Master's thesis

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

Fredrik Opeide

Deep Learning-Based Multi-Camera Situational Awareness and Global Localization for Autonomous Ships

Master's thesis in Cybernetics and Robotics
Supervisor: Edmund Førland Brekke
June 2019

NTNU
Norwegian University of Science and Technology

Fredrik Opeide

Deep Learning-Based Multi-Camera Situational Awareness and Global Localization for Autonomous Ships

Master's thesis in Cybernetics and Robotics
Supervisor: Edmund Førland Brekke
June 2019

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

NTNU
Norwegian University of
Science and Technology

Preface

This is the report for the compulsory final project at the five year MSc program Cybernetics and Robotics at NTNU.
The project has been done in cooperation with Kongsberg Seatex, where I have been provided with with a work place, powerful computers and peripherals, as well as a huge data set with a programming interface.
The task given to me from Seatex was an open problem, to create a camera-based localization system. Otherwise this has been a very self-driven project, as I have been free to select the scope, methods and goals for this project myself.
Many choices in this master thesis are based on results from my project thesis from the 2018 winter semester, titled "Camera-Based Position Estimation for Autonomous Ships in Elevation Mapped Areas".
I would like to express my sincere thanks to my contacts and advisors at Seatex, Torbjørn Barheim, Arild Nøkland, Ståle Smedseng and Henrik Foss, for motivating me and reviewing my progress for both my master- and project-thesis.
I also extend my gratitude to my NTNU advisor Edmund Førland Brekke at ITK for his writing tips and feedback on drafts of this report.

Fredrik Opeide
June 2019

Abstract

I present the design, implementation and testing of a non-drifting 6DoF pose estimation system for ocean vessels based on semantic segmentation of camera images using a deep neural network. Only ship mounted cameras and publicly available geographical height maps are used to accurately estimate the ship's global pose.

The segmentation network is a PSPNet adapted to and trained on a custom dataset from all along the Norwegian coast. Is is trained to label each pixel as either sky, land or ocean. The trained net generalizes well and achieves 99.5% pixel accuracy, but is limited by the imperfectly generated dataset.

The localization uses an arbitrary number of ship-mounted cameras simultaneously and works by comparing the segmented camera images to a virtual reality expected view for each camera, taking into account the camera poses, intrinsic parameters and lens distortion effects. Given an initial ship pose estimate, iterative optimization (ICP) is used to find a new ship pose that better matches the virtual model to the segmented images. Localization with multiple cameras is demonstrated to be much more robust and accurate than single-camera localization. Errors in the camera pose and calibration, as well as inaccurate segmentation induces localization inaccuracy. Under the right conditions sub-meter accuracy can be achieved several km from shore.

The entire system is implemented in python, and runs at ca. 0.2hz for 4 camera images with resolution 1280x960, which is not fast, but promising for a proof of concept prototype.

The localization system using four cameras is tested on test-data sequences from ca. 70 unique regions, and only fails for 3. The 3 failures were enabled by the starboard camera periodically not working, but the fails were ultimately caused by bad image segmentation again caused by some bad training samples. Videos for each localization test sequence are available here
https://drive.google.com/drive/folders/1TPLzuMLonLWutZzT2lJlendu7V0de2xc.

Sammendrag

Jeg presenterer design, implementasjon og testing av et ikke-driftende lokaliseringssystem i seks frihetsgrader for skip, basert på semantisk bildesegmentering ved bruk av et dypt neuralt nettverk. Kun skipsmonterte kameraer og offentlig tilgjengelig høydekartdata brukes til å presist estimere skipets globale posisjon og orientering.
Segmenteringsnettverket er et PSPNet som er tilpasset og trent på et nytt generert datasett langs den norske kysten. Nettet generaliserer godt, og oppnår 99.5% piksel-nøyaktighet, men blir begrenset av imperfeksjoner i det genererte datasettet.
lokaliseringssystemet kan benytte et vilkårlig antall kameraer samtidig, og fungerer ved å sammenligne segmenterte kamerabilder mot virtuelle kamerabilder fra en 3D-modell av terrenget, som tar høyde for hvordan kameraene er montert, deres interne parametre og linseforvrengning. Gitt en initiell posisjon og attitude for et skip, benyttes iterativ optimalisering for å finne en ny posisjon og attitude som bedre matcher den virituelle modellen mot de segmenterte kamerabildene. Lokalisering som bruker flere kameraer vises å være mye mer robust og presist enn lokalisering med ett kamera. Feil i kameramonteringsvinkler og kamerakalibrering, samt unøyaktig bildesegmentering medfører unøyaktigheter i lokaliseringen. Under riktige forhold kan nøyaktigheten bli på under én meter flere km fra kysten.
Hele systemet er implementert i python, og kjører i omtrent 0.2Hz for 4 kameraer med bildeoppløsning 1280x960, som ikke er raskt, men lovende for en prototype ment som konseptbevis.
Lokaliseringssystemet med fire kameraer er testet på testdatasekvenser fra ca. 70 unike regioner, og svikter for kun 3. De 3 sviktene ble lagt til rette for ved at styrbordkameraet periodevis var ute av drift, men ble i bunn og grunn forårsaket av dårlig bildesegmentering, igjen grunnet noen dårlige treningsdata. Videoer fra hver lokaliseringssekvens er tilgjengelig her
https://drive.google.com/drive/folders/1TPLzuMLonLWutZzT2lJlendu7V0de2xc.

Contents

Chapter 1

Introduction

1.1 Motivation

The goals and choices are in large part based on the results from the precursor to this master thesis, my project thesis, where canny edge detection in a single camera was used to estimate the orientation (not full pose) of the ship. In the project thesis I discovered and described the inherent weaknesses of using just one camera for localization. The main weakness of using just one camera is that there is not always enough information in a single image to determine a unique solution for the localization. Furthermore the reliance on a single camera is extremely fault-intolerant. Therefore the natural next step is to use multiple cameras and to also estimate the full 6DoF pose of the ship. Estimating the full pose, not just the orientation, completely removes the need for GNSS, other than perhaps for initialization.

I was also dissatisfied with the performance of canny, which often fails to find the correct edges, and produces a lot of clutter with no way of telling which edges are what. I decided to explore the use of neural networks as they have shown remarkable performance in image understanding, and has overcome the shortcoming of traditional edge detection. Since the advent of powerful GPUs performance as exploded, and research

interest is at an all time high. Many different state of the art neural net architectures are openly available online, and there exists multiple deep learning frame works to choose from. There is an inherently big advantage to using semantic segmentation instead of trying to develop more robust tracking with a better edge detector. This is the fact that the semantic segmentation in itself provides situational awareness, giving information about surrounding areas even if the ship has no idea where exactly it is. Knowing what is what in an image can aid in making a more fail-safe system where the system can navigate to avoid obstacles that it sees despite localization failure. It can also be used to detect reefs, small vessels or other objects the navigation system might otherwise not have known about from map data.

Having a system that resists GPS-failure has become a particularly important research area. This is in part because some areas simply have poor GPS coverage due to natural or man-made structures. However the most important reason is that GPS signals are susceptible to interference and jamming, either unintentionally or maliciously. Signals can even be imitated or spoofed without the recipient detecting this, which is particularly dangerous. Trailer drivers have been known to use GPS jammers to avoid being monitored by their employers, collaterally blocking GPS reception for all nearby devices. In 2018 the GPS reception in the north of Norway was completely blocked on three separate occasions, each disturbance lasting two to three weeks. The Norwegian intelligence service, Etterretningstjenesten, states that the cause was the Russian military testing new equipment for electronic warfare.

Recent law changes open up for more autonomy in the maritime sector, and a wider array of sensors are now allowed to be used in a navigation system, including cameras. Using as many sensors and data sources as possible is important for robust navigation, and so camera-based navigation naturally becomes an important research topic. Furthermore cameras are very cheap sensors compared to LIDAR and radar systems, and are already used in virtually all driving assistance systems in the automotive industry.

Camera-based model tracking, where digital models are detected in an image and local-

ized in relation to a camera is a research area that has been studied for decades. Since height map data is openly available in high detail for the entirety of the Norwegian coast, it stands to reason that it could form the basis for a navigational system, where a digital model of the terrain itself is used as a sort of way-point.

No filtering of the pose is performed. This is both due to time-limitations of the project, and can also be justified by that it better exposes weaknesses in the system. All the systems designed and implemented in this project are very modular, and so the system can be extended later, to compensate for the discovered weaknesses.

1.2 Related Work

1.2.1 Visual Localization

Visual localization is a very popular research topic, and methods can generally be divided into three categories; visual odometry, SLAM and model-based localization. Structure from motion is also related to these methods, but here the focus is not on camera pose estimation, rather just dense reconstruction of the environment [49].
Visual odometry concerns itself with using the camera images to estimate the camera movement between the frames, and can be based on both directly comparing pixels [9] or tracking detected feature-descriptors [29], such as licence-unrestricted ORB-features [35]. A key problem of using monocular cameras that scale is unobservable [36]. This can be remedied by fusing data from a sensor which does in fact observe the scale, such as an IMU [43], or by using depth cameras [20] or a stereo camera setup [46]. The depth measurements can even be imitated by in stead using a deep neural network to predict the depth of a monocular camera image [53]. Feature based visual inertial odometry has been shown to work well for sea surface vehicles in natural environments with large scene depth variations and lighting variations [38].

The visual odometry concept is extended in **SLAM** (Simultaneous localization and mapping), where the system gradually creates a map of it's environment, while simul-

taneously estimating the camera's location within the map [8]. This typically consist of a front-end where visual odometry is used to estimate movement between image frames, and a back-end that performs bundle-adjustment, which uses measurements over time to refine both the camera path in the map and the geometry of the map [14]. By recognizing already discovered areas and using the old map points in current localization the system can remain drift-free within the generated map [1].

Both direct[10] and feature-based [27] SLAM is viable, and methods are often further classified as dense [50] or sparse [12], depending on how many points are used in the created map. Most SLAM methods are developed for single-camera systems, but existing SLAM systems can later be modified to use a multi-camera rig [41].

Stumberg et al. uses a neural network to make a direct slam system more resistant to lighting changes, feeding deep visual descriptors for each pixel into the SLAM-system. Their system is more robust against bad initialization and weather changes than both state-of-the-at direct and indirect methods, as of April 2019 [44]. Deep learning can also be used for semantic labelling of image points, which can be used to build a semantic map of the explored region [37].

