Master's thesis

2021

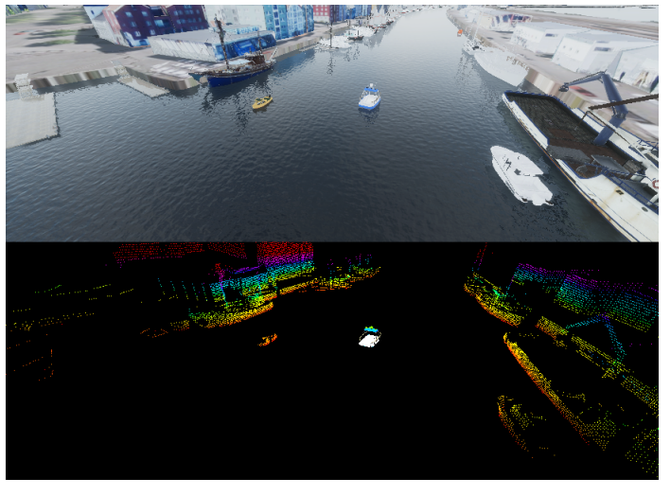
Kjetil Vasstein

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Kjetil Vasstein

A high fidelity digital twin framework for testing exteroceptive perception of autonomous vessels

February 2021



NTNU
Norwegian University of Science and Technology

NTNU
Norwegian University of Science and Technology

NTNU

Norwegian University of
Science and Technology

A high fidelity digital twin framework for testing exteroceptive perception of autonomous vessels

Kjetil Vasstein

Cybernetics and Robotics
Submission date: February 2021
Supervisor: Edmund Førland Brekke
Co-supervisor: Rudolf Mester
Øyvind Smogeli
Tor Arne Johansen

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Abstract

In the future, it is believed that verification of autonomous ships must involve simulation based verification, due to the huge costs involved in real life testings. In contrary to the car industry, where open source digital twin frameworks exists, there is no equivalent solution for the maritime industry. There are also few studies that shows if simulations can be trusted, moreover how they affect algorithms used by autonomous agents such as target trackers. This thesis implements a digital twin framework, capable of simulating camera and lidar data using the Unity game engine, in order to test a simulation verification method, where a real and synthetic dataset with common ground truths are compared using a Hellinger distance on the outputs of a target tracker. The results demonstrate a metric that is capable of measuring fidelity, establishing a quantitative method that can be used for future improvements towards trustworthy simulation based verification for autonomous marine vessels.

Sammendrag

I fremtiden antas det at verifisering av autonome skip må gjøres ved simuleringsbasert verifisering på grunn av de enorme kostnadene som er involvert med ekte testing. I motsetning til bilindustrien, hvor det finnes åpent tilgjengelige rammeverk for digitale tvillinger, finnes det ingen tilsvarende løsning for den maritime industrien. Det er også få studier som viser om simuleringer kan stoles på, og hvordan de påvirker algoritmer som brukes av autonome agenter slik som målsporere. Denne oppgaven implementerer et digitalt tvilling rammeverk, i stand til å simulere kamera og lidar data med bruk av Unity spillmotoren, for å teste ut en valideringsmetode for simuleringer, med å sammenligne ett ekte og syntetisk datasett ved hjelp av en Hellinger distanse fra utgangene til en målsporer. Resultatene viser en metrikk i stand til å måle realismen til simuleringen, og som kan brukes for fremtidige forbedringer mot en pålitelig simuleringsbasert verifisering for autonome marinefartøy.

Acknowledgements

The thesis have relied on multiple contributions from different fields across NTNU and the industry. Without specialised knowledge across multiple disciplines, this thesis would have never existed. Early on, it became obvious to me that in order to complete this task, I had to create a project of mutual interest that tightly involved 3 PhD students, 3 master students, and 1 industry expert. Appendix A and B is made for the sole purpose to acknowledge each participant's work in this project.

Nevertheless, a special thanks goes to Øyvind Kaarstad Helgesen for being an important partner throughout the project period. Thanks for all the patience and help during the last stages of the project, I owe you alot. The same goes to Thomas Skarshaug, the fellow co-founder of Autoferry Gemini, which I must especially thank for having two jobs during this period, one being his job at Zeabuz, the other writing code alongside me in the evenings. Without you, there would never have been any infrastructure capable of distributing the big flow of sensor data from the simulator.

A thank is also needed to Erik Veitch and Thomas Kaland at the Insitute for Design, for doing the tedious work of 3D modeling the target boats used in this thesis. In addition, I want to thank Tobias Rye Torben for his effort in the open simulation platform. Because of you, we can now demonstrate Gemini's relevance for the maritime industry more than ever.

I also want to address the participants in the experiments with the autonomous ferry milliAmpere. The thanks goes to Ingunn Kjønås, Magne Sirnes and Michael Ernesto Lopez for helping me out gathering enough sensor data to write my thesis.

A last thanks goes to my supervisors Edmund Førland Brekke, Rudolf Mester, Tor Arne Johansen and Øyvind Smogeli for guiding me despite of the stressful times during the corona virus. I hope we can continue our work together!

Symbols and Conventions

Conventions

B_{type}	Total data size of type
Z^{d}_{type}	Depth buffer
ϵ^{space}_{type}	Error function
HFOV_{type}	Horisontal field of view
N_{types}	Resolution, e.i number of elements
VFOV_{type}	Vertical field of view
$\hat{\_}$	Hat notation for estimate of variable _
$\mathrm{Q}^{space}_{type}$	Vector and point notation
x^{space}_{type}	First vector element
y^{space}_{type}	Second vector element
z^{space}_{type}	Third vector element
w^{i}_{type}	Homogeneous normalization element in image space projection
$\mathbf{T}^{to\ space}_{from\ space}$	Transformation matrices

Spaces

d	Discrete image space
i	Image space
l	Longitude, latitude, height coordinate space
n	North, East, Down coordinate space
o	Object space
v	View space
w	World space
	Variable in not set in any space

Types

d	Depth

h	Height
k	Time increment
w	Width
c	Camera, e.i based on the Cartesian coordinate system
s	Point cloud, e.i based on the Spherical coordinate system
r	Reference model, e.i ground truth
sb	Ship's bow
ss	Ship's stern
gnss	Ship's stern
number	Index of instance, e.g the initial time t_0
g	Gaussian
li	Lidar
	No type indicates the Generic type, e.i applies in general

Number Sets

$\mathbb{M}$	Natural numbers filtered by the spherical projection filter
$\mathbb{N}$	Natural numbers
$\mathbb{R}$	Real numbers

Physical Constants

e	Eccentricity of the WGS-84 ellipsoid
r_e	Equatorial radius of WGS-84 ellipsoid
R_M	Meridian radius of WGS-84 ellipsoid
R_N	Prime vertical of WGS-84 ellipsoid
v_c	Speed of light in a vacuum

Synthetic camera sensor

a	Aspect ratio of a synthetic cameras image
$\mathbf{N}_\mathrm{c}$	Camera resolution matrix
f	Distance to synthetic cameras far plane
F_{prod}	Dot product between mesh vertices and frustum normals
F_{normal}	Frustum normals
n	Distance to synthetic cameras near plane

Synthetic point cloud sensor

$\mathbf{R}_y(\theta)$	Rotation matrix around y-axis by θ degrees or radians

θ	Spherical parameterization variable longitude
ϕ	Spherical parameterization variable latitude
r	Spherical parameterization variable radius
n_θ	Discretized element of θ
n_ϕ	Discretized element of ϕ
n_r	Discretized element of r

Probability

$\mathbf{P}$	Gaussian covariance matrix
μ	Gaussian expectation value
$\mathcal{N}$	Gaussian probability density function
p	Probability density function (pdf)
$\boldsymbol{X}$	Random variable vector for tracking states
$\boldsymbol{Z}$	Random variable vector for tracking measurements
X	Random variable for state
Z	Random variable for state
$\boldsymbol{x}$	Realisation of $\boldsymbol{X}$
$\boldsymbol{z}$	Realisation of $\boldsymbol{Z}$
x	Realisation of X
z	Realisation of Z
$\mathbf{h}$	Non-linear measurement function for system observations
$\mathbf{H}$	Linear measurement matrix for system observations
$\mathbf{f}$	Non-linear transition function for system process
$\mathbf{F}$	Linear transition matrix for system process
$\mathcal{V}$	Innovation vector
$\mathbf{S}$	Innovation matrix
$\mathbf{W}$	Kalman filter weight matrix
$\mathbf{w}$	Gaussian measurement noise vector
$\mathbf{R}$	Measurement noise matrix
$\mathbf{v}$	Gaussian process noise vector
$\mathbf{Q}$	Process noise matrix
ANEES	Average normal estimation error squared
NEES	Normal estimation error squared

RMSE	Root mean squared error
k	Time increment
$AHED$	Average Hellinger distance
$COVDIFF$	Covariance difference
HED	Hellinger distance

Other Symbols

$\epsilon^{\mathrm{v}}_{beam}$	Beam shape error
ψ^{n}	Ships heading in NED space
ϵ^{v}	Numerical error in depth buffer
$\epsilon^{\mathrm{v}}_{max}$	Maximum numerical error in depth buffer

Abbreviations

AA Autonomous Agent
AI Artificial intelligence
AIS Automatic Identification Systems
ANEES Average Normalized Estimation Error Squared
ANS Autonomous Navigation System
API Application Programming Interface
BD Bhattacharrya coefficient
CCD charged coupled device
CMOS complementary metal oxide semiconductor
CPU Central Processing Unit
DOF Degree Of Freedom
DOTS Data Oriented Technology Stack
EMR Electro Magnetic Radiation
FMI Functional Mockup Interface
FMU Functional Mockup Unity
G-buffering graphic-buffering
GNSS Global Navigation Satellite System
GPST Global Positioning System Time
GPU Graphic Processing Unity
gRPC g remote procedure call
HD High Definition
HDRP High Definition Render Pipeline
HED Hellinger Distance
HLSL High Level Shading Language
I^2C Inter-Integrated Cicuit
iid independent and identically distributed
IMU Inertia Measurement Unit
Indie independent game developers
INS Inertial Navigation System
IP intellectual properties
IR Infrared (camera)
JIPDA Joint Integrated Probabilistic Data Association
lidar Light detection and ranging
LLH Longitude Latitude Height
LOD Level Of Detail
MCU Micro Controller Units
MSS Marine Systems Simulator

NEES Normalized Estimation Error Squared
OSP Open Simulation Platform
PCB Printed Circuit Board
pdf Probability density function
PPK Post Processed Kinematic
PRNG Pseudo random number generator
Protobuf Protocol Buffers
radar Radio detection and ranging
RGB Red Green Blue (camera)
RMSE Root Mean Squared Error
ROS Robot Operating System
RTK real time kinematic
SCC Shore Control Center
SD Secure Digital (card)
UART The Universal Asynchronous Receiver/Transmitter
YOLO You Only Look Once

Contents

Chapter 1

Introduction

1.1 Motivation and background

Thinking back on all the advancement there have been in artificial intelligence (AI), it is hard to believe most of this have happened during the last decade. With the deep learning revolution of 2012, the ImageNet challenge transformed the AI industry forever, leading to breakthroughs in several fields. These breakthroughs soon found its way into autonomous vehicles, fusing itself with cybernetics that had traditionally dominated the field. Today, autonomous agents (AA) are used in the deepest of oceans to the deepest of space, acting in environments that have traditionally been operated by humans. Yet, despite these breakthroughs, we still have little understanding of how AI work, moreover having methods that can tell when they are safe to use.

In the coming decade, there will be an increasing focus on explainability in AI, and its importance for safe autonomy. One of the reasons being classification societies such as DNV seek to give trustworthy evaluations of future vehicles. This is especially important in the maritime industry, where large risks needs to be compensated for by safety. The steady increase in ship complexity regarding software and systems, have not made these tasks any easier. All of this goes to show it has become increasingly difficult to design, build, operate and assure maritime vessels.

When handling problems yet to see any solution, it is wise to remember oneself of the fundamentals in the scientific method. The tradition in engineering science, have been to gather data through experiments in order to validate methods. However, due to the complex operating environment AAs are subject to, little attention have gone to designing experiments for *reproducibility*. Moreover, the lack of attempts of reproducing experiments, indicates an entrance barrier where information from the original experiments or available resources, are missing. As a consequence, subfields within AI such as target tracking, computer vision and machine learning, have relied on relatively low quantities of datasets in order to validate scientific findings. This has resulted in a small and fragile test coverage for AAs, despite them being sensitive to data from different operating environments [1], the risk of dataset biases [2] and the results validity if something where to happen to the original data. The latter being very relevant for data driven methods [3].

Since the probability of bad research have shown to increase with data biases, complexity and few samples [4], a first step towards trustworthy AI, should simply be about making tests and results easier to *reproduce*.

1.2 Problem description and main ideas

An approach to handle such challenges, is through simulations. This has been on the agenda for the car industry for several years, but in resent years, the maritime industries have started to have their take on this as well. Especially the focus have gone to modeling scenarios based on the automatic identification systems (AIS), prevalent in most big ships today. Here AIS have been used for both risk identification [5] and as a basis for simulation-based verification [6]. Unfortunately, as concluded in [5], AIS data alone is not sufficient for evaluating safety of an AA, since agents rely on data from high fidelity sensors, such as camera images, that can be misinterpreted. In addition, not all boats of relevance uses AIS, limiting relevant cases for safety assessments.

Several frameworks have been created to address these issues for the car industry. Carla [7] is one of many car simulators that supports sensor simulations through the use of game engines. But sensor simulations are just a piece of the puzzle. The framework must also allow engineers and scientists to conduct studies using their own external platforms for AA, vessel models and scenario customization. This is known as a digital twin framework, a synthetic replica meant to substitute real life systems for simulation based verification [6].

However, in the maritime sector there is no known digital twin framework available for the public with such functionalities. With this said, there are multiple initiatives that are driving a development towards such a framework. The *open simulation platform* (OSP) makes it possible to exchange maritime models and software packages across companies [8]. Implementation of an AA for the milliAmpere ferry, has been done using the robot operating system (ROS) as a platform, capable of being used in both simulations and on real hardware [9]. In addition, the same ferry has also been virtualized into a simulated Trondheim environment, made by students using the Unity game engine [10].

This simulator is known as *Autoferry Gemini*, an open source project aimed towards creating an equivalent framework for ships as Carla is for cars. The platform supports simulation of multiple electro magnetic radiation (EMR) sensors, through the use of the Unity game engines render pipeline. Models of ray casting sensors, such as lidar and radar, have been shown to run on a graphics processing unit (GPU) using the render pipelines depth buffers [11]. At the beginning of this master thesis, this technique was still limited as modeling errors named *beam shape errors*, emerged through the use of depth buffers [12]. Ray drop modeling for lidars is also absent in contrary to other simulators aimed at autonomous cars [13]. Camera sensors is also modeled, but the lack of available 3D models limits the sensors possible fidelity. However, most of these limitations have only been observed qualitatively, missing a quantitative method to determine the different sensor models fidelity.

This motivates the need of metrics that can be used to measure the fidelity of a simulation, such that improvements for the sensor models can be measured. Several methods already exist for quantifying image and video fidelity [14, 15]. Likewise, for AA relying on sensor fusion, well established performance metrics already exist considering target tracking [16]. Despite this, little attention have been given to measure the AA's change in performance, when switching between real and synthetic datasets. This shows a severe gap in knowledge towards a future with simulation based verification of AA.

1.3 Contributions

The main contribution of this thesis is to enhance Autoferry Gemini, reducing the beam shape error, improving the lidar model and expanding the simulator into a digital twin framework, capable of generating sensor data in a virtual operating environment using scenario inputs. Based on this, a method for verifying the framework will be done by first creating two marine dataset, one from reality, and one reproduced synthetically from the other. The second step will use a Hellinger distance metric to compare the outputs of the same target tracker running on both datasets. The belief is that since the datasets ought to be identical with respect to states, this metric should be able to tell the degree of reproducability of sensor data between simulation and reality, and thus quantifying the sensor models fidelity. Further, by studying established performance metrics for the tracker, this new metric can be contextualized and additionally analysed from known knowledge in the field of sensor fusion. To summarize, this entails:

- Creating a digital twin framework, capable of generating sensor data for AA from scenario inputs.
- Improving Autoferry Gemini's lidar model, by among others reducing the beam shape error.
- Creating two marine dataset containing lidar and camera data, one based on reality, and one reproduced synthetically from the other.
- Introducing a metric that measures the synthetic datasets fidelity
- Study how a tracker behaves between synthetic and real datasets

1.4 Outline of thesis

The thesis begins with describing the digital twin framework in Chapter 2, covering the platform compositions, architecture and communication that makes up the framework. This is followed by the creation of the real dataset in Chapter 3, describing the test setups, sensor and scenario recordings before ending with ground truth processing. The corresponding synthetic dataset is described in Chapter 4, where a thorough analysis of the depth buffer is done to reduce the beam shape error, before the lidar and camera models from [12] are discussed and improved. Chapter 5 starts out with explaining the theory behind target tracking, before the synthetic data is compared to its real counterpart, using various metrics and a discussion of their results. Finaly, Chapter 6 discusses the major findings from the thesis, before giving a final conclusion and suggestions of topics for further work. At the end of the thesis, 2 appendices describe the author's involvement as a project manager for the research group mentioned in the Acknowledgements. Appendix A shows the project plan and divison of labour during the project, while Appendix B contains all the workpackage descriptions the researchers where given.

Chapter 2

Digital twin framework

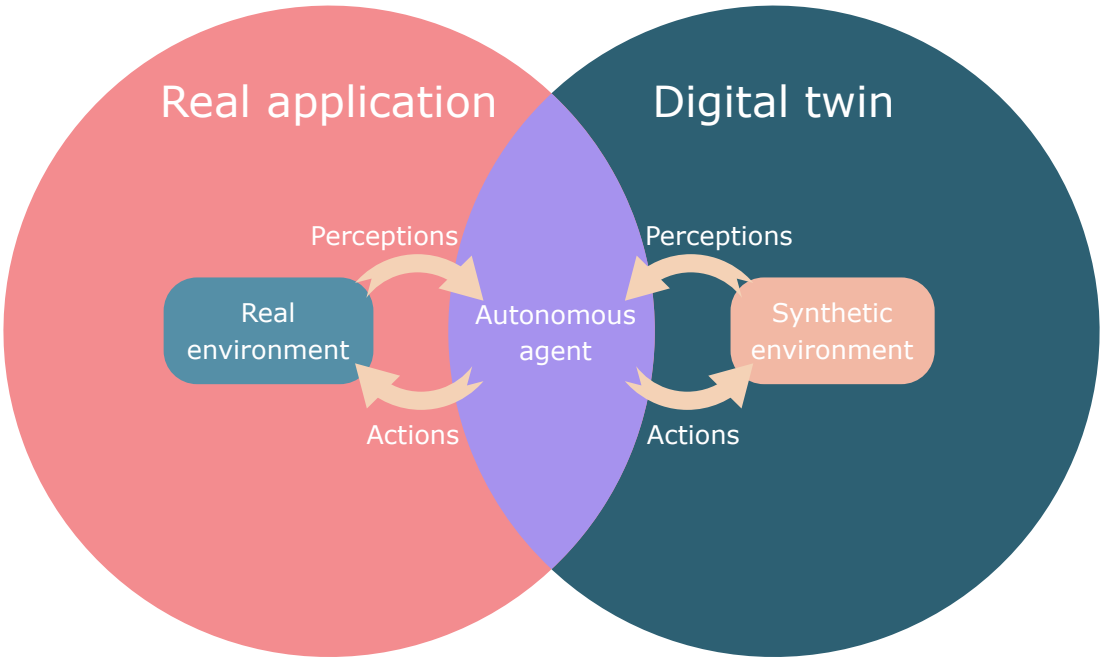


Figure 2.1: In autonomy, a digital twins purpose is to determine an AAs safety before being deployed into the real application

DNV have previously demonstrated the use of a digital twin framework for doing simulation based verification [6], where they focused on the AAs actions in various scenarios. In this thesis we will focus on the agents perceptions in figure 2.1, addressing the fidelity of its exteroceptive sensor models and its following influence on the AAs behaviour. To handle this problem, several software platforms will be used in order to simulate sensors, customize scenarios, and making the data available for the AA. In addition, a method for validating the simulation fidelity is proposed. To achieve this, a software architecture is made using high performant communication links between the platforms to handle the big data traffic caused by simulating sensor data. We will start with introducing the platforms in Section 2.1, before presenting the framework 2.2.

2.1 Platforms

When software becomes to large to be handled by a single team or person, connecting existing software platforms together into a framework becomes a viable solution. It is important to be aware that the resulting framework will itself be considered a platform, but greatly abstracted with loosely connected modules making up the functionality. In this section a brief introduction of the platforms used for creating the Gemini frame are given in the following sections.

2.1.1 Open Simulation Platform

The open simulation platform (OSP) is an industry initiative for creating a cooperative simulation platform for systems engineering and testing, targeted on sharing maritime models without compromising companies intellectual properties (IP). These models can be anything from a propeller hooked to a ship engine, to dynamic positioning systems that automatically maintains the vessel position.

The initiative started with DNV GL observing the increase of "smart" equipment being installed in ships, where the assurance process became increasingly difficult to maintain and perform as technologies progressed. The interconnection these equipment could have towards each other also created challenges in preserving IP when designing new ship systems.

OSP handles these issues by introducing an open platform where companies can create models in what is called functional mockup units (FMU). These can be created by any programming language of the companies liking, where the FMU functions as a wrapper to secure companies IP, as well as creating a standard for software sharing. Following this standard, the units can communicate with each other, allowing third party software to function as decoupled modules in a digital twin framework. However, due to OSPs architecture, the units only support low data rates, making them unsuited for transferring big data streams commonly found in autonomous vessels.

Regardless, any digital twin framework must be able to integrate different vendors equipment in the simulation process in order for being useful in the long run. Using OSP as a bridge to existing software in the industry, makes it a crucial platform to have in a digital twin framework.

2.1.2 Unity

Unity is a cross-platform game engine created and maintained by Unity technologies since 2005. It has previously been associated with the production of low fidelity video games from independent game developers (Indie), but in recent years it has gradually been aiming at the high fidelity game markets, known as AAA-games. This shift has also given the engine a reputation in the automotive simulator market.

It is often compared to the Unreal engine, where it is perceived to have lower performance and fidelity because of its inferior c-sharp scripting language. Unity's investment towards a *data oriented technology stack* (DOTS) have in recent years challenged this perception, even though DOTS is still only in a development phase.

Regardless, this have not stopped automotive simulators such as Carla and LGSVL to use the engine for their digital twin frameworks.

With this said, more common AAA game engines still seems to have an edge regarding image fidelity. With Unreal 5's recent development of real-time Global Illumination from their Lumen system, in conjunction to the virtual geometry implementation using Nanite, Unreal 5 is capable of using high-poly 3D models, traditionally only being used for movie renderings. Cryengine is another popular game engine, where real time ray-tracing does not require specialized GPU hardware to function, as is still the case for both Unreal and Unity. These engines do however lack documentation and a sizeable community when it comes to customizing the render process. This has to do with their marketing traditionally being focused towards professional game studios. As a consequence, the support for customizing the engines rendering process is at the very least an entrance barrier for developers starting outside game companies.

In comparison, Unity's new scriptable render pipeline system is better documented, allowing developers to more freely design how and what the GPU processes. The high definition render pipeline (HDRP) is an example of this, that demonstrates Unity's capability of producing similar graphics as seen from AAA game engines in general. The engines additional support of general purpose GPU programs, allows Autoferry Gemini to customize render pipelines for non traditional applications, such as simulating radar and lidar sensors [11, 12].

Unity is not considered open source, but comes with a fee if the game is commercially released. Another option is the Godot engine, which might be an alternative due to it being open-source. With this said, much of the core functionalities that comes with the commercial engines would need to be implemented from scratch. The open-source approach would therefore require expertise not commonly found outside the video game industries.

2.1.3 Robot Operating System

The robot operating system is a collection of robotics software customizable to fit a vast range of robots and vehicles. It is considered to be a middleware platform, capable of communicating with low level electronic devices, while maintaining much of the benefits of higher level abstractions found in everyday computers. This mixture makes the platform particular adequate for autonomous vessels, as high level abstractions required for target tracking can be implemented along with low level vessel controllers that runs with faster intervals. However it is worth noticing that ROS is not a true real time operating system, but with the release of ROS 2.0 is capable of supporting real-time code. ROS is currently being used for autonomous vessels such as the milliAmpere ferry [9] as a platform for integrating a AA together with actuators and sensors.

2.1.4 Arduino

Arduino is a low level open source electronic platform, often used for prototyping the logic driving various *printed circuit boards* (PCB) with *micro controller units* (MCU). The MCUs are programmed using the Arduino language, a derivation of the C++ language, though substantially limited due to the MCU memory and processing power.

Several PCB devices supports the Arduino platform, ranging from sensors to data storage units. The *the universal asynchronous receiver/transmitter* (UART) and the *inter-integrated cicuit* (I^2C) are common industry standards used for communicating between such devices. These are known as communication protocols, which forms a network of specialised hardware. Due to Arduinos rapid prototyping philosophy, the user is seldom exposed to the technical details communication protocols, MCU and sensors have on the PCB devices.

Even though the Arduino platform uses various industry standards as those mentioned, the platform is not common to use in industrial and production cases. The ease of designing, choosing PCB devices and the rapid prototyping the platform offers, have however been sought after in creating some of the sensor recorders we will come back to in Section 2.2.2.

2.1.5 gRPC

For connecting the platforms used in the Gemini Frame, a general high performance messaging system is needed to transfer both big and small data at high rates. The *g remote procedure call* (gRPC) platform is a system made by Google for streaming data across servers that uses different data systems. This is done by setting up transmitters and receivers on an abstract level, using a shared message definition to serialize and de-serialize data during transfers. Messages can follow multiple serialisation techniques, but *protocol buffers* (Protobuf) are preferred due to it being supported by multiple programming languages, and thus multiple different platforms. In addition, Protobuf supports big data transfers such as raw image streams with rates up to 30-60 frames per second [17]. To achieve this, gRPC relies on creating specialised service programs acting as data transceivers on the client and server side, by auto-generating code from its API language.

2.2 Framework

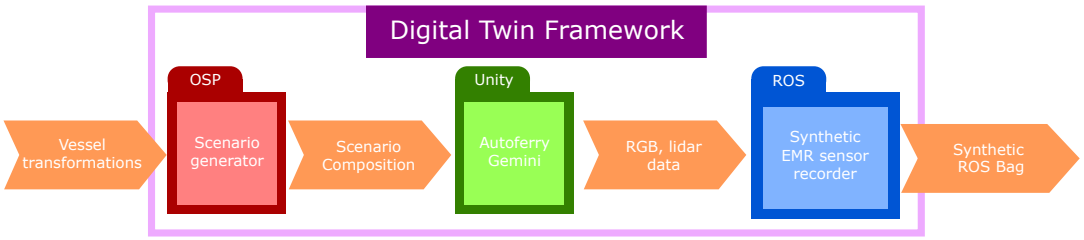


Figure 2.2: The digital twins framework architecture

Connecting some of the platforms mentioned in Section 2.1 forms a digital twin framework. In this section, we will briefly cover the composition of the framework, before introducing a method to validate it with respect to the digital twins purpose (Figure 2.1).

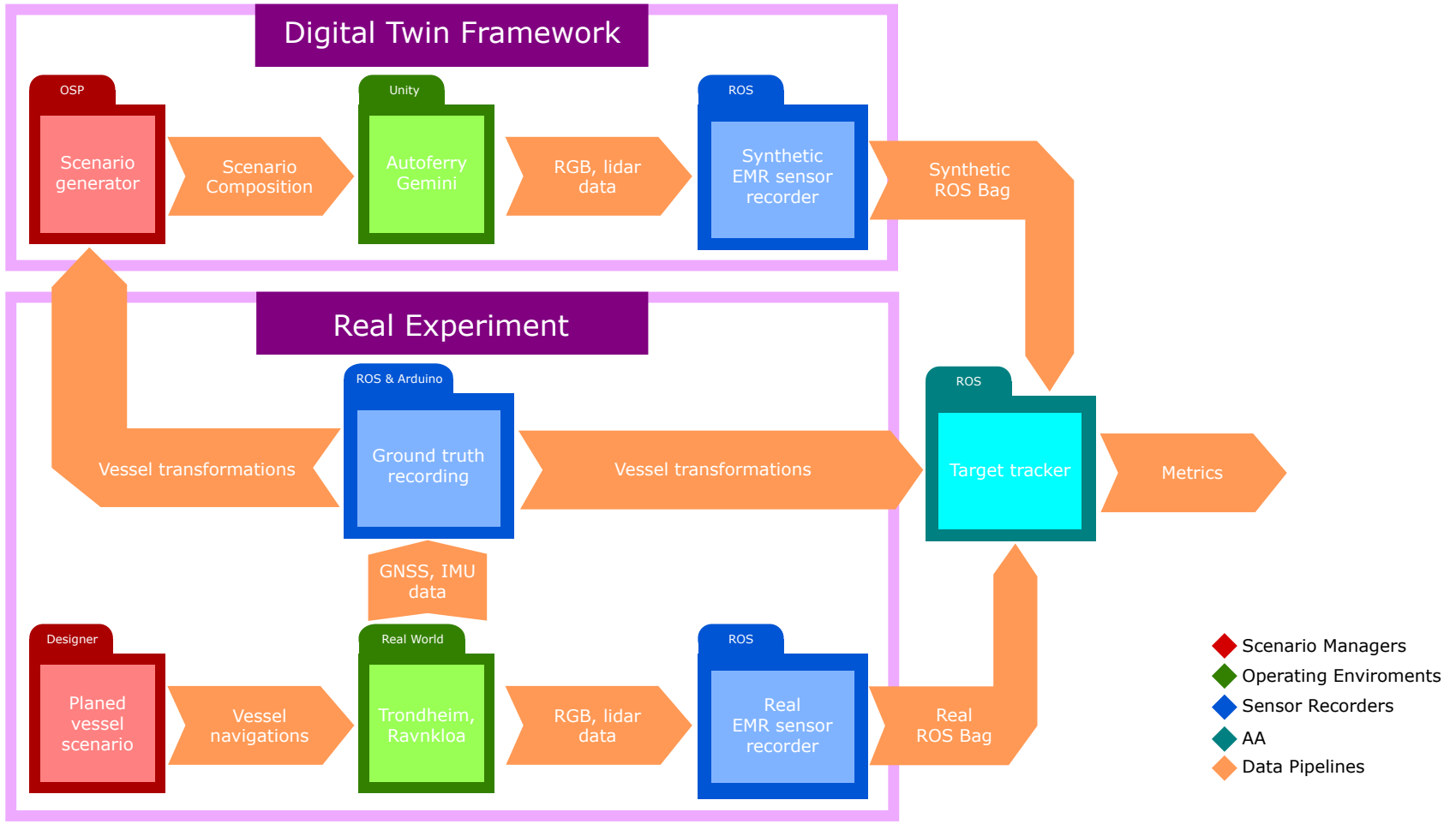
2.2.1 Composition

Previous frameworks have used AIS data to create scenarios in a low fidelity simulating environment [6]. Usually this have consisted of vessel transformations such as position and heading at sea. The same concept is used here, were transformations are feed into a scenario generator which creates a scenario composition of boats roaming around an environment. These vessel movements are feed into Autoferry Gemini [12] were lidar and camera sensors are modeled and transmitted further down the data pipelines. At the end, ROS is used to record the data, storing it as messages in what is called a *ROS bag*. To create the data pipelines, gRPC is used as a middle layer between these platforms, functioning as a glue that holds the framework together 2.2.

2.2.2 Validation

In order to validate the digital twin framework, a method is required to address the fidelity of the simulator. Using the framework described in Section 2.2.1, vessel transformations create a synthetic dataset containing sensor data. Recording a real dataset with sufficient sensors, a synthetic replica can be made from the real data. Since the purpose of a digital twin is to check if AAs are safe for real applications, we can test the AAs perception on the two datasets using existing algorithms. A target tracker in this case makes it possible to evaluate several metrics considering a AAs performance. Comparing the metrics from the two datasets tells us if analysis of AAs made by using synthetic data can be trusted.

Validation method



Chapter 3

Real Dataset

Figure 3.1: Drone photo from scenario 4 during the experiments.
Photo: Mikael Sætereid / Fosen innovasjon

In order to validate the digital twin framework using the architecture in Section 2.2.2, the thesis required more data than existing maritime datasets had at that time. This involved planning several maritime scenarios involving interactions with boats and recording multiple sensors of interest to validate the framework. These recordings needed to be setup, both by creating new equipment and by using existing sensor rigs with exteroceptive sensors. Finally, an experiment was performed by a research group consisting of several master and PhD students, dividing the labour of acquiring data and processing the result. This Chapter begins with describing the dataset content needed to validate the digital twin framework, before an insight of the experiment setup is given. Lastly, Section 3.3 goes through the final recordings, post processing and data validation procedures.

3.1 Dataset content

Since the digital twin framework is based on the synthetic sensors from Autofery Gemini [12], this requires recordings of maritime scenarios with electro magnetics radiation (EMR) sensors in order to compare real with synthetic sensor data. Ground truth data for position, velocity, and heading of the participating boats is also required to be able to reproduce the scenario composition, in addition to the tracker needing target positions and velocities to produce some of the metrics (Section 2.2.2). In order to properly validate the framework, the tracker also needs to be subjected to several scenarios, to see how its performance varies. We begin with introducing the essential sensors recorded in the dataset, before describing the scenario composition in Section 3.1.2.

3.1.1 Sensors

Autoferry Gemini is capable of simulating lidar, radar, RGB and IR cameras, but to limit the scope, we will only use the lidar and RGB cameras in this thesis. In addition, *inertia measurement units* (IMU) and *global navigation satellite systems* (GNSS) are needed to obtain ground truth data.

Lidar

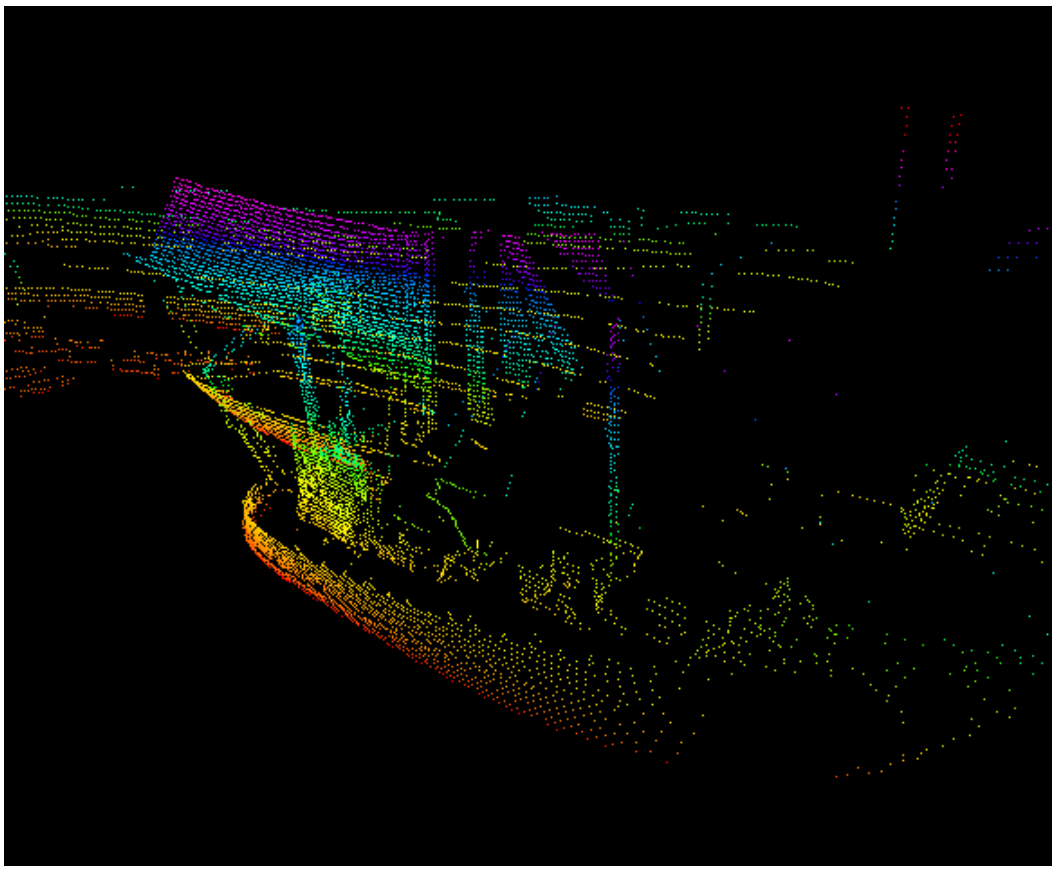


Figure 3.2: Point cloud of a boat

Light detection and ranging (lidar) sends and receives infrarad light rays in an intervall of τ in order to determine the range to a nearby object. This is done by using the the speed of light v_c in a vacuum or in the medium the lidar is operated in to estimate the range to an object:

$$r_{\text{li}} = \frac{v_c \tau}{2}.$$

This tells us the range to an object for one ray instance. Usually lidars are equipped with multiple lasers, and are rotated in order to form points of the surroundings. Several different lidars exist, but the one presented here shoots rays in spherical coordinates, where each point is represented with a latitude θ and longitude ϕ:

$$\text{Q}_{\text{li}} = r \begin{bmatrix} sin(\theta)cos(\phi) \\ sin(\theta)sin(\phi) \\ cos(\theta) \end{bmatrix}. \tag{3.1}$$

Usually lidar sensors gives a collection of points from a full rotation, creating what is knows as a point cloud. An example of this is seen in Figure 3.2

RGB camera

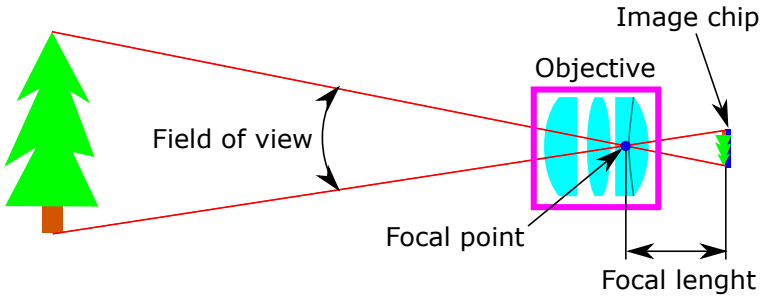


Figure 3.3: Illustration of a cameras functioning

Cameras rely on collecting photons using an image chip usually made of either *complementary metal oxide semiconductors* (CMOS) or *charged coupled devices* (CCD). These chips collects photons of all wavelengths close to the visible light spectrum, using small grid cells known as pixels to form an image. This creates what is known as a monochrome camera. For color cameras, color filters are used to separate red, green an blue wavelengths into individual pixels, usually forming what is known to be a *Bayer filter*.

Cameras are often accompanied with objectives, composed of lenses, mirrors or a combination of them to focus incoming light rays onto the image chip. The objective allows the camera to see different field of views depending on its focal length and chip size. A quick illustration of this is seen in Figure 3.3

GNSS reciever

The *global navigation satellite system* (GNSS) uses satellites to determine a receivers longitude, latitude and height position on the earth. It consists of several satellite systems depending on the receivers location on the earth, but in european waters Galileo is the most prominent. The satellites broadcasts their orbital position, timestamped with a very precise atomic clock which is synced between the satellites. From this, the position is determined by having at least 3 different satellite messages interpreted by the GNSS receiver, giving meter to millimeter precision depending on the data processing technique and the receivers antenna. In addition to satellites, base stations on the ground can also broadcast correction data to the receiver to enhance its precision, using a processing technique called *real time kinematics* (RTK) [18]. Some receivers also supports logging of raw sensor data, such as pseudorange, carrier-phase, Doppler and signal to noise. This allows for non real time processing techniques such as *post process kinematics* (PPK) [18], which can achieve an even higher precision than with RTK, by using correction and raw sensor data after the experiments have taken place.

IMU

Inertial measurement units (IMU) measures a body's spesific force, angular rate and sometimes orientation. It consists of several sensors including accelerometers, gyroscopes, and occasionally magnetometers combined with micro controllers to calculate the measurements from the sensor outputs. In navigation it is often combined with GNSS receivers to form an *inertial navigation system* (INS), which combines the high update rate from the IMU with the absolute position from the GNSS receiver to form a more reliable and usually precise positioning system.

3.1.2 Scenario composition

Figure 3.4: Scenario setup for workpackage R2 with a spesified location, ownship containing EMR sensor modules and targets with ground truth data available

The scenarios takes place at Ravnkloa Tronhdeim in order to make use of the exiting synthetic operating environment for Autoferry Gemini [12]. Here an *ownship* equipped with EMR and positioning sensors are subjected to *target boats* approaching the vessel from different angles. To reproduce the scenarios, each boat is equipped with ground truth sensors.

3.2 Experiment Setup

From the scenario composition in Section 3.1.2, the research group acquired access to the milliAmpere ferry [9] as ownship, accompanied by Finn and Havfruen as target boats. Of these, milliAmpere was equipped with EMR sensors and ground truth sensors, and the target boats with ground truth sensors. In addition, 3 base stations where needed to log extra data of importance. The research group also divided the experiment setup into separate responsibility areas:

- Ingunn Kjønås created scenario descriptions
- Øystein Kaarstad Helgesen configured the ownship for sensor logging
- Magne Sirnes setup a basestation from land to log additional sensor data
- The author fixed ground truth sensors for the target boats

3.2.1 Ownship

The ownship milliAmpere is an autonomous passenger ferry, designed to operate in urban areas to transport people across rivers and channels as an alternative for bridges. Several departments at NTNU uses the ferry for research purposes, in addition to master and PhD projects.

Description

The ferry is bidirectional, and are steered by two azimuth thruster driven by lead-acid batteries. On the top of the ferry, a collection of EMR sensors are attached to a sensor rig. Under the roof, two separate computers running ROS are used to gather data from the sensor rig and for control and navigation. Images of the ferry can be seen in Figure 3.5, alongside with pictures of the 3D model, and dimensions.

EMR sensor rig

The sensor rig attached to milliAmpere consists of 5 RGB and IR cameras, 1 lidar and 1 radar, seen in Figure 3.6.

The RGB cameras are evenly spread across the rigs azimuth axis, starting with the front camera which is aligned with the vessels longitudinal axis. Each camera consist of a *FLIR BFLY-PGE-50s5C-C* image sensor, attached to *Kowa LM6JC* objective which is pitched 15 degrees forward. This gives each camera 2448×2048 colored pixels with a 82° theoretical field of view. Due to bandwidth issues towards milliAmperes computers, only the 3 frontal cameras are used with an update rate of $5Hz$.

