algorithms cannot handle rectilinear environments, due to the fact that critical points are degenerate in this environment. Landmark-based topological coverage algorithm also use the BCD, but cell boundaries are determined by using topological landmarks.

Topological maps are robust against sensor and odometry errors because only a global topological consistency, rather than a metric one, needs to be maintained. Thrun [11] states that this type of map does not require accurate determination of the robot's position. Although this low resolution is also the reason why it's difficult to use them for coverage path planning. A node in a topological map is a landmark and does not correspond to a precise position or area in space. This makes it difficult to mark covered regions [33].

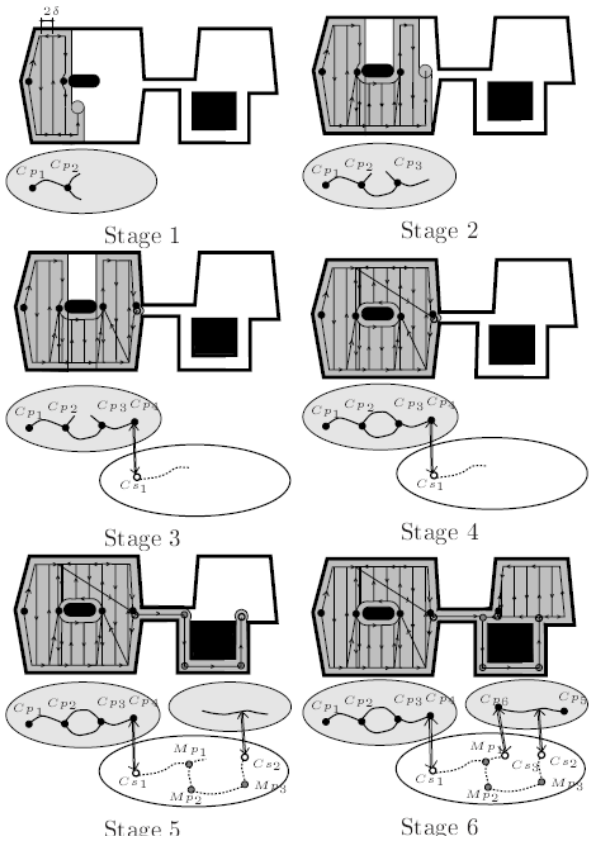


FIGURE 3.20: STAGES OF INCREMENTAL CONSTRUCTION [18]

SLICE DECOMPOSITION

Slice decomposition makes use of simpler landmarks. Galceran and Carreras [63] states that it can handle a large variety of environments including ones with polygonal, elliptical and rectilinear obstacles. Moreover obstacles can be detected from all sides of the robot, allowing a simpler zig-zag pattern without retracting to be used. As a result the generated coverage path is shorter.

Slice decomposition determines cell boundaries when it sees a abrupt change in the topology between segments in consecutive slices, each slice is a sensor sweep line where the $\delta_s x$ is moved to the next time step. Wong and MacDonald states that there are two situations where the abrupt changes occurs:

1. A segment in the previous slice is split by the emergence of a new segment, see figure 3.21
2. (a) and (b).
3. A segment from the previous slice disappears in the current slice, see figure 3.21 (c) and (d).

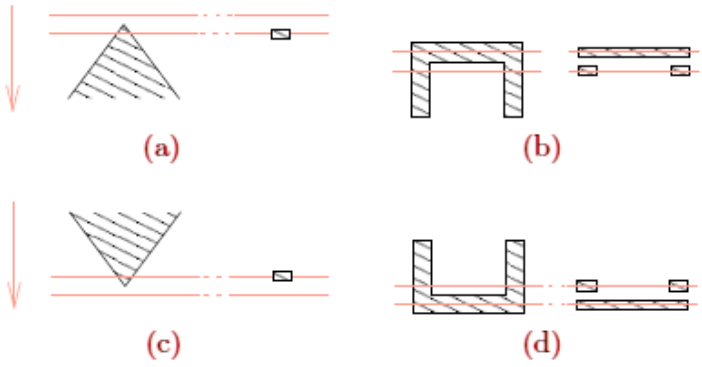


FIGURE 3.21: SPLITTING/MERGING OF SEGMENTS[27]

LISTING 3.1: OFF-LINE SLICE DECOMPOSITION

```
1:  procedure SliceDecomposition
2:      c ∈ {FreeSpaceCell, ObstacleCell}
3:      for all time t do
4:          Move sweep line downwards by Δx
5:          D_{l,t-1} = (..., c_{i-2}, c_{i-1}, c_i, c_{i+1}, c_{i+2}, ...)
6:          for all segments in D_{l,t-1} do
7:              if emergence inside c_i then
8:                  (c_i) ← (c_{e-1}, c_e, c_{e+1})
9:                  D_{l,t} = (..., c_{i-2}, c_{i-1}, c_{e-1}, c_e, c_{e+1}, c_{i+1}, c_{i+2}, ...)
10:             end if
11:             if c_i disappears then
12:                 (c_{i-1}, c_i, c_{i+1}) ← (c_d)
13:                 D_{l,t} = (..., c_{i-2}, c_d, c_{i+2}, ...)
14:             end if
15:         end for
16:     end for
17: end procedure
```

The slice decomposition is a formed by maintaining a list $D_{l,t}$, which consist of active obstacles and free space cells. This list is created via algorithm 3.1. This algorithm consist of two loops. The first one moves the sweep line, while the second one inspects segments and acts if there is a change in situation. At this time it updates the list $D_{l,t}$ marking it landmark. This algorithm does not take into account "*line of sight*". The author of this paper states that disappearance of segment can only be measured from hindsight. Thus backtracking will still be an issue.

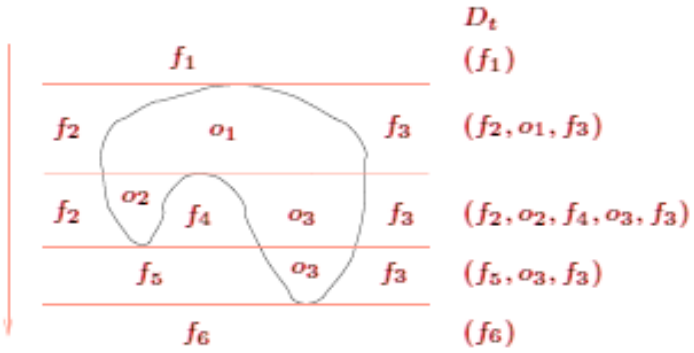


FIGURE 3.22: SLICE DECOMPOSITION GENERATED BY ALGORITHM 3.1 [27]

Wong [33] recognizes the limitations of slice decomposition and proposes a new method "*slice decomposition II*". This is because a robot cannot move inside obstacles, which means that the sweep line is limited to the cell that the robot is in [33]. There are five events that occur during slice decomposition II, as are depicted in figure 3.23. Wong proposes the following events to be used during the on-line algorithm 3.2. If the robot is tethered, for instance with an umbilical, not every cell can be be reached. The restrictions created by this tether can be view as a change of the boundary of the environment.

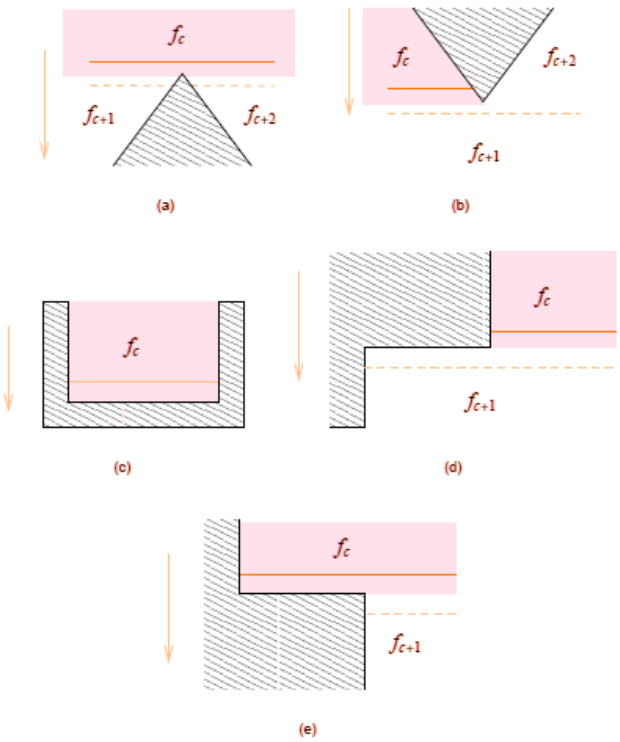


FIGURE 3.23: (a) SPLIT, (b) MERGE, (c) END, (d) LENGTHEN, (e) SHORTEN

ACTION	DESCRIPTION
SPLIT	Free space segment in the previous slice is split into two by the emergence of an obstacle. This is equivalent to obstacle segment emergence in normal Slice Decomposition.
MERGE	Free space segment in the current slice neighbors free spaces other than the free space segment in the previous slice in the direction of the previous slice. This is equivalent to obstacle segment disappearance in normal slice decomposition.
END	The previous free space segment is the final one in the current cell. This is equivalent to free space segment disappearance in the normal version.
LENGTHEN	Free space segment in the current slice neighbors an obstacle segment in addition to the free space segment in the previous slice in the direction of the previous slice. Another way to view this situation is that the current slice is much longer than the previous slice.
SHORTEN	Free space segment in the previous slice neighbors and obstacle segment in addition to the free space segment in the current slice in the direction of the current slice. Another way to view this situation is that the current slice is much shorter than the previous slice.

LISTING 3.2: OFF-LINE SLICE DECOMPOSITION II

```
1:  procedure SliceDecompositionII
2:      O ← inital cell
3:      F ← ∅
4:      while O ≠ ∅ do
5:          f_c ← f ∈ O
6:          move to on (of two) cell boundary of f_c
7:          repeat
8:              move sweep line by Δx towards the opposite cell boundary
9:              if event occur then
10:                 F ← F + f_c
11:                 O ← O − f_c
12:                 if event = split or merge then
13:                     O ← O + f_{c+1}, f_{c+2} if f_{c+1}, f_{c+2} ∉ (O ∪ F)
14:                 end if
15:                 if event = lengthen or shorten then
16:                     O ← O + f_{c+1} if f_{c+1} ∉ (O ∪ F)
17:                 end if
18:             end if
19:         until event occur
20:     end while
21: end procedure
```

ON-LINE TOPOLOGICAL COVERAGE ALGORITHM

By using slice decomposition II, which is described in Section 3.4.2, with a topological map. A crawler can construct it on-line. It now allows in to perform its task in a unknown environment. The topological map embeds the slice decomposition of an environment by using events as nodes. The map is updated whenever relevant information becomes available. The path planner generates a new path un each update, based on the new partial topological map [33].

This topological map is represented as a planar graph, where the nodes represent landmarks (i.e., split, merge, end, lengthen or shorten, such as depicted in figure 3.23) and edges indicate the types of motion required to travel between nodes they are incident upon. For example, whether the edge is next to a wall and which side the wall is on. They also store estimated distances separating the two nodes they connect [58].

This topological coverage algorithm makes use of a state transition diagram 3.24. In this diagram three states are described, each with corresponding algorithm (3.3, 3.4 and 3.5). The crawler is assumed to be placed in a corner near a wall, therefore the boundary state is considered as entry point.

The crawler starts in the *boundary* state by executing algorithm 3.3. During this state the crawler performs a wall following algorithm. When it finds a landmark or it arrives at the end of a strip, it updates graph $\mathbf{G}$. When it's at the end of a strip and the boundary is fully explored it gets in to the state travel, described in algorithm 3.5, otherwise it turns around and continues in the boundary state.

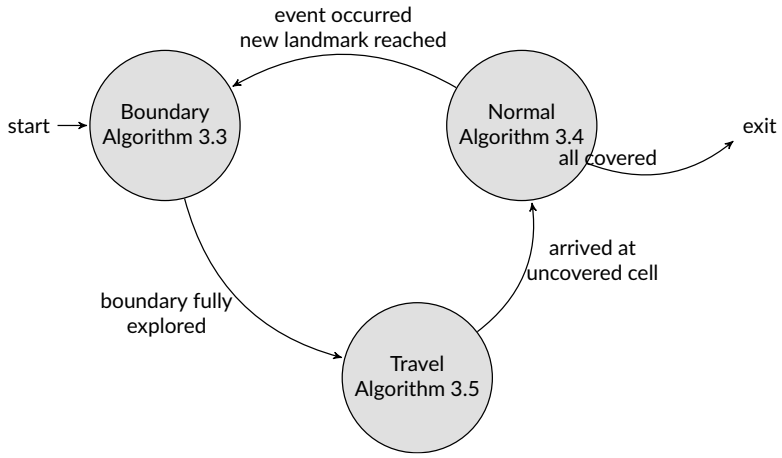


FIGURE 3.24: STATE TRANSITIONS FOR TOPOLOGICAL COVERAGE ALGORITHM

LISTING 3.3: BOUNDARY STATE

```
1:  procedure BoundaryState
2:      loop
3:          move forward along boundary
4:          if at landmark then
5:              update G
6:          end if
7:          if arrive at end of strip then
8:              update G
9:              if boundary fully explored then
10:                 state ⇐ travel
11:             else
12:                 turn around 180°
13:             end if
14:         end if
15:     end loop
16: end procedure
```

Once in the *travel* state, the path is generated that moves the crawler from one cell to another, it does so by implementing line 5 and 6 from algorithm 3.2 described at page 34. Algorithm 3.5 is executed in this state. Once it arrives at an cell its operation state becomes *normal*. A normal boustrophedon zig-zag movement is followed in this state.

LISTING 3.4: NORMAL STATE

```
1:  procedure NormalState
2:      repeat
3:          follow zigzag pattern
4:      until at landmark
5:      update G
6:      state ⇐ boundary
7:  end procedure
```

LISTING 3.5: TRAVEL STATE

```
1:  procedure TravelState
2:      T(n) ⇐ search G
3:      if T(n) = ∅ then
4:          exit algorithm
5:      end if
6:      while T(n) ≠ ∅ do
7:          move towards T(0)
8:          if at T(0) then T(n) ⇐ T(n) – T(0)
9:          end if
10:     end while
11:     state ⇐ normal
12: end procedure
```

LANDMARK RECOGNITION USING NEURAL NETWORKS

Wong [33] proposes a novel idea to classify landmarks using Neural Networks. These are classification algorithms which approximates the operations of a human brain. Wong build a test robot with a 360°rotatable single-beam sonar. Each scan consists of 48 individual sonar-beams taken over a range of 360°. This vector is made independent of orientation, by virtually rotating it so that index 0 would always point towards the direction where the sonar range measured the shortest distance, as depicted in figure 3.25.

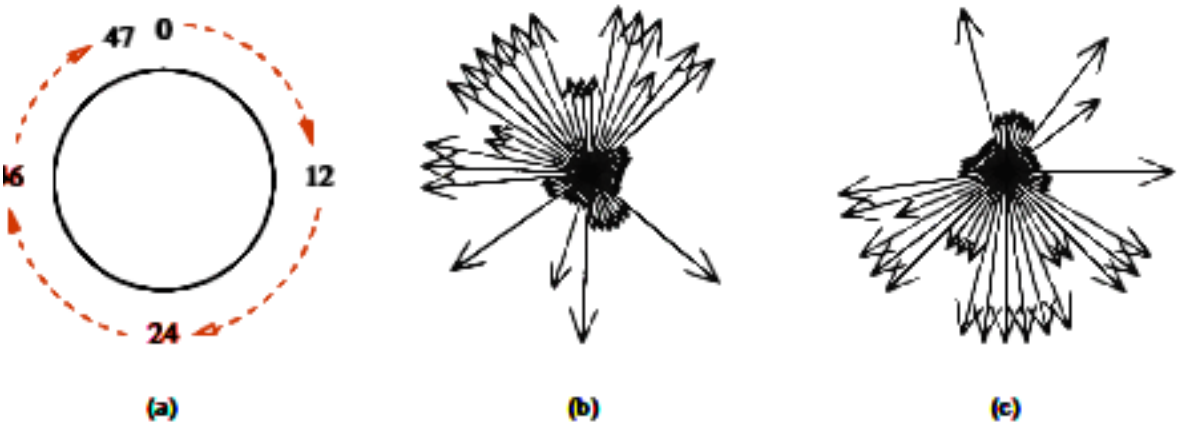


FIGURE 3.25: ROTATION OF SONAR READING TO MOST OCCUPIED DIRECTION [33]

This vector is fed into a Multi-Layer Perceptron (MLP) which distinguishes three different type of classes: free space nodes, obstacle nodes and everything else. This neural network first has to be taught. This is done under supervision, meaning that the landmark type have to be predefined.

3.4.3 GRID-BASED METHODS

Grid-based methods divide the working area into a raster of uniform grid cells. Each cell has an associated value stating whether an obstacles is present or if it's rather free space. The value can either be binary or a probability [70]. These grid cells are typically square in shape, but it's not uncommon to use triangle shaped cell. The size of the cell usually corresponds with the size of a crawler.

Once the environment is mapped onto an uniform grid. An optimal coverage path can be found using the by Choset and Pignon [9] proposed method, whom uses a conventional wave-front algorithm (distance transform) to determine a coverage path. First a start and goal cell has to be assigned. The wave-front algorithm initially assigns a 0 to the goal and then a 1 to all surrounding [red. reachable]

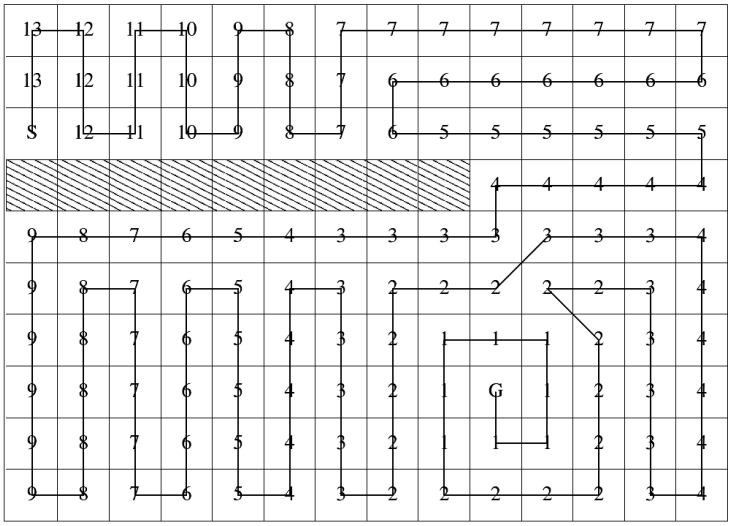


FIGURE 3.26: COVERAGE PATH GENERATED FROM A DISTANCE TRANSFORM [33]

cells. Then all the unmarked cells neighboring the marked 1 are then labeled with a 2. This process repeats until the wave-front crosses the start. Once this occurs, the robot can use gradient descent on this numeric potential function to find a path [19]. This results of algorithm 3.6 is shown in figure 3.26.

LISTING 3.6: OFF-LINE GRID-BASED COMPLETE COVERAGE

```
1:  procedure GridBasedCompleteCoverage
2:      Set start cell to current cell
3:      Set all cells to NOT visited
4:      loop
5:          Find unvisited neighboring cell with highest value
6:          if NO unvisited neighboring cell found then
7:              Mark current cell as visited
8:              Exit procedure                          ▷ Goal reached
9:          end if
10:         if unvisited neighboring cell value ≤ current cell value then
11:             Mark current cell as visited
12:             Exit procdedure                         ▷ Goal reached
13:         end if
14:         Set current cell to neighboring cell        ▷ Move to next cell
15:     end loop
16: end procedure
```

The author states an optimal placement of the start and goal cell is paramount. The algorithm 3.6 does not take into account a deadlock state. Which is illustrated in figure 3.27. In this figure goal and start cell are arbitrarily placed. Execution of the algorithm ensures that the dredge bot is stuck at the farthest cell. Where it remains, if no backtracking over previous visited cells is allowed.

GRID-BASED COVERAGE USING SPANNING TREES

An sub-family of grid-based coverage are systematic spiral paths algorithms. These work by following a systematic spiral spanning tree of a partial grid map. This map is constructed using its on-board sensors [70] and uses two different sizes of grid cells. The smaller grid cell is the same size as the robot. Four of these grid cells then form a mega cell. [33]. The Spiral Spanning Tree Coverage (Spiral-STC) described in algorithm 3.7, works as follows:

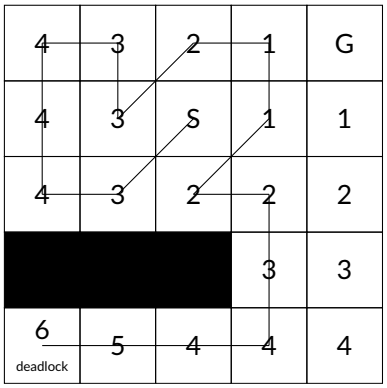


FIGURE 3.27: DEADLOCK STATE DUE TO INCORRECT STARTING POINT

Starting at the current cell, the robot chooses a new travel direction by selecting the first new mega cell in the free space in anti-clockwise direction. Then, a new spanning-tree edge is grown from the current mega cell to the new one. The algorithm is called recursively. The recursion stops only when the current cell has no new neighbors. A mega cell is considered old if at least one of its four smaller cells is covered. As a result of this recursion, the robot moves along one side of the spanning tree until it reaches the end of the tree. At this point, the robot turns around to traverse the other side of the tree [70].

LISTING 3.7: SPIRAL SPANNING TREE COVERAGE

```
1: procedure SpiralSpanningTreeCoverage( w, x )
2:     Mark the current cell x as old
3:     while x has new obstacle-free-4-neighbour cell do
4:         Scan for the first new neighbour of x in anti-clockwise order, starting with the
   parent cell w Call this neighbour y
5:         Construct a spanning-tree edge from x to y.
6:         Move to a subcell of y by following the right-side of the spanning tree edges
7:         Execute SpiralSpanningTreeCoverage(x, y).
8:     end while
9:     if x ≠ startcell then
10:        Move back from x to a subcell of w along the right-side of the spanning tree edges.
11:    end if
12: end procedure
```

Wong [33] and Lee, Baek, Choi, *et al.* [52] states the path can be optimized by using smaller grid cells, however because accurate maneuverability is often an issue, a higher resolution is not the best approached. Lee, Baek, Choi, *et al.* proposes a new method for Spiral-STC by limiting the number of turns, decelerations and accelerations. Mei, Lu, Hu, *et al.* [26] determined by analytically comparing energy efficiency of different coverage algorithm that sharp turns bring about inefficiency. Thus by limiting these, an energy efficient path can be generated. Since the dredge bot will be powered by an external land-based source, this option won't be further explored.

NEURAL NETWORK-BASED COVERAGE ON GRID MAPS

Luo, Yang, Stacey, *et al.* [23] proposes a model capable of planning a real-time path; Which covers every area, in the vicinity of obstacles, to a reasonably extent. A crawlers path is autonomously generated through a dynamic neural activity landscape [23][41]. Luo, Yang, Stacey, *et al.* discretized a 2D space in a grid map where the diagonal length of each grid cell is equal to the robot sweeping radius and then a neuron is associated to each and every grid cell. Each neuron has connections to its immediate 8 neighbors [63]. This architecture is illustrated in figure 3.29.

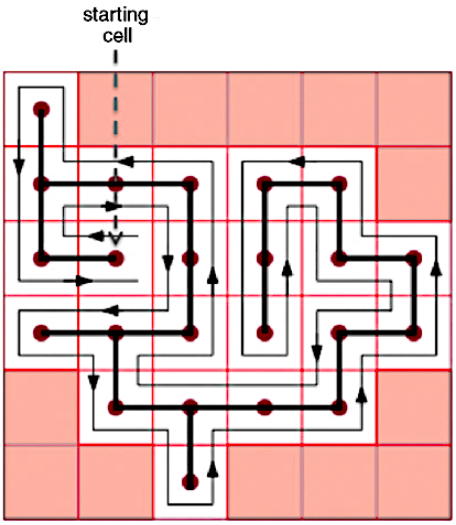


FIGURE 3.28: CPP SPIRAL-STC ALGORITHM [63].

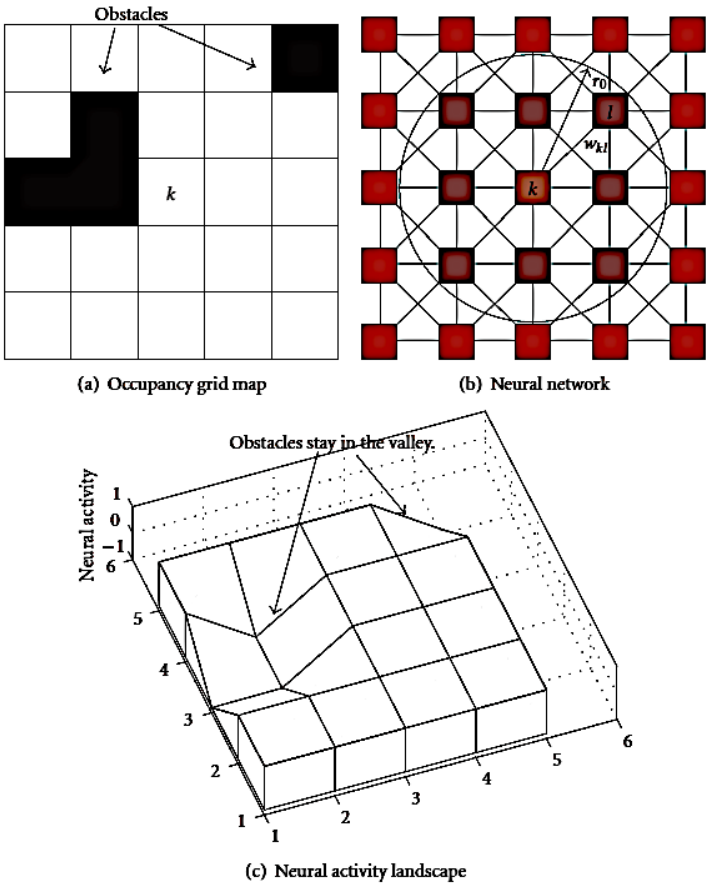


FIGURE 3.29: ARCHITECTURE OF A NEURODYNAMICS MODEL [61]

The proposed neural network, is topologically expressed on a 2-dimensional occupancy grid map. The location of the kth neuron of the neural network represent a location (cell). Where each neuron has local lateral connections to its neighboring neurons, in the small region $[0, r_0]$; Where r_0 is the receptive field radius of the kth neuron, as shown in figure 3.29 (b). The excitatory input results from uncovered area and lateral neural connections, while inhibitory inputs, results from obstacles [61]. The shunting equation 3.40 derived from Hodgkin and Huxley [1] determines the dynamics of each neuron in the network.

$$\frac{\mathrm{d}x_k}{\mathrm{d}t} = -Ax_k + (B - x_k)\left([I_k]^+ + \sum_{l=1}^{m} w_{kl}[n_{a,l}]^+\right) - (D + x_i)[I_i]^- \tag{3.40}$$

Equation 3.40 consist of the following terms: x_k is the kth neuron in the neural network, while A, B and D are non negative constants describing the passive decay rate, and the upper and lower bounds of the neural activity. The terms $[I_k]^+ + \sum_{l=1}^{m} w_{kl}[n_{a,l}]^+$ are the excitatory inputs while $[I_i]^-$ is the inhibitor. These are linear-above and below thresholds defined as $[a]^+ = \max\{a, 0\}$ and $[a]^- = \max\{-a, 0\}$. The connection weight is given by w_{kl} which is assigned between the kth and the lth neuron, which is given by $w_{kl} = f(|q_k - q_l|)$, where $|q_k - q_l|$ is the Euclidean distance between vectors q_k and q_l in the state space, and $f(d)$is a monotonically decreasing function defined as

$$f(d) = \begin{cases} \frac{\mu}{d}, 0 \leq d < r_0 \\ 0, d \geq r_0 \end{cases} \tag{3.41}$$

Where μ and r_0 are positive constants, The external input I_k to the kth neuron is defined in equation 3.42. In this equation E is a large constant.

$$I_k = \begin{cases} E, \text{ if it's an uncovered area} \\ -E, \text{ if it's an obstacle area} \\ 0, \text{ if it's a covered area} \end{cases} \tag{3.42}$$

By properly defining the external inputs from the changing environment and internal neural connections, the unclean areas and obstacles are guaranteed to stay at the peak and the valley of the activity landscape of the neural network, respectively. The unclean areas globally attract the robot in the whole workspace through neural activity propagation, while the obstacles have only local effect in a small region to avoid collisions. The collision-free robot motion is planned in real time based on the dynamic activity landscape of the neural network and the previous robot position, such that all areas will be cleaned and the robot will travel along a smooth zigzag path [41]. An advantage of this method is that can handle non stationary environments (i.e., dynamically changing obstacles) [63].

Yan, Zhu, and Yang further states that if there are power and time limitations, the crawler should travel the path with the least revisited areas and the least turns of moving directions. Therefore, for a given current crawler location at time k, denoted by $\vec{p}_k$, the next crawler location at time $k + 1$, $\vec{p}_{k+1}$ is obtained by equation 3.43. Where c is a positive constant and m_n is the number of neighbouring neurons of the $\vec{p}_k$ neuron, that is all the possible locations of the current location $\vec{p}_k$; variable $n_{a,l}$ is the neural activity of the lth neuron, which is the same as in equation 3.40; y_l is a monotonically increasing function of the difference between the next crawler moving directions, defined in equation 3.44.

$$\vec{p}_{k+1} \Longleftarrow x_{\vec{p}_k} = \max\{n_{a,l} + c\, y_l, l = 1, 2..., m\} \tag{3.43}$$

Where $\Delta\Psi_l \in [0, \pi]$ is the turning angle between the current moving direction and the next moving direction. Here $\Delta\Psi_l = 0$ is straight ahead, while $\Delta\Psi_l = \pi$ stands for a backwards movement [61]. This would indicated that the crawler only has the ability to turn either left or right depending on the convention place on $\Delta\Psi_l$. By defining it as $\Delta\Psi_l \in [-\pi, \pi]$, the crawler has a full turning range. Thus $\Delta\Psi_l = \Psi_l - \Psi_c = |a\tan 2(y_{p_l} - y_{p_c}, x_{p_l} - x_{p_c}) - a\tan 2(y_{p_c} - y_{p_p}, x_{p_c} - x_{p_p})|$

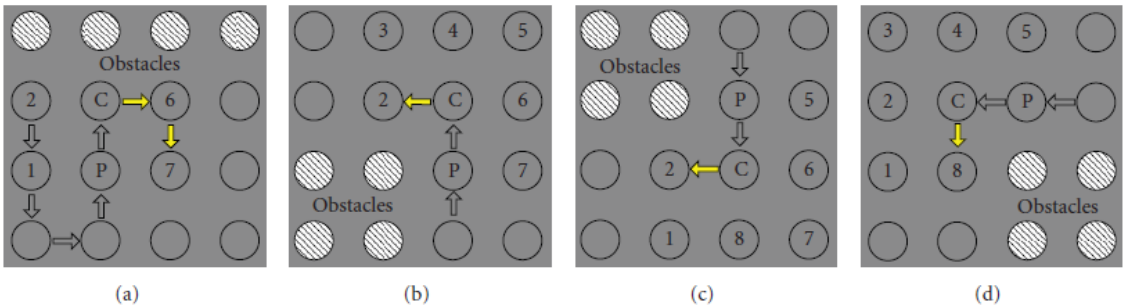


FIGURE 3.30: Four predefined templates. Source: Yan, Zhu, and Yang [61]

$$y_l = 1 - \frac{\Delta\Psi_l}{\pi} \tag{3.44}$$

According to Yan, Zhu, and Yang [61] path planning in the vicinity of some unstructured obstacles may cause overlap, resulting in additional turns, which translates in a higher energy consumption and additional time [52][75][26]. They propose the use of four predefined tables as shown in figure 3.30. These predefined templates are effective to deal with the vicinity situation of unstructured obstacles and enable the crawler to plan a more reasonable path with less overlapping areas.

Grid based neural networks methods lend themselves to be easily integrated with map building. Yan, Zhu, and Yang [61] describes how the sensor model can be used to build an on-line map of the environment where the Dempster Shafer theory is used to filter out uncertainties of the sensor.

4 CHAPTER CONTROLLER DESIGN

A controller that is capable of making complex decisions in a changing environment, requires different levels of abstraction. At the highest level it should be able to form a plan based upon a predefined strategy. A dredging crawler should be able cover a certain area removing a specified amount of soil, until reaching the specified depth. Because the geometry of the environment in which it operates, is unknown from the beginning, it should be able to adjust and change its operating parameters accordingly. Section 4.1 will assess the researched CPP algorithms and decide which one is best suited for the crawler. In order to adopt and change its behavior to best suit the surroundings surrounding it needs to be aware of it and have means of manipulation at its disposal. Section 4.2 will discuss these required peripherals determine the required types.

Discussion on a accurate model for a Kalman filter can takes place in Section 4.3. This also covers a discussion on the state representation for a crawler, the motion model and crawler specific models, such as soil dynamics, dredging and steering. Section 4.4 describes how a controller framework for autonomous maritime vessels should behave, discussing the design patterns.

4.1 STRATEGY DECISION

Researching different CPP algorithms in Section 3.3 showed that a divide can be seen between on-line and off-line algorithms. The latter of these are deemed unsuitable, due to a changing environment. This reduces viable candidate algorithms to the following short-list:

- online Morse-based Boustrophedon Cellular Decomposition (onBCD)
- Morse-based Cellular Decomposition with Voronoi Diagram (BCD-GVD)
- topological Boustrophedeon Cellular Decomposition (topBCD)
- on-line Slice Decomposition II (onSD2)
- on-line Slice Decomposition II with Neural Network (onSD2NN)
- grid based Spanning Tree Coverage (STC)
- Neural Network based Coverage in Grid maps (NN)

The options presented above are judged against different criteria of assorted relevance, refelected in a weight factor. Ranging from 1 nice to have, to 3 really fucking useful. Each algorithm is awarded points for each criteria, which range between 1 and 5, the higher the better it fulfils that criteria. Quatifying the axact performance is impossible at this stage. This introduces a level of uncertainty, which is taken into account in the decision. The total sum for each algorithm is then calculated with the propagation of those uncertainties. The table below shows that all three BCD algorithm easily outweigh the others for our application. Were BCD-GVD, has the highest score but it brings with it a greater uncertainty compared against topBCD. it's good practice in agile developement, to "make it work first" and "refine it later". By choosing topBCD as the best strategy decision the implied risk due to uncertainties is reduced.

	weight	onBCD	BCD-GVD	topBCD	onSD2	onSD2NN	STC	SSTC	NN
online	3	5 ± 0	5 ± 0	5 ± 0	5 ± 0	5 ± 0	5 ± 0	5 ± 0	5 ± 0
time	2	5 ± 2	5 ± 2	5 ± 1	5 ± 2.5	1 ± 2	3 ± 2	3 ± 2	2 ± 1
energy eff.	1	4 ± 1	4 ± 1	4 ± 0.5	5 ± 1	1 ± 0.5	3 ± 1	3 ± 1	2 ± 2
revisits	3	1 ± 0.2	1 ± 0.2	1 ± 0.2	1 ± 0.2	1 ± 0.2	1 ± 0.2	1 ± 0.2	1 ± 0.2
complexity	3	5 ± 1	5 ± 1	4 ± 0.5	1 ± 1	1 ± 2	1 ± 1	1 ± 2	2 ± 0.5
adaptivity	1	2 ± 1	3 ± 1	3 ± 1	5 ± 0.5	5 ± 0.5	3 ± 2	3 ± 2	4 ± 1
TRL	3	5 ± 1	5 ± 1	5 ± 0.5	2 ± 2	2 ± 2	3 ± 1	3 ± 1	1 ± 3
maintainability	2	5 ± 2	5 ± 2	4 ± 1	3 ± 1	3 ± 1	2 ± 1	2 ± 1	2 ± 2
integration	1	4 ± 0.5	4 ± 0.5	4 ± 0.5	2 ± 1	2 ± 1	4 ± 2	4 ± 2	4 ± 2
legislation	1	3 ± 2	3 ± 2	3 ± 2	2 ± 3	2 ± 3	3 ± 2	3 ± 2	2 ± 2
environment	1	3 ± 1	3 ± 1	3 ± 1	2 ± 1	2 ± 1	3 ± 1	3 ± 1	2 ± 1
Σ		**84.0 ±7.6**	**85.0 ±7.6**	**80.0 ±4.4**	**59.0 ±9.3**	**47.0 ±10.2**	**56.0 ±7.2**	**56.0 ±8.9**	**49.0 ±10.8**

4.2 PERIPHERALS

The crawler is comprised of multiple components which are listed in Appendix A. This section will focus on components which are required for autonomous operations. The two drivetrains consists of an electric motor, with an attached gearbox, coupled to a hydraulic motor and a hydraulic system which drives a hydraulic pump. The shaft of the hydraulic motor is attached to a flexible coupling and connected with an Archimedes screw by a rigid flange connection. This drive-train is discussed in detail in Section 4.3.2. The crawler is able to turn if both drivetrains operate at different velocities, which is discussed in Section 4.3.5. The control vector $\vec{u}_k$, consist of two velocity components.

Dead-reckoning is often accomplished with a accelerometer, gyroscope and magnetometer, this sensor array is often expanded with a pressure sensor. These sensors combined will give a good impression of the orientation and position change over time. Keeping in mind that the measurement uncertainty of the pressure sensor will increase during dredging operations. This phenomena is described in Section 3.2.1. Knowing actual rotational velocity of the Archimedes screw will be used to better predict the traveled distance. Something that can be measure with an encoder attached on the shaft of the electro motor. These encoder are also used for the regular Proportional Integral Derivative (PID) controller of the screws. Although we can measure the rotational velocity of the screws we can't know for sure how fast the crawler will actually move. Section 4.3.3 describes how the change in required torque of the drive-train will give a indication of the slippage of the Archimedes screws.

4.3 KALMAN FILTER DESIGN

In Chapter 3.4 it became evident that an accurate estimation of a crawler's position and heading is needed so that it can perform its tasks using CPP algorithms. Because it's not possible to use GPS in an underwater environment, due to the dampening of electric and magnetic fields in water (which was described in Section 3.1.2), an alternative localization method had to be found. Section 3.3 describes the Kalman filter as the industry de-facto. Position estimation for the control will therefore be performed with this algorithm.

A design for an Kalman filter is proposed, using a common array of sensors: accelerometer, gyroscope, magnetometer and a pressure sensor. The workings for each of these are discussed in Section 3.2 and the component selection was made in Section 4.1. The crawler is actuated by changing the rotational speed of the individual Archimedes screws.

The speed of an Archimedes screw driven crawler is a direct function of the pitch of its vanes but this only applies if there is no horizontal soil failure under the screws. Such a phenomena is called slip. It consists of a period in time where the screws turn but generate no forwarding force leading to an inaccurate estimation for the new predicted state. Due to the geometry of an Archimedes screw, this slip coexists with a bulldozer effect created by the vanes acting as a shovel in the soil. This will lead to an

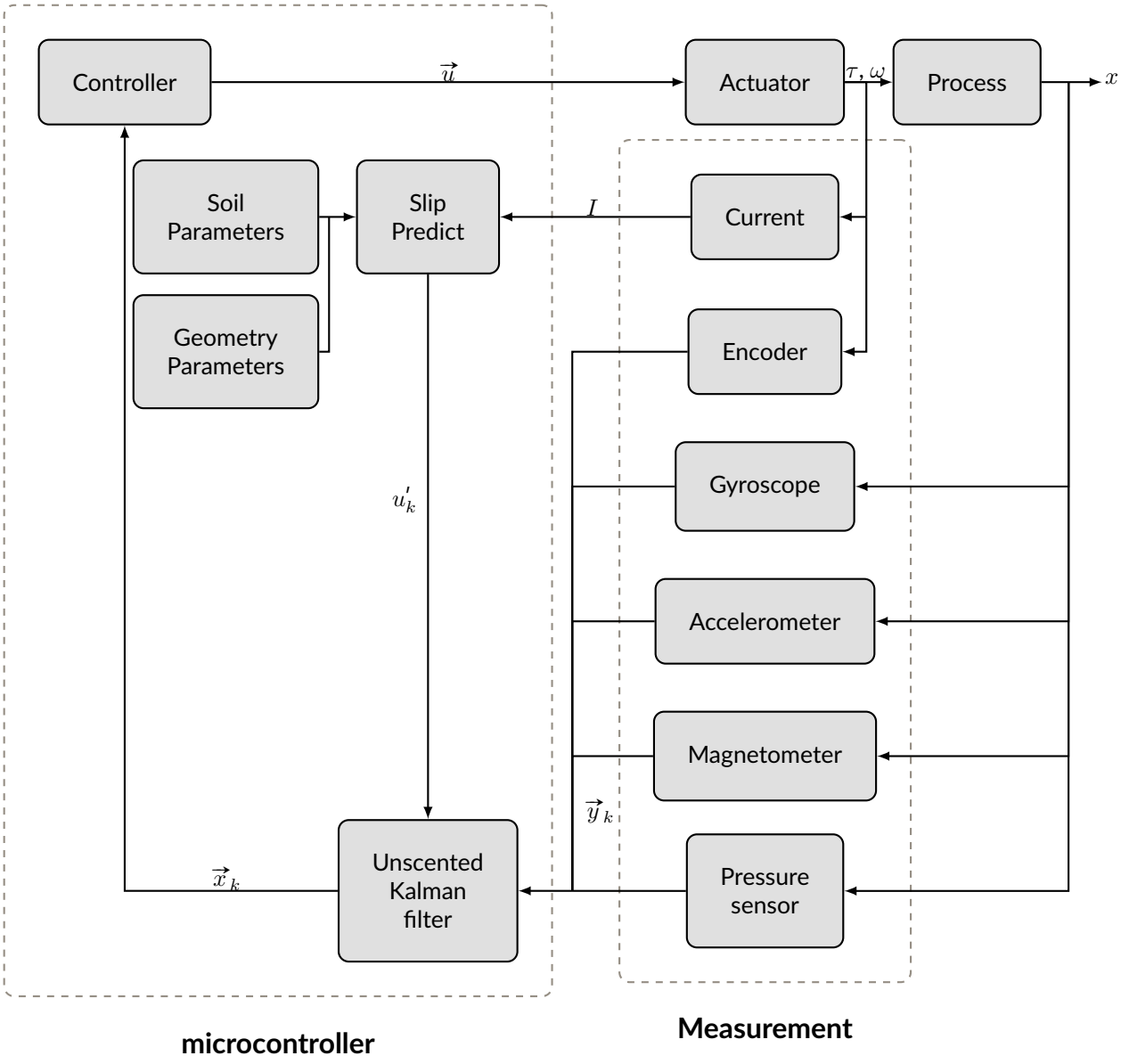


FIGURE 4.1: PROPOSED KALMAN FILTER

increase in torque. it's proposed that by measuring the required torque of the drive-train, a prediction can be made of how much slip has occurred, leading to a better estimation of the future state-vector $\vec{x}_{k+1}$. This behavior and the mathematical model will be discussed in more detail in Section 4.3.2. Figure 4.1 shows the interaction and connectivity of the various components that serve as the input and output of the proposed filter.

Since the physical processes for movement of a crawler are non-linear and the "basic" Kalman filter, described in Section 3.3.2, is limited to linear assumptions, the proposed Kalman filter is of the unscented type. The Unscented Kalman Filter (UKF) uses a deterministic sampling technique known as Unscented Transformation (UT). This technique picks a minimal set of sample points, also known as sigma points, around the mean. The sigma points are then propagated through the non-linear function, from which a new mean and covariance estimate are then formed.

4.3.1 STATE REPRESENTATION

Bahr, Leonard, and Fallon [43] state that, the most generic case of a vehicle operating in 3D Euler-space, such as a crawler, consists of a vector of variables comprised of a vehicle's pose and orientation. The pose is its position in a (global) reference frame $[x\ y\ z]^T$ while its orientation is given in Euler angles

$[\phi_c\ \psi_c\ \theta_c]^T$. The pose vector at time t is then $\vec{x}_k = [x\ y\ z\ \phi_c\ \psi_c \theta_c]^T$, which will be denoted as $\vec{x}_k$ for the remainder of this paper. Beside the pose, the state vector can also contain the first and second derivatives of the pose vector.

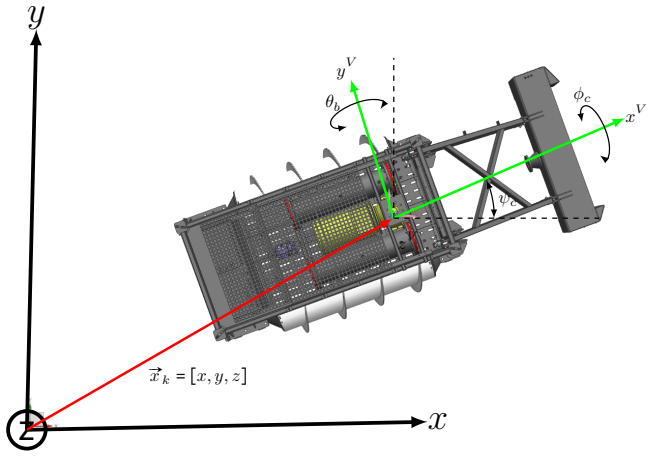


FIGURE 4.2: STATE REPRESENTATION

2D OR 3D

Bahr, Leonard, and Fallon [43] propose the following simplification: all submersible vehicles are outfitted with a pressure sensor which allows them to determine their absolute depth with high accuracy and a high update rate. As a result all underwater navigation systems are only used to resolve the 2D position and all underwater vehicle related localization problems are stated in 2D. This allows for a simplified state vector $\vec{x}_k = [x\ y\ \phi_c\ \psi_c\ \theta_c]^T$.

This simplification does not hold for crawlers or AUV that change in depth in any other direction than collinear with the Earth's gravitational axis. Since this crawler moves over irregular terrain, any pitch or roll will result in a gravitational force that also consists of components along the x-axis and / or y-axis which would then be interpreted as movements in the 2D x-y space. This, coupled with an unreliable pressure-sensor reading due to disturbance of sediment from the sea bed during dredging operations as described in Section 3.2.1, means that the state representation should be in 3D Euler space, as is shown in figure 4.2.

QUATERNIONS

Because the UKF is executed on an embedded device with limited resources, all rotational translations are calculated with the help of quaternions. Which make use of the same concept as complex numbers, while complex numbers define a single imaginary axis perpendiculair to real number axis, quaternions define three imaginary axes perpenedicular to each other and the real numbers. Just as complex numbers are very useful in describing a rotation in a two-dimensional plane, quaternions are highly efficient in a three-dimensional space. They are also immune to gimbal locking. The exact workings are outside of the scope of this thesis, but for those readers who are interested, a good explanation is given by 3Blue1Brown [87] and can be found at `www.youtube.com/watch?v=d4EgbgTm0Bg`. A quaternion is represented as a single column matrix with four rows, consisting of a real valued magnitude and three imaginary components for the axes $\vec{q} = \left[q_s\ q_x\ q_y\ q_z\right]^T$ where the q_s is used to normalize the quaternion, this keeps the error from accumulating. it's important to note that quaternions are non-commutative, thus the order in which they are applied matters to the outcome of the rotation.

4.3.2 MOTION MODEL

it's important to evaluate the effects of control inputs $\vec{u}_k$ on the pose vector $\vec{p}_k = [x, y, z]^T$ such that $\vec{p}_{k+1}$ can be predicted. Since the continuous-time model for the vehicle state's speed and rate can be described as:

$$\vec{p}_{k+1} = f(\vec{x}_k, \vec{u}_k) \tag{4.1}$$

The function $f(\cdot)$ in equation 4.1 can be very complex and is usually non-linear, it has to take into account all external forces that can influence the movement of the crawler. According to Bahr, Leonard, and Fallon [43] the more complex the model, the more accurately it can represent the vehicle dynamics and provide a better prediction of the future pose. However, obtaining such a model requires detailed knowledge of the structure as well as the parameters listed below. All of which influence movement and sensor reading:

- Shape of the crawler[1][2]
- Size of the crawler[1][2]
- Weight of the crawler[1][2]
- Buoyancy of the crawler[1][2]
- Actuators (pumps, motors etc.) and their behavior, such as vibrations
- Forces exerted on the crawler due to fluid transportation through the floating line
- Forces exerted on the crawler due to the buoyancy of the floating line
- Forces exerted on the crawler from the umbilical
- Operating environment
  - Properties of the soil bed [45];[2]
    - Density[2]
    - Cohesion[2]
    - Skin friction[2]
  - Temperature
  - Salinity
  - Current
  - Medium through which it manoeuvres (mixture of water and soil)
- Configuration of the propulsion system:
  - Track variant[2] [45]
    - Length[1][2]
    - Distance between grouser plates[1][2]
    - Width of the track[1][2]
    - Height of the grouser plate[1][2]
  - Archimedes screw variant[2] [46]
    - Length[1][2]
    - Diameter[1][2]
    - Submerged Weight Range (SWR)[1][2]
    - Number of helices[1][2]
    - Pitch angle[1][2]
    - Vane height[1][2]
    - Front bulldozer angel[1][2]
    - slip[2]

[1] Constant parameter
[2] Used in the model

Not all parameters shown above have the same impact on the future state representation. Since calculations on a complex model require more processing power, only parameters with a substantial influence will be used. it's assumed that the biggest influences are drag-forces, due to movement in a fluidium and the interaction with the Archimedes screw in the silt layer. These will, in all probability have a bigger impact than other operating parameters. Forces exerted on the crawler due to a current, floating lines and actuators are — for now — deemed to be a magnitude smaller and can be ignored.

During travel between two points, it's preferred to minimize travel time. This is usually achieved by moving at the fastest sustainable speed. The propulsion system which acts on the submerged soil bed is governed by a lot of interactions on the system as a whole. These are listed in Section 4.3.2. The crawler is propelled by an Archimedes screw. A proven technology for land based vehicles, especially in rough, impassable terrain. They have shown reliable operations, superior traction capabilities and they can be used as a buoyancy body. Typical working territories for Archimedes driven vehicles are dredge deposit sites and swamps [53].

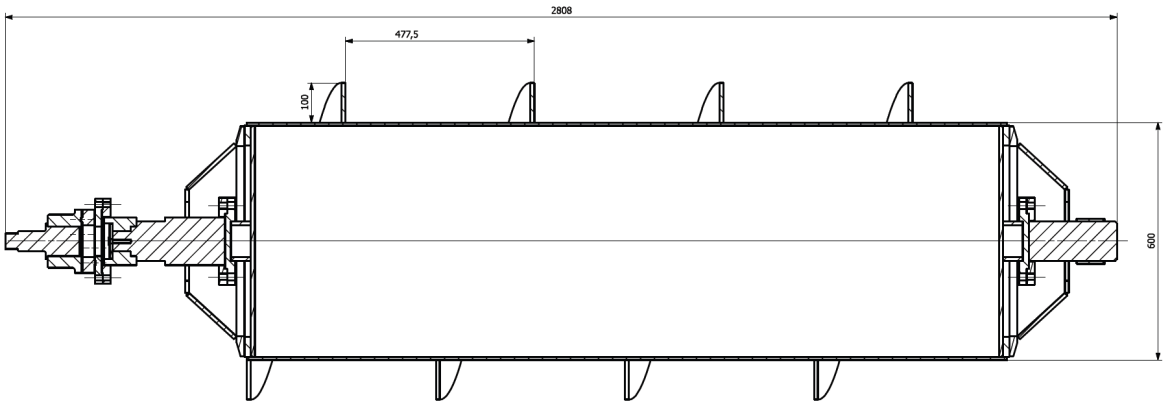


FIGURE 4.3: ARCHIMEDES DRIVE-TRAIN

PROPULSION SYSTEM MODEL

The propulsion system is modelled and shown in Figure 4.4. In this situation, an electrical motor drives a hydraulic pump. This electrical motor delivers a certain torque τ, where the hydraulic pump demands a certain torque. This torque difference $\Delta\tau$ divided by the sum of inertia's in the system and integrated over time translates to a certain rotational speed ω with which this system turns. This hydraulic pump generates a pressure p, which delivers energy to a hydraulic motor that has a pressure drop, or in other words, it converts this energy into a certain torque at a certain rotational speed. The pressure difference Δp, between the hydraulic pump and motor, multiplied with the cross section, in which the hydraulic oil flows, and divided with the total mass of hydraulic oil, integrated over time, results in a fluid velocity. This hydraulic motor drives the Archimedes screws, which interact with the soil and generate a force which translate to a velocity when total friction and resistance is overcome.