Unless tied to the real world using prior information, a model, the localization in SLAM is only in relation to the generated map. Using a prior model which the camera is localized in relation to is called **model-based localization**, and when the global localization of the model itself is known, by extension global localization of the camera is possible. In the case of a visual SLAM system the geometry of the generated map can be aligned with an existing chart of a known region [42], or a pre-made feature map with true locations can be used with loop closure [26][13]. Global camera-based localization does not need use a SLAM system as basis, in fact there is a myriad of different methods for camera localization using prior knowledge.

visual place recognition, also known as topological or topometric localization, is one such method, where a query-image is matched with a database of images from known locations, using some lower dimensional representation of the images [5]. The lower dimensional representation can be feature based, such as [4] which uses a

pre-build large-scale database of mountain contour sections to localize a query image with (sometimes manually) extracted contours, and achieves a 10 second query time over the entirety of Switzerland. A lower dimensional image representation can also be generated by a neural network, trained to give similar descriptors for images of the same area [25] [30]. Topological localization over a large area is inherently slow, but can be used to initialize a more accurate real-time localization system [3].

A marker-based tracking system has knowledge of specific visual descriptors places at known locations on the model, which is an old technique [21] still used in modern systems such as the HTC Vive[28]. Marker-less tracking is the alternative for less controlled environments. Some model based localization systems are based on iteratively aligning contours in a virtual reality model, for example a 2.5D model of an urban environment, to detected edges in the camera image [52][33][23], a process which can be fused with IMU data for robustness [34].

An image can also be registered to a virtual model using other measures than edges and edge-distances, such as aligning whole image regions, not just edges. One such system is PWP3D [32], which uses a statistical model for background segmentation, which is used to match the foreground region with a model silhouette. PWP3D was the first real-time capable system using region-based optimization, achieving this by utilizing GPU acceleration.

Classical edge based and region based methods rely on manual tuning of feature extraction parameters, and do not produce results that are in consistent with what a human recognizes in the image. and so Arth et al. takes advantage of the recent advances in deep neural network to do semantic segmentation of buildings in a monocular camera to generate line segments, which is then iteratively aligned against a 3D model of an urban area [17] to estimate the camera pose. Their system was much more robust than tradition edge detection based camera localization systems. Another similar system by the same authors uses sampled poses around an initial pose estimate for the camera, compares the segmented image to the model viewed from the different poses and uses the best match as the new pose estimate [18]. The same authors propose yet another approach based on semantic segmentation, namely a neural net trained to directly predict a pose change that better aligns a semantically segmented camera

image and a model rendering at the initial estimated pose, taking these two images as input [2]. Wang et al. use also perform updates to a pose estimation by feeding semantically segmented camera images and model renderings into a pose estimating neural network, and achieve additional robustness by fusing it with inertial sensors [45].

1.2.2 Deep semantic segmentation

The original Convolutional Neural Network (CNN) was invented in 1980 by japanese computer scientist Kunihiko Fukushima [11], and forms the basis for virtually all modern image processing neural networks [15]. CNNs stand in contrast to fully connected networks in that they use a set of smaller filters applied over the entire image to extract features, requiring fewer parameters. Lower layers typically represent features such as edges or colored dots, while deeper layer represent more complex features. These extracted features can be used for tasks such as classifying images [22]. ery deep networks have been unstable to train until Resnet introduced the concept of skip-connections that let residual values from past layers skip past layers and just be added to the layer output in stead [16]. This reduces the problem of vanishing gradients, which is a problem with very deep nets.

Deep semantic segmentation is concerned with using deep neural networks to predict the category label of each pixel in an imput image, ie. classify each pixel as one of several pre-determined objects. For example this can entail marking which pixel are part of any humans in an image.For some time fully convolutional neural networks were the state-of-the-art for semantic segmentation accuracy [24], as the very local filters of CNN struggled to grasp the context necessary to correctly label complex scenes [56]. A solution to this was proposed by Zhao et a. with the Pyramid Scene Parsing Network [56]. The PSPNet is build on top of a resnet to extract a feature map, and then uses a pyramid parsing module get different sub-region representations (big and small regions), which are then upsampled and concatenation to form the final feature representation, which carries both local and global context information. This final feature representation is then fed to a CNN which predict pixel labels. The PSPNet

achieved state of the art and came first in ImageNet 2016 scene parsing challenge, PASCAL VOC 2012 benchmark and Cityscapes benchmark. Other novel networks have been developed with PSPNet as a basis, among them ICNet, which is a intended for real-time, and as such sacrifices some accuracy for a huge speed boost, becoming the state-of-the-art among networks of similar processing speed [55].

1.3 Contributions

The main contribution of this project is further developing existing concepts of segmentation-image-to-model tracking to use multiple cameras simultaneously by expressing the pose estimation for each camera in the ship-frame and optimizing with all cameras jointly. The system is such that it only needs publicly available height maps, not relying on a pre-build feature-map or a hand-crafted map. It is also demonstrated that semantic segmentation based localization can work well in natural environments, not just cities, at least when the weather is stable. The model generation and rendering system used in both the generation of segmentation data and in the tracking system is very modular. The system can easily be extended later to include more semantic classes, again by using publicly available data to for example separate buildings from the terrain. The accuracy of the system under perfect conditions is tested and discussed. The system is tested on multiple data sequences from areas it has not been trained on, and the three cases where the system fails are inspected and discussed.

The interface to retrieve images and pose data from the Polarlys dataset, and the Polarlys dataset itself is developed and maintained by Seatex employees, see ch. 3. However, every other system-component described in project report is implemented from scratch in python, by me. The project uses many python packages; Numpy, Scipy, OpenGL, PIL, Pyrr, Keras, OpenCV and Matplotlib have been essential for this project. There are a multitude of practical components to this project, all of which have been completed solely by myself, and so here is a list of them.

- Created a 3D model generator, using publicly available heigthmap data from Geonorge which is first corrected then used to create triangle meshes.
- Created a rendering engine using OpenGL to render the triangle meshes, simulating a camera on a ship, using the camera intrinsic parameters. There is also a simple LOD scheme to show closer terrain in higher detail.
- Created a tool to quantify and correct erroneous camera mounting angles in the Polarlys dataset. Errors related to the camera calibration were also discovered

and documented. This angle correction tool is essentially the system that was developed in the project thesis, just with a much better model generation and rendering system.

- Used the model generation and rendering to create a semantic segmentation dataset of ca. 4000 images, labelling each pixel as sky, land or ocean.
- Created a training pipeline and used this to train a PSPNet on the custom dataset, with great performance and well documented results. Transfer learning was used with weights from a net trained by the original author on the ADE20K dataset. Errors in particular scenarios were discovered to be due to imperfections in the training data.
- Designed, implemented and tested a localization system using multiple cameras simultaneously, based on comparing semantic segmentation of images to a virtual terrain model, taking into account the intrinsic and extrinsic parameters of each camera.

1.4 System overview and structure of report

A simplified illustration of how the localization system's main components interact with each other is shown in figure 1.1. The figure also contains links to each component's main section in the report.

Chapter 3 introduces the dataset and some of it's shortcomings such as poor camera calibration, and how this is dealt with.
Chapter 4 describes the model generation process, and how the generated 3D model is rendered to create a virtual reality equivalent for a real camera.
Chapter 5 details how the rendering system from chapter 4 is used to create a tool to detect and correct inaccurate camera mounting angles in the data set.
chapter 6 describes the process of creating a data set for semantic segmentation. The training process and performance of the trained network is also presented here.
Chapter 7 covers the implementation details of a localization system based on aligning chapter 6's segmented camera images and chapter 4's rendered model images to estimate the pose of the ship.
Chapter 8 details how the localization system is tested on data sequences it has not been trained on, and investigates the best case accuracy as well as failure causes.
Chapter 9 sums up and discusses the most important results, and considers how the system can be developed further.

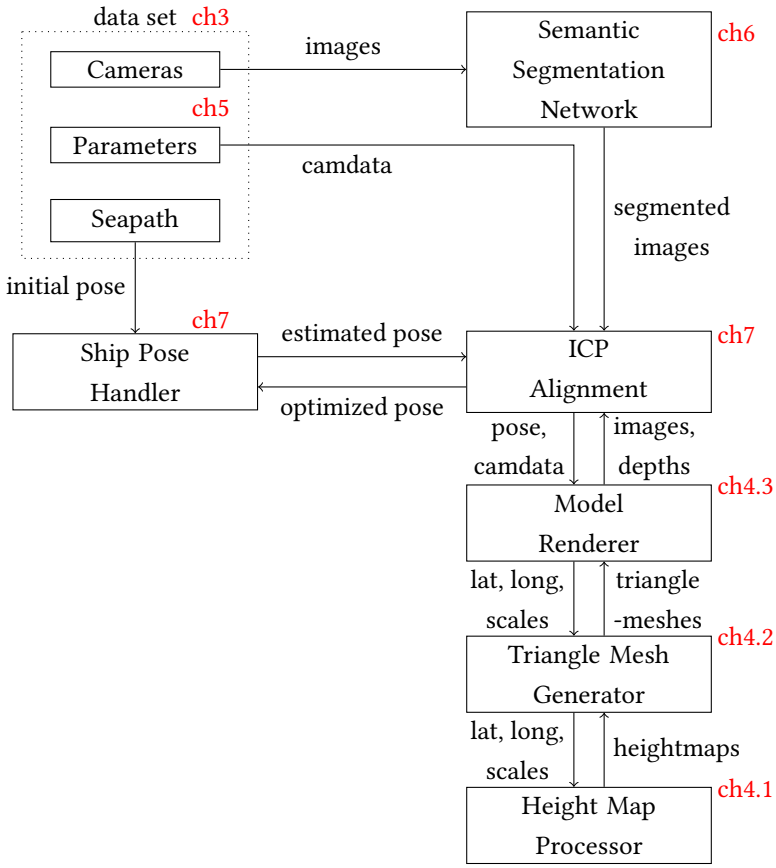


Figure 1.1: System overview with chapter shortcuts

Chapter 2

Theoretical Background

2.1 Kinematics

2.1.1 Rigid transformation matrix

A rigid transformation matrix is a matrix that can be used to apply a rotation and a translation to a point. This can be used to move a point within a reference frame, and it can also be used to change which reference frame a point is defined in reference to. This frame-change is achieved by letting the rigid transformation matrix represent the rotation and translation between the two frames. An example is shown i eq. 2.2. To multiply a 3D vector with a rigid transformation matrix the vector must be converted to homogenous coordinates, which is done by appending a 1, adding an extra dimension to the vector, denoted using over the vector. Converting back from homogenous coordinates is done by dividing all elements by the value of the extra element, and then removing the extra element completely, reducing the dimensionality, this is denoted as a function $\pi()$. Homogenous versions can be created from vectors of any dimensionality. Rigid transformation matrices in 3D space are called the special euclidean group, which is denoted SE(3).

$$T = \begin{bmatrix} \mathbf{R}_{3x3} & \mathbf{T}_{3x1} \\ \mathbf{0}_{1x3} & 0 \end{bmatrix} \tag{2.1}$$

$$X_b = \pi(T_a^b \tilde{X_a}) \tag{2.2}$$

2.1.2 Euler angle parameterization

According to Euler's rotation theorem [47], 3D rotations can be described using three parameters, three angeles. By comparison, rotation matrices have nine parameters, and is overdetermined. Euler angles can follow different convensions, but a common one is ZYX, where the angles are referred to as roll, pitch and yaw. Here the the three angles describe, in sequence, a rotation about the z-axis, yaw, a rotation about the new frame's y-axis, pitch, and a rotation bout the last frames x-axis, roll [48].