For the lidar, a *Velodyne VLP-16 Puck* is attached upside down, rotated 57.3° clockwise relative to the sensor rigs front camera direction, seen from above. It consists of 16 lasers spinning at a 10Hz revolution rate with a theoretical distance of 100m from the lidars center.

The other sensors are listed in Figure 3.6, but are not of importance in this thesis.

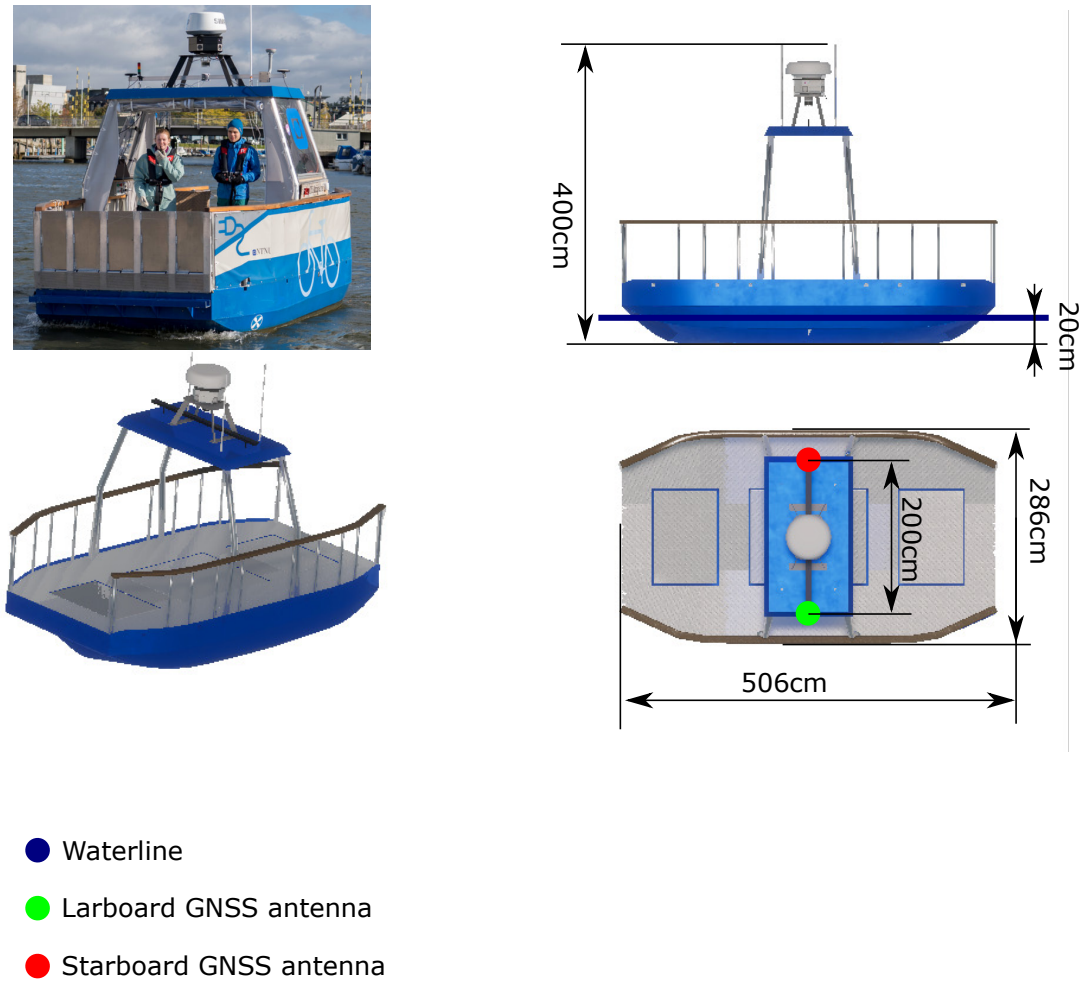
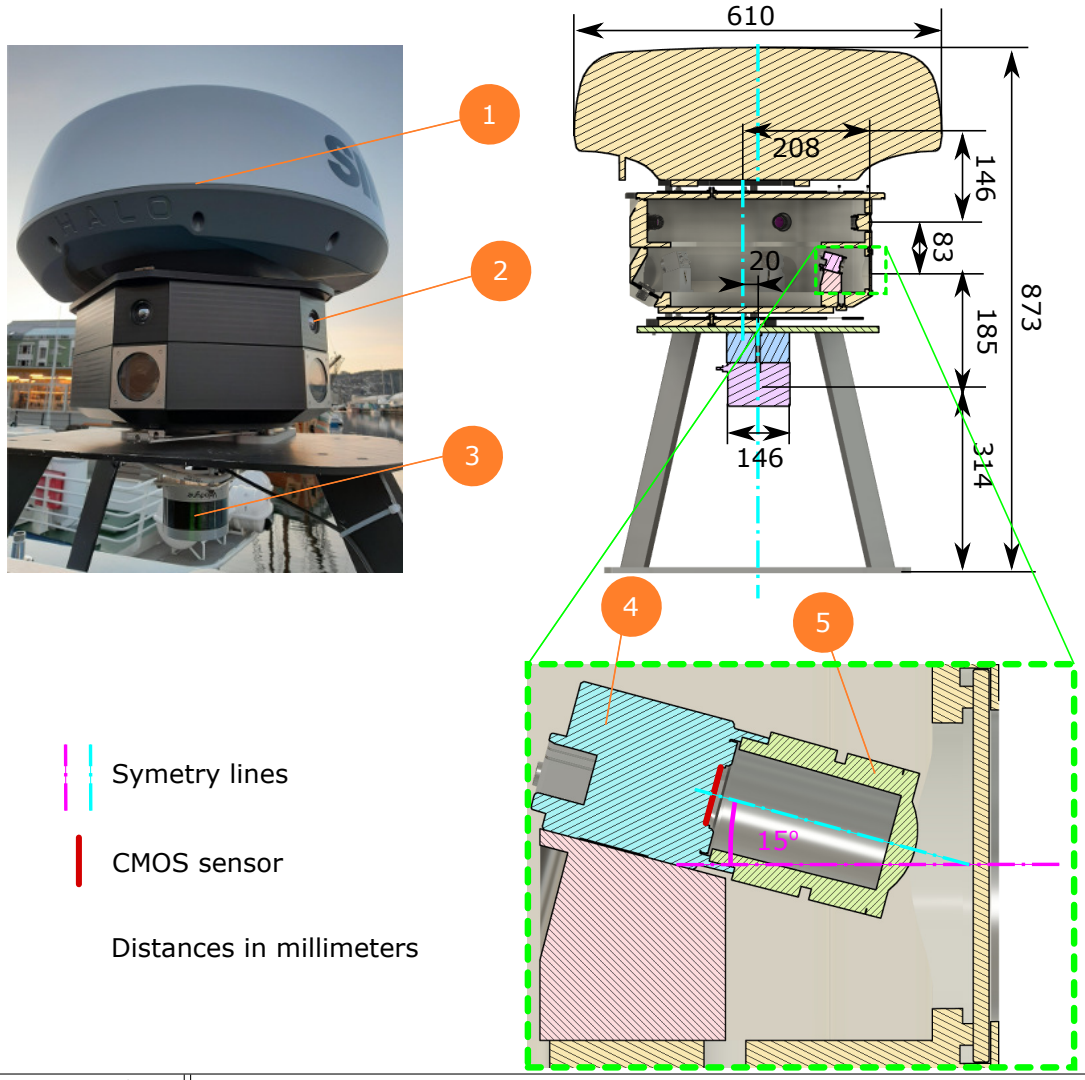


Figure 3.5: Real and synthetic ownship model with dimensions.
Real photo: Mikael Sætereid / Fosen innovasjon

Ground truth sensors

In addition to the sensor rig, milliAmpere is equipped with sensors for INS. First, the *Xsens MTI-20-2A5G4-DK* is used as the vessels IMU. This is combined with two GNSS antennas separated 203 cm from each other, mounted at the roof of milliAmpere seen in Figure 3.5 as red and green dots. These are connected to a *Hemisphere Vector VS330* which calculates both position and heading of the vessel.



Item number	Part	Description
1	Radar	Simrad 4G | Broadband radar
2	IR camera	FLIR Boson 640 | 95° (HFOV)
3	Lidar	Velodyne Puck | VLP-16
4	Image sensor	FLIR Blackfly GigE | BFLY-PGE-50S5C-C, Color
5	Camera objective	Kowa LM6JC | 2/3" 6mm MP C-Mount Lens

Figure 3.6: Ownship sensor rig description

3.2.2 Target ships

Havfruen and Finn are two civilian boats hired inn to function as target ships it the experiment. Havfruen functions as a medium sized boat while Finn functions as a small sized boat to test the trackers performance in different target cases. The boats are equipped with GNSS sensors to establish ground truths, but uses two different systems with different precision.

Description

Havfruen is a cabin cruiser equipped with two high precision target sensor rigs at its roof top, seen in Figure 3.7 as red and green dots. Finn is a small fishing vessel, equipped with two low precision target sensor rigs at its bow and stern positions marked in the same figure as for Havfruen.

Low precision target sensor rig

Originally the plan was to use high precision target sensor rigs for both target ships, but due to technical issues before the experiments took place, an emergency solution was used. Instead two *Garmin eTrex 10* spaced 293cm apart was used as a substitution.

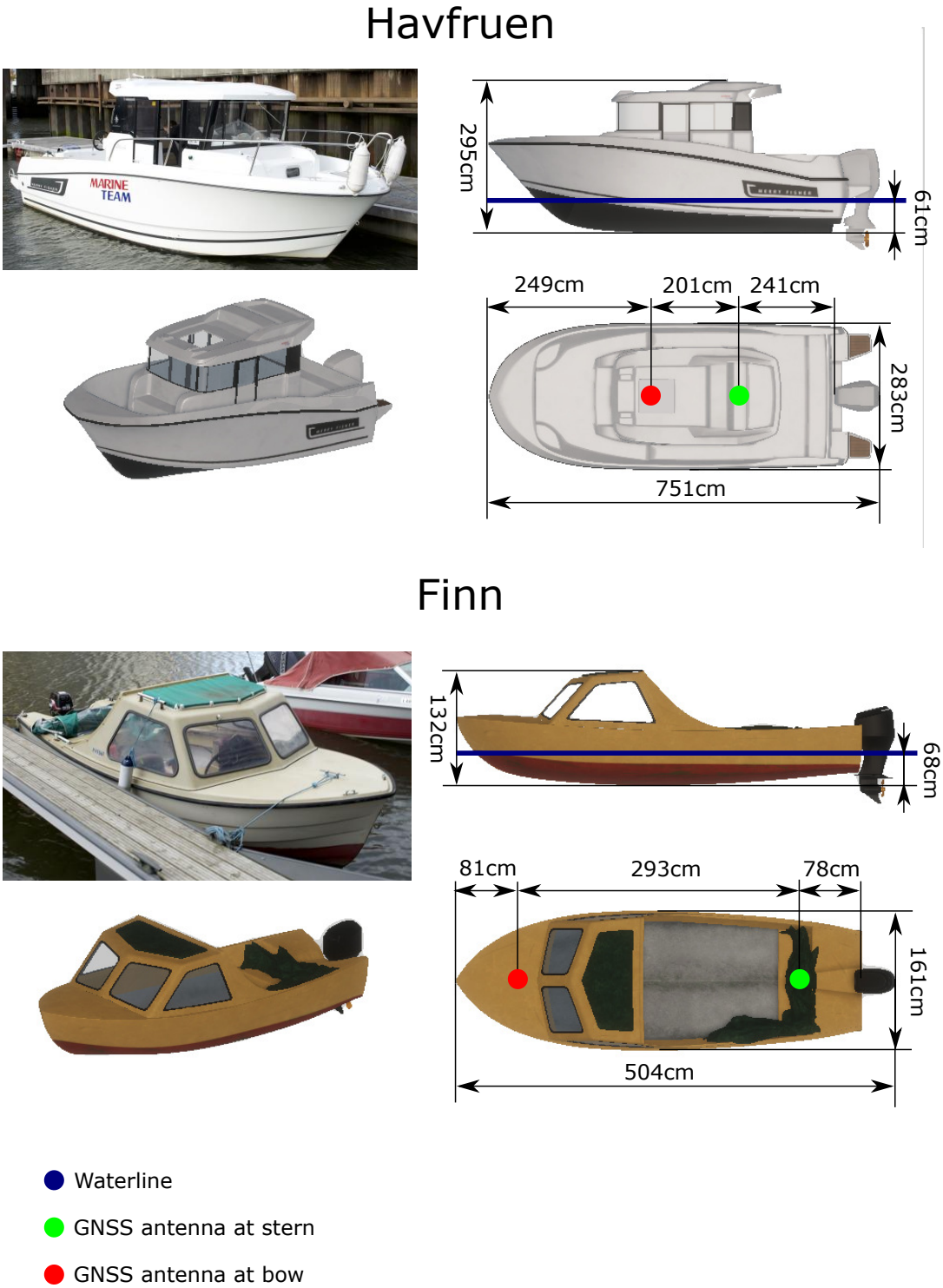


Figure 3.7: Real and synthetic target ship models with dimensions.
Real photos: Thomas Kaland / NTNU

High precision target sensor rig

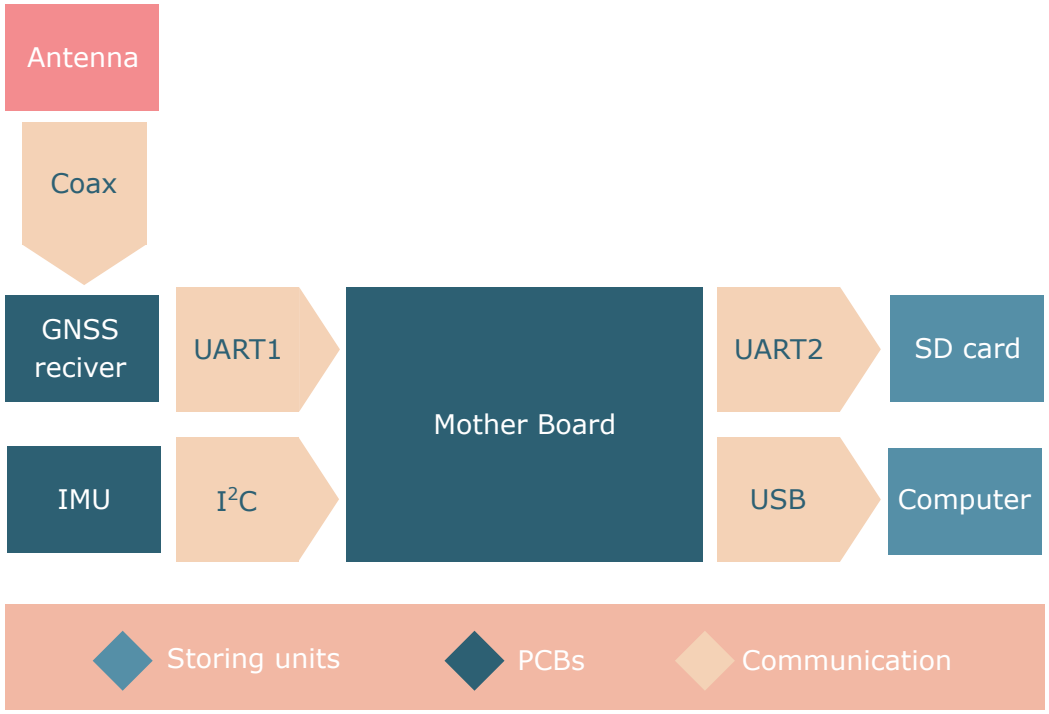
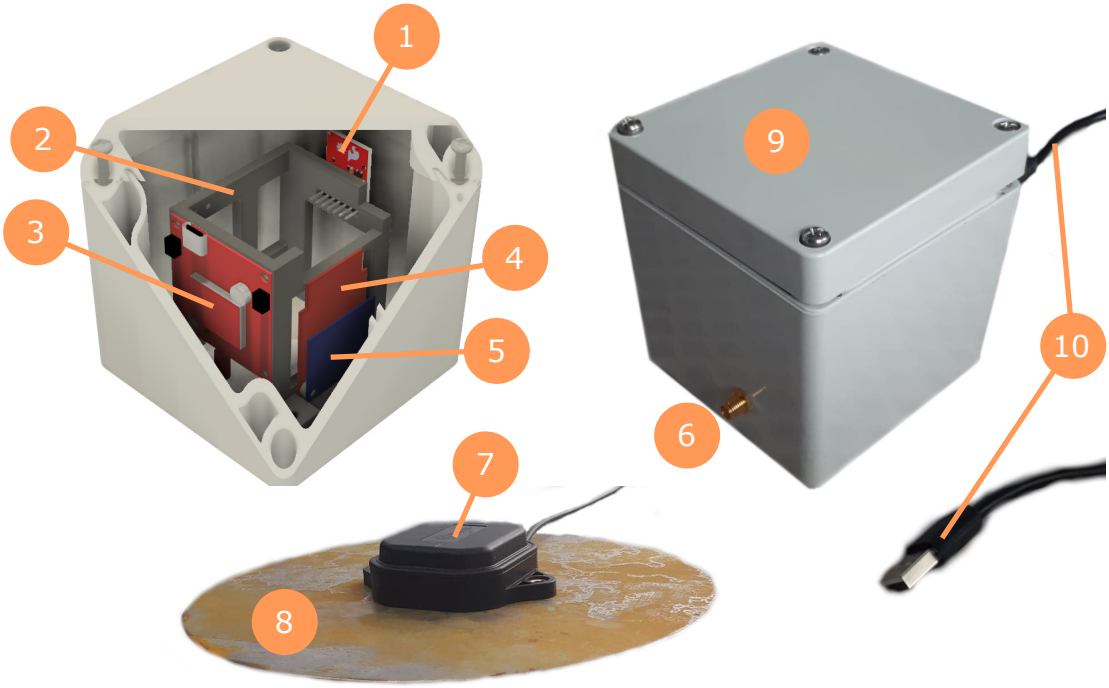


Figure 3.8: Architecture for the high precision target sensor rig

The high precision target sensor rig consists of a GNSS receiver (*ZED-F9P*), IMU (*BNO080*) and a motherboard which handles communications (Figure 3.8). Both sensors outputs raw data into two data logging units for redundancy. This allows the use of PPK processing after experiments. In addition, a 20cm ground plate is placed under the GNSS antenna to reduce multipath disturbances [19], increasing the precision further. Since the IMU contains a magnetometer, components with non-magnetic properties are mostly used, such as plastic and stainless steel. The fully assembled target sensor rig and its bill of material can be seen in Figure 3.9.

Data logging is done via a SD card reader, in addition to a computer using a program called RTKLIB [20, 21], storing the data in RINEX 3.0 file format. The motherboard is used for serial communication and supplying power to the different PCBs. This is done with I^2C towards the IMU, and a UART connection towards the GNSS receiver in order to free up bandwidth for the different serial busses.

The GNSS receiver is configured by using the manufacturers software called U-center, while the IMU is configured through Sparkfuns IMU library for Arduino. We set the receivers update rate to 5Hz, and enable the *UBX-RXM-RAWX* and *UBX-RXM-SFRBX* messages which contains the sensors raw data. For the IMU we set a update rate of 300Hz, timestamping using the GNSS receivers GPS time (GPST) in combination with the motherboards internal clock.



Item number	Part	Description
1	SD reader	SparkFun OpenLog (micro SD)
2	Mounting rack	3D printed
3	GNSS reciever	SparkFun GPS-RTK-SMA Breakout - ZED-F9P
4	Mother board	SparkFun Pro RF - LoRa, 915MHz (SAMD21)
5	IMU	SparkFun VR IMU Breakout - BNO080
6	Antenna plug	SMA plug for GNNS antennas
7	GNSS antenna	Multi-Band Magnetic Mount Antenna - 5m
8	Ground plate	Ø20cm, 1mm thick steel plate
9	Casing	IP68 graded ABS casing
10	USB	USB-A cable with water proof sealing

Figure 3.9: Target ships sensor rig description

To protect the electronics, the rig uses a water tight (IP68) casing dimensioned to be bouyant in case of an accident. To preserve the IP68 grading, blue silicone is used for mounting the antenna plug and a waterproof sealing for the USB cable. Inside the box, a 3D printed mounting rack secures that all connectors of the different PCB's are accessible during debugging.

3.2.3 Base Stations

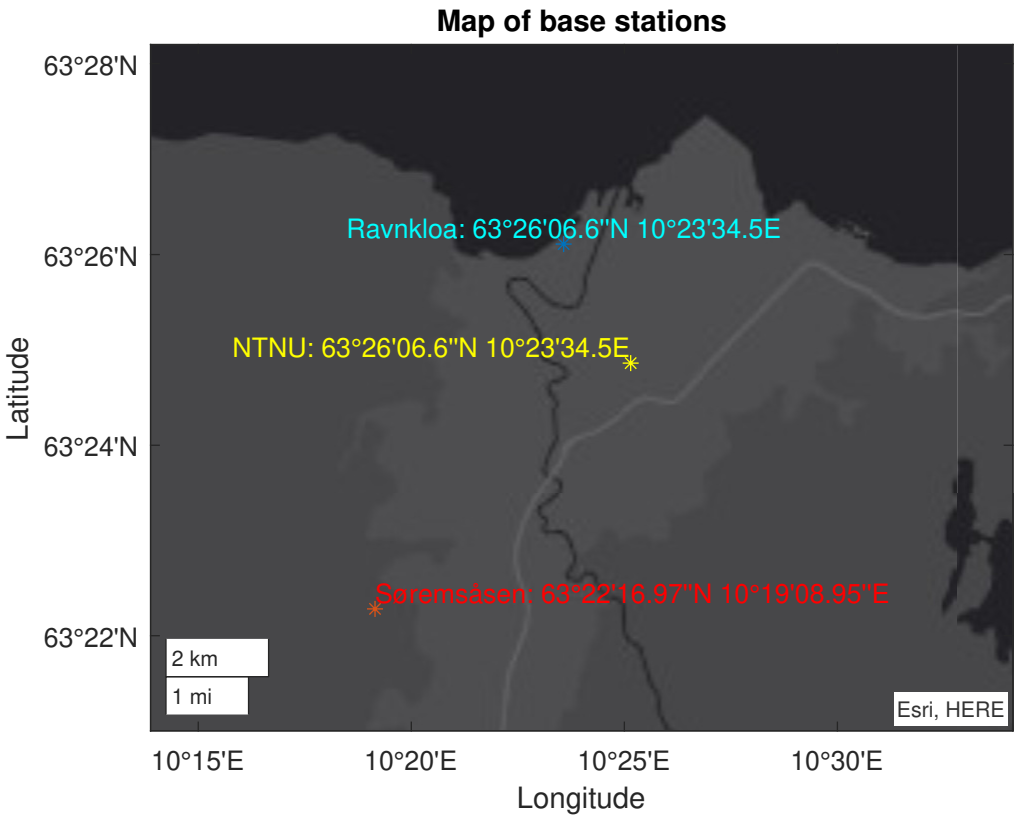


Figure 3.10: Position of base stations

In conjunction with the sensor rigs attached to the boats, base stations spread across Trondheim (Figure 3.10) is utilized to gather additional data. Firstly, GNSS correction data is gathered for the participating boats, including the NTNU base station for milliAmpere's RTK positioning, and Søremsåsen base station for logging data for PPK positioning for Havfruen. An additional basestation at Ravnkloa is used to log additional IR and radar data from the scenarios, for Magne Sirnes's research of joint localization and tracking [22].

3.2.4 Scenario descriptions

Following the scenario composition in Section 3.1.2, Ingunn Kjønås was responsible for designing scenarios. A more detailed scenario description is therefore found in Section 4.2.1 in her project thesis [23]. Originally 15 scenarios operating at Ravnkloa was designed, but due to recording issues for ground truth data during the experiments, only scenarios from the 15.09.2020 was obtained (Section 3.3). The scenarios was intended to fulfill several topics relevant for sensor fusion, such as obscured targets and varying target maneuvers. Figure 3.11 shows an example of this, in addition to defining colors, stop and end positions used for the contending ships throughout this chapter.

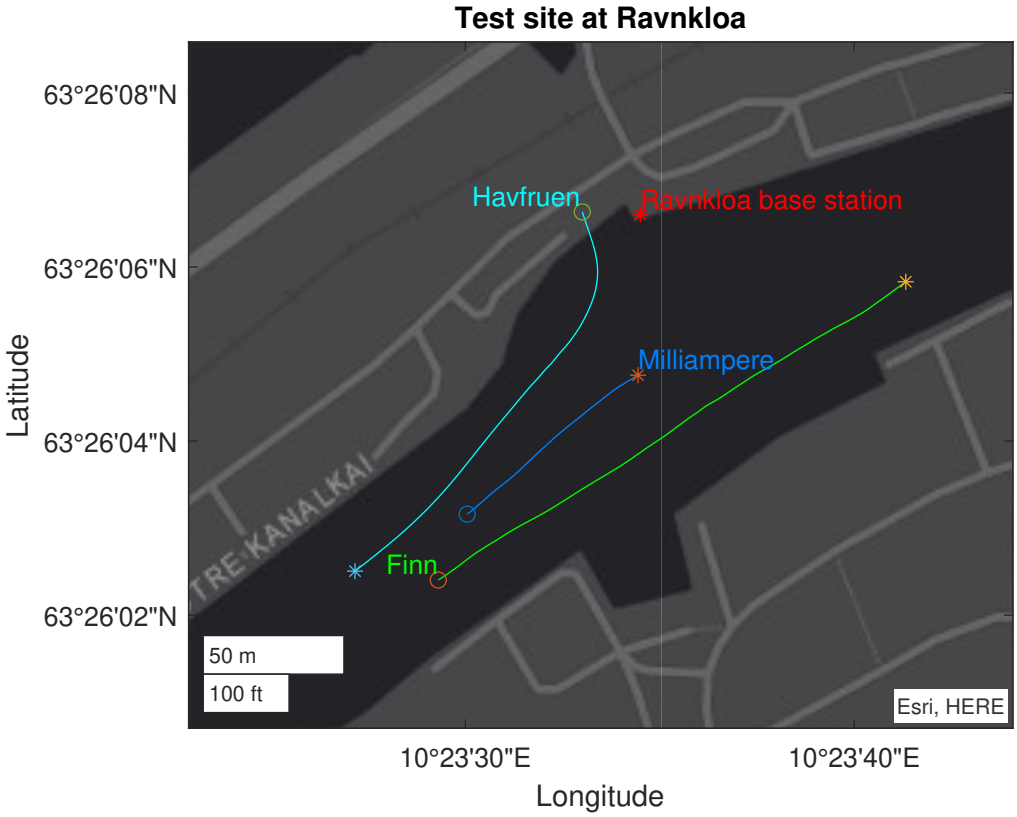


Figure 3.11: Scenario example. Star icons indicates the end positions of vessels, while circles indicates the start positions of the vessels for the scenario

In total, 9 scenarios was planed for the first day of testing, illustrated in Figure 3.12. The first two scenarios was designed to study a ferry crossing a channel. 3-4 consisted of evasive maneuvers from a halted ferry at different distances, while 5 had the Finn boat stop to avoid collision while Havfruen performed an evasive maneuver. 6 and 8 had the target boats follow a line or a constant bearing maneuver, creating obscured target situations. 7 had a similar purpose, where obscured targets suddenly separated to create a surprise situation for the autonomous system. The last scenario focused on a target suddenly appearing out of a tunnel, while a secondary target held a constant course on the opposite side.

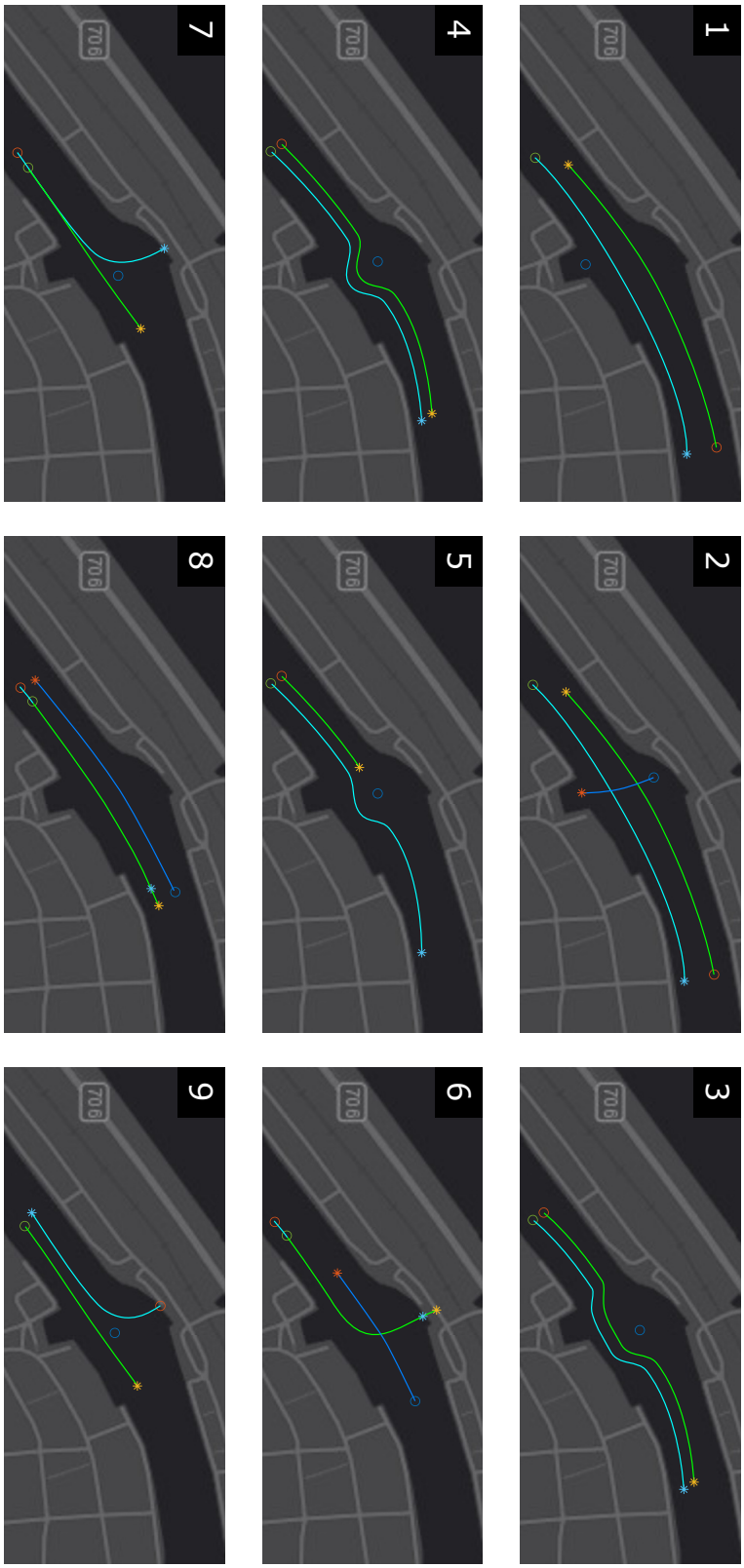


Figure 3.12: Planed scenarios for the first day. The star icons indicates the end position of the vessels during the scenarioes, while circle indicates the beginning.

3.3 Data acquisition

Using the experiment setup described in Section 3.2, tests was conducted at Ravnkloa in Trondheim between 15.09.2020 and 17.09.2020, performed by a research group of 5 peoples. Of these, Ingunn Kjønås and Øystein Kaarstad Helgesen was responsible for sensor recordings and piloting of milliAmpere during the experiments. Finn was piloted by Michael Ernesto Lopez, while the author was responsible for the ground truth recordings in addition to piloting Havfruen. Magne Sirnes managed sensor recordings at Ravnkloa basestation, but with no relevance for the thesis.

3.3.1 EMR recordings

Using the ownship's sensor rig in Figure 3.6, 4 different EMR sensor types was used in the scenario recordings. Some issues where present during the experiments, but sufficient for the purposes of the thesis.

It was discovered that the frontal camera was not in correct focus, in addition to the camera specifications lacking correct calibrating parameters with regards to the field of view. Fortunately, the other camera images had better focus, considered to be good enough for the purposes of this thesis. The lidar data had no apparent issues in the recordings.

3.3.2 Ground truth recordings

During the experiment, several technical issues occurred that resulted in loss of ground truth data for the participating ships. First consisted of a malfunctioning IMU, remedying one of the target sensor rigs useless for estimating heading with only the IMU. Second issue involved communication error with the remaining IMU. Fortunately all the GNSS receivers functioned well during the first day of experiments.

The remaining test days, GNSS data from the Garmin units was lost due to memory issues, and logged data from Havfruen turned out to be error messages instead of GNSS data. Because of these issues, only sufficient datasets recorded for scenario 3 - 9 the first day was obtained (Figure 3.13).

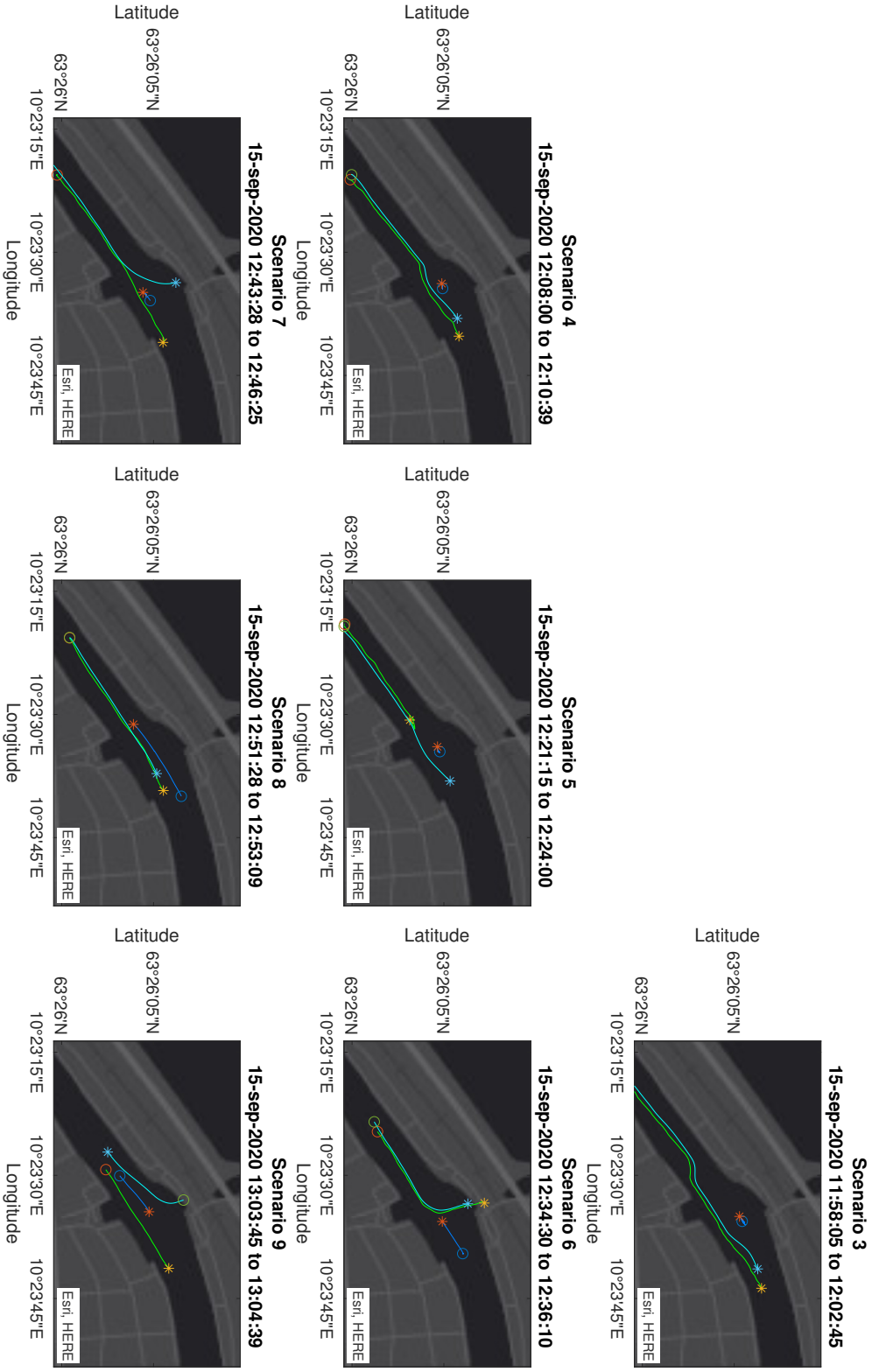


Figure 3.13: Recorded scenarios from the first day. The star icon indicates the end position of the vessels during the scenarioes, while circle indicates the beginning

Post processing and Validation

Since no IMU data was recorded for the participating boats, it was decided to base the ground truth solely on the GNSS data to form a 3 degree of freedom (DOF) ship description consisting of north coordinate y^{n}, east coordinate x^{n} and heading ψ^{n} relative to Ravnkloa base station with longitude y_0^{l} and latitude x_0^{l} given in Figure 3.10. This was done by first interpolating the GNSS data to 10 Hz to remove the unequal update frequencies between the different receivers. Estimating heading would then be possible as the data would be synchronized in GPST. Finally the data was transformed from *Longitude, Latitude, Height* (LLH) to *North, East, Down* (NED) using the *Marine Systems Simulator* (MSS) toolbox [24, 25]:

$$x^n = \frac{x^{\mathrm{l}} - x_0^{\mathrm{l}}}{\mathrm{atan2}(1, R_M)}, \tag{3.2}$$

$$y^n = \frac{y^{\mathrm{l}} - y_0^{\mathrm{l}}}{\mathrm{atan2}(1, R_N \cos(x_0^{\mathrm{l}}))}, \tag{3.3}$$

where the ships longitude y^{l} and latitude x^{l} were calculated for Havfruen by PPK using the open source software package *RTKLIB* [20, 24], single point processed GPX data from the handheld *Garmin eTrex 10* units for the Finn boat, and RTK data from milliAmperes *Hemisphere VS-330*.

As we are only interested in North and East positions, the reference ellipsoid WGS-84 is regarded to be sufficient for satellite navigation systems instead of using the more complex physically based Earth geoid. To account for the Earths elliptic shape, the prime vertical R_N and the meridian radius R_M were calculated as [24]:

$$R_N = \frac{r_e}{\sqrt{1 - e^2 \sin(x_0^{\mathrm{l}})}}, \tag{3.4}$$

$$R_M = R_N \frac{1 - e^2}{\sqrt{1 - e^2 \sin(x_0^{\mathrm{l}})}}, \tag{3.5}$$

using the eccentricity $e = 0.0818$ and equatorial radius $r_e = 6\ 378\ 137m$ parameters from WGS-84. Finally, heading is estimated by calculating the angle between the GNSS antenna positions, placed at the ships bow (sb) and stern (ss):

$$\psi^{\mathrm{n}} = \mathrm{atan2}(y_{\mathrm{sb}}^{\mathrm{n}} - y_{\mathrm{ss}}^{\mathrm{n}}\,,\, x_{\mathrm{sb}}^{\mathrm{n}} - x_{\mathrm{ss}}^{\mathrm{n}}) \ \in [-\pi, \pi), \tag{3.6}$$

giving an angle relative to the North direction.

To validate the GNSS accuracy, a plot of the GNSS recivers relative positions $y_{\mathrm{sb}}^{\mathrm{n}} - y_{\mathrm{ss}}^{\mathrm{n}}$, $x_{\mathrm{sb}}^{\mathrm{n}} - x_{\mathrm{ss}}^{\mathrm{n}}$ was made for the Finn boat and Havfruen (Figure 3.14) to be able to determine and exclude the parts where the GNSS data was likely to be corrupted. The idea being the closer a measured point was to the ground truth circle, the higher the accuracy for that measurement would be.
From this it was seen that Havfruen generally excelled high accuracy for all scenarios except scenario 5, and small parts of 3 and 9, while Finn struggled generally with ground truth for every recorded scenario. The size of this relative error was quantified by:

$$\epsilon_{\mathrm{gnss}}^{\mathrm{n}} = \left| \sqrt{(y_{\mathrm{sb}}^{\mathrm{n}} - y_{\mathrm{ss}}^{\mathrm{n}})^2 + (x_{\mathrm{sb}}^{\mathrm{n}} - x_{\mathrm{ss}}^{\mathrm{n}})^2)} - |\mathrm{Q}_r^{\mathrm{n}}| \right|, \tag{3.7}$$

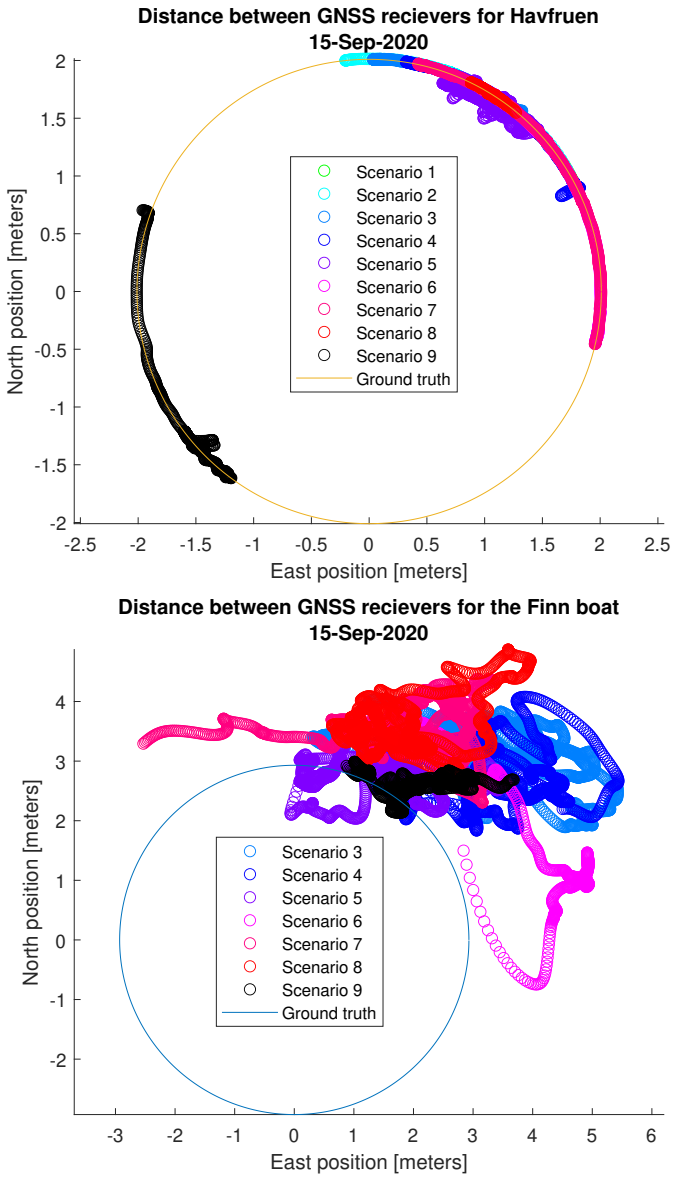


Figure 3.14: GNSS distance errors for both target ships. Using the hand measured distance between the antennas from Figure 3.7 as ground truth, we can see how well the GNSS receivers performed during each scenario

with $|Q_r^n|$ being the ground truth distance between the GNSS antennas described in Section 3.3, valued as $2.93m$ for Finn and $2.01m$ for Havfruen. Figure 3.15 shows that the error is most often at centimeter precision and occasionally decimeter precision for Havfruen, while for the Finn boat (Figure 3.16) its meter precision.