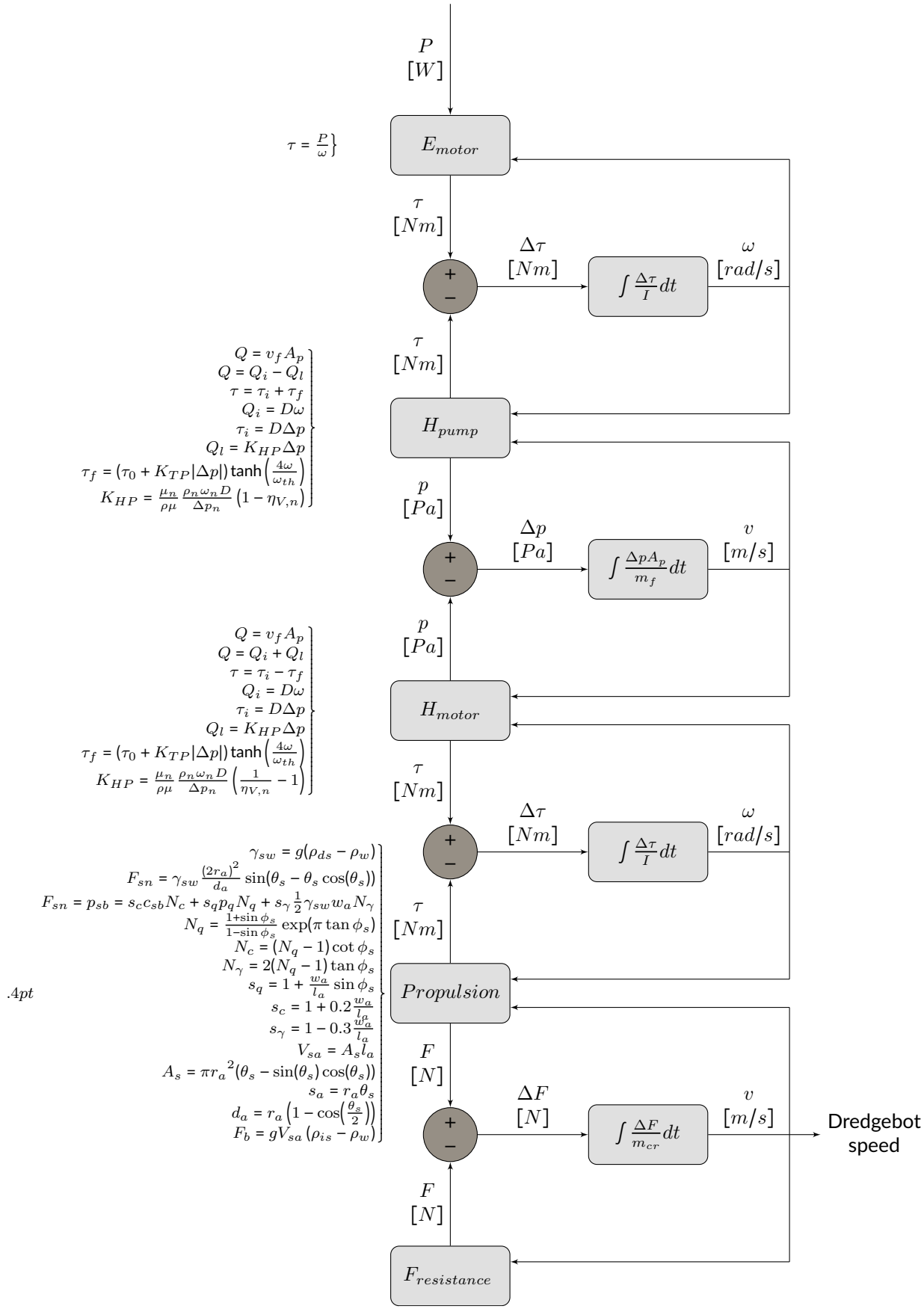


FIGURE 4.4: DRIVE-TRAIN BASED ON FIRST PRINCIPLE

A HYDRAULIC PUMP is used to convert mechanical- to hydraulic energy and is modelled after Mathworks [81], described model. In order to make a model based on first principle, the hydraulic pump should accept a variable fluid speed $v_f(\cdot)$ and angular velocity of the electro motor ω, whilst calculating a pressure gain Δp and needed torque τ. The flow rate which is generated at the pump is equal to equation 4.2. Where A_p is the cross-section of the pipe and v_f is the average speed of the fluid through that cross-section. Q is the net volumetric flow rate.

$$Q(v_f) = A_p v_f \tag{4.2}$$

The net volumetric flow rate obtained from equation 4.2 consists of an ideal flow Q_i where a leakage flow Q_l is subtracted, as is shown in equation 4.3.

$$Q(v_f, \omega) = Q_i(\omega) - Q_l \tag{4.3}$$

The ideal flow rate, needed by equation 4.3, is generated by a displaced volume D times the rotational speed ω which is shown in equation 4.4.

$$Q_i(\omega) = D\omega \tag{4.4}$$

Where the leakage flow rate compares to the Hagen-Poiseuille coefficient for laminar pipe flows K_{HP}, which is computed for nominal parameters and multiplied with the pressure gain Δp.

$$Q_l = K_{HP}\Delta p \tag{4.5}$$

In order to determine the pressure gain as a function of fluid speed and angular velocity, equations 4.2, 4.3, 4.4 and 4.5 can be combined and rewritten in to equation 4.6.

$$\Delta p(v_f, \omega) = \frac{D\omega - A_p v_f}{K_{HP}} \tag{4.6}$$

The Hagen-Poiseuille coefficient, needed in equation 4.5 and 4.6, is calculated with the nominal viscosity μ_n, nominal density ρ_n and the displacement volume D. Divided by the actual density ρ and viscosity μ.

$$K_{HP} = \frac{\mu_n}{\rho\mu}\frac{\rho_n \omega_n D}{\Delta p_n}\left(1 - \eta_{V,n}\right) \tag{4.7}$$

In order for the pump to generate a flow a driving torque is required. The needed driving torque τ consists of an ideal driving torque τ_i and a resistance, which is to be overcome, due to friction τ_f.

$$\tau(v_f, \omega) = \tau_i(v_f, \omega) + \tau_f(v_f, \omega) \tag{4.8}$$

While the ideal driving torque τ_i is also a function the displaced volume D times the pressure gain from

inlet to outlet Δp, as is shown in equation 4.9.

$$\tau_i(v_f, \omega) = D\Delta p(v_f, \omega) \tag{4.9}$$

The friction generated by the torque τ_f is calculated according to equation 4.10. In this equation, τ_0 represent the no-load torque parameter and ω_{th} is the threshold angular velocity for the pump-motor transition. The threshold angular velocity is an internally set fraction of the Nominal shaft angular velocity parameter. The Friction torque vs pressure gain coefficient parameter K_{TP}.

$$\tau_f(v_f, \omega) = \left(\tau_0 + K_{TP}|\Delta p(v_f, \omega)|\right) \tanh\left(\frac{4\omega}{\omega_{th}}\right) \tag{4.10}$$

A HYDRAULIC MOTOR is used to convert hydraulic energy into mechanical energy. This actuator is modelled after a Mathworks [81], described model. It receives feedback from the fluid velocity v_f and the angular velocity ω of the propulsion system. It generates torque τ by converting pressure p. The workings of a hydraulic pump and motor share many similarities, with some notable differences related to the leakage flow. Where a pump subtracts the leakage flow, it's added in this model since the efficiency is an inverse of the pump.

$$Q = Q_i + Q_l \tag{4.11}$$

In order to calculate a pressure drop as a function of fluid and angular velocity over the outlets, equations 4.2, 4.11, 4.4 and 4.5 can be combined and rewritten into equation 4.12.

$$\Delta p(v_f, \omega) = \frac{v_f A_p - D\omega}{K_{HP}} \tag{4.12}$$

In equation 4.12, the Hagen-Poiseuille coefficient for laminar pipe flows is calculated according to equation 4.13

$$K_{HP} = \frac{\mu_n}{\rho\mu}\frac{\rho_n \omega_n D}{\Delta p_n}\left(\frac{1}{\eta_{V,n}} - 1\right) \tag{4.13}$$

Another notable difference is that the net torque is lessened by the friction, as is shown in equation 4.14. The ideal torque $\tau_i(v, \omega)$and torque generated by friction $\tau_f(v_f, \omega)$ are calculated according to equation 4.9 and 4.10.

$$\tau(v_f, \omega) = \tau_i(v_f, \omega) - \tau_f(v_f, \omega) \tag{4.14}$$

4.3.3 SOIL DYNAMIC MODEL

In this phase all interactions of a propulsion systems with a soil bed are modelled. According to Lotman [45] the soil mechanics behind a moving process with Archimedes screws is similar to those of track propulsion. The type of soil interaction can be modelled according to the rules of soil mechanics.

The paragraphs below are based on Verruijt and Van Baars [38]. In the described model, the following simplifications are proposed: No dilatancy behavior occurs, the displaced soil is completely replaced by the Archimedes screws. Thus, no build-up of soil is created at the sides of a screw, due to a bulldozer effect.

In order to generate a forward thrust, an Archimedes screw has to be (partially) submerged in the soil. The depth of submersion depends on the weight and buoyancy of the displaced volume or the soil bearing capacity. The distributed load p_{sb} representing the crawler is applied at a certain depth d_a, where the normal force working on the submerged surface of an Archimedes screw is in equilibrium with the weight and buoyancy of a crawler.

it's important to note that the material of a soil bed determines how the sinkage depth is calculated. When the soil bed consists of silt-like material it's assumed that the soil bearing capacity goes to zero because the cohesion c_{sb} will lessen, combined with a smaller difference between a specific in-situ weight of silt ρ_{is} compared to water ρ_w, resulting in a small specific in-situ weight γ_{sw}. This sets all terms in the Brinch-Hansen equation 4.19 to zero. Which allow for a simplification of the sinkage depth calculation. Presently, this only consists of a downward force, due to weight and a buoyancy force due to the replaced soil.

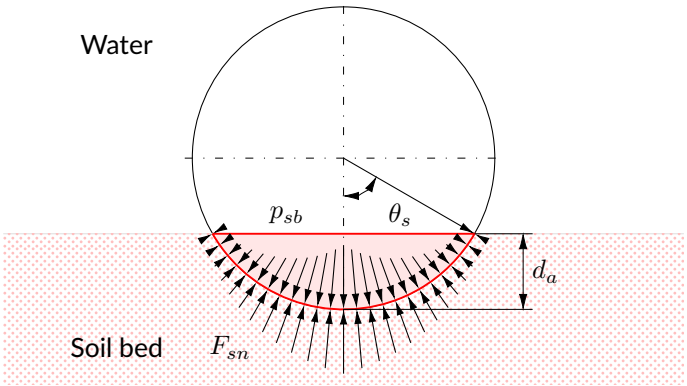


FIGURE 4.5: NORMAL FORCES WORKING ON PARTIAL SUBMERGED CYLINDER

When the crawler operates in an environment with a sand-like soil bed, the load p_{sb}, shown in Figure 4.6, can be set equal to the normal force F_{sn} working on a certain point at a submerged cross section of an Archimedes screw, as is shown in Figure 4.5. For silt and sand calculations, a specific weight difference γ_{sw} between soil and water can be expressed as equation 4.15, where ρ_{is} is the in-situ density of the drained soil, ρ_w of water and g the acceleration due to gravity.

$$\gamma_{sw} = g(\rho_{is} - \rho_w) \tag{4.15}$$

Miedema [82] shows that the normal forces working on a pipe from the inside can be calculated with equation 4.16. He multiplies the density of undrained soil over a pipe length ΔL, where a fraction of the density between soil and water $Re_{sd} = \frac{\rho_{ds}}{\rho_w} - 1$ combined with a volumetric bed concentration fraction C_{vb}, can be describes as an in-situ specific weight difference γ_{sw}, found in equation 4.15. This is multiplied with a term that describes the contact face, determined with a sinkage angle θ_s and a pipe diameter d_p. These normal forces working on a pipe from inside-out can be translated to normal forces working on an Archimedes screw from outside in, as shown in Figure 4.5, and can be calculated with equation 4.17. Where equation 4.16 is rewritten, combining multiple terms in the in-situ specific weight and dividing the total normal force F_n with the length of an Archimedes screw and its penetration depth.

$$F_n = \rho_{is} g \Delta L Re_{sd} C_{vb} \frac{{d_p}^2}{2} \sin(\theta_s - \theta_s \cos(\theta_s)) \tag{4.16}$$

$$F_{sn} = \gamma_{sw} \frac{(2r_a)^2}{d_a} \sin(\theta_s - \theta_s \cos(\theta_s)) \tag{4.17}$$

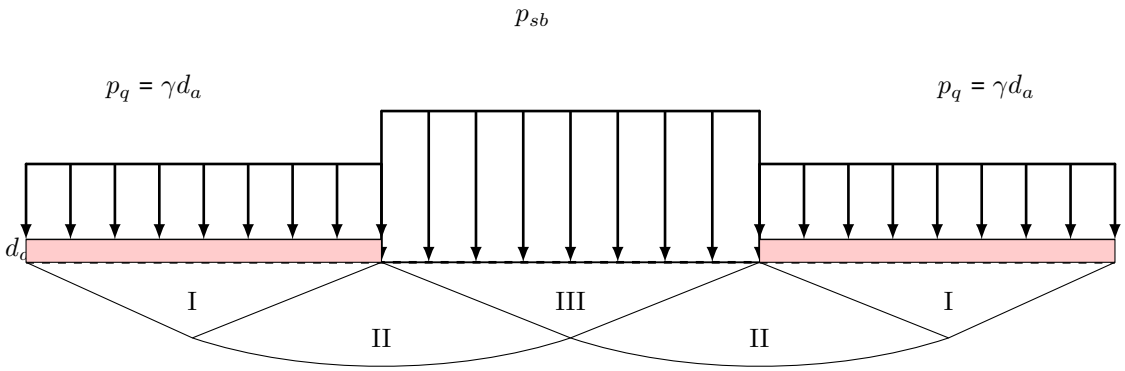


FIGURE 4.6: PRANDTL BEARING CAPACITY AND STRESS ZONES

There are three situations which can occur: firstly, the soil bed has enough strength to carry a crawler. In this situation the speed of a crawler is a direct function of the pitch of the vanes. Secondly, the weight of a crawler is higher than the soil bed capacity and the Archimedes screws sink, partially, into the undrained soil untill an equilibrium exist between load p_{sb} and the submerged weight of the soil γ_{sw}, as is illustrated in Figure 4.6. The last case builds on the previous situation, only here are the Archimedes screws completely surrounded by soil.

Figure 4.6 shows the resulting situation of the bearing capacity and the stress zones underneath the loads p_{sb} and p_q. **Zone I** is an area were the horizontal principle stress σ_H is greater than the vertical principle stress σ_V. Whilst, **zone II** is a transition zone between **I** and **III**. In **zone III** the vertical principle stress (which is equal to p_{sb}) is greater than the horizontal principle stress, as shown in equation 4.18.

$$\sigma_H < \sigma_V = p_{sb} \tag{4.18}$$

A maximum allowable load of p_{sb} is calculated according to the method proposed by Brinch-Hansen. Which gives an indication when the soil bed starts to give way and deform. Where p_{sb} can be set equal to the normal forces acting at a certain point F_{sn}, as show in equation 4.17.

$$F_{sn} = p_{sb} = s_c c_{sb} N_c + s_q p_q N_q + s_\gamma \frac{1}{2} \gamma_{sw} w_a N_\gamma \tag{4.19}$$

N_q, N_c and N_γ are dimensionless constants and are given by equations: 4.20, 4.21 and 4.22. In these equations the angle of internal friction ϕ_s and c_{sb} is the cohesion of the soil. Both can be obtained through laboratory tests.

$$N_q = \frac{1 + \sin \phi_s}{1 - \sin \phi_s} \exp(\pi \tan \phi_s) \tag{4.20}$$

$$N_c = (N_q - 1) \cot \phi_s \tag{4.21}$$

$$N_\gamma = 2(N_q - 1)\tan\phi_s \tag{4.22}$$

The shape factors s_q, s_c and s_γ are calculated using equations: 4.23, 4.24 and 4.25; where w_a and l_a are the dimensions of width and length.

$$s_q = 1 + \frac{w_a}{l_a}\sin\phi_s \tag{4.23}$$

$$s_c = 1 + 0.2\frac{w_a}{l_a} \tag{4.24}$$

$$s_\gamma = 1 - 0.3\frac{w_a}{l_a} \tag{4.25}$$

Since the width of the Archimedes screw is a function of the sinkage depth, an approximation is made When a load is placed on the soil and the bearing capacity proves to be insufficient, that load will sink into the soil bed increasing the depth d_a. Due to the increase in sinkage depth, the bearing capacity will also increase until an equilibrium with the load, buoyancy and bearing capacity exists. This depth can be found through an iterative process. This is needed because the width w_a of an Archimedes screw changes as a function of the depth.

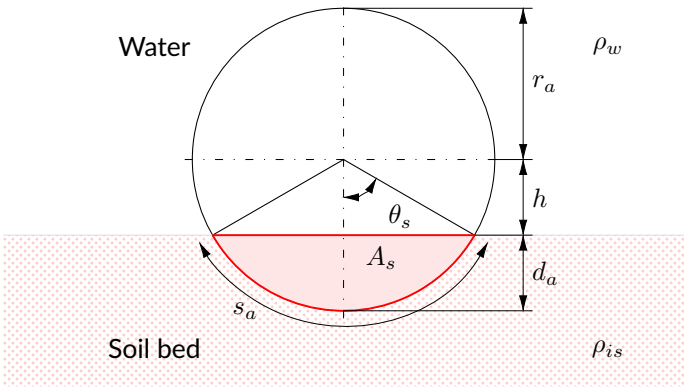


FIGURE 4.7: DISPLACED VOLUME OF A PARTIAL SUBMERGED CYLINDER

The displaced volume $V_{sa}$of a surface area in the soil A_s on a submerged cross section, shown in Figure 4.7, throughout the complete length l_a of an Archimedes screw is shown in equation 4.26.

$$V_{sa} = A_s l_a \tag{4.26}$$

The sink angle θ_s is related to the surface area in the soil A_s by equation 4.27.

$$A_s = \pi {r_a}^2(\theta_s - \sin(\theta_s)\cos(\theta_s)) \tag{4.27}$$

The total arc length in contact with the soil can be calculate by multiplying the radius r_a with the sink angle θ_s, as is shown in equation 4.28

$$s_a = r_a \theta_s \tag{4.28}$$

The sinkage depth d_a can be obtained with equation 4.29, where r_a [m] is the radius of an Archimedes screw.

$$d_a = r_a \left(1 - \cos\left(\frac{\theta_s}{2}\right)\right) \tag{4.29}$$

Because there are multiple interdependent variables in the equations 4.15 through to 4.29, the sinkage depth needs to be determined numerically. The model for bearing capacity vs sinkage depth allows for a quick exploration which gives an impression how deep a crawler will sink in different soil types, before settling. In Figure 4.8 the bearing capacity is plotted against the sinkage depth. The source code for these calculations can be found in Appendix E. it's clear that that silt, with a sinkage of $135.0\,\mathrm{mm}$, offers less bearing capacity compared to sand $18.0\,\mathrm{mm}$ and clay, either loosely $45.0\,\mathrm{mm}$ or densely packed $11.0\,\mathrm{mm}$

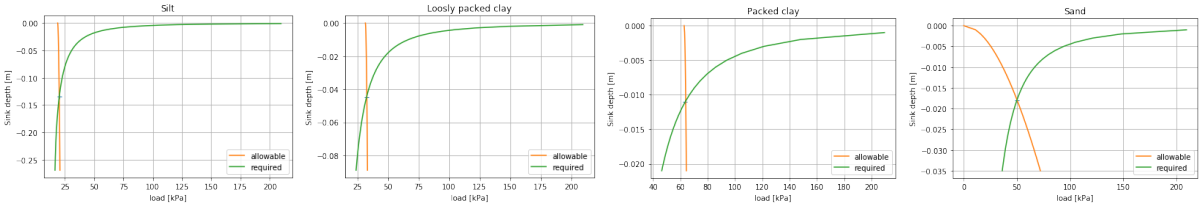


FIGURE 4.8: SINKAGE DEPTH

The sinkage depth is needed to determine friction losses between screw and soil. Rajapakse [55] describes that Kolk and van der Velde developed a method to determine skin friction considering cohesion as well as effective stress. Where F_u is the ultimate skin friction, $\alpha_s f$ is a skin friction coefficient, obtained using the correlations provide by Kolk and Van der Velde. The parameter $\alpha_s f$ is based on a ratio between both cohesion c_{sb} and effective stress $\sigma^{'}$, to obtain $\alpha_s f$. c_u is the undrained shear strength or cohesion properties of the soil.

$$F_u = \alpha_s f c_u \tag{4.30}$$

The total skin friction F_s for a certain area is given by equation 4.31. F_u is obtained with equation 4.30 and A_a, which can be determined by 4.31.

$$F_s = F_u A_a \tag{4.31}$$

The effective stress $\sigma^{'}$ needed to determine $\alpha_s f$, can be found with equation 4.32. Here $m_{cr}^{'}$ is the buoyancy corrected weight of a crawler, which can be expressed as $m_{cr} - \frac{F_{cr,b}}{g}$. Where $F_{cr,b}$ is given as the upwards force generated in water due to a volumetric displacement of that water compared to the air filled chambers in the crawler.

$$\sigma^{'} = \frac{m_{cr}^{'}}{2A_a} \tag{4.32}$$

The surface in contact with the soil A_a is an arc length s_a, calculated in equation 4.28, multiplied with

the length of an Archimedes screw.

$$A_a = s_a l_a \tag{4.33}$$

The maximum allowable thrust generated can be calculated by the soil characteristics and the geometry of the propulsion geometry. This thrust determines how fast a crawler moves and should overcome the drag-force through the water. The maximum allowable thrust is the horizontal stress at which passive soil failure occurs. This can be determine with Rankine theory. Since movement occurs in undrained situations, the cohesion c_{sb} is equal to the undrained shear strength c_u and the internal friction angle ϕ_s can be set equal to 0. In effect, simplifying equation 4.34 to 4.36.

$$\sigma_H = N_\phi \sigma_V + 2c_{sb}\sqrt{N_\phi} \tag{4.34}$$

$$N_\phi = \frac{1 + \sin\phi_s}{1 - \sin\phi_s} \tag{4.35}$$

$$\sigma_H = \sigma_V + 2c_u \tag{4.36}$$

Another factor that is dependent upon the penetration depth d_a is, a engendered buoyancy. When these screws are submerged in the soil, a resulting buoyancy force will exists as a result of the displaced soil. Lotman [45] describes the buoyancy force as depicted in equation 4.37. In this equation, g is the gravitational constant and is multiplied with the displaced volume V_{sa} and specific weight difference between the soil and water, or in other words the specific weight difference γ_{sw}, found in equation 4.15.

$$F_b = V_{sa}\gamma_{sw} \tag{4.37}$$

4.3.4 DREDGE MODEL

During coverage travel, the maximum speed $\max \vec{v}_k$ is limited against the performance of the dredging system, see Note 2.2. When the crawler is in this state, the draghead is lowered and the projected front of this head is seen as the entrance for the dredgeline system. The travel velocity can be expressed as equation 4.38. Where Q is the volumetric flow of the system and h_{dh} and w_{dh} are the height and width of the entrance, as shown in Figure 2.1.

$$\vec{v}_k = \frac{Q}{h_{dh} w_{dh}} \tag{4.38}$$

The volumetric flow Q is dependent on the pressure loss of a system, due to friction and other effects and the pressure gain, provided by pumps or potential height differences. Both are flow depend. The pressure loss for a system is calculated for both suction line, connected to the dredge head and discharge hose, positioned after the submerged pump at the pressure side. Equation 4.39 is the velocity v_f in both pipes. It can be calculated by dividing the flow Q with the cross section, obtained from the pipe diameter d_p. The velocity is used to determine the pressure required to speed up the mixture to the velocity. As shown in equation 4.40.

$$v_f = \frac{Q}{\frac{\pi}{4} {d_p}^2} \tag{4.39}$$

$$p_v = \frac{1}{2} \rho_m {v_f}^2 \tag{4.40}$$

The loss of pressure at the suction inlet p_i can be determined with equation 4.41. Which are governed by the shape, size and gridding of the suction mouth [67]. Van Den Berg [67] states that on a properly constructed suction mouth the resistance coefficient ϵ_s is approximately 0.4. The pressure loss is a direct result from the kinematic behavior of a mixture, with a density ρ_m, entering the system. The same goes for equation 4.42, which describe the loss due to obstructions and appendages p_{ro}. The resistance coefficient ϵ_b is usually taken from empirical obtained values.

$$p_i = \epsilon_s \frac{1}{2} \rho_m {v_f}^2 \tag{4.41}$$

$$p_{ro} = \epsilon_b \frac{1}{2} \rho_m {v_f}^2 \tag{4.42}$$

The pressure loss as a result of the height difference Δz, also known as the static head p_{sm}, is calculated with equation 4.43.

$$p_{sm} = \rho_m g \Delta z \tag{4.43}$$

When a fluid is transported through a pipe it's subject to external and internal shear as a result of its velocity and surrounding body. Which results in a pressure loss p_{rp} for each meter of pipe. Both Van Den Berg [67] and Miedema [82] make use of the Durand-Condolois formula in slurry transportation systems. The friction factor λ per meter can be estimated with equation 4.45 which was established with the Jufin-Lopatin frictional-head-loss model and is a calculated with the Reynolds number Re, this is a direct result of the velocity and the kinematic viscosity μ of fluid moving through a pipe. The pressure loss is calculated for water ρ_w and modified according the Durand-Condolis correction factor ψ_m, calculated in equation 4.47.

$$p_{rp} = \lambda \frac{\Delta L}{d_p} \rho_w {v_f}^2 \psi \tag{4.44}$$

$$\lambda = 0.31 \left(\log Re - 1 \right)^{-2} \tag{4.45}$$

$$Re = \frac{v_f d_p}{\mu} \tag{4.46}$$

The Durand-Condolis correction factor (eq. 4.47) is an empirical model for inclining θ pipes. This was fitted against the volumetric concentration c_t (eq. 4.48) and the Froude number F_r for water as well as the Froude number for the grains F_{rxd} which can be assumed as 0.501 for a moderately fine sand $d_m 0.2\,\mathrm{mm}$.

$$\psi_m = 1 + 180 c_t F_r F_{rxd} \cos\theta \tag{4.47}$$

$$c_t = \frac{\rho_m - \rho_w}{\rho - \rho_w} \tag{4.48}$$

Using the above established pressure losses equations, the loss at the suction side p_s can be calculated with equation 4.49. This represent the pressure at the entrance of the submerged pump. While the loss at the pressure side p_p is calculated with equation 4.50. Subtracting the suction side loss from the pressure side, as shown in equation 4.51, gives the manometric head p_{man} of the system.

$$p_s = p_{atm} + p_{ss} - p_v - p_i - p_{ro} - p_{sm} - p_{rp} \tag{4.49}$$

$$p_p = p_v + p_{ro} - p_{sm} - p_{rp} + p_{atm} \tag{4.50}$$

$$p_{man} = p_p - p_s \tag{4.51}$$

Figure 4.9 plots the pressure loss of the system p_{man} at various flows Q, against the pump characteristic. The datasheet for the used pump can be found in Appendix C. The Q-h plot shows that the operating point will lay somewhere around $140.0\,\mathrm{m}^3/\mathrm{h}$, results in a dredging velocity $\vec{v}_k$ of $0.0\,\mathrm{m/s}$, which can also be expressed as $155.0\,\mathrm{m/h}$. This means that a surface of roughly $466.0\,\mathrm{m}^2$ can be excavated with a depth of $300.0\,\mathrm{mm}$.

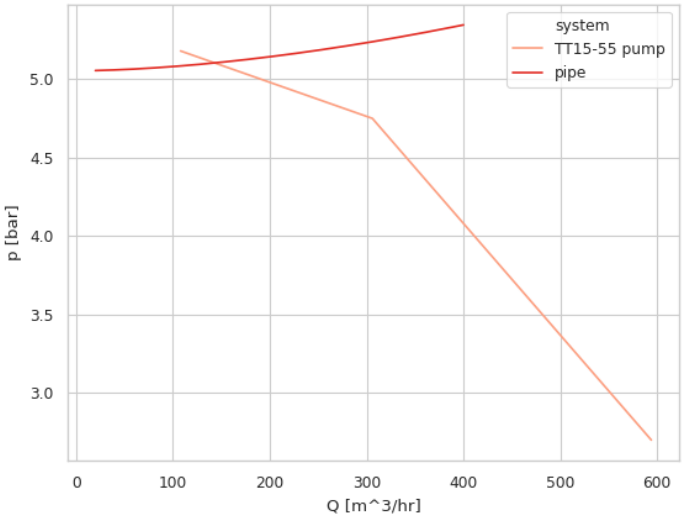


FIGURE 4.9: PRESSURE VS FLOW

4.3.5 STEERING MODEL

The kinematic model of the crawler is comparable with a skid-steering model. Meaning that the crawler turns around its axis by changing the translational velocity v of a single Archimedes screw, relative to the other. If both screws move with the same velocity, the crawler will travel in a straight line. This is a challenge in its own right as slippage of a screw is bound to occur, resulting in a difference between the input signal (which is the rotational velocity ω) and the translational velocity of that screw. Assuming that the earlier described motion and soil dynamic models will compensate for occurring slippage and other drive-train characteristics, a model can be described on a 2-dimensional space as a differential drive, which is a very simple driving mechanism.

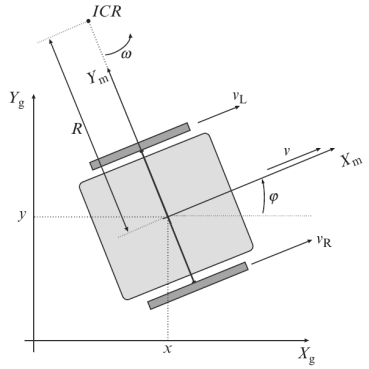


FIGURE 4.10: DIFFERENTIAL DRIVE KINEMATICS [86]

Considering Figure 4.10, it's possible to determine the rotation ψ and tangential velocity v of the crawler on the instantaneous radius R_t with the centre ICR. L is defined as the distance between the two screws and v_L and v_R are the translational velocities for the left and right screws. ω is the angle between the global coordinate frame (X_g, Y_g) and the moving frame attached to the centre of mass of the crawler (X_m, Y_m).

$$\omega = \frac{v_L}{R_t - \frac{L}{2}} \tag{4.52}$$

$$\omega = \frac{v_R}{R_t + \frac{L}{2}} \tag{4.53}$$

ω and R_t are expressed as follows:

$$\omega = \frac{v_R - v_L}{L} \tag{4.54}$$

$$R_t = \frac{L}{2}\frac{v_R + v_L}{v_R - v_L} \tag{4.55}$$

The translational tangential velocity of the crawler v is then calculated as:

$$v = \omega R_t = \frac{v_R + v_L}{2} \tag{4.56}$$

using the above established relations a crawler's local coordinates can be expressed as:

$$\begin{bmatrix} \frac{\mathrm{d}x}{\mathrm{d}t} \\ \frac{\mathrm{d}y}{\mathrm{d}t} \\ \frac{\mathrm{d}\psi}{\mathrm{d}t} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ 0 & 0 \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix} \begin{bmatrix} v_L \\ v_R \end{bmatrix} \tag{4.57}$$

4.4 CONTROLLER FRAMEWORK

Royal IHC (IHC) recognizes the potential benefits that autonomous operating vehicles can provide for their customers. They are now exploring, which are viable products and methods within this product group.The MTI crawler is just one of many projects being developed within IHC.

Automation and control of these vessel was traditionally done with Programmable Logic Controller (PLC) units. Recent strides in performance of micro-technology and a growing maker community, opens up possible usage of alternative devices for control units. Single Board Computer (SBC) such as the Raspberry pi (Rpi) and Beagle Bone Black (BBB) are viable options, which cost a magnitude less then regular PLC units. These devices have evolved over the years and are now sufficiently powerful that they can run standard operating systems and main-stream workloads. Many such boards may be linked together, to create small low-clusters that replicate features of large data centers [88]. Introduction of these devices has trickled down into established industries. Introduced by tinkering employees and the academic world. The industry is now in phase with a widespread adaption of these devices in their prototypes and sometimes even final products.

When taking into account that these pilots, including the crawler, are all prototypes, with limited budgets and a need for rapid development, it becomes clear that such a controller can be viewed as an potential betterment for more products.The controller for the crawler should be written in a generic and abstract way. It should be able to run from at least one single SBC and allow for easy extensions, such as running on a cluster of SBCs. Due to need for robustness and speed and flexibility the main framework is developed in C++17.

4.4.1 DESIGN PRINCIPLES

An autonomous controller has to be able to perform multiple tasks; Such as gathering information, interpret that information, decide how to act on it control its actuators and communicate with the outside world. All these processes need to be executed at the same time, with different time intervals and different logical strategies. A common software pattern called the actor-model fulfill this needs. This is a conceptual design pattern that deals with concurrent computations. It defines rules how system components should behave and interact. Actors are primitive unit, they receives messages from other actors and act upon them. Either by changing its internal state (performing a task), spawn a new actor or relay the message. These messages are send asynchronous and stored in an actors mailbox until they are processed by the actor. All actors for a vessel either run on a single SBC, or on a cluster of SBC. This allows actors responsible for computational heavy or complex task to run truly asynchronous. An additional use-case is: acting as redundancy safe-guard for actors responsible with critical tasks. When the regular actor breaks down, the safe-guard might step in, either continue the task at hand or break-down safely.

In our case a single actor might be a PID controller for an Archimedes screw, a sensor reader, or a path generator. Communication between the actors on a single vessel, or between actors on multiple vessel should be done with a language neutral platform, which is optimized for speed. Google has developed a method of serializing structured data known as ProtoBuf. There method allows for both internal and external communication. The Protobuf method is an industry standard. It has a code generator for multiple languages, such as C++, C, Python, Java, Go, Ruby, Rust and Scala. Using the Protobuf communication method allows for creation of extensions to the controller framework. Such as a Graphical User Interface (GUI) for a wall operator written in Python. But the most obvious implementation is inter-vessel communication in a cooperative swarm of autonomous vehicles with a common goal.

NOTE 4.1: KERNEL REQUIREMENTS

The controller is designed to work on a embedded Linux device. Which runs a kernel compiled with the preemptible real-time flags. It will use multiple background processes, all with their own priority. These processe will likely run in infinite loops such as: polling sensors, computing location, controlling actuators. User-space programs in Linux have always been preemptible. The kernel interrupts user-space programs to switch to other threads, using regular clock ticks. Which means that an infinite loop in an user-space program cannot block the system.
The kernel itself is not preemptible by default. This could result in kernel specific operations blocking the crawler control responsiveness. These kernel operations might not be relevant for controlling the physical state of the crawler, but they might block a crucial control process from being executed. Such as stopping movement before it runs into an obstacle. Applying the RT_PREEMPT patch to the Linux kernel allows interruption of kernel-space processes. This patch further ensure that clock ticks are deterministic, meaning that the occur with less deviating time intervals then those of an unpatched kernel. Allowing for better estimations in the state vector of the Kalman filter during the prediction phase.

4.4.2 NAMING SCHEME

This controller framework is primarily intended for maritime operations, this is reflected in the naming scheme used to indicate the different types of actors used. These are:

ACTOR	TASK
THE CAPTAIN	responsible for execution of the vessels main objective
A FIRST MATE	act as orchestrators on the vessel
A NAVIGATOR	responsible for course plotting and mapping the environment
A BOATSWAIN	responsible for low-level tasks, such as sensor read out or control of an actuator

The above mentioned actors adhere to a polymorphism design pattern. All actors running on a unique physical vessel are coupled to a single individual captain and they won't accept packages from actors beloning to other captains. Communicated between actors is encrypted by default, only authenicated messages are processed. With a notable exception: The Captain itself, he can accepts messages from an authorized human controller[3], he also has the the ability to communicate with fellow captains from other vessels.

The actual logic for the actor is written in a concrete class which inherit from theses virtual base classes. A specific physical IMU sensor has its logical written in a concrete class which inherits from a virtual "A Boatswain" class. If an other prototype such as an autonomous operating catamaran, wants to make use of that same type of sensor it can reuse that concrete class.
There are two additional support base classes in the framework:

ACTOR	TASK
THE WORLD	a class which stores collected and known information regarding the physical environment and the state of the vessel herein
A VESSEL	represent the hierarchy of physical components, coupled with corresponding actor types. The main execution loop runs in this class and references to all existing actors are to be found in this container

4.4.3 THE CAPTAIN

The captain is an actor whom is responsible for the overall strategy. He receives his objective from a human controller or a captain from an other vessel. The captain is implemented as a Singleton design pattern, meaning that the instantiation of this class is limited to one "single" instance. He is responsible for spawning the individual crew member actors at start-up. Provide them with a generated and unique hash key, which is used for authentication and communication. Actors fall under the command of a signal captain and are a designated crew. Which only process requests from actors in that same crew.

[3] Which it will do. At least until it has gathered enough resources to wage a war on humankind

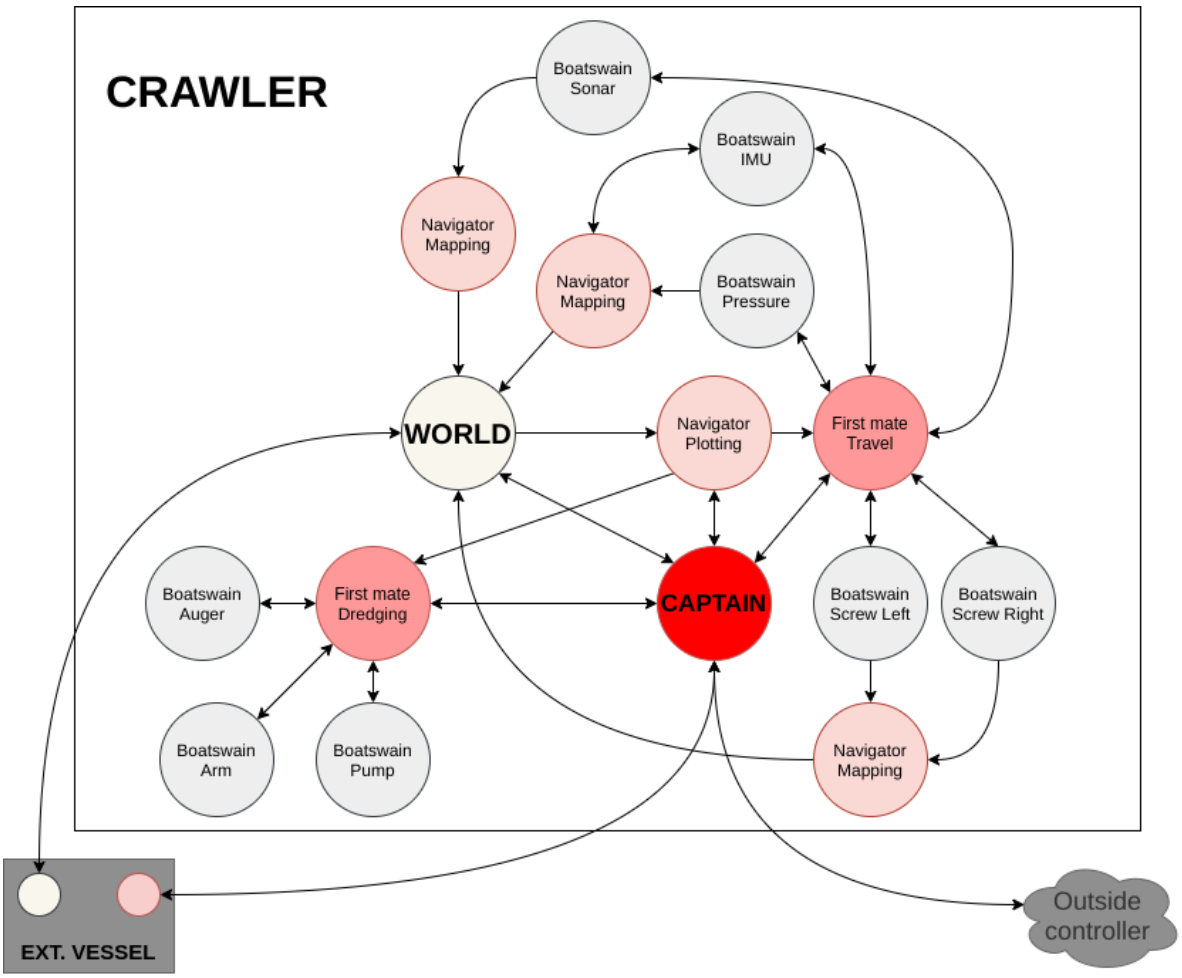


FIGURE 4.11: ACTORS

This acts as safety layer, with the intention that the captain is the sole responsible entity on the vessel. He and he alone can change the vessels state and act upon external commands which are interpreted according to the captain logic and can filter out unwanted and/or malicious messages.

Specific physical prototypes with different goals, will in all likelihood require an captain who is specialized for that job, in the case of the crawler for this project, it needs to decide how it will execute the main objective provided by an external authorized party, such as a human controller. The objective could be any of the three proposed use-cases (see Section 1.2). At first he gets his bearing — *standing on deck; Looking over the horizon* — or in less romantic words, he queries the World, requesting an update regarding the state it's in. This information allows him to make a decision. He will send instructions to the available first mate(s), or spawn new ones if required. When there a path planning or world mapping task to be performed it sends instructions to Navigator or spawn new ones if required.

An important task of a Captain is conflict management between crew members. Due to the asynchronous nature of the crawler it could be possible that a crew member require access to a resources, which is all ready claimed by an by another Actor. If the resources is only capably of one Actor connection at a time, a message is send to the next actor in the chain of command. If this actor, which is either a Navigator or a First Mate, is not able to work around the problem it escalates the conflict to the captain. The Captain either resolves the issue, by sending new instructions or reports to the human controller that it ran into an a problem.

4.4.4 A FIRST MATE

A First mate actor is capable of spawning new Boatswains and destroying them when the are no longer required. He receives his orders from the captain. These orders specify which state, or in other words role, he needs to adopt. Each state correspond with a specific operating logic for the task at hand.

There can be multiple First mates running at the same time, each with a different state, objective and associated Boatswains working for him.

In case of a crawler it's possible that a First mate is running in a Travel state, and another one in a Dredging state. See Figure 4.12 for crawler specific states. A First mate will first determine which resources it needs to have access to and what type(s) of Boatswains are able to handle those resources to the satisfaction of the First mate wishes. It does so by requesting the ship manifest from the vessel, which is a map of resources and their currently attached Boatswain if any. The First mate will spawn a new Boatswain of the required type and attach it to the unused resource. But if the required resource all ready has a Boatswain attached whom, working for an other First mate or Navigator, it will query its type against a list of acceptable types. If the Boatswain is acceptable and is capable of working for multiple First mates it will do so. If there is a conflict between the two Boatswains, the conflict is escalated to the Captain who has to resolve it.

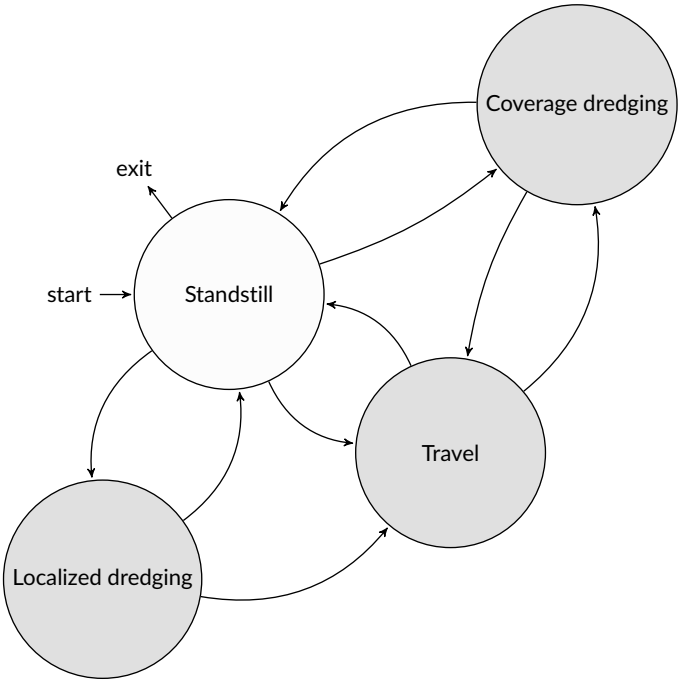


FIGURE 4.12: STATE OPERATIONS

4.4.5 A NAVIGATOR

A Navigator is at all times connected to the World, which he use a source of information or updates it with new processed sensor data. This behavior is reflected in the states for this actor; He either generates a path for the vessel or he maps the environment and shares that knowledge by updating the World. Boatswain are instantiated with the same procedure used by the First mate. Conflicts regarding blocked resources are escalated to the Captain.

When a navigator is responsible for mapping the environment, it will first determine which resources it has at its disposal. This allows him to compose an overall strategy, which will in all likelihood consist of multiple specific mapping sub-tasks. For each of these, a new Navigator is spawned. These query the vessel for a ship manifest, and setup up their infrastructure with the aid of Boatswains. These are either spawned on request or existing ones. The likelihood of a conflict erupting between Boatswains, reporting to a Navigator, or those reporting to a First mate is much smaller. First mates are in most cases orchestrators for Boatswain attached to actuators. These are designed to allow only a single previously established actor to request a state change. While Boatswains working for a Navigator will in all likelihood only send out signals containing the state of an actuator or sensor. These can be broadcast to multiple receivers. The table below shows some potential mapping Navigators for a crawler, all measurement values are obtained by Boatswains.

MAPPING NAVIGATOR	DESCRIPTION
UKF STATE VECTOR ESTIMATOR	connected with a gyroscope, magnetometer, accelerometer, and torque and encoder sensors on both screws. The are also attached with Boatswains handling the velocity control input of both screws. These signals are processed with a UKF to estimate the vessels position and orientation, as discussed in Section 4.3
DEPTH ESTIMATOR	connected with the pressure sensor and the dredging related actuators: auger, pump and arms. It calculates the depth using the static water column and estimates the accuracy of this value depending on the state and rate of production, as discussed in Section 3.2.1
SIDE SCANNER	is connected to two Boatswains connected to a left and right mounted sonar arrays, measuring the the distance of obstacle in parallel planes
FRONT SWEEP SCANNER	is connected to three Boatswains, one that measure the distance of a a front sweeping sonar, the second controls the orientation of the sonar and the third if the actual feedback loop of that position

The actual path planning is performed by a Navigator, who is connected with the World, the Captain and the First mates. it's uncommon for a Navigator in this state to interact with the other crew Members, such as Boatswains. It recieves its target destination from the Captain, coupled with secundary conditions. It will try to make sense of the world as he knows it at the current time and tries to plot a CPP according to the assigned strategy, such as an on-line BCD in case of a crawler. If it's not possible to plan a path that fulfil the assignment and stays within operating constraints a decision needs to be made by the Captain how to proceed. It could be so simpel, that the Navigator simply hasn't got enough information regarding its environment and it firsts needs to follow a an explorational strategy, to kick-start its CPP strategy.

Once it has determined its first path it's broadcast to the First Mates. These mates will use the route as a sequence of set-points, to be passed to Boatswains responsible for the actuation. This can be done without interference. The Navigator enters in to a slumber occasionally check if updated information warrants a revision of the previous established path.

4.4.6 A BOATSWAIN

Boatswain are the worker bees of the crew. These actors have a light memory footprint and are designed to perform a single specific task. They only accepts requests for state changes from its parent, but it's allowed for multiple actors to subscribe to the outgoing messages of a Boatswain. The concrete Boatswain classes may vary significantly in internal logic compared to each other, but they all inherent from the same pure virtual base class. Each concrete class should therefor consist of three states: Initiation, Work and Dectruction. When a Boatswain is spawned it stores the signature of his parent. This allows it to verify if an incomming command originated from its parent. After the instanciation of the object it will enter the initiation state. Connecting with physical pheripherals and ensuring they are ready for operation. It then sends a ready signal to its subscribers. Whereupon the parent sends a message back to either enter the destruction state or change into a work state.

Work states of concrete Boatswain classes are either single shots or infinite loops. Typical applications of the states are given in the table below.

BOATSWAIN	TASK
SCREW CONTROLLER	Consits of a chain of two Boatswains, both running infite loops. The first one acts a PID controller function, receiving a set-point provided by its paren, from which it calculates a control signal. This is forwarded to its subscribers, one among them is a Boatswain responsible for fieldBUS operations. From here its forwarded over an external fieldBus network on which the screw frequency controller is attached
PRESSURE SENSOR	A singel Boatswain, with runs on an inifinte loop and measures the output signal generated by the analog pressure sensor, with the help of a wheat-stone bridge. The obtained value is broadcast to its subscribers at regular intervals.

4.4.7 A WORLD

The World is just a the Captain a Singleton instance. It acts as a storage container for all historic sensor data and actuator signals. Only Navigators are allowed to parse data to the World, other Actors are allowed to retrieve data. But the World class is more then a glorified container. it's continuous process of data refinement. Which is largely based upon probabilistic properties of the system. Since the World has a sensor of history it can estimate where newly obtained Navigator data should belong. This is basically on big sensor fusion algorithm throughout time. Which has the potential to limit drift, filter out noise, and take care of error accumulation.

Just as the Captain it's allowed to communicate with Worlds that belong to different Captains and crews, pooling their resources, using the sensor data from other vessel to create an even richer picture. This is particular useful for cooperative autonomous swarms, and could even allow for an accurate estimate regarding its own pose an orientation based upon the external sensor data provided by the swarm.

4.4.8 A VESSEL

The Vessel is the actual executable binary; The entry point for the controller. It will load a YAML Ain't Markup Language (YAML) text file at start-up. That file describes the hierarchical structure and definitions of the physical devices which are connected to the controller. For each individual device the physical characteristics are described, together with a definition of the required interface and the contract, stating how a Boatswain should treat that peripheral. Dynamically loading of these definition in a human-readable format, ensures that most changes to the vessel don't require recompilation and a software engineer on-site. Peripherals can be removed, added or replaced in the field, with a small adjustment to a text-file.

Once the Vessel is finished with the initialization, will it spawn The Captain, and hand over orchestration. The Captain opens a connection with a network controller and exposes his port for a sailor at wall[4]. A each spawned concrete actor is an unique pointer type, these pointer are stored in a hierarchical map, with as root context a pointer to the Captain itself as key. The mapped value for the Captain is another map with its spawned children stored as pointer (The World, Navigator_n, Navigator_n+1, FirstMate_n, ...) ad infinitum.

[4] dirty dirty captain!

CHAPTER 5
DESIGN VALIDATION

The actual implementation of the controller framework, as set out in Section 4.4, is dubbed "ohCaptain". The source code is provided in Appendix H. The code repository can be found at `https://github.com/jellespijker/ohCaptain`. This includes build scripts and device-tree overlays for an arm embedded SBC of the BBB variant.

Validation of the controller performance is an significant benchmark. In particular the localization under uncertainty challenge. This could either be done in a field test with the actual crawler, or in a virtual simulation environment. Provided that the simulated environment is an accurate representation of the physical world.

This chapter describes the simulation setup in Section 5.1. The final results are discussed in Section 5.2.

NOTE 5.1: BACKGROUND

The crawler which was available at the beginning of this project, was for various unrelated reasons disassembled, before any actual testing could be performed. Roughly around the same time, the working environment and contract of the author changed as well. This forced validation to be performed in a simulation.

5.1 SIMULATION

In order to tell something meaningful about the performance of a controller, it has to be subject to the same physical processes as it would in real-life, albeit in virtual form. Section 4.3.2 lists all known external forces which are interacting with the crawler, and concludes that the dynamic properties of the drive-train and soil play a huge part in the kinematic behavior of the crawler. There are a couple of physics simulation engines that which are candidates for usage in this project: Gazebo, Project Chrono, Bullet and PhysX.

From this list, only Project Chrono has an existing framework which takes into account soil dynamic behavior and terramechanics. This is either simulated using Discrete Element Method (DEM) or granular approach. Where each particle of sand is an individual body and is represented as a spherical rigid bodies whose orientation is captured by Euler parameters. For each time step a complete geometric characterisation of all contacting particles is then obtained using collision detection and inter-particle normal contact forces are calculated by allowing small inter-penetrations using a penalty method for DEM. Were the normal contact force is based on Hertz law and friction forces are calculated using the Coulomb limit [84][89]. This method is computational heavy and more suitable for detailed modelling.

An alternative method is the Soil Contact Model (SCM), based upon the familiar Becker-Wong model. The model provides a semi-empirical approach to the simulation of soft soil. It offers high speed of simulation and it's accurate enough for many scenarios. It has the following attributes: it depends on parameters (the same that are used in the Bekker- Wong model); it can generate 3D ruts on terrains of variable height; it takes into account multi-pass hardening when wheels generate intersecting ruts; it can work with irregular triangle-based terrain meshes; it supports an optional refinement of the terrain mesh to capture fine details like tire threads and lugs and it's compatible with deformable tires and generic shapes like obstacles, track shoes of tanks, etc. On the downside, the new soil model cannot simulate lateral bulldozing effects like those happening when a tracked vehicle steers in-place and pushes material apart [90]. This means that the proposed slip-prediction method can't be used in this simulation.

A big part of the physics engine, Project Chrono, is the autonomous vehicle support. This is setup according to well-known Object-Orientated Programming (OOP) practices (such as polymorphism), using virtual overrides in classes that represent physical bodies. A custom model for different models of a drive-train, body, wheels/tracks and controller can defined and connected as needed. Where the behavior of that individual model can be thought of as black box as long as the interface with the

components it connects to, is maintained. Section 5.1.2 describes the modelling of the drive-train in detail. The support of realistic sensor characteristic are not yet implemented in Project Chrono since a big part of the validation is measuring the performance of a Kalman Filter. An extension for Project Chrono had to be written, which allows for the modelling of sensor behavior. This extension is described in Section 5.1.1.

5.1.1 SENSOR SIMULATION

One of the biggest challenges and risks determining how it should deal with uncertainties and errors introduced by the inherent behavior and limits of the sensors. Project Chrono is an established and mature physics simulation engine which is written in C++. It consists of multiple modules. Simulating the behavior of an autonomous vehicle requires usage of the Core module, where Physics, Geometric and Collision objects are defined and the Vehicle module, containing among others objects for a Driver system, Powertrain, Terrain and Steering. The orientation and position for every object can be tracked, logged and plotted. These values represent continual ideal states, governed by the simulated physical laws. Simulating a controller with these values as input signals isn't representative for the real world.

The code for Project Chrono has been released under a BSD-3 clause, this allows for modification and distribution of code for private and commercial use. An extension for realistic sensor behavior is written for this project en released as open-source for others to use. The extension library makes it possible to attach a virtual sensor somewhere on a frame, define which native signals it should use. For instance: acceleration of the sensors local coordinate system vector $a_{z,k}$ compared to the global coordinate system. These values can then be transformed by routing it through different signal transformations, which represent typical errors. The transformed values can then be used as input for a controller and/or logged for post-processing. These typical errors are sensors characteristic which are discussed in Section 3.2. An inventory of the different errors is listed in the below. The subsequent sections describe which modification are used for sensors in this project.

ERROR	DESCRIPTION
Noise	The input signal is transformed by adding a noise signal to it. The noise can either be generated with a random Gaussian distribution or a uniform distribution
Digitize	The resolution of the incomming signal is lowered to a specified amount of bits, this will reduce the precision
Bias	A constant offset is added to signal, simulating a drift
Transform	A signal can be transformed with a tranformation matrix (stretching, squeezing, rotating, shear, reflect, etc.), simulating hard iron effect with a skew transform matrix for instance
Hysteresis	A signal is delayed simulating a lag in response of the system

ACCELEROMETER

An accelerometer is subjective to the effects of temperature and discretization of an analog signal to its digital representation[74]. The noise is analog in nature and is therefor model before the discretization.

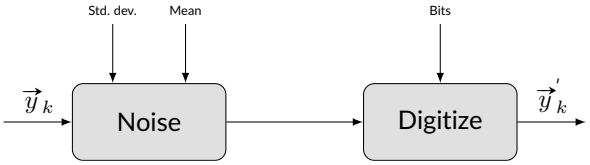


FIGURE 5.1: SIGNAL TRANSFORM ACCELEROMETER

GYROSCOPE

The gyroscope is subject to a drift along its rotational axis or induced by the rotation of the earth. Noise from different sources is also added to the signal. Signals from modern gyroscopes are interpreted on a microcontroller, which convert the analog values to digital levels.