2.1.3 Twist Coordinate Parameterization

A tranformation matrix can be parameterized using lie alegbra. This way the translation matrix is reduced to a 6 parameter representation, $\hat{\xi} \in \mathfrak{se}(3)$, which can be mapped to a rigid body transform via $exp(\hat{\xi}) \in \mathbb{SE}(3)$.

$$\xi = \begin{bmatrix} \mathbf{w} \\ \mathbf{v} \end{bmatrix} = [\omega_1, \omega_2, \omega_3, v_1, v_2, v_3]^T \in \mathbb{R}^6 \tag{2.3}$$

$$\hat{\xi} = \begin{bmatrix} \hat{\mathbf{w}} & \mathbf{v} \\ \mathbf{0} & 0 \end{bmatrix} = \begin{bmatrix} 0 & -\omega_3 & \omega_2 & v_1 \\ \omega_3 & 0 & -\omega_1 & v_2 \\ -\omega_2 & \omega_1 & 0 & v_3 \\ 0 & 0 & 0 & 0 \end{bmatrix} \in \mathfrak{se}(3) \tag{2.4}$$

$$T = exp(\hat{\xi}) \in \mathbb{SE}(3) \tag{2.5}$$

$$exp(\hat{\xi}) \approx \mathbb{I}_{4x4} + \hat{\xi} \tag{2.6}$$

$$exp(\hat{\xi}) = \begin{bmatrix} rodriguez(\mathbf{w}) & \mathbf{v} \\ \mathbf{0} & 0 \end{bmatrix} \tag{2.7}$$

2.2 Cameras

2.2.1 Pin-hole Camera Model

The pinhole camera model is a model of how how the world is projected into the pixels of a camera image. The model consists of a closed chamber with a small hole (pinhole) that lets light in and projects in onto a plane [31]. The definition of the camera frame, taken from the OpenCV documentation, is seen in figure 2.1. This projection is mathematically represented with a camera calibration matrix, also known as projection matrix, K. If the real depth of a pixel is known it can be back projected to 3D using the inverse of the projection matrix.

$$proj(Z) = \pi(KZ) \tag{2.8}$$

$$\pi(Z) = \begin{bmatrix} \frac{z_1}{z_3} \\ \frac{z_2}{z_3} \end{bmatrix} \tag{2.9}$$

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

$$K^{-1} = \begin{bmatrix} \frac{1}{f_x} & 0 & -\frac{c_x}{f_x} \\ 0 & \frac{1}{f_y} & -\frac{c_y}{f_y} \\ 0 & 0 & 1 \end{bmatrix} \tag{2.11}$$

Back-projection of a 2D point X with depth d uses the inverse of the camera matrix K.

$$back(X, d) = dK^{-1} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = \begin{bmatrix} D(X)\frac{x_1 - c_x}{f_x} \\ d\frac{x_2 - c_y}{f_y} \\ d \end{bmatrix} \tag{2.12}$$

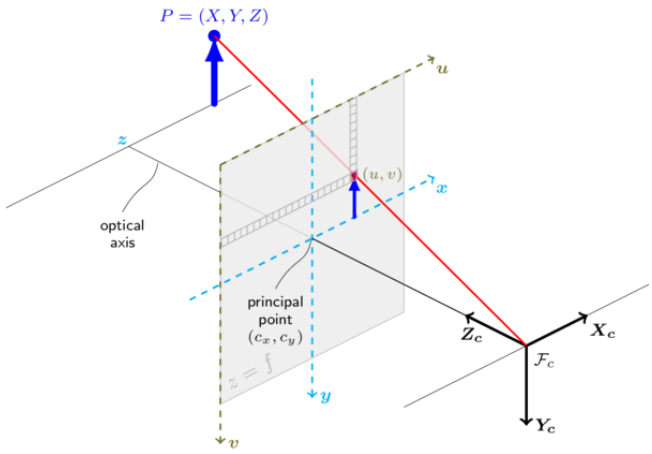


Figure 2.1: Pinhole camera model. Image from the OpenCV documentation

2.2.2 Lens distortion

A real camera has additional effects not captured by the pin-hole camera model. This includes lens distortion effects, where tangential and radial distortion effects are the most prominent. This project uses three parameters to describe the radial distortion, k_1, k_2, k_3, and two for the tangential distortion, p_1, p_2.

2.3 ICP - Iterative Closest Points

2.3.1 ICP

Iterative closest points (ICP) is a class of algorithms used to iteratively align two sets of points [6]. When aligning two point clouds, one cloud is kept fixed while the other is moved. At each iteration every moving point has it's nearest neighbouring point from the fixed cloud determined, then the error between the corresponding points is exspressed as a function of movement in the movable cloud, and the optimal movement is found through optimization. This is done over and over until the point clouds are aligned. This is illustrated for the 2D case in figure 2.2. A common optimization scheme is to use Gauss newton to minimize the sum of squared distances. Before the minimization step outliers can be rejected. Point-to-line or point-to-plane distances are a popular alternative to point-to-point distances as an error metric in ICP, as it typically converges in fewer iterations [7] and is more precise. Point-to-line and point-to-point in 2D is illustrated in figure 2.3.

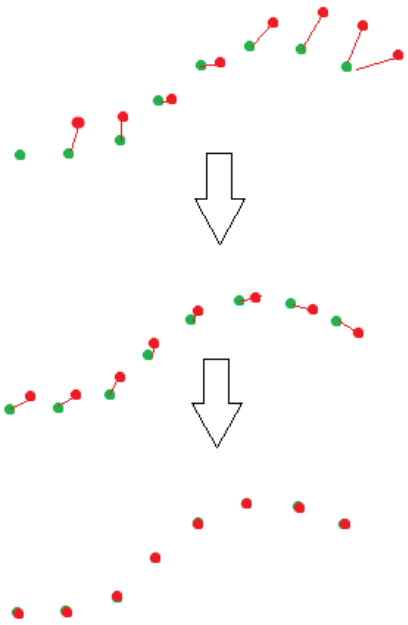


Figure 2.2: Basic 2D ICP

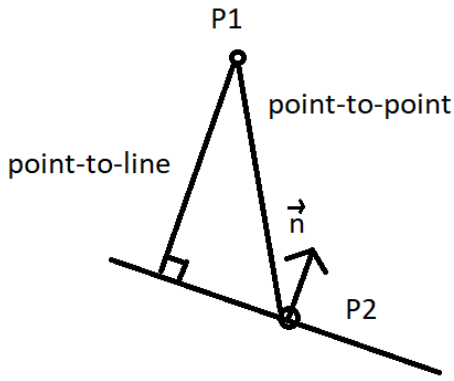


Figure 2.3: point-to-line and point-to-point disatance in 2D

$$poinToLine = n^T(P_2 - P_1) \tag{2.13}$$

Projective ICP is a flavor of ICP where 2D points in a plane are aligned with projections of 3D points onto that plane, a simplified illustration shown in figure 2.4. The movement is typically expressed as a transformation of the camera-frame.

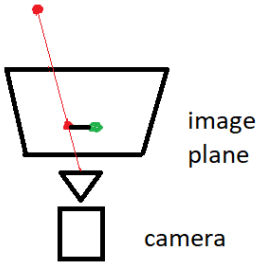


Figure 2.4: Point correspondence in projective ICP. Green is the 2D point, while red is the projected 3D point

2.3.2 Gauss Newton Optimization

Gauss-newton can be used to minimize non-linear least squares problems, eq. 2.14, for example the point distances in an ICP iteration. The error E is a function of some parameter β. The jacobian, J_i for each residual function r_i is calculated, and residuals evaluated at $\beta = 0$ and jacobians are concatenated in their respective arrays r and J, which is used to calculate an update to β that minimizes the error, eq. 2.16.

$$\min E(\beta) = \sum_i (r_i(\beta))^2 \tag{2.14}$$

$$J_i = \frac{\partial r_i}{\partial \beta} \tag{2.15}$$

$$\Delta\beta = -(J^T J)^{-1} J^T r(0) \tag{2.16}$$

2.4 Edge detection

2.4.1 Sobel

Sobel edge detection is done by convolving an image withe sobel operators, which will approximate the x and y gradients of an image. The absolute value of the gradient is then used to determine if there is an edge present in the image or not by comparing it with threshold. The Sobel operators for the x and y directions are

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

2.4.2 Canny

Canny edge detection is a multi stage detection algorithm. First it smooths the image with a gaussian filter. Then it finds the intensity gradients. Then it does non-maximal suppression to remove pixels which most likely are not edges. Then hysteresis thresholding is performed to decide which edges to keep and which to discard. This final step uses two thresholds, max and min. Those above max are definitely strong edges and kept, and those below min are discarded. The remaining edges, between min and max, are kept if they are connected to a strong edge, the rest discarded.