As for the ownship milliAmpere, the *Hemisphere VS-330* did not provide any raw GNSS data nor did it give positional information for each GNSS antenna on the ship, only the quality of measurement from the manufacturers GPGGA messages. This told if the received position was in the *float* or *fixed* state, assuming the error would be of decimeter precision for the first and centimeter precision for the latter, seen in Figure 3.17.

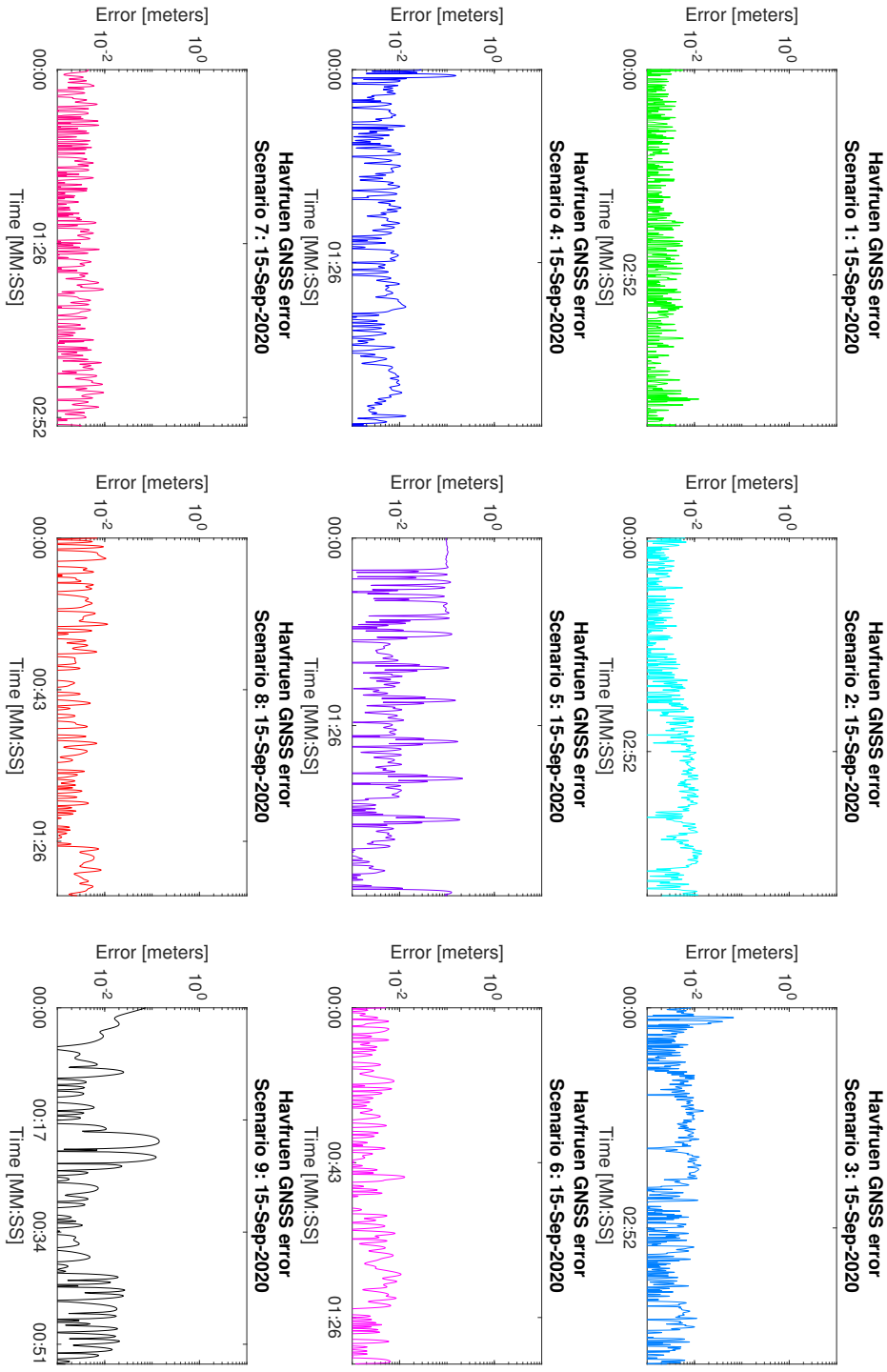


Figure 3.15: Relative error between GNSS recivers for Havfruen

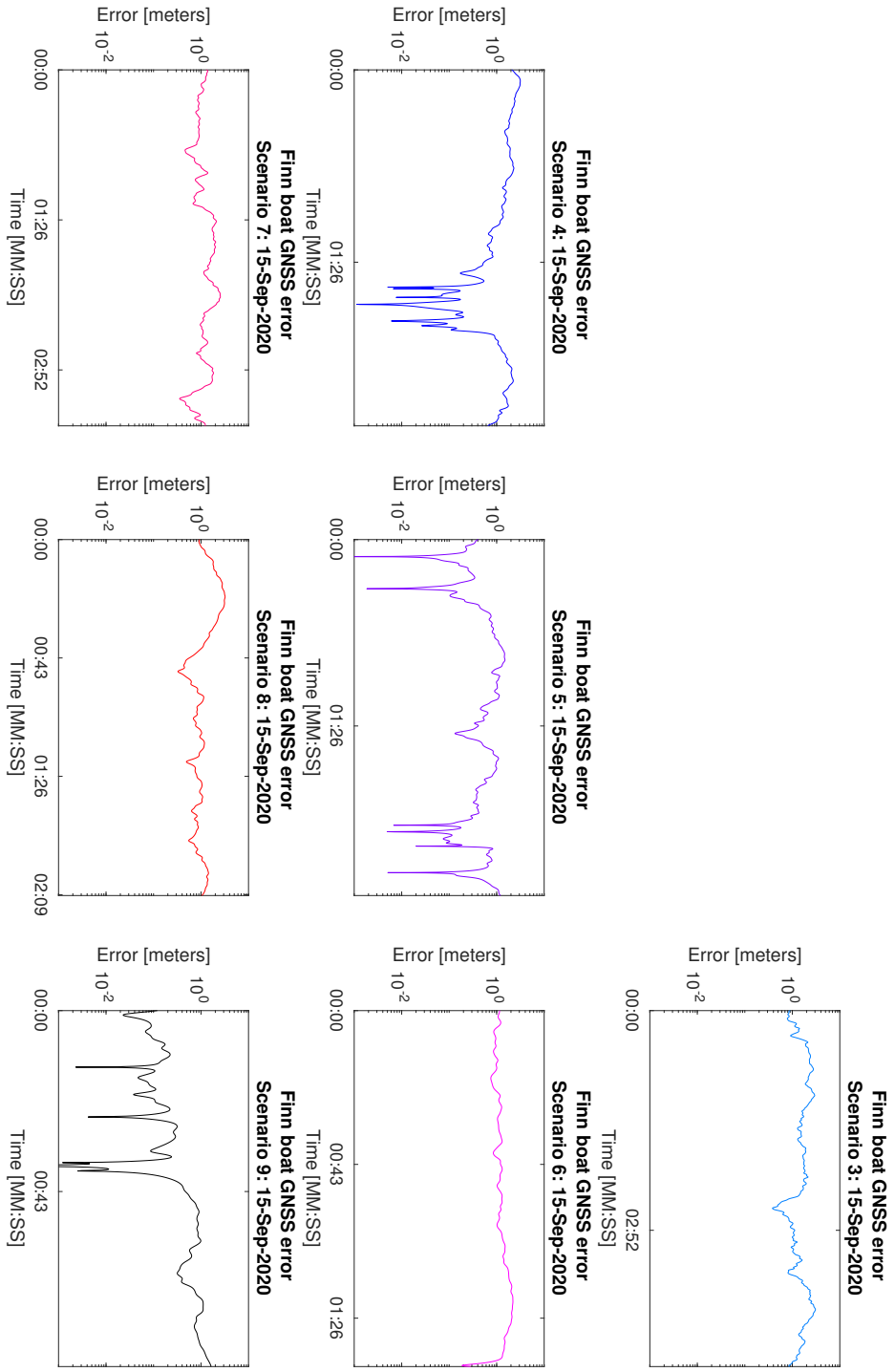


Figure 3.16: Relative error between GNSS recivers for Finn

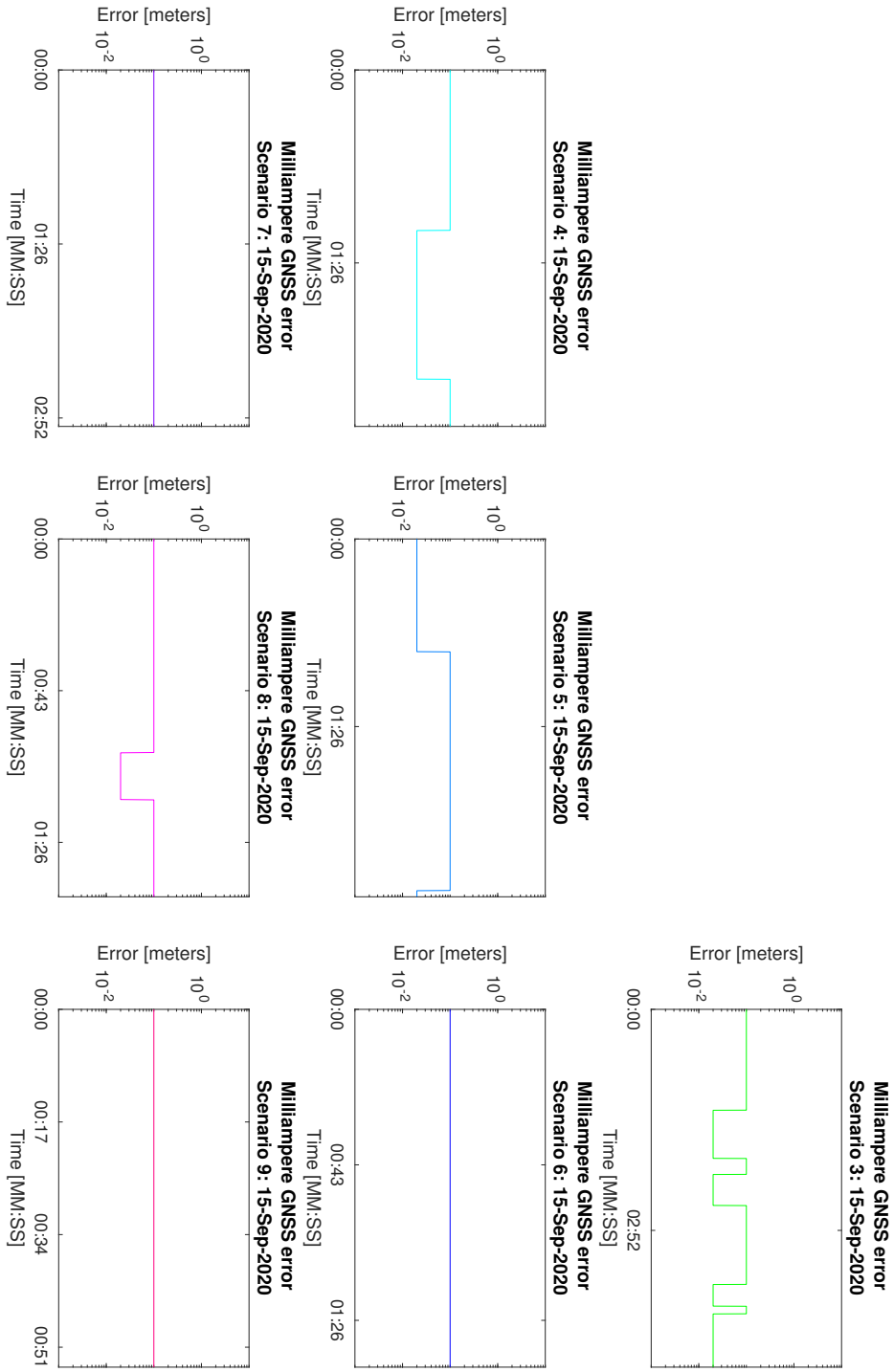


Figure 3.17: GNSS error from quality states for milliAmpere

Chapter 4

Synthetic dataset

Figure 4.1: Synthetic reproduction of the real dataset illustration in Figure 3.1. Image shows scenario 4

The synthetic dataset uses the environment and sensors created in the authors specialization project [11] and later in the published paper of Autoferry Gemini [12]. Here the lidar data and RGB camera images are generated from using depth-buffers provided by the Unity game engine. To continue this work, we will be focusing on improving the lidar model to address the error limitations experienced in the prior research. Since this error comes from using depth-buffers, we begin with deriving techniques that can improve and quantify the error for all sensor types depending on the buffer. Using this as a baseline and using sensor information from Section 3.2, Section 4.2 goes through the individual sensor models choosing simulation parameters that minimizes the error and replicates the real EMR sensors as demonstrated in Figure 4.1.

4.1 Depth-buffers for sensor modeling

The papers regarding *Autoferry Gemini* [12, 11] described how modeling of several EMR sensors was done using computer graphic techniques. This included among others lidar and radar sensors, using point cloud data generated from the GPU's *depth-buffer*. However, the method for creating the point cloud, distorted the individual points depending on their position in the cloud. This resulted in an error which was named *the beam shape error* in the papers. In this chapter, a technique that removes this error is introduced in the second section, followed by a method for determining the numerical error from using depth-buffers. The first section will be dedicated in understanding the basics of how depth-buffers are formed

4.1.1 Introduction to Depth-buffers

In real-time rendering, techniques that balances computational complexity with fidelity is heavily sought after. Multiple methods have been seen over the years, but only graphic-buffering (G-buffering) have been extensively used in real-time rendering applications such as video games. G-buffering functions by creating multiple images containing pixel data such as colours and surface normals called *buffers* (Figure 4.2), before combining them through individual computations to form a final rendered image. One of these buffers is named *the depth-buffer* as it tells how far the camera center is from the object surfaces being rendered.

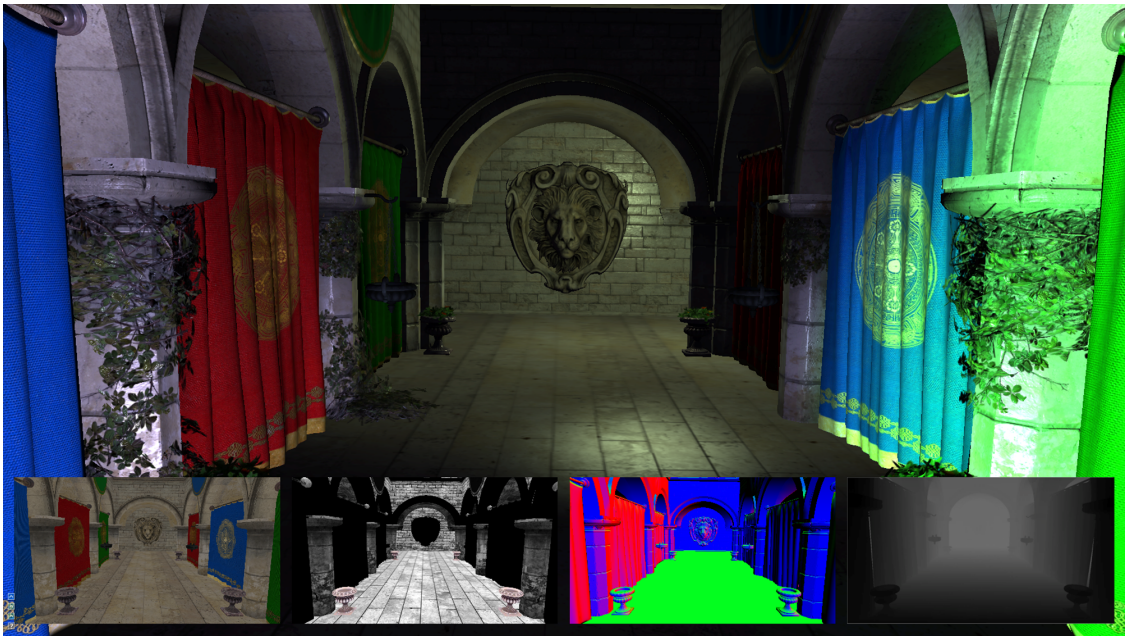


Figure 4.2: The G-buffer contains several buffers seen as images at the bottom. The last buffer is known as the depth-buffer

The process of creating a depth-buffer in video games, begins with transforming individual *vertices* Q among different spaces, which will here be noted by superscript, e.g Q^v for a vertex in view space. Vertices define surface triangles seen in 3D geometry known as a *mesh*, which is created in *Object Space* by various 3D modeling

softwares. The first transformation happens by converting the 3D mesh vertices $\mathrm{Q^o}$ over from object space to the video games *World Space* where scenarios, events and physics takes place. These spacial transformations happens throught matrices, which will use the convention $\mathbf{T}^{to\ space}_{from\ space}$ to tell what spaces are involved and in what direction the transformation is happening:

$$\mathrm{Q^w} = \mathbf{T}^{\mathrm{w}}_{\mathrm{i}}\mathrm{Q^i} \tag{4.1}$$

As world space is a state space of multiple 3D models, a *camera model* is needed to see a portion of it. This begins with having a view space matrix $\mathbf{T}^{\mathrm{v}}_{\mathrm{w}}$ that transforms the vertices from world space to the cameras local coordinate system called *View Space*

$$\mathrm{Q^v} = \mathbf{T}^{\mathrm{v}}_{\mathrm{w}}\mathrm{Q^w} \tag{4.2}$$

Within this space a pinhole camera model is introduced, creating a *frustum* consisting of a near plane and far plane with distances n and f respectively from origin. From Figure 4.3, it is seen that these distances follow the relationship $0 < n < f$. In Unity, the near plane is also known as the image plane, consisting of $\mathrm{N_{c,w}} \times \mathrm{N_{c,h}}$ pixels the final image is rendered to.

To give the camera perspective, the frustum is scaled by a vertical field of view $\mathrm{VFOV_c}$ and a horizontal field of view $\mathrm{HFOV_c}$, where the relationship between the number of pixels $\mathrm{N_{c,h}}, \mathrm{N_{c,w}}$ and the field of view comes from the images aspect ratio:

$$a = \frac{\mathrm{N_{c,w}}}{\mathrm{N_{c,h}}} = \frac{\frac{1}{2}(f-n)\tan(\frac{\mathrm{HFOV_c}}{2})}{\frac{1}{2}(f-n)\tan(\frac{\mathrm{VFOV_c}}{2})} = \frac{\tan(\frac{\mathrm{HFOV_c}}{2})}{\tan(\frac{\mathrm{VFOV_c}}{2})}. \tag{4.3}$$

The camera's frustum serves two purposes: First is using it as a filter to tell if a triangle should be rendered by the camera model or not, by checking the dot product F_{prod} between the triangle's vertices $\mathrm{Q^v}$ and the frustum's surface normals F_{normal}:

$$F_{prod} = \mathrm{Q^v} \cdot F_{normal}. \tag{4.4}$$

If the vertex is within all the 6 planes of the frustum, e.i the dot product with each plane is negative, the vertex passes the test. If not, the vertices triangle is handled through a process called *clipping*, where the triangle is split by the frustums planes or filtered out if all it's vertices lies outside the frustum. The second purpose the frustum serves, is by providing parameters to the projection matrix $\mathbf{T}^{\mathrm{i}}_{\mathrm{v}}$ that transforms all the vertices within the camera's frustum to *Image Space*:

$$\mathbf{T}^{\mathrm{i}}_{\mathrm{v}} := \begin{bmatrix} \frac{\mathrm{N_{c,h}}}{\mathrm{N_{c,w}}}\cot(\frac{\mathrm{VFOV_c}}{2}) & 0 & 0 & 0 \\ 0 & \cot(\frac{\mathrm{VFOV_c}}{2}) & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -2\frac{fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \tag{4.5}$$

$$\begin{bmatrix} \bar{\mathrm{Q}}^{\mathrm{i}} \\ w^{\mathrm{i}} \end{bmatrix} = \mathbf{T}^{\mathrm{i}}_{\mathrm{v}} \begin{bmatrix} \mathrm{Q^v} \\ 1 \end{bmatrix}, \tag{4.6}$$

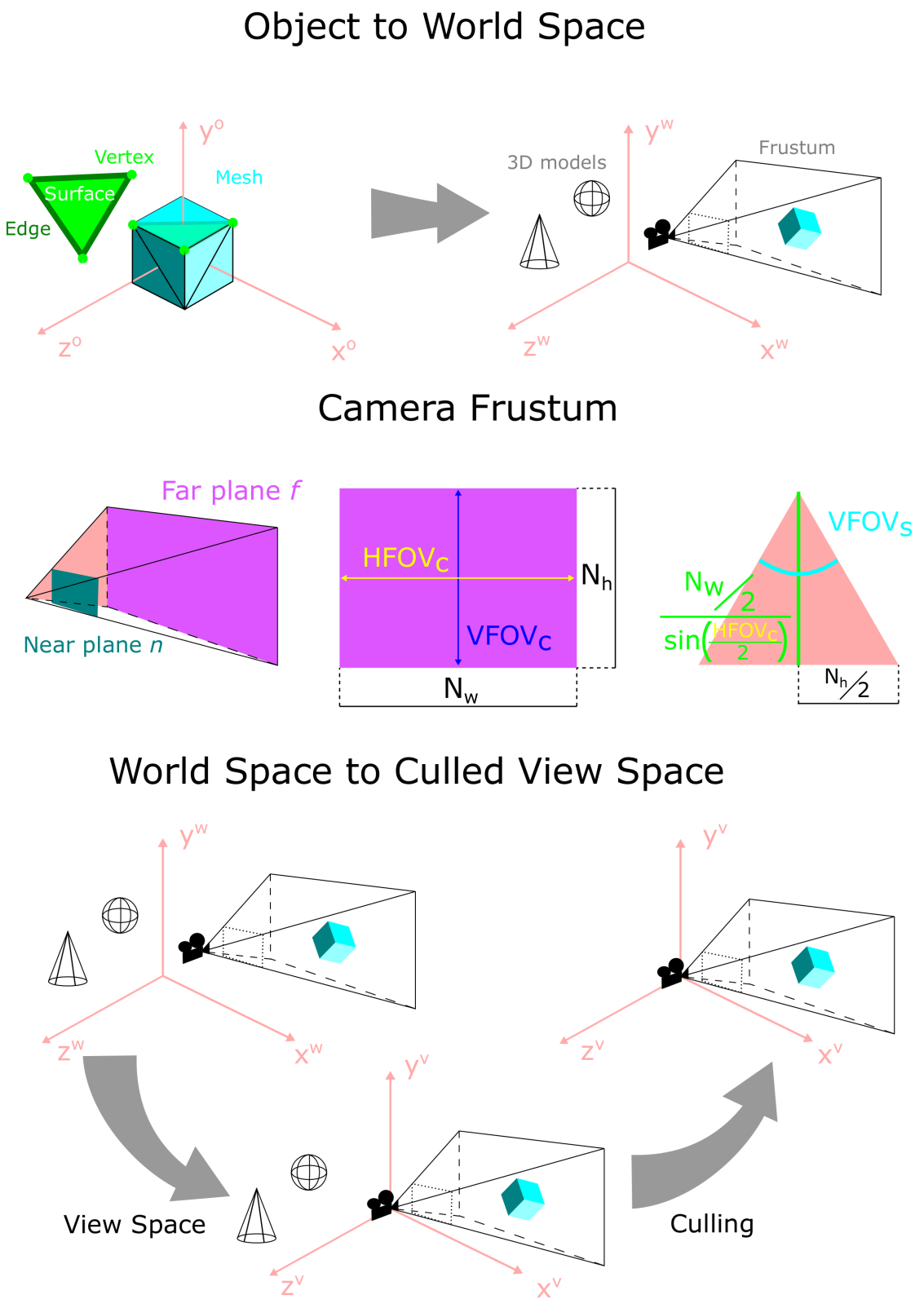


Figure 4.3: The process of going from a 3D model created in an arbitrary modeling software, to a scene view created by a synthetic camera.

Contrary to the other projections seen so far, the transformation into image space creates homogeneous coordinates, where w^{i} is a normalization variable used to get the normalized image coordinates for each vertex:

$$\mathrm{Q}^{\mathrm{i}} = \frac{1}{w^{\mathrm{i}}}\bar{\mathrm{Q}}^{\mathrm{i}} = [x^{\mathrm{i}}, y^{\mathrm{i}}, z^{\mathrm{i}}]^T, \quad x^{\mathrm{i}}, y^{\mathrm{i}}, z^{\mathrm{i}} \in [-1, 1]. \tag{4.7}$$

The last operation is to rasterize each individual triangle's surface in image space to *Discrete Image Space* through a process called *scan conversion* (Figure 4.4). Here interpolation between each vertex position happens using a *scan line algorithm* that rasterizes the triangle's surface into pixels containing depth information.

This is where the depth-buffer technique comes inn, which first creates a buffer containing the pixel value $\mathrm{N}_{\mathrm{c,d}}$ for the far plane. This value is known as the depth precision, usually being 16, 24 or 32 bit depending on the graphics API the operating system supports. As the scan line algorithm creates pixels, it checks if the pixel value it just created is less than the corresponding pixel value currently in the depth buffer. If this is true, the pixel value is stored in the depth-buffer, if not, the pixel value is tossed since a previous rasterized triangle has a pixel value closer to the camera. This happens to each individual triangle, where the mentioned logic guarantees that the final buffer contains pixel values to the closest surfaces, without any need of sorting the triangles beforehand. From this, every vertices, triangle surfaces and unchanged depth-buffer pixels is converted into discrete coordinates $x^{\mathrm{d}} \in [0, \mathrm{N}_{\mathrm{c,w}} - 1]$, $y^{\mathrm{d}} \in [0, \mathrm{N}_{\mathrm{c,h}} - 1]$, $z^{\mathrm{d}} \in [0, \mathrm{N}_{\mathrm{c,d}} - 1]$:

$$\mathrm{Q}^{\mathrm{d}} = \begin{bmatrix} x^{\mathrm{d}} \\ y^{\mathrm{d}} \\ z^{\mathrm{d}} \end{bmatrix} = \frac{1}{2}\begin{bmatrix} \mathrm{N}_{\mathrm{c,w}} & 0 & 0 \\ 0 & \mathrm{N}_{\mathrm{c,h}} & 0 \\ 0 & 0 & \mathrm{N}_{\mathrm{c,d}} \end{bmatrix}(1 + \mathrm{Q}^{\mathrm{i}}) = \frac{1}{2}\mathbf{N}_{\mathrm{c}}(1 + \mathrm{Q}^{\mathrm{i}}), \quad \mathrm{Q}^{\mathrm{d}} \in \mathbb{N}, \tag{4.8}$$

introducing the camera resolution matrix $\mathbf{N}_{\mathrm{c}}$ to contain all the resolution elements. This can also be written in component form by first expanding the terms in Equation 4.5 and 4.6:

$$\begin{aligned} \begin{bmatrix} \bar{\mathrm{Q}}^{\mathrm{i}} \\ w^{\mathrm{i}} \end{bmatrix} = \begin{bmatrix} \bar{x}^{\mathrm{i}} \\ \bar{y}^{\mathrm{i}} \\ \bar{z}^{\mathrm{i}} \\ w^{\mathrm{i}} \end{bmatrix} &= \begin{bmatrix} \frac{\mathrm{N}_{\mathrm{c,h}}}{\mathrm{N}_{\mathrm{c,w}}}\cot(\frac{\mathrm{VFOV}_{\mathrm{c}}}{2}) & 0 & 0 & 0 \\ 0 & \cot(\frac{\mathrm{VFOV}_{\mathrm{c}}}{2}) & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -2\frac{fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}\begin{bmatrix} x^{\mathrm{v}} \\ y^{\mathrm{v}} \\ z^{\mathrm{v}} \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} \frac{\mathrm{N}_{\mathrm{c,h}}}{\mathrm{N}_{\mathrm{c,w}}}\cot(\frac{\mathrm{VFOV}_{\mathrm{c}}}{2})x^{\mathrm{v}} \\ \cot(\frac{\mathrm{VFOV}_{\mathrm{c}}}{2})y^{\mathrm{v}} \\ -\frac{f+n}{f-n}z^{\mathrm{v}} - 2\frac{fn}{f-n} \\ -z^{\mathrm{v}} \end{bmatrix}, \end{aligned}$$

substituting these elements into Equation 4.7 gives:

$$\mathrm{Q}^{\mathrm{i}} = \frac{1}{w^{\mathrm{i}}}\begin{bmatrix} \bar{x}^{\mathrm{i}} \\ \bar{y}^{\mathrm{i}} \\ \bar{z}^{\mathrm{i}} \end{bmatrix} = \begin{bmatrix} -\frac{\mathrm{N}_{\mathrm{c,h}}}{\mathrm{N}_{\mathrm{c,w}}}\cot(\frac{\mathrm{VFOV}_{\mathrm{c}}}{2})\frac{x^{\mathrm{v}}}{z^{\mathrm{v}}} \\ -\cot(\frac{\mathrm{VFOV}_{\mathrm{c}}}{2})\frac{y^{\mathrm{v}}}{z^{\mathrm{v}}} \\ \frac{f+n}{f-n} + 2\frac{fn}{f-n}\frac{1}{z^{\mathrm{v}}} \end{bmatrix},$$

and finally, substituting this into Equation 4.8 gives:

$$\begin{aligned}
\mathrm{Q}^{\mathrm{d}} &= \frac{1}{2}\begin{bmatrix} \mathrm{N_{c,w}} & 0 & 0 \\ 0 & \mathrm{N_{c,h}} & 0 \\ 0 & 0 & \mathrm{N_{c,d}} \end{bmatrix}\left(1 + \begin{bmatrix} -\frac{\mathrm{N_{c,h}}}{\mathrm{N_{c,w}}}\cot(\frac{\mathrm{VFOV_c}}{2})\frac{x^{\mathrm{v}}}{z^{\mathrm{v}}} \\ -\cot(\frac{\mathrm{VFOV_c}}{2})\frac{y^{\mathrm{v}}}{z^{\mathrm{v}}} \\ \frac{f+n}{f-n} + 2\frac{fn}{f-n}\frac{1}{z^{\mathrm{v}}} \end{bmatrix}\right) \\
&= \begin{bmatrix} \frac{\mathrm{N_{c,w}}}{2}\left(1 - \frac{\mathrm{N_{c,h}}}{\mathrm{N_{c,w}}}\cot\left(\frac{\mathrm{VFOV_c}}{2}\right)\frac{x^{\mathrm{v}}}{z^{\mathrm{v}}}\right), \\ \frac{\mathrm{N_{c,h}}}{2}\left(1 - \cot\left(\frac{\mathrm{VFOV_c}}{2}\right)\frac{y^{\mathrm{v}}}{z^{\mathrm{v}}}\right), \\ \frac{\mathrm{N_{c,d}}}{2}\left(1 + \frac{f+n}{f-n} + \frac{2fn\mathrm{N_{c,d}}}{f-n}\frac{1}{z^{\mathrm{v}}}\right) \end{bmatrix}.
\end{aligned}$$

Written in component form as:

$$\begin{aligned}
x^{\mathrm{d}} &= \frac{\mathrm{N_{c,w}}}{2}\left(1 - \frac{\mathrm{N_{c,h}}}{\mathrm{N_{c,w}}}\cot\left(\frac{\mathrm{VFOV_c}}{2}\right)\frac{x^{\mathrm{v}}}{z^{\mathrm{v}}}\right), \\
y^{\mathrm{d}} &= \frac{\mathrm{N_{c,h}}}{2}\left(1 - \cot\left(\frac{\mathrm{VFOV_c}}{2}\right)\frac{y^{\mathrm{v}}}{z^{\mathrm{v}}}\right), \\
z^{\mathrm{d}} &= \frac{\mathrm{N_{c,d}}}{2}\left(1 + \frac{f+n}{f-n} + \frac{2fn\mathrm{N_{c,d}}}{f-n}\frac{1}{z^{\mathrm{v}}}\right).
\end{aligned} \tag{4.9}$$

Q^d is know as the depth-buffer coordinates, but comes from the GPU in a normalized format, giving us the 2D array $Z_{\mathrm{c}}^{\mathrm{d}}(x^{\mathrm{d}}, y^{\mathrm{d}}) = \frac{z^{\mathrm{d}}}{\mathrm{N_{c,d}}} \in [0, 1]$ hereafter know as *the camera depth buffer*. As its obvious that the depth buffer values and its parameters are all in a discrete Cartesian image space, we drop the superscript for the parameters: $Z_{\mathrm{c}}^{\mathrm{d}}(x, y) = Z_{\mathrm{c}}^{\mathrm{d}}(x^{\mathrm{d}}, y^{\mathrm{d}})$.

Contrary to offline renderers which relies on sending a ray through every pixel in the rendered image (ray tracing), depth-buffering relies on using the individual triangles in the scene instead. The motivation for this, is that the the number of triangles in view space, are usually less than the number of pixels in an image. E.g a typical scene in computer games displays 50 000 - 100 000 triangles at a time while the number of pixels in a HD image is 1920x1080 = 2 073 600. This allows rendering techniques that utilizes depth-buffering to use larger image formats and faster renderings. In addition, GPUs uses dedicated *graphics accelerators* that do both the transformations and scan conversion described in this section directly in hardware, pushing the real-time performances even further.

As a final remark, the vertices that have been discussed do not only need to contain positional data, but can also contain data regarding textures such as colours and normal maps. Because of this, the interpolation technique used in the scan conversion, is not only used to create the depth-buffer, but all data associated with the vertices, creating all the buffers seen in Figure 4.2.

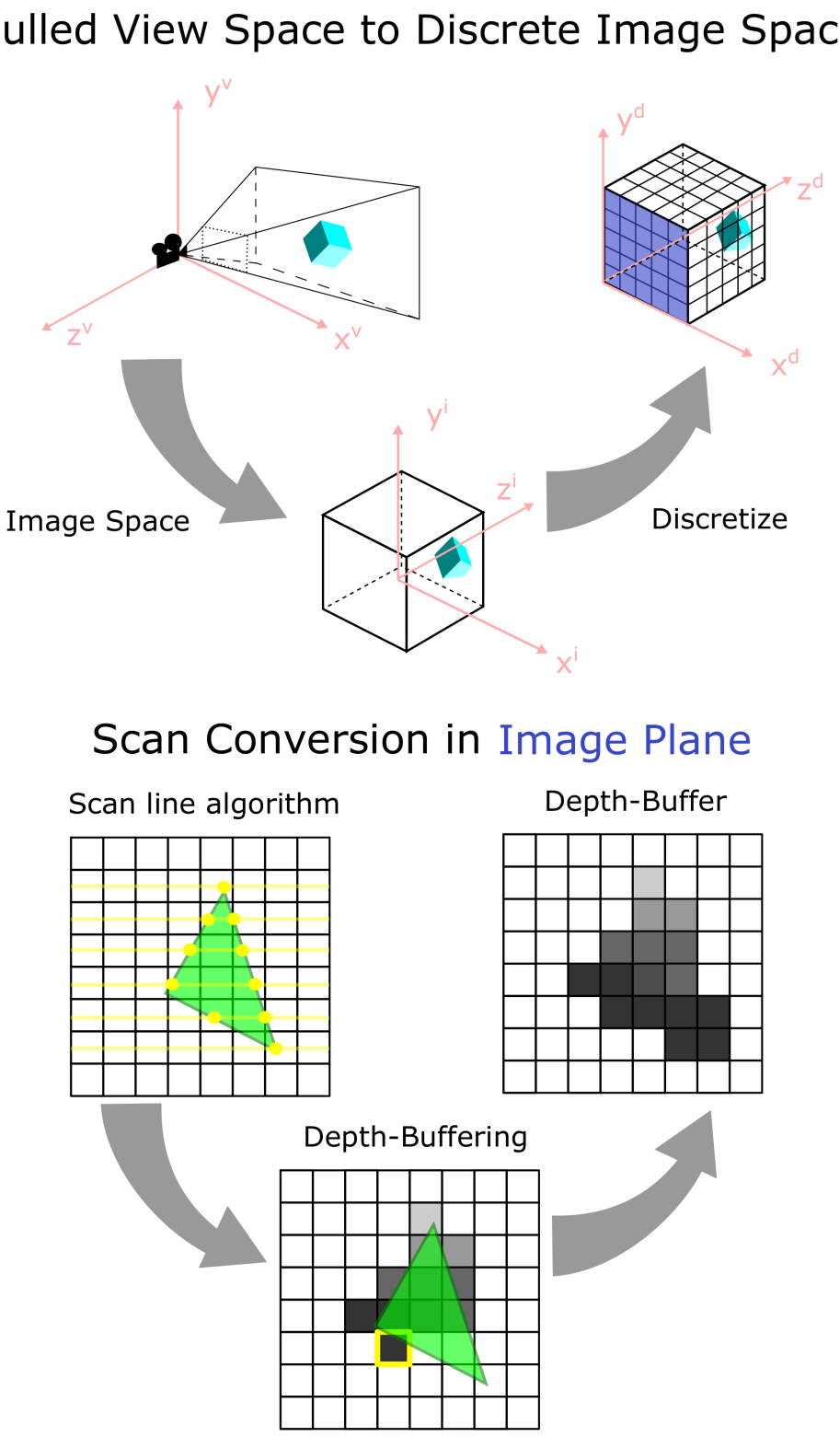


Figure 4.4: The process of going from 3D models defined in a camera's view space, over to a depth buffer

4.1.2 Spherical projection filter

As scene depth can be represented with depth-buffers in discrete Cartesian coordinates, sensor models that uses other systems need approximation techniques to still be able to use the depth-buffer. Autoferry Gemini [12] showed that an approximation to spherical coordinates could be done by distributing N_c virtual cameras evenly across the sensors azimuth axis (Figure 4.5), rendering individual depth-buffers $Z_c^d(x, y)$. However, due to the discrete image space needed for scan conversion, depth-increments propagated planarly relative to the image plane rather than spherically from the camera origin, leading to a beam shape error between the two coordinate systems depending on N_c. The results showed that increasing N_c decreased the real-time performance, in addition to diminishing returns for lowering the error. To improve this, rather than relying on approximating spherical coordinates by increasing N_c, a precomputed *spherical projection filter* telling the best fit pixels from the depth-buffers is created.

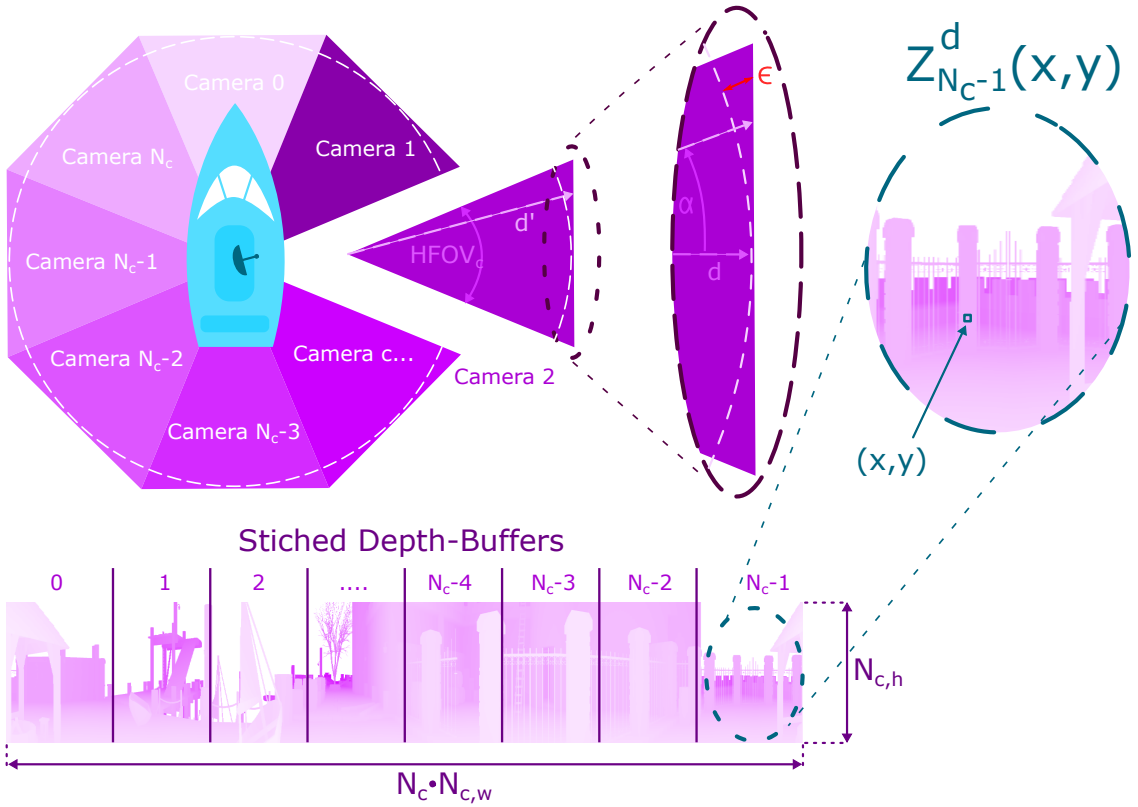


Figure 4.5: The Depth-buffer technique from Autoferry Gemini [12]. Virtual cameras was used to approximate cylindrical beams with convex regular polygons. Stitching the depth-buffers together formed a 2D depth array surrounding the sensor location, with a beam shape error ϵ rising along the cameras far plane.