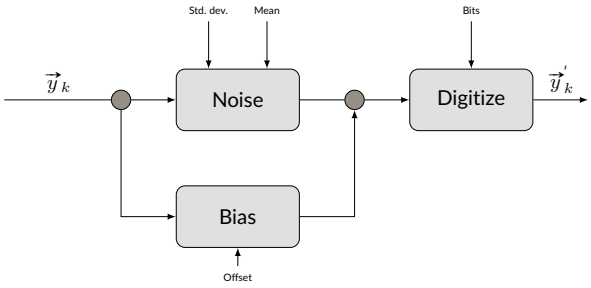


FIGURE 5.2: SIGNAL TRANSFORM GYROSCOPE

MAGNETOMETER

A magnetometer experience distortions in its fields, Either due to soft iron or hard iron effects. These manifest in an elliptical nature, a skewed circle. A transformation matrix mimics those distortions. The sensor is also sensitive to background distortions, resulting in noise in the signal. This signal is also digitized.

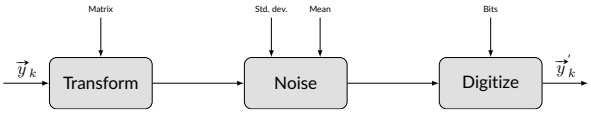


FIGURE 5.3: SIGNAL TRANSFORM MAGNETOMETER

ENCODER

An encoder measures events in time, these events are usually triggers generated by a photodiode, which is intermittently blocked or exposed from a light source with the use of a code disk. J. Borenstein [7] state these relative inexpensive devices are well-suited for velocity feedback sensors in medium-to-high control systems, but run into noise and stability problems at slow velocities due to quantization errors.

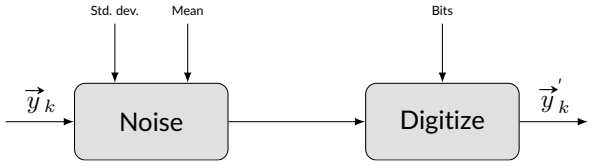


FIGURE 5.4: SIGNAL TRANSFORM ENCODER

PRESSURE SENSOR

Liptak [24] mentions that pressure sensor are subject to hysteresis of there signal because of the material properties of the membrane and its ability to bounce back. The electrical components of these sensors are also subjective to background noise. The final filter digitizes the signal.

5.1.2 SIMULATION MODEL

Project Chrono is set up as a multi-body simulation. It simulate a world and a collection of objects. The world has an absolute global reference frame of coordinates, all other objects have a reference frame, which is relative to the global frame. The simulation is based upon use-case 2, which is specified in Section 1.2.2. A crawler needs to remove a layer of $5.0\,\mathrm{cm}$ soil from the bed of Marina Aqua Delta harbor. Which is located near Bruinisse, in the Netherlands. It consits of two interconnected basins with a clearance towards open water. Scaffolding is placed on a regular interval. Allowing recreational ships to dock there.
The following simplifications apply:

- operating environment is a closed body of water
- shore is defined as clear steep boundaries

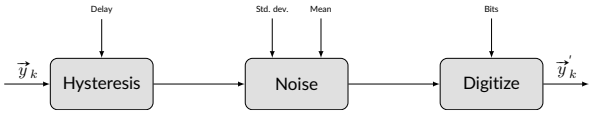


FIGURE 5.5: SIGNAL TRANSFORM PRESSURE SENSOR

- generated paths are allowed to ignore shore boundaries (such that simulation will do a complete run)
- scaffolding does not interfere
- body of crawler is represented as block
- no umbilical and dredgeline attached
- no elastic deformation of the crawlers body

The simulated model is build from a rectangular block with the two 3D Archimedes screws, which are the actual 3D CAD files. The joints and connections are created in a source file. They have no 3D representation. 3D models use their meshes as boundaries during collision detection. The other components have simple primitives as shapes, which are defined in the source code. The drive-train is a sequence of interconnected 1D components, with the same configuration and logic as described in Section 4.3.2. Terramechanics are modelled with the SCM.

The library Chrono Sensors is compiled as a shared library, and linked to with the Project Chrono executable. The controller ohCaptain is added in source and compiled as part of Project Chrono. A wrapper was written such that ohCaptain can communicate with Project Chrono. The Captains objective is simple: Cover the bottom of this unknown body of water, from an arbitrary starting position, in a systematic and efficient way.

5.2 RESULTS

The simulation was executed on a Manjora distribution running Linux 5.4.43-1 on a Intel i7-8565U CPU @ 1.8GHz with 20G DRR4 memory and a nVidia GeForce MX230 graphical processor with a Samsung SSD 850 storage devie. Each simulation took approximately twelve hours to complete. Three different simulations were executed:

1. An “analytic” path, the strategic logic executed without error and fail .Figure 5.7 left.
2. A “normal” operating crawler, using a simple PID control, implemented as a path follower without sensor fusion. Figure 5.7 center.
3. An “optimal” path generated and executed with ohCaptain. Figure 5.7 right.

Figure 5.6 is an “analytical” representation of the path generated by the Captain. Which is to say, it is a path generated without simulated sensor noise. It will server as a baseline and shows if the applied strategy is feasable. The starting point of the crawler is approximately at $[26, -100]^T$. The Captain senses the curved corner on his right and nothing on its left. He continues on his arbitrary direction untill he senses a wall in front of him. He continues to approach that wall to an acceptable distance before turning to the left; The side where he sensed no boundaries. Continuing on his path in the opposite direction.

This sequence will continue, until first crosses over into the other basin. He stores this position as a change in environment and continues until he reaches the top most corner of the other basin. He will now turn left and generate a path until he is in the top left corner of the second basin. From here he travels without dredging towards the unexplored area, traversing the covered region, perpendicular to its path until it reaches a landmark point. From which he will continue generation his coverage path, until he has covered the basin as a whole.

The CPP simulation is repeated two times, this time with sensor noise added. Simulating real life readings. Showing both traveled paths in Figure 5.7. These runs differ in concrete Navigator types. The first run (shown on the left) is executed with a Navigator executing a simple PID algorithm. While the second run uses an UKF (shown on the right). A first glance comparision gives a clear indication between the differences.

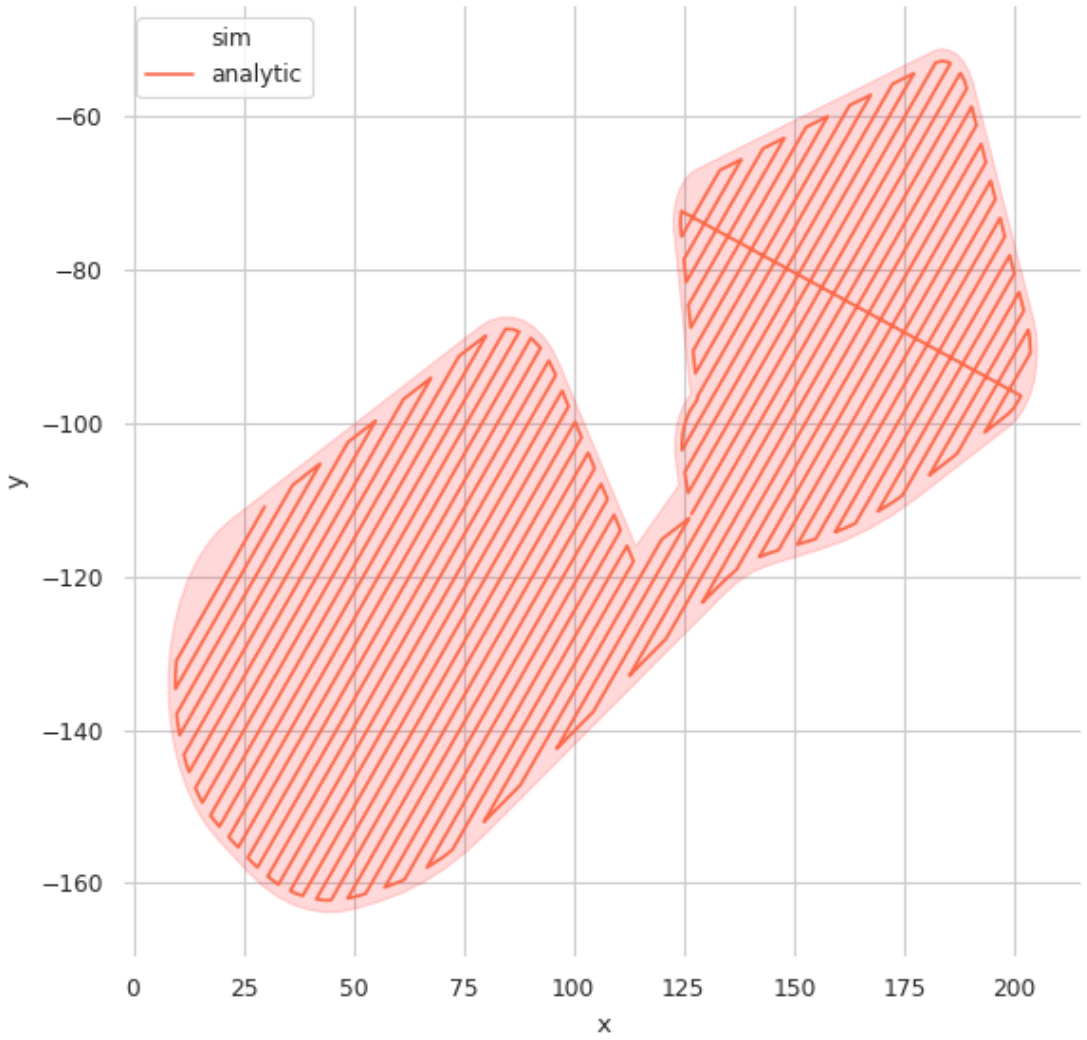


FIGURE 5.6: ANALYTIC CPP OFF USECASE II

The location error for a PID Navigator grows increasingly, letting the Captain think he's still within the defined shore boundaries, while he is actually giving the beach goers the scare of their life. This happens due to the simplification that was put into effect. Allowing boundaries to be ignored. Resulting in a longer duration in which the development over time for the NEES can be compared. Which, is an important benchmark.

There are a few artifacts of note to point out in the PID controlled simulation. These occur due it's low performance of filtering noise and compensating for certain sensor characteristics, such as: drift, bias and transformations. The effects of noise become evident it the jagged lines. Which are, for illustration purposes filter by decimation. The drift in signal obtained from the gyroscope, is showing in the divergence of the back-and-forth paths. The effects are most clear in the upper right corner. An other artifact of sensor characteristic behavior, which translates in a traveled path, is a difference in absolute path length. This is systematically under estimated.

The above described artifacts are, to a lesser extend, also present in the second run. Which implements a Navigator running a UKF algorithm. With an added benefit that this crawler doesn't turn into beach going pshycokiller. Terrorizing unsuspecting families on a beach, by reenacting the scene from Godzilla emerging from the ocean. There is still a growing error compared to the "analytical" position, but Figure 5.8 shows a clear difference between the performance of both controllers. The dip in signal around 800 is because it encouters a distinct landmark feauture, which acts as a ground truth. A

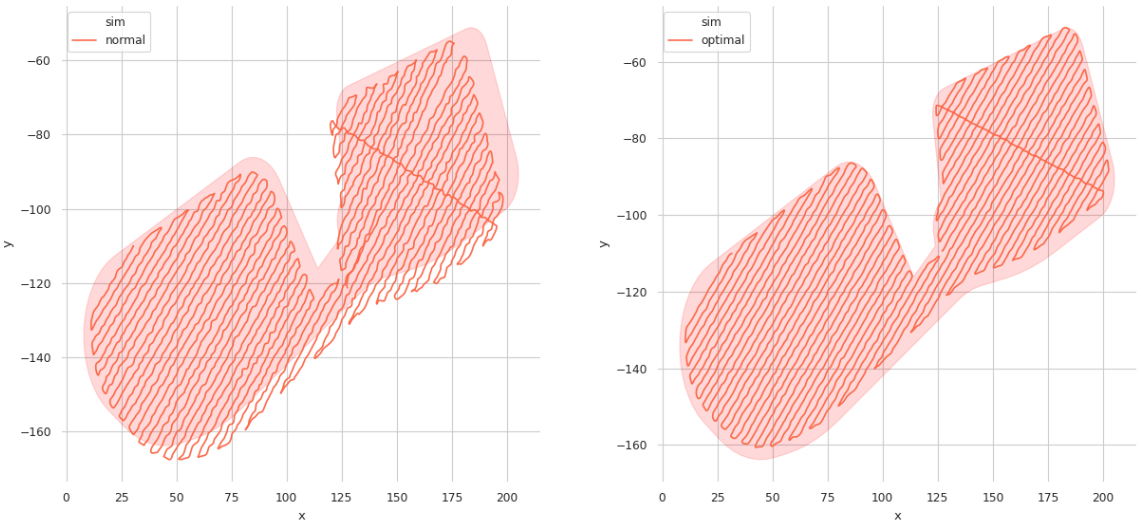


FIGURE 5.7: CPP WITH/WITHOUT KALMAN FILTERING

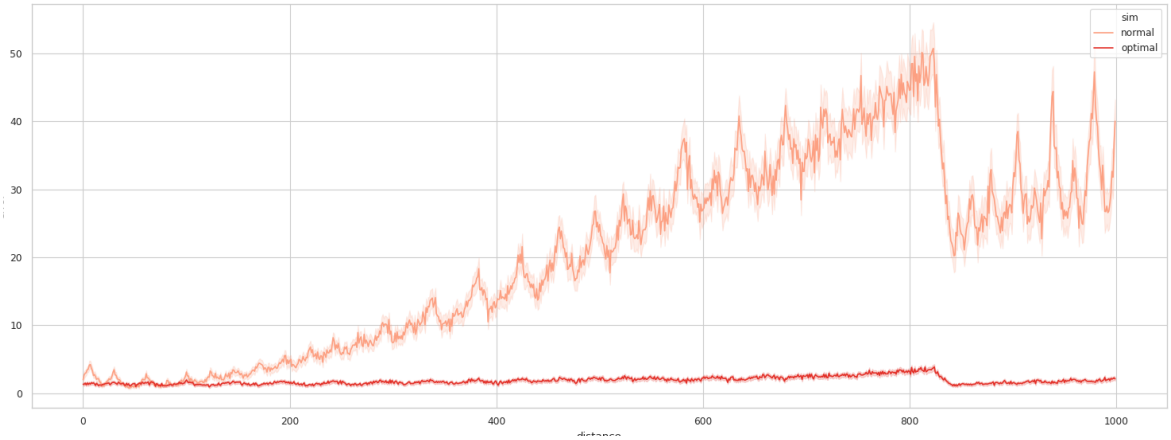


FIGURE 5.8: NEES OF THE CONTROLLER

reference point used by the Captain from which to track the traveled progress anew.

CHAPTER 6 CONCLUSION

To control something you have to know it. A controller for an autonomous operating crawler, encompasses more than implementing a simple state-machine, which tells it to go left or right. It starts with knowing how physics of its environment govern the state change of such a crawler. Only from this point can you go forth and actually determine how such as controller should behave.

This thesis answers how such a controller might behave for an Archimedes driven dredge crawler.

The crawler will operate in a challenging environment, namely underwater. The dampening effects of water are notorious. This is limiting factor in the choice for communication methods. Leaving only an umbilical as a viable choice. The greater challenge of this infamous dampening effect, is that is creates a Global Positioning System (GPS) deprived environment. Where land based vehicles have access to a mature, accurate and fast GPS network, underwater vessel need to resort to dead-reckoning techniques. Which are well-known for their growing error.

Sensor fusion can be a great tool to battle this effect. A distinction is made for sensors that aid in describing the state of the vehicle and those that describe the environment. Were detected landmarks can act as ground truths, anchoring the otherwise unbound error. State sensing is usually done with Inertial Measurement Units (IMUs), consisting of an accelerometer, gyroscope and magnetometer. These are also of use in the case of a crawler. Coupled with a pressure sensor, which measures the water column above the crawler, helps in an accurate estimation of the z position. This accuracy of the estimate will however degrade when dredging operations disturb the soil, which mixes with the water and alter the density.

Localization under uncertainty can be refined with the use of a Kalman filter, which in the case of the crawler is an Unscented Kalman Filter (UKF). Due to the inherent non-linear behavior of the kinematic model. A Kalman filter estimate where the crawler should be according to a state transition matrix F and a input vector $\vec{u}_k$. This estimate is updated with the measured values and a calculated error covariance matrix $\mathbf{P}_k$. Which will become more accurate over time. Filtering out noise and inaccuracies introduced due to sensor characteristics. If the state transistion model is accurate, the Kalman filter will have a great impact on the quality of the overal performance of the controller.

A crawler designed for dredging operations, has to cover an certain area, which might be unknown and can be changeing in time. Obstacles — such as ships — might move, and the crawler has to take that into account. These challenges are known as on-line Coverage Path Planning (CPP) problems. Three categories are explored in this thesis. Morse-based cellular decompositions, landmark-based topological coverage and grid-based methods. These coverage algorithms are in effect “divide and conquer” strategies, and decompose a complex space in sub-regions. Were each sub-regions is covered before moving to the next sub-region. After careful consideration a choice is made for the implementation of a topological Boustrophedeon Cellular Decomposition (topBCD). This is a landmark-based topological type of strategy which borrows the Boustrophedon Cellular Decomposition (BCD) approach from the Morse-based cellular decomposition strategy. BCD literally means “way-of-the-ox”, meaning that the crawler will go back and forth in the sub-regions, as an ox might plow a field.

Implementation of the UKF stands or fall with an accurate kinematic model. Estimating where the crawler will be requires a complete model of the drive-train. Which is model on first-principle, where the output of each component represent an energy transfer to the next. This drive-train actuates two rotating Archimedes screws, the terramechanic model describes how the rotation is actually transformed in a translation of the crawler on the soil bed. It proposes a novel method of estimating slippage by measuring an change in generated torque. Explaining that the blade of the Archimedes screw act as shovel when slippage occurs, resulting in an increase of torque.

A controller framework “ohCaptain” is written in C++, and released under the LGPL v3.0 license. It can run on Single Board Computer (SBC), ranging from expensive dedicated industrial devices, to cheap Raspberry pis (Rpis) or Beagle Bone Blacks (BBBs). Allowing future prototype to benefit from this projecty. Resulting in lower costs and shorter lead times. The framework is specifically developed

for maritime operations and is set-up in a modulair way. Allow for future expansion and specialized algortihms.

The research and design choices have been validated with the help of Project Chrono, a multi body physics simulation engine. This engine has dedicated modules for vehicles and terramechanics; It was however lacking a sensor simulation module. Which was needed to benchmark the performance of the UKF. An extension module was created and released under the MIT license, allowing for the simulation of realistic sensor measurements. Taking into account noise, discretization due to digitization, and transformation of the signal, simulation drift and hard-/soft-iron effects.

Simulating use-case 2, which is a coverage maintenance task of an existing harbor in Bruinisse the Netherlands, show that the chosen topBCD strategy is executed in a viable way. Dividing the harbor into three sub-regions, each completely covered with the path planner. The simulation gives also an insight into the performance of the propossed UKF. Comparing its performance to with a simple Proportional Integral Derivative (PID) controller. It shows a huge difference, were the PID conroller will derail quite early in its travels. Due to an inaccurate position estimation. The UKF controller is able to fully cover the harbor.

The controller shows great potential, but as it is with all research and development papers: “Further research is needed.”

,

7 CHAPTER NOMENCLATURE

SIGN	DESCRIPTION	UNIT	PAGE
a	acceleration	$\mathrm{m/s^2}$	13
A_a	working section of the screw conveyor	$\mathrm{m^2}$	54, 55
A_p	cross section of a pipe	$\mathrm{m^2}$	48–50
A_s	cross section area of in soil submerged volume	$\mathrm{m^2}$	48, 53
a_x	acceleration along the x-axis	$\mathrm{m/s^2}$	14
a_y	acceleration along the y-axis	$\mathrm{m/s^2}$	14
a_z	acceleration along the z-axis	$\mathrm{m/s^2}$	14, 22
$a_{z,k}$	acceleration along the z-axis at time k	$\mathrm{m/s^2}$	22, 23, 66
$a_{z,k-1}$	acceleration along the z-axis at time k-1	$\mathrm{m/s^2}$	22
α_e	attenuation	$\mathrm{dB/m}$	10
$\alpha_s f$	skin friction coefficient	–	54
B_d	slow chancing component of the signal; this is the gyroscope drift	$\mathrm{rad/s}$	15
$\mathbf{B}$	a control-input model which is applied to a control vector	–	21, 22
β	phase factor of a wave	–	10
C	cost a function	–	28
c	singel cell	–	32
c_{sb}	cohesion of a soil bed	Pa	48, 51, 52, 54, 55
c_t	volumetric concentration	–	57
c_u	undrained shear strength of soil	Pa	54, 55
C_{vb}	volumetric bed concentration	–	51
Cp	critical point in graph	–	29
Cp_0	critical point 0 in graph	–	29
Cp_1	critical point 1 in graph	–	29, 30
Cp_2	critical point 2 in graph	–	29, 30

SIGN	DESCRIPTION	UNIT	PAGE
Cp_3	critical point 3 in graph	–	29, 30
Cp_i	critical point ith in graph	–	29
d_a	sinkage depth of a Archimedes screw	m	48, 51–55
$D_{l,t}$	list of line segments	–	32
$D_{l,t}$	line segment at t	–	32
$D_{l,t-1}$	line segment at t-1	–	32
d_p	pipe diameter	m	51, 55, 56
$d_i(x)$	distance between x points and obstacle i	m	29
$\Delta d_{1,2}$	distance between two points	m	10
ΔF	difference in force	N	48
ΔL	length of a pipe	m	51, 56
Δp	pressureloss	Pa	18, 47–50
Δp_n	nominal pressureloss	Pa	48–50
δ_s	range of a sensor-detector-range	m	30, 31
Δt	time difference	s	22–24
$\Delta \tau$	torque difference	N m	47, 48
Δx	distance between nodes	m	32, 34
Δz	height difference	m	56
$\hat{E}$	amplitude of the electric field wave	V/m	9, 10
E_x	Is defined mathematically as a vector field that associates to each point in space the (electrostatic or Coulomb) force per unit of charge exerted on an infinitesimal positive test charge at rest at that point.	V/m	9, 10
ϵ_b	resistance coefficient of the bends, valves, hoses and other obstructions	–	56
ϵ_e	permittivity	–	9, 10
$\epsilon_{N,k}$	NEES at time k	–	24
$\bar{\epsilon}_N$	mean NEES value	–	24
ϵ_s	resistance coefficient of the suction inlet	–	56
$\eta_{V,n}$	nominal Volumetric efficiency	–	48–50

SIGN	DESCRIPTION	UNIT	PAGE
$\exp$	Eulers number	–	10
F	force	N	2, 13, 71
F_b	buoyancy force of a submerged Archimedes screw	N	48, 55
$F_{cr,b}$	buoyancy force of a submerged crawler	N	54
$\mathbf{F}$	a state transition model which is applied to the previous state	–	21, 22
F_n	normal force working on an Archimedes screw	N	51
F_r	Froude number for a pipeline	–	57
F_{rxd}	Froude number of the grains	–	57
F_s	skin friction	N	54
F_{sn}	specific normal force working on an Archimedes screw	$\mathrm{N/kg}$	48, 51, 52
F_u	ultimate skin friction	N	54
g	standard gravity model	$\mathrm{m^2/s}$	18, 22, 48, 51, 54–56
$\mathbf{G}$	adjacency graph	–	28, 34, 35
$\mathbf{G}_1$	adjacency subgraph 1	–	28
$\mathbf{G}_2$	adjacency subgraph 2	–	28
γ_e	propagation constant	m	9, 10
γ_m	specific weight of diluted water during dredging	$\mathrm{N/m^3}$	17, 18
γ_{sw}	specific weight of submerged soil	$\mathrm{N/m^3}$	17, 48, 51, 52, 55
γ_w	specific weight of water	$\mathrm{N/m^3}$	17, 18
h	height difference between radius of screw and its submerged part	m	53
h_{dh}	height of a dredge head	m	55
$\hat{H}$	amplitude of the magnetic field wave	A	9, 10
$\mathbf{H}$	measurement sensitivity matrix defining the linear relationship between state of the dynamic system and measurements that can be made	–	21–23
H_y	is a vector field that describes the magnetic influence of electric charges in relative motion and magnetized materials	$\mathrm{A/m}$	9, 10
I	moment of inertia	$\mathrm{kg\,m^2}$	48
i	imaginary unit	–	10

SIGN	DESCRIPTION	UNIT	PAGE
K_{HP}	Hagen-Poiseuille coefficient for laminar pipe flows	–	48–50
K_{TP}	friction torque vs pressure gain coefficient parameter	–	48, 50
$\bar{\mathbf{K}}$	Kalman gain matrix	–	21–23
L	Distance between the crawlers screws	m	58, 59
l_a	length of an Archimedes screw	m	48, 53, 55
$L_{\alpha,\epsilon}$	electromagnetic signal strength	dB	10
λ	friction factor of a straight pipe	–	56
m	mass	kg	13
m_{cr}	mass of a crawler	kg	48, 54
$m_{cr}^{'}$	total buoyancy corrected mass of a crawler	kg	54
m_f	mass	kg	48
$\vec{m}_{hi}$	hard iron adjusted vector	T	16, 17
m_n	number of neighbouring neurons	–	40
$\vec{m}_{si}$	soft iron adjusted vector	T	17
$\vec{m}$	raw magnetometer vector	T	16
μ	dynamic viscosity	$\mathrm{Pa\,s}$	48–50, 56
μ_e	electromagnetism permeability	$\mathrm{H/m}$	9, 10
μ_n	nominal dynamic viscosity	$\mathrm{Pa\,s}$	48–50
$n_{a,l}$	neural activity of a the lth neuron	–	40
N_c	dimensionless constant used in the Brinch-Hansen equation, related to the cohesion	–	48, 52
N_γ	dimensionless constant used in the Brinch-Hansen equation, related to the specific weight of the soil	–	48, 52, 53
N_ϕ	dimensionless factor of the internal friction angle	–	55
N_q	dimensionless constant used in the Brinch-Hansen equation, related to the load of the surrounding soil	–	48, 52, 53
n_s	stochastic component of a signal	$\mathrm{rad/s}$	15
n_x	number of elements in the state vector	–	24
∇_m	normal of a surface	m	29
ω	angular velocity	$\mathrm{rad/s}$	10, 14, 15, 44, 47–50, 58

SIGN	DESCRIPTION	UNIT	PAGE
ω_n	nominal angular velocity	$\mathrm{rad/s}$	48–50
ω_t	true angular velocity	$\mathrm{rad/s}$	15
ω_{th}	angular velocity threshold	$\mathrm{rad/s}$	48, 50
P	power	W	48
p	pressure	Pa	17, 18, 47, 50
$\mathbf{P}_0$	initialization covariance matrix of state estimation uncertainty	–	21
p_a	atmospheric pressure	Pa	17
p_{atm}	atmospheric pressure	Pa	57
p_i	pressureloss at the suction inlet	Pa	56, 57
$\mathbf{P}_k$	covariance matrix of state estimation uncertainty	–	21–24, 71
$\vec{p}_k$	pose components of the state vector xk	m	40, 45
$\mathbf{P}_{k-1}$	covariance matrix of a priori state estimation uncertainty	–	21, 22
$\vec{p}_{k+1}$	AUV location at time k + 1	m	40
$\vec{p}_{k+1}$	pose components of the state vector xk+1	m	45, 46
p_{man}	manometric head of the dredge pump	Pa	57
p_p	absolute pressure at the outlet of the dredge pump	Pa	57
p_q	pressure generated at a certain depth due to the soil on top of it	Pa	48, 52
p_{ro}	resistance offered by bends, valves, hoses and other obstructions	Pa	56, 57
p_{rp}	resistance of the straightline pipe	Pa	56, 57
p_s	absolute pressure at the entrance of impeller	Pa	57
p_{sb}	load working on a soil bed	Pa	48, 51, 52
p_{sm}	static head in the suction pipe	Pa	56, 57
p_{ss}	static pressure generated by surrounding water	Pa	57
p_v	pressure of water vapor in ambient air	Pa	56, 57
ϕ_c	roll of the crawler on the Euclidean y-axis	rad	45
ϕ_{IMU}	roll of the IMU on the Euclidean y-axis	rad	13, 14, 16
ϕ_s	angle of internal friction	rad	48, 52, 53, 55

SIGN	DESCRIPTION	UNIT	PAGE
ψ	yaw of the crawler on the Euclidean z-axis	rad	56, 58, 59
ψ_c	yaw of the crawler bot on the Euclidean z-axis	rad	45
ψ_{IMU}	yaw of the IMU bot on the Euclidean z-axis	rad	14, 16
ψ_m	Durand-Condolios mixture correction	–	56, 57
Q	volumetric fluid flow	m^3/s	48–50, 55–57
$\vec{q}$	heat flow	–	45
Q_i	ideal volumetric fluid flow	m^3/s	48–50
$\mathbf{Q}_k$	covariance matrix of process estimation uncertainty	–	21, 22
Q_l	volumetric fluid leakage flow	m^3/s	48–50
q_s	real component of a quaternion	–	45
q_{si}	smallest magnitude of a point on the ellipse, and thus the vector of the B-axis	T	17
q_x	imaginary x-axis of a quaternion	–	45
q_y	imaginary y-axis of a quaternion	–	45
q_z	imaginary z-axis of a quaternion	–	45
r	radius	m	30
r_0	the receptive field radius of the kth neuron	m	39, 40
r_a	radius of Archimedes screw	m	48, 52–54
$\mathbf{R}$	covariance matrix of state estimation uncertainty	–	21, 23
r_{si}	greatest magnitude of a point on the ellipse, and thus the vector of the A-axis	T	17
R_t	radius from ICR ro COG of crawler	m	58
Re	Reynolds Number	–	56
Re_{sd}	relative submerged density	–	51
ρ	density of a material	kg/m^3	48–50, 57
ρ_{ds}	density of drained soil	kg/m^3	48, 51
ρ_{is}	in-situ density of soil	kg/m^3	48, 51, 53
ρ_m	density of a mixture	kg/m^3	18, 56, 57
ρ_n	nominal density of a material	kg/m^3	48–50

SIGN	DESCRIPTION	UNIT	PAGE
ρ_w	density of water	$\mathrm{kg/m^3}$	18, 48, 51, 53, 56, 57
S	a movement in a graph	–	28
s_a	soil contact arc length	m	48, 53–55
s_c	shape factor used in the Brinch-Hansen equation, related to the cohesion	–	48, 52, 53
s_γ	shape factor used in the Brinch-Hansen equation, related to the specific weight of the soil	m	48, 52, 53
s_q	shape factor used in the Brinch-Hansen equation, related to the surrounding load of the soil	–	48, 52, 53
s_z	position along the z-axis	m	21, 22, 25
$s_{z,k}$	position along the z-axis at time k	m	22
$s_{z,k-1}$	position along the z-axis at time k-1	m	22, 23
$s_{z,m}$	measured position along the z-axis	m	22, 23
$s_{z,m,k}$	measured position along the z-axis at time k	m	22, 23
σ_e	Electrical conductivity	$\mathrm{S/m}$	10–12
σ_H	horizontal stress	Pa	52, 55
$\sigma^{'}$	effective stress	Pa	54
σ_s	deviation on position	m	23, 24
$\sigma_{s,m}$	deviation on measured position	m	23, 24
σ_V	vertical stress	Pa	52, 55
σ_v	deviation on velocity	$\mathrm{m/s}$	23, 24
$\sigma_{v,m}$	deviation measured on velocity	$\mathrm{m/s}$	23, 24
T	temperature	K	15, 17
t	time	s	10, 15, 25, 32, 45
$T^{'}$	rate of temperature variation	$\mathrm{K/s}$	15
τ	torque	$\mathrm{N\,m}$	44, 47–50
τ_0	initial torque	$\mathrm{N\,m}$	48, 50
τ_f	torque due to friction	$\mathrm{N\,m}$	48–50
τ_i	ideal torque	$\mathrm{N\,m}$	48–50
θ	angle on the x-axis	rad	15, 57

SIGN	DESCRIPTION	UNIT	PAGE
θ_b	pitch of the dredge bot on the Euclidean z-axis	rad	45
θ_c	pitch of the crawler on the Euclidean x-axis	rad	45
θ_{IMU}	pitch of the IMU on the Euclidean x-axis	rad	13, 14, 16
θ_s	sink angle of an Archimedes screw	rad	48, 51–54
θ_{si}	soft iron axis offset along the x-axis	rad	17
$\vec{u}$	unit vector	–	27, 44
$\vec{u}_k$	control inputs	–	2, 21, 22, 43, 45, 46, 71, III
v	specific volume	m^3/kg	58
v_f	velocity of a fluid at a certain point	$\mathrm{m/s}$	48–50, 55, 56
$\vec{v}_k$	measurement noise	–	21, 22, 55, 57
v_L	translational velocity of left Archimedes screw	$\mathrm{m/s}$	58, 59
v_R	translational velocity of right Archimedes screw	$\mathrm{m/s}$	58, 59
V_{sa}	submerged volume of a Archimedes screw	m^3	48, 53, 55
v_z	velocity along the z-axis	$\mathrm{m/s}$	21, 22, 25
$v_{z,k}$	velocity along the z-axis at time k	$\mathrm{m/s}$	22
$v_{z,k-1}$	velocity along the z-axis at time k-1	$\mathrm{m/s}$	22, 23
w_a	width of the Archimedes screw in contact with the soil	m	48, 52, 53
w_{dh}	width of a dredge head	m	55
$\vec{w}_k$	process noise	–	21, 22
x	location along the x-axis	m	29, 38, 44, 45, 59
x_h	hard iron distortion along the x-axis	T	16
$\vec{x}_k$	state vector describing the state of a system at the kth component of x	–	2, 13, 20–23, 44–46, III
$\vec{x}_0$	state vector describing the initial state of a system at the kth component of x	–	21
$\vec{x}_{g,k}$	ground truth state, describing the real state of a system kth component of x	–	24
$\hat{x}_k$	an estimation of the state vector x	–	21–24
$\vec{x}_{k+1}$	a state estimate of x, conditioned on all available measurements at time tk	–	44, III
$\vec{x}_{k-1}$	a priori state of xk, conditioned on all prior measurements, except the one at time tk	–	21

SIGN	DESCRIPTION	UNIT	PAGE
$\tilde{x}_k$	Error of the state of a system at the kth component of x	–	24
x_m	raw magnetometer value along the x-axis	T	16
y	location along the y-axis	m	38, 44, 45, 59
y_1	y-index of greatest magnetic magnitude	T	17
y_h	hard iron distortion along the y-axis	T	16
$\vec{y}_k$	measured values	–	21, 22, 44, 66–68
$\vec{y}_k^{'}$	transformed measured values	–	66–68
y_l	a gls-monotonically increasing function of the difference between the next moving directions	m	40
y_m	raw magnetometer value along the y-axis	T	16
z	height	m	17, 44, 45, 71
z_ϵ	height error	m	18
$\vec{z}_k$	measured values mapped to the state space	–	21–23
z_m	raw magnetometer value along the z-axis	T	16
z_p	height of the pressure sensor with regards to the soil bed	m	18

CHAPTER 8 GLOSSARY

KEY	DESCRIPTION	PAGE
accelerometer	a device that measures proper acceleration	13–15, 20, 43, 63, 66
time of flight acoustic navigation	triangulation of a position using the difference in send and receive time of signal, to calculate the distance from a source	20
adjacency graph	a graph representing depicting all the nodes	28, 30
Archimedes screw	a machine historically used for transferring water from a low-lying body of water into irrigation ditches. The same principle can also be used propel the screw in a medium.	43
bandwidth	a difference between the upper and lower frequencies in a continuous set of frequencies	11
bathymetric map	submerged equivalent of an above-water topographic map	20
bitstream	a sequential binary sequence	6
Chrono Sensors	an extension library for Project Chrono, which allows simulation of realistic sensor values.	68
Coriolis effect	an inertial or fictitious force that acts on objects that are in motion within a frame of reference that rotates with respect to an inertial frame.	14
coverage path	a sequence of steps which coveres a whole area by following a certain path	26, 28, 31, 36
critical point	a value of average degree, which separates networks	27–30
cusp point	are points where its surface normal of the boundary of the free configuration space is non-smooth	30
deadlock state	a standstill situation, from which the algorithm has no means of escape	37
dead-reckoning	the process of calculating one's current position by using a previously determined position, or fix, and advancing that position based upon known or estimated speeds over elapsed time and course	14, 19, 20, 30
Dempster Shafer theory	theory of belief functions, also referred to as evidence theory or Dempster-Shafer theory (DST), is a general framework for reasoning with uncertainty	41
dilatancy	volume increase that may occur during shear	51
draghead	a suction mouth which is dragged across a water body	4, 55
dredgeline	a pipeline which transports excavated slurry	1, 6, 7, 55, 68

KEY	DESCRIPTION	PAGE
electric field	a vector field that associates to each point in space the Coulomb force that would be experienced per unit of electric charge, by an infinitesimal test charge at that point. Electric fields converge and diverge at electric charges and can be induced by time-varying magnetic fields	9
erosion	an action of surface processes (such as water flow or wind) that removes soil	2
geophysical navigation	navigation using landmarks	20
gimbal lock	the loss of one degree of freedom in a three-dimensional space, three-gimbal mechanism that occurs when the axes of two of the three gimbals are driven into a parallel configuration, “locking” the system into rotation in a degenerate two-dimensional space	45
gyroscope	a spinning wheel or disc in which the axis of rotation is free to assume any orientation by itself. When rotating, the orientation of this axis is unaffected by tilting or rotation of the mounting, according to the conservation of angular momentum. Because of this, gyroscopes are useful for measuring or maintaining orientation	13–15, 20, 43, 63, 66, 69
hard iron	a constant additive disturbance in the magnetic field of the magnetometer	16, 17, 66, 67
hopper	a storage container or compartment	3, 4
Kalman filter	an algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more precise than those based on a single measurement alone	6, 13, 14, 20–23, 25, 43, 44
Kalman gain	the relative weight given to the measurements and current state estimate, and can be "tuned" to achieve particular performance. With a high gain, the filter places more weight on the most recent measurements, and thus follows them more responsively. With a low gain, the filter follows the model predictions more closely. At the extremes, a high gain close to one will result in a more jumpy estimated trajectory, while low gain close to zero will smooth out noise but decrease the responsiveness	22, 23
LoRa	a wireless technology that has been developed to enable low data rate communications to be made over long distances by sensors and actuators for M2M and Internet of Things, IoT applications	11
Lorentz force	the combination of electric and magnetic force on a point charge due to electromagnetic fields	15
magnetic field	a magnetic effect of electric currents and magnetic materials. The magnetic field at any given point is specified by both a direction and a magnitude (or strength); as such it’s a vector field	9

KEY	DESCRIPTION	PAGE
magnetometer	a magnetometer is an instrument that measures magnetism, either magnetization of magnetic material like a ferromagnet, or the strength and, in some cases, direction of the magnetic field at a point in space	13, 15, 16, 20, 43, 63, 67
Maxwell's equation	are a set of partial differential equations that, together with the Lorentz force law, form the foundation of classical electrodynamics, classical optics, and electric circuits	9
monotonically	a function between ordered sets that preserves or reverses the given order	40
Morse function	a function for which all critical points are non-degenerate and all critical levels are different	27, 28
NARROW-cell	a cell is located in a narrow space, bound between multiple walls	30
non-commutative	a operation in which the order of the operands determine the results	45
odometry	the use of data from motion sensors to estimate change in position over time	14
off-line	algorithm which plans an optimal path ahead of time, thus which needs to know the environment a priori	20, 26, 28, 42
ohCaptain	a versatile efficient controller framework for the maritime industry	68
on-line	an algorithm which has the ability to adapt when needed	26, 33, 42, 63, 71
optimal path	a sequence of steps which are optimized	26, 28
polarized plane	is a confinement of the electric field vector or magnetic field vector to a given plane along the direction of propagation	9
pressure sensor	can be classified in terms of pressure ranges they measure, temperature ranges of operation, and most importantly the type of pressure they measure	13, 17, 18, 20, 43, 45, 63, 67
Project Chrono	a physics-based modelling and simulation infrastructure based on a platform-independent open-source design implemented in C++	65, 66, 68
quaternion	a number system that extends the complex numbers, they are very useful in describing a rotation involving three dimensions	20, 45
Reeb graph	a mathematical object reflecting the evolution of the level sets of a real-valued function on a manifold	30
sedimentation	the opposite of erosion	2
silt	a granular material of a size between sand and clay	2
slurry	describe a mixture that consist of both solid and fluid phases	3, 4, 56
soft iron	a result of material that distorts the magnetic field of magnetometer, but does not necessarily generate its own magnetic field	16, 67
Trapezoidal rule	a technique for approximating the definite integral	14

KEY	DESCRIPTION	PAGE
traveling salesman	asks the following question: “Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?” it’s an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science	28
umbilical	a electronic cable connecting an underwater vehicle	1, 6, 7, 33, 68, 78
VAST-cell	a cell located in a vast open space	30
Voronoi diagram	a partition of a plane into regions close to each of a given set of objects	26
word	a string of bits representing a value which is stored in memory	15

CHAPTER 9 ACRONYMS

KEY	DESCRIPTION	PAGE
ADC	Analog Digital Conversion	14, 15
ASD	Auger Suction Dredger	4
AUV	Autonomous Underwater Vehicle	7, 19, 45
AW	Acoustic Waves	12
BBB	Beagle Bone Black	59, 65, 71, IV
BCD	Boustrophedon Cellular Decomposition	27, 29, 30, 63, 71, III
BCD-GVD	Morse-based Cellular Decomposition with Voronoi Diagram	42
CIP	Common Industrial Protocol	8
CPP	Coverage Path Planning	2, 6, 26, 42, 43, 63, 68, 71, III
CSD	Cutter Suction Dredger	4
DEM	Discrete Element Method	65
EMW	Electromagnetic Waves	8–10, 12, 19
GPS	Global Positioning System	2, 6, 13, 14, 20, 43, 71, III
GUI	Graphical User Interface	59
GVD	Generalized Voronoi Diagram	30
IEEE	Institute of Electrical and Electronics Engineers	8, 10, 11
IHC	Royal IHC	1, 59
IMU	Inertial Measurement Unit	13, 60, 71, III
IoT	Internet of Things	11
LBL	Long Base Line	20
LLC	Logical Link Control	8, 10
LQE	Linear Quadratic Estimation	20, III
LQG-MP	Linear-Quadratic Gaussian Motion Planning	20

KEY	DESCRIPTION	PAGE
MAC	Media Access Control	8, 10, 11
MEMS	Micro Electro Mechanical System	13–15
MLP	Multi-Layer Perceptron	36
MTI	IHC MTI B.V.	1, 17, 59
NEES	Normalized Estimated Error Squared	24, 25, 69
NN	Neural Network based Coverage in Grid maps	42
onBCD	online Morse-based Boustrophedon Cellular Decomposition	42
onSD2	on-line Slice Decomposition II	42
onSD2NN	on-line Slice Decomposition II with Neural Network	42
OOP	Object-Orientated Programming	65
PID	Proportional Integral Derivative	43, 59, 64, 68, 69, 72, IV
PLC	Programmable Logic Controller	59
RESL	Robotic Embedded Systems Laboratory	11
ROV	Remote Operated Vehicle	7
Rpi	Raspberry pi	1, 59, 71, IV
RRT	Rapidly exploring Random Trees	20
SBC	Single Board Computer	1, 59, 65, 71, IV
SCM	Soil Contact Model	65, 68
SLAM	Simultaneous Localization And Mapping	20
Spiral-STC	Spiral Spanning Tree Coverage	38
STC	grid based Spanning Tree Coverage	42
SW	Sound Waves	12
SWR	Submerged Weight Range	46
TCP	Transmission Control Protocol	8
TIR	Total Internal Reflection	6
topBCD	topological Boustrophedeon Cellular Decomposition	42, 71, 72, III, IV
TRL	Technological Readiness Level	1

KEY	DESCRIPTION	PAGE
TRN	Terrain Relative navigation	20
TSHD	Trailing Suction Hopper Dredger	4
UDP	User Datagram Protocol	8
UKF	Unscented Kalman Filter	2, 44, 45, 63, 69, 71, 72, III, IV
USBL	Ultra Short Base Line	20
UT	Unscented Transformation	44
YAML	YAML Ain't Markup Language	64

10 CHAPTER BIBLIOGRAPHY

[1] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve", *The Journal of physiology*, vol. 117, no. 4, p. 500, 1952. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/pmc1392413/ (visited on 05/10/2016).

[2] R. E. Kalman, "A new approach to linear filtering and prediction problems", *Journal of basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960. [Online]. Available: http://fluidsengineering.asmedigitalcollection.asme.org/article.aspx?articleid=1430402 (visited on 01/02/2017).

[3] I. S. Bogie, "Conduction and magnetic signalling in the sea a background review", *Radio and Electronic Engineer*, vol. 42, no. 10, pp. 447–452, 1972. [Online]. Available: http://digital-library.theiet.org/content/journals/10.1049/ree.1972.0076 (visited on 08/28/2016).

[4] International Standard ISO, *NEN-ISO 7119:1982 Continuous mechanical handling equipment for loose bulk materials - screw conveyors - Design rules of drive power*. International Standard ISO, 1981.

[5] A. Westneat, D. Blidberg, and R. Corell, "Advances in unmanned untethered underwater vehicles", *Unmanned Systems*, vol. 1, no. 3, pp. 8–13, 1983.

[6] Z. L. Cao, Y. Huang, and E. L. Hall, "Region filling operations with random obstacle avoidance for mobile robots", en, *Journal of Robotic Systems*, vol. 5, no. 2, pp. 87–102, Apr. 1988, issn: 1097-4563. doi: 10.1002/rob.4620050202. [Online]. Available: http://onlinelibrary.wiley.com.wiley.stcproxy.han.nl/doi/10.1002/rob.4620050202/abstract (visited on 04/19/2016).

[7] J. Borenstein, *Where am I? Sensors and methods for mobile robot positioning*. Apr. 1996.

[8] K. P. Valavanis, D. Gracanin, M. Matijasevic, R. Kolluru, and G. A. Demetriou, "Control architectures for autonomous underwater vehicles", *IEEE Control Systems*, vol. 17, no. 6, Dec. 1997, issn: 1066-033X. doi: 10.1109/37.642974.

[9] H. Choset and P. Pignon, "Coverage Path Planning: The Boustrophedon Cellular Decomposition", en, in *Field and Service Robotics*, A. Zelinsky, Ed., London: Springer London, 1998, pp. 203–209, isbn: 978-1-4471-1275-4. [Online]. Available: http://link.springer.com/10.1007/978-1-4471-1273-0_32 (visited on 04/17/2016).

[10] Joseph C. Palais, *Fibre Optic Communication*. Prentice Hall, 1998.

[11] S. Thrun, "Learning metric-topological maps for indoor mobile robot navigation", *Artificial Intelligence*, vol. 99, no. 1, pp. 21–71, Feb. 1998, issn: 0004-3702. doi: 10.1016/S0004-3702(97)00078-7. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370297000787 (visited on 04/17/2016).

[12] VBKO Vereniging van waterbouwers in bagger-, kust- en oeverwerken, *Voortgezette Opleiding Uitvoering Baggerwerken*, nl. Leidschendam: VBKO Vereniging van waterbouwers in bagger-, kust- en oeverwerken, 1998, isbn: 90-90-11108-5.

[13] C. K. Chui and G. Chen, *Kalman Filtering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, isbn: 978-3-540-64611-2. [Online]. Available: http://link.springer.com/10.1007/978-3-662-03859-8 (visited on 12/31/2016).

[14] J. G. Webster, *The Measurement, Instrumentation, and Sensors: Handbook*, en. Springer Science & Business Media, 1999, Google-Books-ID: b7UuZzf9ivIC, isbn: 978-3-540-64830-7.

[15] H. Choset, "Coverage of Known Spaces: The Boustrophedon Cellular Decomposition", *Autonomous Robots*, vol. 9, no. 3, pp. 247–253, 2000, issn: 09295593. doi: 10.1023/A:1008958800904. [Online]. Available: http://link.springer.com/10.1023/A:1008958800904 (visited on 04/17/2016).

[16] H. Choset, E. Acar, A. A. Rizzi, and J. Luntz, "Exact cellular decompositions in terms of critical points of morse functions", in *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, vol. 3, IEEE, 2000, pp. 2270–2277. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=846365 (visited on 04/19/2016).

[17] L. L. Whitcomb, "Underwater robotics: Out of the research laboratory and into the field", in *IEEE International Conference on Robotics and Automation, 2000. Proceedings. ICRA '00*, vol. 1, 2000. doi: 10.1109/ROBOT.2000.844135.

[18] E. U. Acar, H. Choset, and P. N. Atkar, "Complete sensor-based coverage with extended-range detectors: A hierarchical decomposition in terms of critical points and voronoi diagrams", in *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, vol. 3, IEEE, 2001, pp. 1305–1311. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=977163 (visited on 04/19/2016).

[19] H. Choset, "Coverage for robotics – A survey of recent results", *Annals of Mathematics and Artificial Intelligence*, vol. 31, no. 1/4, pp. 113–126, 2001, issn: 10122443. doi: 10.1023/A:1016639210559. [Online]. Available: http://link.springer.com/10.1023/A:1016639210559 (visited on 04/17/2016).

[20] W. H. Huang, "Optimal line-sweep-based decompositions for coverage algorithms", in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 1, IEEE, 2001, pp. 27–32. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=932525 (visited on 04/19/2016).

[21] E. U. Acar and H. Choset, "Sensor-based coverage of unknown environments: Incremental construction of morse decompositions", *The International Journal of Robotics Research*, vol. 21, no. 4, pp. 345–366, 2002. [Online]. Available: http://ijr.sagepub.com/content/21/4/345.short (visited on 04/19/2016).

[22] E. U. Acar, H. Choset, A. A. Rizzi, P. N. Atkar, and D. Hull, "Morse decompositions for coverage tasks", *The International Journal of Robotics Research*, vol. 21, no. 4, pp. 331–344, 2002. [Online]. Available: http://ijr.sagepub.com/content/21/4/331.short (visited on 04/19/2016).

[23] C. Luo, S. X. Yang, D. A. Stacey, and J. C. Jofriet, "A solution to vicinity problem of obstacles in complete coverage path planning", in *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, vol. 1, IEEE, 2002, pp. 612–617. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1013426 (visited on 05/09/2016).

[24] B. G. Liptak, *Instrument Engineers' Handbook, Fourth Edition, Volume One: Process Measurement and Analysis*. CRC Press, Jun. 27, 2003, 1914 pp., isbn: 978-1-4200-6402-5.

[25] Z. Feng and R. Allen, "Evaluation of the effects of the communication cable on the dynamics of an underwater flight vehicle", *Ocean Engineering*, vol. 31, Jun. 2004, issn: 0029-8018. doi: 10.1016/j.oceaneng.2003.11.001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002980180400006X.

[26] Y. Mei, Y.-H. Lu, Y. C. Hu, and C. G. Lee, "Energy-efficient motion planning for mobile robots", in *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, vol. 5, IEEE, 2004, pp. 4344–4349. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1302401 (visited on 05/09/2016).

[27] S. C. Wong and B. A. MacDonald, "Complete coverage by mobile robots using slice decomposition based on natural landmarks", in *PRICAI 2004: Trends in Artificial Intelligence*, Springer, 2004, pp. 683–692. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-28633-2_72 (visited on 04/19/2016).

[28] I. F. Akyildiz, D. Pompili, and T. Melodia, "Underwater acoustic sensor networks: Research challenges", *Ad Hoc Networks*, vol. 3, no. 3, pp. 257–279, May 2005, issn: 1570-8705. doi: 10.1016/j.adhoc.2005.01.004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570870505000168 (visited on 08/26/2016).

[29] H. M. Choset, *Principles of Robot Motion: Theory, Algorithms, and Implementation*, en. MIT Press, 2005, isbn: 978-0-262-03327-5.

[30] Edward Tucholski, *Underwater Acoustics and Sonar - SP411*. 2006.

[31] F. J. L. J. S. Wei, J. C. Fang, and J. L. Li, "Improved temperature error model of silicon MEMS gyroscope with inside frame driving [J]", *Journal of Beijing University of Aeronautics and Astronautics*, vol. 11, p. 004, 2006. [Online]. Available: http://en.cnki.com.cn/Article_en/CJFDTotal-BJHK200611004.htm (visited on 12/28/2016).

[32] g. Welch and G. Bishop, "An introduction to the Kalman Filter", en, Jul. 2006.

[33] S. Wong, "Qualitative topological coverage of unknown environments by mobile robots", PhD thesis, ResearchSpace Auckland, 2006. [Online]. Available: https://researchspace.auckland.ac.nz/handle/2292/619 (visited on 04/25/2016).

[34] M.-C. Fang, C.-S. Hou, and J.-H. Luo, "On the motions of the underwater remotely operated vehicle with the umbilical cable effect", *Ocean Engineering*, vol. 34, Jun. 2007, issn: 0029-8018. doi: 10.1016/j.oceaneng.2006.04.014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0029801806001880.

[35] D. Green and R. Perry, *Perry's Chemical Engineers' Handbook, Eighth Edition*, ser. McGraw Hill professional. McGraw-Hill Education, 2007, isbn: 978-0-07-159313-7. [Online]. Available: https://books.google.nl/books?id=tH7IVcA-MX0C.

[36] L. Nicolaescu, *An Invitation to Morse Theory*, en, ser. Universitext. New York, NY: Springer New York, 2007, isbn: 978-0-387-49509-5. [Online]. Available: http://link.springer.com/10.1007/978-0-387-49510-1 (visited on 04/19/2016).

[37] L. Tetley and D. Calcutt, *Electronic Navigation Systems*, en. Routledge, Jun. 2007, Google-Books-ID: mybjgp2gGRsC, isbn: 978-1-136-40724-6.

[38] A. Verruijt and S. Van Baars, *Soil mechanics*. VSSD, 2007. (visited on 05/30/2016).

[39] C. Konvalin, *Technical document: Compensating for tilt, hard iron and soft iron effects*, EN, 2008. [Online]. Available: http://www.sensorsmag.com/sensors/motion-velocity-displacement/compensating-tilt-hard-iron-and-soft-iron-effects-6475 (visited on 12/30/2016).

[40] L. Lanbo, Z. Shengli, and C. Jun-Hong, "Prospects and problems of wireless communication for underwater sensor networks", *Wireless Communications and Mobile Computing*, no. 8, 2008. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/wcm.654/abstract (visited on 08/27/2016).

[41] C. Luo and S. X. Yang, "A bioinspired neural network for real-time concurrent map building and complete coverage robot navigation in unknown environments", *Neural Networks, IEEE Transactions on*, vol. 19, no. 7, pp. 1279–1298, 2008. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4539807 (visited on 05/16/2016).

[42] Training Institute for Dredging, *Ingewijden Training*, en. MTI Holland, 2008.