Chapter 3

Available Data Set

3.1 Ship Seapath

Figure 3.1: The Hurtigrute ship Polarlys

Kongsberg Seatex has done extensive work in collecting data from active ships, and have installed their navigation systems as well as additional sensors on a Hurtigruten ship called Polarlys, shown in 3.1. This ship has traveled up and down the Norwegian coast, collecting images, navigational data, AIS data, radar data and weather data. The approximate route is shown in fig 3.2. The data is collected over the span of more than a year. The dataset is very rich and varying, as there is a big diversity in the terrain along the long Norwegian coast. The ship sees both natural environments and environments more dominated by man-made structures.

The name of the navigation system used is Seapath, which is a product created by Seatex. The system fuses GNSS, IMU and magnetometer data using kalman filtering, giving highly accurate position and attitude measurements for the ship. For the rest of the report the pose data from the Seapath system is just referred to as the ship's seapath. The ship pose, velocity and angular velocities as well as acceleration and angular acceleration is available. The ship's body frame is shown in fig 3.3, which is taken from a report of a survey done by Anko Bluepix, with Konsberg Seatex as a client. The body frame follows the convention for ships with positive x-axis forward along the vessel centerline, positive y-axis out the starboard side and the positive z-axis is down.

The seapath is defined in reference to WGS84, and in the dataset the positon is stored as lat, long, down, while the orientation is parameterized as z-y-x euler angles (yaw, pitch, roll) in degrees. This project uses the dataset's data for seapath, tidal height, camera images and parameters related to the camera pose and calibration. Tidal height is referenced to NN2000.

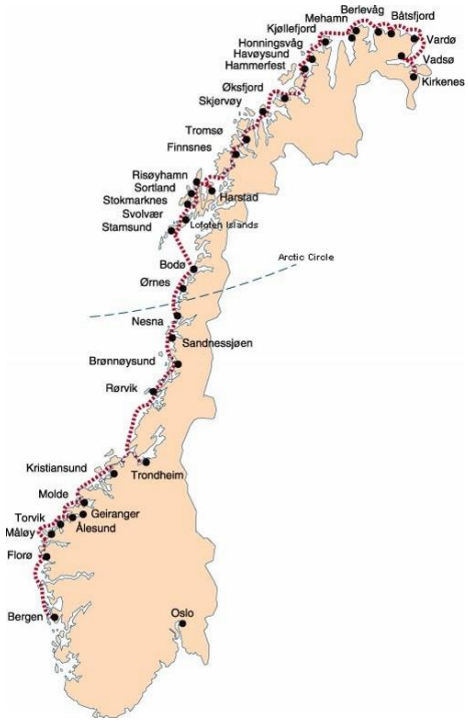


Figure 3.2: The route of hurtigruten, from lunga.no

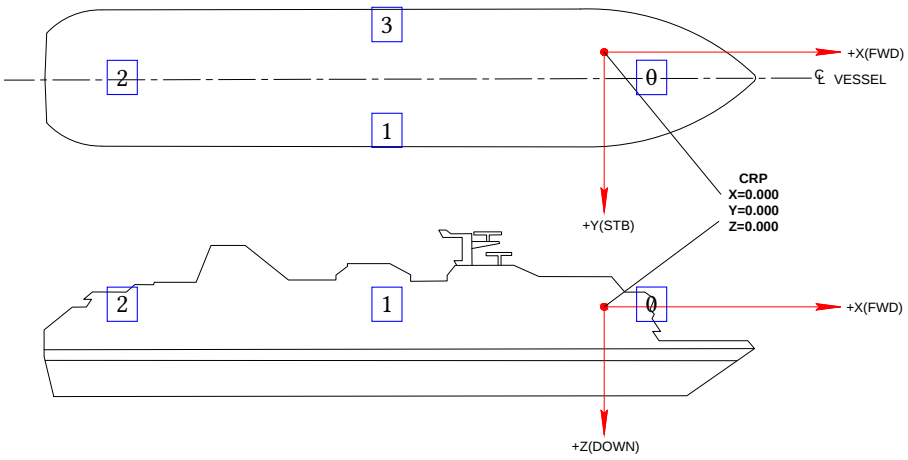


Figure 3.3: Sketch of the ship's reference frame. Camera clusters indicated with blue boxes

3.2 Cameras

3.2.1 Camera position and orientation

The dataset has synchronized the camera images to the seapath data, so that you can retrieve the pose of the ship the exact moment an image is taken. The pose of the camera relative to the ship is also available. On the ship there are four separate camera clusters, indicated in fig 3.3, all looking in different directions. Each of these clusters consist of three individual cameras, which are shielded behind a shared glass dome. The cameras are separate and each have different poses. Together all 12 cameras cover the full 360deg *panoramaviewwithoverlapbetweenimages.Usingonlythemiddlecamerasstillgivesgoodcovera*
y−xeuleranglesindegrees.Theanglesaredefinedforaframeinthecamerawherexpointsforwardoutof

3.2.2 Camera Calibration

The internal camera parameters, or camera intrinsics, are available from the same file and interface as the extrinsics, the camera poses. The calibration was done by Seatex employees before the project thesis was started in August 2018, and has not not done again since. Therefore we won't go into great detail on the calibration process. Suffice to say chessboards were used. The cameras are modelled with the pinhole camera model, and a 5 parameter lens distortion model, 3 radial coefficients and 2 tangential. Each camera has been calibrated individually, and so the intrinsic matrix is unique for each camera, as are the distortion parameters. The Seatex employees calibrated the cameras taking into account the distortion caused by the glass dome. The calibration is discovered to have some error though. Either due to the distortion caused by the glass dome to not be captured by the relatively simple distortion model, or due to the cameras changing their orientation over time, thus looking through the glass at new angles, which requires recalibration. We know for a fact that the angles change over time, and that calibration is not done as a response to angle change. The undistortion is particularly poor for the tilted cameras on the side of each camera cluster, which suggests that the distortion model might not have the expressive power to capture such a complex lens distortion. The middle cameras are all good, likely since they look straight through the dome, resulting in a more symmetric and simple distortion. For this reason only the middle cameras are used for the rest of the project, both in the creation of segmentation training data, and also for the localization tests. The tilted side cameras are simply ignored. The superiority of the middle camera's calibration is apparent in 3.5, where undistorted images from a middle and side camera are compared with their respective model renderings, as per chapter 4. The same mountain ranges are visible in both images, and only in the side camera are they distorted, ruling out 3D model inaccuracies as the cause.

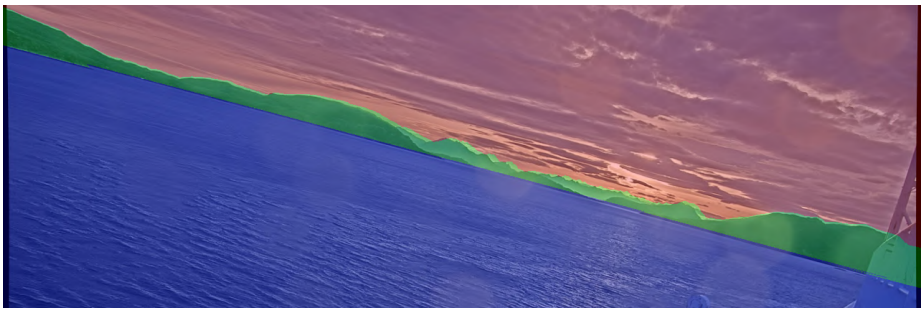
When I later in the project access any images from the dataset, I immediately undo the distortion. This is done using tools from OpenCV. First a new optimal camera matrix is calculated using the original camera matrix, the distortion parameters and the

dimensions of the image using openCV's getOptimalNewCameraMatrix. The images can now be undistorted using OpenCV's function undistort, which takes a distorted image, orignal camera matrix, new camera matrix and distortion coefficients. It returns an undistorted version of the input image, as well as a region of interest that can be used to crop out the black curved areas at the image edges, which are a bi-product of the undistortion process. The black areas are cropped out in later stages, until then the image resolution is kept consistent. From this point forward the camera intrinsic and camera matrix refers not to the original camera matrix, but the new optimal one, as this now represent the pin-hole-model related to the newly undistorted image. An example of an undistorted image is shown in figure 3.4.
I decided to not attempt any recalibration with more complex distortion models, as the scope of the project is already very large, and the main focus is on making a localization proto-type. The middle cameras' calibrations are still of good enough quality to make a proto-type.

(a) Camera image with lens distortion effects (b) Undistorted camera image

Figure 3.4: The result of undistorting a camera image

(a) Forward camera cluster, middle camera



(b) Portside camera cluster, right camera

Figure 3.5: Undistorted and cropped camera images compared with model rendering using optimal camera matrices

Chapter 4

Terrain Model Generation and Rendering Engine

4.1 Heightmap Processing

4.1.1 Acquiring Heightmap Data

The source of the height data used to create the 3D models is a WMS server (web map service) from Geonorge.no, the Norwegian national web service for map data and other georeferenced data, developed and managed by Kartverket. This service offers both DTM data (digital terrengmodell, digital terrain model) and DOM data (digital overflatemodell, digital surface model). DTM models the underlying geological structures, while DOM also covers vegetation and human made structures. There is also data available on NAS devices (network accessible storage) through the local Seatex network, but this does not include DOM data, just DTM.
Heightmap requests to the WMS server must contain lat long bounding box coordinates, which map source the data will be collected from as well as the resolution of the heightmap in pixels. The DOM source data is only scale 1, while for the DTM data there are three separate sources with scale 1, 10 and 50. The scale signifies the distance in meters between each unique sample point. The lower scale data naturally captures more detailed structures than the higher scale data. Lower scale data can be sampled to provide higher scale data by dropping measurements, losing detail in the process. A higher scale source can also be sampled to provide lower scale, duplicating or interpolating measurements between the real sample points.