The previous implementation [12] illustrated in Figure 4.5, uses polar coordinates to describe a circle with radius d and an angle α defining a reference model to quantify the beam shape error $\epsilon^{\mathrm{v}}_{beam}$. Since this error only considers the error spanned in a plane in view space, a more general approach is to instead use a sphere from the spherical coordinates:

$$\mathrm{Q}^{\mathrm{v}}_{\mathrm{s}} := \begin{bmatrix} x^{\mathrm{v}}_{\mathrm{s}} \\ y^{\mathrm{v}}_{\mathrm{s}} \\ z^{\mathrm{v}}_{\mathrm{s}} \end{bmatrix} = r \begin{bmatrix} -sin(\theta)cos(\phi) \\ sin(\phi) \\ -cos(\theta)cos(\phi) \end{bmatrix}, \tag{4.10}$$

using latitude ϕ, longitude θ instead of α and radius r instead of d as parameterization variables. Note that this is the same equation as the coordinates of the lidar points in Equation 3.1, with the difference being *space* and *type* of the vector.

Further, the previous implementation shows that the reference model (4.10) could be used to quantify the beam error by observing that the depth buffers forms a convex regular polygon noted as d' seen in Figure 4.5, giving the error $\epsilon^{\mathrm{v}}_{beam} = d' - d$ when studying one of the polygon lines. This can be generalized into 3D aswell, where we re-define d' to be a vector spanning the view space coordinates calculated from the depth buffer demonstrated in [12]. In addition, we re-define d to be the spherical reference model $d = \mathrm{Q}^{\mathrm{v}}_{\mathrm{s}}$.

The reference model can however be used for other purposes than just quantifying the beam shape error. We can also use it to selectively choose the best fit spherical coordinates from the depth buffers Cartesian coordinates by $d' = d$. This however only applies when mapping from Cartesian to spherical coordinates in the continuous case. Since we saw that the depth buffer went through a rasterization process in Section 4.1.1, such a best fit filter would in worst case give us $d' = d + \epsilon^{\mathrm{v}}_{\mathrm{s}}$ where $\epsilon^{\mathrm{v}}_{\mathrm{s}}$ is the numeric error from approximating the reference model from a depth buffer. This gives us the beam shape error:

$$\epsilon^{\mathrm{v}}_{beam} = d' - d = \epsilon^{\mathrm{v}}_{\mathrm{s}}, \tag{4.11}$$

leaving only the numerical error from the depth buffer's rasterization. This error will be discussed further in Section 4.1.3, for now we will see how such a filter can be made.

We begin with defining a finite quantity of vertices that are going to be in the point cloud. $\mathrm{N}_{\mathrm{s,w}}$ will the resolution of a horizontal sweep, and $\mathrm{N}_{\mathrm{s,h}}$ will be the resolution in the vertical direction commonly know as "lines" or number of lasers in lidar sensors. This gives a total of $\mathrm{N}_{\mathrm{s,h}} \times \mathrm{N}_{\mathrm{s,w}}$ points in the cloud. These vertices will be spread across a section of $\mathrm{Q}^{\mathrm{v}}_{\mathrm{s}}$, defined by the field of views $\mathrm{HFOV}_{\mathrm{s}}$ and $\mathrm{VFOV}_{\mathrm{s}}$ similar to what we saw for $\mathrm{HFOV}_{\mathrm{c}}$ and $\mathrm{VFOV}_{\mathrm{c}}$ in section 4.1.1. In addition the vertices will also span a depth between a synthetic cameras farplane f and near plane n. Using θ, ϕ and r as parameterization variables we get:

$$\begin{aligned} \theta &= \mathrm{HFOV}_{\mathrm{s}} \left(\frac{\mathrm{N}_{\mathrm{c}}}{\mathrm{N}_{\mathrm{s,w}}} n_\theta - \frac{1}{2} \right) \in \left[-\frac{\mathrm{HFOV}_{\mathrm{s}}}{2}, \frac{\mathrm{HFOV}_{\mathrm{s}}}{2} \right], \\ \phi &= \mathrm{VFOV}_{\mathrm{s}} \left(\frac{n_\phi}{\mathrm{N}_{\mathrm{s,h}}} - \frac{1}{2} \right) \in \left[-\frac{\mathrm{VFOV}_{\mathrm{s}}}{2}, \frac{\mathrm{VFOV}_{\mathrm{s}}}{2} \right], \\ r &= (f - n) \frac{n_r}{\mathrm{N}_{\mathrm{c,d}}} + n \in [n, f], \end{aligned} \tag{4.12}$$

using the discretized elements n_ϕ for latitude, n_θ for longitude and n_r for radius. These are distributed with the resolutions we just defined, in addition to the already defined depth puffer precision from Section 4.1.1:

$$n_\theta \in \left[0, \frac{\mathrm{N_{s,w}}}{\mathrm{N_c}}\right),\ n_\phi \in [0, \mathrm{N_{s,h}}),\ n_r \in [0, \mathrm{N_{c,d}}),\quad n_\theta,\, n_\phi,\, n_r \in \mathbb{N}. \tag{4.13}$$

To ensure that no pixels in the depth-buffers are wasted, the field of views of the point cloud and synthetic cameras are adjusted to mach each other:

$$\mathrm{HFOV_c} = \mathrm{HFOV_s} = \frac{2\pi}{\mathrm{N_c}}. \tag{4.14}$$

where $\frac{2\pi}{\mathrm{N_c}}$ comes from distributing the cameras evenly across the azimuth axis of the point cloud such as in Figure 4.5. However, since the camera frustum have a different vertical field of view in the middle of the far plane then on the edges as seen in Figure 4.3, the vertical field of view needs to be adjusted relative to the point clouds field of view. We do this by using the reciprocal of the aspect ratio from Equation 4.3:

$$\frac{1}{a} = \frac{\mathrm{N_{c,h}}}{\mathrm{N_{c,w}}} = \frac{\tan(\frac{\mathrm{VFOV_c}}{2})}{\tan(\frac{\mathrm{HFOV_c}}{2})}, \tag{4.15}$$

in addition to a trigonometric relationship between the angle spanned by one of the far planes vertical edges relative to the camera origin, forming a isosceles triangle seen in Figure 4.3:

$$\frac{\mathrm{N_{c,h}}}{\mathrm{N_{c,w}}} = \frac{\tan(\frac{\mathrm{VFOV_s}}{2})}{\sin(\frac{\mathrm{HFOV_s}}{2})}. \tag{4.16}$$

These relationships gives a function for adjusting the camera's vertical field of view:

$$\mathrm{VFOV_c} = 2\arctan\left(\frac{\tan(\frac{\mathrm{VFOV_s}}{2})}{\cos(\frac{\mathrm{HFOV_s}}{2})}\right). \tag{4.17}$$

To define the spherical projection filter, we first need to project the view space vertices $\mathrm{Q_s^v}$ into discrete image space using equation 4.9:

$$\begin{aligned}
x_\mathrm{s}^\mathrm{d} &= \frac{\mathrm{N_{c,w}}}{2}\left(1 - \frac{\mathrm{N_{c,h}}}{\mathrm{N_{c,w}}}\cot\left(\frac{\mathrm{VFOV_c}}{2}\right)\frac{x_\mathrm{s}^\mathrm{v}}{z_\mathrm{s}^\mathrm{v}}\right),\\
y_\mathrm{s}^\mathrm{d} &= \frac{\mathrm{N_{c,h}}}{2}\left(1 - \cot\left(\frac{\mathrm{VFOV_c}}{2}\right)\frac{y_\mathrm{s}^\mathrm{v}}{z_\mathrm{s}^\mathrm{v}}\right),\\
z_\mathrm{s}^\mathrm{d} &= \frac{\mathrm{N_{c,d}}}{2}\left(1 + \frac{f+n}{f-n} + \frac{2fn\mathrm{N_{c,d}}}{f-n}\frac{1}{z_\mathrm{s}^\mathrm{v}}\right).
\end{aligned}$$

Where we can substitute inn the reference model (4.10) to see how it gets projected into a synthetic cameras depth buffer:

$$\begin{aligned}
x_\mathrm{s}^\mathrm{d} &= \frac{\mathrm{N_{c,w}}}{2}\left(1 - \frac{\mathrm{N_{c,h}}}{\mathrm{N_{c,w}}}\cot\left(\frac{\mathrm{VFOV_c}}{2}\right)\tan(\theta)\right),\\
y_\mathrm{s}^\mathrm{d} &= \frac{\mathrm{N_{c,h}}}{2}\left(1 + \cot\left(\frac{\mathrm{VFOV_c}}{2}\right)\frac{\tan(\phi)}{\cos(\theta)}\right),\\
z_\mathrm{s}^\mathrm{d} &= \frac{\mathrm{N_{c,d}}}{2}\left(1 + \frac{f+n}{f-n} - \frac{2fn\mathrm{N_{c,d}}}{f-n}\frac{1}{r\cos(\theta)\cos(\phi)}\right).
\end{aligned} \tag{4.18}$$

using Equation 4.12 and 4.13 for θ, ϕ, r, 4.17 for $\mathrm{VFOV_c}$, and 4.16 for $\mathrm{N_{c,w}}$, such that the depth buffer is described by $\mathrm{N_{s,w}}$, $\mathrm{N_{s,h}}$, $\mathrm{N_c}$, $\mathrm{N_{c,h}}$, n_θ, n_ϕ, n_r and $\mathrm{VFOV_s}$. From this, we define *the point cloud depth buffer* in the same manner as we did for Z_c^d in Section 4.1.1:

$$Z_\mathrm{s}^\mathrm{d}(x, y) := Z_\mathrm{s}^\mathrm{d}(x_\mathrm{s}^\mathrm{d}, y_\mathrm{s}^\mathrm{d}) = \frac{z_\mathrm{s}^\mathrm{d}}{\mathrm{N_{c,d}}}. \tag{4.19}$$

Studying the parameter equations for Z_s^d, we see that (x,y) only relies on constant terms. This means that we can precompute a masked domain $\mathbb{M} \in \mathbb{N}^{\mathrm{N_{s,w}} \times \mathrm{N_{s,h}}} \subseteq \mathbb{N}^{\mathrm{N_{c,h}} \times \mathrm{N_{c,w}}}$ which defines *the spherical projection filter*, visualized in Figure 4.6. Using this domain creates a filtered version of the camera depth buffer since $Z_\mathrm{s}^\mathrm{d} = Z_\mathrm{c}^\mathrm{d}(x_\mathrm{s}^\mathrm{d}, y_\mathrm{s}^\mathrm{d})$. Since we are sampling from the camera depth-buffer, this requires that $\mathrm{N_{s,h}} \leq \mathrm{N_{c,h}}$ and $\mathrm{N_{s,w}} \leq \mathrm{N_{c,w}}$. As we will see in Section 4.1.3 $\mathrm{N_{c,h}}$ and $\mathrm{N_{c,w}}$ is preferred to be big to decrease the numerical error. However, since the spherical coordinate approximation is now handled by $\mathbb{M}$, $\mathrm{N_c}$ can be lowered to increase both performance and accuracy in comparison to what was shown in the previous version of Autoferry Gemini [12].

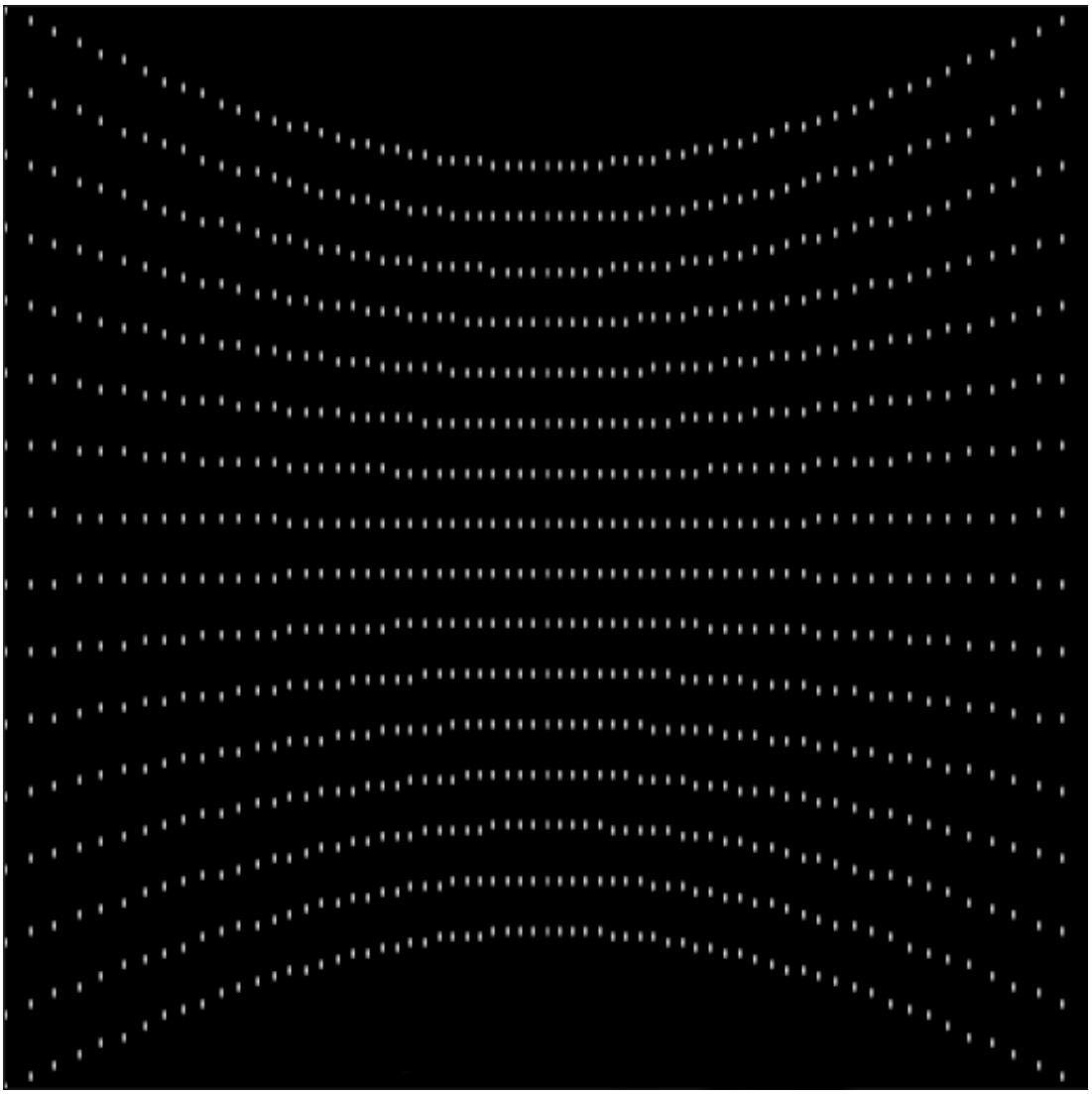


Figure 4.6: An example of a spherical projection filter using $N_{s,w} = 256$, $N_{s,h} = 16$ and $N_c = 4$. The filter functions as an image mask for the cameras depth-buffer, only allowing the white pixel positions to pass through.

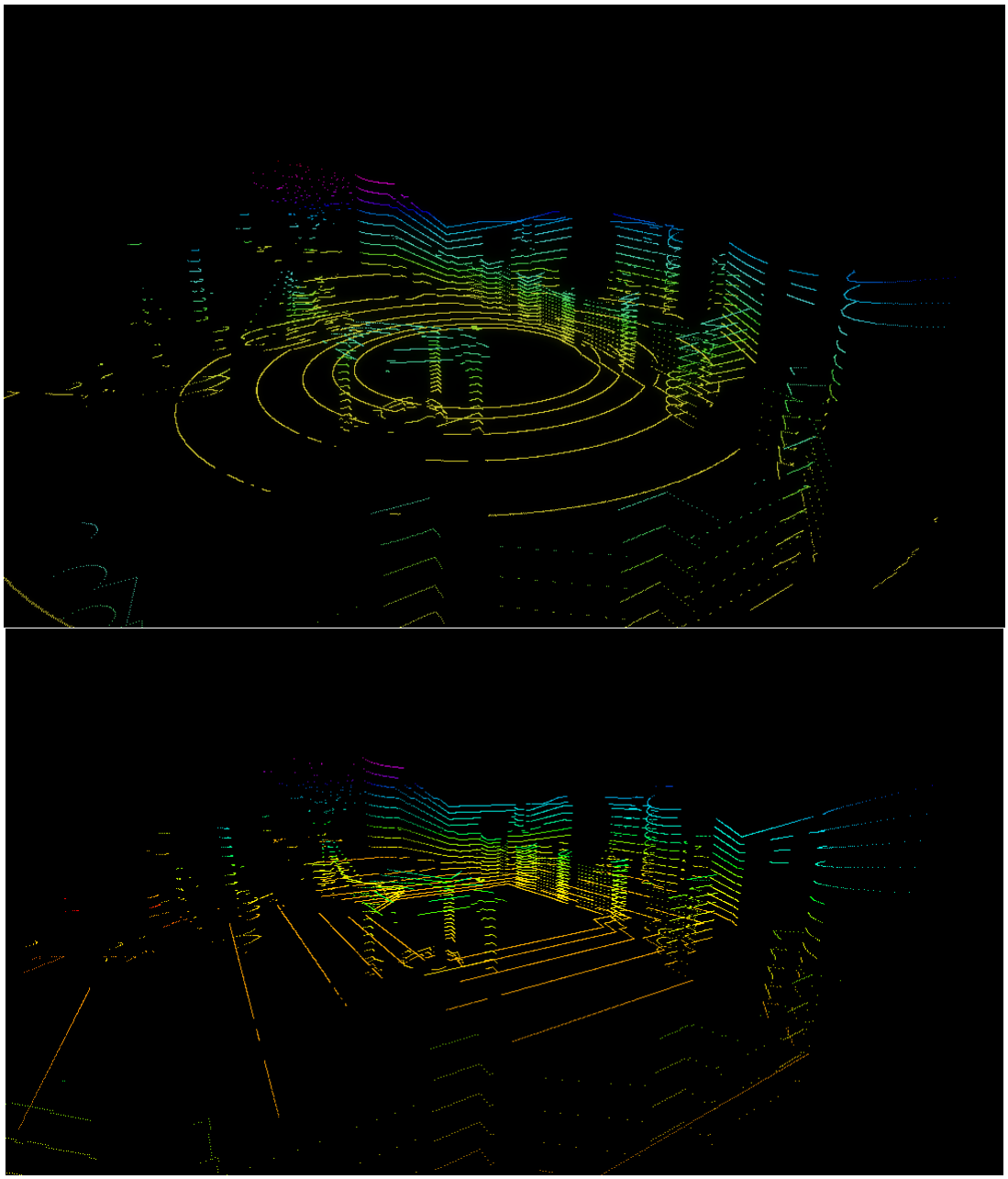


Figure 4.7: Comparison of the beam shape error for a Velodyne VLP-16 lidar model with and without the spherical projection filter in respective order. Both models used a N_c value of 4, with negligible differences in real-time performances. Using the spherical projection filter creates a significantly lower beam shape error

4.1.3 Numerical errors from Depth-Buffers

To be able to compare the performance between the spherical projection filter method contrary to the method used in [12], moreover tell how precise sensor models using depth-buffers are, an error function of the numerical approximation is needed. As the beam shape error [12] relied on the specific error of approximating a circle with polygons, a more general function is needed to account for the finite precision of width, height and range in depth-buffers. To do this, an analysis of the numerical error from transforming 3D coordinates in continuous view space to the discrete image space is needed.

We begin by tranforming a point in view space $\mathrm{Q^v}$ to a point in discrete image space $\mathrm{Q^d}$ using the Equations in 4.9:

$$
\begin{aligned}
x^{\mathrm{d}} &= \frac{\mathrm{N_{c,w}}}{2}\left(1 - \frac{\mathrm{N_{c,h}}}{\mathrm{N_{c,w}}}\cot\left(\frac{\mathrm{VFOV_c}}{2}\right)\frac{x^{\mathrm{v}}}{z^{\mathrm{v}}}\right),\\
y^{\mathrm{d}} &= \frac{\mathrm{N_{c,h}}}{2}\left(1 - \cot\left(\frac{\mathrm{VFOV_c}}{2}\right)\frac{y^{\mathrm{v}}}{z^{\mathrm{v}}}\right),\\
z^{\mathrm{d}} &= \frac{\mathrm{N_{c,d}}}{2}\left(1 + \frac{f+n}{f-n} + \frac{2fn\mathrm{N_{c,d}}}{f-n}\frac{1}{z^{\mathrm{v}}}\right).
\end{aligned}
$$

From these, errors in discrete image space can be made as a function of the arbitrary positions indexed as 1 and 0:

$$
\begin{aligned}
x^{\mathrm{d}}_{\epsilon} &= x^{\mathrm{d}}_1 - x^{\mathrm{d}}_0 = \cot\left(\frac{\mathrm{VFOV_c}}{2}\right)\frac{\mathrm{N_{c,h}}}{2}\left(\frac{z^{\mathrm{v}}_1 x^{\mathrm{v}}_0 - z^{\mathrm{v}}_0 x^{\mathrm{v}}_1}{z^{\mathrm{v}}_1 z^{\mathrm{v}}_0}\right),\\
y^{\mathrm{d}}_{\epsilon} &= y^{\mathrm{d}}_1 - y^{\mathrm{d}}_0 = \cot\left(\frac{\mathrm{VFOV_c}}{2}\right)\frac{\mathrm{N_{c,h}}}{2}\left(\frac{z^{\mathrm{v}}_1 y^{\mathrm{v}}_0 - z^{\mathrm{v}}_0 y^{\mathrm{v}}_1}{z^{\mathrm{v}}_1 z^{\mathrm{v}}_0}\right),\\
z^{\mathrm{d}}_{\epsilon} &= z^{\mathrm{d}}_1 - z^{\mathrm{d}}_0 = -\frac{fn\mathrm{N_{c,d}}}{f-n}\frac{z^{\mathrm{v}}_1 - z^{\mathrm{v}}_0}{z^{\mathrm{v}}_1 z^{\mathrm{v}}_0}.
\end{aligned}
\tag{4.20}
$$

As we are interested in errors happening in view space as a consequence of the finite precision in discrete image space, we set $x^{\mathrm{d}}_{\epsilon} = y^{\mathrm{d}}_{\epsilon} = z^{\mathrm{d}}_{\epsilon} = 1$. This is to study how much error in view space one step is in discrete image space corresponds to. Using this and the Equations in 4.20, the following can be shown:

$$
\begin{aligned}
x^{\mathrm{v}}_{\epsilon} &= x^{\mathrm{v}}_1 - x^{\mathrm{v}}_0 = \frac{z^{\mathrm{v}}_{\epsilon}}{z^{\mathrm{v}}_0}\left(x^{\mathrm{v}}_0 - \tan\left(\frac{\mathrm{VFOV_c}}{2}\right)\frac{2fn\mathrm{N_{c,d}}}{(f-n)\mathrm{N_{c,h}}}\right),\\
y^{\mathrm{v}}_{\epsilon} &= y^{\mathrm{v}}_1 - y^{\mathrm{v}}_0 = \frac{z^{\mathrm{v}}_{\epsilon}}{z^{\mathrm{v}}_0}\left(y^{\mathrm{v}}_0 - \tan\left(\frac{\mathrm{VFOV_c}}{2}\right)\frac{2fn\mathrm{N_{c,d}}}{(f-n)\mathrm{N_{c,h}}}\right),\\
z^{\mathrm{v}}_{\epsilon} &= z^{\mathrm{v}}_1 - z^{\mathrm{v}}_0 = -\frac{(z^{\mathrm{v}}_0)^2}{\mathrm{N_{c,d}}\frac{fn}{f-n} + z^{\mathrm{v}}_0}.
\end{aligned}
$$

This means that the error from using depth-buffers depends on the location in the camera's frustum. As this is general for all view space coordinates, we drop the subscript for the initial positions, to indicate that these error functions can be used by all sensor models using depth-buffers. In addition we introduce the positive

constants $C_1 = \mathrm{N_{c,d}}\frac{fn}{f-n}$ and $C_2 = 2\frac{C_1}{\mathrm{N_{c,h}}}\tan\left(\frac{\mathrm{VFOV_c}}{2}\right)$ to simplify the expressions:

$$\begin{aligned} x_\epsilon^{\mathrm{v}} &= \frac{z_\epsilon^{\mathrm{v}}}{z^{\mathrm{v}}}\left(x^{\mathrm{v}} - C_2\right), \\ y_\epsilon^{\mathrm{v}} &= \frac{z_\epsilon^{\mathrm{v}}}{z^{\mathrm{v}}}\left(y^{\mathrm{v}} - C_2\right), \\ z_\epsilon^{\mathrm{v}} &= -\frac{(z^{\mathrm{v}})^2}{C_1 + z^{\mathrm{v}}}. \end{aligned} \tag{4.21}$$

With this, the total error in the depth buffer for a given position can be expressed as an euclidean error function:

$$\begin{aligned} \epsilon^{\mathrm{v}}(x^{\mathrm{v}}, y^{\mathrm{v}}, z^{\mathrm{v}}) &= \sqrt{(x_\epsilon^{\mathrm{v}})^2 + (y_\epsilon^{\mathrm{v}})^2 + (z_\epsilon^{\mathrm{v}})^2} \\ &= \left|\frac{z_\epsilon^{\mathrm{v}}}{z^{\mathrm{v}}}\right| \sqrt{(x^{\mathrm{v}} - C_2)^2 + (y^{\mathrm{v}} - C_2)^2 + (z^{\mathrm{v}})^2} \\ &= \left|\frac{-z^{\mathrm{v}}}{C_1 + z^{\mathrm{v}}}\right| \sqrt{(x^{\mathrm{v}} - C_2)^2 + (y^{\mathrm{v}} - C_2)^2 + (z^{\mathrm{v}})^2}. \end{aligned} \tag{4.22}$$

Whose gradient can be expressed as:

$$\nabla\epsilon^{\mathrm{v}}(x^{\mathrm{v}}, y^{\mathrm{v}}, z^{\mathrm{v}}) = \left[\frac{\partial\epsilon^{\mathrm{v}}}{\partial x^{\mathrm{v}}}, \frac{\partial\epsilon^{\mathrm{v}}}{\partial y^{\mathrm{v}}}, \frac{\partial\epsilon^{\mathrm{v}}}{\partial z^{\mathrm{v}}}\right]^T, \tag{4.23}$$

using the following components:

$$\begin{aligned} \frac{\partial\epsilon^{\mathrm{v}}}{\partial x^{\mathrm{v}}} &= \left|\frac{z^{\mathrm{v}}}{C_1 + z^{\mathrm{v}}}\right| \frac{x^{\mathrm{v}} - C_2}{\sqrt{(x^{\mathrm{v}} - C_2)^2 + (y^{\mathrm{v}} - C_2)^2 + (z^{\mathrm{v}})^2}}, \\ \frac{\partial\epsilon^{\mathrm{v}}}{\partial y^{\mathrm{v}}} &= \left|\frac{z^{\mathrm{v}}}{C_1 + z^{\mathrm{v}}}\right| \frac{y^{\mathrm{v}} - C_2}{\sqrt{(x^{\mathrm{v}} - C_2)^2 + (y^{\mathrm{v}} - C_2)^2 + (z^{\mathrm{v}})^2}}, \\ \frac{\partial\epsilon^{\mathrm{v}}}{\partial z^{\mathrm{v}}} &= \left|\frac{z^{\mathrm{v}}}{C_1 + z^{\mathrm{v}}}\right| \frac{(z^{\mathrm{v}})^2(C_1 + z^{\mathrm{v}}) + C_1[(x^{\mathrm{v}} - C_2)^2 + (y^{\mathrm{v}} - C_2)^2 + (z^{\mathrm{v}})^2]}{z^{\mathrm{v}}(c_1 + z^{\mathrm{v}})\sqrt{(x^{\mathrm{v}} - C_2)^2 + (y^{\mathrm{v}} - C_2)^2 + (z^{\mathrm{v}})^2}}. \end{aligned}$$

Maximum numerical error

Since ϵ^{v} only tells the error for a given position in a camera's frustum, a method for finding the position(s) that maximizes the error is needed for establishing a more practical position independent error $\epsilon_{max}^{\mathrm{v}}$. Since the domain of the camera's depth buffer $Z_{\mathrm{c}}^{\mathrm{d}}$ and point cloud's depth buffer $Z_{\mathrm{c}}^{\mathrm{d}}$ differs, $\epsilon_{max}^{\mathrm{v}}$ would in reality be sensor dependent. To simplify the analysis while conserving generality, we use the fact that the point cloud model's domain are a subset of the camera model's domain, e.i $\mathbb{M} \subseteq \mathbb{N}^{\mathrm{N_{c,h}} \times \mathrm{N_{c,w}}}$ from Section 4.1.2.

Because of the common coordinate system, the numerical error in the camera depth buffer $Z_{\mathrm{c}}^{\mathrm{d}}$ can be written directly as $\epsilon_{\mathrm{c}}^{\mathrm{v}} = \epsilon^{\mathrm{v}}$, using the camera frustum from Section 4.1.1 as constraints. For the numerical error in the point cloud depth buffer $Z_{\mathrm{s}}^{\mathrm{d}}$, we need to use the reference models parameters $\epsilon_{\mathrm{c}}^{\mathrm{v}}(x, y) = \epsilon^{\mathrm{v}}(x_{\mathrm{s}}^{\mathrm{v}}, y_{\mathrm{s}}^{\mathrm{v}})$ from Equation 4.18, with the constraints from Equation 4.12. Since $\mathbb{M} \subseteq \mathbb{N}^{\mathrm{N_{c,h}} \times \mathrm{N_{c,w}}}$, $\epsilon_{\mathrm{c}}^{\mathrm{v}}$ would at least be an upper-bound estimate for $\epsilon_{\mathrm{s}}^{\mathrm{v}}$, leading to the following statement:

$$\epsilon_s^{\mathrm{v}} < \epsilon^{\mathrm{v}} \leq \epsilon_{max}^{\mathrm{v}}. \tag{4.24}$$

By this, we restrict ourself to finding the maximum error function for the camera model's depth buffer, searching for the point(s) that maximizes ϵ_c^{v}. We note this point as $\mathrm{Q}_{max}^{\mathrm{v}}$ known as the *maxima*, generally found by setting $\nabla\epsilon^{\mathrm{v}}$ to 0, in addition to studying the sign of either $\nabla\epsilon^{\mathrm{v}}$ or $\nabla^2\epsilon^{\mathrm{v}}$. In our case this gives the maxima candidates $z^{\mathrm{v}} = 0 \ \vee \ x^{\mathrm{v}} = y^{\mathrm{v}} = C_2$ from the first two components of $\nabla\epsilon^{\mathrm{v}}$. From the camera frustum seen in Figure 4.3 it can be seen that $0 > -n > z^{\mathrm{v}} > -f$, leaving the only possible maxima at $x^{\mathrm{v}} = y^{\mathrm{v}} = C_2$. Substituting this in the third component of Equation 4.23 gives $z^{\mathrm{v}} = -2C_1$. However, looking at Equation 4.22, $z^{\mathrm{v}} = -C_1$ gives an asymptotic value for the error if z^{v} spans all the way to $-2C_1$. One could say that by letting $n > C_1$, this could be fixed, but this would lead to $n < f(1 - \mathrm{N_{c,d}}) \leq 0$, which violates n being a possitive number. To hinder the error function from exploding at a certain value for z^{v}, $z^{\mathrm{v}} > -C_1$ is set as a condition instead. This leaves none of the maxima candidates inside the frustum. The maximum error must therefore be on the boundaries of ϵ^{v}, namely the planes making up the camera frustum.

As the positions laying on the frustum contains several potential maxima values, a method for finding a relevant analytical solution is by using reasonable conditions on the parameters of the frustum. Since $z^{\mathrm{v}} < 0$, $-f \leq z^{\mathrm{v}} \leq -n$ and $-C_1 < z$ is already justified, the following must be true: $-C_1 < -f \leq z^{\mathrm{v}} \leq -n < 0$. Expanding the terms for $-C_1 < -f$ gives:

$$f < n(\mathrm{N_{c,d}} + 1), \tag{4.25}$$

giving the following gradient component for the error function:

$$\frac{\partial\epsilon^{\mathrm{v}}}{\partial z^{\mathrm{v}}} < 0 \ \ for \ \ -n(\mathrm{N_{c,d}} + 1) < -f \leq z^{\mathrm{v}} \leq -n < 0. \tag{4.26}$$

Further, boundaries for y^{v} can be derived from the camera frustum in Figure 4.3:

$$|y^{\mathrm{v}}| \leq |z^{\mathrm{v}}| \tan\left(\frac{\mathrm{VFOV_c}}{2}\right). \tag{4.27}$$

Using the boundary condition $|z^{\mathrm{v}}| \leq f$ from Equation 4.26 it follows:

$$|y^{\mathrm{v}}| \leq |z^{\mathrm{v}}| \tan\left(\frac{\mathrm{VFOV_c}}{2}\right) \leq f \tan\left(\frac{\mathrm{VFOV_c}}{2}\right) = \frac{\mathrm{N_{c,h}}(f-n)}{2\mathrm{N_{c,d}}n} C_2. \tag{4.28}$$

This creates two boundary cases, one where $\frac{\mathrm{N_{c,h}}(f-n)}{2\mathrm{N_{c,d}}n} < 1$ and one where $\frac{\mathrm{N_{c,h}}(f-n)}{2\mathrm{N_{c,d}}n} \geq 1$, giving additional conditions on f, in addition a condition on $\mathrm{N_{c,h}}$ such that Equation 4.25 still holds:

$$\frac{\mathrm{N_{c,h}}(f-n)}{2\mathrm{N_{c,d}}n} < 1 \ \ when \ \ f < n\left(2\frac{\mathrm{N_{c,d}}}{\mathrm{N_{c,h}}} + 1\right) < n(\mathrm{N_{c,d}} + 1) \ \ and \ \ \mathrm{N_{c,h}} > 2,$$
$$\frac{\mathrm{N_{c,h}}(f-n)}{2\mathrm{N_{c,d}}n} \geq 1 \ \ when \ \ n\left(2\frac{\mathrm{N_{c,d}}}{\mathrm{N_{c,h}}} + 1\right) \leq f < n(\mathrm{N_{c,d}} + 1) \ \ and \ \ \mathrm{N_{c,h}} > 2.$$

However, since $\mathrm{N_{c,d}}$ is usually set very high such that $\frac{\mathrm{N_{c,h}}(f-n)}{2\mathrm{N_{c,d}}n} < 1$, for our purposes it is sufficient to study the first case. This gives $|y^{\mathrm{v}}| < C_2$ such that:

$$\begin{aligned} \frac{\partial \epsilon^{\mathrm{v}}}{\partial y^{\mathrm{v}}} < 0 \;\; \forall \;\; |y^{\mathrm{v}}| \leq |z^{\mathrm{v}}| \tan\left(\frac{\mathrm{VFOV_c}}{2}\right) \;\; \textit{when} \\ f < n\left(2\frac{\mathrm{N_{c,d}}}{\mathrm{N_{c,h}}} + 1\right) \;\; \textit{and} \;\; \mathrm{N_{c,h}} > 2. \end{aligned} \tag{4.29}$$

Similarly, boundaries for x^{v} can be derived from the camera frustum in Figure 4.3, using the condition $|z^{\mathrm{v}}| \leq f$ from Equation 4.26 and substituting $\tan\left(\frac{\mathrm{HFOV_c}}{2}\right)$ with $\frac{\mathrm{N_{c,w}}}{\mathrm{N_{c,h}}} \tan\left(\frac{\mathrm{VFOV_c}}{2}\right)$ from the aspect ratio in Equation 4.3:

$$|x^{\mathrm{v}}| \leq |z^{\mathrm{v}}| \tan\left(\frac{\mathrm{HFOV_c}}{2}\right) \leq f\frac{\mathrm{N_{c,w}}}{\mathrm{N_{c,h}}} \tan\left(\frac{\mathrm{VFOV_c}}{2}\right) = \frac{\mathrm{N_{c,w}}(f-n)}{2\mathrm{N_{c,d}}n} C_2. \tag{4.30}$$

This gives similar cases as we saw with the y-component:

$$\begin{aligned} &\frac{\mathrm{N_{c,w}}(f-n)}{2\mathrm{N_{c,d}}n} < 1 \;\; \textit{when} \;\; f < n\left(2\frac{\mathrm{N_{c,d}}}{\mathrm{N_{c,w}}} + 1\right) < n(\mathrm{N_{c,d}} + 1) \;\; \textit{and} \;\; \mathrm{N_{c,w}} > 2, \\ &\frac{\mathrm{N_{c,w}}(f-n)}{2\mathrm{N_{c,d}}n} \geq 1 \;\; \textit{when} \;\; n\left(2\frac{\mathrm{N_{c,d}}}{\mathrm{N_{c,w}}} + 1\right) \leq f < n(\mathrm{N_{c,d}} + 1) \;\; \textit{and} \;\; \mathrm{N_{c,w}} > 2, \end{aligned}$$

and as we did with y^{v}, studying the first case is sufficient in our case since $\mathrm{N_{c,d}}$ is large, giving us $|x^{\mathrm{v}}| < C_2$ such that:

$$\begin{aligned} \frac{\partial \epsilon^{\mathrm{v}}}{\partial x^{\mathrm{v}}} < 0 \;\; \forall \;\; |x^{\mathrm{v}}| \leq |z^{\mathrm{v}}| \tan\left(\frac{\mathrm{HFOV_c}}{2}\right) \;\; \textit{when} \\ f < n\left(2\frac{\mathrm{N_{c,d}}}{\mathrm{N_{c,h}}} + 1\right) \;\; \textit{and} \;\; \mathrm{N_{c,w}} > 2. \end{aligned} \tag{4.31}$$

Using Equation 4.26, 4.29 and 4.31, it shows that $\nabla \epsilon^{\mathrm{v}} < 0$ meaning that the maxima $\mathrm{Q}^{\mathrm{v}}_{max}$ is located at the negative boundary conditions of $(x^{\mathrm{v}}, y^{\mathrm{v}}, z^{\mathrm{v}})$. Substituting this into Equation 4.22 gives an equation for the maximum numerical error in a cameras depth-buffer:

$$\begin{aligned} \mathrm{Q}^{\mathrm{v}}_{max} &= -f\left[\tan\left(\frac{\mathrm{HFOV_c}}{2}\right), \tan\left(\frac{\mathrm{VFOV_c}}{2}\right), 1\right]^T, \\ \epsilon^{\mathrm{v}}_{max} &= \epsilon^{\mathrm{v}}(\mathrm{Q}^{\mathrm{v}}_{max}) \end{aligned} \tag{4.32}$$

In addition the following parameter conditions on the frustum needs to be met in order to use the maximum error function:

$$f < n \arg\min\left[2\frac{\mathrm{N_{c,d}}}{\mathrm{N_{c,h}}} + 1\,,\, 2\frac{\mathrm{N_{c,d}}}{\mathrm{N_{c,w}}} + 1\right] \;\; \textit{and} \;\; \mathrm{N_{c,w}},\, \mathrm{N_{c,h}} > 2. \tag{4.33}$$

Deciding parameters for sensor models

Several parameters are required to use the error function and maximum error function for depth-buffers. As the sensor models have different specifications, the remaining error parameters might become subject to interpretations, some of which are subjective. This section is therefore created to clarify and give suggestions for quantifying the remaining parameters for the sensor models.

For the camera models, $\mathrm{HFOV_c}$, $\mathrm{VFOV_c}$, $\mathrm{N_{c,h}}$ and $\mathrm{N_{c,w}}$ are given by the camera specifications, leaving us with defining proper values for n, f and $\mathrm{N_{c,d}}$.

As mentioned in Section 4.1.1, in general $\mathrm{N_{c,d}}$ are decided by the graphics APIs the operating system supports, in addition to the video card used. However, since the Unity game engine functions as a unified platform across operating systems, $\mathrm{N_{c,d}}$ are limited of being 2^{16} or 2^{24}. The former are usually associated with mobile devices while the latter are used for desktop computers with sufficient video memory. Usually, increasing $\mathrm{N_{c,d}}$ will improve the overall precision since the z_ϵ^{v} from Equation 4.21 are directly affected by it. $\mathrm{N_{c,d}}$ also directly increases C_1 which in turn decreases ϵ^{v}. Because of this, setting $\mathrm{N_{c,d}} = 2^{24}$ are preferred.

When it comes to the far plane f, the value depends on the situation. For open sea conditions, choosing the value close to the distance to the horizon: $5km$ might be sufficient. However, if subjected to hilly environments or if the cameras are placed at high altitudes, this soon becomes a problem as observations can happen at greater distances. In contrary, for Urban environments containing tall buildings the distance to the horizon might be larger than necessary. It is also worth noticing that a large faffects the error since the precision from $\mathrm{N_{c,d}}$ gets distributed across a larger distance.

Similar statements can be done for the near plane n, which increases the error as it approaches 0. Here one might argue that additional real life camera parameters such as the *circle of confusion* and *f-numbers* could be used to decide the proper near plane distance through calculating the *depth of field*. However, such additions to the camera model are outside the scope of the thesis, since objects of interest (such as the target ships) are far from the cameras. A simpler approach is to choose n to be less than the distance to the closest object in a scenario, e.i the value of the n also depends on the situation.

When it comes to the point cloud model, we are given n, f, and $\mathrm{VFOV_s}$ from the sensor specifications. In addition, since $\mathrm{HFOV_c}$, $\mathrm{VFOV_c}$ and $\mathrm{N_{c,w}}$ can be calculated from the Equations 4.14, 4.17 and 4.16 respectively, this leaves us with defining proper values for $\mathrm{N_{c,h}}$, $\mathrm{N_{c,d}}$ and $\mathrm{N_c}$.