[43] A. Bahr, J. J. Leonard, and M. F. Fallon, "Cooperative localization for autonomous underwater vehicles", *The International Journal of Robotics Research*, vol. 28, no. 6, pp. 714–728, 2009. (visited on 05/20/2016).

[44] Hagman, Elias, "Design of a High Speed, Short Range Underwater Communication System", PhD thesis, Eidgenossische Technische Hochschule Zurich, 2009.

[45] R. Lotman, "Applicable theory for the modeling of the propulsion system of the SMT", Dec. 9, 2009.

[46] A. van der Zee, "Prediction of the terramechanic performance of a SMT", IHC Dredgers B.V., Dec. 18, 2009, p. 113.

[47] M. Ainslie, *Principles of Sonar Performance Modelling*, en. Springer Science & Business Media, Sep. 2010, isbn: 978-3-540-87662-5.

[48] S. Tully, G. Kantor, and H. Choset, "Leap-frog path design for multi-robot cooperative localization", in *Field and service robotics*, Springer, 2010, pp. 307–317. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-13408-1_28 (visited on 04/18/2016).

[49] M. Garcia, S. Sendra, M. Atenas, and J. Lloret, "Underwater wireless ad-hoc networks: A survey", *Mobile ad hoc networks: Current status and future trends*, 2011. (visited on 08/27/2016).

[50] s. Jiang and S. Georgakopoulos, "Electromagnetic Wave Propagation into Fresh Water", en, Apr. 2011.

[51] C. Kownacki, "Optimization approach to adapt Kalman filters for the real-time application of accelerometer and gyroscope signals' filtering", *Digital Signal Processing*, vol. 21, no. 1, pp. 131–140, Jan. 2011, issn: 1051-2004. doi: 10.1016/j.dsp.2010.09.001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1051200410001843 (visited on 12/29/2016).

[52] T.-K. Lee, S.-H. Baek, Y.-H. Choi, and S.-Y. Oh, "Smooth coverage path planning and control of mobile robots based on high-resolution grid map representation", *Robotics and Autonomous Systems*, vol. 59, no. 10, pp. 801–812, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0921889011000996 (visited on 05/10/2016).

[53] R. Lotman and M. A. Grima, "Deep Sea Mining With an Archimedes Screw Driven Vehicle", in *ASME 2011 30th International Conference on Ocean, Offshore and Arctic Engineering*, American Society of Mechanical Engineers, 2011.

[54] J. R. Nistler and M. F. Selekwa, "Gravity compensation in accelerometer measurements for robot navigation on inclined surfaces", *Procedia Computer Science*, Complex adaptive sysytems, vol. 6, pp. 413–418, Jan. 2011, issn: 1877-0509. doi: 10.1016/j.procs.2011.08.077. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050911005424 (visited on 12/27/2016).

[55] R. A. Rajapakse, *Geotechnical Engineering Calculations and Rules of Thumb*. Butterworth-Heinemann, Apr. 2011, isbn: 978-0-08-055903-2.

[56] F. M. White, *Fluid Mechanics*, en. McGraw Hill, 2011, isbn: 978-0-07-352934-9.

[57] M. C. Domingo, "An overview of the internet of underwater things", *Journal of Network and Computer Applications*, vol. 35, no. 6, pp. 1879–1890, Nov. 2012, issn: 1084-8045. doi: 10.1016/j.jnca.2012.07.012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804512001646 (visited on 08/26/2016).

[58] E. Galceran and M. Carreras, "Coverage path planning for marine habitat mapping", in *Oceans, 2012*, IEEE, 2012, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6404907 (visited on 04/25/2016).

[59] J. Lloret, S. Sendra, M. Ardid, and J. J. P. C. Rodrigues, "Underwater Wireless Sensor Communications in the 2.4 GHz ISM Frequency Band", *Sensors (Basel, Switzerland)*, vol. 12, no. 4, Mar. 2012, issn: 1424-8220. doi: 10.3390/s120404237. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3355409/.

[60] S. Ramakrishna and I. Nissen, "Next generation cognitive system approaches in the underwater communication area", *Applied Ocean Research*, vol. 38, pp. 136–141, Oct. 2012, issn: 0141-1187. doi: 10.1016/j.apor.2012.07.007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141118712000685 (visited on 08/26/2016).

[61] M. Yan, D. Zhu, and S. X. Yang, "Complete coverage path planning in an unknown underwater environment based on DS data fusion real-time map building", *International Journal of Distributed Sensor Networks*, vol. 2012, 2012. [Online]. Available: http://www.hindawi.com/journals/ijdsn/2012/567959/abs/ (visited on 05/10/2016).

[62] B. d'Andréa-Novel and M. D. Lara, *Control Theory for Engineers: A Primer*, en. Springer Science & Business Media, May 2013, Google-Books-ID: gWpCSlBTNr8C, isbn: 978-3-642-34324-7.

[63] E. Galceran and M. Carreras, "A survey on coverage path planning for robotics", *Robotics and Autonomous Systems*, vol. 61, no. 12, Dec. 2013, issn: 0921-8890. doi: 10.1016/j.robot.2013.09.004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S092188901300167X.

[64] E. Galceran, S. Nagappa, M. Carreras, P. Ridao, and A. Palomer, "Uncertainty-driven survey path planning for bathymetric mapping", in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, IEEE, 2013, pp. 6006–6012. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6697228 (visited on 05/17/2016).

[65] G. Hattab, M. El-Tarhuni, M. Al-Ali, T. Joudeh, and N. Qaddoumi, "An Underwater Wireless Sensor Network with Realistic Radio Frequency Path Loss Model", en, *International Journal of Distributed Sensor Networks*, vol. 9, no. 3, p. 508 708, Mar. 2013, issn: , 1550-1477. doi: 10.1155/2013/508708. [Online]. Available: http://dsn.sagepub.com/content/9/3/508708 (visited on 08/30/2016).

[66] M. T. Leccadito, T. Bakker, R. Niu, and R. H. Klenke, "A Kalman Filter Based Attitude Heading Reference System Using a Low Cost Inertial Measurement Unit", PhD thesis, Master's thesis, Virginia Commonwealth University, 2013. [Online]. Available: http://arc.aiaa.org/doi/pdf/10.2514/6.2015-0604 (visited on 12/27/2016).

[67] C. Van Den Berg, *IHC Merwede Handbook for Centrifugal Pumps and Slurry Transportation*. MTI Holland, 2013.

[68] J. Claus, "Design and Development of an Inexpensive Acoustic Underwater Communications and Control System", en, PhD thesis, Florida institute of technology, Florida, 2014.

[69] P. Freitas, "Evaluation of Wi-Fi Underwater Networks in Freshwater", PhD thesis, UNIVERSIDADE DO PORTO, Porto, 2014.

[70] E. Galceran, "Coverage path planning for autonomous underwater vehicles", en, PhD thesis, Universitat de Girona, 2014.

[71] ——, "Towards Coverage path planning for autonomous underwater vehicles", en, PhD thesis, Universitat de Girona, 2014.

[72] G. van der Schrieck, *Dredging Technology - Guest lecture notes CIE5300 Issue 2014*, en. GLM van der SCHRIECK BV, 2014.

[73] Wei Gao, Yalong Liu, and Bo Xu, "Robust Huber-Based Iterated Divided Difference Filtering with Application to Cooperative Localization of Autonomous Underwater Vehicles", *Sensors (14248220)*, vol. 14, no. 12, pp. 24 523–24 542, Dec. 2014, issn: 14248220. doi: 10.3390/s141224523. [Online]. Available: http://search.ebscohost.com/login.aspx?direct=true&db=a9h&AN=100145980&lang=nl&site=ehost-live (visited on 05/20/2016).

[74] F. Abyarjoo, A. Barreto, J. Cofino, and F. R. Ortega, "Implementing a sensor fusion algorithm for 3d orientation detection with inertial/magnetic sensors", in *Innovations and Advances in Computing, Informatics, Systems Sciences, Networking and Engineering*, Springer, 2015, pp. 305–310. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-06773-5_41 (visited on 12/27/2016).

[75] M. Algabri, H. Mathkour, H. Ramdane, and M. Alsulaiman, "Comparative study of soft computing techniques for mobile robot navigation in an unknown environment", *Computers in Human Behavior*, vol. 50, pp. 42–56, Sep. 2015, issn: 0747-5632. doi: 10.1016/j.chb.2015.03.062. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0747563215002605 (visited on 04/17/2016).

[76] Y. Feng, X. Li, and X. Zhang, "An Adaptive Compensation Algorithm for Temperature Drift of Micro-Electro-Mechanical Systems Gyroscopes Using a Strong Tracking Kalman Filter", *Sensors*, vol. 15, no. 5, pp. 11 222–11 238, 2015. [Online]. Available: http://www.mdpi.com/1424-8220/15/5/11222/htm (visited on 12/28/2016).

[77] D. C. Giancoli, *Physics Principles with Applications*, en. Pearson Education, Limited, Jan. 2015, Google-Books-ID: jYIyngEACAAJ, isbn: 978-1-292-05712-5.

[78] M. S. Grewal and A. P. Andrews, *Kalman Filtering: Theory and Practice with MATLAB*, en. John Wiley & Sons, Feb. 2015, Google-Books-ID: Sgx9BgAAQBAJ, isbn: 978-1-118-98496-3.

[79] S. Babani, A. A. Bature, M. I. Faruk, and N. K. Dankadai, "COMPARATIVE STUDY BETWEEN FIBER OPTIC AND COPPER IN COMMUNICATION LINK", 2016. [Online]. Available: http://www.academia.edu/download/34322554/comparative-study-between-fiber-optic-and-copper-in-communication-link.pdf (visited on 10/03/2016).

[80] Jolectra, *PLC diagrams Besturingspaneel container - 216.156.P1*. Jun. 2016.

[81] Mathworks, *Mechanical-to-hydraulic power conversion device - Simulink - MathWorks Benelux*. 2016. [Online]. Available: https://nl.mathworks.com/help/physmod/hydro/ref/fixeddisplacementpump.html (visited on 11/14/2016).

[82] S. A. Miedema, *Slurry Transport Fundamentals, A historical Overview & The Delft Head Loss & Limit Deposit Velocity Framework*. TU Delft, Jun. 2016.

[83] W. Vlasblom, "Designing Dredging Equipment", TU Delft, Delft University of Technology, Lecture notes, 2016.

[84] A. Recuero, R. Serban, B. Peterson, H. Sugiyama, P. Jayakumar, and D. Negrut, "A high-fidelity approach for vehicle mobility simulation: Nonlinear finite element tires operating on granular material", *Journal of Terramechanics*, vol. 72, pp. 39–54, Aug. 2017. doi: 10.1016/j.jterra.2017.04.002.

[85] Roger R Labbe jr, *Kalman and Bayesian Filters in Python*. Jan. 2017.

[86] *Wheeled mobile robotics: from fundamentals towards autonomous systems*, MATLAB examples, Oxford: Butterworth-Heinemann, an imprint of Elsevier, 2017, 491 pp.

[87] 3Blue1Brown, *Visualizing quaternions (4d numbers) with stereographic projection - YouTube*, 2018. [Online]. Available: https://www.youtube.com/watch?v=d4EgbgTm0Bg (visited on 05/24/2020).

[88] S. J. Johnston, P. J. Basford, C. S. Perkins, H. Herry, F. P. Tso, D. Pezaros, R. D. Mullins, E. Yoneki, S. J. Cox, and J. Singer, "Commodity single board computer clusters and their applications", *Future Generation Computer Systems*, vol. 89, pp. 201–212, Dec. 2018, issn: 0167739X.

[89] R. Serban, N. Olsen, D. Negrut, A. Recuero, and P. Jayakumar, "A co-simulation framework for high-performance, high-fidelity simulation of ground vehicle-terrain interaction", *International Journal of Vehicle Performance*, vol. 5, Aug. 8, 2018. doi: 10.1504/IJVP.2019.10021232.

[90] A. Tasora, D. Mangoni, and D. Negrut, *An Overview of the Chrono Soil Contact Model (SCM) Implementation*. Aug. 9, 2018.

[91] R. v. d. Wetering, "Ihc mti crawler - final report", IHC MTI B.V., report, 2018.

APPENDICES

APPENDIX A
CRAWLER PARTLIST

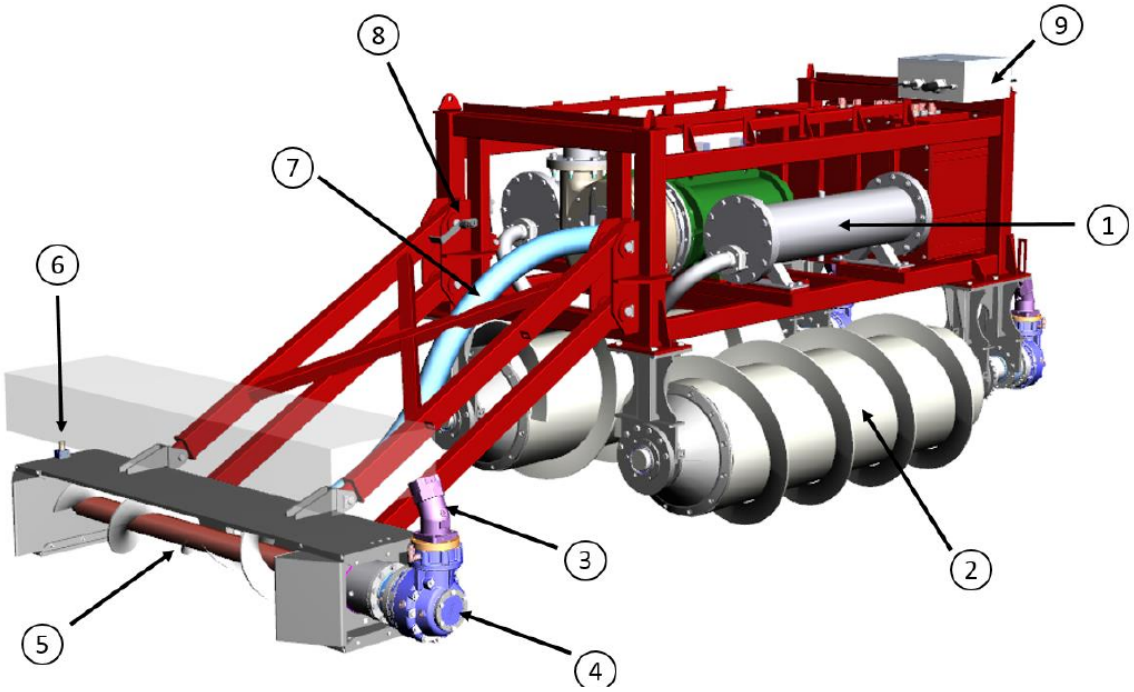


FIGURE A.1: MAJOR PARTLIST SIDEVIEW [91]

NO	DESCRIPTION
1	Oil buffer
2	Archimedes screw propulsion
3	Hydraulic motor
4	Gearbox
5	Auger
6	RPM sensor auger
7	Flexible suction hose 100mm
8	Dredge head angle sensor
9	Termination box, interface between crawler and umbilical

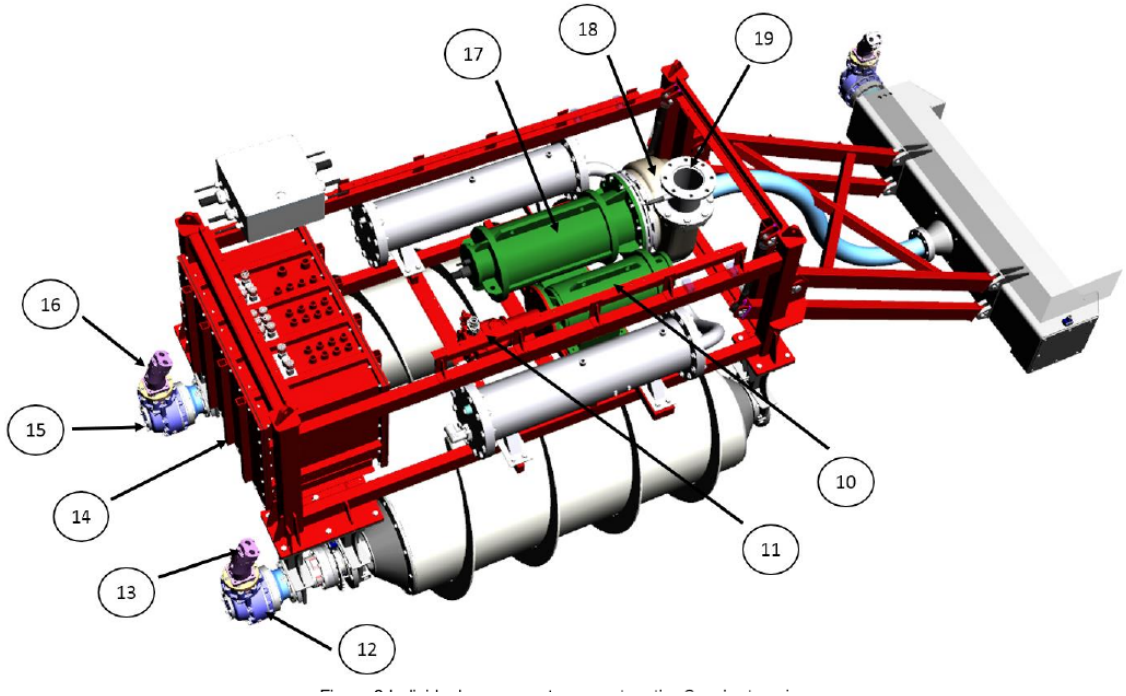


FIGURE A.2: MAJOR PARTLIST TOPVIEW [91]

NO	DESCRIPTION
10	Electric motor
11	Hydraulic pump
12	Gearbox
13	Hydraulic motor
14	Connection box
15	Gearbox
16	Hydraulic motor
17	Electric motor
18	IHC TT 150 dredge pump
19	Discharge

APPENDIX B
APPLIED RESEARCH METHODS

The used research strategy is described for the purpose of transparency and quality control. It serves as the basis for chapter 3. By defining keywords and queries, setting boundaries and specifying the databases, potential sources are filtered on relevance. These are then read and reviewed, if they are indeed relevant and adhere to stated selection criteria, they are used in this study.

APPLIED PARAMETERS

Parameters for a research study The applied parameters for each search query are listed below:

PARAMETER	VALUE
Language of publication	English or Dutch.
Area of research	Engineering, Maritime, Artificial Intelligence, Sensors, Dredging.
Industry sector	Maritime, Robotics, Mining.
Geographical area	World wide.
Time period	1995 till present.
Types of literature	Peer review papers, MSc thesis, Ph.D. thesis, scientific books, (inter-)national standards.

KEYWORDS AND QUERIES

- "CPP" **OR** "coverage path planning"
  - **AND** "underwater"
  - **AND** "cellular decomposition"
    - **AND** "Morse"
    - **AND** "Trapezoidal"
    - **AND** "Boustrophedon"
  - **AND** "landmark" **OR** "topological"
    - **AND** "slice decomposition"
    - **AND** "neural networks"
  - **AND** "grid"
    - **AND** "spanning tree" **OR** "STC"
    - **AND** "neural networks"
    - **AND** "probability" **OR** "certainty"
  - **AND** "cooperative localization"
- "auger" **OR** "screw conveyor"
  - **AND** "production" **OR** "flow"
  - **AND** "dredging" **OR** "dredge head"
- "underwater" **AND** "communication"
  - **AND** "wireless"
    - **AND** "protocol"
    - **AND** "electromagnetic"

        * **AND** "acoustic"
        * **AND** "optical"
        * **AND** "environment"
    - **AND** "umbilical"
        * **AND** "environment"
- "IMU" **OR** "Inertial Measurement Unit"
    - **AND** "gyro" **OR** "gyroscope"
        * **AND** "error"
        * **AND** "temperature"
    - **AND** "accelerometer"
        * **AND** "error"
        * **AND** "temperature"
        * **AND** "gravity"
    - **AND** "magnetometer"
        * **AND** "error"
        * **AND** "temperature"
- "pressure" **AND** "sensor"
    - **AND** "underwater"
    - **AND** "error"
    - **AND** "temperature"
    - **AND** "water" **AND** "depth"
    - **AND** "resolution"
- "Kalman filter"
    - **AND** "gyro" **OR** "gyroscope"
    - **AND** "accelerometer"
    - **AND** "magnetometer"
    - **AND** "quaternions"
    - **AND** "AHRS" **OR** "Attitude and heading reference system"
    - **AND** "extended"
    - **AND** "unscented"

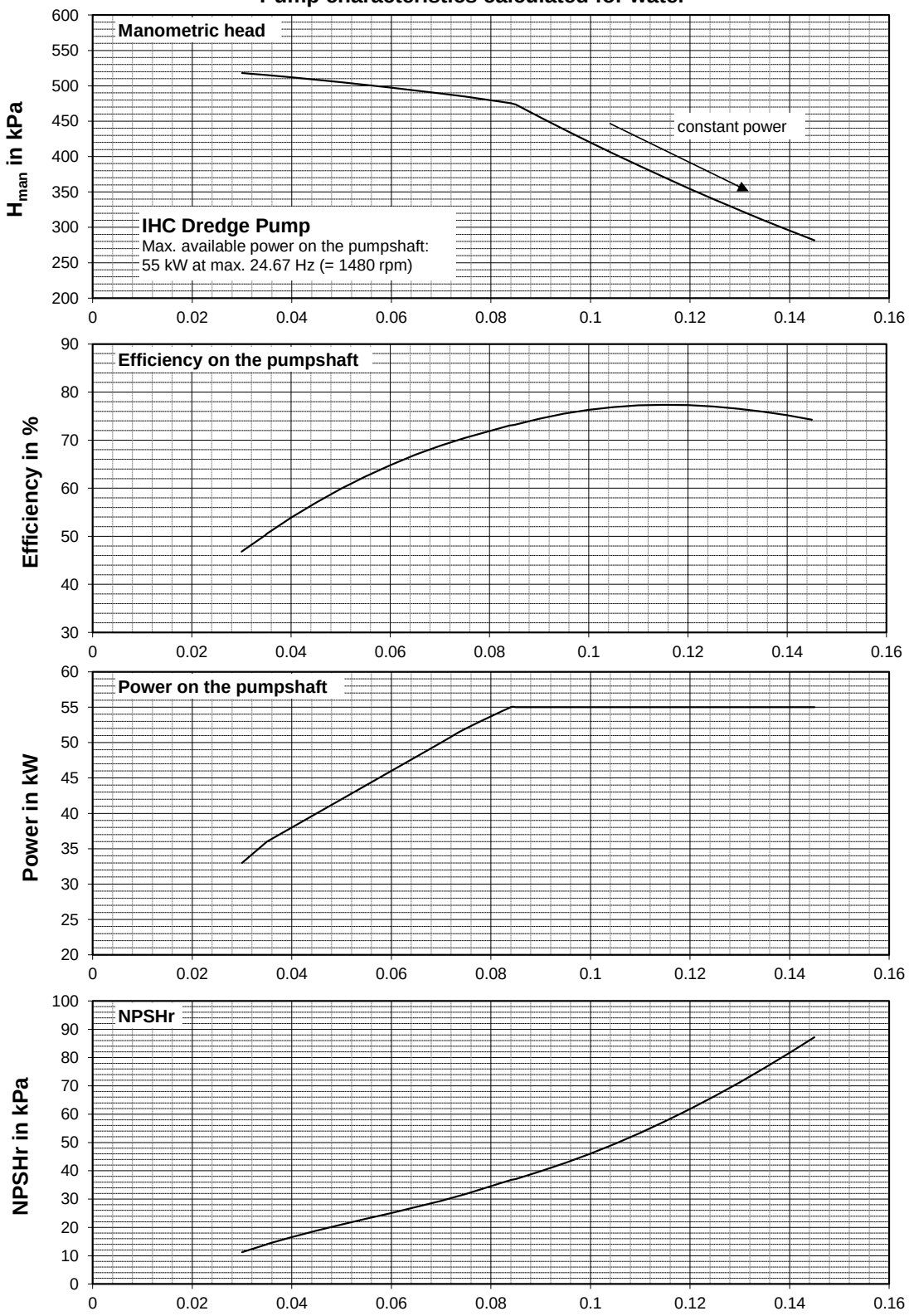
DATABASES AND SEARCH ENGINES

DATABASE	TYPE
Academic Search Complete EBSCO	More then 10.000 digital academic magazines
Google Scholar	Scientific Internet search engine
Microsoft Academic Research	Scientific Internet search engine
NEN Connect	Search engine for (inter-)national norms ISO / NEN
Science direct	Over 2.000 scientific magazines
Springer link	Over 2.500 scientific magazines
Wiley Online Library	Almost 1.000 scientific magazines
MTeye	MTI Library consisting of roughly $700\mathrm{m}$ of technical books related to soil, sea, mining and engineering
My own Library	A mere $20\mathrm{m}$ of technical books, related to math, engineering, programming, electronics and artificial intelligence

IHC Merwede

APPENDIX C
DATASHEET PUMP TT15-55

Pump characteristics calculated for water



This prognosis is based on measurements according to ISO 2548 (Class C) code for acceptance testing of pumps

Pump type IHC 37.5-9.5-15, with a 3-bladed impeller		Report:	Page:
Date: 12-Jul-2011	© Copyright IHC	GO 4294	

APPENDIX D
KALMAN SOURCES

LISTING D.1: FALLING BALL EXAMPLE

```
import matplotlib.pyplot as pl

pl.rcParams['legend.loc'] = 'best'
import numpy as np
from numpy import dot
from scipy.linalg import inv


def build_real_values():
    num_of_time_steps = 100
    dt = 0.2
    t = np.linspace(start=0., stop=num_of_time_steps * dt, num=num_of_time_steps)
    s = np.zeros((num_of_time_steps, 1))
    v = np.zeros((num_of_time_steps, 1))
    a = np.zeros((num_of_time_steps, 1))

    a = np.ones((num_of_time_steps, 1)) * 9.81  # Build accelaration profile
    for i in range(1, num_of_time_steps):
        v[i] = v[i - 1] + a[i] * dt  # Build speed profile
        s[i] = s[i - 1] + v[i] * dt + 0.5 * a[i] * dt ** 2  # Build position profile

    return [t, dt, s, v, a]


def build_measurement_values(t, S):
    y = np.zeros((t.size, 2, 1))
    S_m = np.random.normal(0., 50, (len(S), 1)) + S
    y[:, 0] = S_m
    return y


def build_control_values(t, a):
    u = np.ones((t.size, 2, 1)) * a[0]
    return u


def init_kalman(t, dt):
    phi_s = 5

    F = np.array([
        [1., dt],
        [0., 1.]
    ])

    B = np.array([
        [0.5 * dt ** 2, 0.],
        [0, dt]
    ])

    H = np.array([
        [1., 0.],
        [0., 0.0]
```

```
    ])

    Q = np.array([
        [(1 / 3) * dt ** 3, 0.5 * dt ** 2],
        [0.5 * dt ** 2, dt]
    ]) * phi_s

    R = np.array([
        [40**2, 0.0],
        [0.0, 15**2]
    ])
    v = np.random.normal(0, 25e-2, (t.size, 2, 1))
    w = np.random.normal(0, 25e-2, (t.size, 2, 1))
    return [F, B, H, Q, R, v, w]


def kalman(t, kalman_values, u, z, error):
    x = np.zeros((t.size, 2, 1))
    P = np.zeros((t.size, 2, 2))
    P[0, :, :] = np.array([
        [error[0] ** 2, 0.],
        [0., error[0] ** 2]
    ])

    xhat = np.zeros((t.size, 2, 1))
    y = np.zeros((t.size, 2, 1))

    F = kalman_values[0]
    B = kalman_values[1]
    H = kalman_values[2]
    Q = kalman_values[3]
    R = kalman_values[4]
    v = kalman_values[5]
    w = kalman_values[6]

    K = np.zeros((t.size, 2, 2))

    for k in range(1, t.size):
        xhat[k] = dot(F, x[k - 1]) + dot(B, u[k]) + w[k]
        Phat = dot(F, dot(P[k - 1], F.T)) + Q

        y[k] = z[k] - dot(H.T, xhat[k])

        S = dot(H.T, dot(Phat, H)) + R
        S = np.linalg.inv(S)
        K[k] = dot(Phat, dot(H, S))
        x[k] = xhat[k] + dot(K[k], y[k])

        P[k] = dot(np.eye(2) - dot(K[k], H.T), Phat)
    return [x, K, P, xhat, z]


def NEES(xs, est_xs, ps):
    est_err = xs - est_xs
    err = np.zeros(xs[:, 0].size)
    i = 0
    for x, p in zip(est_err, ps):
        err[i] = (np.dot(x.T, inv(p)).dot(x))
        i += 1
    return err
```

```
def plot_results(t, x, xground, a, u, y, K, P, xhat, z, nees):
    pl.figure()
    pl.subplot(311)
    pl.plot(xground[:, 0])
    pl.plot(x[:, 0], '+')
    pl.plot(z[:, 0], '.')
    pl.subplot(312)
    pl.plot(xground[:, 1])
    pl.plot(xhat[:, 1], 'o')
    pl.plot(x[:, 1])
    pl.subplot(313)
    pl.plot(nees)
    pl.tight_layout()
    pl.savefig('fallingBall.png')
    pl.show()


def save_results(t, xground, x, z, nees):
    # save csv
    datPos = np.zeros((t.size, 4, 1))
    datPos[:, 0] = t.reshape((t.size, 1))
    datPos[:, 1] = xground[:, 0]
    datPos[:, 2] = x[:, 0]
    datPos[:, 3] = z[:, 0]
    np.savetxt('fallingBallPos.dat', datPos, delimiter=',')

    datSpeed = np.zeros((t.size, 3, 1))
    datSpeed[:, 0] = t.reshape((t.size, 1))
    datSpeed[:, 1] = xground[:, 1]
    datSpeed[:, 2] = x[:, 1]
    np.savetxt('fallingBallSpeed.dat', datSpeed, delimiter=',')

    datNEES = np.zeros((t.size + 1, 2, 1))
    datNEES[0:t.size, 0] = t.reshape((t.size, 1))
    datNEES[0:t.size, 1] = nees.reshape(t.size, 1)
    datNEES[-1, 0] = t[-1]
    datNEES[-1, 1] = 0.0
    np.savetxt('fallingBall_NEES.dat', datNEES, delimiter=',')

    np.savetxt('fallingBall_meanNEES.dat', [np.mean(nees)], delimiter=',')


def main():
    [t, dt, s, v, a] = build_real_values()
    z = build_measurement_values(t, s)
    u = build_control_values(t, a)
    [F, B, H, Q, R, vv, w] = init_kalman(t, dt)
    error = [40, 10]
    kalman_values = [F, B, H, Q, R, vv, w]
    x, K, P, xhat, y = kalman(t, kalman_values, u, z, error)
    xground = np.zeros(x.shape)
    xground[:, 0] = s
    xground[:, 1] = v
    nees = NEES(xground, x, P)
    print(np.mean(nees))
    save_results(t, xground, x, z, nees)
    plot_results(t, x, xground, a, u, y, K, P, xhat, y, nees)


if __name__ == '__main__':
    main()
```

E APPENDIX BEARING CAPACITY CALCULATION

LISTING E.1: TORQUEPREDICTION.PY

```
from enum import Enum
from math import pi, atan, acos, cos

import numpy as np
from AOD.Material import *

from AOD.Unit import *


class Position(Enum):
    Inner = 0
    Outer = 1.
    Middle = 0.5


class ForceType(Enum):
    Dry = 0
    Wet = 1
    Buoyancy = 2


class Friction(object):
    force = 0. * ureg['N']
    angle = 0. * ureg['rad']

    @property
    def x(self):
        return self.force * cos(self.angle)

    @property
    def y(self):
        return self.force * sin(self.angle)


class Cylinder(object):
    _d = 0.6 * ureg['m']  # private diameter of the screw
    _l = 1.92 * ureg['m']  # private length of the screw

    def __init__(self, d=None, l=None, no_of_helices=1):
        """ Constructor """
        if d is not None:
            self._d = d
        if l is not None:
            self._l = l

    @property
    def d(self):
        """ Gets the diameter of the cylinder """
        return self._d

    @d.setter
    def d(self, value):
```

```
        """ Sets the diameter of the cylinder """
        self._d = value

    @property
    def l(self):
        """ Gets the length of the cylinder """
        return self._l

    @l.setter
    def l(self, value):
        """ Sets the length of the cylinder """
        self._l = value

    @property
    def r(self):
        """ Gets the radius of the cylinder"""
        return self._d / 2

    @property
    def O(self):
        """ Gets the circumfence of the cylinder"""
        return pi * self.d

    @property
    def V(self):
        """ Volume of the cylinder"""
        return 0.25 * pi * self.d ** 2 * self.l

    def buoyancy(self, layers, depth):
        """ Buoyancy force generated by the volume of air in the cylinder """
        F = self.V * (layers['Fluid'].rho - layers['Air'].rho) * g
        return F.to('N')

    @property
    def A_t(self):
        return pi / 4 * self.d ** 2

    def theta(self, depth):
        x = (self.r - depth) / self.r
        t = np.arccos(x) * ureg['rad']
        return t

    def A_s(self, depth):
        """ Submerged cross-section area of the cylinder """
        theta = self.theta(depth).magnitude
        a_s = self.r ** 2 * (theta - np.sin(theta) * np.cos(theta))  # TODO check with Edwin
         ↪ why pi has to be removed
        a_s[depth >= self.d] = self.A_t
        return a_s

    def s(self, depth):
        """ Soil contact arc """
        theta = self.theta(depth)
        return theta * self.r

    def A_c(self, depth):
        """ Soil contact surface """
        return  self.s(depth) * self.l

    def V_s(self, depth):
        """ Submerged volume of the cylinder """
        return self.A_s(depth=depth) * self.l
```

```
    def B_acc(self, depth):
        """ Returns the width' and d' for a square representing the submerged cylinder"""
        x = (self.r - depth) / self.r
        b = 2 * self.r * (1 - (x) ** 2) ** 0.5
        b[depth >= self.r] = self.d
        d_acc = self.A_s(depth) / b
        d_acc = np.nan_to_num(d_acc)
        return [b, d_acc]

    def friction(self, force, helix, soil, depth):
        """ Frictional force on the cylinder due to contact with the soil
        Returns the total force generated by the skin friction on the cylinder"""
        # TODO ask Rick why this is not a function of depth
        # TODO make function for moving screw
        f_fr = Friction()
        f_fr.angle = helix.alpha()
        if soil.c.magnitude == 0.:
            # RvdW method
            f_fr.force = force * soil.delta
        else:
            # RL method
            f_s = soil.alpha(weight=force * g, area=self.A_s(depth)) * soil.c
            f_fr.force = f_s * self.A_s(depth)
        f_fr.force.to('N')
        return f_fr

    def torque(self, force, helix, soil, depth):
        t_cyl = self.friction(force=force, helix=helix, soil=soil, depth=depth).force *
         ↪  self.r
        return t_cyl


class Helix(object):
    """ Definition of the Helix"""
    _p = 0.478 * ureg['m']  # private pitch of the helix
    _h = 0.1 * ureg['m']  # private vane height
    _cylinder = None  # private cylinder definition

    def __init__(self, cylinder, p=None, h=None):
        if p is not None:
            self.p = p
        if h is not None:
            self.h = h
        self._cylinder = cylinder

    @property
    def p(self):
        return self._p

    @p.setter
    def p(self, value):
        self._p = value

    @property
    def h(self):
        return self._h

    @h.setter
    def h(self, value):
        self._h = value
```

```
    def d(self, pos: Position = Position.Outer):
        return self._cylinder.d + (2. * pos.value * self.h)

    def O(self, pos: Position = Position.Outer):
        return pi * self.d(pos=pos)

    def r(self, pos: Position = Position.Outer):
        return self.d(pos=pos) / 2

    def alpha(self, pos: Position = Position.Inner):
        return atan(self.p / self.O(pos)) * ureg['rad']

    @property
    def no_threads(self):
        return self._cylinder.l / self.p

    def l(self, pos: Position = Position.Outer):
        return self.no_threads * (self.p ** 2 + (self.d(pos) * pi) ** 2) ** 0.5

    @property
    def A(self):
        return self.l(pos=Position.Middle) * self.h

    def theta(self, depth):
        x = (self.r() - depth) / self.r()
        t = np.arccos(x) * ureg['rad']
        return t

    def A_s(self, depth):
        """ Submerged area of the helix, Simplified by approaching it
         as a number of cirlces that are place n times on the cylinder """
        depth += self.h
        theta = self.theta(depth).magnitude
        a_s = self.r() ** 2 * (theta - np.sin(theta) * np.cos(theta))  # TODO check with
        ↪ Edwin why pi has to be removed
        a_s *= self.no_threads
        a_s[depth >= self.d()] = self.A
        a_s *= 2  # TODO check if area needs to be multiplied by two, two-sides of a vane
        return a_s

    def friction(self, load, depth, soil):
        """ Frictional force generated by the vanes"""
        fr_h = Friction()
        fr_h.force = soil.sigma_h(load) * self.A_s(depth) * soil.delta
        fr_h.angle = self.alpha(Position.Middle)
        return fr_h

    def torque(self, load, depth, soil):
        t_h = self.friction(load=load, depth=depth, soil=soil).x * self.r(Position.Middle)
        return t_h


class Screw(object):
    """ Archimedes screw"""
    cylinder = Cylinder()
    Helices = []

    def __init__(self, d=None, l=None, no_of_helices=1):
        """ Constructor """
        if d is not None:
            self.cylinder._d = d
        if l is not None:
```

```
            self.cylinder._l = l
        self.no_helices = no_of_helices

    @property
    def no_helices(self):
        return len(self.Helices)

    @no_helices.setter
    def no_helices(self, value):
        self.Helices = [Helix(cylinder=self.cylinder) for h in range(value)]

    def buoyancy(self, layers, depth):
        """ Gets the buoyancy of the screw, corrected for the depth in the Soil """
        F = -g * (
            (layers['Air'].rho - layers['Fluid'].rho) * self.cylinder.V + (
                layers['Fluid'].rho - layers['Soil'].rho_ins) * self.cylinder.V_s(
                depth=depth))
        return F.to('N')

    @property
    def helix_A(self):
        """ Gets the total surface of the helix """
        A = 0. * ureg['m**2']
        for h in self.Helices:
            A += h.A
        return A

    @property
    def helix_no_Threads(self):
        """ Gets the total number of threads for the screw """
        thr = 0. * ureg['dimensionless']
        for h in self.Helices:
            thr + h.no_threads

    @property
    def r(self):
        """ Gets the main radius of the cylinder """
        return self.cylinder.r

    @property
    def helix(self):
        return self.Helices[0]

    def torque(self, force, load, depth, soil):
        t_s = self.cylinder.torque(force=force, helix=self.helix,
                                   soil=soil, depth=depth)  # Torque as a result from
                                    ↪ friction against the cylinder
        for h in self.Helices:
            t_s += h.torque(load=load, depth=depth, soil=soil)  # Torque due to each helix
        return t_s.to('N*m')


class Bot(object):
    """ Representing the bot, consisting of n amount of screws, weight and buyoancy """
    Screws = []  # The different screws
    buoyancy = 0. * ureg['N']  # Buoyancy of the bot (with out the screws)
    _weight_dry = 5.1e3 * ureg['kg']  # private weight of the bot
    _depth = 0. * ureg['m']  # private depth of the bot, taken at the bottom of the cylinders
    _v = 0.4 * ureg['m/s']  # private speed of the bot

    def __init__(self, no_of_screws=2):
        """ Constructor"""
```

```
        self.no_screw = no_of_screws

    @property
    def Screw(self):
        return self.Screws[0]

    @property
    def no_screw(self):
        return len(self.Screws)

    @no_screw.setter
    def no_screw(self, value):
        self.Screws = [Screw() for s in range(value)]

    @property
    def depth(self):
        return self._depth

    @depth.setter
    def depth(self, value):
        self._depth = value

    @property
    def weight_dry(self):
        return self._weight_dry

    @weight_dry.setter
    def weight_dry(self, value):
        self._weight_dry = value

    def F(self, forcetype: ForceType = ForceType.Dry, layers=None):
        force = 0. * ureg['N']
        if forcetype == ForceType.Dry:
            force = self.weight_dry * g
        elif forcetype == ForceType.Wet:
            force = self.F() - self.F(ForceType.Buoyancy, layers)
        elif forcetype == ForceType.Buoyancy:
            force = self.buoyancy
            for s in self.Screws:
                force += s.buoyancy(layers=layers, depth=self.depth)
        return force.to('N')

    def torque(self, load, layers, depth=None):
        if depth is None:
            depth = self.depth
        t_b = 0. * ureg['N*m']
        for s in self.Screws:
            f = self.F(ForceType.Wet, layers=layers) / self.no_screw
            t_b += s.torque(force=f, load=load, depth=depth, soil=layers['Soil'])
        return t_b
```

LISTING E.2: BOT.PY

```
from enum import Enum
from math import pi, atan, acos, cos

import numpy as np
from AOD.Material import *

from AOD.Unit import *
```

```
class Position(Enum):
    Inner = 0
    Outer = 1.
    Middle = 0.5


class ForceType(Enum):
    Dry = 0
    Wet = 1
    Buoyancy = 2


class Friction(object):
    force = 0. * ureg['N']
    angle = 0. * ureg['rad']

    @property
    def x(self):
        return self.force * cos(self.angle)

    @property
    def y(self):
        return self.force * sin(self.angle)


class Cylinder(object):
    _d = 0.6 * ureg['m']  # private diameter of the screw
    _l = 1.92 * ureg['m']  # private length of the screw

    def __init__(self, d=None, l=None, no_of_helices=1):
        """ Constructor """
        if d is not None:
            self._d = d
        if l is not None:
            self._l = l

    @property
    def d(self):
        """ Gets the diameter of the cylinder """
        return self._d

    @d.setter
    def d(self, value):
        """ Sets the diameter of the cylinder """
        self._d = value

    @property
    def l(self):
        """ Gets the length of the cylinder """
        return self._l

    @l.setter
    def l(self, value):
        """ Sets the length of the cylinder """
        self._l = value

    @property
    def r(self):
        """ Gets the radius of the cylinder"""
        return self._d / 2

    @property
```

```
    def O(self):
        """ Gets the circumfence of the cylinder"""
        return pi * self.d

    @property
    def V(self):
        """ Volume of the cylinder"""
        return 0.25 * pi * self.d ** 2 * self.l

    def buoyancy(self, layers, depth):
        """ Buoyancy force generated by the volume of air in the cylinder """
        F = self.V * (layers['Fluid'].rho - layers['Air'].rho) * g
        return F.to('N')

    @property
    def A_t(self):
        return pi / 4 * self.d ** 2

    def theta(self, depth):
        x = (self.r - depth) / self.r
        t = np.arccos(x) * ureg['rad']
        return t

    def A_s(self, depth):
        """ Submerged cross-section area of the cylinder """
        theta = self.theta(depth).magnitude
        a_s = self.r ** 2 * (theta - np.sin(theta) * np.cos(theta))  # TODO check with Edwin
         ↪ why pi has to be removed
        a_s[depth >= self.d] = self.A_t
        return a_s

    def s(self, depth):
        """ Soil contact arc """
        theta = self.theta(depth)
        return theta * self.r

    def A_c(self, depth):
        """ Soil contact surface """
        return  self.s(depth) * self.l

    def V_s(self, depth):
        """ Submerged volume of the cylinder """
        return self.A_s(depth=depth) * self.l

    def B_acc(self, depth):
        """ Returns the width' and d' for a square representing the submerged cylinder"""
        x = (self.r - depth) / self.r
        b = 2 * self.r * (1 - (x) ** 2) ** 0.5
        b[depth >= self.r] = self.d
        d_acc = self.A_s(depth) / b
        d_acc = np.nan_to_num(d_acc)
        return [b, d_acc]

    def friction(self, force, helix, soil, depth):
        """ Frictional force on the cylinder due to contact with the soil
        Returns the total force generated by the skin friction on the cylinder"""
        # TODO ask Rick why this is not a function of depth
        # TODO make function for moving screw
        f_fr = Friction()
        f_fr.angle = helix.alpha()
        if soil.c.magnitude == 0.:
            # RvdW method
```

```
            f_fr.force = force * soil.delta
        else:
            # RL method
            f_s = soil.alpha(weight=force * g, area=self.A_s(depth)) * soil.c
            f_fr.force = f_s * self.A_s(depth)
        f_fr.force.to('N')
        return f_fr

    def torque(self, force, helix, soil, depth):
        t_cyl = self.friction(force=force, helix=helix, soil=soil, depth=depth).force *
        ↪ self.r
        return t_cyl


class Helix(object):
    """ Definition of the Helix"""
    _p = 0.478 * ureg['m']  # private pitch of the helix
    _h = 0.1 * ureg['m']  # private vane height
    _cylinder = None  # private cylinder definition

    def __init__(self, cylinder, p=None, h=None):
        if p is not None:
            self.p = p
        if h is not None:
            self.h = h
        self._cylinder = cylinder

    @property
    def p(self):
        return self._p

    @p.setter
    def p(self, value):
        self._p = value

    @property
    def h(self):
        return self._h

    @h.setter
    def h(self, value):
        self._h = value

    def d(self, pos: Position = Position.Outer):
        return self._cylinder.d + (2. * pos.value * self.h)

    def O(self, pos: Position = Position.Outer):
        return pi * self.d(pos=pos)

    def r(self, pos: Position = Position.Outer):
        return self.d(pos=pos) / 2

    def alpha(self, pos: Position = Position.Inner):
        return atan(self.p / self.O(pos)) * ureg['rad']

    @property
    def no_threads(self):
        return self._cylinder.l / self.p

    def l(self, pos: Position = Position.Outer):
        return self.no_threads * (self.p ** 2 + (self.d(pos) * pi) ** 2) ** 0.5
```

```
    @property
    def A(self):
        return self.l(pos=Position.Middle) * self.h

    def theta(self, depth):
        x = (self.r() - depth) / self.r()
        t = np.arccos(x) * ureg['rad']
        return t

    def A_s(self, depth):
        """ Submerged area of the helix, Simplified by approaching it
         as a number of cirlces that are place n times on the cylinder """
        depth += self.h
        theta = self.theta(depth).magnitude
        a_s = self.r() ** 2 * (theta - np.sin(theta) * np.cos(theta))  # TODO check with
        ↪  Edwin why pi has to be removed
        a_s *= self.no_threads
        a_s[depth >= self.d()] = self.A
        a_s *= 2  # TODO check if area needs to be multiplied by two, two-sides of a vane
        return a_s

    def friction(self, load, depth, soil):
        """ Frictional force generated by the vanes"""
        fr_h = Friction()
        fr_h.force = soil.sigma_h(load) * self.A_s(depth) * soil.delta
        fr_h.angle = self.alpha(Position.Middle)
        return fr_h

    def torque(self, load, depth, soil):
        t_h = self.friction(load=load, depth=depth, soil=soil).x * self.r(Position.Middle)
        return t_h


class Screw(object):
    """ Archimedes screw"""
    cylinder = Cylinder()
    Helices = []

    def __init__(self, d=None, l=None, no_of_helices=1):
        """ Constructor """
        if d is not None:
            self.cylinder._d = d
        if l is not None:
            self.cylinder._l = l
        self.no_helices = no_of_helices

    @property
    def no_helices(self):
        return len(self.Helices)

    @no_helices.setter
    def no_helices(self, value):
        self.Helices = [Helix(cylinder=self.cylinder) for h in range(value)]

    def buoyancy(self, layers, depth):
        """ Gets the buoyancy of the screw, corrected for the depth in the Soil """
        F = -g * (
            (layers['Air'].rho - layers['Fluid'].rho) * self.cylinder.V + (
                layers['Fluid'].rho - layers['Soil'].rho_ins) * self.cylinder.V_s(
                depth=depth))
        return F.to('N')
```

```
    @property
    def helix_A(self):
        """ Gets the total surface of the helix """
        A = 0. * ureg['m**2']
        for h in self.Helices:
            A += h.A
        return A

    @property
    def helix_no_Threads(self):
        """ Gets the total number of threads for the screw """
        thr = 0. * ureg['dimensionless']
        for h in self.Helices:
            thr + h.no_threads

    @property
    def r(self):
        """ Gets the main radius of the cylinder """
        return self.cylinder.r

    @property
    def helix(self):
        return self.Helices[0]

    def torque(self, force, load, depth, soil):
        t_s = self.cylinder.torque(force=force, helix=self.helix,
                                   soil=soil, depth=depth)  # Torque as a result from
                                    ↪ friction against the cylinder
        for h in self.Helices:
            t_s += h.torque(load=load, depth=depth, soil=soil)  # Torque due to each helix
        return t_s.to('N*m')


class Bot(object):
    """ Representing the bot, consisting of n amount of screws, weight and buyoancy """
    Screws = []  # The different screws
    buoyancy = 0. * ureg['N']  # Buoyancy of the bot (with out the screws)
    _weight_dry = 5.1e3 * ureg['kg']  # private weight of the bot
    _depth = 0. * ureg['m']  # private depth of the bot, taken at the bottom of the cylinders
    _v = 0.4 * ureg['m/s']  # private speed of the bot

    def __init__(self, no_of_screws=2):
        """ Constructor"""
        self.no_screw = no_of_screws

    @property
    def Screw(self):
        return self.Screws[0]

    @property
    def no_screw(self):
        return len(self.Screws)

    @no_screw.setter
    def no_screw(self, value):
        self.Screws = [Screw() for s in range(value)]

    @property
    def depth(self):
        return self._depth

    @depth.setter
```

```
    def depth(self, value):
        self._depth = value

    @property
    def weight_dry(self):
        return self._weight_dry

    @weight_dry.setter
    def weight_dry(self, value):
        self._weight_dry = value

    def F(self, forcetype: ForceType = ForceType.Dry, layers=None):
        force = 0. * ureg['N']
        if forcetype == ForceType.Dry:
            force = self.weight_dry * g
        elif forcetype == ForceType.Wet:
            force = self.F() - self.F(ForceType.Buoyancy, layers)
        elif forcetype == ForceType.Buoyancy:
            force = self.buoyancy
            for s in self.Screws:
                force += s.buoyancy(layers=layers, depth=self.depth)
        return force.to('N')

    def torque(self, load, layers, depth=None):
        if depth is None:
            depth = self.depth
        t_b = 0. * ureg['N*m']
        for s in self.Screws:
            f = self.F(ForceType.Wet, layers=layers) / self.no_screw
            t_b += s.torque(force=f, load=load, depth=depth, soil=layers['Soil'])
        return t_b
```