4.1.2 Missing and Erroneous Heightmap Data

0 is used as both the value for signaling that there is no available data, that the pixel is ocean, and also as just a normal height-value, varying from region to region. For some regions water is 0, while for others negative values are used for areas under water. Note that 0 does not mean waterline, the 0 reference is used according to NN2000. This inconsistency created a lot of instability in the model generation, and so an assumption was made that negative values could be rounded up to 0, and it would be mostly unnoticeable. This causes problems for some areas, where the tide goes lower than 0 and the ship data happens to be sampled at low tide. But the benefit of

simpler generation outweighs having to delete a couple of bad model files.
A big problem with using the DOM data is that some areas were never measured. This results in huge chunks missing from the terrain models. Luckily there is a default value for missing regions, 0. Thus the DOM data is replaced with DTM data wherever DOM==0. One such example is shown in 4.1. A further complication is that for some areas the DTM of scale 1 is also missing data. This is rare, but in these areas the DTM data of scale 10 can be downsampled and used in lieu of the scale 1 data. This gives much lower resolution and very course structures, and for shorelines and small islands this is particularly visible. However a visual inspection is done of all the heightmaps and their renderings, and many recovered models are perfectly valid. The bad models are are simply discarded.

(a) Incomplete DOM data (b) Replacement DTM data (c) Combined DOM and DTM

Figure 4.1: Replacing missing DOM data with DTM data. Model of the Trondheim Fjord

Another problem is that in the DOM data set is that the ocean is measured just like any part of the terrain. If these measurements are not removed, the model would include a huge high-res structure representing the ocean, that also is rigid. Since it is rigid, it does not generally represent the ocean at the time we're interesed in, since the tidal height varies. And since it's high-res it's a huge waste of resources. An illustration of this is shown in 4.2, where a small DOM heightmap of a dock area is converted to a triangle mesh (see section 4.2) and visualized in Meshlab. The solution is to use the more reliable DTM data to create an ocean-mask, that can be used to mask

out the DOM ocean measurements, while still using the DOM data for land structures. The ocean-mask is simply a mask of where DTM==0, and masked areas in the DOM are replaced with DTM data.
Since the DTM is used as an ocean mask, structures in areas the DTM says is ocean will not be included in the final terrain model. This mostly affects structures such as poles and markers. I experimented with filtering them out by including DOM values that were higher than some threshold, either set by tidal height or the lowest values in the heightmaps, but I found no simple solution that gave consistently good models across different regions. The available data is just not good enough to create perfect models without much turmoil. It could probably be done using some more advanced filtering technique, but due to the scope of the project there is not time enough to develop each sub-system to their full potential.

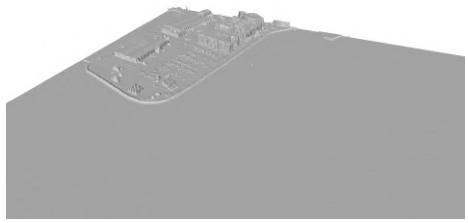


Figure 4.2: Meshlab visualization of a small DOM heightmap directly converted to triangle mesh without removing ocean

4.1.3 NED height correction

The map uses the NED coordinate frame, approximating the spherical earth by it's tangent plane. Since it is an approximation, it causes errors when the distance from the center increases. At 50km from the center, the approximation error is significant, 196m. These distances are not uncommon in the images, and still medium distance objects suffer from the approximation error. When comparing a render with real images it is very obvious. To fix this I calculate the curvature height drop, which is a function of distance from the map center where the tangent plane intersects the earth. This

is calculated for a grid with the same dimensions as the heightmap, which is then subtracted from the heightmap. The height drop is calculated as follows. h is the height drop, d the distance from map center and r is the earth radius. The image is also warped in the NE-plane, but this is already done by the WMS interface. This is an approximation, but it is perfectly sufficient for our purposes. The rendered model after the height correction can be aligned well with the properly undistorted camera images. Of course this assumes that the vessel will only move near the map center, as the pose is defined in NED, and a new map must be loaded if the ship were to move to far from the center.

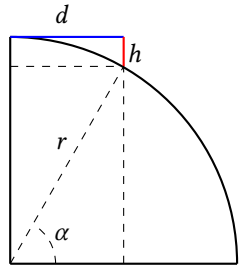


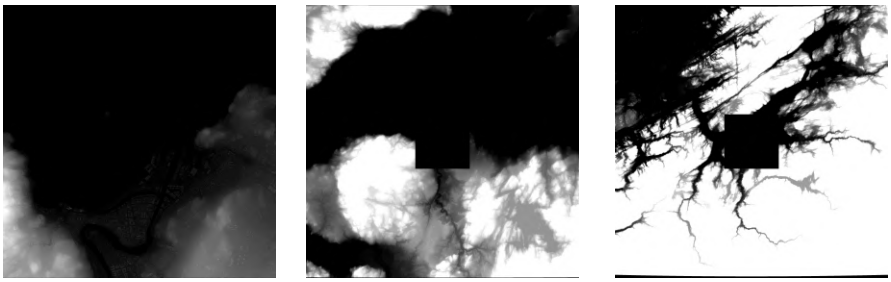
Figure 4.3: NED height approximation error h (red) at distance d (blue) from NED origin, the tangent plane intersection with Earth

$$cos(\alpha) = \frac{d}{r}$$
$$\alpha = cos^{-1}(\frac{d}{r})$$

$$sin(\alpha) = 1 - \frac{h}{r}$$
$$h = r(1 - sin(cos^{-1}(\frac{d}{r}))) \tag{4.1}$$

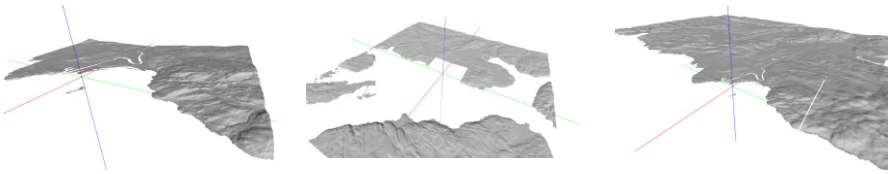
4.1.4 Varying Levels of Detail

Model areas closer to the ship requires a lower scale, to show the details of the environment, like docks or individual trees, while far away areas can have lower resolution without affecting their appearance in the camera images. To achieve this multiple models of different scales are created and used simultaneously. To use all these different scale models they must fit together, i.e. the overlap in the center must be resolved. The larger heightmap has it's center area of equivalent size to the smaller heightmap removed. The area is removed by setting the heights to 0, which is defined as ocean. This way the area will be removed for the rendering, but a sloped wall going down to the ocean will remain at the edge of the cut area. Conversion of heihtmap to triangle mesh is covered in ch. 4.2. This sloped wall masks any potential gaps in the seam between the smaller and larger model. Now the resulting combination has more detail in the center, and larger less detailed areas outisde of the center. Scale 1 is the most details we can get for close structures, but a heightmap of scale 1 would cover a very small area before scale 10 takes over, depending on the height and width of the heighmap. A good combination of pixel resolution and scales seems to be 3000x3000 pixel resolution, and scales 2, 10 and 50. The resolution is as big as as the WMS allows, and each scale is a 5x multiple of the others. For the largest heightmap the distance from the map center to a map edge is 75km. Objects at this distance are generally not visible, either due to the atmosphere, being hidden behind other terrain or due to being over the horizon. An example of heightmaps made to fit together is shown in figure 4.4 and 4.5.



(a) Scale 2 (b) Scale 10 (c) Scale 50

Figure 4.4: Heightmaps of different scales, made to fit together. Maps are of the Trondheim Fjord



(a) DMT with scale 10 (b) DTM with scale 50 (c) Combined models

Figure 4.5: Meshlab visualization of two models made to fit together. Models are of the Trondheim Fjord

4.1.5 Ocean Tides

The ocean measurements from the DOM data were removed due to generally not being at the correct height, and for requiring too much data despite just being a surface. The ocean is added back into the world by being modelled as a separate object from the terrain. Since it is a curved surface it cannot simply be modelled as just a large square, but 1m resolution is not necessary either. I create an array of 500x500 pixels, with scale=300. The curvature does not change noticeably in 300m, and with this resolution and scale, the ocean is exacly the same size as the largest heightmap. A flat grid with height 0 is created, and then the curvature height adjustment is applied to form a representation of a curved surface, that will act as the ocean in our model. Later when

rendering, the ocean vertices' height values can be shifted to move the ocean to the correct tidal height.

4.2 Triangle Mesh Creation

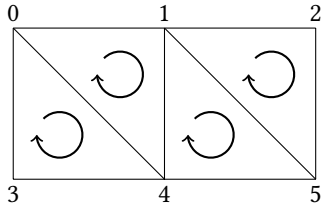
The heightmaps processed from the previous section 4.1 must be converted to a format compatible with modern model rendering techniques. A square mesh structure was considered since the heightmap is a square grid, but even though I would be spared half the polygon faces, I went with triangle faces, since it is easier to check if a face is defined correctly. Furthermore 4 vertices are not necessarily in a plane, unlike 3, which may complicate the square mesh generation. Finally triangles are the standard polygonal mesh structure, and graphics hardware is optimized for triangles, and eventually converts all polygons into triangles [19]. The triangle mesh is represented using indexing, to avoid duplication of vertices, thus using less memory.

The triangle mesh generation algorithm takes as input a heightmap, a color, the scale of the map as well as a landmask signifying weather a point in the heightmap is land (1) or ocean (0). It outputs a list of 3D vertices, and a list of triangle faces that models the land terrain in the heightmap. It also outputs a list specifying the color of each vertex, which are all set to the input color. Land terrain is created to be green. When creating the curved ocean object described in 4.1.5, the landmask is set as all ones to create a mesh from all the data, and the color is blue.

A triangle face consists of three indices, where each index corresponds to a vertex in the vertex array. The three vertices forming a triangle face are listed such that the sequence forms a clockwise movement when viewed from the intended outside of the mesh. This is shown in figure 4.6. from each square in the grid there exists the potential for two triangles, one upper (includes top right corner) and one lower (includes bottom left corner). The vertices are defined in a NED-frame, and the origin is calculated as the center of the map, accounting for half pixel offsets.

The algorithm results in no vertex duplicates nor any unused vertices being included in the vertex list. It also avoids creating any triangles that are wholly ocean, while keeping the ocean vertices necessary to form the shoreline triangles, so that the terrain does not appear to have gaps at the transition between land and ocean. Which vertices are discarded and which are used to form triangles is visualized in 4.7. Meticulous optimization and use of numpy indexing has reduced the mesh creation time from minutes

(similar system in project thesis) down to a second for a 3000x3000 heightmap. At these matrix sizes and at the current speed the biggest bottleneck is simply initializing new numpy arrays, as it's a more complex data type than the native python list. The mesh data arrays are cached as a .npy binary file, to be loaded quickly when rendering in a new region.