Since we have already justified the value of $\mathrm{N_{c,d}}$, we need to decide what to do with $\mathrm{N_{c,h}}$. Considering that we chose $|x^{\mathrm{v}}|$, $|y^{\mathrm{v}}| < C_2$ in Section 4.1.3, decreasing C_2 would decrease the numerical error from Equation 4.22. Rewriting C_2 with Equation 4.17 and 4.14 to mach the point cloud sensor parameters we can see that:

$$C_2 = 2\frac{C_1}{\mathrm{N_{c,h}}}\tan\left(\frac{\mathrm{VFOV_c}}{2}\right) = 2\frac{C_1}{\mathrm{N_{c,h}}}\frac{\tan(\frac{\mathrm{VFOV_s}}{2})}{\cos(\frac{\pi}{\mathrm{N_c}})}, \tag{4.34}$$

meaning that increasing $\mathrm{N_{c,h}}$ decreases ϵ^{v}. It is therefore preferred to choose both $\mathrm{N_{c,h}}$ and $\mathrm{N_{c,d}}$ as high as possible. Intuitively speaking, this can be understood as

increasing a real photo's resolution and color depth, giving us a more detailed image. However, this comes with a cost in video memory, which limits the possible size such an image could have. The total data size of the depth buffers can be written as bytes:

$$\begin{aligned} B &= \frac{\log_2(\mathrm{N_{c,d}})}{8}\mathrm{N_{c,w}N_{c,h}N_c}, \\ B_\mathrm{s} &= \frac{\log_2(\mathrm{N_{c,d}})}{8}\frac{\sin(\frac{\pi}{\mathrm{N_c}})}{\tan(\frac{\mathrm{VFOV_s}}{2})}\mathrm{N_{c,h}^2N_c}, \end{aligned} \tag{4.35}$$

where the binary logarithm and division by 8 comes from converting $\mathrm{N_{c,d}}$ to pixel size in bytes, and the terms in the final equation comes from using Equation 4.14 and 4.16. The first equation is general, applying to both lidar and camera sensors, while the latter is a special case used for lidar.

Choosing a value for the parameter B_s instead of $\mathrm{N_{c,h}}$ is far easier, since B_s can be decided from the video cards available video memory. Therefore, by rearranging the terms we get an equation for $\mathrm{N_{c,h}}$ as a function of the variables B_s and $\mathrm{N_c}$:

$$\mathrm{N_{c,h}} = \sqrt{\frac{8}{\log_2(\mathrm{N_{c,d}})}\frac{\tan(\frac{\mathrm{VFOV_s}}{2})}{\sin(\frac{\pi}{\mathrm{N_c}})}\frac{B_\mathrm{s}}{\mathrm{N_c}}}, \tag{4.36}$$

Currently, B_s for commercially available GPUs are between 1 and 32 gigabytes. Although modern hardware are able to run multiple GPUs in parallel, the real limit of B_s would at best be a proportion of this.

Lastly, the number of cameras $\mathrm{N_c}$ being used to render depth buffers, are yet another situation based parameter. As shown in [12], increasing $\mathrm{N_c}$ decreases the beam shape error with diminishing returns. This is still true for the numerical error, which can be seen from studying Equation 4.34. Increasing $\mathrm{N_c}$ in effect lets the cosine term approach 1 with a decreasing pace as seen in Figure 4.8.

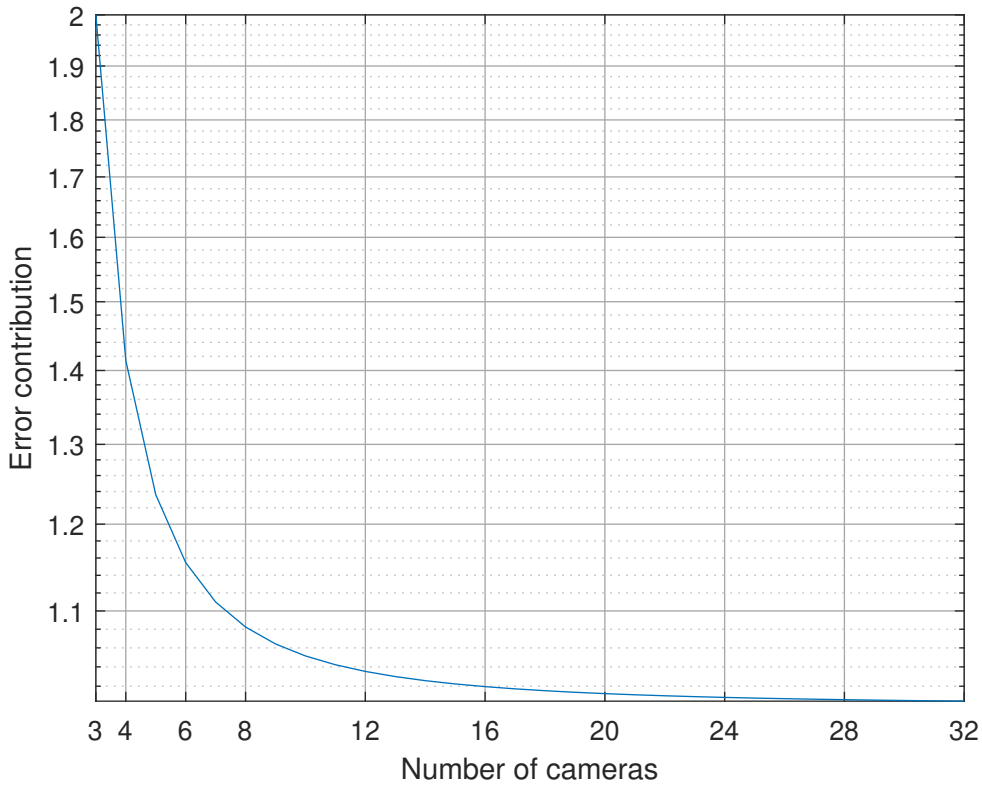


Figure 4.8: Displays $\frac{1}{\cos(\frac{\pi}{N_c})}$. This illustrates the diminishing return of decreasing the numerical error by increasing the number of cameras. Using more than 8 cameras almost halves the numerical error in comparison of using 3

The figure shows that $N_c \geq 3$ such that C_2 does not approach infinity. In addition it shows the diminishing returns of increasing the number of cameras in order for the numeric error to decrease. What is also still true from [12], is that increasing N_c increases execution time, since the computations we saw in Section 4.1.1 must be run N_c times. With these two considerations in mind, one might want to choose $N_c = 3$ for situations where fast computations are needed, while $N_c \geq 8$ for situations where a doubling of precision are required and cannot be achieved solely on increasing B_s.

4.2 EMR sensor modeling

We have until now focused on depth-buffers which plays a vital part in simulating the EMR sensors in Autoferry Gemini. In order to replicate the real data, additional information is needed about e.g 3D models and sensor calibrations, in addition to addressing how we choose our simulation parameters with respect to the numerical error we saw in Section 4.1.3. Looking back at the digital twin architecture in Figure 2.2, we also need to address how the communication between the platforms are done in order to generate synthetic data. We will begin with looking at the camera and lidar sensor in section 4.2.1 and 4.2.2 respectively, before we discuss how we choose the simulation parameters based on the numerical error in Section 4.2.3. Further, information about the 3D models is given in Section 4.2.4, before we end the chapter with how the sensor recordings are done, using the different platforms involved in the digital twin framework from Section 4.2.5.

4.2.1 RGB camera

The RGB camera sensor extends on previous work done in [11] and [12]. Since the synthetic camera uses the same pin hole camera model and rasterization process as in Section 4.1.1, this section will only cover how the images was made similar to the real counterpart.

Camera placement and calibration

Due to the lack of camera calibration, and properly documented camera placements, manual calibration was performed to the synthetic RGB cameras. This consisted of choosing field of view angles HFOV_c and VFOV_c using real images as reference, such as seen in Figure 4.9. In addition the cameras focal point along the cameras image axis (cyan color) in Figure 3.6, needed to be placed 36mm from the camera sensor instead of the 6mm specified by the camera lens manufacturer (Figure 3.6) in order for the images to become similar.

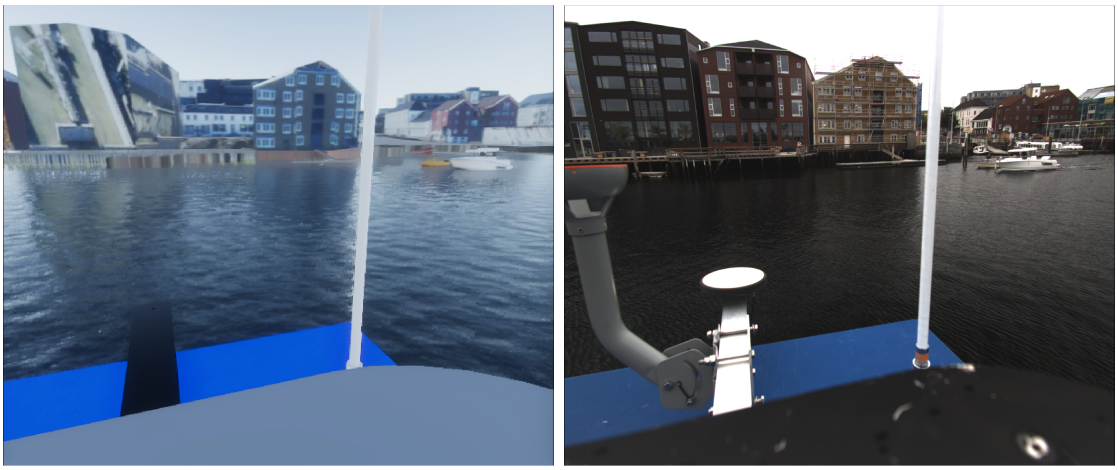


Figure 4.9: Synthetic camera parameters was calibrated using the real image counterpart

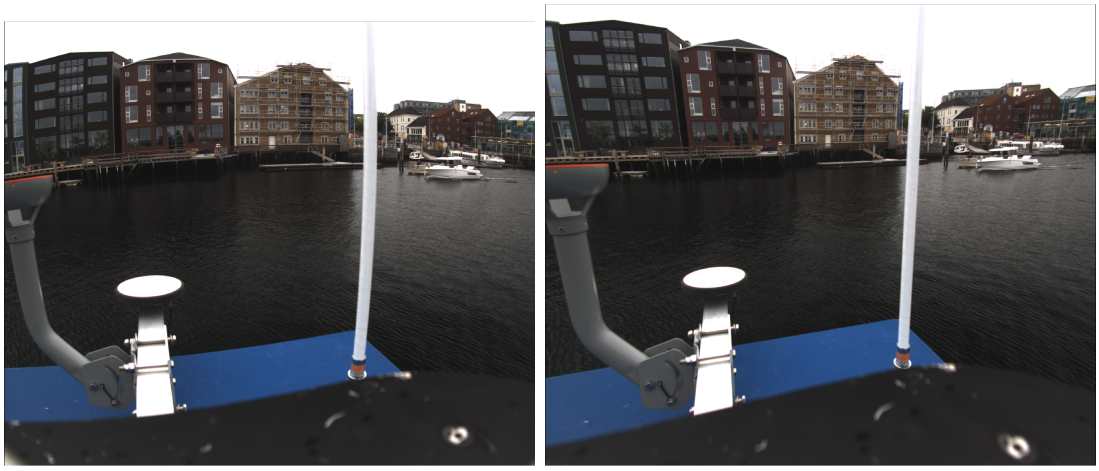


Figure 4.10: Distorted images such as the one to the left is flatened using ROS's Plub Bob algorithm. Right image is the post processed image

Accounting for lens distortion

In contrary to real camera images that are distorted through lenses, the pinhole camera preserves geometry, as well as having no depth of field, e.i the focus are the same for all objects regardless of the distance from the camera. The former is especially important to consider since object detectors are trained on flattened images where the lens distortions are accounted for. Because of this, instead of giving the synthetic images a distortion effect, we remove the real images distortions using ROS's inbuilt *Plumb bob* algorithm, flattening the real images to become similar to the pinhole camera model as shown in Figure 4.10.

4.2.2 Lidar

Using the newly introduced method in Section 4.1.2, the lidar model improves on the numerical errors found in previous work [11, 12].Sending this sensor data to ROS, does however put the previous work a bit out of context regarding the mathematics. In addition, in real lidars, rays are not always returning back to the transmitter, creating *ray drops*. This section seeks to enlighten the reader of how the previous work [11, 12] can be transformed to connect with the upcoming ROS messages in Section 4.2.5.

Generating Point cloud

Using the newly established depth-buffer $Z_{\mathrm{s}}^{\mathrm{d}}(x^{\mathrm{d}}, y^{\mathrm{d}})$, the process of generating a point cloud is done by reversing the steps in Section 4.1.1. First of we represent the image coordinates of the depth-buffer with:

$$\mathrm{Q}_{\mathrm{c}}^{\mathrm{d}} = [x^{\mathrm{d}}, y^{\mathrm{d}}, Z_{\mathrm{s}}^{\mathrm{d}}(x^{\mathrm{d}}, y^{\mathrm{d}})]^T \tag{4.37}$$

Then converting it to image space

$$\mathrm{Q}_{\mathrm{c}}^{\mathrm{i}} = 2\mathbf{N}_{\mathrm{c}}^{-1}\mathrm{Q}_{\mathrm{c}}^{\mathrm{d}} - 1. \tag{4.38}$$

To project each point of the depth-buffer back to view space, the projection matrix is first inverted: $\mathbf{T}_{\mathrm{i}}^{\mathrm{v}} = (\mathbf{T}_{\mathrm{v}}^{\mathrm{i}})^{-1}$. Then the steps of generating homogeneous coordinates and denormalisation takes place.

$$\begin{bmatrix} \bar{\mathrm{Q}}_{\mathrm{c}}^{\mathrm{v}} \\ \bar{w}_{\mathrm{c}}^{\mathrm{v}} \end{bmatrix} = \mathbf{T}_{\mathrm{i}}^{\mathrm{v}} \begin{bmatrix} \mathrm{Q}_{\mathrm{c}}^{\mathrm{i}} \\ 1 \end{bmatrix}, \tag{4.39}$$

$$\mathrm{Q}_{\mathrm{c}}^{\mathrm{v}} = \frac{1}{\bar{w}_{\mathrm{c}}^{\mathrm{v}}} \bar{\mathrm{Q}}_{\mathrm{c}}^{\mathrm{v}}, \tag{4.40}$$

Giving us point cloud data seen from one camera. In order to cover a whole lidar sweep as in Figure 4.5, $\mathrm{N_c}$ cameras are angled evenly across the lidars azimuth angle. This gives each camera it's own local point cloud: $\mathrm{Q}_0^{\mathrm{v}}$, ..., $\mathrm{Q}_{\mathrm{N_c}-2}^{\mathrm{v}}$, $\mathrm{Q}_{\mathrm{N_c}-1}^{\mathrm{v}}$. These are then united using a rotation matrix that transform each point to the first clouds reference frame:

$$\mathbf{R}_y(\mathrm{c}) := \begin{bmatrix} \cos(\frac{2\pi}{\mathrm{N_c}}c) & 0 & \sin(\frac{2\pi}{\mathrm{N_c}}c) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\frac{2\pi}{\mathrm{N_c}}c) & 0 & \cos(\frac{2\pi}{\mathrm{N_c}}c) & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.41}$$

$$\mathrm{Q}_{\mathrm{li}}^{\mathrm{v}} = \mathbf{R}_y(\mathrm{c})\mathrm{Q}_{\mathrm{c}}^{\mathrm{v}}. \tag{4.42}$$

Where $\mathrm{Q}_{\mathrm{li}}^{\mathrm{v}}$ represents a point in the lidars point cloud, using the same reference frame as $\mathrm{Q}_0^{\mathrm{v}}$.

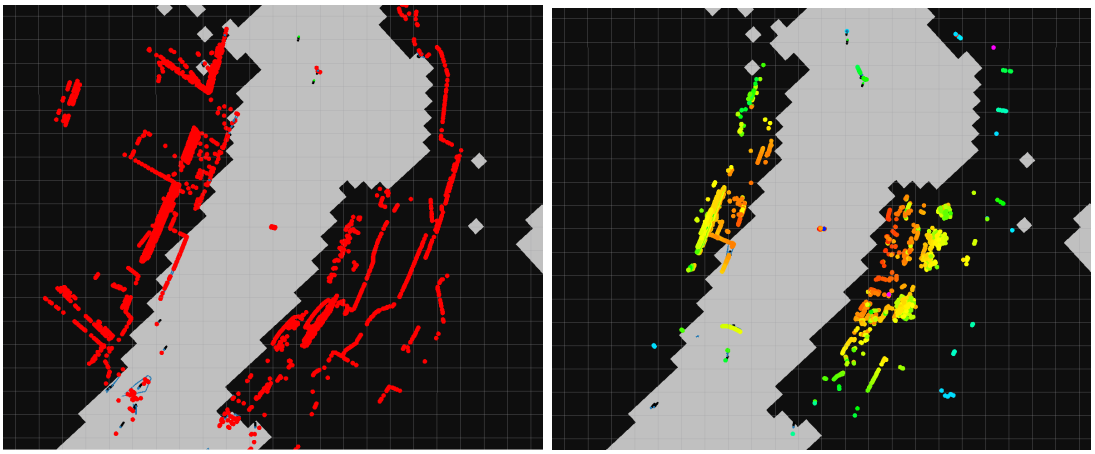


Figure 4.11: Point cloud data from both synthetic and real datasets at scenario 6. Notice how much more detail is present in the synthetic cloud to the left versus the real to the right

Ray drop

Modeling a lidars ray drop is currently a field of research, especially relevant in the car industries [13]. The paper introduces a machine-learning technique to reproduce a lidars behaviour. This is however outside the scope of the thesis, but the paper also mentions the use of modeling ray drops with introducing random noise. True noise is however not reprodusible, e.i when we simulate the same scenario over again we get different results. Since reputability is of importance for this application a *Pseudo random number generator* (PRNG) is used to mimic randomness.

Usually, programming languages comes with PRNGs running on the CPU. However, GPU's are instead programmed with *shader languages* [11] which usually does not support such generators. Unfortunately for us, the *High Level Shading Language* (HLSL) used for implementing the lidar is not an exception here. This comes from a variety of reasons, but most has to do with the GPU's parallel architecture, where a common program are run in parallel on multiple threads. Since random numbers are independent of each other, this means that each program running on a thread needs a way to become independent of each other.

Since GPU applications can either aim to be fast or slow, there is no design for PRNGs that fits every use case. Because of this, several PRNGs exists, where the one used here are Nathan Reeds implementation [26].

Using this, we give each point Q_{li}^{v} a probability of 80% to drop, e.i not return any result. This seems high in comparison to the 10% used in [13], but when comparing the synthetic and real point clouds in Figure 4.11, we might still be too low.

The elements in Q_{li}^{v} can now be used directly into the *PointCloud2* message used by ROS.

4.2.3 Numerical error

Each sensor has its own numerical errors depending on specifications from the sensor manufacturer in addition to simulation parameters connected with the use of depth buffers. Section 4.1.3 is used as a guideline for deciding the simulation parameters for the different sensors, while the remaining parameters to calculate the numerical errors (4.32), (4.22) and (4.21) uses information from the sensors listed in Figure 3.6. Some restrictions is also done in order for the simulation to run properly on the hardware listed in Section 4.2.5, in addition to the current stability the digital twin framework possesses. Table 4.1 contains all the sensor parameters used to create the synthetic dataset, and resulting values of errors (4.32) and video memory size (4.35) on the right hand side.

Sensors	$\mathrm{VFOV_c}$	$\mathrm{HFOV_c}$	$\mathrm{N_c}$	$\mathrm{N_{c,h}}$	$\mathrm{N_{c,w}}$	$\mathrm{N_{c,d}}$	n	f	$\epsilon^{\mathrm{v}}_{max}$	B
RGB	65.4	75.0	1	512	612	2^{24}	0.08	10^4	**140**	$9.4 \cdot 10^4$
Lidar	41.5	45.0	8	3104	32768	2^{24}	0.05	100	**0.04**	$3.3 \cdot 10^8$

Table 4.1: Parameters from sensor specifications and calculations from Section 4.1.3 with resulting numerical error. Opposed to using *radians* as in Section 4.1.3, the field of views used here are shown as degrees

Camera error

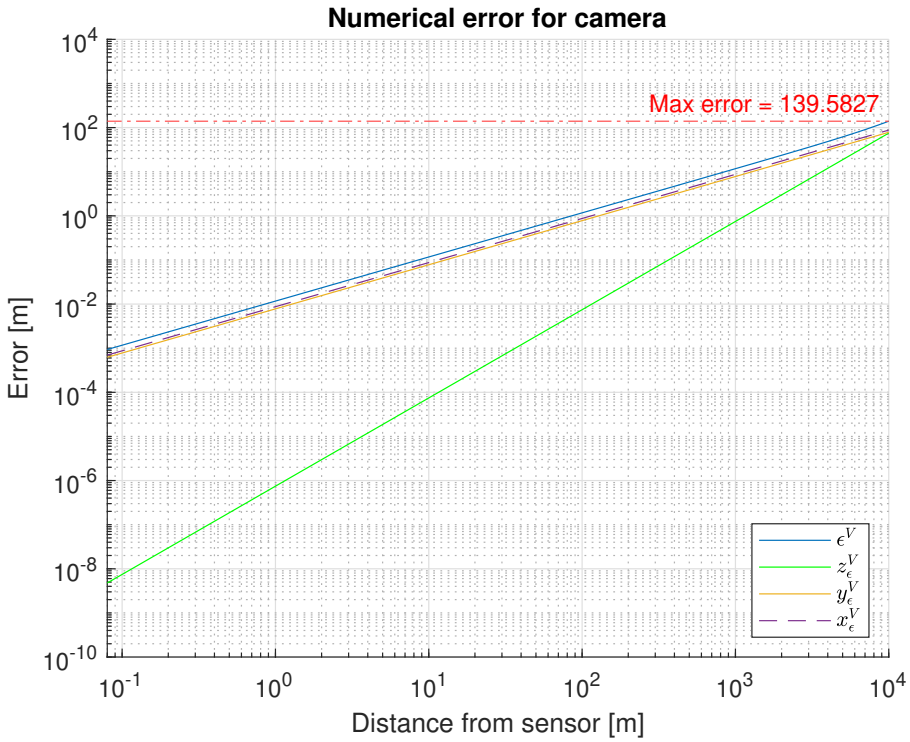


Figure 4.12: The total, component-wise and maximum errors with respect to distance for the camera sensor. Beware of the logarithmic axes

From Section 4.2.1, the field of views in Table 4.1 is set manually, instead of following the sensor specifications. The number of cameras $N_c = 1$ in this case, not to be confused with the number of cameras on the sensor rig. This is because the error for one camera, applies to all cameras. Camera resolutions are set to 612×512 even though the real sensor specifications are 2448×2048. We can safely do so, since the real images are cropped to this resolution before being used in the tracking algorithm we will see in Chapter 5. Further, the near plane is set to 8 cm from the focal point, in order to render outside the sensor rig containing the cameras (Figure 3.6). Meanwhile, a 10km distance is chosen for the far plane to be on the safe side regarding background renderings.

The maximum error of ca. 140m seems big, but considering how large a pixel projects at 10km, it does not seem unreasonable. In fact, the component errors x_ϵ^v and y_ϵ^v will also be present in the real camera, since they represent pixel errors. In contrary, the depth component error z_ϵ^v is of bigger interest, since it is not present in real cameras. With this said, depth information is not directly present in rendered images either. Renderings do rely on depth information for certain effects, such as lightning, but we will not delve further into rendering topics than what is covered in Section 4.1.1. To justify this, the depth error is very small in comparison to the other component errors as seen from Figure 4.12, being less than 1cm at a 100m distance.

Lidar error

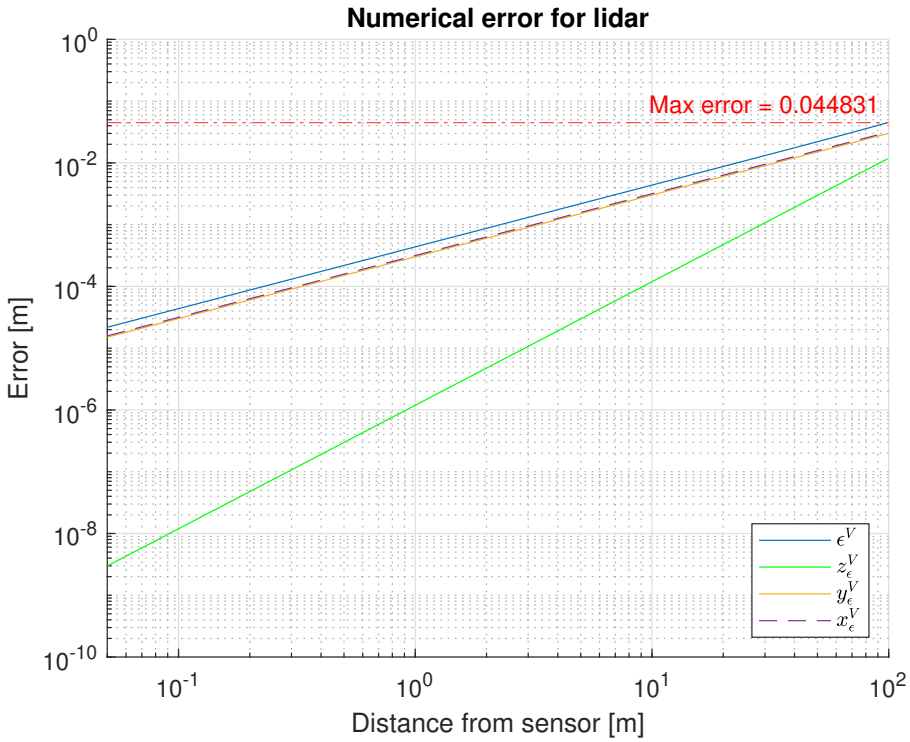


Figure 4.13: The total, component-wise and maximum errors with respect to distance for the lidar sensor. Beware of the logarithmic axes

Specifications from the VLP-16 sensor tells that the vertical spread of the point cloud is at $\text{VFOV}_\text{s} = 30°$ giving $\text{VFOV}_\text{c} = 41.5°$ from using Equation 4.17. In addition, 8 cameras are used to get the benefit of double precision (Figure 4.8). Since the computer hardware uses video memory for other purposes than just running EMR sensors, $\text{N}_\text{c,w}$ and $\text{N}_\text{c,h}$ is chosen such that $B = 3.3 \cdot 10^8$ for the lidar according to Equation 4.35. The far plane is set to the theoretical maximum distance for the VLP-16, and the near plane to 5cm to get all potential blind zones coming from the struts on the sensor rig (Figure 3.6).

This results in a maximum error of ca. 4cm (±2cm) which is within the ±3cm accuracy seen in the VLP-16's specifications. This might seem a bit vague, but considering that the specification do not give any information of how the accuracy varies with distance, moreover information about the distribution the accuracy is based on, this becomes our best bet.

4.2.4 3D modeling

Since all of the EMR sensors uses depth-buffers in order to work, 3D models becomes an essential asset for sensor modeling. Good 3D models heightens the possible fidelity a simulation can give, but it also creates challenges in obtaining models. For the digital twin framework, the 3D models can be categorized into boats and the environment they operate in.

Boats

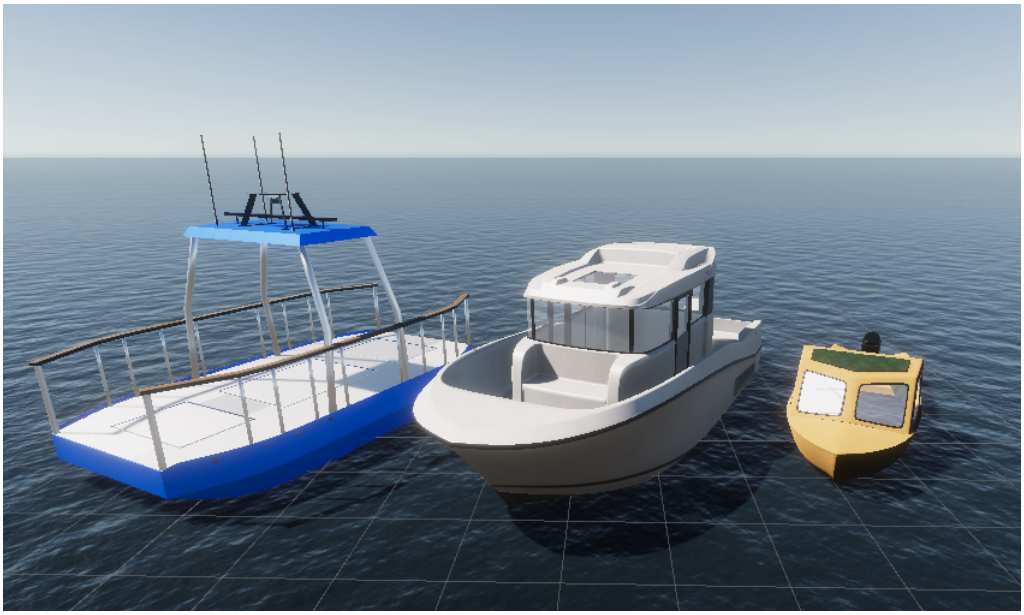


Figure 4.14: 3D models of the participating boats in the scenario description (Section 3.2.4). From left to right: milliAmpere, Havfruen and Finn

The target boats participating in the scenarios are modeled by Erik Veitch and Thomas Kaland as replicas of their real counterpart. This was done as contributions to the project mentioned in the acknowledgement and appendices. A 3D model of the milliAmpere ferry already existed from previous work done by the author in a student project at NTNU [10], a precursor to Autoferry Gemini. These models can be seen in Figure 4.14, and are also depicted alongside their real counterparts in Figure 3.7 and 3.5 where dimensions are given.

In addition, several open sourced 3D boats designed for academic and educational purposes were obtained from the websites *cgtrader.com* and *download-free3d.com*. These functioned as anchored boats at the harbour area in Trondheim, seen in Figure 4.15.

Environment

The thesis uses the same environment of Trondheim as Autoferry Gemini, originating from *Trondheim Kommune* [27]. Using the boats previously mentioned, the environment is populated with boats in areas close to where the scenarios take place (Figure 4.15 and 4.16).

Figure 4.15: Image of the harbour containing 3D models obtained from open source websites

Figure 4.16: 3D city model of Trondheim, previously configured and used in the authors specialisation project [11] and in Autoferry Gemini [12]

4.2.5 Sensor recording

Following the digital twin architecture from Figure 2.2, the sensor recording starts by sending scenario data from OSP to Unity. This is followed by generating synthetic data based on the sensor models that have been discussed in the current Chapter. Finally, the sensor data is sent to ROS to be recorded into a ROS-bag. This whole process is done by running the digital twin framework on a computer with the following specifications:

- *OS*: Windows 10 Home, 64-bit
- *CPU*: Intel(R) Core(TM) i5-6600K @ 3.50GHz, 4 Core(s) CPU
- *Memory*: 16 GB RAM, 512 GB SSD
- *GPU*: XFX Radeon RX 480 8GB RS

OSP to Unity

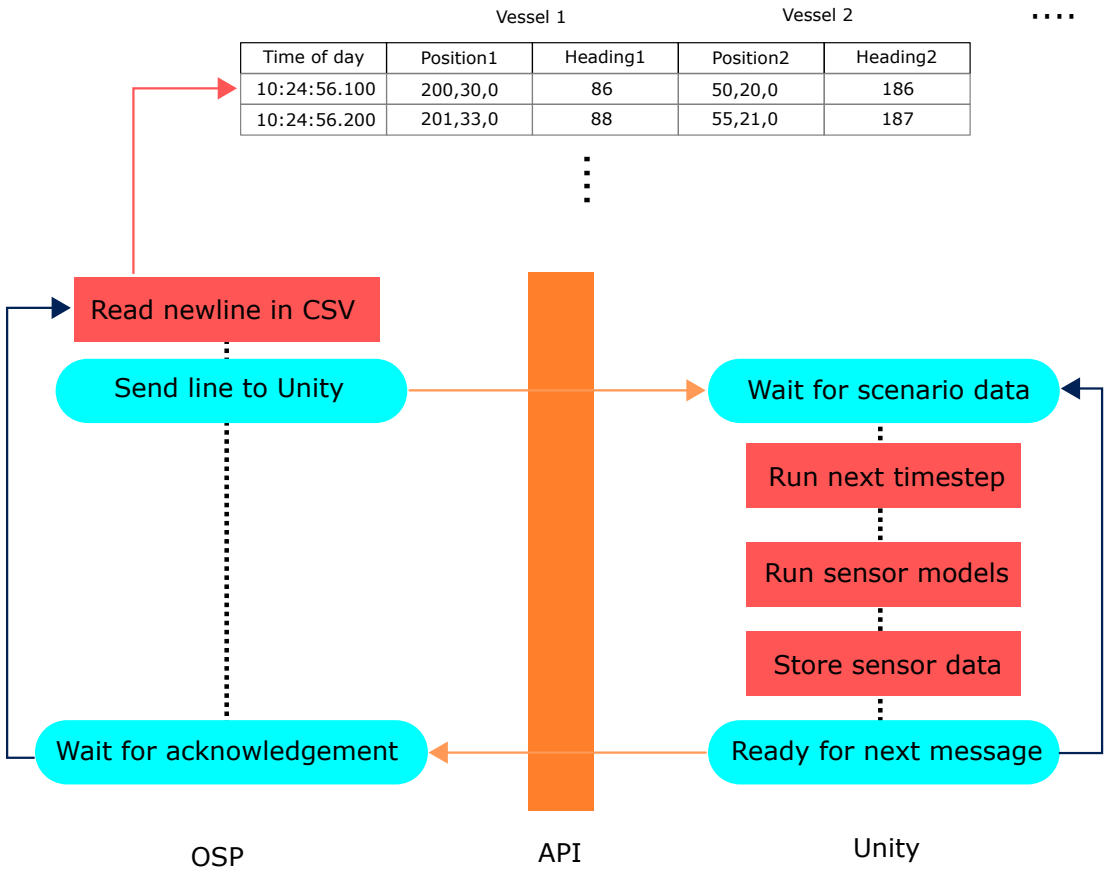


Figure 4.17: Communication between Unity and OSP

Using the ground truths established in Section 3.3.2, each vessels position and heading is read by the scenario generator implemented in OSP (Figure 2.2). The generator is made by Tobias Rye Torben, functioning as a dummy demonstration of how scenarios in the future can be generated from software implemented in OSP. For the sake of the thesis, it functions as scenario manager which controls Unity.

Using a gRPC API developed by Thomas Skarshaug [17], Unity halts the simulation until a vessel message arrives from OSP seen in Figure 4.17. After the message is received, the 3D models is Section 4.2.4 is translated using Unity's scripting language, before the sensor models creates EMR data. Finally, Unity calls on ROS to store the data, ending with a new request to OSP for the next vessel transformations.

Unity to ROS

At the end of the datapipeline seen in Figure 2.2, ROS records the sensor data into a ROS-bag by using the same API as in Figure 4.17, sending sensor data instead of vessel transformations. In order to utilize this functionality, ROS uses a standardized set of messaging types to record sensor data, making it easier to run third party programs from a common interface. In our case, the messages of importance are *Pointcloud2* for lidar data, and the *Image* type for the RGB camera data.

In every ROS-message there exist a header, which is created by ROS itself, containing information about when the message was received. The data is represented as a list of raw bytes known as *byte strings*, where the message contains the computer systems *endianness*. This tells us if a single bytes most significant bit is at the beginning or the end of the byte, which is needed to format the data correctly between different hardware and data systems. In our case our system supports the latter, know as *little endian*, the opposite of *big endian* format.

```
# sensor_msgs/Image.msg

Header header          # Header timestamp

uint32 height          # image height
uint32 width           # image width

string encoding        # Encoding of pixels --

uint8 is_bigendian     # Is the data bigendian?
uint32 step            # Full row length in bytes
uint8[] data           # Image data

```

To mimic the real camera data, the encoding is set to *bgr8*, with image *height* and *width* given as 612 and 512 pixels respectively. This means that the byte string comes in as blue, green and red bytes, making up $512 \times 612 \times 3$ elements for the *data*. The *step* message variable in this case is an over parameterisation, not needed for the message to function. Nevertheless, for clarity it is set to 612×3.

When it comes to the lidars point cloud *data*, each point is "encoded" according to the *PointField* datatype. For the VLP-16 the encoding follows Table 4.2.

name	offset	datatype	count
x	0	7	1
y	4	7	1
z	8	7	1
intensity	12	7	1
ring	16	4	1
time	18	7	1

Table 4.2: *PointField* data for Velodyne VLP-16 puck in ROS

However, since the data is generated by a GPU, the *datatype* of the *ring* element is changed to 7 for performance reasons. Fortunately, for this application only the positional information are of importance.

```
# sensor_msgs/PointField.msg

# Datatypes
uint8 INT8    = 1, uint8 UINT8   = 2, uint8 INT16   = 3,
uint8 UINT16  = 4, uint8 INT32   = 5, uint8 UINT32  = 6,
uint8 FLOAT32 = 7, uint8 FLOAT64 = 8

string name      # Name of field
uint32 offset    # Offset from start of point struct
uint8  datatype  # Datatype enumeration, see above
uint32 count     # How many elements in the field

```

This means that the byte string consists of 24 bytes for each point in a lidarscan, known as *point_step* in the *PointCloud2* message. The number of points are specified by setting the *height* to 1, and the width to 16×1024 corresponding to 16 lasers with a scan resolution of 1024. Finally, the *row_step* and *is_dense* element is set to 0 and 1 respectively to match the contents of the real dataset's ROS-bag created by Øystein Kaarstad Helgesen in Section 3.3.1.

```
# sensor_msgs/PointCloud2.msg

Header header

# 2D structure of the point cloud
uint32 height
uint32 width

PointField[] fields

bool    is_bigendian # Is the data bigendian?
uint32  point_step   # Length of a point in bytes
uint32  row_step     # Length of a row in bytes
uint8[] data         # Size of point data (row_step*height)

bool is_dense        # True if there are no invalid points

```

Chapter 5

Data Comparison

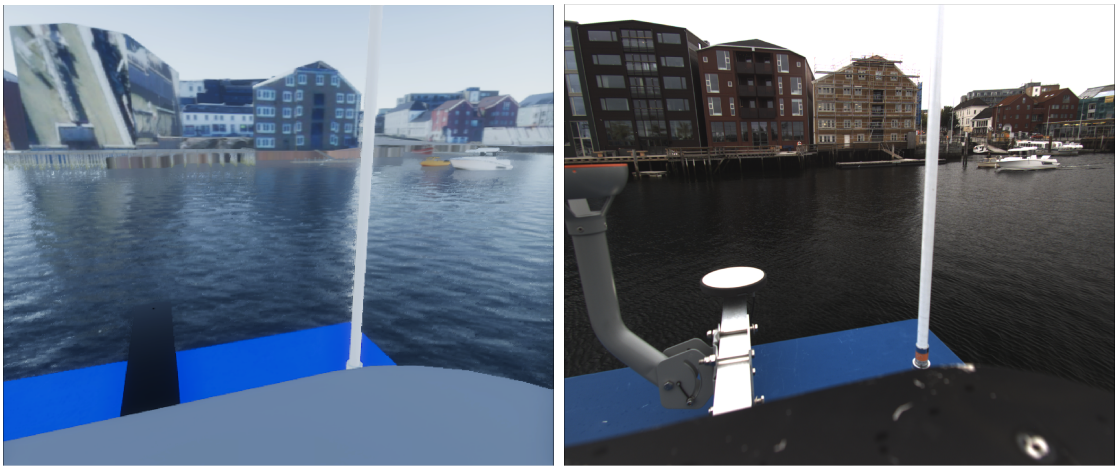


Figure 5.1: Example of synthetic and real image data for scenario 3

Comparing reality to simulation is not an easy task, especially when considering the abundance of detail reality consists of. Generally, this is done by proposing a model which is then validated through empirical data. Choosing how the measurement should be compared to the model, is however not that trivial. If we aim to show that the data are different, there is hardly any need to do much analysis at all, which is obvious from comparing the images in Figure 5.1. On the other hand, we can likewise see similarities, such as having an ocean, a sky, and boats passing by. At this point, its worth remembering what the end goal of the simulation is: "validating autonomous ships". This tells us it is not our human perception that needs be tested. It is what the autonomous agent perceives, that is interesting.

In the field of target tracking, *performance metrics* already exist as measurables to aid researchers to evaluate and improve their algorithms. Using the same metrics for doing dataset comparison, gives us a way to improve the simulation in the future, but it also tells us if the autonomous agent can use simulation as a substitution to real data. This chapter begins with talking about target tracking, before the metrics are introduced. In the last section of this chapter, the different metrics and interpretation of some of their characteristics, are done by comparing the datasets from Chapter 3 and 4.

5.1 Target tracking

Since we have choosen to compare the synthetic and real datasets by studying the performance of a target tracker, some fundamental topics are required to understand both the tracker and how its performance are evaluated. The first section will cover basic probability theory used throughout this chapter, before a section discussing the use of filters to track boats is presented. This is followed by a high level explanation of the tracker in use, and ending the section with explaining typical performance metrics used by today's tracking algorithms. This section is largely based on the book: *Fundamentals of Sensor Fusion* by Edmund Førland Brekke [16], though simplified in order to cover the most relevant topics.

5.1.1 Probability Theory

The target tracking used in this work relies on a probabilistic approach. Because of this we will first cover some of the fundamentals, before describing what random variables are and their connection to probability density functions. Lastly, we will cover the multivariate Gaussian distribution, a cornerstone for being able to explain the filter used in the tracker.

Fundamentals

Probability theory is about assigning numbers to events, which in turn tells us how likely they are to happen [16]. Events in this chapter are noted as large characters, such as A and B, while the probability of an event happening is noted with the operator $Pr\{\cdot\}$.

Moreover, the conventional boolean operators $\cup$ and $\cap$ are respectively know as the union/or operator, and intersection/and operator. This is used among others to define the *independence* of events:

$$Pr\{A \cap B\} = Pr\{A\}Pr\{B\}. \tag{5.1}$$

Further, the *conditional probability* where the probability for A happening given that B have happened, is defined by using the | operator:

$$Pr\{A|B\} = \frac{Pr\{A \cap B\}}{Pr\{B\}}. \tag{5.2}$$

Since the intersection operator is commutative, $Pr\{A \cap B\} = Pr\{A|B\}Pr\{B\} = Pr\{B|A\}Pr\{A\}$, where solving one of the conditions gives the *Bayes rule*:

$$Pr\{A|B\} = \frac{Pr\{B|A\}Pr\{A\}}{Pr\{B\}}. \tag{5.3}$$

Random variables and probability functions

Using events in mathematical approaches can be very cumbersome because of their generality. Because of this, random variables are often used to give a quantitative perspective and possibility of creating and calculating probabilistic models. Random

variables are however still known as events. This is done by representing them as abstract outcome spaces, functioning as sources for sampling concrete quantities known as *realisations*. Realisations are represented as lowercase letters to their event / random variable counterparts.

To describe the probability of a random variable, *probability density functions* (pdf) are used, described by the operator $p(\cdot)$ where the realisation of the random variable is used as parameter. It is also worth noting pdf's are used to describe *stochastic processes* from random variables, the random equivalent to functions using real variables.