LISTING E.3: MATERIAL.PY

```
from math import sqrt, sin, exp, tan, pi, cos

from AOD.Unit import *
import sys


class Material(object):
    """Material class, the parent of all used materials"""
    _rho = 1000. * ureg['kg/m**3']  # private density
    _T = 15 * ureg['degC']  # private temperature
    _rho_func = None  # private function for materials where density is a function of
     ↪ temperature
    _depth = 1. * ureg['m']

    def __str__(self):
        return r'Properties at ' + str(self.T.to('degC')) + '\n' + 'density: ' +
         ↪ str(self.rho)

    def __init__(self, rho=None, T=None, rho_func=None):
        """Constructor for Materials"""
        if rho is not None:
            self._rho = rho
        if T is not None:
            self._T = T
        else:
            self.T = 15
        if rho_func is not None:
            self._rho_func = rho_func
```

```
    @property
    def depth(self):
        return self._depth

    @depth.setter
    def depth(self, value):
        self._depth = value

    @property
    def T(self):
        """ Returns the current temperature of model"""
        return self._T

    @T.setter
    def T(self, value):
        """ Sets the new temperature, in degrees Celsius """
        if type(value) is type(ureg['degC']):
            self._T = value
        else:
            self._T = value * ureg['degC']

    @property
    def rho(self):
        """ Gets the density of a material"""
        return self._rho

    @rho.setter
    def rho(self, value):
        """ Sets the density of a material"""
        self._rho = value


class Fluid(Material):
    _P = 0. * ureg['Pa']  # private pressure of the fluid
    _mu = 0. * ureg['Pa*s']  # private dynamic viscosity
    _mu_func = None  # private dynamic viscosity

    def __init__(self, rho=None, rho_func=None, T=None, P=None, mu=None, mu_func=None):
        Material.__init__(self, T=T, rho=rho, rho_func=rho_func)
        if P is not None:
            self._P = P
        if mu is not None:
            self._mu = mu
        if mu_func is not None:
            self._mu_func = mu_func

    @property
    def P(self):
        return self._P

    @property
    def mu(self):
        """Returns dynamic viscosity, when mu function is specified the function
        is applied, otherwise the private variable is returned"""
        if self._mu_func is not None:
            return self._mu_func(self.T.to('degC').magnitude)
        else:
            return self._mu

    @mu.setter
    def mu(self, value):
```

```
        """ Sets the dynamic viscosity"""
        self._mu = value

    @property
    def nu(self):
        """ returns the kinematic viscosity"""
        return (self.mu / self.rho).to('m**2/s')


class Air(Fluid):
    _R = 0. * ureg['J/(kg*K)']  # private specific gas constant for dry air

    def __init__(self, T=None, P=None, R=None):
        if T is None:
            T = 15. * ureg['degC']
        if P is None:
            P = 101.325e3 * ureg['Pa']
        if R is None:
            self._R = 287.05 * ureg['J/(kg*K)']
        Fluid.__init__(self, rho=1.225 * ureg['kg/m**3'],
                       rho_func=lambda p, r, t: p.to_base_units() / (r.to_base_units() *
                        ↪  t.to('K')),
                       T=T, P=P, mu=1.789e-5 * ureg['Pa*s'])
        self.depth = float('inf') * ureg['m']

    @property
    def R(self):
        return self._R

    @R.setter
    def R(self, value):
        self._R = value

    @property
    def rho(self):
        """ Returns the density of a air, when the material has a
        rho_function specified, the density is calculated using the ideal gas law, otherwise it's
        taken from the private variable """
        if self._rho_func is not None:
            return self._rho_func(self.P, self.R, self.T)
        else:
            return self._rho


class Water(Fluid):
    """ Water material """

    def __init__(self, T=None, P=None):
        """ Constructor of the water class"""
        if T is not None:
            self._T = T
        self._rho = 999.7 * ureg['kg/m**3']
        # Water density dependency on temperature (validity 5<T_f<100C) (Matousek, 2004)
        self._rho_func = \
            lambda T: (999.7 - 0.10512 * (T - 10) - 0.005121 * (T - 10) ** 2 + 0.00001329 *
             ↪  (T - 10) ** 3) * ureg[
                'kg/m**3']
        # Water dynamic & kinematic viscosity as function of temperature (Matousek, 2004)
        self._mu_func = \
            lambda T: (0.10 / (2.1482 * ((T - 8.435) + sqrt(8078.4 + (T - 8.435) ** 2)) -
             ↪  120)) * ureg['Pa*s']
        self.depth = 30. * ureg['m']
```

```
    @property
    def rho(self):
        """ Returns the density of water, when the material has a
        rho_function specified, the density is calculated, otherwise it's
        taken from the private variable """
        if self._rho_func is not None:
            return self._rho_func(self._T.to('degC').magnitude)
        else:
            return self._rho


class Soil(Material):
    """ Definitions for the different types of soils """
    _c = 3.e3 * ureg['Pa']  # private cohesion
    _phi = 0. * ureg['degree']  # private internal friction angle
    _k0 = 0.54 * ureg['dimensionless']  # private coeff of lateral earth press
    _delta = 0. * ureg['rad']  # private external friction angle
    _rho_ins = 1300. * ureg['kg/m**3']  # private In-situ bottom density
    _alpha = {0.2: 0.95,
              0.3: 0.77,
              0.4: 0.7,
              0.5: 0.65,
              0.6: 0.62,
              0.7: 0.6,
              0.8: 0.56,
              0.9: 0.55,
              1.: 0.53,
              1.1: 0.52,
              1.2: 0.5,
              1.3: 0.49,
              1.4: 0.48,
              1.5: 0.47,
              1.6: 0.42,
              1.7: 0.41,
              1.8: 0.41,
              1.9: 0.42,
              2.0: 0.41,
              2.1: 0.41,
              2.2: 0.4,
              2.3: 0.4,
              2.4: 0.4,
              2.5: 0.4,
              3.0: 0.39,
              4.0: 0.39}

    def __str__(self):
        return Material.__str__(self) + ' \nCohesion: ' + str(self.c) +
         ↪  ' \nint. friction angle (phi): ' + str(
            self.phi) + '\ncoeff of lateral earth pressure: ' + str(self.k0) +
             ↪  '\nexternal friction angle: ' + str(
            self.delta) + '\nIn-situ density: ' + str(self.rho_ins)

    def __init__(self, rho=None, T=None, rho_ins=None, c=None, phi=None, k0=None,
     ↪  delta=None):
        """ Constructor for soil materials"""
        Material.__init__(self, rho, T)
        if rho_ins is not None:
            self.rho_ins = rho_ins
        if c is not None:
            self.c = c
        if phi is not None:
```

```
            self.phi = phi
        if k0 is not None:
            self.k0 = k0
        if delta is not None:
            self.delta = delta
        self.depth = -float('inf') * ureg['m']

    def alpha(self, weight, area):
        ratio = (weight / area).item(0)
        prev_key = sys.float_info.min
        if ratio in self._alpha.keys():
            return self._alpha[ratio]
        else:
            if len(self._alpha) == 1 or list(self._alpha.keys())[0] > ratio:
                return list(self._alpha.values())[0]
            for key in self._alpha.items():
                if key[0] > ratio:
                    dR = ratio - prev_key
                    dAlpha = self._alpha[key[0]] - self._alpha[prev_key]
                    return self._alpha[prev_key] + dR * dAlpha / (key[0] - prev_key)
                else:
                    prev_key = key[0]
        return self._alpha[prev_key]

    def alpha_bulldozer(self, sigma_sb, sigma_cb):
        ratio = sigma_sb / sigma_cb
        alpha = (-0.0262 * ratio ** 3 + 0.223 * ratio ** 2 - 0.6143 * ratio + 0.9509) + 0.1
        return alpha

    def gamma(self, layers):
        """ Submerged soil weight """
        return g * (layers['Soil'].rho_ins - layers['Fluid'].rho)

    def sigma_h(self, load):
        """ Max Horizontal total soil stress assuming soil is at rest """
        return load * self.k0

    @property
    def c(self):
        """ Gets the soil cohesion """
        return self._c

    @c.setter
    def c(self, value):
        """ Sets the soil cohesion """
        self._c = value

    @property
    def phi(self):
        """ Gets the soil internal friction angle """
        return self._phi

    @phi.setter
    def phi(self, value):
        """ Sets the soil internal friction angle """
        self._phi = value.to('rad')

    @property
    def k0(self):
        """ Gets the soil coeff. of lateral earth pressure """
        return self._k0
```

```
    @k0.setter
    def k0(self, value):
        """ Sets the soil coeff. of lateral earth pressure"""
        self._k0 = value

    @property
    def delta(self):
        """ Gets the soil external friction angle """
        return self._delta

    @delta.setter
    def delta(self, value):
        """ Sets the soil external friction angle """
        self._delta = value.to('rad')

    @property
    def rho_ins(self):
        """ Gets the soil In-situ density """
        return self._rho_ins

    @rho_ins.setter
    def rho_ins(self, value):
        """ Sets teh soil In-situ density """
        self._rho_ins = value.to('kg/m**3')

    @property
    def N_q(self):
        """ Dimensionless constants in the Brinch-Hansen model (verruijt, 2009) """
        return (1 + sin(self.phi)) / (1 - sin(self.phi)) * exp(pi * tan(self.phi))

    @property
    def N_gamma(self):
        return 2 * (self.N_q - 1) * tan(self.phi)

    @property
    def N_c(self):
        if self.phi == 0.:
            return 2 * pi
        else:
            return (self.N_q - 1) * (1 / tan(self.phi))  # TODO check if cot is 1/tan(alpha)

    def S_c(self, B, L):
        return 1. + 0.2 * (B / L)

    def S_q(self, B, L):
        return 1. + (B / L) * sin(self.phi)

    def S_gamma(self, B, L):
        return 1. - 0.3 * (B / L)

    def i_c(self, p=None, t=None):
        """ Inclination factor currently not used """
        if p is not None and t is not None:
            ic = 1 - (t / (self.c * p * tan(self.phi)))
        return 1.

    def i_q(self, p=None, t=None):
        """ Inclination factor currently not used """
        iq = self.i_c(p, t) ** 2
        return 1.

    def i_gamma(self, p=None, t=None):
```

```
        """ Inclination factor currently not used """
        igamma = self.i_c(p, t) ** 3
        return 1.

    def p_allow(self, q, layers, B, L):
        """ Allowed load according to Brinch Hansen  """
        return self.i_c() * self.S_c(B, L) * self.c * self.N_c \
               + self.i_q() * self.S_q(B, L) * q * self.N_q \
               + self.i_gamma() * self.S_gamma(B, L) * 0.5 * self.gamma(layers) * B *
                ↪  self.N_gamma


class Silt(Soil):
    """ Predefined type of Soil, namely silt"""

    def __init__(self):
        """ Constructor """
        Soil.__init__(self, rho=2650. * ureg['kg/m**3'], rho_ins=1300. * ureg['kg/m**3'],
         ↪  c=3.e3 * ureg['Pa'],
                      phi=0. * ureg['degree'],
                      k0=0.54 * ureg['dimensionless'], delta=0. * ureg['degree'])


class Loose_clay(Soil):
    """ Predefined type of Soil, namely Loose clay"""

    def __init__(self):
        """ Constructor """
        Soil.__init__(self, rho=2650. * ureg['kg/m**3'], rho_ins=1400. * ureg['kg/m**3'],
         ↪  c=5.e3 * ureg['Pa'],
                      phi=0. * ureg['degree'],
                      k0=0. * ureg['dimensionless'], delta=0. * ureg['degree'])


class Packed_clay(Soil):
    """ Predefined type of Soil, namely Packed clay"""

    def __init__(self):
        """ Constructor """
        Soil.__init__(self, rho=2650. * ureg['kg/m**3'], rho_ins=1800. * ureg['kg/m**3'],
         ↪  c=10.e3 * ureg['Pa'],
                      phi=0. * ureg['degree'],
                      k0=1. * ureg['dimensionless'], delta=0. * ureg['degree'])


class River_clay(Soil):

     ↪  """ Predefined type of Soil, namely River clay used during the test with project 64120-R02"""

    def __init__(self):
        """ Constructor """
        Soil.__init__(self, rho=2650. * ureg['kg/m**3'], rho_ins=(1.86 * ureg['kg/l'] + 1.88
         ↪  * ureg['kg/l']) / 2,
                      c=(4.2 * ureg['kPa'] + 3.1 * ureg['kPa']) / 2,
                      phi=0. * ureg['degree'],
                      k0=1. * ureg['dimensionless'], delta=0. * ureg['degree'])


class Sand(Soil):
    """ Predefined type of Soil, namely Packed clay"""

    def __init__(self):
```

```
        """ Constructor """
        Soil.__init__(self, rho=2650. * ureg['kg/m**3'], rho_ins=2000. * ureg['kg/m**3'],
         ↪  c=0. * ureg['Pa'],
                      phi=35. * ureg['degree'],
                      k0=0. * ureg['dimensionless'], delta=35. / 3. * ureg['degree'])
```

LISTING E.4: MODEL.PY

```
import numpy as np
from AOD.Material import *
from AOD.Unit import *

from AOD.Bot import *


class World(object):
    """ The representation of fluid and soil layer(s), and their interaction,
     currently only one soil layer is supported """
    Layers = {}

    _T = 15 * ureg['degC']  # private temperature of both fluid and soil

    def __init__(self, layers=None, T=15 * ureg['degC'], depths=None):
        """Constructor specifying the materials for layers, the model temperature and
        the depth for each layer, as a dict. The soilbed, demarcation between soil and fluid
        is 0. [m], where downwards is specified as negative and upwards as positive"""
        if layers is None:
            self.Layers['Air'] = Air()
            self.Layers['Fluid'] = Water()
            self.Layers['Soil'] = Silt()
        self.T = T
        if depths is not None:
            self.layerdepths = depths

    @property
    def layerdepths(self):

         ↪  """ Getter for the depth of each layer, as a dict. The soilbed, demarcation between soil and fluid
        is 0. [m], where downwards is specified as negative and upwards as positive"""
        d = {}
        for key, l in self.Layers.items():
            d[key] = l.depth
        return d

    @layerdepths.setter
    def layerdepths(self, value):

         ↪  """Setter for the depth for each layer, as a dict. The soilbed, demarcation between soil and fluid
        is 0. [m], where downwards is specified as negative and upwards as positive"""
        for key, l in self.Layers.items():
            if key in value:
                l.depth = value[key]
            else:
                print(key + ' depth not passed to layers, assuming: ' + str(l.depth))

    @property
    def n(self):
        """ Gets the porosity of the soilbed """
        return (self.Layers['Soil'].rho - self.Layers['Soil'].rho_ins) / (
            self.Layers['Soil'].rho - self.Layers['Fluid'].rho)

    @property
```

```
    def T(self):
        """ Gets the temperature of the model """
        return self._T

    @T.setter
    def T(self, value):
        """ Sets the temperature of the model, and subsequently, that of the fluid
        and soil"""
        for key, l in self.Layers.items():
            l.T = value
        self._T = value

    @property
    def S(self):
        """ Gets the specific gravity / rel. density of the soilbed """
        return self.Layers['Soil'].rho / self.Layers['Fluid'].rho

    @property
    def gamma(self):
        """ Gets the submerged soil weight of the soilbed """
        return (g * (self.Layers['Soil'].rho_ins - self.Layers['Fluid'].rho)).to('N/m**3')


class Model(object):
    """ The complete model, with setup and solver"""
    world = World()
    bot = Bot()

    def __init__(self, world=None, bot=None):
        if world is not None:
            self.world = world
        if bot is not None:
            self.bot = bot

    def solve_sinkdepth(self, depth=None, resolution=None):
        max_sink_depth = 10. * ureg['m']
        if resolution is None:
            resolution = 1.e-3 * ureg['m']
        if depth is None:
            depth = np.arange(start=0., stop=max_sink_depth.magnitude,
             ↪  step=resolution.magnitude) * ureg['m']
        self.bot.depth = depth
        [B_acc, d_acc] = self.bot.Screw.cylinder.B_acc(depth=depth)
        force = self.bot.F(forcetype=ForceType.Wet, layers=self.world.Layers)
        force /= self.bot.no_screw
        p_load = force / (B_acc * self.bot.Screw.cylinder.l)
        gamma = self.world.Layers['Soil'].gamma(self.world.Layers)
        q = gamma * depth
        p_allow = self.world.Layers['Soil'].p_allow(q, self.world.Layers, B_acc,
         ↪  self.bot.Screw.cylinder.l)

        p_eps = np.sign(p_allow - p_load)
        sink_depth = -1. * ureg['m']
        load = -1. * ureg['Pa']
        for i in range(2, len(depth)):
            if p_eps[i] * p_eps[i - 1] == -1:
                sink_depth = round(depth[i], 3)
                load = p_load[i]
                if i < int(max_sink_depth.magnitude / (2 * resolution.magnitude)):
                    p_allow = p_allow[:i * 2]
                    p_load = p_load[:i * 2]
                    depth = depth[:i * 2]
```

```
                break

        if sink_depth == -1.:
            print('Soil bearing capacity insuficient, solution does not converge within :' +
             ↪  str(max_sink_depth))

        return [p_allow.to('Pa'), p_load.to('Pa'), depth.to('m'),
         ↪  np.array([sink_depth.magnitude]) * ureg['m'],
                np.array([load.magnitude]) * ureg['Pa']]

    def solve_torque(self, depth, load):
        self.bot.depth = depth.copy()
        torque_req = self.bot.torque(load=load, layers=self.world.Layers)
        return torque_req

    def determine_max_torque(self):
        pass
```

LISTING E.5: UNIT.PY

```
from pint import UnitRegistry, set_application_registry

ureg = UnitRegistry(autoconvert_offset_to_baseunit=True)  # Allows for unit save calculations
Q_ = ureg.Quantity  # Allows for custom quantities to be registered
g = 9.80665 * ureg['m/s**2']  # Gravitational constant
```

APPENDIX F
SIMULATION SOURCES

LISTING F.1: CRAWLERSIM.CPP

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#include "chrono/core/ChStream.h"
#include "chrono/core/ChRealtimeStep.h"
#include "chrono/utils/ChUtilsInputOutput.h"

#include "chrono_vehicle/ChConfigVehicle.h"
#include "chrono_vehicle/ChVehicleModelData.h"
#include "chrono_vehicle/terrain/RigidTerrain.h"
#include "chrono_vehicle/driver/ChIrrGuiDriver.h"
#include "chrono_vehicle/driver/ChDataDriver.h"
#include "chrono_vehicle/wheeled_vehicle/utils/ChWheeledVehicleIrrApp.h"

#include "Crawler.h"

#include "chrono_thirdparty/filesystem/path.h"

using namespace chrono;
using namespace chrono::irrlicht;
using namespace chrono::vehicle;
using namespace chrono::vehicle::crawler;

// =============================================================================

// Initial vehicle location and orientation
ChVector<> initLoc(0, 0, 1.0);
ChQuaternion<> initRot(1, 0, 0, 0);
// ChQuaternion<> initRot(0.866025, 0, 0, 0.5);
// ChQuaternion<> initRot(0.7071068, 0, 0, 0.7071068);
// ChQuaternion<> initRot(0.25882, 0, 0, 0.965926);
// ChQuaternion<> initRot(0, 0, 0, 1);
```

```
enum DriverMode { DEFAULT, RECORD, PLAYBACK };
DriverMode driver_mode = DEFAULT;

// Visualization type for vehicle parts (PRIMITIVES, MESH, or NONE)
VisualizationType chassis_vis_type = VisualizationType::MESH;
VisualizationType suspension_vis_type = VisualizationType::PRIMITIVES;
VisualizationType steering_vis_type = VisualizationType::PRIMITIVES;
VisualizationType wheel_vis_type = VisualizationType::MESH;

// Collision type for chassis (PRIMITIVES, MESH, or NONE)
ChassisCollisionType chassis_collision_type = ChassisCollisionType::NONE;

// Type of powertrain model (SHAFTS, SIMPLE)
// PowertrainModelType powertrain_model = PowertrainModelType::SHAFTS;

// Drive type (FWD)
// DrivelineType drive_type = DrivelineType::FWD;

// Type of tire model (RIGID, TMEASY)
TireModelType tire_model = TireModelType::RIGID;

// Rigid terrain
RigidTerrain::Type terrain_model = RigidTerrain::BOX;
double terrainHeight = 0;      // terrain height (FLAT terrain only)
double terrainLength = 100.0;  // size in X direction
double terrainWidth = 100.0;   // size in Y direction

// Point on chassis tracked by the camera
ChVector<> trackPoint(0.0, 0.0, 1.75);

// Contact method
ChMaterialSurface::ContactMethod contact_method = ChMaterialSurface::SMC;
bool contact_vis = false;

// Simulation step sizes
double step_size = 1e-3;
double tire_step_size = step_size;

// Simulation end time
double t_end = 1000;

// Time interval between two render frames
double render_step_size = 1.0 / 50;  // FPS = 50

// Output directories
const std::string out_dir = GetChronoOutputPath() + "Sedan";
const std::string pov_dir = out_dir + "/POVRAY";

// Debug logging
bool debug_output = false;
double debug_step_size = 1.0 / 1;  // FPS = 1

// POV-Ray output
bool povray_output = false;

// =============================================================================

int main(int argc, char* argv[]) {
  GetLog() << "Copyright (c) 2019 Jelle Spijker, using Project Chrono version: " <<
   ↪  CHRONO_VERSION << "\n\n";
```

```
// -------
// Create systems
// -------

// Create the Crawler vehicle, set parameters, and initialize
Crawler ihc_crawler;
ihc_crawler.SetContactMethod(contact_method);
ihc_crawler.SetChassisCollisionType(chassis_collision_type);
ihc_crawler.SetChassisFixed(false);
ihc_crawler.SetInitPosition(ChCoordsys<>(initLoc, initRot));
//ihc_crawler.SetPowertrainType(powertrain_model);
//ihc_crawler.SetDriveType(drive_type);
ihc_crawler.SetTireType(tire_model);
ihc_crawler.SetTireStepSize(tire_step_size);
ihc_crawler.SetVehicleStepSize(step_size);
ihc_crawler.Initialize();

VisualizationType tire_vis_type = VisualizationType::MESH;// :
 ↪  VisualizationType::PRIMITIVES;

ihc_crawler.SetChassisVisualizationType(chassis_vis_type);
ihc_crawler.SetSuspensionVisualizationType(suspension_vis_type);
ihc_crawler.SetSteeringVisualizationType(steering_vis_type);
ihc_crawler.SetWheelVisualizationType(wheel_vis_type);
ihc_crawler.SetTireVisualizationType(tire_vis_type);

// Create the terrain
RigidTerrain terrain(ihc_crawler.GetSystem());

std::shared_ptr<RigidTerrain::Patch> patch;
switch (terrain_model) {
  case RigidTerrain::BOX:
    patch = terrain.AddPatch(ChCoordsys<>(ChVector<>(0, 0, terrainHeight - 5), QUNIT),
                             ChVector<>(terrainLength, terrainWidth, 10));
    patch->SetTexture(vehicle::GetDataFile("terrain/textures/tile4.jpg"), 200, 200);
    break;
  case RigidTerrain::HEIGHT_MAP:
    patch = terrain.AddPatch(CSYSNORM,
     ↪  vehicle::GetDataFile("terrain/height_maps/test64.bmp"), "test64", 128,
                             128, 0, 4);
    patch->SetTexture(vehicle::GetDataFile("terrain/textures/grass.jpg"), 16, 16);
    break;
  case RigidTerrain::MESH:
    patch = terrain.AddPatch(CSYSNORM, vehicle::GetDataFile("terrain/meshes/test.obj"),
     ↪  "test_mesh");
    patch->SetTexture(vehicle::GetDataFile("terrain/textures/grass.jpg"), 100, 100);
    break;
}
patch->SetContactFrictionCoefficient(0.9f);
patch->SetContactRestitutionCoefficient(0.01f);
patch->SetContactMaterialProperties(2e7f, 0.3f);
patch->SetColor(ChColor(0.8f, 0.8f, 0.5f));
terrain.Initialize();

// Create the vehicle Irrlicht interface
ChWheeledVehicleIrrApp app(&ihc_crawler.GetVehicle(), &ihc_crawler.GetPowertrain(),
 ↪  L"Crawler Demo");
app.SetSkyBox();
app.AddTypicalLights(irr::core::vector3df(30.f, -30.f, 100.f), irr::core::vector3df(30.f,
 ↪  50.f, 100.f), 250, 130);
app.SetChaseCamera(trackPoint, 6.0, 0.5);
app.SetTimestep(step_size);
```

```
app.AssetBindAll();
app.AssetUpdateAll();

// ---------
// Initialize output
// ---------

if (!filesystem::create_directory(filesystem::path(out_dir))) {
  std::cout « "Error creating directory " « out_dir « std::endl;
  return 1;
}
if (povray_output) {
  if (!filesystem::create_directory(filesystem::path(pov_dir))) {
    std::cout « "Error creating directory " « pov_dir « std::endl;
    return 1;
  }
  terrain.ExportMeshPovray(out_dir);
}

std::string driver_file = out_dir + "/driver_inputs.txt";
utils::CSV_writer driver_csv(" ");

// ------------
// Create the driver system
// ------------

// Create the interactive driver system
ChIrrGuiDriver driver(app);

// Set the time response for steering and throttle keyboard inputs.
double steering_time = 1.0;  // time to go from 0 to +1 (or from 0 to -1)
double throttle_time = 1.0;  // time to go from 0 to +1
double braking_time = 0.3;   // time to go from 0 to +1
driver.SetSteeringDelta(render_step_size / steering_time);
driver.SetThrottleDelta(render_step_size / throttle_time);
driver.SetBrakingDelta(render_step_size / braking_time);

// If in playback mode, attach the data file to the driver system and
// force it to playback the driver inputs.
if (driver_mode == PLAYBACK) {
  driver.SetInputDataFile(driver_file);
  driver.SetInputMode(ChIrrGuiDriver::DATAFILE);
}

driver.Initialize();

// --------
// Simulation loop
// --------

if (debug_output) {
  GetLog() « "\n\n============ System Configuration ============\n";
  ihc_crawler.LogHardpointLocations();
}

//output vehicle mass
std::cout«"VEHICLE MASS: "«my_sedan.GetVehicle().GetVehicleMass()«std::endl;

// Number of simulation steps between miscellaneous events
int render_steps = (int)std::ceil(render_step_size / step_size);
int debug_steps = (int)std::ceil(debug_step_size / step_size);
```

```
// Initialize simulation frame counter and simulation time
ChRealtimeStepTimer realtime_timer;
int step_number = 0;
int render_frame = 0;
double time = 0;

if (contact_vis) {
  app.SetSymbolscale(1e-4);
  app.SetContactsDrawMode(ChIrrTools::eCh_ContactsDrawMode::CONTACT_FORCES);
}

while (app.GetDevice()->run()) {
  time = ihc_crawler.GetSystem()->GetChTime();

  // End simulation
  if (time >= t_end)
    break;

  // Render scene and output POV-Ray data
  if (step_number % render_steps == 0) {
    app.BeginScene(true, true, irr::video::SColor(255, 140, 161, 192));
    app.DrawAll();
    app.EndScene();

    if (povray_output) {
      char filename[100];
      sprintf(filename, "%s/data_%03d.dat", pov_dir.c_str(), render_frame + 1);
      utils::WriteShapesPovray(ihc_crawler.GetSystem(), filename);
    }

    render_frame++;
  }

  // Debug logging
  if (debug_output && step_number % debug_steps == 0) {
    GetLog() << "\n\n============ System Information ============\n";
    GetLog() << "Time = " << time << "\n\n";
    ihc_crawler.DebugLog(OUT_SPRINGS | OUT_SHOCKS | OUT_CONSTRAINTS);
  }

  // Collect output data from modules (for inter-module communication)
  double throttle_input = driver.GetThrottle();
  double steering_input = driver.GetSteering();
  double braking_input = driver.GetBraking();

  // Driver output
  if (driver_mode == RECORD) {
    driver_csv << time << steering_input << throttle_input << braking_input << std::endl;
  }

  // Update modules (process inputs from other modules)
  driver.Synchronize(time);
  terrain.Synchronize(time);
  ihc_crawler.Synchronize(time, steering_input, braking_input, throttle_input, terrain);
  app.Synchronize(driver.GetInputModeAsString(), steering_input, throttle_input,
   ↪  braking_input);

  // Advance simulation for one timestep for all modules
  double step = realtime_timer.SuggestSimulationStep(step_size);
  driver.Advance(step);
  terrain.Advance(step);
  ihc_crawler.Advance(step);
```

```
    app.Advance(step);

    // Increment frame number
    step_number++;
  }

  if (driver_mode == RECORD) {
    driver_csv.write_to_file(driver_file);
  }

  return 0;
}
```

LISTING F.2: CHCRAWLERVEHICLE.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_CHCRAWLERVEHICLE_H
#define CHRONO_CHCRAWLERVEHICLE_H

#include <vector>

#include "chrono_vehicle/ChVehicle.h"
#include "chrono_vehicle/wheeled_vehicle/ChWheel.h"
#include "chrono_vehicle/wheeled_vehicle/ChDriveline.h"
#include "chrono_vehicle/wheeled_vehicle/ChSteering.h"
#include "chrono_vehicle/wheeled_vehicle/ChSuspension.h"

namespace chrono {
namespace vehicle {

/// @addtogroup vehicle_crawler
/// @{

/// Base class for chrono crawler vehicle systems.
/// This class provides the interface between the vehicle system and other
/// systems (tires, driver, etc.).
/// The reference frame for a vehicle follows the ISO standard: Z-axis up, X-axis
/// pointing forward, and Y-axis towards the left of the vehicle.
class CH_VEHICLE_API ChCrawlerVehicle : public ChVehicle {
 public:
```

```
/// Construct a vehicle system with a default ChSystem.
ChCrawlerVehicle(const std::string &name,
 ↪  ///< [in] vehicle name
                 ChMaterialSurface::ContactMethod contact_method = ChMaterialSurface::NSC
                  ↪  ///< [in] contact method
);

/// Construct a vehicle system using the specified ChSystem.
ChCrawlerVehicle(const std::string &name, ///< [in] vehicle name
                 ChSystem *system         ///< [in] containing mechanical system
);

/// Destructor.
virtual ~ChCrawlerVehicle() = default;

/// Get the name of the vehicle system template.
virtual std::string GetTemplateName() const override { return "CrawlerVehicle"; }

/// Get the specified suspension subsystem.
std::shared_ptr<ChSuspension> GetSuspension(int id) const { return m_suspensions[id]; }

/// Get the specified steering subsystem.
std::shared_ptr<ChSteering> GetSteering(int id) const { return m_steerings[id]; }

/// Get a handle to the specified vehicle wheel subsystem.
std::shared_ptr<ChWheel> GetWheel(const WheelID &wheel_id) const { return
 ↪  m_wheels[wheel_id.id()]; }

/// Get a handle to the vehicle's driveline subsystem.
std::shared_ptr<ChDriveline> GetDriveline() const { return m_driveline; }

/// Get the vehicle total mass.
/// This includes the mass of the chassis and all vehicle subsystems, but not the mass of
 ↪  tires.
virtual double GetVehicleMass() const override;

/// Get the current global vehicle COM location.
virtual ChVector<> GetVehicleCOMPos() const override;

/// Get a handle to the vehicle's driveshaft body.
virtual std::shared_ptr<ChShaft> GetDriveshaft() const override { return
 ↪  m_driveline->GetDriveshaft(); }

/// Get the angular speed of the driveshaft.
/// This function provides the interface between a vehicle system and a
/// powertrain system.
virtual double GetDriveshaftSpeed() const override;

/// Return the number of axles for this vehicle.
virtual int GetNumberAxles() const = 0;

/// Get a handle to the specified wheel body.
std::shared_ptr<ChBody> GetWheelBody(const WheelID &wheel_id) const;

/// Get the global location of the specified wheel.
const ChVector<> &GetWheelPos(const WheelID &wheel_id) const;

/// Get the orientation of the specified wheel.
/// The wheel orientation is returned as a quaternion representing a rotation
/// with respect to the global reference frame.
const ChQuaternion<> &GetWheelRot(const WheelID &wheel_id) const;
```

```
/// Get the linear velocity of the specified wheel.
/// Return the linear velocity of the wheel center, expressed in the global
/// reference frame.
const ChVector<> &GetWheelLinVel(const WheelID &wheel_id) const;

/// Get the angular velocity of the specified wheel.
/// Return the angular velocity of the wheel frame, expressed in the global
/// reference frame.
ChVector<> GetWheelAngVel(const WheelID &wheel_id) const;

/// Get the angular speed of the specified wheel.
/// This is the angular speed of the wheel axle.
double GetWheelOmega(const WheelID &wheel_id) const;

/// Get the complete state for the specified wheel.
/// This includes the location, orientation, linear and angular velocities,
/// all expressed in the global reference frame, as well as the wheel angular
/// speed about its rotation axis.
WheelState GetWheelState(const WheelID &wheel_id) const;

/// Return the vehicle wheelbase.
virtual double GetWheelbase() const = 0;

/// Return the vehicle wheel track of the specified suspension subsystem.
double GetWheeltrack(int id) const { return m_suspensions[id]->GetTrack(); }

/// Return the minimum turning radius.
/// A concrete wheeled vehicle class should override the default value (20 m).
virtual double GetMinTurningRadius() const { return 20; } //TODO: determine min turning
 ↪  radius

/// Return the maximum steering angle.
/// This default implementation estimates the maximum steering angle based on a bicycle
 ↪  model
/// and the vehicle minimum turning radius.
virtual double GetMaxSteeringAngle() const; //TODO: determine max turning radius

/// Set visualization type for the suspension subsystems.
/// This function should be called only after vehicle initialization.
void SetSuspensionVisualizationType(VisualizationType vis);

/// Set visualization type for the steering subsystems.
/// This function should be called only after vehicle initialization.
void SetSteeringVisualizationType(VisualizationType vis);

/// Set visualization type for the wheel subsystems.
/// This function should be called only after vehicle initialization.
void SetWheelVisualizationType(VisualizationType vis);

/// Enable/disable collision between the chassis and all other vehicle subsystems.
/// This only controls collisions between the chassis and the tire systems.
virtual void SetChassisVehicleCollide(bool state = False) override;

/// Enable/disable output from the suspension subsystems.
void SetSuspensionOutput(int id, bool state);

/// Enable/disable output from the steering subsystems.
void SetSteeringOutput(int id, bool state);

/// Enable/disable output from the driveline subsystem.
void SetDrivelineOutput(bool state);
```

```
  /// Initialize this vehicle at the specified global location and orientation.
  /// This base class implementation only initializes the chassis subsystem.
  /// Derived classes must extend this function to initialize all other wheeled
  /// vehicle subsystems (steering, suspensions, wheels and driveline).
  virtual void Initialize(const ChCoordsys<> &chassisPos,  ///< [in] initial global position
   ↪  and orientation
                          double chassisFwdVel = 0         ///< [in] initial chassis forward
                           ↪  velocity
  ) override;

  /// Update the state of this vehicle at the current time.
  /// The vehicle system is provided the current driver inputs (throttle between
  /// 0 and 1, steering between -1 and +1, braking between 0 and 1), the torque
  /// from the powertrain, and tire forces (expressed in the global reference
  /// frame).
  virtual void Synchronize(double time,                      ///< [in] current time
                           double steering,                  ///< [in] current steering input
                            ↪  [-1,+1]
                           double powertrain_torque,         ///< [in] input torque from
                            ↪  powertrain
                           const TerrainForces &tire_forces  ///< [in] vector of tire force
                            ↪  structures
  );

  /// Log current constraint violations.
  virtual void LogConstraintViolations() override;

  /// Return a JSON string with information on all modeling components in the vehicle system.
  /// These include bodies, shafts, joints, spring-damper elements, markers, etc.
  virtual std::string ExportComponentList() const override;

  /// Write a JSON-format file with information on all modeling components in the vehicle
   ↪  system.
  /// These include bodies, shafts, joints, spring-damper elements, markers, etc.
  virtual void ExportComponentList(const std::string &filename) const override;

  /// Output data for all modeling components in the vehicle system.
  virtual void Output(int frame, ChVehicleOutput &database) const override;

 protected:
  ChSuspensionList m_suspensions;            ///< list of handles to suspension subsystems
  std::shared_ptr<ChDriveline> m_driveline;  ///< handle to the driveline subsystem
  ChSteeringList m_steerings;                ///< list of handles to steering subsystems
  ChWheelList m_wheels;                      ///< list of handles to wheel subsystems
}; /// ChCrawlerVehicle
} /// vehicle
} /// chrono

#endif //CHRONO_CHCRAWLERVEHICLE_H
```

LISTING F.3: CHCRAWLERVEHICLE.CPP

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
```

```
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//
#include <fstream>

#include "ChCrawlerVehicle.h"

#include "chrono_thirdparty/rapidjson/document.h"
#include "chrono_thirdparty/rapidjson/prettywriter.h"
#include "chrono_thirdparty/rapidjson/stringbuffer.h"

namespace chrono {
namespace vehicle {

ChCrawlerVehicle::ChCrawlerVehicle(const std::string &name, ChMaterialSurface::ContactMethod
 ↪  contact_method)
    : ChVehicle(name, contact_method) {}

ChCrawlerVehicle::ChCrawlerVehicle(const std::string &name, ChSystem *system) :
 ↪  ChVehicle(name, system) {}

// -----------------------------------
// Initialize this vehicle at the specified global location and orientation.
// This base class implementation only initializes the chassis subsystem.
// Derived classes must extend this function to initialize all other wheeled
// vehicle subsystems (steering, suspensions, wheels, and driveline).
// -----------------------------------
void ChCrawlerVehicle::Initialize(const ChCoordsys<> &chassisPos, double chassisFwdVel) {
  m_chassis->Initialize(m_system, chassisPos, chassisFwdVel,
   ↪  WheeledCollisionFamily::CHASSIS);
}

// -----------------------------------
// Update the state of this vehicle at the current time.
// The vehicle system is provided the current driver inputs (throttle between
// 0 and 1, steering between -1 and +1), the torque
// from the powertrain, and tire forces (expressed in the global reference
// frame).
// The default implementation of this function invokes the update functions for
// all vehicle subsystems.
// -----------------------------------
void ChCrawlerVehicle::Synchronize(double time,
                                   double steering,
                                   double powertrain_torque,
                                   const TerrainForces &tire_forces) {
  // Apply powertrain torque to the driveline's input shaft.
  m_driveline->Synchronize(powertrain_torque);

  // Let the steering subsystems process the steering input.
  for (unsigned int i = 0; i < m_steerings.size(); i++) {
    m_steerings[i]->Synchronize(time, steering);
  }
```

```
  // Apply tire forces to spindle bodies.
  for (unsigned int i = 0; i < m_suspensions.size(); i++) {
    m_suspensions[i]->Synchronize(LEFT, tire_forces[2 * i]);
    m_suspensions[i]->Synchronize(RIGHT, tire_forces[2 * i + 1]);
  }

  m_chassis->Synchronize(time);
}

// -----------------------------------
// Set visualization type for the various subsystems
// -----------------------------------
void ChCrawlerVehicle::SetSuspensionVisualizationType(VisualizationType vis) {
  for (size_t i = 0; i < m_suspensions.size(); ++i) {
    m_suspensions[i]->SetVisualizationType(vis);
  }
}

void ChCrawlerVehicle::SetSteeringVisualizationType(VisualizationType vis) {
  for (size_t i = 0; i < m_steerings.size(); ++i) {
    m_steerings[i]->SetVisualizationType(vis);
  }
}

void ChCrawlerVehicle::SetWheelVisualizationType(VisualizationType vis) {
  for (size_t i = 0; i < m_wheels.size(); ++i) {
    m_wheels[i]->SetVisualizationType(vis);
  }
}

// -----------------------------------
// Enable/disable collision between the chassis and all other vehicle subsystems
// This only controls collisions between the chassis and the tire systems.
// -----------------------------------
void ChCrawlerVehicle::SetChassisVehicleCollide(bool state) {
  if (state) {
    // Chassis collides with tires
    m_chassis->GetBody()->GetCollisionModel()-
     ↪  >SetFamilyMaskDoCollisionWithFamily(WheeledCollisionFamily::TIRES);
  } else {
    // Chassis does not collide with tires
    m_chassis->GetBody()->GetCollisionModel()-
     ↪  >SetFamilyMaskNoCollisionWithFamily(WheeledCollisionFamily::TIRES);
  }
}

// -----------------------------------
// Enable/disable output from the various subsystems
// -----------------------------------
void ChCrawlerVehicle::SetSuspensionOutput(int id, bool state) {
  m_suspensions[id]->SetOutput(state);
}

void ChCrawlerVehicle::SetSteeringOutput(int id, bool state) {
  m_steerings[id]->SetOutput(state);
}

void ChCrawlerVehicle::SetDrivelineOutput(bool state) {
  m_driveline->SetOutput(state);
}

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

```
// Calculate and return the total vehicle mass
// ------------------------------------
double ChCrawlerVehicle::GetVehicleMass() const {
  double mass = m_chassis->GetMass();

  for (auto wheel : m_wheels) {
    mass += wheel->GetMass();
  }

  for (auto steering : m_steerings) {
    mass += steering->GetMass();
  }

  for (auto suspension : m_suspensions) {
    mass += suspension->GetMass();
  }

  return mass;
}

// ------------------------------------
// Calculate and return the current vehicle COM location
// ------------------------------------
ChVector<> ChCrawlerVehicle::GetVehicleCOMPos() const {
  ChVector<> com(0, 0, 0);
  com += m_chassis->GetMass() * m_chassis->GetCOMPos();

  for (auto wheel: m_wheels) {
    com += wheel->GetMass() * wheel->GetCOMPos();
  }

  for (auto steering : m_steerings) {
    com += steering->GetMass() * steering->GetCOMPos();
  }

  for (auto suspension : m_suspensions) {
    com += suspension->GetMass() * suspension->GetCOMPos();
  }

  return com / GetVehicleMass();
}

// ------------------------------------
// ------------------------------------
std::shared_ptr<ChBody> ChCrawlerVehicle::GetWheelBody(const WheelID &wheel_id) const {
  return m_suspensions[wheel_id.axle()]->GetSpindle(wheel_id.side());
}

const ChVector<> &ChCrawlerVehicle::GetWheelPos(const WheelID &wheel_id) const {
  return m_suspensions[wheel_id.axle()]->GetSpindlePos(wheel_id.side());
}

const ChQuaternion<> &ChCrawlerVehicle::GetWheelRot(const WheelID &wheel_id) const {
  return m_suspensions[wheel_id.axle()]->GetSpindleRot(wheel_id.side());
}

const ChVector<> &ChCrawlerVehicle::GetWheelLinVel(const WheelID &wheel_id) const {
  return m_suspensions[wheel_id.axle()]->GetSpindleLinVel(wheel_id.side());
}

ChVector<> ChCrawlerVehicle::GetWheelAngVel(const WheelID &wheel_id) const {
  return m_suspensions[wheel_id.axle()]->GetSpindleAngVel(wheel_id.side());
```

```
}

double ChCrawlerVehicle::GetWheelOmega(const WheelID &wheel_id) const {
  return m_suspensions[wheel_id.axle()]->GetAxleSpeed(wheel_id.side());
}

// ---------------------------------------
// Return the complete state (expressed in the global frame) for the specified
// wheel body.
// ---------------------------------------
WheelState ChCrawlerVehicle::GetWheelState(const WheelID &wheel_id) const {
  WheelState state;

  state.pos = GetWheelPos(wheel_id);
  state.rot = GetWheelRot(wheel_id);
  state.lin_vel = GetWheelLinVel(wheel_id);
  state.ang_vel = GetWheelAngVel(wheel_id);

  ChVector<> ang_vel_loc = state.rot.RotateBack(state.ang_vel);
  state.omega = ang_vel_loc.y();

  return state;
}

// ---------------------------------------
// ---------------------------------------
double ChCrawlerVehicle::GetDriveshaftSpeed() const {
  return m_driveline->GetDriveshaftSpeed();
}

// ---------------------------------------
// Estimate the maximum steering angle based on a bicycle model, from the vehicle
// minimum turning radius, the wheelbase, and the track of the front suspension.
// TODO: determine accurate max steering angle
// ---------------------------------------
double ChCrawlerVehicle::GetMaxSteeringAngle() const {
  return std::asin(GetWheelbase() / (GetMinTurningRadius() - 0.5 * GetWheeltrack(0)));
}

// ---------------------------------------
// Log constraint violations
// ---------------------------------------
void ChCrawlerVehicle::LogConstraintViolations() {
  GetLog().SetNumFormat("%16.4e");

  // Report constraint violations for the suspension joints
  for (size_t i = 0; i < m_suspensions.size(); i++) {
    GetLog() << "\n-- AXLE " << i << " LEFT side suspension constraint violations\n\n";
    m_suspensions[i]->LogConstraintViolations(LEFT);
    GetLog() << "\n-- AXLE " << i << " RIGHT side suspension constraint violations\n\n";
    m_suspensions[i]->LogConstraintViolations(RIGHT);
  }

  // Report constraint violations for the steering joints
  for (size_t i = 0; i < m_steerings.size(); i++) {
    GetLog() << "\n-- STEERING subsystem " << i << " constraint violations\n\n";
    m_steerings[i]->LogConstraintViolations();
  }

  GetLog().SetNumFormat("%g");
}
```

```
std::string ChCrawlerVehicle::ExportComponentList() const {
  rapidjson::Document jsonDocument;
  jsonDocument.SetObject();

  std::string template_name = GetTemplateName();
  jsonDocument.AddMember("name", rapidjson::StringRef(m_name.c_str()),
   ↪  jsonDocument.GetAllocator());
  jsonDocument.AddMember("template", rapidjson::Value(template_name.c_str(),
   ↪  jsonDocument.GetAllocator()).Move(),
                         jsonDocument.GetAllocator());

  {
    rapidjson::Document jsonSubDocument(&jsonDocument.GetAllocator());
    jsonSubDocument.SetObject();
    m_chassis->ExportComponentList(jsonSubDocument);
    jsonDocument.AddMember("chassis", jsonSubDocument, jsonDocument.GetAllocator());
  }

  rapidjson::Value suspArray(rapidjson::kArrayType);
  for (auto suspension : m_suspensions) {
    rapidjson::Document jsonSubDocument(&jsonDocument.GetAllocator());
    jsonSubDocument.SetObject();
    suspension->ExportComponentList(jsonSubDocument);
    suspArray.PushBack(jsonSubDocument, jsonDocument.GetAllocator());
  }
  jsonDocument.AddMember("suspension", suspArray, jsonDocument.GetAllocator());

  rapidjson::Value sterringArray(rapidjson::kArrayType);
  for (auto steering : m_steerings) {
    rapidjson::Document jsonSubDocument(&jsonDocument.GetAllocator());
    jsonSubDocument.SetObject();
    steering->ExportComponentList(jsonSubDocument);
    sterringArray.PushBack(jsonSubDocument, jsonDocument.GetAllocator());
  }
  jsonDocument.AddMember("steering", sterringArray, jsonDocument.GetAllocator());

  rapidjson::StringBuffer jsonBuffer;
  rapidjson::PrettyWriter<rapidjson::StringBuffer> jsonWriter(jsonBuffer);
  jsonDocument.Accept(jsonWriter);

  return jsonBuffer.GetString();
}

void ChCrawlerVehicle::ExportComponentList(const std::string &filename) const {
  std::ofstream of(filename);
  of « ExportComponentList();
  of.close();
}

void ChCrawlerVehicle::Output(int frame, ChVehicleOutput &database) const {
  database.WriteTime(frame, m_system->GetChTime());

  if (m_chassis->OutputEnabled()) {
    database.WriteSection(m_chassis->GetName());
    m_chassis->Output(database);
  }

  for (auto suspension : m_suspensions) {
    if (suspension->OutputEnabled()) {
      database.WriteSection(suspension->GetName());
      suspension->Output(database);
    }
```

```
  }

  for (auto steering : m_steerings) {
    if (steering->OutputEnabled()) {
      database.WriteSection(steering->GetName());
      steering->Output(database);
    }
  }
}

} /// vehicle
} /// chrono
```

LISTING F.4: CRAWLERVEHICLE.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_CRAWLER_VEHICLE_H
#define CHRONO_CRAWLER_VEHICLE_H

#include <vector>

#include "chrono/core/ChCoordsys.h"
#include "chrono/physics/ChMaterialSurface.h"
#include "chrono/physics/ChSystem.h"

#include "ChCrawlerVehicle.h"

#include "chrono_models/ChApiModels.h"
#include "chrono_models/vehicle/ChVehicleModelDefs.h"

#include "Crawler_Chassis.h"
#include "Crawler_Driveline.h"

namespace chrono {
namespace vehicle {
namespace crawler {
class CH_MODELS_API Crawler_vehicle : public ChCrawlerVehicle {
 public:
  Crawler_vehicle(const bool fixed = false,
                  ChMaterialSurface::ContactMethod contact_method = ChMaterialSurface::NSC,
```

```
                ChassisCollisionType chassis_collision_type = ChassisCollisionType::NONE);

  Crawler_vehicle(ChSystem *system,
                const bool fixed = false,
                ChassisCollisionType chassis_collision_type = ChassisCollisionType::NONE);

  ~Crawler_vehicle();

  virtual int GetNumberAxles() const override { return 2; } //TODO: check if it should be 1

  virtual double GetWheelbase() const override { return 3.0; } //TODO: What is the actual
   ↪  wheel base for the crawler?

  virtual double GetMinTurningRadius() const override { return 2.0; } //TODO: What is the
   ↪  actual min turning radius?

  double GetMaxSteeringAngle() const override { return 25.0 * CH_C_DEG_TO_RAD; } //TODO:
   ↪  check actual value

  void SetInitWheelAngVel(const std::vector<double> &omega) {
    assert(omega.size() == 2);
    m_omega = omega;
  }

  virtual void Initialize(const ChCoordsys<> &chassisPos, double chassisFwdVel = 0) override;

  // Log debugging information
  void LogHardpointLocations(); /// suspension hardpoints at design
  void DebugLog(int what); /// forces and lengths, contsrainets, etc.

 private:
  void Create(bool fixed, ChassisCollisionType chassis_collision_type);

  std::vector<double> m_omega;
}; /// Crawler_vehicle
} /// crawler
} /// vehicle
} /// chrono

#endif //CHRONO_CRAWLER_VEHICLE_H
```

LISTING F.5: CRAWLERVEHICLE.CPP

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
```

```
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#include "chrono/assets/ChSphereShape.h"
#include "chrono/assets/ChTriangleMeshShape.h"
#include "chrono/utils/ChUtilsInputOutput.h"

#include "chrono_vehicle/ChVehicleModelData.h"

#include "Crawler_vehicle.h"

namespace chrono {
namespace vehicle {
namespace crawler {
Crawler_vehicle::Crawler_vehicle(const bool fixed,
                                 ChMaterialSurface::ContactMethod contact_method,
                                 ChassisCollisionType chassis_collision_type)
    : ChCrawlerVehicle("Crawler", contact_method), m_omega({0, 0}) {
  Create(fixed, chassis_collision_type);
}

Crawler_vehicle::Crawler_vehicle(ChSystem *system, const bool fixed, ChassisCollisionType
 ↪  chassis_collision_type)
    : ChCrawlerVehicle("Crawler", system), m_omega({0, 0}) {
  Create(fixed, chassis_collision_type);
}

void Crawler_vehicle::Create(bool fixed, ChassisCollisionType chassis_collision_type) {
  // ----------------------
  // Create the chassis subsystem
  // ----------------------
  m_chassis = std::make_shared<Crawler_Chassis>("Chassis", fixed, chassis_collision_type);

  // ----------------------
  // Create the suspension subsystems
  // ----------------------
  m_suspensions.resize(2);
  m_suspensions[0] = std::make_shared<Sedan_DoubleWishbone>("FrontSusp");
  m_suspensions[1] = std::make_shared<Sedan_MultiLink>("RearSusp");

  // ---------------
  // Create the steering subsystem
  // ---------------
  m_steerings.resize(1);
  m_steerings[0] = std::make_shared<Sedan_RackPinion>("Steering");

  // ---------
  // Create the wheels
  // ---------
  m_wheels.resize(4);
  m_wheels[0] = std::make_shared<Sedan_WheelLeft>("Wheel_FL");
  m_wheels[1] = std::make_shared<Sedan_WheelRight>("Wheel_FR");
  m_wheels[2] = std::make_shared<Sedan_WheelLeft>("Wheel_RL");
  m_wheels[3] = std::make_shared<Sedan_WheelRight>("Wheel_RR");

  // ----------
  // Create the driveline
  // ----------
  m_driveline = std::make_shared<Sedan_Driveline2WD>("Driveline");
}
```

```
} /// crawler
} /// vehicle
} /// chrono
```

LISTING F.6: CRAWLER.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_CRAWLER_H
#define CHRONO_CRAWLER_H

#include <array>
#include <string>

#include "chrono_models/ChApiModels.h"
#include "Crawler_vehicle.h"
#include "Crawler_SimpleFluidPowertrain.h"
#include "Crawler_ArchmidesTire.h"

namespace chrono {
namespace vehicle {
namespace crawler {

class CH_MODELS_API Crawler {
 public:
  Crawler();
  Crawler(ChSystem *system);

  ~Crawler();

  void SetContactMethod(ChMaterialSurface::ContactMethod val) { m_contactMethod = val; }

  void SetChassisFixed(bool val) { m_fixed = true; }
  void SetChassisCollisionType(ChassisCollisionType val) { m_chassisCollisionType = val; }

  void SetTireType(TireModelType val) { m_tireType = val; }

  void SetInitPosition(const ChCoordsys<> &pos) { m_initPos = pos; }
  void SetInitFwdVel(double fwdVel) { m_initFwdVel = fwdVel; }
  void SetInitWheelAngVel(const std::vector<double> &omega) { m_initOmega = omega; }
```

```
  void SetVehicleStepSize(double step_size) { m_vehicle_step_size = step_size; }
  void SetTireStepSize(double step_size) { m_tire_step_size = step_size; }

  ChSystem *GetSystem() const { return m_vehicle->GetSystem(); }
  ChCrawlerVehicle &GetVehicle() const { return *m_vehicle; }
  std::shared_ptr<ChChassis> GetChassis() const { return m_vehicle->GetChassis(); }
  std::shared_ptr<ChBodyAuxRef> GetChassisBody() const { return m_vehicle->GetChassisBody();
   ↪  }
  ChPowertrain &GetPowertrain() const { return *m_powertrain; }
  ChTire *GetTire(WheelID which) const { return m_tires[which.id()]; }
  double GetTotalMass() const;

  void Initialize();

//  void LockAxleDifferential(int axle, bool lock) { m_vehicle->LockAxleDifferential(axle,
 ↪  lock); }

  void SetAerodynamicDrag(double Cd, double area, double water_density);

  void SetChassisVisualizationType(VisualizationType vis) {
   ↪  m_vehicle->SetChassisVisualizationType(vis); }
  void SetSuspensionVisualizationType(VisualizationType vis) {
   ↪  m_vehicle->SetSuspensionVisualizationType(vis); }
  void SetSteeringVisualizationType(VisualizationType vis) {
   ↪  m_vehicle->SetSteeringVisualizationType(vis); }
  void SetWheelVisualizationType(VisualizationType vis) {
   ↪  m_vehicle->SetWheelVisualizationType(vis); }
  void SetTireVisualizationType(VisualizationType vis);

  void Synchronize(double time,
                   double steering_input,
                   double throttle_input,
                   const ChTerrain &terrain);

  void Advance(double step);

  void LogHardpointLocations() { m_vehicle->LogHardpointLocations(); }
  void DebugLog(int what) { m_vehicle->DebugLog(what); }

 protected:
  ChMaterialSurface::ContactMethod m_contactMethod;
  ChassisCollisionType m_chassisCollisionType;
  bool m_fixed;

  TireModelType m_tireType;

  double m_vehicle_step_size;
  double m_tire_step_size;

  ChCoordsys<> m_initPos;
  double m_initFwdVel;
  std::vector<double> m_initOmega;

  bool m_apply_drag;
  double m_Cd;
  double m_area;
  double m_water_density;

  ChSystem *m_system;
  Crawler_vehicle *m_vehicle;
  ChPowertrain *m_powertrain;
  std::array<ChTire *, 2> m_tires;
```

```
  double m_tire_mass;
};

}
}
}

#endif //CHRONO_CRAWLER_H
```

LISTING F.7: CRAWLER.CPP

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#include "chrono/ChConfig.h"

#include "chrono_vehicle/ChVehicleModelData.h"

#include "Crawler.h"

namespace chrono {
namespace vehicle {
namespace crawler {

// -----------------------------------
Crawler::Crawler()
    : m_system(NULL),
      m_vehicle(NULL),
      m_powertrain(NULL),
      m_tires({{NULL, NULL}}),
      m_contactMethod(ChMaterialSurface::NSC),
      m_chassisCollisionType(ChassisCollisionType::NONE),
      m_fixed(false),
      m_tireType(TireModelType::RIGID),
      m_vehicle_step_size(-1),
      m_tire_step_size(-1),
      m_initFwdVel(0),
      m_initPos(ChCoordsys<>(ChVector<>(0, 0, 1), QUNIT)),
      m_initOmega({0, 0, 0, 0}),
      m_apply_drag(false) {}
```

```
Crawler::Crawler(ChSystem *system)
    : m_system(system),
      m_vehicle(NULL),
      m_powertrain(NULL),
      m_tires({{NULL, NULL}}),
      m_contactMethod(ChMaterialSurface::NSC),
      m_chassisCollisionType(ChassisCollisionType::NONE),
      m_fixed(false),
      m_tireType(TireModelType::RIGID),
      m_vehicle_step_size(-1),
      m_tire_step_size(-1),
      m_initFwdVel(0),
      m_initPos(ChCoordsys<>(ChVector<>(0, 0, 1), QUNIT)),
      m_initOmega({0, 0, 0, 0}),
      m_apply_drag(false) {}

Crawler::~Crawler() {
  delete m_vehicle;
  delete m_powertrain;
  delete m_tires[0];
  delete m_tires[1];
}

void Crawler::SetAerodynamicDrag(double Cd, double area, double water_density) {
  m_Cd = Cd;
  m_area = area;
  m_water_density = water_density;

  m_apply_drag = true;
}

void Crawler::Initialize() {
  // Create and initialize the crawler vehicle
  m_vehicle = m_system ? new Crawler_vehicle(m_system, m_fixed, m_chassisCollisionType)
                       : new Crawler_vehicle(m_fixed, m_contactMethod,
                        ↪  m_chassisCollisionType);

  m_vehicle->SetInitWheelAngVel(m_initOmega);
  m_vehicle->Initialize(m_initPos, m_initFwdVel);

  if (m_vehicle_step_size > 0) {
    m_vehicle->SetStepsize(m_vehicle_step_size);
  }

  // If specified, enable aerodynamic drag
  if (m_apply_drag) {
    m_vehicle->GetChassis()->SetAerodynamicDrag(m_Cd, m_area, m_water_density);
  }

  // Create and initialize the powertrain system
  m_powertrain
  new Crawler_SimpleFluidPowertrain('Powertrain');
  m_powertrain->Initialize(GetChassisBody(), m_vehicle->GetDriveshaft());

  // Create the Archimedes tires and set parameters depending on type
  switch (m_tireType) {
    case TireModelType::RIGID: {
      GetLog() « "Init RIGID" « "\n";
      bool use_mesh = (m_tireType == TireModelType::RIGID_MESH);
      Crawler_ArchmidesTire *tire_L = new Crawler_ArchmidesTire("L", use_mesh);
      Crawler_ArchmidesTire *tire_R = new Crawler_ArchmidesTire("R", use_mesh);
```

```
      m_tires[0] = tire_L;
      m_tires[1] = tire_R;

      break;
    }
    default: {
      break;
    }
  }

  // Initialize the tires
  m_tires[0]->Initialize(m_vehicle->GetWheelBody(FRONT_LEFT), LEFT);
  m_tires[1]->Initialize(m_vehicle->GetWheelBody(FRONT_RIGHT), RIGHT);

  m_tire_mass = m_tires[0]->ReportMass();
}

void Crawler::SetTireVisualizationType(VisualizationType vis) {
  for (auto Tire : m_tires) {
    Tire->SetVisualizationType(vis);
  }
}

void Crawler::Synchronize(double time,
                          double steering_input,
                          double throttle_input,
                          const ChTerrain &terrain) {
  TerrainForces tire_forces(2);
  WheelState wheel_states[2];

  tire_forces[0] = m_tires[0]->GetTireForce();
  tire_forces[1] = m_tires[1]->GetTireForce();

  wheel_states[0] = m_vehicle->GetWheelState(FRONT_LEFT);
  wheel_states[1] = m_vehicle->GetWheelState(FRONT_RIGHT);

  double powertrain_torque = m_powertrain->GetOutputTorque();

  double driveshaft_speed = m_vehicle->GetDriveshaftSpeed();

  m_tires[0]->Synchronize(time, wheel_states[0], terrain);
  m_tires[1]->Synchronize(time, wheel_states[1], terrain);

  m_powertrain->Synchronize(time, throttle_input, driveshaft_speed);

  m_vehicle->Synchronize(time, steering_input, powertrain_torque, tire_forces);
}

void Crawler::Advance(double step) {
  for (auto Tire : m_tires) {
    Tire->Advance(step);
  }

  m_powertrain->Advance(step);

  m_vehicle->Advance(step);
}

double Crawler::GetTotalMass() const {
  return m_vehicle->GetVehicleMass() + 2 * m_tire_mass;
}
```

```
} /// crawler
} /// vehicle
} /// chrono
```