(a) Triangle mesh with vertex indices

[0, 1, 4]
[1, 2, 5]
[0, 4, 3]
[1, 5, 4]

(b) Triangle faces using vertex indices

Figure 4.6: Construction of triangle faces with shared vertices

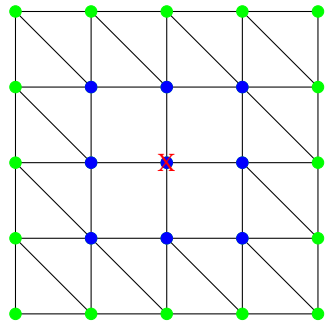


Figure 4.7: Heightmap with land (green, height>0) and ocean (blue, height=0) converted to triangle mesh (triangles). Unused vertex marked with red X

The mesh generation script is rather short, but manages to do quite a bit. A vertex is to be included in the vertex list if it is a part of any triangle that has at least one land vertex. A mask signifying if a vertex is the origin vertex of an upper or lower triangle is calculated using the logical OR operation on shifted slices of the landmask, shifted such that the vertices of a triangle share the same position. Since ocean is 0,

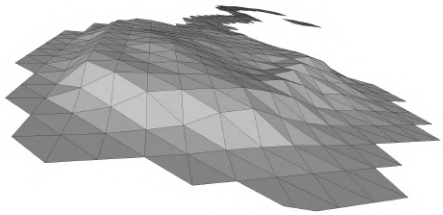


Figure 4.8: Meshlab visualization of how the 3D models consist of trianges

the OR needs needs just one land vertex in the triangle to yield valid. A new mask signifying weather or not a vertex is part of any triangle at all, not just the origin of one, is constructed by taking slices of a 0-initialized array and summing in the validity arrays for triangle origin, sliced such that the valid origin vertex affects the vertices of the triangle they would form. A vertex list of all the valid vertices can now be constructed, calculating each vertex's scaled vector distance from origin. I take the cumulative sum of the validity array, and this is used as the vertex index values. Since it is a cumulative sum of a mask (0 or 1) the valid vertices tick the index count by 1, while invalid vertexes are ignored. This way each valid vertex is attributed an unique index. These indices also correspond to each vertex's position in a flattened list of the valid vertices. This index matrix is used to create the upper and lower triangle faces by concatenating three index array slices, sliced according to the shape of the triangle, and masking out the valid ones using the already computed validity masks. This is done separately for the upper and the lower triangle faces, since they have different shapes and require different slicing. The upper and lower triangle faces are concatenated to create one single triangle face list, that references the already created vertex list. The color array is finally created by filling an array of the same size as the vertex array with the inputted color value. The vertex array, triangle face array, and the color array are then cached in a single .npy file, to be loaded later.

4.3 Model Rendering with OpenGL

The goal of the model creating and rendering engine is to simulate a real camera view. The simulated view, or render, of an associated camera image will provide information about what is visible in the camera image. Each pixels is labelled as either sky, land or ocean, which is encoded in the rendering as colors. Red sky, green land and blue ocean. The rendering system is initialized for a geographical region and time, after which the model can be rendered from arbitrary view points with arbitrary camera intrinsic matrices. Different camera views can be rendered in succession without re-initialization. The renderer is implemented as a python class.
The rendering system is initialized with 6 parameters. It is given the desired resolution of the rendered image, a list of triangle meshes for the land, and a single triangle mesh for the ocean, as well as the tidal height.
A rendering call takes 3 arguments. The first argument is a rigid transformation matrix between the ship pose and the NED frame at the center of the map, with the ship frame as in 3.1. The second argument is a rigid transformation matrix between the ship frame and the camera mounting frame as in 3.2.1. The third argument is a camera intrinsic matrix.

4.3.1 Modelling the Real Camera in OpenGL

The camera lens distortion effects are already accounted for by undistorting the camera images, detailed in chapter 3.2.2. This means the lens effects are irrelevant for the rendering, and only the pose of the camera frame relative to the world frame, as well as the camera intrinsic matrix is required to map 3D model points to the correct pixels in the rendered image. This mapping from model to pixels is unique in OpenGL.
The transformation matrix to express a point in the world frame in the ship frame is denoted T_w^s. Similarly the matrix for ship to camera mount is T_s^m. The transformation matrices are created with a helper function from the python library pyrr, which takes into account the row-col convention OpenGL uses for matrix multiplication. The helper function takes as input euler andgles and a position vector.
Now, the camera mounting frame is not the same as the traditional camera frame, and

so a matrix for mount to cam is specified as T_m^c, which is just a sequence of rotations of 90 deg around z and x. This is necessary because the projection from camera frame to pixel coordinates assumes the points are expressed in the traditional camera frame. Additionally openGL follows yet another convention where the camera is assumed to be looking in direction of negative z-axis. This is handled by T_{GL}, which is calculated with a pyrr helper function called lookat. Then to the projection from camera frame to pixel coordinates. This projection matrix is based on the camera intrinsix matrix, but modified specifically for OpenGL, which has some additional functionality, as well as some different conventions. This projection matrix is denoted P_{GL}. This matrix, when used in OpenGL, also ignores points which are closer than *near* or farther away than far. Near and far are set to 5 and 10^5 respectively, which is closer and farther than any point is expected to be to the camera.

$$P_{GL} = \begin{bmatrix} 2\frac{f_x}{w} & 0 & 0 & 0 \\ 0 & 2\frac{f_y}{h} & 0 & 0 \\ 2\frac{c_x}{w} - 1 & 2\frac{c_y}{h} - 1 & \frac{far+near}{near-far} & -1 \\ 0 & 0 & 2\frac{far*near}{near-far} & 0 \end{bmatrix} \tag{4.2}$$

The full transformation from a point expressed in model/world coordinates to openGL pixel coordinates is denoted MVP (model, view, projection).

$$MVP = P_{GL} T_{GL} T_m^c T_s^m T_w^s \tag{4.3}$$

This MVP matrix is 4x4 float, and is passed to the the vertex shader as a glUniformMatrix4fv. The vertex shader multiplies the incoming vertex points with MVP and outputs this as gl_Position, the vertex shader also passes the incoming vertex colors to the fragment shader. The fragment shader simply outputs the incoming color as FragColor. The shader code is stored in text files, and during initialization of the renderer they are compiled into a shader program and set to be used by calling glUseProgram(shaderProgram). The code for the fragment-fragment and vertex-shaders is shown in the following listing.

Listing 4.1: Vertex Shader

```
#version 330 core
layout(location=0) in vec3 vertexPos_modelspace;
layout(location=1) in vec3 vertexColor;
out vec3 color;
uniform mat4 MVP;
void main() {
    //output pos of vertex in clip space
    gl_Position = MVP*vec4(vertexPos_modelspace, 1);
    color = vertexColor;
}
```

Listing 4.2: Fragment Shader Code

```
#version 330 core
in vec3 color;
out vec3 FragColor;
void main() {
    FragColor = color;
}
```

4.3.2 Triangle Mesh Loading

In order to render the triangle mesh described in the previous section 4.2, the vertex list, and the triangle face list and the color list will be loaded into the GPU's memory. This is done with the use of VBOs (vertex buffer objects). Since multiple separate heightmaps are converted into triangle meshes, and are made to be rendered simultaneously, the triangle meshes will be combined into a single triangle mesh, with a single vertex list and a single list of triangle faces. Since the models were created individually, all their triangle face lists assume that their corresponding vertex lists start at index 0. When concatenating the vertex lists into one, an index offset is added to the triangle face lists. This offset ensures that the triangle faces reference the correct indices in the new concatenated vertex lists. The offset is equal to the number of vertices that have already been added to the concatenated list.
When loading it is specified which .npy file is the ocean, and during loading the vertices

are shifted to the correct tidal height. All this is implemented as a python class called MeshContainer, which is used by the renderer as a data container.
Before rendering, the list of vertices, list of vertex colors, and and list of triangle faces are put in two attribute buffers and an index buffer respectively. In the current implementation data is intended to be loaded once and not be modified. This means that filling the buffers using a GL_STATIC_DRAW hint will give optimal performance.

4.3.3 Rendering and Retrieving Data

The rendering happens off-screen without actually opening a window and changing pixels on a physical screen. This is accomplished in openGL with the use of a frame buffer object. The frame buffer object consists of two render buffer objects; one for the color image and one for depth data. Both of these must be initialized and OpenGL must be told to render into the framebuffer, not the real screen. After a render call the image and depth can then be retrieved from the frame buffer.

GLFW is a lightweight library used to handle low-level OS tasks that OpenGL itself doesn't provide the necessary mechanisms for. GLFW provides a programming interface for creating and managing windows with openGL context. A python binding for GLFW is being used. Even though the renderer does 'windowless' or 'offscreen' rendering, openGL still requires a context to function. There are alternatives available, such as FreeGLUT, but GLFW is chosen primarily because there exists some extensive beginner friendly tutorials for it.

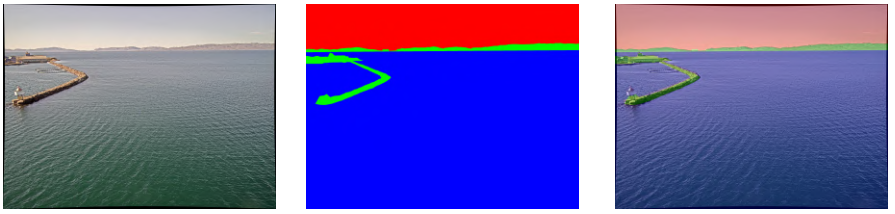
The rendered image should not have any color blending, so that the rendering color directly encodes the semantic class of the pixel. Pure red is sea, pure green is land, and pure blue is sky. To stop the blending of edges, GL_POLYGON_SMOOTH is disabled. Having the land, sky and ocean entities completely separated by color channel makes some processing later very simple. The loaded ocean mesh is already blue, and the loaded terrain mesh is already green. The sky is not modelled by any actual objects, and so it makes sense to use a background color for the rendering to simulate the virtual sky. This is done by calling the glClearColor function with the color blue (0,0,255) as an argument.