A general property of all pdf's are that their area sum up to 1:

$$\int_{-\infty}^{\infty} p(x)dx = 1. \tag{5.4}$$

Multivariate Gaussian distribution

The most important pdf when speaking of target tracking is the Gaussian distribution. This stems from the beneficial properties it has towards linear transformations, moreover how the *central limit theorem* tells us that the sum of many independent samples, goes towards a Gaussian distribution. For multivariate settings, the Gaussian distribution is defined as:

$$\mathcal{N}(\boldsymbol{x}; \mu, \mathbf{P}) = \frac{1}{(2\pi)^{\frac{\mathrm{N_g}}{2}} |\mathbf{P}|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\mu)^T \mathbf{P}^{-1}(\boldsymbol{x}-\mu)\right), \tag{5.5}$$

for a vector $\boldsymbol{x}$ of size $\mathrm{N_g}$ subjected to the expectation value μ and covariance matrix $\mathbf{P}$. The distributions first property of importance is *linearity*. If the a linear transformation $\mathbf{y} = \mathbf{F}\boldsymbol{x}$ is subject to a random variable $\boldsymbol{X}$ by the distribution $p_{\boldsymbol{X}(x)} = \mathcal{N}(\boldsymbol{x}; \mu, \mathbf{P})$ the distribution of the transformation results in:

$$p(\mathbf{y}) = \mathcal{N}(\mathbf{y}; \mathbf{F}\mu, \mathbf{F}\mathbf{P}\mathbf{F}^T). \tag{5.6}$$

Furthermore, a second property to be aware of is the *product identity*:

$$\mathcal{N}(\boldsymbol{z}; \mathbf{H}\boldsymbol{x}, \mathbf{R})\mathcal{N}(\boldsymbol{x}; \bar{\boldsymbol{x}}, \bar{\mathbf{P}}) = \mathcal{N}(\boldsymbol{z}; \bar{\boldsymbol{z}}, \mathbf{S})\mathcal{N}(\boldsymbol{x}; \hat{\boldsymbol{x}}, \hat{\mathbf{P}}), \tag{5.7}$$

given that the variables involved obey:

$$\begin{aligned} \bar{\boldsymbol{z}} &= \mathbf{H}\bar{\boldsymbol{x}} \\ \hat{\boldsymbol{x}} &= \bar{\boldsymbol{x}} + \mathbf{W}(\boldsymbol{z} - \mathbf{H}\bar{\boldsymbol{x}}) \\ \mathbf{S} &= \mathbf{R} + \mathbf{H}\bar{\mathbf{P}}\mathbf{H}^T \\ \hat{\mathbf{P}} &= (\mathbf{I} - \mathbf{W}\mathbf{H})\bar{\mathbf{P}} \\ \mathbf{W} &= \bar{\mathbf{P}}\mathbf{H}^T\mathbf{S}^{-1}. \end{aligned} \tag{5.8}$$

where $\mathbf{H}$ and $\mathbf{F}$ will be covered further in the section about the Kalman Filter.

5.1.2 Filters

In order to track targets, information about observation and model prediction must be handled to give good estimates of several relevant target states. This typically consist of position and velocity, but heading can also be relevant if the ANS handles other vessels. There are several ways to handle target tracking, but most of them are based on the *Bayesian Filter* which we will cover in the first section. From this, assumptions are made towards linear models and Gaussian processes, where the *Kalman filter* is known for being the optimal solution. However, these assumptions are often too strict to function due to the non-linearity of the real world. The last section therefore introduces the *Extended Kalman Filter*, which is used in the complete tracking algorithm.

Bayesian Filter

We begin with defining a pdf that tells us about the targets state estimates $\boldsymbol{x}$ given all of our observations $\boldsymbol{z}$. We define this as our *belief distribution*:

$$bel(\boldsymbol{x}_k) = p(\boldsymbol{x}_k|\boldsymbol{z}_{1:k}), \tag{5.9}$$

using k to note a time instance, and the ":" operator to note a sequence of instances:

$$\boldsymbol{z}_{1:k} = \boldsymbol{z}_1,\ ...,\ \boldsymbol{z}_{k-1},\ \boldsymbol{z}_k.$$

In this case, the current belief uses all previous and present observation instances to tell what the present target states are. Using the pdf equivalent of Bayes Rule (5.3), we can rewrite (5.9) to:

$$bel(\boldsymbol{x}_k) \propto p(\boldsymbol{z}_k \,|\, \boldsymbol{x}_k,\ \boldsymbol{z}_{1:k-1}) p(\boldsymbol{x}_k \,|\, \boldsymbol{z}_{1:k-1}), \tag{5.10}$$

where we use the proportional equal operator $\propto$ to ignore the normalisation terms in Bayes Rule. Using Equation 5.4, the last term can formed as an integral, know as the Chapman-Kolmogorov equation:

$$p(\boldsymbol{x}_k \,|\, \boldsymbol{z}_{1:k-1}) = \int p(\boldsymbol{x}_k,\ \boldsymbol{x}_{k-1} \,|\, \boldsymbol{z}_{1:k-1}) d\boldsymbol{x}_{k-1}, \tag{5.11}$$

This equation can be simplified further by letting the process model follow the *Markov property*:

$$p(\boldsymbol{x}_k \,|\, \boldsymbol{x}_1,\ ...,\ \boldsymbol{x}_{k-1},\ \boldsymbol{x}_k,\ \boldsymbol{z}_1,\ ...,\ \boldsymbol{z}_{k-1},\ \boldsymbol{z}_k) = p(\boldsymbol{x}_k \mid \boldsymbol{x}_{k-1}), \tag{5.12}$$

telling us that the stochastic processes only relies on the latest state, not the history of states before it. Using this and the conditional probability (5.2), a *Markov chain* follows:

$$p(\boldsymbol{x}_k,\ \boldsymbol{x}_{k-1} \,|\, \boldsymbol{z}_{1:k-1}) = p(\boldsymbol{x}_k \,|\, \boldsymbol{x}_{k-1}) p(\boldsymbol{x}_{k-1} \,|\, \boldsymbol{z}_{1:k-1}) \tag{5.13}$$

This gives a simpler expression for the Chapman-Kolmogorov equation:

$$p(\boldsymbol{x}_k \,|\, \boldsymbol{z}_{1:k-1}) = \int p(\boldsymbol{x}_k \,|\, \boldsymbol{x}_{k-1}) p(\boldsymbol{x}_{k-1} \,|\, \boldsymbol{z}_{1:k-1}) d\boldsymbol{x}_{k-1}, \tag{5.14}$$

giving us:

$$bel(\boldsymbol{x}_k) \propto p(\boldsymbol{z}_k \mid \boldsymbol{x}_k,\, \boldsymbol{z}_{1:k-1}) \int p(\boldsymbol{x}_k \mid \boldsymbol{x}_{k-1}) p(\boldsymbol{x}_{k-1} \mid \boldsymbol{z}_{1:k-1}) d\boldsymbol{x}_{k-1}. \tag{5.15}$$

The *Markov property* also simplifies the measurement model:

$$p(\boldsymbol{z}_k \mid \boldsymbol{x}_k,\, \boldsymbol{z}_{1:k-1}) = p(\boldsymbol{z}_k | \boldsymbol{x}_k), \tag{5.16}$$

which leads us to the final recursive *Bayes Filter*:

$$\begin{aligned} bel(\boldsymbol{x}_k) &\propto p(\boldsymbol{z}_k \mid \boldsymbol{x}_k) \int p(\boldsymbol{x}_k \mid \boldsymbol{x}_{k-1}) p(\boldsymbol{x}_{k-1} \mid \boldsymbol{z}_{1:k-1}) d\boldsymbol{x}_{k-1}, \\ &\propto p(\boldsymbol{z}_k \mid \boldsymbol{x}_k) \int p(\boldsymbol{x}_k \mid \boldsymbol{x}_{k-1}) bel(\boldsymbol{x}_{k-1}) d\boldsymbol{x}_{k-1}. \end{aligned} \tag{5.17}$$

For reasons of intuition, this equation is often divided into recursive *prediction* and *correction* steps:

$$\begin{aligned} Prediction(\boldsymbol{x}_k) &= \int p(\boldsymbol{x}_k \mid \boldsymbol{x}_{k-1})\; Correction(\boldsymbol{x}_{k-1})\; d\boldsymbol{x}_{k-1}, \\ Correction(\boldsymbol{x}_k) &\propto \quad p(\boldsymbol{z}_k \mid \boldsymbol{x}_k)\; Prediction(\boldsymbol{x}_k), \end{aligned} \tag{5.18}$$

where the targets states is first estimated using only the process model, before being corrected by the measurement model.

Kalman Filter

Even though the Bayes Filter creates a general technique for tracking targets, it is not possible to use directly. In order to implement a target tracker, a specialised version that makes further assumptions of the filter is needed. The *Kalman Filter* is a closed-form solution to this problem, where we are assuming generally Gaussian distributions for stochastic processes and initialization, in addition to linear models that are known. To summarize, we are given:

$$\begin{aligned} p(\boldsymbol{x}_k | \boldsymbol{x}_{k-1}) &= \mathcal{N}(\boldsymbol{x}_k; \mathbf{F}\boldsymbol{x}_{k-1}, \mathbf{Q}), \\ p(\boldsymbol{z}_k | \boldsymbol{x}_k) &= \mathcal{N}(\boldsymbol{z}_k, \mathbf{H}\boldsymbol{x}_k, \mathbf{R}), \\ p(\boldsymbol{x}_0) &= \mathcal{N}(\boldsymbol{x}_0, \hat{\boldsymbol{x}_0}, \mathbf{P}_0). \end{aligned} \tag{5.19}$$

$\mathbf{H}$ and $\mathbf{F}$ are in this case known as the linear measurement and process model matrices, with corresponding covariances $\mathbf{R}$ and $\mathbf{Q}$ driven by i.i.d white noise vector. Using these assumptions with the product identity (5.7), one can derive from the Bayesian Filter (5.3) a set of equations that can be implemented as an algorithm (Algorithm 1).

This creates an recursive algorithm, that initializes with $\boldsymbol{x}_0$, $\mathbf{P}_0$, e.i we need to know the initial targets state estimations and corresponding distributions in order to track the object. After predicting the first step, the latest target state measurements $\boldsymbol{z}_k$ is used to perform the algorithms correction step, before an endless recursion is achieved by feeding the return values into another function call. To separate the variables time instances in the prediction step from the correction step, the notation $k|k-1$ is used for prediction.

Algorithm 1 Kalman Filter Algorithm

Require: $\mathbf{H}, \mathbf{F}, \mathbf{Q}, \mathbf{R}, \boldsymbol{x}_0, \mathbf{P}_0$

procedure KALMANFILTER($\hat{\boldsymbol{x}}_{k-1}, \mathbf{P}_{k-1}, \boldsymbol{z}_k$)
  Prediction Step:
  $\hat{\boldsymbol{x}}_{k|k-1} \leftarrow \mathbf{F}\hat{\boldsymbol{x}}_{k-1}$
  $\hat{\boldsymbol{z}}_{k|k-1} \leftarrow \mathbf{H}\hat{\boldsymbol{x}}_{k|k-1}$
  $\mathbf{P}_{k|k-1} \leftarrow \mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^T + \mathbf{Q}$
  Correction Step:
  $\mathcal{V}_k \leftarrow \boldsymbol{z}_k - \hat{\boldsymbol{z}}_{k|k-1}$
  $\mathbf{S}_k \leftarrow \mathbf{H}\mathbf{P}_{k|k-1}\mathbf{H}^T + \mathbf{R}$
  $\mathbf{W}_k \leftarrow \mathbf{P}_{k|k-1}\mathbf{H}^T\mathbf{S}_k^{-1}$
  $\hat{\boldsymbol{x}}_k \leftarrow \hat{\boldsymbol{x}}_{k|k-1} + \mathbf{W}_k\mathcal{V}_k$
  $\mathbf{P}_k \leftarrow (\mathbf{I} - \mathbf{W}_k\mathbf{H})\mathbf{P}_{k|k-1}$
  return $\hat{\boldsymbol{x}}_k, \mathbf{P}_k, \mathcal{V}_k, \mathbf{S}_k$

For the observant reader, many parallels between the equations in the algorithm and the conditions from the Gaussian product identity (5.8) can be drawn. This simply comes as a consequence when deducing the Kalman Filter equations using the product identity, the Kalman Filter inherits similar conditions [16]. As with the Bayesian Filter, the algorithm can also be broken down to a prediction and correction step (5.18).

Extended Kalman Filter

Due to the nonlinear nature of target tracking, the assumptions of the Kalman Filter is often to strict to handle real case events. An easy solution to this is to soften the assumption regarding linear models, treating both the process and measurement models as the nonlinear functions $\mathbf{f}(\boldsymbol{x}_{k-1})$ and $\mathbf{h}(\boldsymbol{x}_k)$ respectively. There exists several ways to linearize these functions. For the Extended Kalman Filter approach, the previous state estimations $\hat{\boldsymbol{x}}_{k-1}$ and $\hat{\boldsymbol{x}}_{k|k-1}$ is used as linearization points in the first order *Taylor approximation* of the nonlinear functions:

$$\begin{aligned} \mathbf{f}(\boldsymbol{x}_{k-1}) &\approx \mathbf{f}(\hat{\boldsymbol{x}}_{k-1}) + \mathbf{F}(\hat{\boldsymbol{x}}_{k-1})(\boldsymbol{x}_{k-1} - \hat{\boldsymbol{x}}_{k-1}), \\ \mathbf{h}(\boldsymbol{x}_k) &\approx \mathbf{h}(\hat{\boldsymbol{x}}_{k|k-1}) + \mathbf{H}(\hat{\boldsymbol{x}}_{k|k-1})(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_{k|k-1}), \end{aligned} \tag{5.20}$$

where the $\mathbf{H}$ and $\mathbf{F}$ becomes the Jacobians calculated from the non-linear functions:

$$\begin{aligned} \mathbf{F}(\hat{\boldsymbol{x}}_{k-1}) &= \frac{\partial}{\partial \boldsymbol{x}_{k-1}}\mathbf{f}(\boldsymbol{x}_{k-1})\bigg|_{\boldsymbol{x}_{k-1}=\hat{\boldsymbol{x}}_{k-1}} \\ \mathbf{H}(\hat{\boldsymbol{x}}_{k|k-1}) &= \frac{\partial}{\partial \boldsymbol{x}_k}\mathbf{h}(\boldsymbol{x}_k)\bigg|_{\boldsymbol{x}_k=\hat{\boldsymbol{x}}_{k|k-1}} \end{aligned} \tag{5.21}$$

This gives a new set of Gaussian processes:

$$\begin{aligned} p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}) &= \mathcal{N}(\boldsymbol{x}_k - \mathbf{f}(\hat{\boldsymbol{x}}_{k-1}); \mathbf{F}(\boldsymbol{x}_{k-1} - \hat{\boldsymbol{x}}_{k-1}), \mathbf{Q}), \\ p(\boldsymbol{z}_k|\boldsymbol{x}_k) &= \mathcal{N}(\boldsymbol{z}_k, \mathbf{h}(\boldsymbol{x}_k), \mathbf{R}), \\ p(\boldsymbol{x}_0) &= \mathcal{N}(\boldsymbol{x}_0, \hat{\boldsymbol{x}}_0, \mathbf{P}_0). \end{aligned} \tag{5.22}$$

Since no changes are made to the Gaussian assumptions, and we still use linear models when dealing with the product identity (5.7), the Extended Kalman Filter uses a similar deduction from the Bayesian Filter as the linear Kalman Filter. Because of this, Algorithm 2 only differs in the prediction step of Algorithm 1

Algorithm 2 Extended Kalman Filter Algorithm

Require: $\mathbf{h}, \mathbf{f}, \mathbf{Q}, \mathbf{R}, \boldsymbol{x}_0, \mathbf{P}_0$

procedure ExtendedKalmanFilter($\hat{\boldsymbol{x}}_{k-1}, \mathbf{P}_{k-1}, \boldsymbol{z}_k$)

Prediction Step:

$\hat{\boldsymbol{x}}_{k|k-1} \leftarrow \mathbf{f}(\hat{\boldsymbol{x}}_{k-1})$

$\hat{\boldsymbol{z}}_{k|k-1} \leftarrow \mathbf{h}(\hat{\boldsymbol{x}}_{k|k-1})$

$\mathbf{F} \leftarrow \frac{\partial}{\partial \boldsymbol{x}_{k-1}} \mathbf{f}(\boldsymbol{x}_{k-1})|_{\boldsymbol{x}_{k-1}=\hat{\boldsymbol{x}}_{k-1}}$

$\mathbf{H} \leftarrow \frac{\partial}{\partial \boldsymbol{x}_k} \mathbf{h}(\boldsymbol{x}_k)|_{\boldsymbol{x}_k=\hat{\boldsymbol{x}}_{k|k-1}}$

$\mathbf{P}_{k|k-1} \leftarrow \mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^T + \mathbf{Q}$

Correction Step:

$\mathcal{V}_k \leftarrow \boldsymbol{z}_k - \hat{\boldsymbol{z}}_{k|k-1}$

$\mathbf{S}_k \leftarrow \mathbf{H}\mathbf{P}_{k|k-1}\mathbf{H}^T + \mathbf{R}$

$\mathbf{W}_k \leftarrow \mathbf{P}_{k|k-1}\mathbf{H}^T\mathbf{S}_k^{-1}$

$\hat{\boldsymbol{x}}_k \leftarrow \hat{\boldsymbol{x}}_{k|k-1} + \mathbf{W}_k\mathcal{V}_k$

$\mathbf{P}_k \leftarrow (\mathbf{I} - \mathbf{W}_k\mathbf{H})\mathbf{P}_{k|k-1}$

return $\hat{\boldsymbol{x}}_k, \mathbf{P}_k, \mathcal{V}_k, \mathbf{S}_k$

5.1.3 Tracker

Kalman filters alone are not much of use for tracking without a system categorizing where the measurements originated from. This is the purpose of a tracker, and the problem they solve is known as a data association problem. As with the filters there exists multiple trackers. The tracker we will be using is the *Joint Integrated Probabilistic Data Association* JIPDA, made by Øystein Kaarstand Helgesen in his masters's thesis [28]. This is known as a multi-target tracker, which relies on a statistical approach to solve the data association problem, by considering all the probabilities of possible association events.

Validation gate

Since this can be a large task due to the size of the measurement space, especially when multiple targets are present, a validation gate is used to limit the number of possible associations. By only considering measurements in the vicinity of the predicted measurements of the filter, a smaller space is formed by introducing the following limit:

$$\mathcal{V}_k^T \mathbf{S}_k^{-1} \mathcal{V}_k < g^2 \tag{5.23}$$

In our case, $g = 1$ due to the small distances between targets in the Ravnkloa channel.

JIPDA assumptions

Further assumptions must also be made to solve the data association problem.

- Tracks are assumed to be already initialized for targets in question.
- Then by treating the existence of a targets track as an event with a calculable probability, tracks can be either maintained or terminated. For the JIPDA, we assume three possibilities for existences of a target: the target exists and is visible, the target exists but is not visible, the target does not exist.
- The visibility is determined by measurements of the target, assumed to occur with a specified probability.
- At most one measurement can be associated with a target.
- The remaining measurements is known as clutter, e.i assumed to be false alarms.
- Clutter is i.i.d of each other.

These assumptions give rise to a multitude of formulas and tuning parameters for the tracker. For the sake of reproducibility, the reader is referred to Table 6.5 in Øysteins Master's thesis [28] for these parameters, and Chapter 5.3 in the same thesis for deriving the JIPDA formulas.

Extended Kalman Filter

JIPDA alone will only make tracks continue to live or die of, not creating new ones. Track initialisation is quite a different topic, but is necessary to fulfill the assumptions of both the JIPDA and the Kalman filters in order to give values to $\mathbf{P}_0$ and $\hat{\boldsymbol{x}}_0$. In our case, tracks initializes after two unassosiated measurements within a set distance occurs. This distance depends on the targets max velocity and measurement timestep: $d = Vel_{max} \cdot \Delta T$. In our case, the max velocity is set to 10 m/s, while the time step varies around 100ms due to the sensor rates on cameras and the lidar.

The remaining requirements for the Kalman filter are the process and measurement models, respectfully, $\mathbf{f}$ and $\mathbf{h}$. A constant velocity model is used for the process model (Chapter 6.4.1 in [28]), a bearing model is used for the camera sensor (Chapter 2.1.8 in [28]) model, while a range-bearing model is used for the lidar sensor model (Chapter 6.4.2 in [28]). These models are tuned using the diagonal elements of the Kalman filters noise matrices, or the single scalar for tuning the cameras bearing model:

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}_{11} & 0 \\ 0 & \mathbf{Q}_{22} \end{bmatrix}, \quad \mathbf{R}_{\text{li}} = \begin{bmatrix} \mathbf{R}_{\text{li},11} & 0 \\ 0 & \mathbf{R}_{\text{li},22} \end{bmatrix}, \quad \mathbf{R}_{\text{c}} = \mathbf{R}_{\text{c},11} \tag{5.24}$$

5.1.4 Performance Metrics

The thesis have until this point made us aware of the Kalman Filter, its role in target trackers, and discussed some of the design parameters these entail. This leaves us with explaining how the remaining parameters $\mathbf{R}$ and $\mathbf{Q}$ are tuned in order for the filter outputs to give consistent errors (*filter consistency*). From [16], this can be done by studying several criteria:

1. The state errors should be acceptable as zero mean.
2. The state errors should have magnitude commensurate with the state covariance yielded by the filter.
3. The innovations should be acceptable as zero mean.
4. The innovations should have magnitude commensurate with the innovation covariance yielded by the filter.
5. The innovations should be acceptable as white.

Point 3-5 handles internal consistency of the filter, namely the the prediction step, and are often used when ground truth values for filter estimates are either unavailable or of poor quality. In our case, Section 3.3.2 ensures a viable ground truth, at least for Havfruen. In this case the filter consistency can be determined by point 2, by using the *normal estimation error squared* (NEES) metric. In addition, the *root mean square error* (RMSE) is used to determine the accuracy of the estimates $\hat{\boldsymbol{x}}_k$ with respect to the ground truth.

RMSE

The most intuitively metric is looking on the physical distance between the state estimates and their corresponding ground truth $\boldsymbol{x}_r$

$$\text{RMSE}(\hat{\boldsymbol{x}}_k, \boldsymbol{x}_r) = \sqrt{(\hat{\boldsymbol{x}}_k - \boldsymbol{x}_r)^T(\hat{\boldsymbol{x}}_k - \boldsymbol{x}_r)} \tag{5.25}$$

NEES

NEES is based on the Mahalanobis distance squared, which can be intuitively understood as normalising the estimates on the standard deviation. This results in values that can be interpreted as the *filter confidence* for its state estimations.

$$\text{NEES}(\hat{\boldsymbol{x}}_k, \boldsymbol{x}_r, \mathbf{P}_k) = (\hat{\boldsymbol{x}}_k - \boldsymbol{x}_r)^T\mathbf{P}_k^{-1}(\hat{\boldsymbol{x}}_k - \boldsymbol{x}_r) \tag{5.26}$$

ANEES

Often studying raw NEES values for each timestep k can be a tedious process. Therefore when comparing the performances between two different tunings, an averaged NEES value across all non-zero data elements N_k in a scenario are often used:

$$\text{ANEES} = \frac{1}{\text{N}_k}\sum_{k=0}^{\text{N}_k}\text{NEES}(\hat{\boldsymbol{x}}_k, \boldsymbol{x}_r, \mathbf{P}_k) \tag{5.27}$$

A special property of ANEES and NEES, are that they form a χ^2-distribution given that the filter model are correct. Strictly speaking, a second criteria is needed for

ANEES where the summarized NEES values ought to be a white process for this to be true. However, the deviations from the χ^2-distribution is seldom to large to make any difference.

This distribution can be used to form an upper and lower limit of where the values of ANEES and NEES should be present. Using a 5% confidence interval the following boundaries can be established [16]:

$$\begin{aligned} \text{ANEES}_{upper} &= \frac{1}{\text{N}_k}\chi^2(0.975, \text{N}_k\text{N}_\text{g})^{-1}, \\ \text{ANEES}_{lower} &= \frac{1}{\text{N}_k}\chi^2(0.025, \text{N}_k\text{N}_\text{g})^{-1}, \end{aligned} \tag{5.28}$$

where the distributions second parameter is known as the distributions degree of freedom, found by taking the product between the filters number of states N_g and number of non-zero data samples N_k. If ANEES is higher than the upper limit, we say the filter is acting overconfident while it being lower the filter is acting underconfident.

Track-Truth associations

The JIPDA tracker gives us multiple tracks coming from associating measurements to tracks, but it does not determine which ground truth is associated with what track. This association is done by letting each track subscribe to the closest ground truth within 10m. The track with the closest distance to a ground truth is considered to be the target boat. If no targets subscribe to a ground truth as time elapses, a track loss emerges, meaning that the performance metrics at this time instance become undefined as they cannot be calculated. For RMSE and NEES this mean we observe fractured tracking results, but for ANEES it simply ignores the undefined instances when summing, and letting N_k only contain the number of valid instances.

5.2 Dataset validation

In this section we will look at how the synthetic dataset from Chapter 4 is compared to the real dataset from Chapter 3. We will begin with describing the comparison method that introduces a comparison metric, clarifying the evaluation settup, and giving a evaluation strategy used throughout the chapter. In Section 5.2.2 we study how the comparison metric maps to the target trackers performance metrics from Section 5.1.4 using the lidar sensor. Finally, in Section 5.2.3 we give a qualitative analysis of how the image detection from [28] performs on the synthetic dataset.

5.2.1 Comparison method

In section 5.1.4 we became familiar with the most common metrics in evaluating the performance of target trackers. These relied on studying the output of the trackers Kalman filters, in addition to having a ground truth as a reference of the accuracy the filter presented. Unfortunately, the target performance metrics only study properties of single pdfs which in itself is useful from a tuning perspective, but not that much for comparing datasets. In order to compare the performance of a tracker running on a synthetic versus a real dataset (Figure 5.2), a more comprehensive metric is needed.

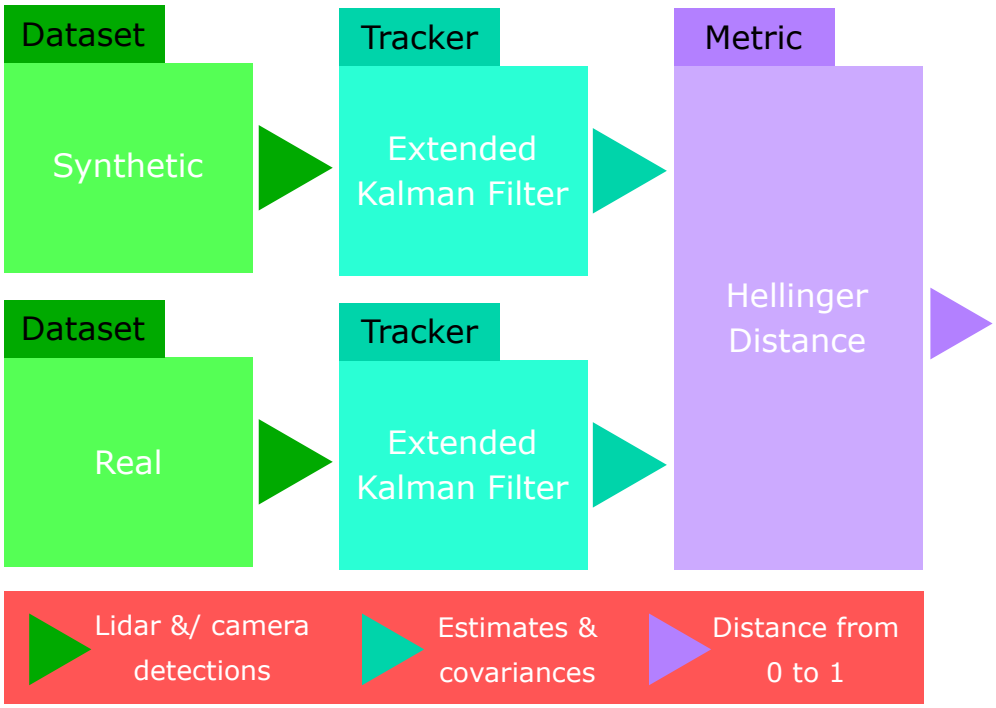


Figure 5.2: Dataset comparison method

Hellinger distance

The Hellinger distance HD is such a metric, which in its general form can be written as [29]:

$$HED(p_1, p_2) = \sqrt{1 - BC(p_1, p_2)}, \tag{5.29}$$

where p_1 and p_2 are two arbitrary density functions, and BC being the Bhattacharrya coefficient. The coefficient is special since it is both used as a metric but is also commonly used in other multivariate metrics such as the Bhattacharrya and Hellinger distances. If we assume the pdfs to be Gaussian, the coefficient simplifies giving us the following Hellinger distance [29]:

$$HED(\mathcal{N}_1, \mathcal{N}_2) = \sqrt{1 - \sqrt{\frac{\sqrt{|\mathbf{P}_1||\mathbf{P}_2|}}{\left|\frac{\mathbf{P}_1+\mathbf{P}_2}{2}\right|}} \exp\left\{-\frac{1}{8}\text{NEES}\left(\mu_1, \mu_2, \frac{\mathbf{P}_1 + \mathbf{P}_2}{2}\right)\right\}}, \tag{5.30}$$

where μ_i and $\mathbf{P}_i$ are the respective expected value and covariance matrix of the Gaussian pdfs $\mathcal{N}_i$ for $i = 1,\ 2$. Since it is based on NEES, it is easy to see the relationship the Hellinger distance have to the performance metrics for target trackings. The Hellinger distance does in contrary take into account the differences in covariances, easily seen by setting $\mu_1 = \mu_2$. This gives us the following term for the covariance differences:

$$COVDIFF(\mathcal{N}_1, \mathcal{N}_2) = \sqrt{1 - \sqrt{\frac{\sqrt{|\mathbf{P}_1||\mathbf{P}_2|}}{\left|\frac{\mathbf{P}_1+\mathbf{P}_2}{2}\right|}}}, \tag{5.31}$$

Further, the maximum distance of 1 is achieved for both COVDIFF and HED when either $\mathcal{N}_1$ or $\mathcal{N}_2$ gives a probability of zero while the other still has a non-zero probability. This can happen from either the covariance matrices being different, or from having large NEES values in HED [29]. When the distributions are equal, the distance is 0 under the opposite conditions.

Evaluation setup

The tracker outputs $\hat{\boldsymbol{x}}_k$ and $\mathbf{P}_k$ for a target boat comes from Algorithm 2:

$$\hat{\boldsymbol{x}}_k = [Pos_x, Vel_x, Pos_y, Vel_y]^T \in \mathbb{R}^4, \quad \mathbf{P}_k \in \mathbb{R}^{4\times4}, \tag{5.32}$$

where both x, y positions and velocities of the targets are estimated states giving us $\text{N}_\text{g} = 4$. To anchor these states to an absolute truth, the estimates are differentiated with the ground truth data $\boldsymbol{x}_r$, coming from the GNSS results in Chapter 3. Further, $\boldsymbol{x}_r$ follows the same convention as (5.32), using numerical differentiation estimates based on position to give ground truth data for the velocity states. Putting these values into (5.25) - (5.27) we calculate the performance metrics. Since each scenario is of different lengths, N_k is set accordingly to calculate the ANEES bounds (5.28).

For the dataset comparisons, there is no need for the ground truth as it cancels out when comparing the pdfs using the Hellinger distance. Instead, we form the

synthetic and real Gaussian distributions using the output parameters from the Kalman Filters directly:

$$\begin{aligned} \mathcal{N}_{synt,k} &= \mathcal{N}(\boldsymbol{x};\, \boldsymbol{x}_{k,synt},\, \mathbf{P}_{k,synt}), \\ \mathcal{N}_{real,k} &= \mathcal{N}(\boldsymbol{x};\, \boldsymbol{x}_{k,real},\, \mathbf{P}_{k,real}), \end{aligned} \tag{5.33}$$

where *synt* and *real* sub notation is used to separate the datasets. This is then used to calculate the Hellinger distance from Equation 5.30 as $HED(\mathcal{N}_{synt,k}, \mathcal{N}_{real,k})$.

To make the comparison easier, an averaged Hellinger distance is used for each scenario for both Havfruen and Finn:

$$AHED = \frac{1}{\mathrm{N}_{synt,real}} \sum_{k=0}^{\mathrm{N}_{synt,real}} HED(\mathcal{N}_{synt,k}, \mathcal{N}_{real,k}) \tag{5.34}$$

Such as with ANEES, invalid data may occur for AHED aswell. The difference being that the invalid time instances, does not only come from track-truth associations as in Section 5.1.4, but also from track differences between the synthetic and real datasets aswell. For this thesis, we choose to ignore the invalid time instances as we did with ANEES, only considering the occasions where there are common ground between between the datasets. This means that $\mathrm{N}_{synt,real} \leq \mathrm{N}_k$, and that $HED(\mathcal{N}_{synt,k}, \mathcal{N}_{real,k})$ is only summed over valid time instances. It is worth mentioning that AHED values will from this not be able to determine the amount of track losses.

Evaluation strategy camera

In addition to the Hellinger distance, camera images is compared to each other using a YOLO4 detector [30] trained on the *MS COCO* dataset [31], running on both real and synthetic images. This detector outputs bounding boxes of detected boats in an image, where a confidence interval between 0 and 1 is given, telling how certain the AI is of its prediction. Since the digital twin framework does not yet support an automatic image labeling process, moreover a method to compare images quantitatively, only a few samples are compared to give a qualitative impression of the data comparison and reproducibility.

Evaluation strategy lidar

The evaluation is done by fusing only lidar data, where the Extended Kalman filter is tuned based on scenario 3 from Section 3.2.4 with the help of the performance metrics discussed in Section 5.1.4. The tuning is performed twice by Øystein Karstad Helgesen, once using the synthetic dataset and once using the real dataset to see if there is any noticeable differences using different parameters. Rest of the Tracker parameters from Section 5.1.3 are kept constant.

Each tuning is given a AHED value foreach recorded Scenario, depicted in Figure 3.13. In addition three AHED values are calculated from the datasets, one for each tuning parameters mentioned above, and one for all tunings.

5.2.2 Lidar evaluation

The tracker was run on both Havfruen and Finn using only lidar data from the datasets in Figure 5.2, using tuning parameters from Table 5.1.

Tunings	$\mathbf{Q}_{11}$	$\mathbf{Q}_{22}$	$\mathbf{R}_{\text{li},11}$	$\mathbf{R}_{\text{li},22}$
Synthetic	0.3	0.3	56	0.044
Real	0.2	0.2	150	0.04

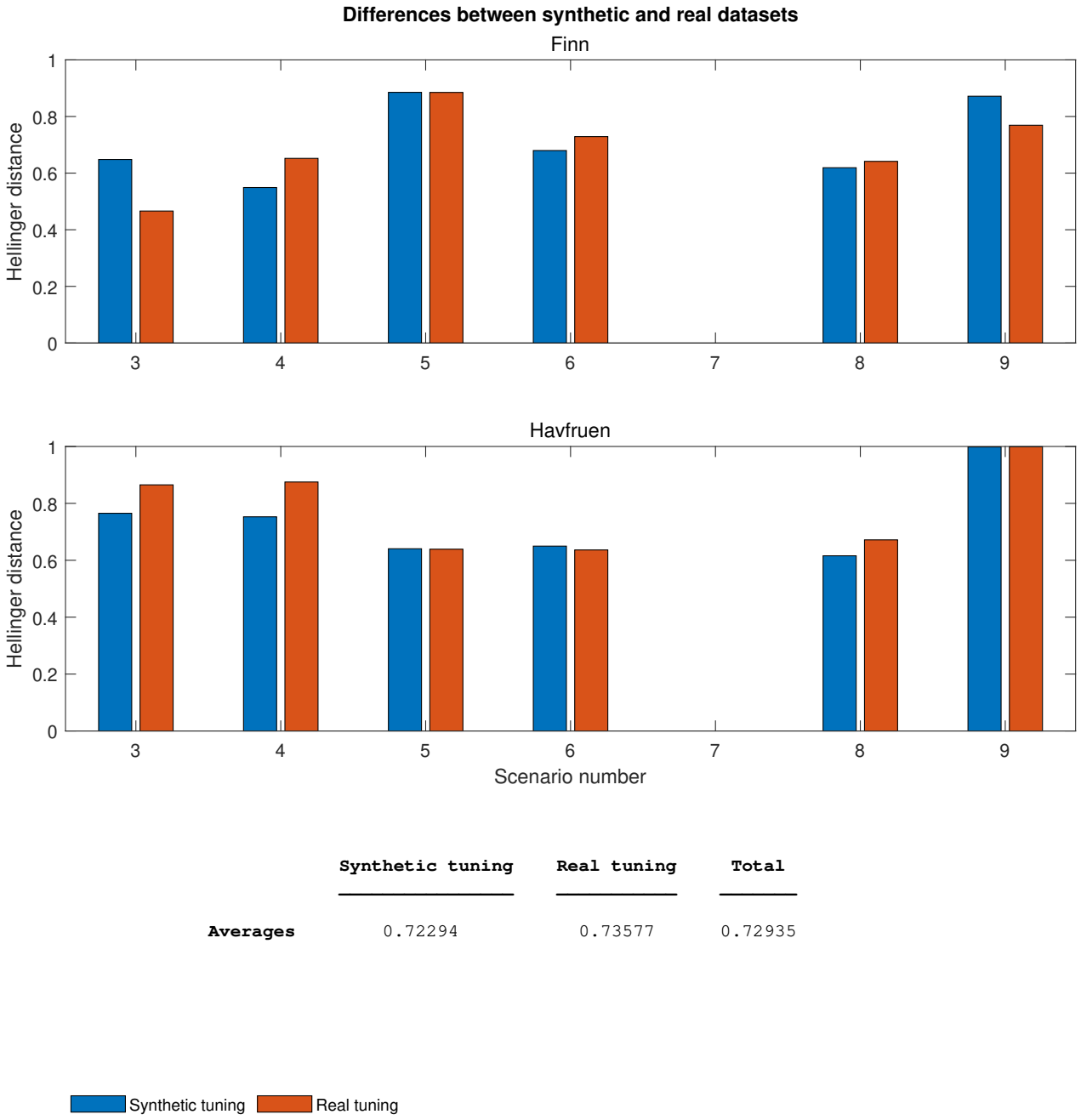
Table 5.1: Tracker tunings for Lidar evaluation

From these settings, results from the Hellinger distance was obtained, depicted in Figure 5.3. Studying the different tunings, and especially the averages, we see that there is not much differences in the distances. There are however exceptions to this, especially in scenario 3 for Finn, where the distance is at its lowest for the real tuning. This leaves us with the conclusion that tuning parameters do affect how similar the datasets are, but on average this difference is less than 1.3%.

A bigger concern is the size of the averaged distance when we disregard the different tunings, and the spread between the worst and best case distances. On average, we have a fairly high distance of ~ 0.73, indicating that the datasets are more dissimilar than similar. Further, the differences between the best and worst case distances are respectively, the real tuning of scenario 3 for Finn, with a distance of 0.43, and scenario 9 for Havfruen, with a distance of 0.99. Though this spread is concerning, it is also beneficial since it gives us the opportunity to study how different Hellinger distances affects the performance metrics.

The most similar scenario, are however 7, where both datasets agreed upon the tracker not being able to initialize a track on either Finn or Havfruen. It is however not much we can analyse from this, other than all the metrics being undefined at every time instance of the scenario. Since both trackers agree on there being no track to report, the Hellinger distance is set to 0 to indicate full agreement between the datasets.

Other scenarios that show interesting results are scenario 6 and 8 for Havfruen, even though this is not obvious from looking at the Hellinger distances. Instead, this comes from studying the performance metrics for the trackers in Figure 5.12 and 5.15. Therefore, the rest of this section will cover how NEES, RMSE and ANEES behaves for the four scenarios of interest.