LISTING F.8: CRAWLERCHASSIS.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_CRAWLER_CHASSIS_H
#define CHRONO_CRAWLER_CHASSIS_H

#include <string>

#include "chrono_vehicle/chassis/ChRigidChassis.h"

#include "chrono_models/ChApiModels.h"
#include "chrono_models/vehicle/ChVehicleModelDefs.h"

namespace chrono {
namespace vehicle {
namespace crawler {
class CH_MODELS_API Crawler_Chassis : public ChRigidChassis {
 public:
  Crawler_Chassis(const std::string &name,
                  bool fixed = false,
                  ChassisCollisionType chassis_collision_type = ChassisCollisionType::NONE);

  ~Crawler_Chassis() = default;

  /// Return the mass of the chassis body.
  virtual double GetMass() const override { return m_mass; }

  /// Return the inertia tensor of the chassis body.
  virtual const ChMatrix33<> &GetInertia() const override { return m_inertia; }

  /// Get the location of the center of mass in the chassis frame.
  virtual const ChVector<> &GetLocalPosCOM() const override { return m_COM_loc; }

 protected:
  ChMatrix33<> m_inertia;
```

```
  static const double m_mass;
  static const ChVector<> m_inertiaXX;
  static const ChVector<> m_inertiaXY;
  static const ChVector<> m_COM_loc;
}; /// Crawler_Chassis
} /// crawler
} /// vehicle
} /// chrono

#endif //CHRONO_CRAWLER_CHASSIS_H
```

LISTING F.9: CRAWLERCHASSIS.CPP

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//
#include "chrono/assets/ChTriangleMeshShape.h"
#include "chrono/utils/ChUtilsInputOutput.h"

#include "chrono_vehicle/ChVehicleModelData.h"

#include "Crawler_Chassis.h"

namespace chrono {
namespace vehicle {
namespace crawler {

// -----------------------------------
// Static variables
// -----------------------------------
const double Crawler_Chassis::m_mass = 1250; //Todo: actual mass of chassis
const ChVector<> Crawler_Chassis::m_inertiaXX(222.8, 944.1, 1053.5); // Todo: actual inertia
 ↪  XX
const ChVector<> Crawler_Chassis::m_inertiaXY(0, 0, 0); // Todo: actual inertia XY
const ChVector<> Crawler_Chassis::m_COM_loc(0, 0, 0.2); // Todo: actual Center of Mass

Crawler_Chassis::Crawler_Chassis(const std::string &name, bool fixed, ChassisCollisionType
 ↪  chassis_collision_type)
    : ChRigidChassis(name, fixed) {
  m_inertia.SetElement(0, 0, m_inertiaXX.x());
  m_inertia.SetElement(1, 1, m_inertiaXX.y());
  m_inertia.SetElement(2, 2, m_inertiaXX.z());
```

```
  m_inertia.SetElement(0, 1, m_inertiaXY.x());
  m_inertia.SetElement(0, 2, m_inertiaXY.y());
  m_inertia.SetElement(1, 2, m_inertiaXY.z());
  m_inertia.SetElement(1, 0, m_inertiaXY.x());
  m_inertia.SetElement(2, 0, m_inertiaXY.y());
  m_inertia.SetElement(2, 1, m_inertiaXY.z());

  //// TODO: A more appropriate contact shape from primitives
  BoxShape box1(ChVector<>(0.0, 0.0, 0.1), ChQuaternion<>(1, 0, 0, 0), ChVector<>(1.0, 0.5,
   ↪  0.2));

  m_has_primitives = true;
  m_vis_boxes.push_back(box1);

  m_has_mesh = true;
  m_vis_mesh_name = "crawler_chassis_POV_geom";
  m_vis_mesh_file = "sedan/sedan_chassis_vis.obj"; // Todo: add correct mesh

  m_has_collision = (chassis_collision_type != ChassisCollisionType::NONE);
  switch (chassis_collision_type) {
    case ChassisCollisionType::PRIMITIVES:
      m_coll_boxes.push_back(box1);
      break;
    case ChassisCollisionType::MESH:
      m_coll_mesh_names.push_back("sedan/sedan_chassis_col.obj"); // Todo: add mesh of
       ↪  crawler chassis
      break;
    default:
      break;
  }
}

} /// crawler
} /// vehicle
} /// chrono
```

LISTING F.10: CRAWLERARCHMIDESTIRE.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_CRAWLER_ARCHMIDESTIRE_H
```

```
#define CHRONO_CRAWLER_ARCHMIDESTIRE_H

class Crawler_ArchmidesTire {

};

#endif //CHRONO_CRAWLER_ARCHMIDESTIRE_H
```

LISTING F.11: CRAWLERARCHMIDESTIRE.CPP

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#include "Crawler_ArchmidesTire.h"
```

LISTING F.12: CHHYDRAULICSTEERING.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_CHHYDRAULICSTEERING_H
#define CHRONO_CHHYDRAULICSTEERING_H
```

```
#include "chrono_vehicle/ChApiVehicle.h"
#include "chrono_vehicle/wheeled_vehicle/ChSteering.h"

namespace chrono {
namespace vehicle {
class CH_VEHICLE_API ChHydraulicSteering : public ChSteering {
 public:
  ChHydraulicSteering(const std::string &name);

  virtual ~ChHydraulicSteering() = default;

  /// Get the name of the vehicle subsystem template.
  virtual std::string GetTemplateName() const override { return "RotaryArm"; }

  /// Initialize this steering subsystem.
  /// The steering subsystem is initialized by attaching it to the specified
  /// chassis body at the specified location (with respect to and expressed in
  /// the reference frame of the chassis) and with specified orientation (with
  /// respect to the chassis reference frame).
  virtual void Initialize(std::shared_ptr<ChBodyAuxRef> chassis,  ///< [in] handle to the
   ↪  chassis body
                          const ChVector<> &location,             ///< [in] location relative
                           ↪  to the chassis frame
                          const ChQuaternion<> &rotation          ///< [in] orientation
                           ↪  relative to the chassis frame
  ) override;

  /// Add visualization assets for the steering subsystem.
  /// This default implementation uses primitives.
  virtual void AddVisualizationAssets(VisualizationType vis) override;

  /// Remove visualization assets for the steering subsystem.
  virtual void RemoveVisualizationAssets() override;

  /// Update the state of this steering subsystem at the current time.
  /// The steering subsystem is provided the current steering driver input (a
  /// value between -1 and +1).  Positive steering input indicates steering
  /// to the left. This function is called during the vehicle update.
  virtual void Synchronize(double time,     ///< [in] current time
                           double steering  ///< [in] current steering input [-1,+1]
  ) override;

  /// Get the total mass of the steering subsystem.
  virtual double GetMass() const override;

  /// Get the current global COM location of the steering subsystem.
  virtual ChVector<> GetCOMPos() const override;

  /// Log current constraint violations.
  virtual void LogConstraintViolations() override;

}; // ChHydraulicSteering
} // vehicle
} // chrono

#endif //CHRONO_CHHYDRAULICSTEERING_H
```

LISTING F.13: CHHYDRAULICSTEERING.CPP

```
// MIT License
//
```

```
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#include "ChHydraulicSteering.h"

namespace chrono {
namespace vehicle {

ChHydraulicSteering::ChHydraulicSteering(const std::string &name) : ChSteering(name) {}

void ChHydraulicSteering::Initialize(std::shared_ptr<ChBodyAuxRef> chassis,
                                     const ChVector<> &location,
                                     const ChQuaternion<> &rotation) {
  m_position = ChCoordsys<>(location, rotation);

  // TODO: workout initialize no visualization needed at this time

}

void ChHydraulicSteering::AddVisualizationAssets(VisualizationType vis) {
  ChPart::AddVisualizationAssets(vis);
}

void ChHydraulicSteering::RemoveVisualizationAssets() {
  ChPart::RemoveVisualizationAssets();
}

void ChHydraulicSteering::Synchronize(double time, double steering) {
  // TODO: implement
}

double ChHydraulicSteering::GetMass() const {
  return 0;
}

ChVector<> ChHydraulicSteering::GetCOMPos() const {
  return ChVector<>();
}

void ChHydraulicSteering::LogConstraintViolations() {
  ChSteering::LogConstraintViolations();
}

} // vehicle
```

```
} // chrono
```

LISTING F.14: HYDRAULICSTEERING.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_HYDRAULICSTEERING_H
#define CHRONO_HYDRAULICSTEERING_H

#include "chrono_vehicle/ChApiVehicle.h"
#include "ChHydraulicSteering.h"

#include "chrono_thirdparty/rapidjson/document.h"

namespace chrono {
namespace vehicle {
class CH_VEHICLE_API HydraulicSteering : public ChHydraulicSteering {
 public:
  HydraulicSteering(const std::string &filename);
  HydraulicSteering(const rapidjson::Document &d);
  ~HydraulicSteering() {};

 private:
  virtual void Create(const rapidjson::Document &d) override;
}; // HydraulicSteering
} /// vehicle
} /// chrono



#endif //CHRONO_HYDRAULICSTEERING_H
```

LISTING F.15: HYDRAULICSTEERING.CPP

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
```

```
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#include "HydraulicSteering.h"
#include "chrono_vehicle/utils/ChUtilsJSON.h"

#include "chrono_thirdparty/rapidjson/filereadstream.h"

using namespace rapidjson;

namespace chrono {
namespace vehicle {

HydraulicSteering::HydraulicSteering(const std::string &filename) : ChHydraulicSteering("") {
  FILE *fp = fopen(filename.c_str(), "r");

  char readBuffer[65536];
  FileReadStream is(fp, readBuffer, sizeof(readBuffer));
  fclose(fp);

  Document d;
  d.ParseStream<ParseFlag::kParseCommentsFlag>(is);

  Create(d);

  GetLog() << "Loaded JSON: " << filename.c_str() << "\n";
}

HydraulicSteering::HydraulicSteering(const rapidjson::Document &d) : ChHydraulicSteering("")
 ↪  {
  Create(d);
}

void HydraulicSteering::Create(const rapidjson::Document &d) {
  ChPart::Create(d);

  // TODO: implement specific initialization
}

} /// vehicle
} /// chrono
```

LISTING F.16: CRAWLERDRIVELINE.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
```

```
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_CRAWLER_DRIVELINE_H
#define CHRONO_CRAWLER_DRIVELINE_H

class Crawler_Driveline {

};

#endif //CHRONO_CRAWLER_DRIVELINE_H
```

LISTING F.17: CRAWLERDRIVELINE.CPP

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#include "Crawler_Driveline.h"
```

LISTING F.18: CRAWLERSIMPLEFLUIDPOWERTRAIN.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
```

```
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_CRAWLER_SIMPLEFLUIDPOWERTRAIN_H
#define CHRONO_CRAWLER_SIMPLEFLUIDPOWERTRAIN_H

class Crawler_SimpleFluidPowertrain {

};

#endif //CHRONO_CRAWLER_SIMPLEFLUIDPOWERTRAIN_H
```

LISTING F.19: CRAWLERSIMPLEFLUIDPOWERTRAIN.CPP

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#include "Crawler_SimpleFluidPowertrain.h"
```

LISTING F.20: CMAKELISTS.TXT

```
cmake_minimum_required(VERSION 3.5)
project(CrawlerSim
        VERSION 0.1.0
        LANGUAGES CXX)

SET(SKETCH crawler_sim)
```

```
SET(MODEL_FILES
        Crawler.h
        Crawler.cpp
        Crawler_vehicle.h
        Crawler_vehicle.cpp
        ChCrawlerVehicle.h
        ChCrawlerVehicle.cpp
        HydraulicSteering.cpp
        HydraulicSteering.h
        ChHydraulicSteering.cpp
        ChHydraulicSteering.h
        Crawler_SimpleFluidPowertrain.cpp
        Crawler_SimpleFluidPowertrain.h
        Crawler_ArchmidesTire.cpp
        Crawler_ArchmidesTire.h
        Crawler_Chassis.cpp
        Crawler_Chassis.h
        Crawler_Driveline.cpp
        Crawler_Driveline.h)

# TODO use find_package(Boost)
include_directories(Boost_INCLUDE_DIRS)
find_package(Chrono
             COMPONENTS Vehicle Postprocessing Irrlicht
             CONFIG)

MESSAGE(STATUS "... add ${SKETCH}")
INCLUDE_DIRECTORIES(${CMAKE_CURRENT_LIST_DIR})

ADD_EXECUTABLE(${SKETCH} ${SKETCH}.cpp ${MODEL_FILES})
SOURCE_GROUP("" FILES ${SKETCH}.cpp)

SET_TARGET_PROPERTIES(${SKETCH} PROPERTIES
        COMPILE_FLAGS "${CH_CXX_FLAGS}"
        LINK_FLAGS "${CH_LINKERFLAG_EXE}")

#TARGET_LINK_LIBRARIES(${SKETCH}
#        ChronoEngine
#        ChronoEngine_vehicle
#        ChronoModels_vehicle
#        ChronoEngine_postprocess)
#ADD_DEPENDENCIES(${SKETCH}
#        ChronoEngine
#        ChronoEngine_postprocess)
#INSTALL(TARGETS ${SKETCH} DESTINATION ${CH_INSTALL_DEMO})
```

G APPENDIX CHRONO SENSORS SOURCES

LISTING G.1: CHSENSOR.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_SENSOR_CHSENSOR_H
#define CHRONO_SENSOR_CHSENSOR_H

#include "chrono_vehicle/ChVehicle.h"
#include "chrono_sensor/ChFunction_Sensor.h"

namespace chrono {
namespace vehicle {
namespace sensor {

/// Base class for a vehicle sensor system.
template<class T>
class CH_VEHICLE_API ChSensor {
 public:
  ChSensor()
      : m_sample_rate(0.),
        m_delay(0.),
        m_log_filename(""),
        m_prev_sample_time(0.),
        m_prev_delay_time{0.},
        m_sample(true),
        m_write(true) {}

  ChSensor(ChVehicle &vehicle, double sample_rate = 0., double delay = 0.)
      : m_vehicle(vehicle),
        m_sample_rate(sample_rate),
        m_delay(delay),
        m_log_filename(""),
        m_prev_sample_time(0.),
```

```
      m_prev_delay_time{delay},
      m_sample(true),
      m_write(true) {};

virtual ~ChSensor() = default;

/// Initialize this Sensor System
virtual void Initialize() {};

ChVehicle &Get_Vehicle() const { return m_vehicle; }

void Set_Vehicle(ChVehicle &Vehicle) { m_vehicle = &Vehicle; }

double Get_SampleRate() const { return m_sample_rate; }

void Set_SampleRate(double SampleRate) { m_sample_rate = SampleRate; }

void Set_Input(T input) { m_input = input; };

T &Get_Input() { return m_input; };

void Set_Output(T output) { m_output = output; };

T &Get_Output() { return m_output; };

/// Update the state of this driver system at the current time.
virtual void Synchronize(double time) {
  update_time(time, m_prev_sample_time, m_sample_rate, m_sample);
  auto dt = time - m_prev_delay_time[0];
  if (dt >= m_sample_rate) {
    m_write = true;
    m_prev_delay_time.push_back(time);
  } else {
    m_write = false;
  }
};

/// Advance the state of this driver system by the specified time step
virtual void Advance(double step) {
  if (m_sample) {
    auto aquired = m_input;
    for (auto transform : m_transform) {
      aquired = transform->Get_y(aquired);
    }
    m_aquired.push_back(aquired);
  }
  if (m_write) {
    m_output = m_aquired[0];
    m_aquired.erase(m_aquired.begin());
    m_prev_delay_time.erase(m_prev_delay_time.begin());
  }
}

/// Initialize output file for recording sensor inputs.
bool LogInit(const std::string &filename) {
  m_log_filename = filename;

  std::ofstream ofile(filename.c_str(), std::ios::out);
  if (!ofile)
    return false;

  ofile « "Time, Input, Output" « std::endl;
```

```
    ofile.close();
    return true;
  };

  /// Record the current sensor inputs to the log file.
  bool Log(double time) {
    if (m_log_filename.empty())
      return false;

    std::ofstream ofile(m_log_filename.c_str(), std::ios::app);
    if (!ofile)
      return false;

    ofile « time « ", " « m_input « ", " « m_output « std::endl;
    ofile.close();
    return true;
  }

 protected:
  ChVehicle &m_vehicle;
  double m_sample_rate;
  T m_input;
  std::vector<T> m_aquired;
  T m_output;
  std::vector<std::shared_ptr<ChFunction_Sensor<T»> m_transform;
  double m_prev_sample_time;
  std::vector<double> m_prev_delay_time;
  double m_delay;
  bool m_sample;
  bool m_write;

 private:
  std::string m_log_filename;
  void update_time(const double &time, double &prev_time, const double &condition, bool
    ↪  &set_condition) {
    double dt = time - prev_time;
    if (dt >= condition) {
      set_condition = true;
      prev_time = time;
    } else {
      set_condition = false;
    }
  }
};
} /// sensor
} /// vehicle
} /// chrono
#endif //CHRONO_SENSOR_CHSENSOR_H
```

LISTING G.2: CHFUNCTIONSENSOR.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
```

```
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_SENSOR_CHFUNCTION_SENSOR_H
#define CHRONO_SENSOR_CHFUNCTION_SENSOR_H

#include <typeinfo>

#include "chrono/core/ChApiCE.h"
#include "chrono/core/ChClassFactory.h"

namespace chrono {
namespace vehicle {
namespace sensor {

enum FunctionType {
  FUNCT_CUSTOM,
  FUNCT_NOISE,
  FUNCT_BIAS,
  FUNCT_DIGITIZE
};

template<typename T = double>
class ChApi ChFunction_Sensor {
 public:
  ChFunction_Sensor() : M_BSL(1., BDF_STEP_LOW) { M_BSL.Normalize(); };
  ChFunction_Sensor(const ChFunction_Sensor &other) : M_BSL(other.M_BSL) {};
  virtual ~ChFunction_Sensor() = default;

  /// "Virtual" copy constructor.
  virtual ChFunction_Sensor *Clone() const = 0;

  /// Return the unique function type identifier.
  virtual FunctionType Get_Type() const { return FUNCT_CUSTOM; }

  // THE MOST IMPORTANT MEMBER FUNCTIONS
  // At least Get_y() should be overridden by derived classes.

  /// Return the y value of the function, at position x.
  virtual T Get_y(const T &x) const = 0;

  /// Return the dy/dx derivative of the function, at position x.
  /// Note that inherited classes may also avoid overriding this method,
  /// because this base method already provide a general-purpose numerical differentiation
  /// to get dy/dx only from the Get_y() function. (however, if the analytical derivative
  /// is known, it may better to implement a custom method).
  virtual T Get_y_dx(const T &x) const {
    if constexpr(std::is_same<T, ChQuaternion<>>::value) {
      ChQuaternion<> dy = Get_y(x * M_BSL) - Get_y(x);
      dy /= BDF_STEP_LOW;
      dy.Normalize();
      return dy;
    } else {
```

```
      return ((Get_y(x + BDF_STEP_LOW) - Get_y(x)) / BDF_STEP_LOW);
    }
  }

  /// Return the ddy/dxdx double derivative of the function, at position x.
  /// Note that inherited classes may also avoid overriding this method,
  /// because this base method already provide a general-purpose numerical differentiation
  /// to get ddy/dxdx only from the Get_y() function. (however, if the analytical derivative
  /// is known, it may be better to implement a custom method).
  virtual T Get_y_dxdx(const T &x) const {
    if constexpr(std::is_same<T, ChQuaternion<»::value) {
      ChQuaternion<> dy = Get_y_dx(x * M_BSL) - Get_y_dx(x);
      dy /= BDF_STEP_LOW;
      dy.Normalize();
      return dy;
    } else {
      return ((Get_y_dx(x + BDF_STEP_LOW) - Get_y_dx(x)) / BDF_STEP_LOW);
    }
  };

  /// Return the weight of the function (useful for
  /// applications where you need to mix different weighted ChFunctions)
  virtual double Get_weight(T x) const { return 1.0; };

  /// Return the function derivative of specified order at the given point.
  /// Note that only order = 0, 1, or 2 is supported.
  virtual T Get_y_dN(T x, int derivate) const {
    switch (derivate) {
      case 0:return Get_y(x);
      case 1:return Get_y_dx(x);
      case 2:return Get_y_dxdx(x);
      default:return Get_y(x);
    }
  }

  /// Update could be implemented by children classes, ex. to launch callbacks
  virtual void Update(const double x) {}

  /// Method to allow serialization of transient data to archives
  virtual void ArchiveOUT(ChArchiveOut &marchive) {
    // version number
    marchive.VersionWrite<ChFunction_Sensor<T»();
  }

  /// Method to allow de-serialization of transient data from archives.
  virtual void ArchiveIN(ChArchiveIn &marchive) {
    // version number
    int version = marchive.VersionRead<ChFunction_Sensor<T»();
  }

 private:
  ChQuaternion<> M_BSL;
};

} /// sensor
} /// vehicle
} /// chrono
#endif //CHRONO_SENSOR_CHFUNCTION_SENSOR_H
```

LISTING G.3: CHFUNCTIONSENSORBIAS.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_SENSOR_CHFUNCTION_SENSORBIAS_H
#define CHRONO_SENSOR_CHFUNCTION_SENSORBIAS_H

#include "chrono/core/ChVectorDynamic.h"
#include "ChFunction_Sensor.h"

namespace chrono {
namespace vehicle {
namespace sensor {

template<typename T = double>
class ChApi ChFunction_SensorBias : public ChFunction_Sensor<T> {
 public:
  ChFunction_SensorBias<T>() = default;
  explicit ChFunction_SensorBias<T>(const T &bias) : m_bias(bias) {};
  ChFunction_SensorBias<T>(const ChFunction_SensorBias<T> &other) : m_bias(other.m_bias) {}

  ChFunction_SensorBias<T> *Clone() const override {
    return new ChFunction_SensorBias<T>(*this);
  }

  FunctionType Get_Type() const override {
    return FUNCT_BIAS;
  }

  bool operator==(const ChFunction_SensorBias &rhs) const {
    return m_bias == rhs.m_bias;
  }

  bool operator!=(const ChFunction_SensorBias &rhs) const {
    return !(rhs == *this);
  }

  T Get_y(const T &x) const override {
    if constexpr(std::is_same<T, ChQuaternion<»::value) {
      ChQuaternion<> y = x * m_bias;
      y.Normalize();
```

```
      return y;
    } else {
      return x + m_bias;
    }
  }

  T Get_Bias() const {
    return m_bias;
  }

  void Set_Bias(const T &Bias) {
    m_bias = Bias;
  }

 protected:
  T m_bias;
};

} /// sensor
} /// vehicle
} /// chrono

#endif //CHRONO_SENSOR_CHFUNCTION_SENSORBIAS_H
```

LISTING G.4: CHFUNCTIONSENSORDIGITIZE.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_SENSOR_CHFUNCTION_SENSORDIGITIZE_H
#define CHRONO_SENSOR_CHFUNCTION_SENSORDIGITIZE_H

#include <array>

#include "ChFunction_Sensor.h"

namespace chrono {
namespace vehicle {
namespace sensor {
template<typename T = double>
using opt_vect_t = typename std::conditional<std::is_same<T, ChQuaternion<>>::value,
                                             ChVector<>, T>::type;
```

```
template<typename T = double>
class ChApi ChFunction_SensorDigitize : public ChFunction_Sensor<T> {
 public:
  ChFunction_SensorDigitize<T>() {
    static_assert(
        std::is_same<T, double>::value || std::is_same<T, ChVector<>>::value ||
        ↪  std::is_same<T, ChQuaternion<>>::value,

        ↪  "ChFunction_SensorDigitize requires a double, chrono::ChVector<double> of ChQuaternion<double> typ
    m_range = T(0.);
    m_bits = 0.;
    m_res = T(0.);
  };

  ChFunction_SensorDigitize<T>(const double &bits, const opt_vect_t<T> &range) {
    static_assert(
        std::is_same<T, double>::value || std::is_same<T, ChVector<>>::value ||
        ↪  std::is_same<T, ChQuaternion<>>::value,

        ↪  "ChFunction_SensorDigitize requires a double, chrono::ChVector<double> of ChQuaternion<double> typ
    m_range = range;
    m_bits = bits;
    m_res = Calc_Resolution(m_range, m_bits);
  }

  ChFunction_SensorDigitize<T>(const ChFunction_SensorDigitize<T> &other)
      : m_range(other.m_range), m_res(other.m_res), m_bits(other.m_bits) {}

  ChFunction_SensorDigitize<T> *Clone() const override {
    return new ChFunction_SensorDigitize<T>(*this);
  }

  FunctionType Get_Type() const override {
    return FUNCT_DIGITIZE;
  }

  T Get_y(const T &x) const override {
    if constexpr(std::is_same<T, ChQuaternion<>>::value) {
      auto x_p = ChVector<>(x.e1(), x.e2(), x.e3());
      auto x_d_vec = ChVector<>(m_res * Round(x_p / m_res));
      return ChQuaternion<>(x.e0(), x_d_vec).GetNormalized();
    } else {
      return m_res * Round(x / m_res);
    }
  }

  opt_vect_t<T> &Get_Range() const {
    return m_range;
  }

  void Set_Range(const opt_vect_t<T> &Range) {
    m_range = Range;
    m_res = Calc_Resolution(m_range, m_bits);
  }

  double Get_Bits() const {
    return m_bits;
  }

  void Set_Bits(const double Bits) {
    m_bits = Bits;
```

```
    m_res = Calc_Resolution(m_range, m_bits);
  }

 protected:
  constexpr opt_vect_t<T> Calc_Resolution(const opt_vect_t<T> &range, const double bits) {
    return range / pow(2., bits);
  }

  opt_vect_t<T> Round(const opt_vect_t<T> &x) const {
    if constexpr(std::is_same<T, double>::value) {
      return round(x);
    } else {
      opt_vect_t<T> ret;
      for (int i = 0; i < 3; ++i) {
        ret[i] = round(x[i]);
      }
      return ret;
    }
  };

  opt_vect_t<T> m_range;
  opt_vect_t<T> m_res;
  double m_bits;
};

} /// sensor
} /// vehicle
} /// chrono
#endif //CHRONO_SENSOR_CHFUNCTION_SENSORDIGITIZE_H
```

LISTING G.5: CHFUNCTIONSENSORNOISE.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_SENSOR_CHFUNCTION_SENSORNOISE_H
#define CHRONO_SENSOR_CHFUNCTION_SENSORNOISE_H

#include <random>
#include <chrono>

#include "ChFunction_Sensor.h"
```

```
#include "chrono/core/ChVectorDynamic.h"
#include "chrono/core/ChVector.h"
#include <chrono/core/ChQuaternion.h>

namespace chrono {
namespace vehicle {
namespace sensor {

template<typename T = double>
class ChApi ChFunction_SensorNoise : public ChFunction_Sensor<T> {
 public:
  ChFunction_SensorNoise() : m_mean(0.), m_stddev(0.) {
    static_assert(
        std::is_same<T, double>::value || std::is_same<T, ChVector<»::value ||
        ↪  std::is_same<T, ChQuaternion<»::value,

        ↪  "ChFunction_SensorNoise requires a double, chrono::ChVector<double> of ChQuaternion<double> type");
    m_gen = std::make_shared<std::default_random_engine>(Get_Seed());
  };

  ChFunction_SensorNoise(const T &Mean,
                         const T &Stddev)
      : m_mean(1), m_stddev(1) {
    m_mean = Mean;
    m_stddev = Stddev;
    m_gen = std::make_shared<std::default_random_engine>(Get_Seed());
  };

  ChFunction_SensorNoise(const ChFunction_Sensor<T> &other)
      : m_mean(other.m_mean), m_stddev(other.m_stddev), m_gen(other.m_gen) {};

  ChFunction_SensorNoise<T> *Clone() const override {
    return new ChFunction_SensorNoise<T>(*this);
  };

  bool operator==(const ChFunction_SensorNoise &rhs) const {
    return m_gen == rhs.m_gen &&
        static_cast<T>(m_mean) == static_cast<T>(rhs.m_mean) &&
        static_cast<T>(m_stddev) == static_cast<T>(rhs.m_stddev);
  }

  bool operator!=(const ChFunction_SensorNoise &rhs) const {
    return !(rhs == *this);
  }

  FunctionType Get_Type() const override {
    return FUNCT_NOISE;
  }

  T Get_y(const T &x) const override {
    if constexpr(std::is_same<T, ChQuaternion<»::value) {
      return x * Get_Noise(m_mean, m_stddev);
    } else {
      return x + Get_Noise(m_mean, m_stddev);
    }
  };

  T &Get_Mean() const {
    return m_mean;
  }
```

```
  void Set_Mean(const T &Mean) {
    m_mean = Mean;
  }

  T &Get_Stddev() const {
    return m_stddev;
  }

  void Set_Stddev(const T &Stddev) {
    m_stddev = Stddev;
  }

 protected:
  static unsigned int Get_Seed() {
    typedef std::chrono::high_resolution_clock seed_clock;
    seed_clock::time_point beginning = seed_clock::now();
    seed_clock::duration d = seed_clock::now() - beginning;
    unsigned seed = d.count();
    return seed;
  };

  T Get_Noise(const T &mean, const T &stddev) const {
    if constexpr(std::is_same<T, double>::value) {
      return Get_Noise_Scalar(mean, stddev);
    } else {
      size_t last_elem;
      if constexpr(std::is_same<T, ChVector<>>::value) {
        last_elem = 3;
      } else {
        last_elem = 4;
      }
      T ret;
      for (int i = 0; i < last_elem; ++i) {
        ret[i] = Get_Noise_Scalar(mean[i], stddev[i]);
      }
      return ret;
    }
  };

  double Get_Noise_Scalar(const double &mean, const double &stddev) const {
    std::normal_distribution<double> dist(mean, stddev);
    return dist(*m_gen);
  }

  T m_mean;
  T m_stddev;
  std::shared_ptr<std::default_random_engine> m_gen;
};
} /// sensor
} /// vehicle
} /// chrono
#endif //CHRONO_SENSOR_CHFUNCTION_SENSORNOISE_H
```

LISTING G.6: ACCELEROMETER.H

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
```

```
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#ifndef CHRONO_SENSOR_ACCELEROMETER_H
#define CHRONO_SENSOR_ACCELEROMETER_H

#include "ChSensor.h"
#include "chrono_sensor/ChFunction_SensorNoise.h"
#include "chrono_sensor/ChFunction_SensorDigitize.h"

namespace chrono {
namespace vehicle {
namespace sensor {

class CH_VEHICLE_API Accelerometer : public ChSensor<ChVector<>> {
 public:
  Accelerometer(ChVehicle &vehicle, const double sample_rate, const double delay);
  void Initialize(const double &bits,
                  const ChVector<> &range,
                  const ChVector<> &mean,
                  const ChVector<> &stddev);

  std::shared_ptr<ChFunction_SensorDigitize<ChVector<>>> Get_DigitalTransform();
  std::shared_ptr<ChFunction_SensorNoise<ChVector<>>> Get_NoiseTransform();

};
} /// sensor
} /// vehicle
} /// chrono
#endif //CHRONO_SENSOR_ACCELEROMETER_H
```

LISTING G.7: ACCELEROMETER.CPP

```
// MIT License
//
// Copyright (c) 2019 Jelle Spijker
//
// Permission is hereby granted, free of charge, to any person obtaining a copy
// of this software and associated documentation files (the "Software"), to deal
// in the Software without restriction, including without limitation the rights
// to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
// copies of the Software, and to permit persons to whom the Software is
// furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in all
// copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
// IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
```

```
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
// LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
// OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
// SOFTWARE.
//

#include "chrono_sensor/Accelerometer.h"

namespace chrono {
namespace vehicle {
namespace sensor {

Accelerometer::Accelerometer(ChVehicle &vehicle, const double sample_rate, const double
 ↪  delay) : ChSensor<ChVector<»(
    vehicle,
    sample_rate,
    delay) {
  auto noise = std::make_shared<ChFunction_SensorNoise<ChVector<»>();
  auto digitize = std::make_shared<ChFunction_SensorDigitize<ChVector<»>();
  m_transform.push_back(noise);
  m_transform.push_back(digitize);
}

void Accelerometer::Initialize(const double &bits,
                               const ChVector<> &range,
                               const ChVector<> &mean,
                               const ChVector<> &stddev) {
  Get_DigitalTransform()->Set_Bits(bits);
  Get_DigitalTransform()->Set_Range(range);
  Get_NoiseTransform()->Set_Mean(mean);
  Get_NoiseTransform()->Set_Stddev(stddev);
  ChSensor::Initialize();
}

std::shared_ptr<ChFunction_SensorDigitize<ChVector<»» Accelerometer::Get_DigitalTransform() {
  return std::dynamic_pointer_cast<ChFunction_SensorDigitize<ChVector<»»(m_transform[1]);
}

std::shared_ptr<ChFunction_SensorNoise<ChVector<»» Accelerometer::Get_NoiseTransform() {
  return std::dynamic_pointer_cast<ChFunction_SensorNoise<ChVector<»»(m_transform[0]);
}
} /// sensor
} /// vehicle
} /// chrono
```

LISTING G.8: CMAKELISTS.TXT

```
project(chrono_sensor VERSION 0.1 LANGUAGES CXX)

set(SRC_FILES
        src/Accelerometer.cpp
        src/Gyroscope.cpp)

set(HDR_FILES
        include/chrono_sensor/ChSensor.h
        include/chrono_sensor/ChFunction_Sensor.h
        include/chrono_sensor/ChFunction_SensorNoise.h
        include/chrono_sensor/ChFunction_SensorBias.h
        include/chrono_sensor/ChFunction_SensorDigitize.h
        include/chrono_sensor/Gyroscope.h
        )
```

```
add_library(chrono_sensor SHARED ${SRC_FILES} ${HDR_FILES})


target_include_directories(chrono_sensor PUBLIC
        $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
        $<INSTALL_INTERFACE:include>
        PRIVATE src)
target_compile_options(chrono_sensor PUBLIC -pthread -fopenmp -march=native -msse4.2
 ↪  -mfpmath=sse -march=native -mavx)
target_compile_definitions(chrono_sensor PUBLIC "CHRONO_DATA_DIR=\"${CHRONO_DATA_DIR}\"")
target_link_libraries(chrono_sensor PUBLIC ${CHRONO_LIBRARIES})

# 'make install' to the correct locations (provided by GNUInstallDirs).
install(TARGETS chrono_sensor EXPORT chrono_sensorConfig
        ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
        LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
        RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR})  # This is for Windows
install(DIRECTORY include/ DESTINATION ${CMAKE_INSTALL_INCLUDEDIR})

# This makes the project importable from the install directory
# Put config file in per-project dir (name MUST match), can also
# just go into 'cmake'.
install(EXPORT chrono_sensorConfig DESTINATION share/chrono_sensor/cmake)

# This makes the project importable from the build directory
export(TARGETS chrono_sensor FILE chrono_sensorConfig.cmake)
```

H APPENDIX OHCAPTAIN

LISTING H.1: PT100.H

```
//
// Created by peer23peer on 7/18/16.
//

#pragma once

#include "../Core/Sensor.h"

#include <utility>

#include <boost/units/systems/temperature/celsius.hpp>

namespace oCpt {
    namespace components {
        namespace sensors {
            /*!
             * A standard PT100 sensors
             */
            class PT100 : public Sensor {
            public:
                typedef quantity<celsius::temperature, double> ReturnValue_t; //<!
                 ↪ Temperature is in Celsius TODO implement conversion to Kelvin

                PT100(iController::ptr controller, World::ptr world, std::string id, uint8_t
                 ↪ pinid, uint8_t device);

                ~PT100();

                void updateSensor();

                void run();

                void stop();

                void init();

                void setCalibrationTemperature(std::pair<ReturnValue_t, ReturnValue_t>
                 ↪ temparature,
                                               std::pair<uint16_t, uint16_t> analogeValue);

            private:
                uint16_t _analogeValue;
                uint8_t _device = 0;
                uint8_t _pinid = 0;
                ReturnValue_t _dy_dx = 1.0 * celsius::degree;
                ReturnValue_t _constant = 0.0 * celsius::degree;
            };
        }
    }
}
```

LISTING H.2: GPS.H

```
//
// Created by peer23peer on 7/25/16.
//

#pragma once

#include "../Core/Sensor.h"
#include "../Core/Controller.h"

namespace oCpt {
    namespace components {
        namespace sensors {
            class Gps : public Sensor {
            public:
                typedef oCpt::World::Location::gpsPoint_t ReturnValue_t;

                Gps(iController::ptr controller, World::ptr world, std::string id,
                 ↪  std::string device,
                             unsigned int baudrate);

                ~Gps();

                void updateSensor();

                void run();

                void stop();

                void setIOservice(boost::shared_ptr<boost::asio::io_service> ioservice);

            protected:
                std::string device_;
                protocol::Serial::ptr serial_;

                void interpretMsg();
            };
        }
    }
}
```

LISTING H.3: RAZOR.H

```
//
// Created by peer23peer on 10/14/16.
//
#pragma once

#include "../Core/Sensor.h"

#include <boost/units/systems/si/angular_velocity.hpp>
#include <boost/units/systems/si/magnetic_flux_density.hpp>

namespace oCpt {
    namespace components {
        namespace sensors {
            class Razor : public Sensor {
            public:
                typedef struct ReturnValue {
                    quantity<si::angular_velocity, float> gyro[3];
                    quantity<si::magnetic_flux_density, float> mag[3];
```

```
                    quantity<si::acceleration, float> acc[3];
                } ReturnValue_t;

                enum Mode {
                    CONT = 0,
                    REQ = 1
                };

                Razor(iController::ptr controller, World::ptr world, std::string id,
                 ↪  std::string device, unsigned int baudrate, Mode mode = Mode::REQ, uint8_t
                 ↪  freq = 50);

                ~Razor();

                void updateSensor();

                void run();

                void stop();

                void init();

                void setIOservice(boost::shared_ptr<boost::asio::io_service> ioservice);

            private:
                std::string device_;
                protocol::Serial::ptr serial_;
                protocol::Serial::cb_func cb;
                Mode mode_;
            public:
                Mode getMode() const;

                void setMode(Mode mode);

                uint8_t getFreq() const;

                void setFreq(uint8_t freq);

            private:
                uint8_t freq_ = 50;

                void fillReturnValue(ReturnValue_t &retVal, float *values);

                void msgHandler(const unsigned char *data, size_t size);

                bool checkLRC(std::vector<char *> data);
            };
        }
    }
}
```

LISTING H.4: BEAGLEBONEBLACK.H

```
//
// Created by peer23peer on 7/23/16.
//

#pragma once

#include "../Core/Controller.h"

#include <utility>
```

```
namespace oCpt {
    namespace components {
        namespace controller {
            class BBB : public ARM {
            public:
                //typedef boost::shared_ptr<iController> ptr;
                BBB(World::ptr world);

                virtual ~BBB();
            };

        }
    }
}
```

LISTING H.5: MEETCATAMARAN.H

```
//
// Created by peer23peer on 7/23/16.
//

#pragma once

#include "../Core/Vessel.h"

namespace oCpt {
    namespace vessels {
        class Meetcatamaran : public Vessel {
        public:
            Meetcatamaran();

            virtual ~Meetcatamaran();
        private:

        };
    }
}
```

LISTING H.6: LORA$_{RN2483}$.H

```
//
// Created by peer23peer on 8/12/16.
//
#pragma once

#include "../Core/Communication.h"

//Yes point to point is possible . Ive tried this myself. use this initialisation on both TX
 ↪  and RX
//"sys reset"
//"radio set mod lora"
//"radio set freq 868000000"
//"radio set pwr 14"
//"radio set sf sf12"
//"radio set afcbw 125"
//"radio set rxbw 250"
//"radio set fdev 5000"
//"radio set prlen 8"
//"radio set crc on"
//"radio set cr 4/8"
//"radio set wdt 0"
```

```
//"radio set sync 12"
//"radio set bw 250"
//
//on the receiver side use these commands:
//"mac pause"
//radio rx 0"
//
//on the transmitter side use these commands to send a packet eg FFh
//"mac pause"
//"radio tx FF"

namespace oCpt {
    namespace components {
        namespace comm {
            class LoRa_RN2483 : public LoRa {
            public:

                LoRa_RN2483(const std::string &id, const std::string &device,
                            World::ptr world = World::ptr(new World()),
                            iController::io_t ioservice = iController::io_t(new
                             ↪  boost::asio::io_service()));

                virtual ~LoRa_RN2483();
            };
        }
    }
}
```

LISTING H.7: COMMUNICATION.H

```
//
// Created by peer23peer on 7/25/16.
//

#pragma once

#include <boost/shared_ptr.hpp>
#include <boost/signals2.hpp>
#include <boost/asio/steady_timer.hpp>
#include <boost/asio.hpp>

#include <string>
#include <vector>
#include <deque>
#include <typeinfo>

#include "Controller.h"
#include "World.h"
#include "Exception.h"

namespace oCpt {

    /*!
     * The interface for communication devices
     */
    class iComm {
    public:
        typedef boost::shared_ptr<iComm> ptr; //<! A typedef to the shared_ptr
        typedef boost::signals2::signal<void()> signal_t; //<! a typedef to the signal

        /*!
```

```
         * A Message struct. Each device should receive and send messages with this type,
↪  consisting of Payload in the format of a string and a time when it was send or received
         */
        struct Message {
            typedef boost::shared_ptr<Message> ptr; //<! A shared_ptr to a Message

            /*!
             * A Message constructor taking the payload and the time
             * @param payload A string containing the payload TODO make generic with template
             * @param stamp A time stamp, when the message was received, or is send
             */
            Message(const std::string &payload, const World::Time::timepoint_t &stamp)
                    : Payload(payload),
                      Stamp(stamp) {};

            /*!
             * The deconstructor
             */
            ~Message() {};
            std::string Payload; //<! The payload of the message TODO make generic
            World::Time::timepoint_t Stamp; //<! The time stamp
        };

        /*!
         * The constructor for the communication interface
         * @param id The ID of the communication device
         * @param device The device path eq. /dev/ttyS0
         * @param world  A shared_ptr to the world with a default to a newly created one
         * @param ioservice  A shared_ptr to an ASIO Input Output service with a newly
↪  created one as default
         */
        iComm(const std::string &id, const std::string &device, World::ptr world =
         ↪  World::ptr(new World()),
              iController::io_t ioservice = iController::io_t(new
               ↪  boost::asio::io_service()));

        /*!
         * The deconstructor
         */
        virtual ~iComm();

        /*!
         * a pure virtual function which runs the communication device
         */
        virtual void run() = 0;

        /*!
         * A pure virtual function which stops the communication device
         */
        virtual void stop() = 0;

        /*!
         * A pure virtual function which initializes the communication device
         */
        virtual void initialize() = 0;

        /*!
         * A pure virtual function which sends the message
         * @param msg the Message, consisting of a payload and a time stamp
         */
        virtual void sendMessage(Message msg) = 0;
```

```
        /*!
         * A pure virtual function with a shared_ptr to the first in queue received message,
↪  this function will hold the current thread
         * @return a shared_ptr pointing towards the queued Message
         */
        virtual Message::ptr recieveMessage() = 0;

        /*!
         * A pure virtual function which performs the polling for a new message on a seperate
↪  threads, so it won't block the current one, it needs to send a signal when the message is
↪  received
         */
        virtual void recieveAsyncMessage() = 0;

        /*!
         * Returns the ID of the communication device
         * @return a string with the ID
         */
        const std::string &getId() const;

        /*!
         * Set the ID of the communication device
         * @param id The ID of the communication device
         */
        void setId(const std::string &id);

        /*!
         * Get the type of communication device
         * @return a string with type of device eq. modem, serial, LoRa, WiFi
         */
        const std::string &getTypeOfComm() const;

        /*!
         * Set the type of communication device. eq. modem, serial, LoRa, WiFi
         * @param typeOfComm string representing the type of communication
         */
        void setTypeOfComm(const std::string &typeOfComm);

        /*!
         * Get a pointer to the first message in Queu
         * @return
         */
        Message::ptr readFiFoMsg();

        /*!
         * A que with received Message::ptr
         * @return a pointer to the Message que
         */
        std::deque<Message::ptr> *getMsgQueue();

        /*!
         * The ASIO Input Output service handling the messages
         * @param ioservice a shared_ptr to a IO service
         */
        void setIoservice(const iController::io_t &ioservice);

        /*!
         * A signal which is send when a new Message
         */
        signal_t msgRecievedSig;
    protected:
        std::string id_; //<! The communication device
```

```
        std::string typeOfComm_; //<! The type of communication device
        std::string device_; //<! The path to the device
        boost::posix_time::milliseconds timer_; //<! A timer value which can be set to
         ↪ send/receive at intervals TODO implment set/get
        std::deque<Message::ptr> msgQueue_; //<! A deque with the first recieved messages in
         ↪ the front and the last in the back
        iController::io_t ioservice_; //<! a shared_ptr to the ASIO IO service
        World::ptr world_; //<! a shere_ptr to the World
    };

    /*!
     * Communication class for the LoRa protocol. The current class is mostly based on node 2
    ↪ node communication TODO rewrite sho it will allow mesh network communication. Most of the
    ↪ commands are taken from http://ww1.microchip.com/downloads/en/DeviceDoc/40001784B.pdf
     */
    class LoRa : public iComm {
    public:

        /*!
         * The modulation mode of the LoRa module
         */
        enum ModulationMode {
            LORA = 1, //<! Long Rang Low Power mode
            FSK = 2 //<! Frequency-shift keyring mode
        };

        /*!
         * The Spreading factor
         */
        enum SpreadingFactor {
            SF7 = 7,
            SF8 = 8,
            SF9 = 9,
            SF10 = 10,
            SF11 = 11,
            SF12 = 12
        };

        /*!
         * The bandwidth
         */
        enum BandWidth {
            BW250 = 1, //<! 250 kHz
            BW200 = 2, //<! 200 kHz
            BW166_7 = 3,//<! 166.7 kHz
            BW125 = 4, //<! 125 kHz
            BW100 = 5, //<! 100 kHz
            BW83_3 = 6, //<! 83.3 kHz
            BW62_5 = 7, //<! 62.5 kHz
            BW50 = 8, //<! 50 kHz
            BW41_7 = 9, //<! 41.7 kHz
            BW31_3 = 10, //<! 31.3 kHz
            BW25 = 11, //<! 25 kHz
            BW20_8 = 12, //<! 20.8 kHz
            BW15_6 = 13, //<! 15.6 kHz
            BW12_5 = 14, //<! 12.5 kHz
            BW10_4 = 15, //<! 10.4 kHz
            BW7_8 = 16, //<! 7.8 kHz
            BW6_3 = 17, //<! 6.3 kHz
            BW5_2 = 18, //<! 5.2 kHz
            BW3_9 = 19, //<! 3.9 kHz
            BW3_1 = 20, //<! 3.1 kHz
```

```
            BW2_6 = 21 //<! 2.6 kHz
        };

        /*!
         * The Coding rate of the signal
         */
        enum CodingRate {
            CR4_5 = 4, //<! CR 4/5
            CR4_6 = 3, //<! CR 4/6
            CR4_7 = 2, //<! CR 4/7
            CR4_8 = 1 //<! CR 4/8
        };

        /*!
         * The radio bandwidth
         */
        enum RadioBandWidth {
            RBW500 = 500, //<! 500 kHz
            RBW250 = 250, //<! 250 kHz
            RBW125 = 125 //<! 125 kHz
        };

        /*!
         * Perform a get or a set command or otherwise none
         */
        enum GetSet {
            GET,
            SET,
            NONE
        };

        /*!
         * Types of radio commands
         */
        enum RadioCommand {
            MOD, //<! modulation mode
            FREQ, //<! current operation requency for the radio
            PWR, //<! output power level used by the radio during transmission
            SF, //<! Spreading Factor to be used during transmission
            AFCBW, //<! automatic frequency correction bandwidth
            RXBW, //<! operational receive bandwidth
            FSKBITRATE, //<! frequency shift keyring bitrate
            FDEV, //<! frequency deviation allowed by the end device
            PRLEN, //<! preamble length used during transmissions
            CRC, //<! CRC header to be used
            CR, //<! Coding rate to be used
            WDT, //<! time-out limit for the radio watchdog timer
            SYNC, //<! sync word to be used
            BW, //<! value used for the bandwidth
            rRX,
            rTX
        };

        /*!
         * Type to control the MAC layer, currently only pause is used, because node 2 node
↪  communication doesn't use MAC
         */
        enum MacCommand {
            PAUSE,
            RESET,
            mTX,
            JOIN,
```

```
            SAVE,
            FORCEENABLE,
            RESUME
        };

        /*!
         * LoRa device constructor
         * @param id the ID of the device as an string
         * @param device the device path eq. /dev/ttyS0
         * @param world  shared_ptr to the World default = a newly created World
         * @param ioservice  shared_ptr to an IO service, default is a newly created IO
↪  service
         */
        LoRa(const std::string &id, const std::string &device, World::ptr world =
         ↪  World::ptr(new World()),
             iController::io_t ioservice = iController::io_t(new boost::asio::io_service()));

        virtual ~LoRa();

        virtual void run() override;

        virtual void stop() override;

        virtual void initialize() override;

        virtual void sendMessage(Message msg) override;

        virtual Message::ptr recieveMessage() override;

        virtual void recieveAsyncMessage() override;

    protected:
        void messageRecieved();

        std::string bandWidthToString(const BandWidth &value);

        std::string codingRateToString(const CodingRate &value);

        void stringToHex(const std::string str, std::string &hexStr, const bool capital =
         ↪  true);

        void hexToString(const std::string hexStr, std::string &str);

        /*!
         * Convert the value of a Type T to a hexidecimal string, which can be send to a LoRa
↪  device, such that it can be transmitted
         * @tparam T the type of value, to be converted
         * @param value the to be converted value
         * @param capital boolean indicating if the hexidecimal string should consist of
↪  capital letters
         * @return a string with the value as hexidecimal values
         */
        template<typename T>
        std::string encodeTypeToHex(T value, bool capital = true){
            std::string retVal;
            retVal.resize(sizeof(T) * 2);
            static const char a = capital ? 0x40 : 0x60;
            char *begin = reinterpret_cast<char *>(&value);
            for (uint8_t i = 0; i < sizeof(T); i++) {
                unsigned char p = *(begin + i);
                char c = (p >> 4) & 0xF;
                retVal[i * 2] = c > 9 ? (c - 9) | a : c | '0';
```

```
            retVal[i * 2 + 1] = (p & 0xF) > 9 ? (p - 9) & 0xF | a : p & 0xF | '0';
        }
        return retVal;
    };

    /*!
     * A command string builder for MAC commands currently onlu PAUSE implemented
     * @tparam T the type of MAC command eq. MacCommand::PAUSE
     * @param cmd the MacCommand to be performed
     * @param value the Value to be send
     * @param prop Additional properties
     * @return a string which can be send to the LoRa module eq. "mac set pause"
     */
    template<typename T>
    std::string buildMacCmdString(MacCommand cmd, T value = 0, GetSet prop = NONE) {
        std::string retVal = "mac ";
        if (prop == SET) {
            retVal.append("set ");
        } else if (prop == GET) {
            retVal.append("get ");
        }
        switch (cmd) {
            case PAUSE:
                retVal.append("pause");
                break;
            default:
                retVal.append("");
                //TODO implement all mac commands
        }
        retVal.append("\r\n");
        return retVal;
    };

    /*!
     * A command string builder for radio commands
     * @tparam T
     * @param cmd
     * @param value
     * @param prop
     * @return
     */
    template<typename T>
    std::string buildRadioCmdString(RadioCommand cmd, T value = 0, GetSet prop = SET) {
        //TODO implement get set command builder now it's just setter
        //TODO make it work with values of the string type
        std::string retVal = "radio ";
        if (prop == SET) {
            retVal.append("set ");
        } else if (prop == GET) {
            retVal.append("get ");
        }
        switch (cmd) {
            case MOD:
                switch (value) {
                    case FSK:
                        retVal.append("mod fsk");
                        break;
                    default:
                        retVal.append("mod lora");
                }
                break;
            case FREQ:
```