The screen buffer must be emptied between each render, so as to muddle new renderings with old data. This is done by calling glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT), | signifying bitwise OR.
Vertex attribute array 0 and 1 are in turn enabled and bound to the vertex buffer and vertex color buffer respectively. This way the vertices and their colors will be available to the shader. Then the GL_ELEMENT_ARRAY_BUFFER is bound to the triangle face buffer. glDrawElements is called with GL_TRIANGLES, specifying that the element array buffer contrains triangles, and also the size of the triangle face list. The model has now been rendered, and the rendered data is available in buffers.
The use of a frambufferobject makes the rendering process slightly more manual. To make the render data accessible, first the glReadBuffer function is called with GL_COLOR_ATTACHMENT0. This loads the render data into buffers that can be accessed with glReadPixels. The rendered image is retrieved using glReadPixels with GL_RGB_COMPONENT as argument. The data is parsed using numpy's frombuffer function, supplying the datatype uint8. Since the buffer data is shapeless, it is reshaped back into a 2D array using the known image dimensions.
The depth data is retrieved using glReadPixels with GL_DEPTH_COMPONENT as argument. To make openGL calculate the depth data, GL_DEPTH_TEST must have been enabled. We're still using numpy's frombuffer function, this time with datatype float32, to parse the data. This data is also reshaped to the original image dimensions. The depth values are specified using some internal openGL units, which has a nonlinear relationship with metric depth. The openGL depths can be converted to meters using eq 4.4, where near and far are the near and far clip distances, planes outside of which vertices won't be rendered. The depth value is the distance along a back-projected ray to the first object it hits. That is, it is the actual distance from the camera, not just the distance from the image plane.

$$depth_{metric} = \frac{(\frac{near*far}{near-far})}{(\frac{depth_{gl}-far}{far-near})} \tag{4.4}$$

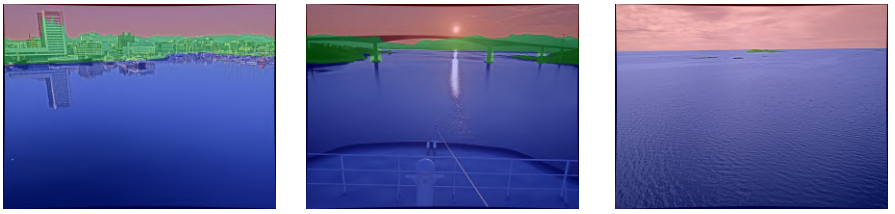
An example of a model rendering using the real ship pose and camera data is

shown in figure 4.9. The rendering is a compared to a real camera image, and it can be seen that they match very well. I is apparent that a good rendering can be used as semantic ground truth for a camera image. Figure 4.10 shows a rendenderings of models with some malformed structures. (a) also demonstrates the fact that boats are not part of the rendering, and are subsequently labelled by whatever is behind them. Overly malformed models are just discarded, but another problem that affects all renderings is the fact that the camera angles are somewhat inaccurate, causing a slight misalignment between the camera image and the rendering. This can be seen in figure 4.11 where the rendering is tilted so that it is below the real terrain. This is looked into in the next chapter, chapter 5.



(a) Undistorted camera image (b) Rendered image (c) Camera and render overlay

Figure 4.9: Decent model rendering compared with corresponding real camera image



(a) malformed building (b) malformed bridge (c) missing small reef

Figure 4.10: Bad model rendering compared with corresponding real camera image

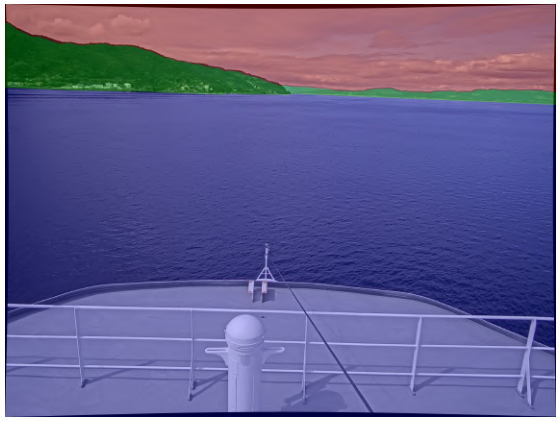


Figure 4.11: Inaccurate camera angle in render-camera overlay

Chapter 5

Camera Mounting Angle Correction

5.0.1 Orientation estimation with ICP

The idea is to do edge alignment between canny edges in a real image and edges from a rendering given ship and camera pose. The edges from the camera are pixel coordinates, while edges for the rendering are back-projected to 3D using the camera intrinsics and the rendered depth map. The orientation is then optimized using ICP, where the error measure is the squared distance between the re-projected render edges and their nearest neighbour edges from the camera image edges. The re-projection of the 3D render edges is a function of the change in camera orientation, for which an optimal value is found iteratively. This change in camera orientation is parameterized on-manifold using lie algebra. Since the camera angles were believed to change slowly over time, the tools was not intended to be very robust or reliable, and only had to be accurate for a fraction of the attempts, and the outliers could be filtered out afterwards. Besides, at this point in the project the more accurate and reliable pose estimation tool described in ch. 7 was not developed yet. This entire camera angle correction tool is essentially a re-implementation of the system developed in the project thesis in the previous semester.

The process starts with detecting edges in the camera image by using canny edge detection with parameters 50,175. The parameters were chosen to capture as many of the real contour edges at the cost of more clutter, since the camera angles are pretty close to the correct values already, so missing contours would be a bigger problem than clutter.
Then an initial view is rendered using an initial orientation estimate and calculating the edges of the render image using the sobel operator. Only the edges corresponding to the transition from land to sky in the model is used by setting the ocean to be the same color as the land. This is because the canny edges at the top of the terrain is more reliable than the canny edges near the shoreline, and so these are the only ones we want to align our model with. This is illustrated in figure 5.1. Furthermore the completely flat shoreline when far away adds little to the optimization other than

slowing down horizontal convergence if a match is even found. As specified in ch. 3, the images have already been undistorted, and the corresponding camera intrinsics are used in the rendering. The edges are sampled randomly for a speed increase and then back-projected. The edges are not re-rendered after every step in orientation since change in orientation barely changes the relationship between the edges. ICP is instead done by just rotation the edges with the current cumulative change in orientation and then re-projecting them from there. Multiple runs of this ICP-scheme is run which each render anew, but each run only renders once. The point-to-point error metric has slow convergence properties, and so multiple runs are needed.

(a) Camera image, zoomed

(b) Canny edges of camera image, zoomed

(c) Rendering of model's sky-land transition, zoomed

Figure 5.1: Side by side comparison of camera image, canny edges of camera image, and model rendering

The update to the orientation at each step is calculated analytically, shown at the end of this section. the orientation change is initialized as $\mathbb{I}_{3x3}$, and since it is defined in the camera image frame, when the ICP iterations are done the orientation is converted to specify a new rotation from ship to camera-mount. The resulting rotation matrix is then converted to Euler angles, which is what the data set uses.

X_i is a render contour point in the 3D camera frame, and x_i' is the corresponding nearest real image contour point in the 2D image plane. K is the camera intrinsic matrix. ω is the orientation parameterization, and J is the jacobian for a point correspondence's residual error. The papers [39] and in particular [40] were useful for seeing what the jacobian is supposed to look like, although they skipped the calculation.

$$r_i(\omega, R) = x_i' - \pi(Kexp(\hat{\omega})RX_i) \tag{5.1}$$

$$\pi(X) = \begin{bmatrix} \frac{X_1}{X_3} \\ \frac{X_2}{X_3} \end{bmatrix} \tag{5.2}$$

$$exp(\hat{\omega}) \approx \mathbb{I}_{3x3} + \hat{\omega} \tag{5.3}$$

$$J_i = \left.\frac{\partial r_i(\omega, R)}{\partial \omega}\right|_{\omega=\mathbf{0}} = -\frac{\partial \pi}{\partial K(\mathbb{I}_{3x3} + \hat{\omega})RX_i} \left.\frac{\partial K(\mathbb{I}_{3x3} + \hat{\omega})RX_i}{\partial \omega}\right|_{\omega=\mathbf{0}} \tag{5.4}$$

$$\frac{\partial \pi(X)}{\partial X} = \begin{bmatrix} \frac{1}{X_3} & 0 & -\frac{X_1}{(X_3)^2} \\ 0 & \frac{1}{X_3} & -\frac{X_2}{(X_3)^2} \end{bmatrix} \tag{5.5}$$

$$\tilde{X} = RX \tag{5.6}$$

$$K\tilde{X} = \begin{bmatrix} f_x\tilde{X_1} + c_x\tilde{X_3} \\ f_y\tilde{X_2} + c_y\tilde{X_3} \\ \tilde{X_3} \end{bmatrix} \tag{5.7}$$

$$\left.\frac{\partial \pi(K(\mathbb{I}_{3x3}+\hat{\omega})RX_i)}{\partial K(\mathbb{I}_{3x3}+\hat{\omega})RX_i}\right|_{\omega=\mathbf{0}} = \frac{\partial \pi(K\tilde{X})}{\partial K\tilde{X}} = \begin{bmatrix} \frac{1}{\tilde{X}_3} & 0 & -\frac{f_x\tilde{X}_1+c_x\tilde{X}_3}{(\tilde{X}_3)^2} \\ 0 & \frac{1}{\tilde{X}_3} & -\frac{f_y\tilde{X}_2+c_y\tilde{X}_3}{(\tilde{X}_3)^2} \end{bmatrix} \tag{5.8}$$

Using a vector-by-vector identity K can be kept outside, as it does not depend on ω

$$\begin{aligned} \left.\frac{\partial K(\mathbb{I}_{3x3}+\hat{\omega})RX_i}{\partial \omega}\right|_{\omega=\mathbf{0}} &= K\frac{\partial \begin{bmatrix} 1 & -\omega_3 & \omega_2 \\ \omega_3 & 1 & -\omega_1 \\ -\omega_2 & \omega_1 & 1 \end{bmatrix}\tilde{X}}{\partial \omega} \\ &= K\frac{\partial \begin{bmatrix} \tilde{X}_1 - \omega_3\tilde{X}_2 + \omega_2\tilde{X}_3 \\ \omega_3\tilde{X}_1 + \tilde{X}_2 - \omega_1\tilde{X}_3 \\ -\omega_2\tilde{X}_1 + \omega_1\tilde{X}_2 + \tilde{X}_3 \end{bmatrix}}{\partial \omega} = K\begin{bmatrix} 0 & \tilde{X}_3 & -\tilde{X}_2 \\ -\tilde{X}_3 & 0 & \tilde{X}_1 \\ \tilde{X}_2 & -\tilde{X}_1 & 0 \end{bmatrix} \end{aligned} \tag{5.9}$$