	Synthetic tuning	Real tuning	Total
Averages	0.72294	0.73577	0.72935

Figure 5.3: Hellinger distances for each target boat using two sets of tuning parameters (Table 5.1)

Best Case

From scenario 3 the track of the Finn boat gave the best dataset comparisons, when the tuning was performed on the real dataset. This is controversially also the scenario which had the biggest deviation regarding the tunings (Figure 5.3), indicating the scenario's sensitivity to the tuning parameters. Regardless, the NEES and RMSE values for the real tuning seen in Figure 5.5 show an extraordinary case where most of the functions characteristics where reproduced. This is despite the problems the Finn boat had with establishing a trustworthy ground truth (Figure 3.16). If we have at Figure 5.4, it becomes obvious that the covariance matrices

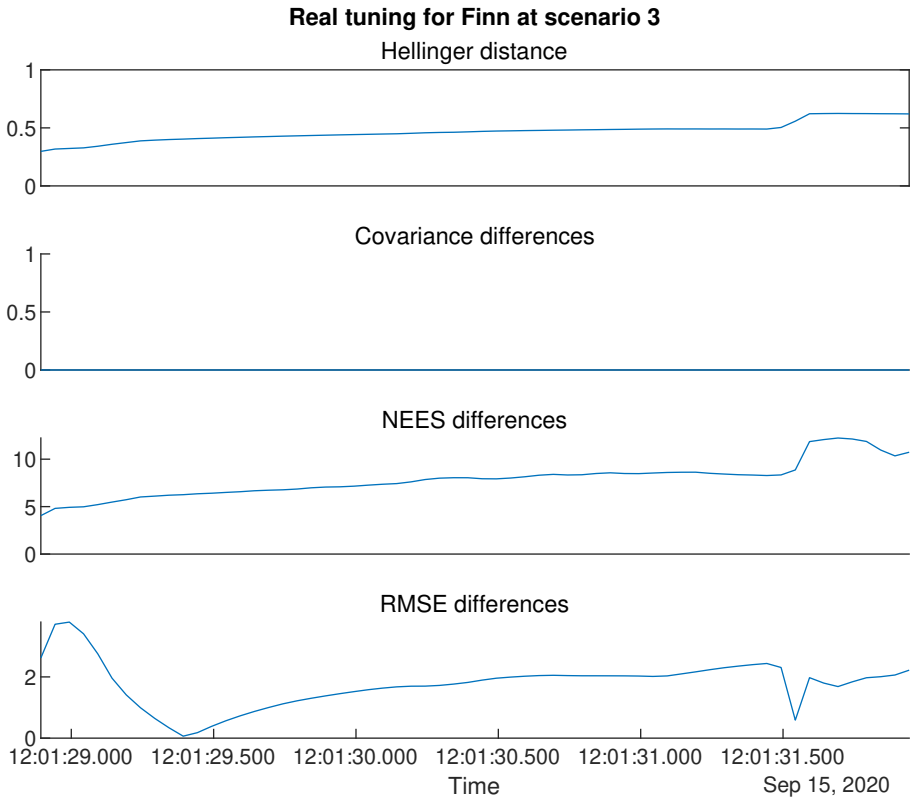
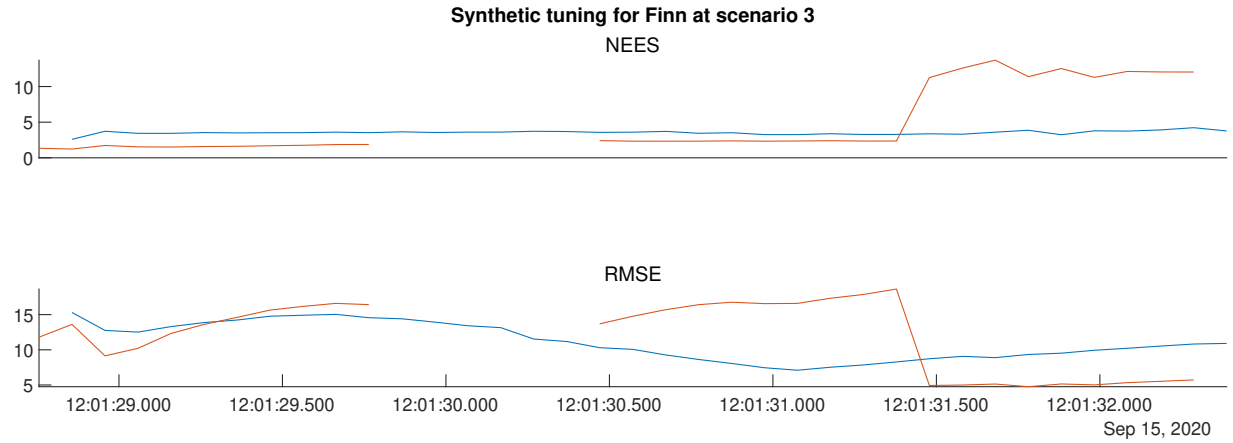
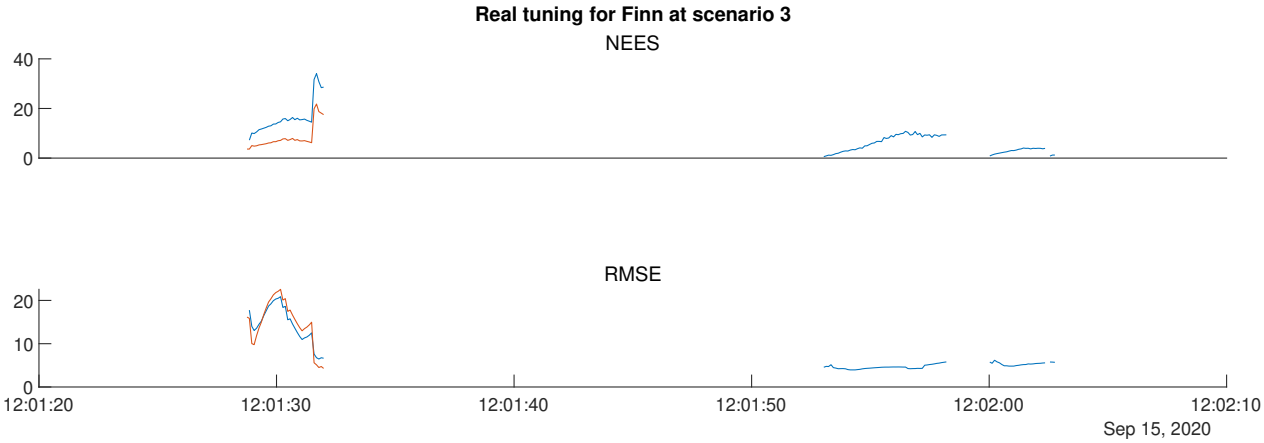


Figure 5.4: Hellinger distance with covariance differences(5.31). NEES and RMSE differences is calculated by using the RMSE of the corresponding synthetic and real dataset metrics

are similar in this case. Looking at NEES and RMSE differences we see that the Hellinger distance are driven by NEES. This is not a surprise since we know from (5.30) that when the covariance matrices are identical, NEES dictates the distance. This means that the remaining Hellinger distance are relatively low in this case, due to the low difference between the NEES values. However, much of this indicates a "lucky shoot" case, where a suitable tuning capable of disregarding the dataset differences could be the reason. Despite this, it is encouraging to see that the simulation in its current form is capable of producing such results.



	ANEES	Upper bound	Lower bound
Synthetic dataset	3.5353	4.6997	3.356
Real dataset	5.0075	4.6997	3.356



	ANEES	Upper bound	Lower bound
Synthetic dataset	8.4607	4.7341	3.327
Real dataset	8.2706	4.7341	3.327

Figure 5.5: Tuning metrics for Finn at scenario 3, using only lidar measurements

Worst Case

In contrary to the best case, the worst case shows a completely different story. Despite of the accurate ground truth for Havfruen (Figure 3.15), scenario 9 has by far the worst tracking results across all datasets. From the Hellinger distances (Figure 5.3), the worst case topped with a distance near 1, telling us the multivariate Gaussians strongly disagree with each other. In Figure 5.9 the synthetic RMSE and NEES show little resemblance to their real conterpart, and the ANEES values are in addition way of the χ^2-distribution bounds, regardless of the tunings. Looking at Figure 5.6 we get an impression of why this is so.

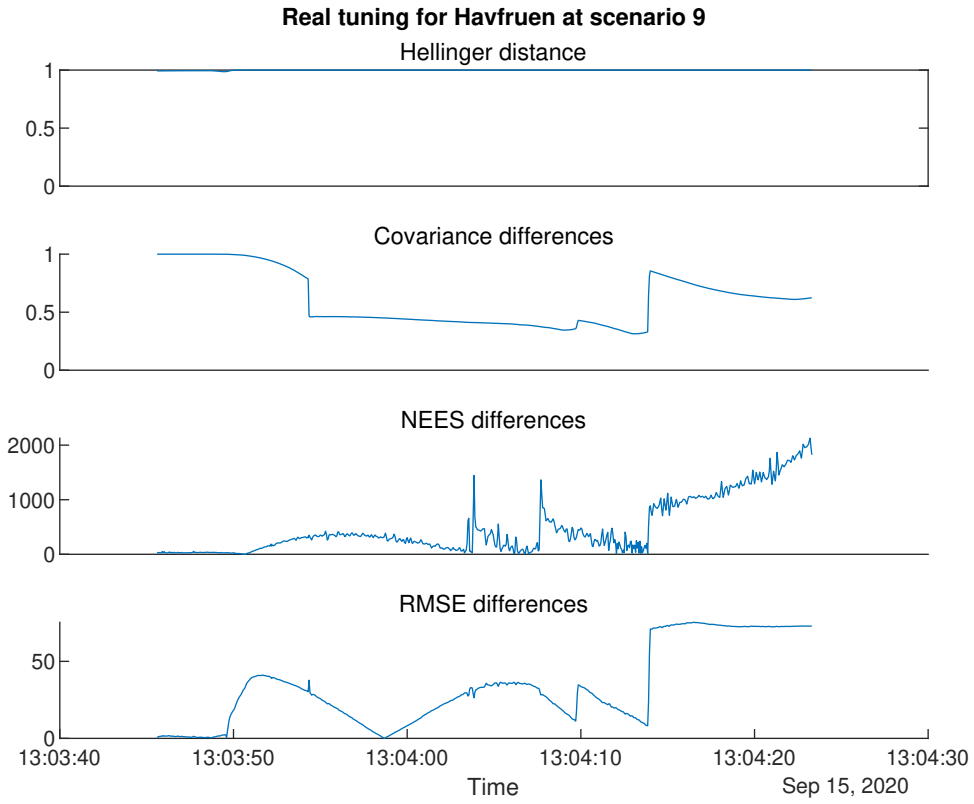


Figure 5.6: Hellinger distance with covariance differences(5.31). NEES and RMSE differences is calculated by using the RMSE between synthetic and real dataset metrics

Here the Hellinger distance tells us that the synthetic and real distributions are dissimilar for every timestep. In the beginning, we see that this has to do with the Covariance matrices being different. Then at 13:03:55, the tracker switches target for the synthetic dataset (Figure 5.7), creating a track jump that can be observed as a spike or drop in both the RMSE and covariance differences respectively (Figure 5.6). At this point, the covariance matrices seems to be more similar than before, but due to the high NEES differences, the Hellinger distance finds no similarities between the tracker results.

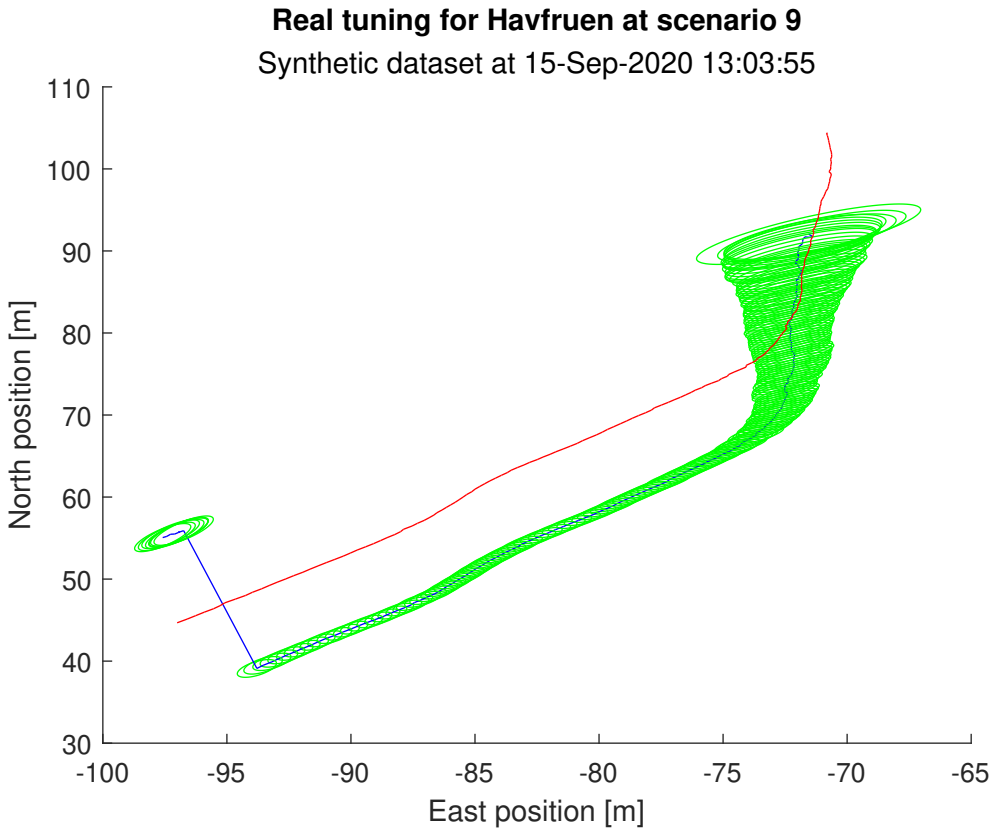


Figure 5.7: Track jump 10 seconds after first track initialisation for the synthetic dataset. Red line is ground truth data, blue is estimated positions while green indicates the covariance ellipses at each time instance. Track begins in the upper right corner of the image

Studying the covariance differences further, we see that target jumps happen between 2-3 times during the scenario. A possible cause for this can be the differences between the synthetic and real environment with respect to 3D models. Looking at Figure 5.8, we see that there are allot of possible targets the tracker could find interesting when trying to track Havfruen which travels closely besides the shoreline. Figure 5.7 also confirms this, as the jump happens towards the area of anchored boats. Considering the selection of boats in the environment between the two datasets are vastly different, the tracker might have jumped to a target witch is not present in the real world, and vice versa.

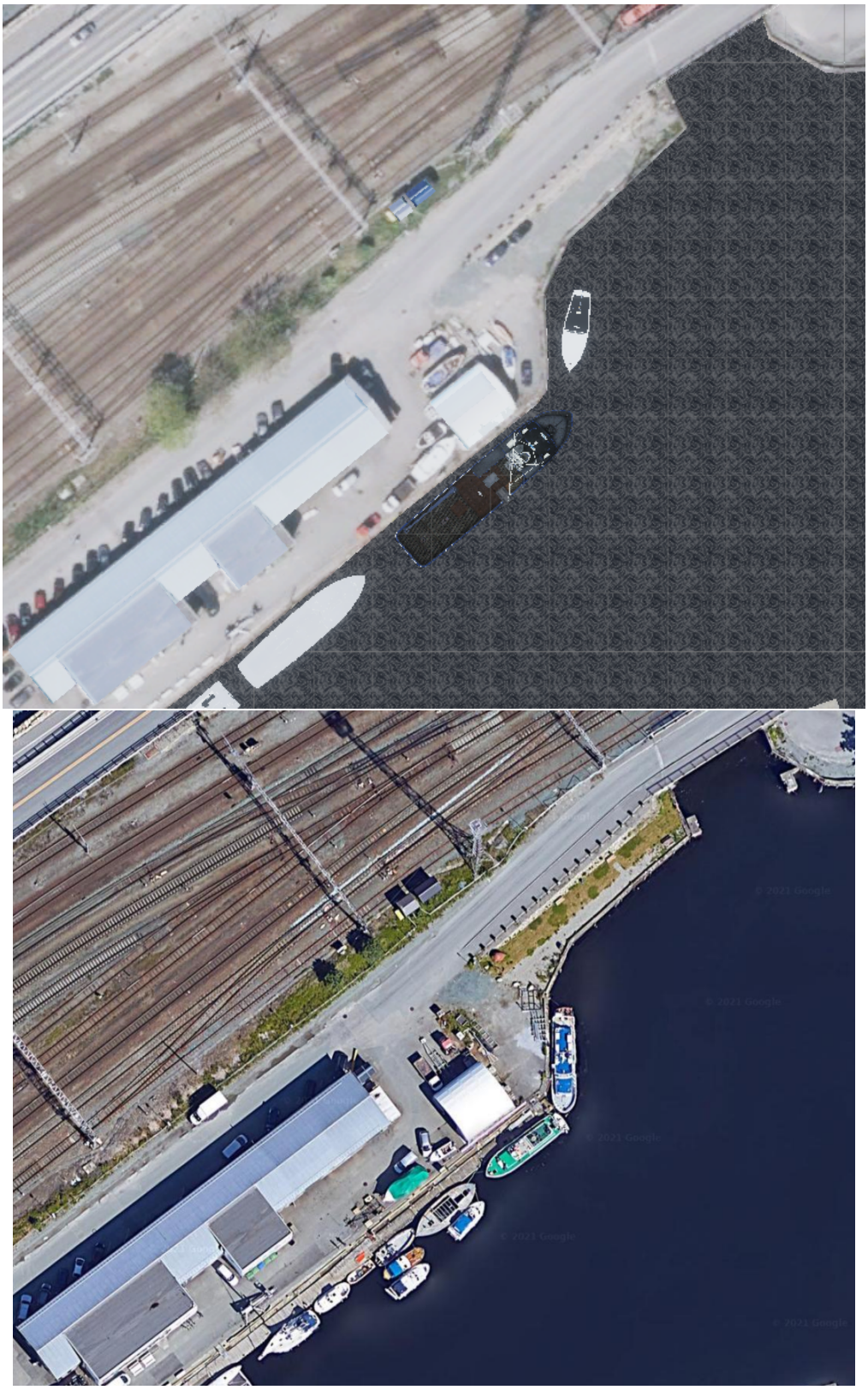
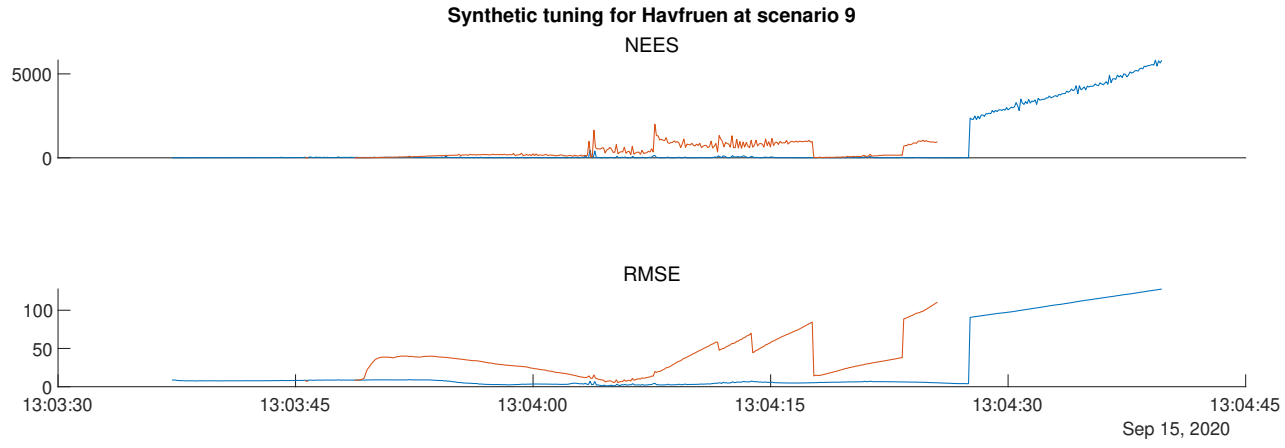
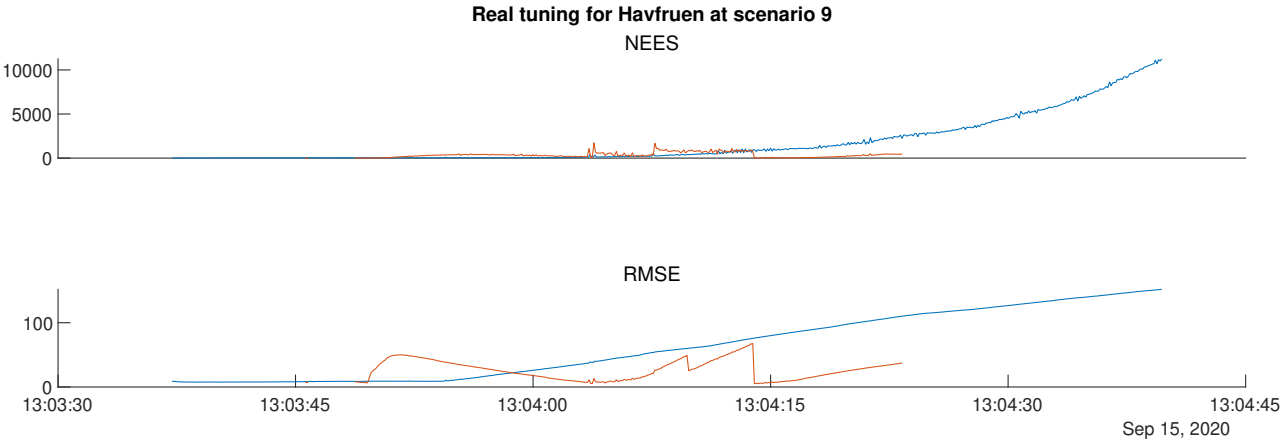


Figure 5.8: Synthetic and real environment from top to bottom respectively. Notice the difference in the types, sizes and number of boats anchored to the harbour



	ANEES	Upper bound	Lower bound
Synthetic dataset	779.97	4.1985	3.8063
Real dataset	402.8	4.1985	3.8063



	ANEES	Upper bound	Lower bound
Synthetic dataset	1809.8	4.2042	3.8008
Real dataset	353.92	4.2042	3.8008

Figure 5.9: Tuning metrics for Havfruen at scenario 9, using only lidar measurements

Range dissipation and track initialization Case

A reoccurring characteristics across most of the scenarios are the synthetic datasets early track initialisation and late track loss in comparison to the real dataset. This is both present in the worst case results (Figure 5.9) and in this sections Figure 5.12, showing metrics for Havfruen at scenario 6. We see from the figure that RMSE and NEES values initializes ~ 10 seconds earlier with the synthetic than with the real, causing different effects on the metrics.

The similar ANEES values between the datasets serves as a reminder of the importance of having multiple metrics to evaluate the data. In this case, ANEES ignores the characteristics of time, while NEES and RMSE clearly shows a difference in the datasets. Since we know that lidar is the only sensor in use that could influence these metrics, this means that the synthetic lidar is responsible for the early initialization.

When studying the point clouds in both datasets (Figure 5.10), it becomes obvious that the synthetic lidar does not model the max distance properly since the point cloud data of Havfruen is available at a greater distance for the synthetic than with the real. Moreover, when looking at the environment, we see that the range dissipation of the real lidar is not similar to the synthetic. One could suggest that this comes from inaccurate 3D modeling, which is to a certain extend true. However the dimensions of the channel, some of the nearby buildings and boat positions are accurate on a macro level. In addition, the 3D models of the target boats are especially accurate since they where modeled with this project in mind (Chapter 3). Since the lidar model uses the max distance given by the manufactures specifications, the inevitably conclusion is that the synthetic lidar does not model the range dissipation present in the real sensor.

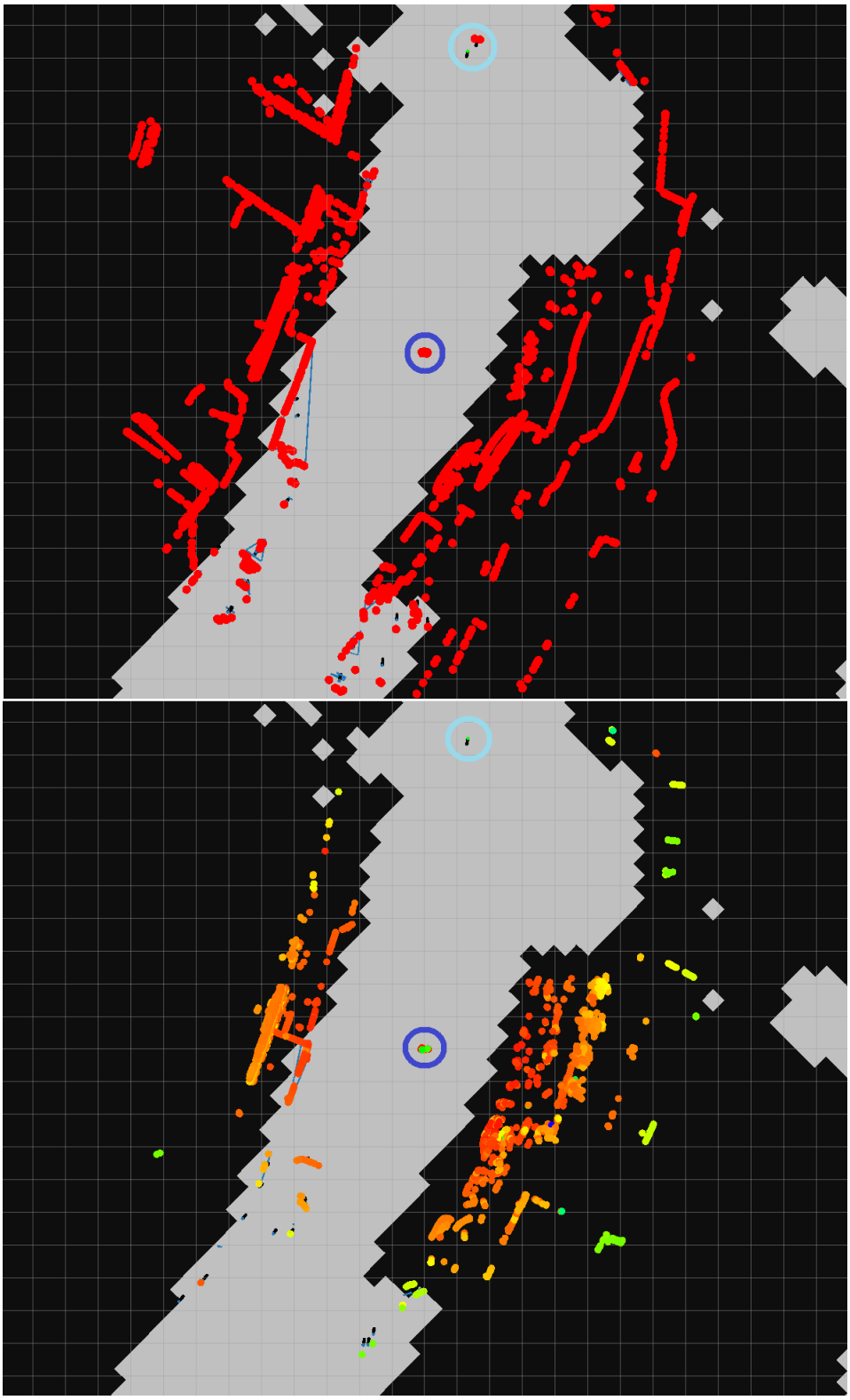


Figure 5.10: Comparison of raw point cloud data between the datasets at 12:36:20 for scenario 6. Synthetic at the top, real at the bottom. Dark blue circle shows milliAmpere's position, while light blue circle shows Havfruens position. The ground truth is depicted as a green dot inside Havfruens circle. The remaining points are raw cloud data. Notice the raw data being present in Havfruens ellipse for the synthetic data and not in the real

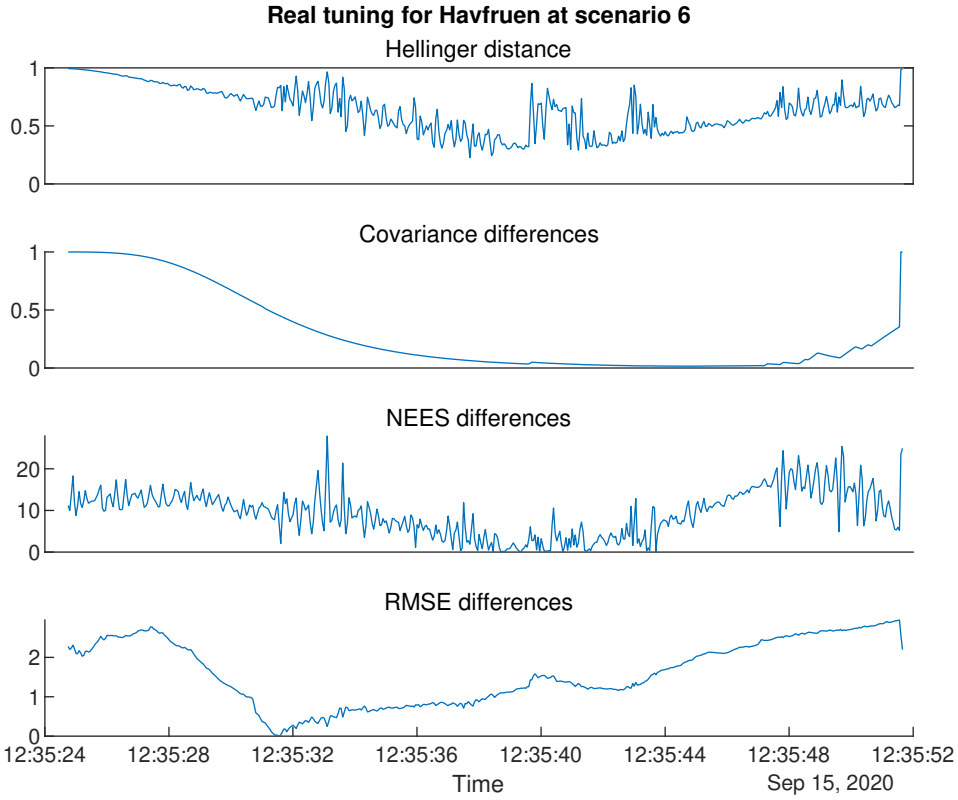
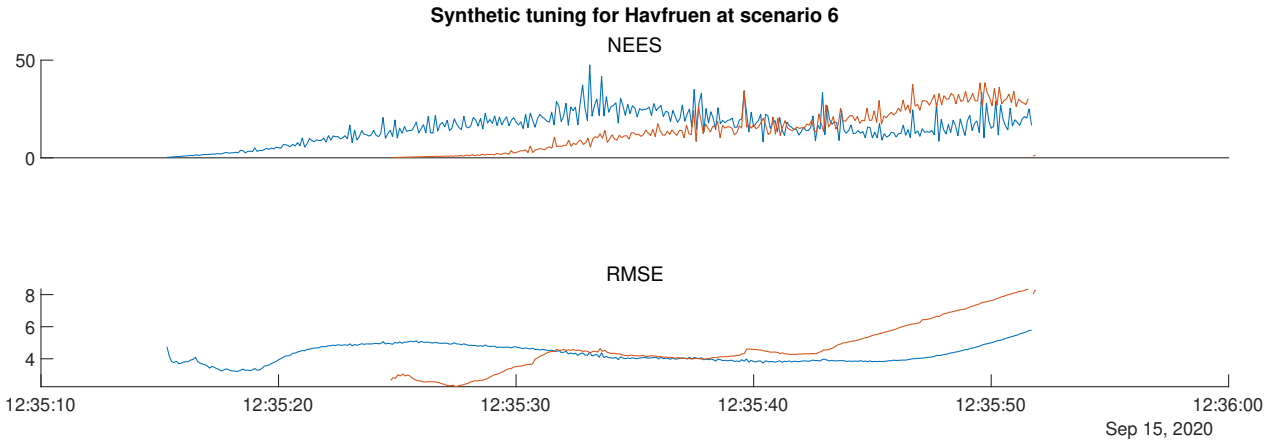
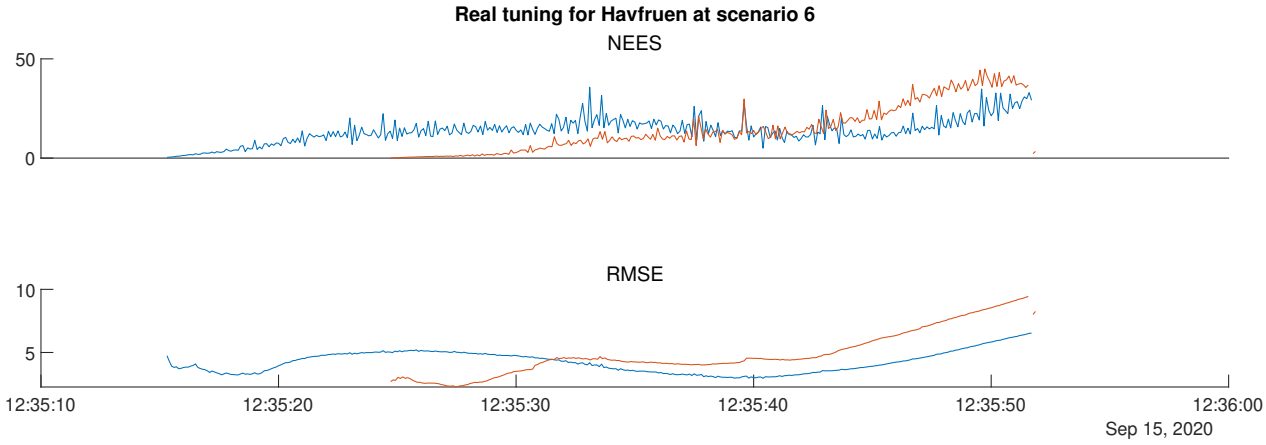


Figure 5.11: Hellinger distance with covariance differences (5.31). NEES and RMSE differences is calculated by using the RMSE between synthetic and real dataset metrics

Early initialization could also effect the whole track sequence, since we know from Section 5.1.2, that theoretically the estimation and covariance matrices should become more accurate as more data runs through the prediction and correction steps. From the assumption done using the Extended Kalman filter, this might not be completely true, but at the very least the previous calculations should affect the next outputs of the filter. Looking at Figure 5.12 much of this is confirmed when comparing the RMSE and NEES values for both the synthetic and real datasets, regardless of tunings. Here the synthetic cases ends up with having lower RMSE and more consistent NEES as time goes by. An exception from this is in the beginning of the real dataset, which might be from the assumptions just mentioned or a lucky initialization. Considering the latter case, looking back at the best case in Figure 5.5, we see that the initialization does indeed happen simultaneously for both datasets. If we compare the covariance differences between this case (Figure 5.11) and the best case (Figure 5.4) we see that the early initialization leads to huge differences in covariance similarities, which in turn impacts the Hellinger distance. This goes to show that early track initializations can be a major contributor to the differences between the datasets.



	ANEES	Upper bound	Lower bound
Synthetic dataset	15.509	4.2423	3.7648
Real dataset	14.531	4.2423	3.7648



	ANEES	Upper bound	Lower bound
Synthetic dataset	13.839	4.2423	3.7648
Real dataset	14.9	4.2423	3.7648

Figure 5.12: Tuning metrics for Havfruen at scenario 6, using only lidar measurements

Ground truth case

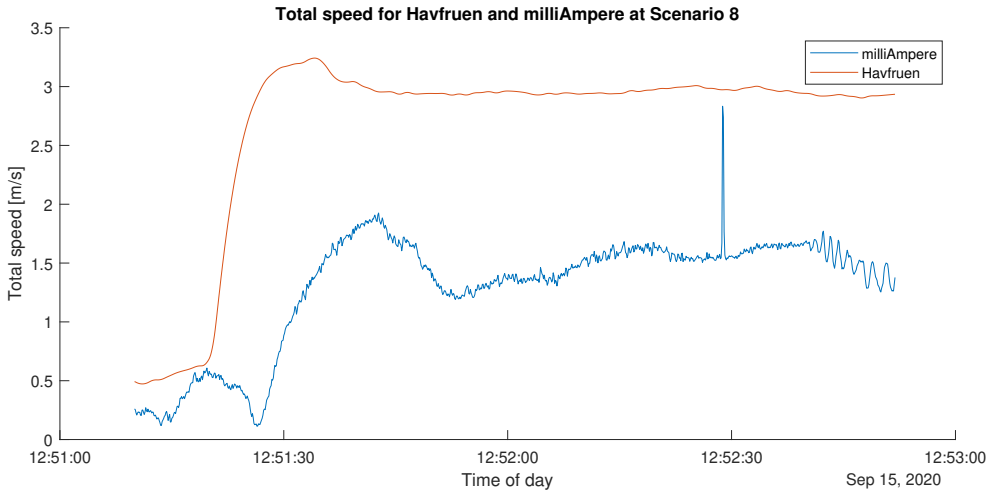


Figure 5.13: Speed estimates using positional GNSS data for the ownship and Havfruen from Section 3.3.2. Comparing this with Figure 5.15 servers as an example of how abnormalities in the ground truth propagates to different evaluation metrics

Until now we have seen dissimilarities between datasets focusing on sensor modelling of lidar. We have also seen that the best case are capable of reproducing most of the metric characteristics. However there have not been any discussion of where these characteristics could come from. Scenario 8 for Havfruen is an interesting case in this matter, since it displays a similar ability to reproduce NEES characteristics as with the best case. The discrepancy spike at 12:52:28 in Figure 5.15 is an example of this. Since we know RMSE and NEES both relies on the ground truth data, there is a possibility that non perfect ground truth data propagates to different evaluation metrics.

This could cause problems when we try to evaluate the characteristics the dataset metrics seems to show. To give an impression of what this means, the discrepancy spike in both NEES and RMSE, comes from a velocity jump from milliAmpere's ground truth data seen in Figure 5.13. This means that the characteristics seen in the best case from Figure 5.5 might just as well be the result of inconsistent ground truth data obtained from Finn (Figure 3.14). This opens up yet another question: If the characteristics we perceive from the metrics is just a result of inaccurate ground truths, how can we tell if a dataset comparison is any worse than another?

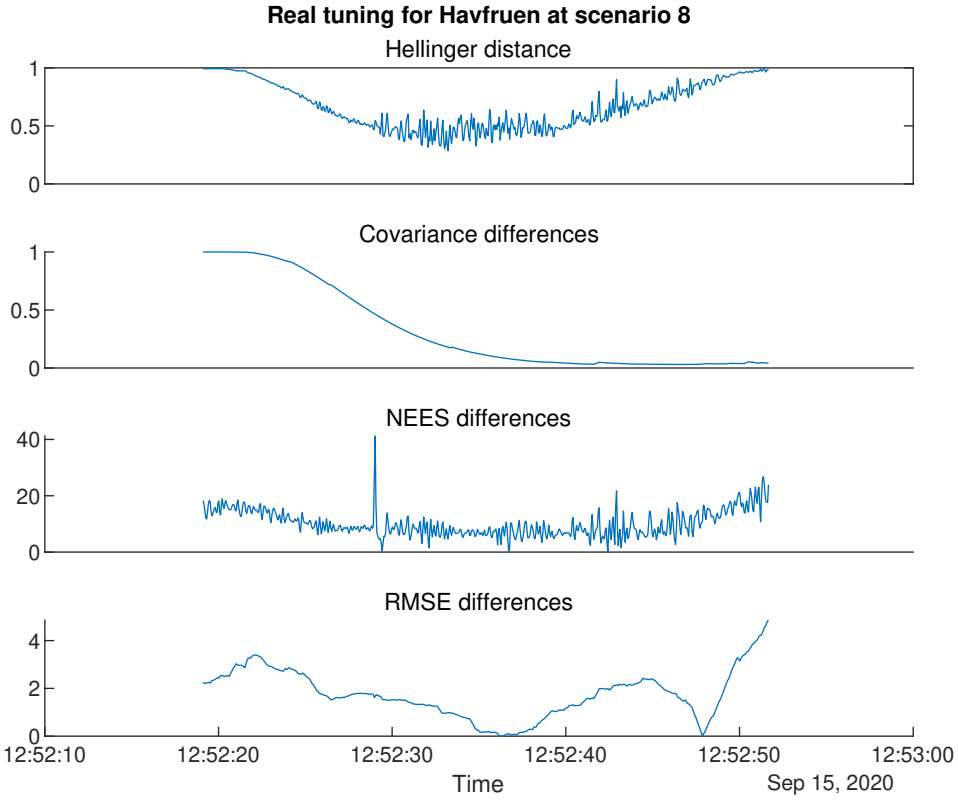
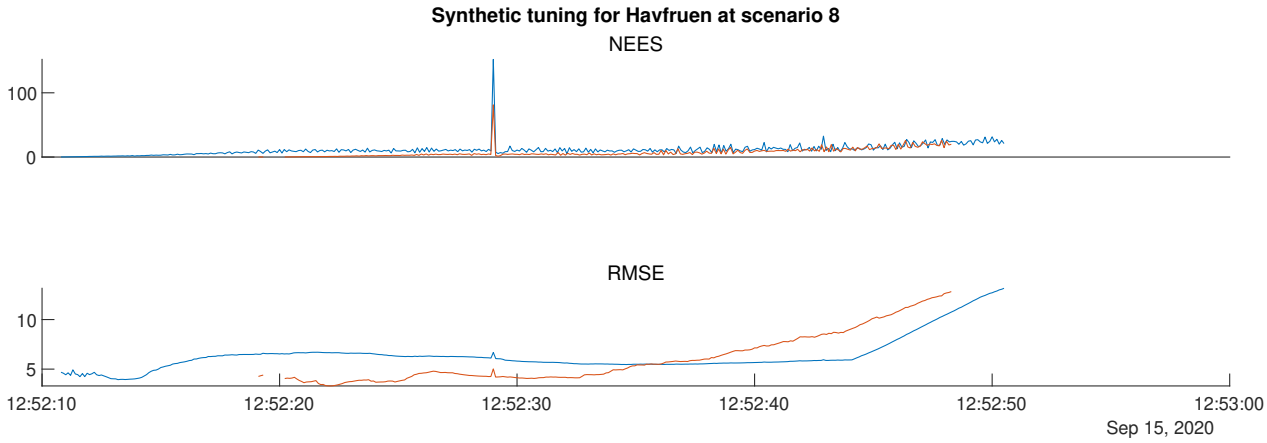
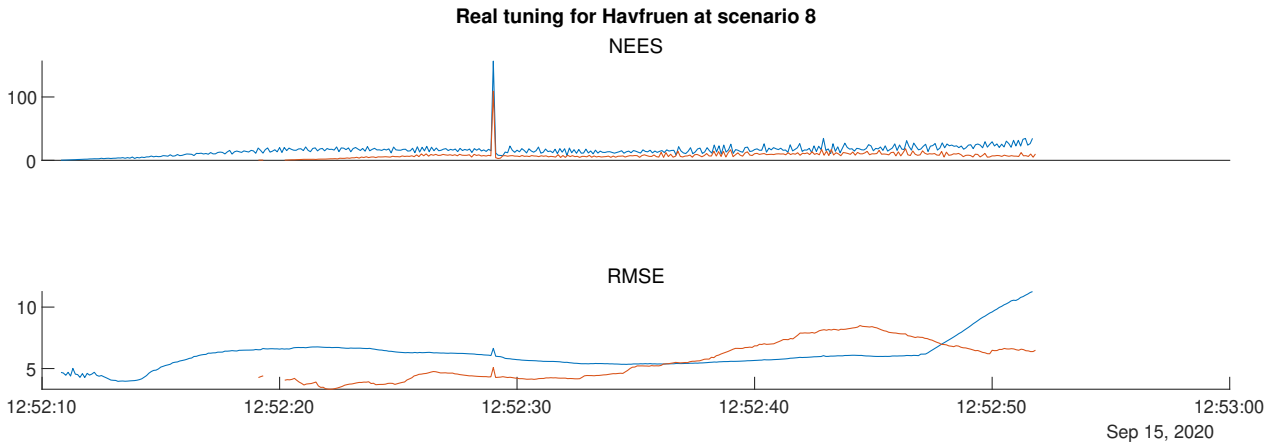


Figure 5.14: Hellinger distance with covariance differences (5.31). NEES and RMSE differences is calculated by using the RMSE between synthetic and real dataset metrics

Luckily, in comparison to NEES the Hellinger distance does not depend on the ground truth directly, which can be seen from the absent spike in Figure 5.14, meaning that some discrepancies coming from the ground truth does not affect the evaluation of the dataset similarities. Since the best case has a relatively low Hellinger distance, there is evidence that the similar characteristics comes from a successful dataset reproduction. With this said, its worth to keep in mind even if the Hellinger distance is not affected by the ground truth directly, it is affected by the synthetic data being generated from the ground truth. Taking this into account, the remaining differences we see in NEES for the best case, might be a result of Finn's poor ground truth containing more prevalent biases (Figure 3.16).