```
                retVal.append("freq ").append(std::to_string(value));
                break;
            case PWR:
                retVal.append("pwr ").append(std::to_string(value));
                break;
            case SF:
                retVal.append("sf sf").append(std::to_string(static_cast<int >(value)));
                break;
            case AFCBW:
                retVal.append("afcbw ").append(bandWidthToString(static_cast<BandWidth>(value)));
                break;
            case RXBW:
                retVal.append("rxbw ").append(bandWidthToString(static_cast<BandWidth>(value)));
                break;
            case FSKBITRATE:
                //TODO implement when FSK is supported
                break;
            case FDEV:
                retVal.append("fdev ").append(std::to_string(static_cast<int>(value)));
                break;
            case PRLEN:
                retVal.append("prlen ").append(std::to_string(static_cast<int>(value)));
                break;
            case CRC:
                if (value) {
                    retVal.append("crc on");
                } else {
                    retVal.append("crc off");
                }
                break;
            case CR:
                retVal.append("cr ").append(codingRateToString(static_cast<CodingRate>(value)));
                break;
            case WDT:
                retVal.append("wdt ").append(std::to_string(value));
                break;
            case SYNC:
                retVal.append("sync ").append(std::to_string(value));
                break;
            case BW:
                retVal.append("bw ").append(std::to_string(static_cast<int>(value)));
                break;
            case rRX:
                retVal.append("rx 0");
                break;
        }
        retVal.append("\r\n");
        return retVal;
    }

    std::string buildRadioCmdString(RadioCommand cmd, std::string value, GetSet prop =
     ↪  SET) {
        std::string retVal = "radio ";
        if (prop == SET) {
            retVal.append("set ");
        } else if (prop == GET) {
            retVal.append("get ");
        }
        if (cmd == rTX) {
```

```
                std::string msg = "";
                stringToHex(value, msg);
                retVal.append("tx ").append(msg);
            }
            retVal.append("\r\n");
            return retVal;
        }

        unsigned long calculateDownTime(unsigned int payload);

        void write(const std::string &value);

        void rx();

        void macpause();

        bool proceed_;
        bool ignoreWarn_;
        bool listen_;
        protocol::Serial serial_;//!< SerialPort for UART communication with the chip
        unsigned int baudrate_; //<! BaudRate for UART communication with the chip
        ModulationMode mod_; //!< Modulation mode
        unsigned long freq_; //!< Frequency between 433050000..4347900000 or
         ↪  863000000...870000000
        int8_t pwr_; //!< Power of transmission between -3...15
        SpreadingFactor sf_; //!< Spreading factor of the signal
        BandWidth afcbw_; //!< Automatic frequency correction in kHz
        BandWidth rxbw_; //!< Signal bandwidth in kHz
        uint fskBitRate_; //!< FSK bitrate between 1...300000
        uint fdev_; //!< Frequency deviation between 0...200000
        uint prlen_; //!< Preamble length between 0...65535
        bool crc_; //!< CRC Header on or off
        CodingRate cr_; //!< The coding rate
        unsigned long wdt_; //!< WatchDog 0...4294967295. Set to 0 to disable
        unsigned int sync_; //!< Sync word
        RadioBandWidth bw_; //!< RadioBandWidth in kHz
        bool sendAllowed_;
    };


}
```

LISTING H.8: LITERALS.H

```
// Copyright (c) 2013 Andrew Gascoyne-Cecil
//
// Permission is hereby granted, free of charge, to any person obtaining a
// copy of this software and associated documentation files (the "Software"),
// to deal in the Software without restriction, including without limitation
// the rights to use, copy, modify, merge, publish, distribute, sublicense,
// and/or sell copies of the Software, and to permit persons to whom the Software
// is furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
// OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
// WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
// CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

```
#pragma once

#include <boost/units/systems/si.hpp>

namespace boost {

namespace units {

namespace literals {

using namespace si;

#define BOOST_UNITS_LITERAL(suffix, unit, val, prefix, multiplier) \
quantity<unit, long double> operator "" _##prefix##suffix(long double x) \
{ \
    return quantity<unit, long double>(x * multiplier * val); \
} \
quantity<unit, unsigned long long> operator "" _##prefix##suffix(unsigned long long x) \
{ \
    return quantity<unit, unsigned long long>(x * multiplier * val); \
}

#define BOOST_UNITS_LITERAL_SET(suffix, unit, val) \
BOOST_UNITS_LITERAL(suffix, unit, val, Y, 1000000000000000000000000.0) \
BOOST_UNITS_LITERAL(suffix, unit, val, Z, 1000000000000000000000.0) \
BOOST_UNITS_LITERAL(suffix, unit, val, E, 1000000000000000000.0) \
BOOST_UNITS_LITERAL(suffix, unit, val, P, 1000000000000000.0) \
BOOST_UNITS_LITERAL(suffix, unit, val, T, 1000000000000.0) \
BOOST_UNITS_LITERAL(suffix, unit, val, G, 1000000000.0) \
BOOST_UNITS_LITERAL(suffix, unit, val, M, 1000000.0) \
BOOST_UNITS_LITERAL(suffix, unit, val, k, 1000.0) \
BOOST_UNITS_LITERAL(suffix, unit, val, h, 100.0) \
BOOST_UNITS_LITERAL(suffix, unit, val, da, 10.0) \
BOOST_UNITS_LITERAL(suffix, unit, val, , 1.0) \
BOOST_UNITS_LITERAL(suffix, unit, val, d, 0.1) \
BOOST_UNITS_LITERAL(suffix, unit, val, c, 0.01) \
BOOST_UNITS_LITERAL(suffix, unit, val, m, 0.001) \
BOOST_UNITS_LITERAL(suffix, unit, val, u, 0.000001) \
BOOST_UNITS_LITERAL(suffix, unit, val, n, 0.00000001) \
BOOST_UNITS_LITERAL(suffix, unit, val, p, 0.00000000001) \
BOOST_UNITS_LITERAL(suffix, unit, val, f, 0.00000000000001) \
BOOST_UNITS_LITERAL(suffix, unit, val, a, 0.00000000000000001) \
BOOST_UNITS_LITERAL(suffix, unit, val, z, 0.00000000000000000001) \
BOOST_UNITS_LITERAL(suffix, unit, val, y, 0.00000000000000000000001)

BOOST_UNITS_LITERAL_SET(m, length, metre)
BOOST_UNITS_LITERAL_SET(g, mass, 0.001 * kilogram)
BOOST_UNITS_LITERAL_SET(s, time, second)
BOOST_UNITS_LITERAL_SET(A, current, ampere)
BOOST_UNITS_LITERAL_SET(K, temperature, kelvin)
BOOST_UNITS_LITERAL_SET(mol, amount, mole)
BOOST_UNITS_LITERAL_SET(cd, luminous_intensity, candela)
BOOST_UNITS_LITERAL_SET(Hz, frequency, hertz)
BOOST_UNITS_LITERAL_SET(rad, plane_angle, radian)
BOOST_UNITS_LITERAL_SET(sr, solid_angle, steradian)
BOOST_UNITS_LITERAL_SET(N, force, newton)
BOOST_UNITS_LITERAL_SET(Pa, pressure, pascal)
BOOST_UNITS_LITERAL_SET(J, energy, joule)
BOOST_UNITS_LITERAL_SET(W, power, watt)
BOOST_UNITS_LITERAL_SET(C, electric_charge, coulomb)
BOOST_UNITS_LITERAL_SET(V, electric_potential, volt)
```

```
BOOST_UNITS_LITERAL_SET(F, capacitance, farad)
BOOST_UNITS_LITERAL_SET(ohm, resistance, ohm)
BOOST_UNITS_LITERAL_SET(S, conductance, siemens)
BOOST_UNITS_LITERAL_SET(Wb, magnetic_flux, weber)
BOOST_UNITS_LITERAL_SET(T, magnetic_flux_density, tesla)
BOOST_UNITS_LITERAL_SET(H, inductance, henry)
BOOST_UNITS_LITERAL_SET(degC, temperature, kelvin + 273.15 * kelvin)
BOOST_UNITS_LITERAL_SET(lm, luminous_flux, lumen)
BOOST_UNITS_LITERAL_SET(lx, illuminance, lux)
BOOST_UNITS_LITERAL_SET(Bq, activity, becquerel)
BOOST_UNITS_LITERAL_SET(Gy, absorbed_dose, gray)
BOOST_UNITS_LITERAL_SET(Sv, dose_equivalent, sievert)
BOOST_UNITS_LITERAL_SET(kat, catalytic_activity, katal)
BOOST_UNITS_LITERAL_SET(min, time, 60.0 * second)
BOOST_UNITS_LITERAL_SET(h, time, 60.0 * 60.0 * second)
BOOST_UNITS_LITERAL_SET(day, time, 60.0 * 60.0 * 24.0 * second)
BOOST_UNITS_LITERAL_SET(deg, plane_angle, M_PI / 180.0 * radian)
BOOST_UNITS_LITERAL_SET(l, volume, 0.001 * cubic_meter)
BOOST_UNITS_LITERAL_SET(L, volume, 0.001 * cubic_meter)
BOOST_UNITS_LITERAL_SET(t, mass, 1000.0 * kilogram)

} // namespace literals

} // namespace units

} // namespace boost
```

LISTING H.9: ACTUATOR.H

```
//
// Created by peer23peer on 7/23/16.
//

#pragma once

#include <boost/shared_ptr.hpp>

#include <string>
#include <vector>

#include "Controller.h"
#include "Exception.h"

namespace oCpt {

    class iActuator {
    public:
        typedef boost::shared_ptr<iActuator> ptr;

        iActuator();

        virtual ~iActuator();

        virtual void setActuator() = 0;

        virtual void run() = 0;

        virtual void stop() = 0;
    };

    class Actuator : public iActuator {
    public:
```

```
        Actuator();

        virtual ~Actuator() override;

        virtual void setActuator() override;

        virtual void run() override;

        virtual void stop() override;

    };
}
```

LISTING H.10: SENSOR.H

```
//
// Created by peer23peer on 7/15/16.
//

#pragma once

#include <boost/shared_ptr.hpp>
#include <boost/signals2.hpp>
#include <boost/chrono.hpp>
#include <boost/asio/steady_timer.hpp>
#include <boost/asio.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
#include <boost/any.hpp>

#include <string>
#include <vector>

#include "Controller.h"
#include "Exception.h"

#define CAST(x, t) boost::any_cast<t::ReturnValue_t>(x) /*<! CAST the return value of a
 ↪  generic boost::any object, which can change for each sensor to a the proper return value.
 ↪  where the first parameter is the getState().Value and the second is the Sensor Class.
 */

namespace oCpt {

    /*!
     * Each sensor that is used should adhere to the sensor interface. A sensor consits of an
 ↪  connection to a controller, such as a ARM device and the world. The sensor needs to be
 ↪  initiated with the construct, where afterwards the init fucntion is called. The sensor
 ↪  should then be registerd by the Boatswain, using Boatswain::registerSensor(). This
 ↪  ensures that the boatswain can run the sensors. Some sensors are automatically update,
 ↪  whilst other need a manual action, such it's common practice to call the
 ↪  iSensor::updateSensor(). Once the value is update, a new Boost::Signal2 is fired, which
 ↪  allow for the main function to obtain the State of the sensor. Via iSensor::getState().
 ↪  Since the return value of a sensor can vary, it's important to note that the final sensor
 ↪  should include a typedef with the return type named ReturnValue_t. After a sensor update
 ↪  is given or a signal is recieved, the return value can be CAST using the macro CAST(x,t)
     */
    class iSensor {
    public:
        typedef boost::shared_ptr<iSensor> ptr; //<! Boost shared_ptr for a sensor
        typedef boost::signals2::signal<void()> signal_t; //<! Signal type which is emited
         ↪  when a state is changed
        typedef boost::any generic_t; //<! Generic returnvalue, which allows for type-safe
         ↪  Polymorphic return calls
```

```
struct State {
    generic_t Value; //<! a generic return value
    World::Time::timepoint_t Stamp; //<! A Epoch when this value was obained
}; //<! State of a sensor at a certain time

/*!
 * Constructor of iSensor
 * @param controller a shared_ptr of the controller where the sensor is hooked to
 * @param world a shared_ptr of the world in which the vessel operates
 * @param id a identifying name of the sensor
 * @param typeOfSensor a identifying category for the sensor
 */
iSensor(iController::ptr controller, World::ptr world, std::string id, std::string
 ↪  typeOfSensor = "");

/*!
 * Deconstructor of the sensor
 */
virtual ~iSensor();

/*!
 * pure virtual function for the updating of a sensor
 */
virtual void updateSensor() = 0;

/*!
 * pure virtual function for running of the sensor
 */
virtual void run() = 0;

/*!
 * pure virtual function for stopping the sensor
 */
virtual void stop() = 0;

/*!
 * pure virtual function for initializing the sensor
 */
virtual void init() = 0;

/*!
 * pure virtual function for registering the Input Output service
 * @param ioservice teh Input Output service used by Boost ASIO
 */
virtual void setIOservice(boost::shared_ptr<boost::asio::io_service> ioservice) = 0;

/*!
 * Equal operator determining if this sensor is equal with the pointer
 * @param rhs shared_ptr with the other sensor
 * @return returns either true or false
 */
virtual bool operator==(iSensor::ptr rhs);

/*!
 * Get the number of milliseconds when this sensor should be updated
 * @return returns a boost::posix_time::milliseconds type
 */
const boost::posix_time::milliseconds &getTimer() const;

/*!
 * set the number of milliseconds when this sensor should be updated
```

```
         * @param timer  the number of milliseconds as an boost::posix_time::milliseconds
↪  type
         */
        void setTimer(const boost::posix_time::milliseconds &timer);

        /*!
         * get the signal that is to be fired when the state is updated
         * @return the signal_t
         */
        signal_t &getSig();

        /*!
         * gets the last State of the sensor
         * @return the State object. Remember to CAST the value like such
↪  <sensorClass>::ReturnValue_t ret = CAST(<sensorname>->getState().Value, <sensorClass>);
         */
        const State &getState() const;

    protected:
        std::string id_; //<! the string identifier of the sensor
    public:
        /*!
         * get the current ID
         * @return returns the ID as string
         */
        const std::string &getID() const;

        /*!
         * sets the ID of the sensor
         * @param id identifying string
         */
        void setID(const std::string &id);

        /*!
         * get the type of sensor
         * @return  category identifying string
         */
        const std::string &getTypeOfSensor() const;

        /*!
         * sets the category of the sensor, suchs as GPS, temperature
         * @param typeOfSensor category identifying string
         */
        void setTypeOfSensor(const std::string &typeOfSensor);

    protected:
        std::string typeOfSensor_; //<! Type of sensor
        iController::ptr controller_; //< shared_ptr to the controller
        World::ptr world_; //<! shared_ptr to the world
        boost::posix_time::milliseconds timer_; //<! milliseconds interval when an update
         ↪  should take place
        signal_t sig_; //<! the sensor reading complete signal
        State state_; //<! the current state of the sensor
        bool sensorRunning_ ; //<! indication if the sensor is running
        boost::shared_ptr<boost::asio::io_service> ioservice_;
    };

    /*!
     * Implementation of the iSensor interface
     */
    class Sensor : public iSensor {
    public:
```

```
        /*!
         * Constructor of Sensor
         * @param controller a shared_ptr of the controller where the sensor is hooked to
         * @param world a shared_ptr of the world in which the vessel operates
         * @param id a identifying name of the sensor
         * @param typeOfSensor a identifying category for the sensor
         */
        Sensor(iController::ptr controller, World::ptr world, std::string id, std::string
         ↪  typeOfSensor);

        /*!
         * Deconstructor of the Sensor class
         */
        virtual ~Sensor() override;

        /*!
         * virtual function which performs a sensor update, obtaining a  new value and
↪  sending a signal afterwards
         */
        virtual void updateSensor() override;

        /*!
         * virtual function starting the run service for the IO
         */
        virtual void run() override;

        /*!
         * virtual function stopping the run
         */
        virtual void stop() override;

        /*!
         * Initialize the sensor
         */
        virtual void init() override;

        /*!
         * Setting the used Asynchronous Input Output service
         * @param ioservice ASIO IO service, which handles the async calls from multiple
↪  sensors
         */
        virtual void setIOservice(boost::shared_ptr<boost::asio::io_service> ioservice)
         ↪  override;


    };
}
```

LISTING H.11: CONSTANTS.H

```
// Copyright (c) 2013 Andrew Gascoyne-Cecil
//
// Permission is hereby granted, free of charge, to any person obtaining a
// copy of this software and associated documentation files (the "Software"),
// to deal in the Software without restriction, including without limitation
// the rights to use, copy, modify, merge, publish, distribute, sublicense,
// and/or sell copies of the Software, and to permit persons to whom the Software
// is furnished to do so, subject to the following conditions:
//
// The above copyright notice and this permission notice shall be included in
// all copies or substantial portions of the Software.
//
```

```
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
// OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
// FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
// AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
// WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
// CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

#pragma once

#include <boost/units/systems/si.hpp>
#include <boost/units/cmath.hpp>
#include "literals.h"

namespace boost {

namespace units {

namespace constants {

using namespace si;
using namespace literals;

long double operator "" _LD(unsigned long long x) { return static_cast<long double>(x); }
long double operator "" _LD(long double x) { return x; }

const auto M_PI_LD = static_cast<long double>(M_PI);
const auto c = 299792458.0_m / 1_s;
const auto G = 6.67384E-11_m * 1_m * 1_m / 1_kg / 1_s / 1_s;
const auto h = 6.62606957E-34_J * 1_s;
const auto hbar = h / 2_LD * M_PI_LD;
const auto u0 = 4E-7_N * M_PI_LD / 1_A / 1_A;
const auto eps0 = 1_LD / u0 / c / c;
const auto Z0 = u0 * c;
const auto ke = 1_LD / (4_LD * M_PI * eps0);
const auto e = 1.602176565E-19_C;
const auto me = 9.10938291E-31_kg;
const auto mp = 1.672621777E-27_kg;
const auto uB = e * hbar / (2_LD * me);
const auto G0 = 2_LD * e * e / h;
const auto KJ = 2_LD * e / h;
const auto uN = e * h / (2_LD * mp);
const auto RK = h / (e * e);
const auto alpha = u0 * e * e * c / (2_LD * h);
const auto Rinf = alpha * alpha * me * c / (2_LD * h);
const auto a0 = alpha / (4_LD * M_PI_LD * Rinf);
const auto re = e * e / (4_LD * M_PI_LD * eps0 * me * c * c);
const auto Eh = 2_LD * Rinf * h * c;
const auto R = 8.3144621_J / 1_K / 1_mol;
const auto atm = 101325_Pa;
const auto lP = sqrt(hbar * G / (c * c * c));
const auto mP = sqrt(hbar * c / G);
const auto tP = sqrt(hbar * G / (c * c * c * c * c));
const auto NA = 6.02214129E23_LD / 1.0_mol;
const auto k = R / NA;
const auto kB = k;
const auto F = NA * e;
const auto c1 = 2_LD * M_PI * h * c * c;
const auto c2 = h * c / k;
const auto sigma = M_PI_LD * M_PI_LD * k * k * k * k / (60_LD * hbar * hbar * hbar * c * c);
const auto b = h * c / (4.965114231_LD * k);
const auto g = 9.80665_m / 1_s / 1_s;
```

```
} // namespace constants

} // namespace units

} // namespace boost
```

LISTING H.12: CAPTAIN.H

```
//
// Created by peer23peer on 7/23/16.
//

#pragma once

#include <boost/shared_ptr.hpp>

#include "World.h"

namespace oCpt {
    class iCaptain {
    public:
        typedef boost::shared_ptr<iCaptain> ptr;

        iCaptain(World::ptr world);

        virtual ~iCaptain();

        virtual void run() = 0;

        virtual void stop() = 0;

        virtual void initialize() = 0;

        const boost::shared_ptr<bool> &getStopThread_() const;

        void setStopThread_(const boost::shared_ptr<bool> &stopThread_);

    protected:
        boost::shared_ptr<bool> stopThread_;
        boost::shared_ptr<bool> localStopThread_;
        World::ptr world_;
    };

    class Captain : public iCaptain {
    public:
        Captain(World::ptr world);

        virtual ~Captain() override;

        virtual void run() override;

        virtual void stop() override;

        virtual void initialize() override;

    };
}
```

LISTING H.13: BOATSWAIN.H

```
//
// Created by peer23peer on 7/21/16.
```

```
//

#pragma once

#include <thread>
#include <vector>

#include <boost/asio.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>
#include <boost/shared_ptr.hpp>
#include <boost/enable_shared_from_this.hpp>
#include <boost/bind.hpp>
#include <boost/ref.hpp>

#include "Controller.h"
#include "Sensor.h"
#include "Actuator.h"
#include "Communication.h"

namespace oCpt {

    /*!
     * The Boatswain performs all the labours tasks, suchs updateing and interpretting sensor
 ↪  readings, setting actuators according to the Captain wishes, updating the state
 ↪  representation of the vessel in the World. Each Boatswain runs on its own thread. it's
 ↪  possible for a vessel to have multiple Boatswains, responsible for multiple tasks, such
 ↪  as communication, localization, steering. Each Boatswain has to adhere to the iBoatswain
 ↪  interface.
     */
    class iBoatswain : public boost::enable_shared_from_this<iBoatswain> {
    public:
        typedef boost::shared_ptr<iBoatswain> ptr; //<! Boost shared_ptr for a Boatswain
        typedef boost::shared_ptr<boost::asio::deadline_timer> timerPtr; //<! Boost
         ↪  shared_ptr Dealine timer //TODO check if this is still needed could be left over
         ↪  from old construct

        /*!
         * Constructor for a iBoatswain
         * @param controller a shared_ptr to the controller with which teh Boatswain
 ↪  interacts
         */
        iBoatswain(iController::ptr controller);

        /*!
         * Deconstructor for the iBoatswain
         */
        virtual ~iBoatswain();

        /*!
         * pure virtual function for running the boatswain and his registered sensors
         */
        virtual void run() = 0;

        /*!
         * pure virtual function for stopping the run task
         */
        virtual void stop() = 0;

        /*!
         * pure virtual function of initialzing the Boatswain
         */
        virtual void initialize() = 0;
```

```
        /*!
         * Pure virtual function for registering a new sensor with the Boatswain
         * @param sensor a shared_ptr to a Sensor which need to maintained by the Boatswain
         */
        virtual void registerSensor(iSensor::ptr sensor) = 0;

        /*!
         * Pure virtual function for registering a new actuator with the Boatswain
         * @param actuator a shared_ptr to an Actuator which need to be maintained by the
↪   Boatswain
         */
        virtual void registerActuator(iActuator::ptr actuator) = 0;

        /*!
         * Pure virtual function for registering a new communication device which
         * @param comm
         */
        virtual void registerComm(iComm::ptr comm) = 0;

        /*!
         * get if the thread is stopped
         * @return returns if the thread should stop
         */
        const boost::shared_ptr<bool> &getStopThread() const;

        /*!
         * set the value of the stopthread
         * @param stopThread //<! a shared_ptr to the boolean
         */
        void setStopThread(const boost::shared_ptr<bool> &stopThread);

        /*!
         * get the used Input Output service
         * @return a shared_ptr to the ASIO io service
         */
        boost::shared_ptr<boost::asio::io_service> &getIOservice();

    protected:
        boost::shared_ptr<boost::asio::io_service> ioservice_; //<! a shared_ptr to the used
         ↪  io service;
        iController::ptr controller_; //<! a shared_ptr to the controller
        std::vector<timerPtr> timers_; //<! a vector consisting of multiple dead_line timers
        std::vector<iSensor::ptr> timerSensors_; //<! a vector of Sensor which need to be
         ↪  timed
        std::vector<iSensor::ptr> manualSensors_; //<! a vector of Sensor which need to be
         ↪  manually updated
        boost::shared_ptr<bool> stopThread_; //<! a shared_ptr to a boolean, which is
         ↪  connected with other threads that need to be stopped/started simultaneous
        boost::shared_ptr<bool> localStopThread_;

        /*!
         * Pure virtual function for resetting the timer
         * @param sensor
         */
        virtual void resetTimer(iSensor::ptr sensor) = 0;
    };

    /*!
```

```
 * The Boatswain performs all the labours tasks, suchs updateing and interpretting sensor readings, setting actuators according to the Captain wishes, updating the state representation of the vessel in the World. Each Boatswain runs on its own thread. it's possible for a vessel to have multiple Boatswains, responsible for multiple tasks, such as communication, localization, steering. Each Boatswain has to adhere to the iBoatswain interface.
 */
class Boatswain : public iBoatswain {
public:
    /*!
     * The constructor for a Boatswain
     * @param controller a shared_ptr to the controller with which teh Boatswain interacts
     */
    Boatswain(iController::ptr controller);

    /*!
     * Deconstructor
     */
    virtual ~Boatswain() override;

    /*!
     * Make the Boatswain work and execute the actuators, sensors and communications
     */
    virtual void run() override;

    /*!
     * Stop the execution of the tasks
     */
    virtual void stop() override;

    /*!
     * Initialize the Boatswain
     */
    virtual void initialize() override;

    /*!
     * Register a new Sensor with the Boatswain. If the Timer for the Sensor is set to a value greater the 0, the Sensor is registered with the timerSensors_ and a timer is set. Otherwise the Sensor is registered as a manualSensors_
     * @param sensor a shared_ptr to a Sensor
     */
    virtual void registerSensor(iSensor::ptr sensor) override;

    /*!
     * Register a new Actuator with the Boatswain
     * @param actuator a shared_ptr to an Actuator
     */
    virtual void registerActuator(iActuator::ptr actuator) override;

    /*!
     * Register a new iComm device by setting a shared IO service
     * @param comm
     */
    virtual void registerComm(iComm::ptr comm) override;

protected:
    /*!
     * reset the timer of a Sensor
     * @param sensor a shared_ptr to the Sensor
     */
    void resetTimer(iSensor::ptr sensor) override;
```

```
    };
}
```

LISTING H.14: VESSEL.H

```
//
// Created by peer23peer on 7/16/16.
//

#pragma once

#include <boost/shared_ptr.hpp>
#include <boost/enable_shared_from_this.hpp>

#include <vector>
#include <thread>

#include "World.h"
#include "Controller.h"
#include "Captain.h"
#include "Boatswain.h"
#include "Actuator.h"
#include "Communication.h"

namespace oCpt {
    /*!
     * The interface for each vessel
     */
    class iVessel {
    public:
        typedef boost::shared_ptr<iVessel> ptr; //!< Boost shared_ptr to a vessel

        /*!
         * Constructor of the vessel interface
         * @return
         */
        iVessel();

        /*!
         * Constructor of the vessel interface
         * @param controller shared_ptr to the controller
         * @return
         */
        iVessel(iController::ptr controller);

        /*!
         * Deconstructor
         */
        virtual ~iVessel();

        /*!
         * Initialize the vessel
         */
        virtual void initialize() = 0;

        /*!
         * Run the vessel normal operations
         */
        virtual void run() = 0;

        /*!
```

```
         * Stop the vessel, everything except critical parts, which are needed to survive
         */
        virtual void stop() = 0;

        /*!
         * Get the stop thread variable
         * @return shared_ptr for each each thread;
         */
        const boost::shared_ptr<bool> &getStopThread() const;

        /*!
         * Set the stop thread variable
         * @param stopThread a shared_ptr for all threads
         */
        void setStopThread(const boost::shared_ptr<bool> &stopThread);

    protected:
        boost::shared_ptr<bool> stopThread_; //!< The global shared pointer for stopping all
         ↪  threads
    };

    /*!
     * The vessel base class
     */
    class Vessel : public iVessel {
    public:
        /*!
         * The constructor for a vessel
         * @return
         */
        Vessel();

        /*!
         * The constructor for a vessel
         * @param controller shared_ptr to the controller
         * @return
         */
        Vessel(iController::ptr controller);

        /*!
         * The deconstructor
         */
        virtual ~Vessel();

        /*!
         * Initialize the vessel
         */
        virtual void initialize() override;

        /*!
         * Run the vessel normal operations
         */
        virtual void run() override;

        /*!
         * Stop the vessel, everything except critical parts, which are needed to survive
         */
        virtual void stop() override;

    protected:
        World::ptr world_; //!< a shared_ptr to the world needed for time and location
         ↪  keeping
```

```
        iController::ptr controller_; //!< a shared_ptr to the controller needed for sensors,
         ↪  actuators and coommunication
        iCaptain::ptr captain_; //!< The captain for strategical planning
        iBoatswain::ptr boatswain_; //!< The boatswain, the worker asynchronize operations
         ↪  for actuators, sensors, and communication
        std::vector<iSensor::ptr> sensors_; //<! sensor vector
        std::vector<iActuator::ptr> actuators_; //<! actuator vector
        std::vector<iComm::ptr> comm_; //<! communication device vector
    };

}
```

LISTING H.15: EXCEPTION.H

```
//
// Created by peer23peer on 7/17/16.
//

#pragma once

#include <iostream>
#include <exception>
#include <string>

namespace oCpt {
    class oCptException : public std::exception {
    public:
        oCptException(std::string msg = "exception!", int id = -1) : _msg(msg), _id(id) {}

        ~oCptException() throw() {}

        const char *what() const throw() { return _msg.c_str(); }

    private:
        std::string _msg;
        int _id;
    };
}
```

LISTING H.16: WORLD.H

```
//
// Created by peer23peer on 7/15/16.
//

#pragma once

#include <vector>

#include <boost/shared_ptr.hpp>
#include <boost/chrono.hpp>
#include <boost/geometry.hpp>

#include <boost/units/io.hpp>
#include <boost/units/systems/si.hpp>
#include <boost/units/systems/angle/degrees.hpp>

//#include "constants.h"
//#include "literals.h"

using namespace boost::units;
//using namespace boost::units::literals;
```

```
//using namespace boost::units::constants;

namespace oCpt {
    /*!
     * The World class, this class is an shared pointer where the boatswain can place the
 ↪  state representation of the vessel at a certain time, which allows the captain to plan
 ↪  the strategic decisions
     */
    class World {
    public:
        typedef boost::shared_ptr<World> ptr; //!< Boost shared_ptr to a World class

        /*!
         * The Time class all things time related, which allow for easy consite time
 ↪  manupulation trhough out the classes TODO add elapsed time function with operator+ and
 ↪  operator- to return quantity<seconds>
         */
        class Time {
        public:
            typedef boost::shared_ptr<Time> ptr; //!< Boost shared_ptr to a Time class
            typedef boost::chrono::steady_clock::period tick_period; //!< a tick period for a
             ↪  steady clock
            typedef boost::chrono::steady_clock clock_t; //<! A steady clock taking time from
             ↪  the OS
            typedef boost::chrono::time_point<clock_t> timepoint_t; //<! A point in time
             ↪  generated by a steady clock
        private:
            clock_t timeClock_; //<! The steady clock
        public:
            /*!
             * Constructor of the Time class
             */
            Time();

            /*!
             * Deconstructor of the Time class
             */
            virtual ~Time();

            /*!
             * A template class to Log generic values at an certain epoch in time
             * @tparam T Type of value to log
             */
            template<typename T>
            class Log {
            public:
                typedef boost::shared_ptr<Log> ptr; //<! A Boost shared pointer to a Log
                 ↪  class
            private:
                timepoint_t _epoch; //<! An point in in type
                T _value; //<! The Value to be stored
            public:

                /*!
                 * Constructor of the Log class
                 */
                Log() {}

                /*!
                 * Constructor of the Log class
                 * @param value The Value to store
                 * @param epoch the Time point, with a default to the now moment
```

```
             */
            Log(const T &value, const timepoint_t &epoch = clock_t::now()) {
                this->_value = value;
                this->_epoch = epoch;
            }

            /*!
             * Deconstructor of the Log class
             */
            virtual ~Log() {}

            /*!
             * Get the current Epoch
             * @return returns a time point when the Log has taken place
             */
            const timepoint_t &getEpoch() const {
                return _epoch;
            }

            /*!
             * Gets the current value
             * @return returns the value at an certain time
             */
            const T &getValue() const {
                return _value;
            }
        };

        template<typename T>
        using History = std::vector<boost::shared_ptr<Log<T>>>; //<! A vector of Log items

        /*!
         * get the current TimeClock
         * @return returns the time clock
         */
        clock_t &getTimeClock();

        /*!
         * Get the current time, as in now
         * @return returns a timepoint_t which is now
         */
        timepoint_t now();
    };

    /*!
     * A location in the World
     */
    class Location {
    public:
        typedef quantity<boost::units::degree::plane_angle, double> degree_t;

        enum cardinal_direction {
            North = 110, /**< enum value North */
            South = 115, /**< enum value South */
            East = 101, /**< enum value East */
            West = 119 /**< enum value West */
        }; //<! The cardinal direction of a hemisphere

        typedef struct coordinate {
            degree_t value; //<! number of degrees
            cardinal_direction direction; //<! Direction
```

```
        } coordinate_t; //<! A coordinate value type, consiting of a number and a
         ↪ cardianal direction

        typedef struct gpsPoint {
            coordinate_t longitude; //<! The longitude of a GPS coordinate
            coordinate_t latitude; //<! The latitude of a GPS coordinate
            double height; //<! The height of location

            /*!
             * Convert a gps coordinate to a text string
             * @return a text string eq. 5.000E,52.000N
             */
            std::string toString();
        } gpsPoint_t; //<! A GPS point consiting of longtitude, latitude and height value

        typedef boost::shared_ptr<Location> ptr; //<! Boost shared_ptr to a Location

        struct RoutePoint {
            typedef boost::shared_ptr<RoutePoint> ptr; //<! Boost shared_ptr to a
             ↪ Routepoint
            Time::timepoint_t TimePoint; //<! A point in time when this waypoint should
             ↪ or has been reached
            gpsPoint_t Location; //<! A GPS location of the waypoint
        }; //<! A way point on a route

        /*!
         * Constructor for the Location
         */
        Location();

        /*!
         * Deconstruction for the Location
         */
        virtual ~Location();

        /*!
         * get the current Location
         * @param newMeasurement should a new measurement be executed? or is the lattest
↪  log sufficient
         * @return returns the last Way point
         */
        RoutePoint::ptr getCurrentLocation(bool newMeasurement = false);

        /*!
         * Add a new waypoint to the history log
         * @param routePoint a waypoint
         */
        void push_back(RoutePoint::ptr routePoint);

        /*!
         * Get the complete location history
         * @return returns a vector with shared_ptr of all waypoints reached
         */
        std::vector<RoutePoint::ptr> getLocationHistory();

        /*!
         * Convert a string to a cardinal direction
         * @param str North/north/N/n / West,west,W,w / South,south,S,s / East,east,E,e
↪  are taken as argument
         * @return a cardinal direction
         */
        static cardinal_direction stocd(std::string str);
```

```
        private:
            RoutePoint::ptr currentLocation_; //<! a pointer to the last location
            std::vector<RoutePoint::ptr> LocationHistory; //<! The location history
        };


        /*!
         * Constructor for a World
         */
        World();

        /*!
         * Deconstuctor for a World
         */
        virtual ~World();

        /*!
         * get the current time object
         * @return returns Time
         */
        Time &getTime();

        /*!
         * Get the current Epoch
         * @return returns a timepoint representing now
         */
        Time::timepoint_t now();

    protected:
        Time time_; //<! The time object

    };


}
```

LISTING H.17: CONTROLLER.H

```
//
// Created by peer23peer on 7/15/16.
//

#pragma once

#include <boost/shared_ptr.hpp>
#include <boost/shared_array.hpp>
#include <boost/enable_shared_from_this.hpp>
#include <boost/system/error_code.hpp>
#include <boost/date_time/posix_time/posix_time_duration.hpp>
#include <boost/asio.hpp>
#include <boost/function.hpp>
#include <boost/bind.hpp>
#include <boost/signals2.hpp>
#include <boost/filesystem.hpp>

#include <string>
#include <vector>
#include <mutex>
#include <stdexcept>
#include <deque>
```

```
#include "World.h"

#define MAX_READ_LENGTH 4096 //<! Maximum number of bytes that the serialport should read at
 ↪ a time

#define BBB_CAPE_MNGR "/sys/devices/platform/bone_capemgr/slots" //<! The path to the bone
 ↪ capemanger slots

#define GPIO_BASE_PATH "/sys/class/gpio/" //<! The path to the gpio

#define ADC_IO_BASE_PATH "/sys/bus/iio/devices/iio:device" //<! the path to the ADC channels
#define ADC_VOLTAGE_PATH "/in_voltage" //<! the voltage file, the value stored in this value
 ↪ is the 1.8V / 4096 voltage
#define ADC_VOLTAGE_SUB_PATH "_raw" //<! supplement to the ADC_VOLTAGE_PATH

#define MODULE_PATH "/proc/modules" //<! Path to the modules loaded

namespace oCpt {

    namespace protocol {

        /*!
         * Functions and routines related to the Linux userspace. Checking if a file exist,
 ↪ if capes or modules are loaded etc.
         */
        class userspace {
        public:
            /*!
             * The constructor
             */
            userspace();

            /*!
             * The deconstructor
             */
            virtual ~userspace();

        protected:
            /*!
             * Checks if a Linux module is loaded
             * @param modName the name of the module as string
             * @return either true or false
             */
            bool modLoaded(std::string modName);

            /*!
             * Checks if a file exist
             * @param fileName the filename as string
             * @return either true or false
             */
            bool fileExist(std::string fileName);

            /*!
             * Checks if a Device Tree overlay is loaded
             * @param dtboName The devicetree overlay as a string
             * @return either true or false
             */
            bool dtboLoaded(std::string dtboName);

            /*!
             * The standard Mutex
             * TODO check if this is really needed for the current setup
```

```
             */
            std::mutex usMutex;
        };

        /*!
         * The Analogue to Digital converter class. This class reads the voltage of an
↪   analogie pin, from user space.
         */
        class adc : public userspace {
        public:
            typedef boost::shared_ptr<adc> ptr; //<! Boost shared_ptr

            /*!
             * The constructor of the adc class
             * @param id the pin ID as an uint8_t value
             * @param device the device or chip which handles the communication with the
↪   analogue pins
             * @param modName the name of the modules which needs to be loaded
             * TODO check if it's allways needed to load a module
             */
            adc(uint8_t id, uint8_t device, std::string modName = "");

            /*!
             * The deconstrcutor
             */
            virtual ~adc();

            /*!
             * gets the current raw voltage level as resolution
             * @return the raw voltage level as uint16_t
             */
            uint16_t &getValue();

            /*!
             * Checks if adc object is the same
             * @param rhs other adc object, to be checked against
             * @return either true or false
             */
            bool operator==(const adc &rhs);

            /*!
             * Compare function
             * @param id ID to be checked
             * @param device Device name to be checked
             * @return either true or false
             */
            bool compare(const uint8_t &id, const uint8_t &device = 0);

        private:
            uint8_t id_ = 0; //<! The pin ID
            uint8_t device_ = 0; //<! The device ID
            std::string path_ = ""; //<! the user-space path to the device
            uint16_t value_ = 0; //<! The last read value
        };

        /*!
         * A General Pin Input Output class. This is the class that handles gpio's in user
↪   space. Each pin can be set as either input or output, and have a High or a Low
↪   out-/input. When a pin is set as input, it can be polled on the edge, execute a function
↪   or send a signal on the rising, falling or changing edge of the signal
         */
        class gpio : public userspace {
```

```
    public:
        typedef boost::shared_ptr<gpio> ptr; //<! Boost shared_ptr for a gpio
        typedef boost::signals2::signal<void()> signal_t; //<! the signal typedef
        typedef std::function<void()> cb_func; //<! the funcion typedef which can be
         ↪ bound

        /*!
         * The Direction of the pin
         */
        enum Direction {
            INPUT = 105, //<! A pin that is set as input
            OUTPUT = 111 //<! A pin that is set as output
        };

        enum Value {
            LOW = 48, //<!  The pin is (set) low
            HIGH = 49 //<! The pin is (set) high
        };

        enum Edge {
            NONE = 110, //<! The edge of the signal is unimportant, usually when the pin
             ↪ is Direction is set to Direction::OUTPUT
            RISING = 114, //<! Polling is triggered when the signal is rising, changing
             ↪ from a Value::LOW to a Value::HIGH signal
            FALLING = 102, //<! Polling is triggered when the signal is falling, changing
             ↪ for a Value::HIGH to Value::LOW signal
            BOTH = 98 //<! Polling is triggered when the signal changes, either from
             ↪ Value::HIGH to Value::LOW or from Value::LOW to Value::HIGH
        };

        /*!
         * The constructor for the gpio class
         * @param pinNumber the pin pumber (in user-space mapping)
         * @param direction the Direction of a pin, with a default value as
↪ Direction::INPUT
         * @param value the start Value of a pin. With a default value of Value::LOW
         * @param edge the Edge of a pin with the default value of Edge::NONE
         */
        gpio(int pinNumber, Direction direction = INPUT, Value value = LOW, Edge edge =
         ↪ NONE);

        /*!
         * The deconstructor
         */
        ~gpio();

        /*!
         * Get the current pin number
         * @return an int representing the pin number in user-space mapping
         */
        int getPinNumber() const;

        /*!
         * Set the new pinbumber (don't use yet)
         * @param pinNumber the pinmuber to be set
         */
        void setPinNumber(int pinNumber);

        /*!
         * Get the current value of the pin, if the Direction is set to Direction::INPUT
↪ the value is obtained from the user space, otherwise the value is read from object itself
         * @return either Value::HIGH or Value::LOW
```

```
             */
            Value getValue() const;

            /*!
             * Set the current value, if the Direction is set to Direction::OUTPUT the value
↪  is set ti userspace, either it's set to object itself
             * @param value
             */
            void setValue(Value value);

            /*!
             * Get the current Direction. note this doesn't take into accoutn external
↪  changes done outside this library
             * @return either Direction::INPUT or Direction::Output
             */
            Direction getDirection() const;

            /*!
             * Set the Direction of the pin
             * @param direction the Direction of the pin
             */
            void setDirection(Direction direction);

            /*!
             * Get the current Edge of the pin. If the Direction is set to Direction::INPUT
↪  the value is set in user-space, otherwise it's set in the object itself
             * @return
             */
            Edge getEdge() const;

            /*!
             * Set the Edge of the of the pin. if the Direction is set to Direction::INPUT,
↪  the value is set in user-space, otherwise it's set in the object itself
             * @param edge
             */
            void setEdge(Edge edge);

            /*!
             * Set a new Callbackfunction which is called on a certain Edge
             * @param cb the callback function
             */
            void setCallbackFunction(cb_func cb);

            /*!
             * Wait for the occurance of a change in Edge, corresponding with the set value
↪  of Edge. When the change is detected the callbackfunction is called. This function blocks
↪  the current thread.
             */
            void waitForEdge();

            /*!
             * Wait for the ocurrance of a change in Edge, corresponding with the set value
↪  of Edge. When the chance is detected the callbackfunction is called. This function
↪  creates a new thread, allowing the current thread to run unhindered
             */
            void waitForEdgeAsync();

            /*!
             * Static function which creates a vector containing new gpio shared_ptr for each
↪  pin that is currently exported in the user space.
             * @return A vector with shared_ptr's of all exported gpio's in user-space
             */
```

```
            static std::vector<ptr> exportedGpios();

            /*!
             * Toggle the value_ of the pin if Value::High then the value_ is set to
↪  Value::LOW
             */
            void toggle();

            /*!
             * The signal that is send if the internal callback fucntion is executed
             */
            signal_t signalChanged;

        private:
            int pinNumber_; //<! The pin number
            Value value_; //<! The value of the pin
            Direction direction_; //<! The direction of the pin
            Edge edge_; //<! The edge of the pin
            std::string gpiopath_; //<! The path pointing to the gpio in user-space
            cb_func cb_; //<! A pointer to the callbackfunction
            bool threadRunning_; //<! Is pin polling running on a seperate thread

            /*!
             * The internal callback function, which triggers the signalChanged signal
             */
            void internalCbFunc();

            /*!
             * Export the pin in user-space
             * @param number the pin number to be exported
             */
            void exportPin(const int &number);

            /*!
             * Unexport the pin in user-space
             * @param number the pin number to be unexported
             */
            void unexportPin(const int &number);

            /*!
             * Static generic function returning the value in user-space of either Direction,
↪  Edge or Value. Depending on the typename
             * @tparam T The value to return either the Value, Edge or Direction
             * @param number the pin as number
             * @return The read value as either Value, Edge or Direction
             */
            template<typename T>
            static T readPinValue(const int &number) {
                std::string path = GPIO_BASE_PATH;
                path.append("gpio" + std::to_string(number));
                return readPinValue<T>(path);
            }

            /*!
             * Static generic function returning the value in user-space of either Direction,
↪  Edge or Value. Depending on the typename. This function is quicker then the overload
↪  function taking the pin number as int. and is therefore preffered to obtain the Value.
             * @tparam T The value to return either the Value, Edge or Direction
             * @param path the pin as user-space path
             * @return the read value as either Value, Edge or Direction
             */
            template<typename T>
```

```
            static T readPinValue(std::string path) {
                T retVal;
                // Set the file path determined by the requested return type
                if (std::is_same<T, Value>::value) {
                    path.append("/value");
                } else if (std::is_same<T, Direction>::value) {
                    path.append("/direction");
                } else if (std::is_same<T, Edge>::value) {
                    path.append("/edge");
                }

                // Get the first character in the file and cast it to the enum. The enums
                 ↪  have corresponding values, so this is a very quick conversion
                std::ifstream fs(path);
                char c;
                fs.get(c);
                retVal = static_cast<T>(c);
                fs.close();
                return retVal;
            }

            /*!
             * Write the value to the pin. The T parameter determines which value to set
             * @tparam T the type could either be Value, Direction ro Direction
             * @param number the pin number as an integer
             * @param value the Value to be set
             */
            template<typename T>
            void writePinValue(const int &number, const T &value) {
                std::string path = GPIO_BASE_PATH;
                path.append("gpio" + std::to_string(number));
                writePinValue<T>(path, value);
            }

            /*!
             * Write the value to the pin, The T paramter determines which value to set. This
↪  overload is quicker then the one taking the integer and is therefore preferred
             * @tparam T the type could either be Value, Direction or Edge
             * @param path the pin as an user-space path
             * @param value the Value to write
             */
            template<typename T>
            void writePinValue(std::string path, const T &value) {
                if (std::is_same<T, Value>::value) {
                    path.append("/value");
                    std::ofstream fs(path);
                    fs << (value - 48);
                    fs.close();
                    return;
                } else if (std::is_same<T, Direction>::value) {
                    path.append("/direction");
                    std::ofstream fs(path);
                    switch (value) {
                        case Direction::OUTPUT:
                            fs << "out";
                            break;
                        case Direction::INPUT:
                            fs << "in";
                            break;
                    }
                    fs.close();
                    return;
```

```
            } else if (std::is_same<T, Edge>::value) {
                path.append("/edge");
                std::ofstream fs(path);
                switch (value) {
                    case Edge::NONE:
                        fs << "none";
                        break;
                    case Edge::RISING:
                        fs << "edge";
                        break;
                    case Edge::FALLING:
                        fs << "falling";
                        break;
                    case Edge::BOTH:
                        fs << "both";
                        break;
                }
                fs.close();
            }
            return;
        }
    };

    /*!
     * Communication via the serial port, using an Asynchrounous Input Output setup,
↪ provided by Boost. All communication is handled on the background via a io_service. When
↪ data is recieved a callback function is called. This can either be an external function
↪ or an internal one, which sends a signal for each new line. The lines can then be read
↪ using a FiFo function.
     */
    class Serial : public userspace {
    public:
        typedef boost::shared_ptr<Serial> ptr; //<! a Boost shared_ptr
        typedef std::function<void(const unsigned char *,
                                   size_t)> cb_func; //<! Callback function typedef of
                                    ↪ the function that is called when data is recieved
        typedef boost::asio::serial_port_base::parity parity_t; //<! Parity typedef
        typedef boost::asio::serial_port_base::character_size character_size_t; //<!
         ↪ Character size typedef
        typedef boost::asio::serial_port_base::flow_control flow_control_t; //<! Flow
         ↪ control typedef
        typedef boost::asio::serial_port_base::stop_bits stop_bits_t; //<! Stopbit
         ↪ typedef
        typedef boost::shared_ptr<boost::asio::io_service> io_service_t; //<! io_service
         ↪ typedef
        typedef boost::asio::serial_port serialport_t; //<! Serialport typedef
        typedef boost::signals2::signal<void()> signal_t; //<! Signal typedef

        /*!
         * Constructor of the Serial class
         * @param device a string representing the device path eq. /dev/tty0
         * @param baudrate the baudrate of the device eq. 9600, 57600, 115200
         * @param ioservice the io service to be used standard it's a new service
         * @param parity the parity of the Serial port with a standard parity of
↪ Parity_t::none
         * @param csize The character_size with a standard value of 8
         * @param flow The flow control of thye device with a standard value of
↪ flow_control_t::none
         * @param stop The stop bit of the device with a standard value of
↪ stop_bits_t::none
         * @param maxreadlentgh the Maximum buffer with a standard value of 4096
         */
```

```
        Serial(const std::string &device, unsigned int baudrate,
               io_service_t ioservice = io_service_t(new boost::asio::io_service()),
               parity_t parity = parity_t(parity_t::none),
               character_size_t csize = character_size_t(8),
               flow_control_t flow = flow_control_t(flow_control_t::none),
               stop_bits_t stop = stop_bits_t(stop_bits_t::one),
               unsigned int maxreadlentgh = MAX_READ_LENGTH);

        /*!
         * Open teh serial port
         */
        void open();

        /*!
         * Start the io_service on a sepearte thread
         */
        void start();

        /*!
         * Checks if the port is open
         * @return either true or false
         */
        bool isOpen();

        /*!
         * Closes the port when it's open
         */
        void close();

        /*!
         * Write a message to the port
         * @param msg a string with the payload
         * @return either true or false depending if writting was sucsesfull or not
         */
        bool write(const std::string &msg);

        /*!
         * Write a message as a vector of unsigned chars
         * @param data the message to be send
         * @return either true or false depending if writting was sucsesfull or not
         */
        bool write(const std::vector<unsigned char> &data);

        /*!
         * Set a new callback function
         * @param cb_function the callback function
         */
        void setReadCallback(cb_func cb_function);

        /*!
         * set a new IO service
         * @param io_ptr a shared_ptr to the new IO service
         */
        void setIOservice(boost::shared_ptr<boost::asio::io_service> io_ptr);

        /*!
         * Get the complete returnMsg que
         * @return a deque with all the return lines
         */
        std::deque<std::string> *getReturnMsgQueue();

        /*!
```

```
             * Gets the first recieved message, which is then removed from the queue
             * @return
             */
            std::string readFiFoMsg();

            /*!
             * The signal which is send when a new line has been recieved or the buffer is
↪  full
             */
            signal_t msgRecievedSig;

        protected:
            /*!
             * The internal callback function, which handles messages longer then
↪  maxreadlentgh and splits the message with \r\n
             * @param data the buffer obtained by the serial port
             * @param size the size obtained
             */
            void internalCallback(const unsigned char *data, size_t size);

            /*!
             * The callback function which is called after the port is closed
             * @param error an past trhough boost::system::error_code
             */
            void closeCallback(const boost::system::error_code &error);

            /*!
             * The callbackfunction to be performed when reading is complete
             * @param error boost::system::error_code if an error is presented
             * @param bytes_transferred number of bytes that are transfered
             */
            void readComplete(const boost::system::error_code &error, size_t
             ↪  bytes_transferred);

            /*!
             * When the writing is finished call this function, which will write the next
↪  message if present
             * @param msg the message to write as an vector of unsigned char
             */
            void writeCallback(const std::vector<unsigned char> &msg);

            /*!
             * Start with the write sequence
             */
            void writeStart();

            /*!
             * restart the write process when the previous write is finished
             * @param error
             */
            void writeComplete(const boost::system::error_code &error);

            /*!
             * Start the reading process
             */
            void ReadStart();

            unsigned int maxReadLength_; //<! The maximum length of the buffer
        public:
            /*!
             * Set the maximum buffer of the Serial class
             * @param maxReadLength the number of bytes
```

```
             */
            void setMaxReadLength(unsigned int maxReadLength);

        protected:
            std::deque<std::vector<unsigned char» msgQueue_; //<! The send message queu
            std::deque<std::string> returnMsgQueue_; //<! The receive message queu
            std::string msg_; //<! The message
            std::string receivedMsg_; //<! the recieved message as string
            unsigned char read_msg[MAX_READ_LENGTH]; //<! the message buffer
            cb_func callback_; //<! pointer to the callback function
            io_service_t ioservice_; //<! a pointer to the Input Output service
            std::string device_; //<! the path to the device
            unsigned int baudrate_; //<! the baudrate
            parity_t parity_; //<! the parity
            character_size_t csize_; //<! the character size
            flow_control_t flow_; //<! the control flow
            stop_bits_t stop_; //<! the stop bits
            serialport_t serialport_; //<! a pointer to the serial port
            bool firstMsg = true; //<! is this the first message recieved
        };
    }

    /*!
     * The interface for a controller. Each controller like for instance a Beaglebone black,
↪   Raspberry PI or x64 computer, should adhere to this interface
     */
    class iController {
    public:
        typedef boost::shared_ptr<iController> ptr; //<! the boost shared_ptr typedef
        typedef boost::shared_ptr<boost::asio::io_service> io_t; //<! The ASIO Input Output
         ↪  service typedef

        /*!
         * The constructor
         * @param world a pointer to the World
         */
        iController(World::ptr world);

        /*!
         * The deconstructor
         */
        virtual ~iController();

        /*!
         * A pure virtual function which gets a pointer to all available ADC, if present.
↪   TODO check how it handles no ADC presents
         * @return a vector with pointers the all available ADCs
         */
        virtual std::vector<protocol::adc::ptr> *getAdcVector() = 0;

        /*!
         * A pure virtual function which gets a Pointer to a specific ADC
         * @param id The pin ID
         * @param device the device ID
         * @return a pointer to the requested ADC
         */
        virtual protocol::adc::ptr getADC(uint8_t id, uint8_t device) = 0;

    protected:
        std::vector<protocol::adc::ptr> adcVector_; //<! A vector with shared_ptrs of
         ↪  available ADC
        World::ptr world_; //<! A pointer to the World
```

```
    };

    /*!
     * An ARM like controller. Currently only ARM devices are implemented
     */
    class ARM : public iController {
    public:
        /*!
         * The constructor of an ARM controller
         * @param world a shared_ptr to the World
         */
        ARM(World::ptr world);

        /*!
         * The deconstructor
         */
        virtual ~ARM();

        /*!
         * Obtain a vector of available ADCs
         * @return
         */
        virtual std::vector<protocol::adc::ptr> *getAdcVector();

        /*!
         * Get a specific shared_ptr to an ADC
         * @param id the pin ID
         * @param device the device ID
         * @return returns the specified ADC
         */
        virtual protocol::adc::ptr getADC(uint8_t id, uint8_t device);
    };
}
```