	ANEES	Upper bound	Lower bound
Synthetic dataset	11.186	4.2326	3.7739
Real dataset	7.2777	4.2326	3.7739



	ANEES	Upper bound	Lower bound
Synthetic dataset	15.742	4.22	3.7858
Real dataset	7.8735	4.22	3.7858

Figure 5.15: Tuning metrics for Havfruen at scenario 8, using only lidar measurements

5.2.3 Camera evaluation

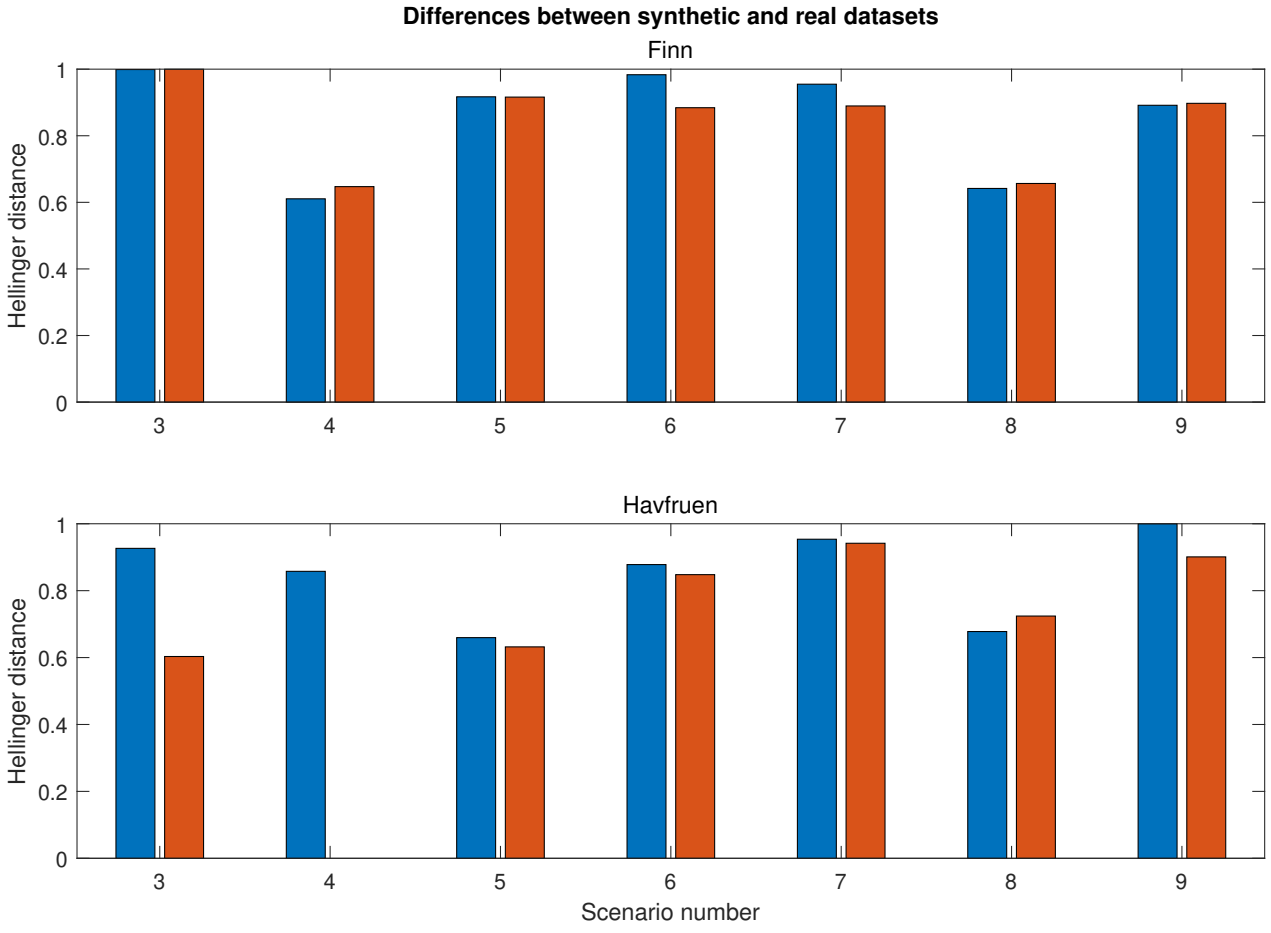
The tracker was run using both lidar and camera data, using the same tuning parameters for Lidar as from Table 5.1, with the only addition of a bearing parameter for the camera model (Table 5.2)

Tunings	$\mathbf{Q}_{11}$	$\mathbf{Q}_{22}$	$\mathbf{R}_{\text{li},11}$	$\mathbf{R}_{\text{li},22}$	$\mathbf{R}_{\text{c},11}$
Synthetic	0.3	0.3	56	0.044	0.09
Real	0.2	0.2	150	0.04	0.0026

Table 5.2: Tracker tunings for Lidar and EO fusion evaluation

Studying the differences in Hellinger distances between Figure 5.3 and 5.16 shows a overall worsening of the Hellinger distance by 0.1. This goes to show the introduction of a camera detector made the dataset similarities even worse, raising the question if this has something to do with the sensor fidelity of the camera sensor.

To answer this question the image detector from Section 5.2.1 was run on multiple scenarios where several interesting observations where made. Both similarities and differences between the datasets was found, using a qualitative analysis considering both the AI's bounding box perception, and geometry of the content in the scene.



	Synthetic tuning	Real tuning	Total
Averages	0.85365	0.81082	0.83303

Figure 5.16: Hellinger distances for each target boat using two sets of tuning parameters (Table 5.2)

Low confidence case

The camera detection of the target boats, is observed to have a generally lower confidence score for synthetic vs real images for most of the scenarios. Figure 5.17 shows an example of this, where Havfruen is detected with a confidence score of 0.35 and 0.64 for the synthetic and real datasets respectively. As lower confidence score does not necessary have an impact on what is being observed (e.i we still have a correct detection), the problem arises with the confidence threshold, used to determine if we have a detection or not. This might unfavorably drop more detections on the synthetic datasets, than the real. The generally high Hellinger distances across the scenarios seems to support this. An solution to this might just be to lower the treshold for synthetic data. This problem is however not always present, as seen on the confidence values on the two target boats on the left side of Figure 5.18.

Since the detector being used is only trained on real data, one might say this is an unfair comparison, and that the detector should have been trained on synthetic data as well. However, the purpose of this comparison, is to show that the real model the AI is based on, perceives missing features in the synthetic dataset. This shows that there are still room for improvements regarding the rendering process. Some of these differences might have something to do with the lightning conditions, as we see the synthetic images being considerably brighter than the real, while others might come from missing features in the 3D models.

Camera calibration and 3D model accuracy case

From both Figure 5.18 and 5.17, we see to a large degree the geometry of the target boats and environment to be well in sync with each other. After the calibration process mentioned in Section 4.2.1, there are however still differences, the first being the camera rotation. By using milliAmpere's antenna as a reference, we see the real image being slightly rotated clockwise. This is also observed by comparing the shorelines of Figure 5.17.

It is also questionable if the manual calibration of the cameras field of view is accurate, which can be seen by comparing the bounding box positions of the Finn boat in Figure 5.18. Comparing these images, it seems like the the synthetic have a slightly lower field of view in comparison to the real. We know from before that data from the Finn boat is not reliable, considering its inconsistent ground truths from Figure 3.16. With this said, if we compare the roof of the building in the upper left image corners in Figure 5.17, we see the same tendency of a slightly lower field of view. However, doing the same roof comparison to the red houses to the right, the field of view seems almost pixel perfect.

An answer to this could be insufficient accuracy in the 3D models of the environment, which does not come as a surprise as we can clearly see several missing buildings, trees, and other features in the synthetic images. This lack of content does however not seem to affect the detection of target boats.

Figure 5.17: Images taken of Havfruen from scenario 8 using the front left camera. The confidence values shows that the image detector have a poorer performance at detecting boats from the synthetic dataset.

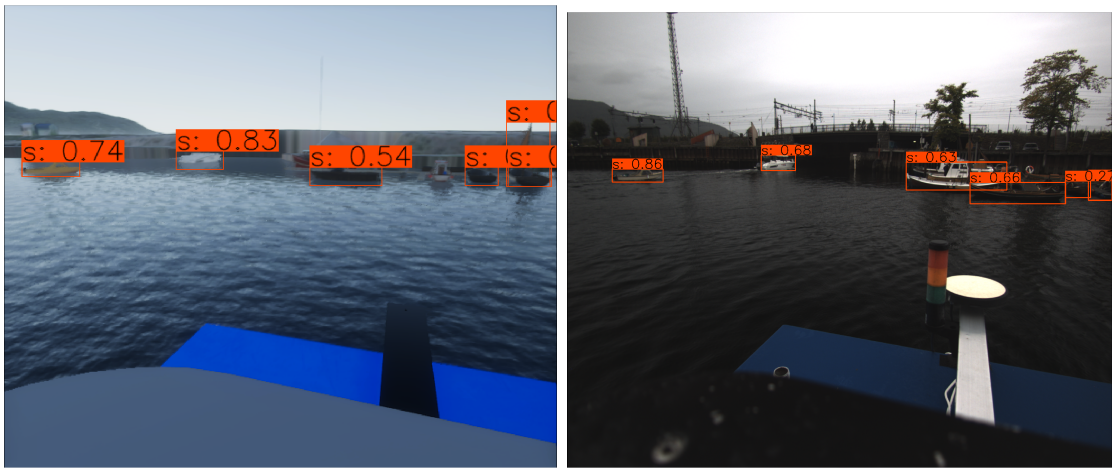


Figure 5.18: Images taken from scenario 6 using the front right camera. In this case, the detection probability is higher than in Figure 5.17. From left to right in each image: Finn, Havfruen, dummy boats

5.2.4 Results from a digital twins perspective

We have now seen the Hellinger distance been used to measure the similarities between datasets for multiple cases, interpreting some of the results from a target tracking perspective. In effect, Hellinger measures the degree we are able to reproduce the scenarios when comparing sensor data from the lidar and camera sensors, but also the accuracy of ship positions and surroundings. In effect, this is the whole simulated perception fidelity. However, from a digital twins point of view, being able to reproduce a scenario is not the biggest of interest. Rather, the motivation of using the digital twins in the first place, is to test AAs safety by simulating a large quantity of scenarios one would normally not be able to do in real life. This puts requirements on the sensor models fidelity rather than the whole simulated perception fidelity the Hellinger distance have measured. Since the sensor models are considered to be a part of the whole simulated perception, this leads to the question if the Hellinger distance is able to measure the sensor models fidelity.

From evaluating the different sensors in Section 5.2.2 and 5.2.3, it is apparent this is true to some degree. By fusing camera and lidar data in Section 5.2.3, the Hellinger distance increases in comparison from only using the lidar. This tells us that the camera data is sufficiently different to increase the difference between the datasets. Analysing the differences further, we observed a noticeable difference in the detectors confidence between real and synthetic data, indicating the rise of the Hellinger distance could come from a measurement of the cameras fidelity.

However, other cases that increases the Hellinger distance, such as track jumps and early initialisation seems to differ. The latter case was shown to be caused by improper modeling of the range dissipation for the lidar, meaning this was indeed a fidelity case concerning sensor modeling. The former case on the other hand, can not be connected to sensor modeling since it was likely caused by poor reproduction of the environment. This gives us an interesting weakness of the Hellinger distance, where large Hellinger distances can come from not reproducing the surroundings accurately, rather than being linked to the sensor models fidelity. On the contrary, when the surroundings are reproduced accurately, much indicates that the remaining Hellinger distance is a measurement of the sensor models fidelity. Unfortunately, we are not able to tell when we are met with the former or latter, without using other metrics such as NEES and RMSE in combination with studying raw data and scenario play-troughs.

Despite this, we are still able to analyse performance differences of the AA when considering the whole simulated perception fidelity. From Figure 5.16 and 5.3 we have seen that different tuning parameters causes differences in how the AA compares synthetic and real data. In other words, we are able to change its perception of what is similar and what is different between the datasets, just as we humans could do with the introduction images (Figure 5.1). This is especially noticeable when looking at the best case in Figure 5.5, where NEES and RMSE differences are small for the real tuning and large for the synthetic tuning. A point of further interest is therefore if we are able to design the AA perspective to pay less attention to the dataset differences, to rely less on the digital twins sensor model fidelity.

Chapter 6

Closing Remarks

6.1 Discussion

Simulation based verification is expected to play a vital role in certifying autonomous vessels in the coming decade. This is due to its scalability in test coverage, safe test environments, and cost effectiveness towards designing and assuring AA. In order for this to work, the simulation must show similar behaviour as one would expect from a real world implementation, this including exteroceptive sensors. There are today few ways of measuring the fidelity of these sensor models, moreover the differences they play in the behaviour of AA, in order to tell if a high fidelity digital twin framework could one day substitute the role of conventional verification.

The work demonstrated here, shows the implementation of a new digital twin framework, created for testing AA by using high fidelity sensor models. This was done by merging several platforms from existing work in the maritime field [12, 28, 8] to meet a similar standard as seen in automotive simulators [7], leading to a new test ground for marine applications driven by scenario inputs. In addition, this framework was designed with the intention to reproduce real world data by simulating EMR sensors using the Unity game engine, giving a way to measure the fidelity of sensor models. This is believed to be a step towards a trustworthy simulation based verification system.

To give a first impression of this process, recording of a real life dataset was done in Trondheim, including data from several EMR and ground truth sensors with the purpose of reproducing the experiment. In total 7 scenarios was recorded for test coverage, while ground truth data was analysed for establishing a notion of accuracy of the individual boats participating. The ground truth variation for the participating boats, equipment failure during testing that reduced the number of scenarios, and cameras being out of focus when later analysed, goes to show some of the imperfections of the recorded experiment.

From these recordings, a synthetic dataset was created in the same environment based on the ground truth data. We also improved the beam shape error by introducing a spherical projection filter, and implemented a simple ray drop model [13] to improve the lidar model in Autoferry Gemini [12]. From studying the depth buffer technique used in these simulations, a general error formula of the remaining numerical error was derived. By analysing the formula and making reasonable as-

sumptions, an equation for maximum error was obtained, which allowed us to make a guide in choosing good simulation parameters to minimize the error based on sensor type and use case. From this, the synthetic dataset recording was setup considering the real life sensor specifications and the hardware running the simulation. Due to a lack of proper documentation and calibration parameters regarding RGB cameras, some differences between synthetic and real images was seen. Similarly, with no documentation of the lidars characteristics towards ray drops and range dissipation, reproduced data was observed to be more perfect and precise in contrary to it real world counterpart.

Using the synthetic dataset and real dataset, a comparison was made by studying the outputs of a JIPDA tracker [28]. A Hellinger metric was used to compare the distributions coming from the Extended Kalman filter outputs, in addition to a YOLO detector running over synthetic and real images in order to give additional qualitative information of the similarities and differences. Using these evaluation tools, several different cases were observed involving both lidar and lidar-camera fusion. These cases showed that the Hellinger distance could be linked to measuring sensor fidelity, but that it was highly dependent on the surroundings being well reproduced. Different tuning parameters also seemed to change the AA perception of the dataset similarities, potentially giving large Hellinger differences for scenarios. In the end, the average distances across tunings and scenarios ranged from 0 to 1, ended up at 0.73 for lidar only fusion, and 0.83 for lidar-camera fusion.

6.2 Conclusion

In this thesis we have implemented a digital twin framework by enhancing Autoferry Gemini with better lidar models and connecting it towards other platforms to generate sensor data from scenario inputs. The framework have been tested by reproducing a synthetic dataset from a real life recording, studying the behaviour of a AA to compare several metrics in order to measure fidelity. From this we have seen how sensitive AA are towards synthetic data, and the difficulties in using this to determine sensor fidelity. This have shown that there are still work to be done in order to trust simulation of AA, but that we are now capable of measuring different aspects of fidelity, a stepping stone towards being able to improve the simulation.

6.3 Further work

The large Hellinger distances opens up many different paths for further work. Finding ways to lower the Hellinger distance is of utmost importance in order to trust simulations, but also analysing more what and how the Hellinger distance measures fidelity is just as important. The list below shows some of the potential directions future research can choose to address these concerns:

- The thesis have only considered the perspective of a JIPDA tracker which was shown to be sensitive to synthetic data. In order for a high fidelity simulator to have any validity, multiple perspectives from different AA algorithms must be tested as well.
- The Hellinger distance shows indications of being able to model sensor fidelity, but depends on how well the environment is reproduced. This leads to the question if the Hellinger distance is subject to other factors as well, such as biases in ground truth data used to create the synthetic dataset. Because of this, the Hellinger distance might not be used in an optimal manner. A step towards removing some of these factors, is to create new datasets which is recorded and reproduced synthetically, where the surrounding environment is not of any concern. This can typically be done in open waters. In addition, gathering a more precise and consistent ground truth off all ships can remove some of the potential biases or discrepancies from the participating ships. Preferably the same ground truth sensors should be used across all ships, with raw GNSS data available to obtain the best precision.
- Another reason for the large Hellinger distances comes from the lidar model. The range dissipation happening as light rays travels out in space is currently not modeled in the lidar. Improving the lidar model further is crucial in this case. This can be done by utilizing more information from the render pipeline to improve the fidelity of lidar rays, such as using the surface normals and reflection parameters in order to model the returned laser beams intensity and/or return rate. Empirical studies of the real lidar might also be necessary to see if any characteristics can be used in the simulator.
- The increase in Hellinger distance when using synthetic images has much to do with the dataset used to train the YOLO detector. Since the AAs behaviour depends both on its tuning parameters and the simulation fidelity, this open up the question if we should design the AA to have a more robust perception towards dataset differences, e.i focusing more on the dataset similarities in order to rely less on the sensor model fidelity. A first step to study this question is by creating an automated image and labeling pipeline for the digital twin framework, where synthetic data could be used for making the detector more confident in detecting boats from synthetic images.
- Another use case of the pipeline mentioned above, is to give ground truth data for bounding boxes. This can give quantitative measurements of the differences in the bounding box predictions for the synthetic and real datasets, instead of only relying on qualitative judgements done in the thesis.

Bibliography

[1] Brekke E F, Wilthil E F, Eriksen B O H, Kufoalor D K M, Helgesen Ø K, Hagen I B, Breivik M and Johansen T A 2019 The autosea project: Developing closed-loop target tracking and collision avoidance systems *Journal of Physics: Conference Series* **1357** 012020

[2] Zou J and Schiebinger L 2018 Ai can be sexist and racist — it's time to make it fair *Nature* **559** 324–326

[3] Dalsnes B R, Hexeberg S, Flåten A L, Eriksen B H and Brekke E F 2018 The neighbor course distribution method with gaussian mixture models for ais-based vessel trajectory prediction *2018 21st International Conference on Information Fusion* pp 580–587

[4] Ioannidis J 2005 Why most published research findings are false *Public library of science: medicine* **2** 124

[5] Bakdi A, Glad I, Vanem E and Engelhardtsen 019 Ais-based multiple vessel collision and grounding risk identification based on adaptive safety domain *Journal of Marine Science and Engineering* **8** 5

[6] Pedersen T A, Glomsrud J A, Ruud E L, Simonsen A, Sandrib J and Eriksen B O H 2020 Towards simulation-based verification of autonomous navigation systems *Safety Science* **129** 104799

[7] Dosovitskiy A, Ros G, Codevilla F, Lopez A and Koltun V 2017 CARLA: An open urban driving simulator *Proceedings of the 1st Annual Conference on Robot Learning* (*Preprint* `arXiv/1711.03938`)

[8] DNV 2021 Open simulation platform , accessed 02.01.2021, URL: https://opensimulationplatform.com

[9] Sæther B 2019 *Development and Testing of Navigation and Motion Control Systems for milliAmpere* Master's thesis NTNU

[10] Hem A G, Leinebø D, Opheim H V, Vasstein K and Skarshaug T 2019 Digital tvilling , Project report from a group in the village Autoferge in Experts in Team at NTNU

[11] Vasstein K 2020 Real time simulation of EMR sensors for autonomous ferries , Specialization project NTNU

[12] Vasstein K, Brekke E F, Mester R and Eide E 2020 Autoferry gemini: a real-time simulation platform for electromagnetic radiation sensors on autonomous ships *IOP Conference Series: Materials Science and Engineering* vol 929 p 012032

[13] Manivasagam S, Wang S, Wong K, Zeng W, Sazanovich M, Tan S, Yang B, Ma W C and Urtasun R 2020 Lidarsim: Realistic lidar simulation by leveraging the

real world *Proceedings of Computer Vision and Pattern Recognition* (*Preprint* `arXiv/2006.09348`)
[14] Mueller F, Bernard F, Sotnychenko O, Mehta D, Sridhar S, Casas D and Theobalt C 2018 Ganerated hands for real-time 3d hand tracking from monocular rgb *Proceedings of Computer Vision and Pattern Recognition* pp 49–59
[15] Mallya A, Wang T, Sapra K and Liu M 2020 World-consistent video-to-video synthesis *Proceedings of the European Conference on Computer Vision* (*Preprint* `arXiv/2007.08509`)
[16] Brekke E F 2018 Fundamentals of sensor fusion: Target tracking, navigation and slam , Text book from the course TTK4250 at NTNU
[17] Skarshaug T 2020 *Developing and using a virtual platform to test cyber security for autonomous vehicles and vessels* Master's thesis NTNU
[18] Przybilla H J and Baeumker M 2020 RTK and PPK: GNSS-Technologies for direct georeferencing of UAV image flights *Conference of FIG Working Week* pp 1–17
[19] Ublox 2016 Achieving centimeter level performance with low cost antennas , White paper - R01
[20] Everett T 2020 RTKLIB optimized for single and dual frequency low cost gps receivers , accessed 10.09.2020, URL: https://github.com/rtklibexplorer/RTKLIB
[21] RTKLIB 2013 *Manual* , version 2.4.2, accessed 08.09.2020, URL: http://www.rtklib.com/prog/manual_2.4.2.pdf
[22] Sirnes M 2020 Joint localization and tracking: experiments and preliminary work on active sensors , Specialization project NTNU
[23] Kjønås I 2020 Detection of vessels in infrared images from a shore-mounted camera , Specialization project NTNU
[24] Farrell J A 2008 *Aided Navigation: GPS with High Rate Sensors* vol 1 (New York: McGraw-Hill) p 233
[25] Fossen T I and Perez T 2004 Marine systems simulator (MSS) , accessed 22.09.2020, URL: https://github.com/cybergalactic/MSS
[26] Reed N 2013 Quick and easy gpu random numbers in d3d11 , accessed 22.11.2020, URL: http://www.reedbeta.com/blog/quick-and-easy-gpu-random-numbers-in-d3d11/
[27] Trondheim-Kommune 2014 Digital 3D-bymodell for deler av Trondheim kommune , accessed 02.12.2020, URL: https://data.norge.no/datasets/fff3a365-cd82-448e-97a2-05aade8f6cf1
[28] Øystein Kaarstad Helgesen 2020 *Sensor Fusion for Detection and Tracking of Maritime Vessels* Master's thesis NTNU
[29] Abou Moustafa K T, De La Torre F and Ferrie F P 2010 Designing a metric for the difference between gaussian densities *Proceedings of Brain, Body and Machine* ed Angeles J and Boulet B pp 57–70
[30] Bochkovskiy A, Wang C Y and Liao H Y M 2020 Yolov4: Optimal speed and accuracy of object detection (*Preprint* `2004.10934`)
[31] Anonymous tensorflow-yolov4-tflite , accessed 10.08.2020, URL: https://github.com/hunglc007/tensorflow-yolov4-tflite
[32] Hussein B 2018 *The Road to Success* vol 1 (Bergen: Fagbokforlaget) pp 129–139

Appendices

Appendix A

Project Management

Due to the required expertise in implementing the architecture chart in Section 2.2.2, it was decided in order for the project to succeed in the given amount of time it needed to involve multiple researcher with both domain expertise and knowledge of the platforms in Section 2.1. This involved one PhD student with expertise in target tracking and ROS, one PhD student with expertise in OSP, and the last PhD student with expertise in ship design, all of which had a direct or indirect interest of the project outcome in their research. In addition 2 employees from both NTNU and Zeabuz would be assisting in the development of the framework, and two master students and a PhD student in recording an experiment.

In order to manage the project, work packages describing each contributors participation, together with a chart showing their dependency to each other was created. An introduction and reference to the work packages will be covered in the first section, followed by a description of AON-network used as a tool for making the project plan in the last section.

A.1 Work package

A work package is the smallest unit of work a project can be broken down to, and are one of the *project managers* (in this case the author) primary tool in organising the *workers*. They contain information necessary for finishing and defining completion of a job. There is however no complete standard of their content since jobs vary between different projects. With this said, there are some recurring topics which will be covered in this section.

Multiple workers can be assigned to the same work package, but there must always be one package owner representing the workers in a work package. The *package owner* has the primary responsibility of reporting the work progress to the project manager and finishing what is in the work package *description*. In order to know when a work package is finished, each package must have a statement that can be tested, known as a *deliverable*. The deliverable functions both as a safeguard for when it is proper to start the next work package, in addition to giving the project manager a view of the project progression, giving hints if adjustments to the plan must be made to reach the project goals in time.

To calculate the total project time, each work package is given an estimated time

from discussions with the assigned workers for the package. This is then recalculated into dates based on the projects point of view: *early start and finish*, *late start and finish*. These timespans describes respectfully the best and worst case scenarios for when a work package can start and finish without beginning prematurely or delaying the project.

At last, additional data for clarifying the plan for the manager and workers can be added, such as the *owner organisation* and *receivables*. The latter is usually added such that the workers do not need to look up the deliverables of previous work packages.

A.2 AON-Network

To determine the timespans of *early start and finish*, and *late start and finish* for the workpackages, the interconnection between work packages must be described, something which can be done by using *AON-networks*. This is done by representing each work package with a box containing a short title of the package followed by an index for reference, and surrounding numbers representing the early start and finish from left to right at the top, and likewise for the late start and finish at the bottom. These numbers indicate the week number from when the project started.

The common practise is to first chain the work packages together with arrows to define both the workflow and dependencies. From this, one can decide based on the work package descriptions if the work can start before the previous package or must start after its completion. These cases are indicated by either placing the arrow heads at the right portion of the work package for the former, or placing them at the left portion for the latter. In addition to this, numbers can be placed on the arrows to indicate delays in week, if its believed that slack is needed to loosen up uncertainties from certain work packages.

The leaf nodes at the left side of the AON-network must now initialize their early start, depending on the workers time schedule or as early as possible. Early finish is found by adding the early start with the work package's estimated time described in the previous section. The early start of the next work package inherits the latest earliest finish from the preceding work packages in the chain, added by potential delays. At this point, the early start and finish for each work package can be filled in by repeating this procedure until the last work package has it earliest finish.

The last work package's latest finish can now be set equal to it's earliest finish, or adding project slack to accommodate for project uncertainties. For this project no such slack where given due to the time contraints. To find the remaining early start and finish, the previous procedure is reversed, decrementing instead of incrementing numbers and working from right to left. In addition the late finish inherits the earliest latest start of the succeeding work packages, subtracted by potential delays. For more information of these procedures see [32].

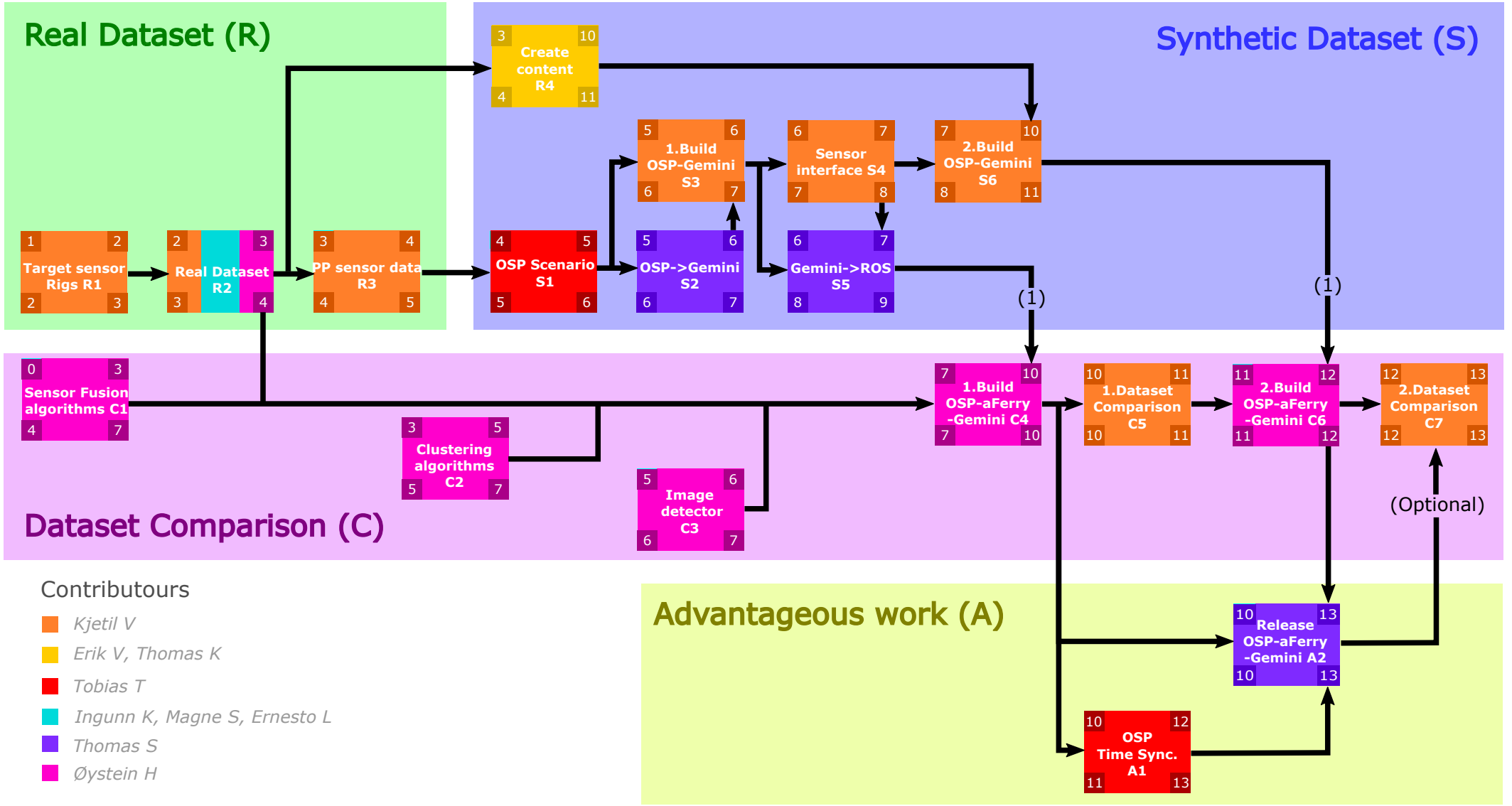
A.3 Project plan

Before Project start, a plan incorporating work packages and a AON-network was given to the project contributors. The plan was based on the architecture defined in Section 2.2.2, defining work packages in cooperation with the contributors. This resulted in 19 work packages described in Appendix B. From these packages, the creation of a AON-network with respect to the contributors time schedules gave a total project time of 13 weeks starting at 01.09.2020 and ending 30.11.2020. During the project timespan, the plan was open to changes through *revisions*.

The final revision of the AON network on the next page shows all the work packages involved, using colors to acknowledge each participants contribution to the project and to separate the authors contribution from the rest. Due to the overall size of the project, chapters in the thesis will be focusing on the work packages of the author (colored brown), but give short summaries of the other contributors work when possible. Most of the work packages are also categorized and indexed with the thesis chapters if a deeper understanding is needed when reading the work packages in Appendix B.

AON-network workpackages

Project timeline: 01.09.2020 - 30.11.2020
Revision: 08.10.2020



Appendix B

Work package descriptions

R 1 Target sensor Rigs

Package Owner	Kjetil V
Owner Org.	NTNU
Participants	Kjetil V
Early start & finish	07.09.2020 - 13.09.2020
Late start & finsih	14.09.2020 - 20.09.2020
Description: Make or find suitable sensor rigs for the target boats in Figure 3.4	
Deliverables: 2 sensor rigs containing sensors for measuring: • *Heading* • *Position*	

R 2 Real Dataset

Package Owner	Kjetil V
Owner Org.	NTNU
Participants	Kjetil V, Øystein H, Ingunn K, Magne S, Ernesto L
Early start & finish	14.09.2020 - 20.09.2020
Late start & finsih	21.09.2020 - 27.09.2020

Description: Gather or create a real-life dataset consisting of a location, ownship and targets as in Figure 3.4:

Location

- *Available terrain data*

Ownship

- *EMR sensors: Radar, optical camera*
- *Ground truth sensors for positioning and orientation*
- *Sensor specifications, positions and orientations*
- *Ground truth sensors for positioning and orientation*
- *Available pictures and dimensions of vessel*

Targets

- *Ground truth sensors for positioning and orientation*
- *Sensor specifications, positions and orientations*
- *Available pictures and dimensions of vessel*

and optionally:

Location

- *Specified time of day*
- *Near Trondheim / Trondheimsfjorden*
- *Weather specified or shown from picture*

Ownship

- *Additional EMR sensors: lidar and IR camera*
- *Available 3D model*

Targets

- *Available 3D model*

Deliverables:

- *Deliver raw heading and positional data for both the ownship and targets in whatever format the sensors use*
- *Record the ownship's EMR sensor data by creating ROS topics in a ROS-bag*

R 3 Post process sensor data

Package Owner	Kjetil V
Owner Org.	NTNU
Participants	Kjetil V
Early start & finish	21.09.2020 - 27.09.2020
Late start & finish	28.09.2020 - 04.10.2020
Receivables: Raw GNSS and IMU data	
Description: Process sensor data	
Deliverables: A CSV files containing heading and positional information for both the ownship and the target ships	

R 4 Create content

Package Owner	Erik V
Owner Org.	NTNU
Participants	Erik V, Thomas K
Early start & finish	21.09.2020 - 08.11.2020
Late start & finish	28.09.2020 - 15.11.2020
Description: Gather information to create 3D models of the target boats in Figure 3.4	
Deliverables: Create two 3D models with textures of the target boats, and import them inn to Unity	

S 1 Open simulation platform scenario

Package Owner	Tobias T
Owner Org.	NTNU
Participants	Tobias T
Early start & finish	28.09.2020 - 04.10.2020
Late start & finish	05.10.2020 - 11.10.2020
Receivables: CSV files containing heading and position of the target ships and ownship, such as shown in Figure 4.17	
Description: Create a scenario in OSP from the received CSV files, and send them to Gemini	
Deliverables: Deliver OSP FMU-modules that reads from the CSV files line by line and prepare it for delivery via Gemini's API interface. The procedure are decribed in Figure 4.17	

S 2 Open simulation platform to Gemini

Package Owner	Thomas S
Owner Org.	Zeabus
Participants	Thomas S
Early start & finish	05.10.2020 - 11.10.2020
Late start & finish	12.10.2020 - 18.10.2020
Receivables: OSP FMU's and CSV files containing heading and positions	
Description: Receive OSP-messages on the format described on the top of Figure 4.17. Deliver the OSP-messages to Gemini via Gemini's API	
Deliverables: Update Gemini's gRPC API to handle the protobuf interface defined in OSP's FMU's. Print the messages in Unity's debug log.	

S 3 1.Build of the OSP-Gemini platform

Package Owner	Kjetil V
Owner Org.	NTNU
Participants	Kjetil V
Early start & finish	05.10.2020 - 11.10.2020
Late start & finish	12.10.2020 - 18.10.2020
Receivables: An updated Gemini version ready for communication with OSP's FMU's	
Description: Recreate the scenario from Figure 3.4 in Gemini	
Deliverables: Spawn and create a state driven control system of the 3 vessels using the incoming OSP messages to Gemini. Use the Trondheim city model and three 3D dummy boats, when running the scenario from OSP to verify the implementation.	

S 4 Sensor interface

Package Owner	Kjetil V
Owner Org.	NTNU
Participants	Kjetil V
Early start & finish	12.10.2020 - 18.10.2020
Late start & finish	19.10.2020 - 25.10.2020
Receivables: OSP-Gemini build	
Description: Deliver sensor formats for lidar, radar and VL cameras, and prepare their data for delivery to ROS	
Deliverables: Structure the sensor information into native arrays, formatting the arrays with respect to angles for lidar and radar.	

S 5 Gemini to ROS

Package Owner	Thomas S
Owner Org.	Zeabus
Participants	Thomas S
Early start & finish	12.10.2020 - 18.10.2020
Late start & finish	26.10.2020 - 01.11.2020
Receivables: • *OSP-Gemini build* • *Rosbag with real sensor data for lidar, radar and VL cameras* • *Formated sensor data for lidar, radar and VL cameras*	
Description: Send sensor data from Gemini to ROS	
Deliverables: Deliver a Gemini API system towards ROS, consisting of: • *A gRPC server at ROS that publishes timestamped sensor data on the same ROS topics as for the real dataset* • *A gRPC client in Gemini that collects sensor data according to the sensor format* Verify the implementation by listening on one of the created ROS topics for sensor data	

S 6 2.Build OSP-Gemini

Package Owner	Kjetil V
Owner Org.	NTNU
Participants	Kjetil V
Early start & finish	19.10.2020 - 08.11.2020
Late start & finish	26.10.2020 - 15.11.2020
Description: Improve the sensor models from Workpackage R3	
Receivables: • *Updated Gemini version ready for communication with OSP* • *Sensor formats for lidar, radar and VL cameras*	
Deliverables: • *Improved sensor models that reduces the beam shape error [12] for lidar and radar* • *Optional: Improve the radar model to take into account the radar cross section area* • *Replace dummy vessels with 3D models from Workpackage R4*	

C 1 Sensor fusion algorithms

Package Owner	Øystein H
Owner Org.	NTNU
Participants	Øystein H
Early start & finish	01.09.2020 - 20.09.2020
Late start & finish	28.09.2020 - 18.10.2020
Description: Create and tune several tracking algorithms, with suitable metrics for measuring performance	
Deliverables: • *JPDA-algorithm implemented in matlab/python* • *Tracker outputs: estimates, covariance matrices* Tune and verify the Implementation by using an existing dataset or by using simulated data.	

C 2 lidar clustering

Package Owner	Øystein H
Owner Org.	NTNU
Participants	Øystein H
Early start & finish	21.09.2020 - 04.10.2020
Late start & finish	05.10.2020 - 18.10.2020
Description: Setup lidar clustering algorithm from aFerry code.	
Deliverables: • *A lidar clustering algorithm from lidar point cloud* Validate by using a separate dataset or from simulated data	

C 3 VL Detector

Package Owner	Øystein H
Owner Org.	NTNU
Participants	Øystein H
Early start & finish	05.10.2020 - 11.10.2020
Late start & finish	12.10.2020 - 18.10.2020
Description: Implement an image detection algorithm for boats.	
Deliverables: Implement an existing object detection algorithm such as Yolo with pretrained weights, and validate the results by testing object detection on a separate dataset containing boats.	

C 4 1.Build of OSP-aFerry-Gemini

Package Owner	Øystein H
Owner Org.	NTNU
Participants	Øystein H, Audun G H
Early start & finish	19.10.2020 - 08.11.2020
Late start & finish	19.10.2020 - 08.11.2020
Description: Setup different autonomy systems in ROS to run on both synthetic and real data.	
Receivables: • *RBG image detector trained on boats* • *Clustering algorithms* • *JIPDA tracking algorithms with suitable performance metrics* • *Synthetic sensor data on specified ROS topics from Workpackage S5* • *Real sensor data from specified ROS topics from Workpackage R2*	
Deliverables: Implement the autonomy systems from C1 in ROS with the clustering algorithms in C2, and VL detector in C3, using repositories in aFerry as aid. This consists of: • *A lidar fusion* • *A lidar-camera fusion* • *Filter outputs as specified in C1* Save the recorded data for the autonomy systems above for both the synthetic (S5) and real datasets (R2).	

C 5 1.Dataset comparison

Package Owner	Kjetil V
Owner Org.	NTNU
Participants	Kjetil V
Early start & finish	09.11.2020 - 15.11.2020
Late start & finish	09.11.2020 - 15.11.2020
Description: Evaluate the filter outputs by creating performance metrics for the different datasets, and come up with a list of implementation improvements	
Receivables: Filter outputs for the synthetic and real datasets	
Deliverables: List of proposed improvements regarding both the performance metrics and sensor models	

C 6 2.Build OSP-aFerry-Gemini

Package Owner	Øystein H
Owner Org.	NTNU
Participants	Øystein H
Early start & finish	16.11.2020 - 22.11.2020
Late start & finish	16.11.2020 - 22.11.2020
Description: Implement suggested improvements	
Receivables: • *List of improvements* • *Final OSP-Gemini build*	
Deliverables: • *As far as possible, try to implement the proposed improvements if any* • *Save a new set of recorded data metrics for the autonomy systems in C4 above for both the synthetic (S5) and real datasets (R2)*	

C 7 2.Dataset comparison

Package Owner	Kjetil V
Owner Org.	NTNU
Participants	Kjetil V
Early start & finish	23.11.2020 - 29.11.2020
Late start & finish	23.11.2020 - 29.11.2020
Description: Final analysis of results	
Receivables: Final filter recordings for the synthetic and real datasets for the improved sensor models	
Deliverables: Check if there have been any change in the performance metrics since C5. Give a final conclusion of the results	

A 1 OSP time synchronization between ROS and Gemini

Package Owner	Tobias R T
Owner Org.	NTNU
Participants	Tobias R T
Early start & finish	09.11.2020 - 22.11.2020
Late start & finish	16.11.2020 - 29.11.2020
Description: Create a time synchronization system that guarantees synchronization between Unity and ROS	
Receivables: Build of OSP-aFerry-Gemini	
Deliverables: FMU's on OSP's side, as well as integration with Milliampere's low level control system used in communication with aFerry	

A 2 Release OSP-aFerry-Gemini

Package Owner	Thomas S
Owner Org.	Zeabus
Participants	Thomas S
Early start & finish	09.11.2020 - 29.11.2020
Late start & finish	09.11.2020 - 29.11.2020
Description: Create a final release of OSP-aFerry-Gemini	
Receivables: Last OSP-aFerry-Gemini build, in addition to the time synchronization system between ROS and Gemini.	
Deliverables: Script that takes a scenario file (CSV) and a Rosbag in as argument, and runs the complete OSP-aFerry-Gemini build. Verify implementation by testing the released version on existing Rosbags and scenario files used in C6	