LISTING H.18: TASK.H

```
//
// Created by peer23peer on 7/9/16.
//

#pragma once

#include <boost/shared_ptr.hpp>

#include <list>

#include "Vessel.h"

namespace oCpt {

    /*! \brief Task interface, all tasks need to adhere to this structure
     *
     * This interface make sure that all task adheres to the same runtime rules and
     * enable run-time polymorphism
     */
    class iTask {
    public:
        typedef boost::shared_ptr<iTask> ptr; //!< Boost shared_ptr to a task
        typedef std::list<iTask::ptr> taskqueue; //!< A list of shared pointer tasks

        taskqueue Work;
```

```
class Status {
public:
    typedef boost::shared_ptr<iTask::Status>
            ptr; //!< Boost shared_ptr to the task status

    /*!
     * Constructor of the iTask
     * @return
     */
    Status();

    /*!
     * Deconstructor
     */
    virtual ~Status();

    /*!
     * Show the progress of the task
     * @return double between 0..1
     */
    double progress();

    /*!
     * Returns the running state of the task
     * @return bool where running is true
     */
    bool running();

    /*!
     * Returns if the task was completed succesfully
     * @return bool where a succesfully completed task is true, task in progress
     * or failed are false
     */
    bool successful();

private:
    double _progress = 0.0; //<! The current progress of task
    bool _running = false;  //<! Returns true if a task is running
    bool _successful =
            false; //<! Returns true if a task is completed succesfuly
};

/*!
 * Enumeration indicating which type of task the object is
 */
enum TypeOf {
    ROUTE = 1, WORK = 2
};

/*!
 * Constructor of the interface
 * @return
 */
iTask(iVessel::ptr vessel, bool concurrent = false);

/*!
 * Deconstructor of the interface
 */
virtual ~iTask();

/*!
 * The start command for a task
```

```
     */
    virtual void start() = 0;

    /*!
     * Retrieves the Status of a task
     * @return Boost shared_ptr of the task status
     */
    virtual iTask::Status::ptr status() = 0;

    /*!
     * The stop command for a task
     */
    virtual void stop() = 0;

protected:
    bool _concurrent = false;   //!< Allowed to run as a seperate thread
    Vessel::ptr _vessel = nullptr; //!< Pointer to the world
};

/*!
 * The Base Task class
 */
class Task : public iTask {
public:
    /*!
     * The contructor
     * @return
     */
    Task(Vessel::ptr vessel, bool concurrent = false);

    /*!
     * The deconstructor
     */
    virtual ~Task();

    /*!
     * The start command for a task
     */
    virtual void start();

    /*!
     * Retrieves the Status of a task
     * @return Boost shared_ptr of the task status
     */
    virtual iTask::Status::ptr status();

    /*!
     * The stop command for a task
     */
    virtual void stop();

protected:
    iTask::Status::ptr _status; //!< a boost share_ptr to the status of a task
    TypeOf _typeof;             //!< Indicating the type of a task
};

/*!
 * An object repsressenting route related tasks
 */
class RouteTask : public Task {
public:
    /*!
```

```
     * Constructor of the interface
     * @return
     */
    RouteTask(Vessel::ptr vessel, bool concurrent = false);

    /*!
     * The deconstructor
     */
    virtual ~RouteTask();

protected:
};

/*!
 * An object representing work related tasks
 */
class WorkTask : public Task {
public:
    /*!
     * Constructor of the interface
     * @return
     */
    WorkTask(Vessel::ptr vessel, bool concurrent = false);

    /*!
     * The deconstructor
     */
    virtual ~WorkTask();

protected:
};

/*!
 * \brief An object representing a coverage path task
 *
 *  All these types of tasks need a robot to cover a complete region in order to
 * perform their tasks. According to \citet{cao_region_1988} such a mobile robot
 * should use the following criteria, for a region filling operation:
 *  1. The mobile robot must move through an entire area, i.e., the overall
 * travel must cover a whole region.
 *  2. The mobile robot must fill the region without overlapping paths.
 *  3. Continuous and sequential operations without any repetition of paths is
 * required of the robot.
 *  4. The robot must avoid all obstacles in a region.
 *  5. Simple motion trajectories (e.g., straight lines or circles) should be
 * used for simplicity in control.
 *  6. An "optimal" path is desired under the available conditions. it's not
 * always possible to satisfy all these criteria for a complex environment.
 * Sometimes a priority consideration is required.
 */
class CoveragePathTask : public RouteTask {
public:
    /*!
    * Constructor of the interface
    * @return
    */
    CoveragePathTask(Vessel::ptr vessel, bool concurrent = false);

    /*!
     * The deconstructor
     */
    virtual ~CoveragePathTask();
```

```
protected:
};

/*!
 * \brief An object representing a follow the target task
 *
 * All these types of tasks need to follow a (moving) target
 */
class FollowTask : public RouteTask {
public:
    /*!
     * Constructor of the interface
     * @return
     */
    FollowTask(Vessel::ptr vessel, bool concurrent = false);

    /*!
     * The deconstructor
     */
    virtual ~FollowTask();

protected:
};

/*!
 * \brief An object representing a normal A to B type of path planning
 *
 * All these types of tasks need to plann an optimum route between A and B,
 * either in time, energy consumption or
 */
class PathTask : public RouteTask {
public:
    /*!
     * Constructor of the interface
     * @return
     */
    PathTask(Vessel::ptr vessel, bool concurrent = false);

    /*!
     * The deconstructor
     */
    virtual ~PathTask();

protected:
};

/*!
 * \brief An Object representing a data logging task
 *
 * All these types of tasks make use of a sensor to record and log
 */
class LogTask : public WorkTask {
public:
    /*!
    * Constructor of the interface
    * @return
    */
    LogTask(Vessel::ptr vessel, bool concurrent = true);

    /*!
     * The deconstructor
```

```
         */
        virtual ~LogTask();

    protected:
    };

    /*!
     * \brief An Object representing a dredging task
     *
     * All these types tasks make use of an actuator and sensors to perform dredging
     * tasks
     */
    class DredgeTask : public WorkTask {
    public:
        /*!
         * Constructor of the interface
         * @return
         */
        DredgeTask(Vessel::ptr vessel, bool concurrent = true);

        /*!
         * The deconstructor
         */
        virtual ~DredgeTask();

    protected:
    };

    class SensorTask : public WorkTask {
    public:
        SensorTask(Vessel::ptr vessel, bool concurrent = true);

        virtual ~SensorTask();

    protected:
    };

    class ActuatorTask : public WorkTask {
    public:
        ActuatorTask(Vessel::ptr vessel, bool concurrent = true);

        virtual ~ActuatorTask();

    protected:

    };

    class CommunicationTask : public WorkTask {
    public:
        CommunicationTask(Vessel::ptr vessel, bool concurrent = true);

        virtual ~CommunicationTask();

    protected:
    };
}
```

LISTING H.19: PT100.CPP

```
//
// Created by peer23peer on 7/20/16.
//
```

```
#include "../../include/Sensors/PT100.h"
#include "../../include/Core/Sensor.h"
#include <iostream>

namespace oCpt {
    namespace components {
        namespace sensors {
            PT100::PT100(iController::ptr controller, World::ptr world, std::string id,
             ↪ uint8_t pinid, uint8_t device)
                    : Sensor(controller, world, id, "PT100"),
                      _pinid(pinid),
                      _device(device) {

            }

            PT100::~PT100() {

            }

            void PT100::updateSensor() {
                _analogeValue = controller_->getAdcVector()->at(_pinid)->getValue();
                ReturnValue_t ret = _constant;
                ret += _dy_dx * static_cast<double>(_analogeValue);
                state_.Value = ret;
                state_.Stamp = world_->now();
            }

            void PT100::setCalibrationTemperature(std::pair<ReturnValue_t, ReturnValue_t>
             ↪ temparature,
                                                  std::pair<uint16_t, uint16_t> analogeValue)
                                                   ↪ {
                _dy_dx = (temparature.second - temparature.first);
                _dy_dx /= static_cast<double>(analogeValue.second - analogeValue.first);
                _constant = temparature.first - _dy_dx;
                _constant *= static_cast<double>(analogeValue.first);
            }

            void PT100::run() {
                updateSensor();
                sig_();
            }

            void PT100::stop() {
                Sensor::stop();
            }

            void PT100::init() {
                Sensor::init();
            }
        }
    }
}
```

LISTING H.20: GPS.CPP

```
#include "../../include/Sensors/Gps.h"

#include <boost/tokenizer.hpp>
#include <boost/foreach.hpp>

#include <string>
```

```
namespace oCpt {
    namespace components {
        namespace sensors {
            using namespace protocol;

            Gps::Gps(iController::ptr controller, World::ptr world, std::string id,
                                                std::string device, unsigned int baudrate)
                    : Sensor(controller, world, id, "GPS"){
                serial_ = Serial::ptr( new Serial(
                        device,
                        baudrate,
                        Serial::io_service_t(new boost::asio::io_service()),
                        Serial::parity_t(Serial::parity_t::none),
                        Serial::character_size_t(8),
                        Serial::flow_control_t(Serial::flow_control_t::none),
                        Serial::stop_bits_t(Serial::stop_bits_t::one), 37));

                serial_->msgRecievedSig.connect(boost::bind(&Gps::interpretMsg, this));
            }

            Gps::~Gps() {
                if (serial_->isOpen()) {
                    serial_->close();
                }
            }

            void Gps::updateSensor() {
                Sensor::updateSensor();
            }

            void Gps::run() {
                if (!sensorRunning_) {
                    Sensor::run();
                    serial_->start();
                }
            }

            void Gps::stop() {
                if (sensorRunning_) {
                    Sensor::stop();
                    if (serial_->isOpen()) {
                        serial_->close();
                    }
                }
            }

            void Gps::setIOservice(boost::shared_ptr <boost::asio::io_service> ioservice) {
                serial_->setIOservice(ioservice);
                Sensor::setIOservice(ioservice);
            }

            void Gps::interpretMsg() {
                std::string msg = serial_->readFiFoMsg();
                boost::char_separator<char> sep(",");
                typedef boost::tokenizer<boost::char_separator<char>> tokenizer_t;
                tokenizer_t tok(msg, sep);
                if ((*tok.begin()).compare("$GPGLL") != 0) return;

                ReturnValue_t ret;
                size_t i = 0;
                auto iBegin = tok.begin();
```

```
                std::advance(iBegin, 1);
                auto iEnd = iBegin;
                for (size_t j = 0;j < 4; j++){
                    std::advance(iEnd, 1);
                    if (iEnd.at_end()) {
                        return;
                    }
                }
                bool correctData = false;
                std::for_each(iBegin, iEnd,[&](auto it){

                    switch (i++) {
                        case 0:
                            ret.latitude.value =
                             ↪ static_cast<World::Location::degree_t>(std::stod(it) *
                             ↪ degree::degree);
                            ret.latitude.value /= 100;
                            break;
                        case 1:
                            ret.latitude.direction = World::Location::stocd(it);
                            break;
                        case 2:
                            ret.longitude.value =
                             ↪ static_cast<World::Location::degree_t>(std::stod(it)  *
                             ↪ degree::degree);
                            ret.longitude.value /= 100;
                            break;
                        case 3:
                            ret.longitude.direction =  World::Location::stocd(it);
                            correctData = true;
                            break;
                    }
                });

                if (correctData) {
                    state_.Stamp = world_->now();
                    state_.Value = ret;
                    sig_();
                }
            }
        }
    }
}
```

LISTING H.21: RAZOR.CPP

```
//
// Created by peer23peer on 10/14/16.
//

#include "../../include/Sensors/Razor.h"
#include "../../include/Core/Boatswain.h"

namespace oCpt {
    namespace components {
        namespace sensors {
            using namespace protocol;

            Razor::Razor(iController::ptr controller, World::ptr world, std::string id,
             ↪ std::string device,
                         unsigned int baudrate, Mode mode, uint8_t freq)
                : Sensor(controller, world, id, device),
```

```
            cb(boost::bind(&Razor::msgHandler, this, _1, _2)),
            mode_(mode),
            freq_(freq) {
        serial_ = Serial::ptr(
                new protocol::Serial(device,
                                     baudrate,
                                     Serial::io_service_t(new
                                      ↪ boost::asio::io_service()),
                                     Serial::parity_t(Serial::parity_t::none),
                                     Serial::character_size_t(8),
                                      ↪ Serial::flow_control_t(Serial::flow_control_t::none),
                                     Serial::stop_bits_t(Serial::stop_bits_t::one),
                                      ↪ 37));

        ReturnValue_t retVal;
        float val[9] = {0};
        fillReturnValue(retVal, val);
        state_.Value = retVal;
        state_.Stamp = world_->now();
    }

    Razor::~Razor() {
        if (serial_->isOpen()) {
            serial_->close();
        }
    }

    void Razor::updateSensor() {
        if (mode_ == Mode::REQ) {
            boost::chrono::milliseconds t =
             ↪ boost::chrono::duration_cast<boost::chrono::milliseconds>(
                    world_->now() - state_.Stamp);
            if (t.count() >
                20) { //The highest allowable update frequency 50 [Hz} don't
                 ↪ introduce unwanted chatter
                Sensor::updateSensor();
                serial_->write("#f");
            }
        }
    }

    void Razor::run() {
        if (!sensorRunning_) {
            Sensor::run();
            serial_->start();
        }
    }

    void Razor::stop() {
        if (sensorRunning_) {
            Sensor::stop();
            if (serial_->isOpen()) {
                serial_->write("#0");   // disable continues mode
                serial_->close();
            }
        }
    }

    void Razor::setIOservice(boost::shared_ptr<boost::asio::io_service> ioservice) {
        serial_->setIOservice(ioservice);
        Sensor::setIOservice(ioservice);
```

```
}

void Razor::init() {
    // TODO implement sensor continues mode
    serial_->open();
    serial_->start();
    //serial_->write("#O");   // Disable continues mode
    //serial_->write("#o1");   // Enable continues mode
    serial_->setReadCallback(cb);
}

void Razor::fillReturnValue(Razor::ReturnValue_t &retVal,
                            float *values) {
    retVal.acc[0] = values[0] * si::meter_per_second_squared;
    retVal.acc[1] = values[1] * si::meter_per_second_squared;
    retVal.acc[2] = values[2] * si::meter_per_second_squared;
    retVal.mag[0] = values[3] * si::tesla;
    retVal.mag[1] = values[4] * si::tesla;
    retVal.mag[2] = values[5] * si::tesla;
    retVal.gyro[0] = values[6] * si::radian_per_second;
    retVal.gyro[1] = values[7] * si::radian_per_second;
    retVal.gyro[2] = values[8] * si::radian_per_second;
}

void Razor::msgHandler(const unsigned char *data, size_t size) {
    if (size != 37) return;
    //std::istringstream str;
    //str.rdbuf()->pubsetbuf(reinterpret_cast<char *>(const_cast<unsigned char
     ↪  *>(data)), size);
    //std::string d = str.str();
    std::vector<char *> d;
    for (size_t i = 0; i < size; i++) {
        d.push_back(reinterpret_cast<char *>(const_cast<unsigned char
         ↪  *>(&data[i])));
    }
    if (checkLRC(d)) {
        ReturnValue_t retVal;
        float *val = new float[9];
        for (int i = 0; i < 9; i++) {
            val[i] = *(reinterpret_cast<const float *>((data + (i * 4))));
        }
        fillReturnValue(retVal, val);
        delete[] val;
        state_.Value = retVal;
        state_.Stamp = world_->now();
        sig_();
    }
}

bool Razor::checkLRC(std::vector<char *> data) {
    char LRC = 0;
    for (size_t i = 0; i < data.size(); i++) {
        LRC ^= *data[i];
    }
    if (LRC != *data[data.size() - 1]) {
        return false;
    }
    return true;
}

Razor::Mode Razor::getMode() const {
    return mode_;
```

```
            }

            void Razor::setMode(Razor::Mode mode) {
                Razor::mode_ = mode;
                if (mode == Mode::CONT) {
                    serial_->write("#1");
                } else {
                    serial_->write("#0");
                }
            }

            uint8_t Razor::getFreq() const {
                return freq_;
            }

            void Razor::setFreq(uint8_t freq) {
                Razor::freq_ = freq;
                serial_->write("#u" + freq);
            }
        }
    }
}
```

LISTING H.22: BEAGLEBONEBLACK.CPP

```
//
// Created by peer23peer on 7/23/16.
//

#include "../../include/Controllers/BeagleboneBlack.h"


namespace oCpt {
    namespace components {
        namespace controller {
            BBB::BBB(World::ptr world)
                    : ARM(world) {
                // Init IIO device 0 port 0..6 load
                for (uint8_t i = 0; i < 7; i++) {
                    protocol::adc::ptr adc_in(new protocol::adc(i, 0, "ti_am335x_adc"));
                    adcVector_.push_back(adc_in);
                }
            }

            BBB::~BBB() {}
        }
    }
}
```

LISTING H.23: MEETCATAMARAN.CPP

```
//
// Created by peer23peer on 7/23/16.
//

#include "../../include/Vessels/Meetcatamaran.h"
#include "../../include/Controllers/BeagleboneBlack.h"
#include "../../include/Sensors/PT100.h"
#include "../../include/Communication/LoRa_RN2483.h"

#include <boost/bind.hpp>
#include <iostream>
```

```
using namespace oCpt::components;

namespace oCpt {
    namespace vessels {
        Meetcatamaran::Meetcatamaran() {
            controller_ = controller::BBB::ptr(new controller::BBB(world_));

            sensors::PT100::ptr tempSensor1(new sensors::PT100(controller_, world_,
             ↪ "tempSensor1", 0, 0));
            tempSensor1->setTimer(boost::posix_time::milliseconds(5000));
            boatswain_->registerSensor(tempSensor1);
            sensors_.push_back(tempSensor1);

            comm::LoRa_RN2483::ptr rn2483(new comm::LoRa_RN2483("rn2483", "/dev/ttyACM0"));
            boatswain_->registerComm(rn2483);
            comm_.push_back(rn2483);

        }

        Meetcatamaran::~Meetcatamaran() {
            boatswain_->stop();
        }

    }
}
```

LISTING H.24: LORA$_{RN2483}$.CPP

```
//
// Created by peer23peer on 8/12/16.
//

#include "../../include/Core/Communication.h"
#include "../../include/Communication/LoRa_RN2483.h"

namespace oCpt {
    namespace components {
        namespace comm {

            LoRa_RN2483::LoRa_RN2483(const std::string &id, const std::string &device,
             ↪ World::ptr world,
                                     iController::io_t ioservice)
                    : LoRa(id, device, world, ioservice) {

            }

            LoRa_RN2483::~LoRa_RN2483() {

            }
        }
    }
}
```

LISTING H.25: SENSOR.CPP

```
//
// Created by peer23peer on 7/15/16.
//

#include "../../include/Core/Sensor.h"
#include "../../include/Core/Boatswain.h"
```

```
namespace oCpt {

    iSensor::iSensor(iController::ptr controller, World::ptr world, std::string id,
     ↪  std::string typeOfSensor)
            : controller_(controller),
              world_(world), id_(id),
              typeOfSensor_(typeOfSensor),
              timer_(0), sensorRunning_(false) {
        state_.Value = static_cast<uint8_t >(0);
    }

    iSensor::~iSensor() {}

    const boost::posix_time::milliseconds &iSensor::getTimer() const {
        return timer_;
    }

    void iSensor::setTimer(const boost::posix_time::milliseconds &timer) {
        iSensor::timer_ = timer;
    }

    iSensor::signal_t &iSensor::getSig()  {
        return sig_;
    }

    const iSensor::State &iSensor::getState() const {
        return state_;
    }

    bool iSensor::operator==(iSensor::ptr rhs) {
        return (id_.compare(rhs->id_) == 0 && typeOfSensor_.compare(rhs->typeOfSensor_) == 0
         ↪  &&
                controller_ == rhs->controller_);
    }

    const std::string &iSensor::getID() const {
        return id_;
    }

    void iSensor::setID(const std::string &id) {
        iSensor::id_ = id;
    }

    const std::string &iSensor::getTypeOfSensor() const {
        return typeOfSensor_;
    }

    void iSensor::setTypeOfSensor(const std::string &typeOfSensor) {
        iSensor::typeOfSensor_ = typeOfSensor;
    }

    Sensor::Sensor(iController::ptr controller, World::ptr world, std::string id, std::string
     ↪  typeOfSensor)
            : iSensor(controller, world, id, typeOfSensor) {

    }

    Sensor::~Sensor() {

    }
```

```
    void Sensor::updateSensor() {

    }

    void Sensor::run() {
        sensorRunning_ = true;
    }

    void Sensor::stop() {
        sensorRunning_ = false;
    }

    void Sensor::init() {

    }

    void Sensor::setIOservice(boost::shared_ptr<boost::asio::io_service> ioservice) {
        ioservice_ = ioservice;
    }
}
```

LISTING H.26: COMMUNICATION.CPP

```
//
// Created by peer23peer on 7/25/16.
//

#include <sstream>

#include "../../include/Core/Communication.h"
#include "../../include/Core/Exception.h"

namespace oCpt {

    iComm::iComm(const std::string &id, const std::string &device, World::ptr world,
     ↪  iController::io_t ioservice)
            : ioservice_(ioservice),
              timer_(0),
              id_(id),
              device_(device),
              world_(world) {

    }

    iComm::~iComm() {

    }

    const std::string &iComm::getId() const {
        return id_;
    }

    void iComm::setId(const std::string &id) {
        iComm::id_ = id;
    }

    const std::string &iComm::getTypeOfComm() const {
        return typeOfComm_;
    }

    void iComm::setTypeOfComm(const std::string &typeOfComm) {
        iComm::typeOfComm_ = typeOfComm;
```

```
}

iComm::Message::ptr iComm::readFiFoMsg() {
    if (!msgQueue_.empty()) {
        Message::ptr retVal = msgQueue_.front();
        msgQueue_.pop_front();
        return retVal;
    }
    return nullptr;
}

void iComm::setIoservice(const iController::io_t &ioservice) {
    iComm::ioservice_ = ioservice;
}

std::deque<iComm::Message::ptr> *iComm::getMsgQueue() {
    return &msgQueue_;
}

LoRa::LoRa(const std::string &id, const std::string &device, World::ptr world,
 ↪  iController::io_t ioservice)
        : iComm(id, device, world, ioservice),
          baudrate_(57600),
          serial_(device_, 57600, ioservice_),
          mod_(LORA),
          freq_(868000000),
          pwr_(14),
          sf_(SF12),
          afcbw_(BW125),
          rxbw_(BW250),
          fdev_(5000),
          prlen_(8),
          crc_(true),
          cr_(CR4_8),
          wdt_(0),
          sync_(12),
          bw_(RBW250),
          sendAllowed_(true),
          proceed_(true),
          listen_(false) {

}

LoRa::~LoRa() {

}

unsigned long LoRa::calculateDownTime(unsigned int payload) {
    double sf = static_cast<double>(sf_);
    double bw = static_cast<double>(bw_);
    double prlen = static_cast<double>(prlen_);
    double T_sym = std::pow(2, sf) / bw;
    double H = crc_ ? 1.0 : 0.0;
    double T_preamble = (prlen + 4.25) * T_sym;
    double cr = static_cast<double>(cr_);
    double ceil = std::ceil((8 * payload - 4 * sf + 28 + 16 - 20 * H) / (4 * sf));
    double payloadSymbNB = 8;
    payloadSymbNB += std::max(ceil * cr + 4, 0.0);
    double T_payloadSymbNb = payloadSymbNB * T_sym;
    double T_packet = T_preamble + T_payloadSymbNb;
    return static_cast<unsigned long>(T_packet);
}
```

```
std::string LoRa::bandWidthToString(const LoRa::BandWidth &value) {
    switch (value) {
        case BW250:
            return "250";
        case BW200:
            return "200";
        case BW166_7:
            return "166.7";
        case BW125:
            return "125";
        case BW100:
            return "10";
        case BW83_3:
            return "83.3";
        case BW62_5:
            return "62.5";
        case BW50:
            return "50";
        case BW41_7:
            return "41.7";
        case BW31_3:
            return "31.3";
        case BW25:
            return "25";
        case BW20_8:
            return "20.8";
        case BW15_6:
            return "15.6";
        case BW12_5:
            return "12.5";
        case BW10_4:
            return "10.4";
        case BW7_8:
            return "7.8";
        case BW6_3:
            return "6.3";
        case BW5_2:
            return "5.2";
        case BW3_9:
            return "3.9";
        case BW3_1:
            return "3.1";
        default:
            return "2.6";
    }
}

std::string LoRa::codingRateToString(const LoRa::CodingRate &value) {
    switch (value) {
        case CR4_5:
            return "4/5";
        case CR4_6:
            return "4/6";
        case CR4_7:
            return "4/7";
        default:
            return "4/8";
    }
}

void LoRa::messageRecieved() {
```

```
        Message::ptr msg(new Message(serial_.readFiFoMsg(), world_->now()));
        if (msg->Payload.compare("invalid_param") == 0 && !ignoreWarn_) {
            throw new oCptException("Invalid parameter send to LoRa device");
        } else if (msg->Payload.compare("ok") == 0) {
            //TODO handle ok
        } else {
            if (msg->Payload.find("radio_rx ") != std::string::npos) {
                hexToString(msg->Payload.substr(10), msg->Payload);
                msgQueue_.push_back(msg);
                msgRecievedSig();
            }
        }

        proceed_ = true;
        if (listen_ && msg->Payload.compare("busy") != 0) {
            rx();
        }
    }

    void LoRa::run() {

    }

    void LoRa::stop() {

    }

    void LoRa::initialize() {
        //Init and open Serial port
//        serial_.setReadCallback(boost::bind(&LoRa::messageRecieved, this));
        serial_.msgRecievedSig.connect(boost::bind(&LoRa::messageRecieved, this));
        serial_.open();
        serial_.start();

        //clear UART write buffer
        ignoreWarn_ = true;
        write("\r\n");
        ignoreWarn_ = false;

        //Write radio protocol to LoRa chip
        write(buildRadioCmdString<ModulationMode>(MOD, mod_));
        write(buildRadioCmdString<unsigned long>(FREQ, freq_));
        write(buildRadioCmdString<int8_t>(PWR, pwr_));
        write(buildRadioCmdString<SpreadingFactor>(SF, sf_));
        write(buildRadioCmdString<BandWidth>(AFCBW, afcbw_));
        write(buildRadioCmdString<BandWidth>(RXBW, rxbw_));
        write(buildRadioCmdString<uint>(FDEV, fdev_));
        write(buildRadioCmdString<uint>(PRLEN, prlen_));
        write(buildRadioCmdString<bool>(CRC, crc_));
        write(buildRadioCmdString<CodingRate>(CR, cr_));
        write(buildRadioCmdString<unsigned long>(WDT, wdt_));
        write(buildRadioCmdString<unsigned long>(SYNC, sync_));
        write(buildRadioCmdString<RadioBandWidth>(BW, bw_));

        //Put device in listen mode
        macpause();
        listen_ = true;
        rx();
    }

    void LoRa::sendMessage(iComm::Message msg) {
        if (msg.Stamp < world_->now() && sendAllowed_) {
```

```
            //Send the message
            serial_.write(buildMacCmdString<int>(PAUSE));
            serial_.write(buildRadioCmdString(rTX, msg.Payload, NONE));

            //TODO set downtime timer to comply with government send times

            // Put device in listen mode
            rx();
        } else {
            //TODO implement timer released send
        }
    }

    iComm::Message::ptr LoRa::recieveMessage() {
        return nullptr;
    }

    void LoRa::recieveAsyncMessage() {

    }

    void LoRa::stringToHex(const std::string str, std::string &hexStr, const bool capital) {
        hexStr.resize(str.size() * 2);
        static const char a = capital ? 0x40 : 0x60;

        for (size_t i = 0; i < str.size(); i++)
        {
            char c = (str[i] >> 4) & 0xF;
            hexStr[i * 2] = c > 9 ? (c - 9) | a : c | '0';
            hexStr[i * 2 + 1] = (str[i] & 0xF) > 9 ? (str[i] - 9) & 0xF | a : str[i] & 0xF |
             ↪  '0';
        }
//        std::stringstream ss;
//        for (auto c : value) {
//            ss << std::hex << (int)c;
//        }
//        return ss.str();
    }

    void LoRa::write(const std::string &value) {
        proceed_ = false;
        serial_.write(value);
        while (!proceed_) {
            usleep(1000);
        }
    }

    void LoRa::rx() {
        //Put device in listen mode
        serial_.write(buildRadioCmdString<int>(rRX, 0, NONE));
    }

    void LoRa::macpause() {
        serial_.write(buildMacCmdString<int>(PAUSE)); //will pause for ~49 days
        //TODO implement a timer if needed?
    }

    void LoRa::hexToString(const std::string hexStr, std::string &str) {
        str.resize((hexStr.size() + 1) / 2);

        for (size_t i = 0, j = 0; i < str.size(); i++, j++)
        {
```

```
            str[i] = (hexStr[j] & '@' ? hexStr[j] + 9 : hexStr[j]) « 4, j++;
            str[i] |= (hexStr[j] & '@' ? hexStr[j] + 9 : hexStr[j]) & 0xF;
        }
    }
}
```

LISTING H.27: TASK.CPP

```
//
// Created by peer23peer on 7/9/16.
//

#include "../../include/Core/Task.h"

namespace oCpt {
    iTask::iTask(iVessel::ptr vessel, bool concurrent) {
        _concurrent = concurrent;
        _vessel = vessel;
    }

    iTask::~iTask() {}

    iTask::Status::Status() {}

    iTask::Status::~Status() {}

    double iTask::Status::progress() { return _progress; }

    bool iTask::Status::running() { return _running; }

    bool iTask::Status::successful() { return _successful; }

    Task::Task(Vessel::ptr vessel, bool concurrent) : iTask(vessel, concurrent) {
        _status = iTask::Status::ptr(new iTask::Status());
    }

    Task::~Task() {}

    void Task::start() {
        std::for_each(Work.begin(), Work.end(), [&](iTask::ptr &T) {

        });
    }

    iTask::Status::ptr Task::status() { return _status; }

    void Task::stop() {}

    RouteTask::RouteTask(Vessel::ptr vessel, bool concurrent) : Task(vessel, concurrent) {
        _typeof = iTask::TypeOf::ROUTE;
    }

    RouteTask::~RouteTask() {}

    WorkTask::WorkTask(Vessel::ptr vessel, bool concurrent) : Task(vessel, concurrent) {
        _typeof = iTask::TypeOf::WORK;
    }

    WorkTask::~WorkTask() {}

    CoveragePathTask::CoveragePathTask(Vessel::ptr vessel, bool concurrent) :
     ↪  RouteTask(vessel, concurrent) {}
```

```
    CoveragePathTask::~CoveragePathTask() {}

    FollowTask::FollowTask(Vessel::ptr vessel, bool concurrent) : RouteTask(vessel,
     ↪ concurrent) {}

    FollowTask::~FollowTask() {}

    PathTask::PathTask(Vessel::ptr vessel, bool concurrent) : RouteTask(vessel, concurrent)
     ↪ {}

    PathTask::~PathTask() {}

    LogTask::LogTask(Vessel::ptr vessel, bool concurrent) : WorkTask(vessel, concurrent) {}

    LogTask::~LogTask() {}

    DredgeTask::DredgeTask(Vessel::ptr vessel, bool concurrent) : WorkTask(vessel,
     ↪ concurrent) {}

    DredgeTask::~DredgeTask() {}

    SensorTask::SensorTask(Vessel::ptr vessel, bool concurrent) : WorkTask(vessel,
     ↪ concurrent) {}

    SensorTask::~SensorTask() {}

    ActuatorTask::ActuatorTask(Vessel::ptr vessel, bool concurrent) : WorkTask(vessel,
     ↪ concurrent) {}

    ActuatorTask::~ActuatorTask() {}

    CommunicationTask::CommunicationTask(Vessel::ptr vessel, bool concurrent) :
     ↪ WorkTask(vessel, concurrent) {}

    CommunicationTask::~CommunicationTask() {}
}
```

LISTING H.28: ACTUATOR.CPP

```
//
// Created by peer23peer on 7/23/16.
//

#include "../../include/Core/Actuator.h"

namespace oCpt {

    iActuator::iActuator() {

    }

    iActuator::~iActuator() {

    }

    Actuator::Actuator() : iActuator() {

    }

    Actuator::~Actuator() {
```

```
    }

    void Actuator::setActuator() {

    }

    void Actuator::run() {

    }

    void Actuator::stop() {

    }
}
```

LISTING H.29: WORLD.CPP

```
//
// Created by peer23peer on 7/15/16.
//

#include "../../include/Core/World.h"
#include <algorithm>
#include <string>

namespace oCpt {

    World::Time::Time() {}

    World::Time::~Time() {}

    World::Time::clock_t &oCpt::World::Time::getTimeClock() {
        return timeClock_;
    }

    World::Time::timepoint_t oCpt::World::Time::now() {
        return timeClock_.now();
    }

    World::World() {

    }

    World::~World() {}

    World::Time &World::getTime() {
        return time_;
    }

    World::Time::timepoint_t World::now() {
        return time_.now();
    }

    World::Location::Location() {

    }

    World::Location::~Location() {

    }
```

```
    World::Location::RoutePoint::ptr World::Location::getCurrentLocation(bool newMeasurement)
     ↪ {
        return oCpt::World::Location::RoutePoint::ptr(); //TODO implement obtain current
         ↪ Location
    }

    void World::Location::push_back(World::Location::RoutePoint::ptr routePoint) {
        //TODO implement way point log
    }

    std::vector<World::Location::RoutePoint::ptr> World::Location::getLocationHistory() {
        return std::vector<World::Location::RoutePoint::ptr>();
    }

    World::Location::cardinal_direction World::Location::stocd(std::string str) {
        char s;

        /*
         * Get the first character to be and make it lower case
         */
        if (str.at(0) <= 'Z' && str.at(0) >= 'A') {
            s = str.at(0) - ('Z' - 'z');
        } else {
            s = str.at(0);
        }

        return static_cast<cardinal_direction >(s);
    }

    std::string World::Location::gpsPoint::toString() {
        std::ostringstream strs;
        strs « latitude.value « static_cast<char>(latitude.direction - 32) « ", " «
         ↪ longitude.value « static_cast<char>(longitude.direction - 32);
        return strs.str();
    }
}
```

LISTING H.30: CONTROLLER.CPP

```
//
// Created by peer23peer on 7/15/16.
//

#include "../../include/Core/Controller.h"
#include "../../include/Core/Exception.h"

#include <thread>

#include <boost/make_shared.hpp>

namespace oCpt {

    namespace protocol {

        userspace::userspace() {}

        userspace::~userspace() {}

        bool userspace::modLoaded(std::string modName) {
            //TODO check if Mutex is needed, probably

            // if the module name to be checked is empty return true
```

```
    if (modName.compare("") == 0) { return true; }
    std::ifstream fs(MODULE_PATH); //open the module path
    std::string line;
    // Read the lines and check if the module name is in one of them. If this is the
     ↪ case, close the file and return true
    while (std::getline(fs, line)) {
        if (line.find(modName, 0) != std::string::npos) {
            fs.close();
            return true;
        }
    }
    // The complete file has been run through without a hit, close the file and
     ↪ return false
    fs.close();
    return false;
}

bool userspace::fileExist(std::string fileName) {
    // Check the file stat of Linux
    struct stat buffer;
    return (stat(fileName.c_str(), &buffer) == 0);
}

bool userspace::dtboLoaded(std::string dtboName) {
    //Open the cape manager path
    std::ifstream fs(BBB_CAPE_MNGR);
    std::string line;
    // check if the device tree overlay name is present. If that is the case close
     ↪ the file and return true
    while (std::getline(fs, line)) {
        if (line.find(dtboName, 0) != std::string::npos) {
            fs.close();
            return true;
        }
    }
    // The complete file has been checked and no hit, close it and return false
    fs.close();
    return false;
}

adc::adc(uint8_t id, uint8_t device, std::string modName) {
    device_ = device;
    id_ = id;
    //construct the complete path, from the device and pin ID
    std::stringstream ss;
    ss << ADC_IO_BASE_PATH << std::to_string(device_) << ADC_VOLTAGE_PATH <<
     ↪ std::to_string(id_)
       << ADC_VOLTAGE_SUB_PATH;
    path_ = ss.str();
    // Check if necessary module is loaded
    if (!modLoaded(modName)) {
        throw oCptException("Exception! Module not loaded", 0);
    }
    // Check if the path is correct
    if (!fileExist(path_)) {
        throw oCptException("Exception! Userspacefile doesn't exist", 1);
    }
}

adc::~adc() {}

uint16_t &adc::getValue() {
```

```
    //TODO check if Mutex is needed for a read operation
    //Open the file, read the value and close if
    std::ifstream fs;
    fs.open(path_.c_str());
    fs >> value_;
    fs.close();
    return value_;
}

bool adc::operator==(const adc &rhs) {
    return (rhs.path_.compare(path_) == 0);
}

bool adc::compare(const uint8_t &id, const uint8_t &device) {
    return (id_ == id && device_ == device);
}

Serial::Serial(const std::string &device, unsigned int baudrate, io_service_t
 ↪  ioservice,
               Serial::parity_t parity,
               Serial::character_size_t csize,
               Serial::flow_control_t flow,
               Serial::stop_bits_t stop,
               unsigned int maxreadlentgh)
        : device_(device),
          baudrate_(baudrate),
          ioservice_(ioservice),
          parity_(parity),
          csize_(csize),
          flow_(flow),
          stop_(stop),
          serialport_(*ioservice.get(), device_),
          receivedMsg_(""),
          maxReadLength_(maxreadlentgh) {
    callback_ = boost::bind(&Serial::internalCallback, this, _1, _2);
}

void Serial::open() {
    //If the port is closed, try opening it
    if (!isOpen()) {
        try {
            serialport_ = serialport_t(*ioservice_.get(), device_);
        } catch (const std::exception &e) {
            std::cerr << "Unable to open device: " << device_ << std::endl;
            throw;
            //TODO better error handling
        }
    }

    //Set the settings for the specified port
    serialport_.set_option(boost::asio::serial_port_base::baud_rate(baudrate_));
    serialport_.set_option(parity_);
    serialport_.set_option(csize_);
    serialport_.set_option(flow_);
    serialport_.set_option(stop_);
}

bool Serial::isOpen() {
    return serialport_.is_open();
}

void Serial::close() {
```

```
        //Close the port if it's open and set the callback to the internal closeCallback
         ↪ if needed
        if (serialport_.is_open()) {
            ioservice_->post(boost::bind(&Serial::closeCallback,
                                         this,
                                         boost::system::error_code()));
        }
    }

    void Serial::setReadCallback(cb_func cb_function) {
        callback_ = cb_function;
    }

    void Serial::setIOservice(boost::shared_ptr<boost::asio::io_service> io_ptr) {
        ioservice_ = io_ptr;
    }

    void Serial::closeCallback(const boost::system::error_code &error) {
        if (error && (error != boost::asio::error::operation_aborted)) {
            std::cerr << "Error: " << error.message() << std::endl;
        }
        serialport_.close();
    }

    void Serial::start() {
        if (!isOpen()) {
            throw std::runtime_error("Serial port interface not open");
        }
        ReadStart();

        std::thread io_thread(boost::bind(&boost::asio::io_service::run, ioservice_));
        io_thread.detach();
//          ioservice_->run();
        //TODO check if run io_service should be called from here? threaded? or from
         ↪ boatswain
    }

    void Serial::ReadStart() {
        if (isOpen()) {
            serialport_.async_read_some(boost::asio::buffer(read_msg, maxReadLength_),
                                        boost::bind(&Serial::readComplete,
                                                    this,
                                                    boost::asio::placeholders::error,

                                                     ↪ boost::asio::placeholders::bytes_transferred));
        }
    }

    void Serial::readComplete(const boost::system::error_code &error, size_t
     ↪ bytes_transferred) {
        if (!error) {
            callback_(const_cast<unsigned char *>(read_msg), bytes_transferred);
            ReadStart();
        } else {
            closeCallback(error);
        }
    }

    std::deque<std::string> *Serial::getReturnMsgQueue() {
        return &returnMsgQueue_;
    }
```

```
std::string Serial::readFiFoMsg() {
    std::string msg = returnMsgQueue_.front();
    returnMsgQueue_.pop_front();
    //TODO check if it's necceasry to resend the msg receive signal if there are
     ↪ still msgs in the que
    return msg;
}

void Serial::internalCallback(const unsigned char *data, size_t size) {
    //Create an stringstream and set the pointer for its buffer towards the obtained
     ↪ char buffer
    std::istringstream str;
    str.rdbuf()->pubsetbuf(reinterpret_cast<char *>(const_cast<unsigned char
     ↪ *>(data)), size);
    //Fill the last obtained strings if needed
    if (!receivedMsg_.empty()) {
        std::string appMsg = "";
        std::getline(str, appMsg);
        receivedMsg_.append(appMsg);
        if (receivedMsg_.back() == 13) {
            receivedMsg_.pop_back();
            returnMsgQueue_.push_back(receivedMsg_);
            receivedMsg_ = "";
            msgRecievedSig();
        } else {
            return;
        }
    }

    while (std::getline(str, receivedMsg_)) {
        if (receivedMsg_.back() == 13) {
            receivedMsg_.pop_back();
            returnMsgQueue_.push_back(receivedMsg_);
            receivedMsg_ = "";
            msgRecievedSig();
        }
    }

}

bool Serial::write(const std::string &msg) {
    //convert the msg string to a vector of chars
    std::vector<unsigned char> msg_vec(msg.begin(), msg.end());
    return write(msg_vec);
}

bool Serial::write(const std::vector<unsigned char> &data) {
    if (!isOpen()) {
        return false;
    }
    /*TODO check if this is executed in the same thread
     *
↪  (http://www.boost.org/doc/libs/1_42_0/doc/html/boost_asio/reference/io_service/post.html)
     * The io_service::post guarantees that the handler will only be called in a
↪  thread in which the run(),
     * run_one(), poll() or poll_one() member functions is currently being invoked.*/
    ioservice_->post(boost::bind(&Serial::writeCallback, this, data));
    return true;
}

void Serial::writeCallback(const std::vector<unsigned char> &msg) {
```

```
    bool write_inProgress = !msgQueue_.empty(); //writing in progress when the que
     ↪ isn't empty
    msgQueue_.push_back(msg); // push the message to the back of the que
    if (!write_inProgress) {
        writeStart(); // Send the first message
    }
}

void Serial::writeStart() {
    //Write the characters asynchronous and bind the writeComplete function
    boost::asio::async_write(serialport_,
                             boost::asio::buffer(&msgQueue_.front()[0],
                                                 msgQueue_.front().size()),
                             boost::bind(&Serial::writeComplete,
                                         this,
                                         boost::asio::placeholders::error));

}

void Serial::writeComplete(const boost::system::error_code &error) {
    // If writing of a message was oke, remove the message from the write que and
     ↪ perform the next write from the que
    if (!error) {
        msgQueue_.pop_front();
        if (!msgQueue_.empty()) {
            writeStart();
        }
    } else {
        closeCallback(error);
    }
}

void Serial::setMaxReadLength(unsigned int maxReadLength) {
    Serial::maxReadLength_ = maxReadLength;
}

int gpio::getPinNumber() const {
    return pinNumber_;
}

void gpio::setPinNumber(int pinNumber) {
    //TODO change path, check if path is correct, set the Edge, Direction etc from
     ↪ the new pin number
    gpio::pinNumber_ = pinNumber;
}

gpio::Value gpio::getValue() const {
    if (direction_ == Direction::INPUT) {
        return readPinValue<Value>(pinNumber_);
    }
    return value_;
}

void gpio::setValue(gpio::Value value) {
    if (direction_ == Direction::OUTPUT) {
        writePinValue<Value>(gpiopath_, value);
    }
    gpio::value_ = value;
}

gpio::Direction gpio::getDirection() const {
    return direction_;
```

```
}

void gpio::setDirection(gpio::Direction direction) {
    writePinValue<Direction>(gpiopath_, direction_);
    gpio::direction_ = direction;
}

gpio::Edge gpio::getEdge() const {
    if (direction_ == Direction::INPUT) {
        return readPinValue<Edge>(gpiopath_);
    }
    return edge_;
}

void gpio::setEdge(gpio::Edge edge) {
    if (direction_ == Direction::OUTPUT) {
        writePinValue<Edge>(gpiopath_, edge);
    }
    gpio::edge_ = edge;
}

gpio::gpio(int pinNumber, gpio::Direction direction, gpio::Value value, gpio::Edge
 ↪  edge)
        : pinNumber_(pinNumber),
          direction_(direction),
          value_(value),
          edge_(edge),
          threadRunning_(false) {
    gpiopath_ = GPIO_BASE_PATH;
    gpiopath_.append("gpio" + std::to_string(pinNumber_));
    exportPin(pinNumber_);
    writePinValue<Direction>(gpiopath_, direction_);
    writePinValue<Edge>(gpiopath_, edge_);
    writePinValue<Value>(gpiopath_, value_);
    cb_ = gpio::cb_func(boost::bind(&gpio::internalCbFunc, this));
    //= boost::bind(&gpio::internalCbFunc);
}

gpio::~gpio() {
    unexportPin(pinNumber_);
}

std::vector<gpio::ptr> gpio::exportedGpios() {
    std::vector<gpio::ptr> gpios;
    boost::filesystem::path p(GPIO_BASE_PATH);
    const unsigned int sizeOfBasePath = p.string().size();

    //! Iterate through all exported pins
    boost::filesystem::directory_iterator end_itr;
    for (boost::filesystem::directory_iterator itr(p); itr != end_itr; ++itr) {
        std::string path = itr->path().string();
        if ((path.find("gpio", sizeOfBasePath) != std::string::npos) &&
            (path.find("chip", sizeOfBasePath) == std::string::npos)) {
            const unsigned int loc = path.find("gpio", sizeOfBasePath) + 4;
            const std::string nr = path.substr(loc);
            int pinnumber = std::atoi(nr.c_str());
            Direction direction = readPinValue<Direction>(pinnumber);
            Value value = readPinValue<Value>(pinnumber);
            Edge edge = readPinValue<Edge>(pinnumber);
            gpio::ptr gpio_ptr(new gpio(pinnumber, direction, value, edge));
            gpios.push_back(gpio_ptr);
        }
```

```
    }
    return gpios;
}

void gpio::exportPin(const int &number) {
    const std::string exportPath = std::string(GPIO_BASE_PATH) + "export";
    std::ofstream fs;
    try {
        fs.open(exportPath);
    } catch (std::ofstream::failure const &ex) {
        //TODO error handling
        return;
    }
    fs << std::to_string(number) << std::endl;
    fs.close();
    usleep(250000);
}

void gpio::unexportPin(const int &number) {
    const std::string exportPath = std::string(GPIO_BASE_PATH) + "unexport";
    std::ofstream fs;
    try {
        fs.open(exportPath);
    } catch (std::ofstream::failure const &ex) {
        //TODO error handling
        return;
    }
    fs << std::to_string(number) << std::endl;
    fs.close();
    usleep(250000);
}

void gpio::toggle() {
    writePinValue<Value>(gpiopath_, static_cast<Value>(value_ ^ 1));
    //TODO write optimized function currently around 7kHz
}

void gpio::setCallbackFunction(gpio::cb_func cb) {
    cb_ = cb;
}

void gpio::internalCbFunc() {
    signalChanged();
}

void gpio::waitForEdge() {
    if (direction_ == Direction::OUTPUT) {
        return;
    }
    int fd, i, epollfd, count = 0;
    struct epoll_event ev;
    epollfd = epoll_create(1);
    if (epollfd == -1) {
        //TODO error handling
    }
    std::string path = gpiopath_;
    path.append("/value");
    if ((fd = open(path.c_str(), O_RDONLY | O_NONBLOCK)) == -1) {
        //TODO error handling
    }
    ev.events = EPOLLIN | EPOLLET | EPOLLPRI;
    ev.data.fd = fd;
```

```
            if (epoll_ctl(epollfd, EPOLL_CTL_ADD, fd, &ev) == -1) {
                //TODO errror handling
            }

            while (count <= 1) {
                i = epoll_wait(epollfd, &ev, 1, -1);
                if (i == -1) {
                    count = 5;
                    //TODO error handling
                } else {
                    count++;
                }
            }
            cb_();
            close(fd);
        }

        void gpio::waitForEdgeAsync() {
            std::thread poll_thread(boost::bind(&gpio::waitForEdge, this));
            poll_thread.detach();
        }

    }

    iController::iController(World::ptr world)
            : world_(world) {}

    iController::~iController() {}

    ARM::ARM(World::ptr world)
            : iController(world) {}

    ARM::~ARM() {}

    protocol::adc::ptr ARM::getADC(uint8_t id, uint8_t device) {
        std::for_each(adcVector_.begin(), adcVector_.end(), [&](protocol::adc::ptr &A) {
            if (A->compare(id, device)) { return A; }
        });
        return protocol::adc::ptr();
    }

    std::vector<protocol::adc::ptr> *ARM::getAdcVector() {
        return &adcVector_;
    }
}
```

LISTING H.31: BOATSWAIN.CPP

```
//
// Created by peer23peer on 7/21/16.
//
#include "../../include/Core/Boatswain.h"

namespace oCpt {
    iBoatswain::iBoatswain(iController::ptr controller)
            : controller_(controller) {
        localStopThread_ = boost::shared_ptr<bool>(new bool {false});
        ioservice_ = boost::shared_ptr<boost::asio::io_service>(new
         ↪ boost::asio::io_service());
    }
```

```
iBoatswain::~iBoatswain() {

}

const boost::shared_ptr<bool> &iBoatswain::getStopThread() const {
    return stopThread_;
}

void iBoatswain::setStopThread(const boost::shared_ptr<bool> &stopThread) {
    iBoatswain::stopThread_ = stopThread;
}

boost::shared_ptr<boost::asio::io_service> &iBoatswain::getIOservice() {
    return ioservice_;
}

Boatswain::Boatswain(iController::ptr controller) : iBoatswain(controller) {

}

Boatswain::~Boatswain() {

}

void Boatswain::run() {
    std::for_each(manualSensors_.begin(), manualSensors_.end(), [](iSensor::ptr &P) {
        P->run();
    });
    ioservice_->run();
}

void Boatswain::stop() {
    *localStopThread_ = true;
}

void Boatswain::initialize() {

}

void Boatswain::registerSensor(iSensor::ptr sensor) {
    /*!
     * If the timer is set for the sensor, create a new timer service, register the
↪ sensor with the timer sensors, and set the callback functions to execute the Sensor::run
↪ function and the internall resetTimer function.
     * If the timer is not set register the Sonsor with the manual sensor.
     */
    if (sensor->getTimer().total_microseconds() > 0) {
        timerPtr timer(new boost::asio::deadline_timer(*ioservice_.get(),
          ↪  sensor->getTimer()));
        timer->async_wait(boost::bind(&iSensor::run, sensor));
        timer->async_wait(boost::bind(&Boatswain::resetTimer, this, sensor));
        timers_.push_back(timer);
        timerSensors_.push_back(sensor);
    } else {
        sensor->setIOservice(ioservice_);
        manualSensors_.push_back(sensor);
        //TODO implement sensors update polling
        //TODO implement sensor update manual request
    }
}

void Boatswain::registerActuator(iActuator::ptr actuator) {
```

```
        //TODO implement actuator shit
    }

    void Boatswain::resetTimer(iSensor::ptr sensor) {
        /*!
         * Don't excute if the thread is stopped
         */
        if (*stopThread_ || *localStopThread_) {
            return;
        }
        /*!
         * Find the current index of the sensor, this could maybe optimized by using a
 ↪  mappping list
         */
        int i = -1;
        std::find_if(timerSensors_.begin(), timerSensors_.end(), [&](iSensor::ptr &S) {
            i++;
            return S == sensor;
        });

        /*!
         * Set the new timer. drift isn't taken into account at the current time.
         */
        //TODO eliminate drift
        timers_[i] = timerPtr(new boost::asio::deadline_timer(*ioservice_.get(),
         ↪  sensor->getTimer()));
        timers_[i]->async_wait(boost::bind(&iSensor::run, sensor));
        timers_[i]->async_wait(boost::bind(&Boatswain::resetTimer, this, sensor));
    }

    void Boatswain::registerComm(iComm::ptr comm) {
        comm->setIoservice(ioservice_);
    }

}
```

LISTING H.32: VESSEL.CPP

```
//
// Created by peer23peer on 7/16/16.
//

#include "../../include/Core/Vessel.h"

namespace oCpt {

    iVessel::iVessel() { }

    iVessel::iVessel(iController::ptr controller) {}

    iVessel::~iVessel() {}

    const boost::shared_ptr<bool> &iVessel::getStopThread() const {
        return stopThread_;
    }

    void iVessel::setStopThread(const boost::shared_ptr<bool> &stopThread) {
        iVessel::stopThread_ = stopThread;
    }

    Vessel::Vessel() {
        world_ = World::ptr(new World());
```

```
        captain_ = Captain::ptr(new Captain(world_));
        boatswain_ = Boatswain::ptr(new Boatswain(controller_));

        stopThread_ = boost::shared_ptr<bool>( new bool{false});
        captain_->setStopThread_(stopThread_);
        boatswain_->setStopThread(stopThread_);
    }

    Vessel::Vessel(iController::ptr controller)
            : controller_(controller) {
        world_ = World::ptr(new World());
        captain_ = Captain::ptr(new Captain(world_));
        boatswain_ = Boatswain::ptr(new Boatswain(controller_));
    }

    Vessel::~Vessel() {}

    void Vessel::initialize() {
        boatswain_->initialize();
        captain_->initialize();
    }

    void Vessel::run() {
        std::thread bs_thread(boost::bind(&iBoatswain::run, boatswain_)); //TODO write thread
         ↪  wrapper
        captain_->run();
        bs_thread.join();
        //TODO make a work queue
    }

    void Vessel::stop() {
        *stopThread_ = true;
    }

}
```

LISTING H.33: CAPTAIN.CPP

```
//
// Created by peer23peer on 7/23/16.
//

#include "../../include/Core/Captain.h"

namespace oCpt {

    Captain::Captain(World::ptr world) : iCaptain(world) {
        localStopThread_ = boost::shared_ptr<bool>(new bool{false});
    }

    Captain::~Captain() {

    }

    void Captain::run() {
        while (!*stopThread_ || !*localStopThread_) {
            sleep(1000);
        }
    }

    void Captain::stop() {
        *localStopThread_ = true;
```

```
    }

    void Captain::initialize() {

    }

    iCaptain::iCaptain(World::ptr world) {

    }

    iCaptain::~iCaptain() {

    }

    const boost::shared_ptr<bool> &iCaptain::getStopThread_() const {
        return stopThread_;
    }

    void iCaptain::setStopThread_(const boost::shared_ptr<bool> &stopThread_) {
        iCaptain::stopThread_ = stopThread_;
    }
}
```