Inserting 5.8 and 5.9 back into 5.4 and multiplying K into the leftmost matrix yields

$$\begin{aligned} J_i &= \begin{bmatrix} \frac{1}{\tilde{X}_3} & 0 & -\frac{f_x\tilde{X}_1+c_x\tilde{X}_3}{(\tilde{X}_3)^2} \\ 0 & \frac{1}{\tilde{X}_3} & -\frac{f_y\tilde{X}_2+c_y\tilde{X}_3}{(\tilde{X}_3)^2} \end{bmatrix} K \begin{bmatrix} 0 & \tilde{X}_3 & -\tilde{X}_2 \\ -\tilde{X}_3 & 0 & \tilde{X}_1 \\ \tilde{X}_2 & -\tilde{X}_1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} \frac{f_x}{\tilde{X}_3} & 0 & -\frac{f_x\tilde{X}_1}{(\tilde{X}_3)^2} \\ 0 & \frac{f_y}{\tilde{X}_3} & -\frac{f_y\tilde{X}_2}{(\tilde{X}_3)^2} \end{bmatrix} \begin{bmatrix} 0 & \tilde{X}_3 & -\tilde{X}_2 \\ -\tilde{X}_3 & 0 & \tilde{X}_1 \\ \tilde{X}_2 & -\tilde{X}_1 & 0 \end{bmatrix} \end{aligned} \tag{5.10}$$

The jacobi matrices and residual errors are concatenated into their own respective larger matrices, which are used to calculate the optimal Gauss-Newton update step

$$\Delta\omega = -(J^T J)^{-1} J^T r(\omega = 0, R) \tag{5.11}$$

$$R \leftarrow exp(\Delta\omega)R \tag{5.12}$$

5.0.2 orientation estimation example

The convergence of the third euler parameter, relating to yaw of the camera, has a very slow convergence, seen in 5.2. This is due to the error metric used in the ICP. The error metric that is implemented is a point-to-point error. The ICP assumes that model points and image points that are close to each other correspond, and minimize their distance. When terrain contours are flat, a horizontal camera offset still yields a good match between the model and the image, working against the convergence to the global minimum. Furthermore the edge tracking is mostly just meant as a way to correct camera angles to create an accurate data set for semantic segmentation, in a way bootstrapping a better tracking system. The canny tracking method is particularly vulnerable to the presence man made structures, which often are very distinct. They give rise to a myriad of edges not related to the land-sky contours or the land-ocean contours, trapping the optimization in one of many local minima.

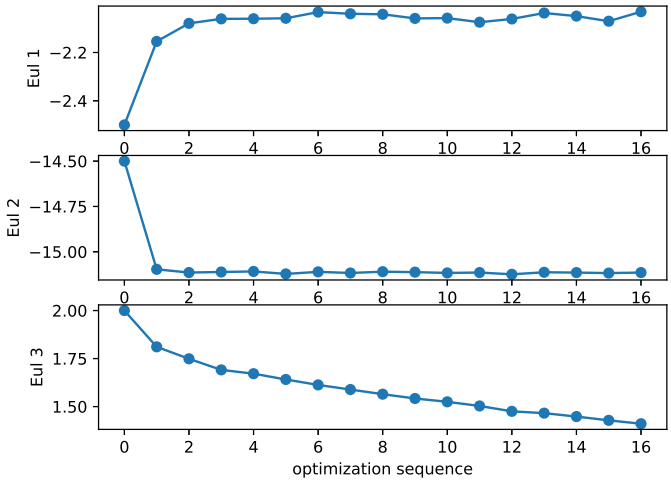
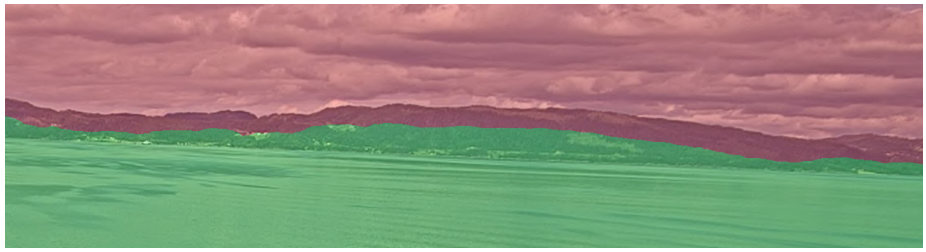
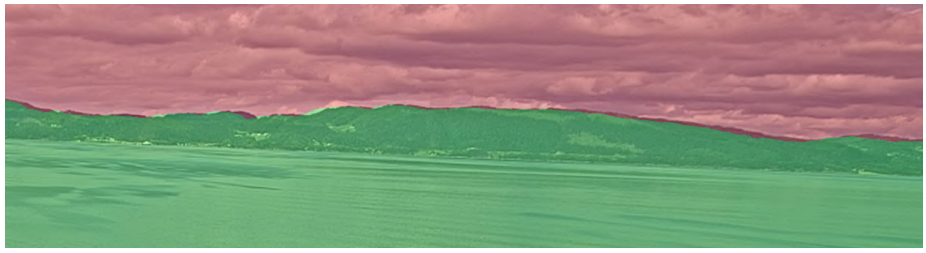


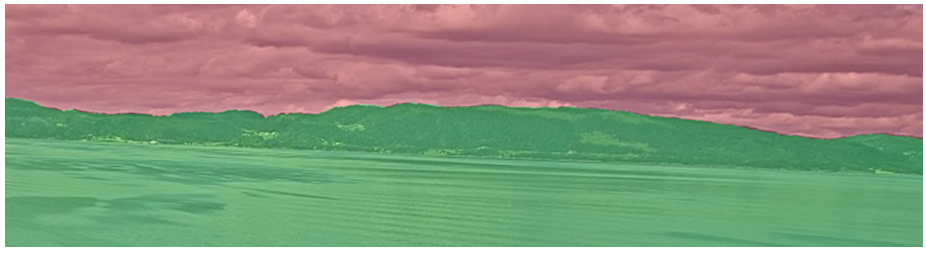
Figure 5.2: Camera orientation euler angles (degrees) during automatic angle correction



(a) Initial camera angle values, zoomed



(b) 1st iteration camera angle optimization, zoomed



(c) 15th iteration camera angle optimization, zoomed

Figure 5.3: Model rendering overlaid camera images during automatic camera angle correction

5.0.3 Correcting Camera Angles for Larger Time Periods

Using the list of timestamps from which segmentation data later will be generated (ch. 6.1), the angle optimization procedure is run for each timestamp. This creates a time-sequence of optimized camera angles, showing how the angle error evolves. At each time step the estimation is initialized at the current camera angle from the data set. For many timestamps there are insufficient contours to track, either due to the camera view or due to poor detection with canny. Furthermore there is a lot of contour clutter from the sky and ocean and the terrain itself, the cost of wanting to detect faint contours of far away mountains. Angles are optimized for each camera individually in 3 DoF, expressed in euler angles. The starboard side camera system was malfunctioning for a period of time, resulting in a shorter sequence for that camera (cam 1,1). The specific datetime corresponding to each point in each camera's sequence is stored in a file during the run, together with the angle values, but are omitted from the plot, as the essential information is the fact that the angles change, not specifically when. The time span for the data is around 4 months. The optimization over the timestamps is done in two consecutive runs. To save time the first run is done using every other timestamp. All points in the sequence are optimized independently from each other. The initial angle values at each time stamp are loaded from the data set, and may change in steps over time. The first run is used to manually update the data set's angle values for specified time periods, accounting for noise and outliers. Using the new angles a second run is performed, this time for all the timestamps. In this run in it appeared that the camera angles were now satisfactory, and no further change was made.

The second run clearly shows that the camera angles change periodically, apparently oscillating around some mean. This oscillation is very visible for the first ca. 100 datetimes in the second run, for all the cameras, because these datetimes are closer together and more consistently spaced. The eul2 parameter is the parameter with he most consistent and large oscillation. This is the parameter specifying the forward tilt of the camera, making the camera look upwards or downwards. The angles typically change over the course of a day, indicating that some cyclic event is responsible. This

could be the day itself; as the ship warms up and cools down the metal hull might get warped, thus slightly changing the orientation of the camera. The camera angles can also change their values in larger more permanent steps, shown in the port-side cam figure 5.11. Attitude bias being the cause of the angle errors is ruled out, as the camera angle errors do not correspond to the some change in ship pose. For some periods the cameras, facing in different directions, are all starting to tilt more upwards. A ship attitude bias would cause the aft camera to tilt down as the front camera tilts upwards.

5.0.3.1 First Angle Correction Run

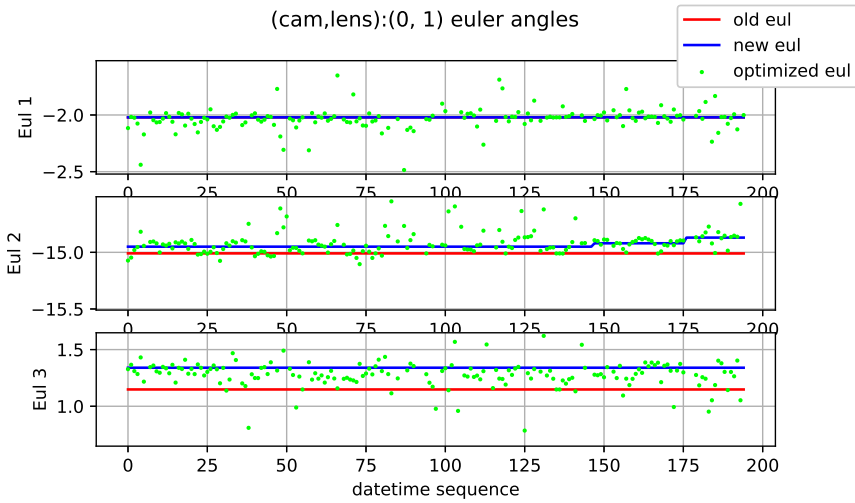


Figure 5.4: First angle optimization run, front cam

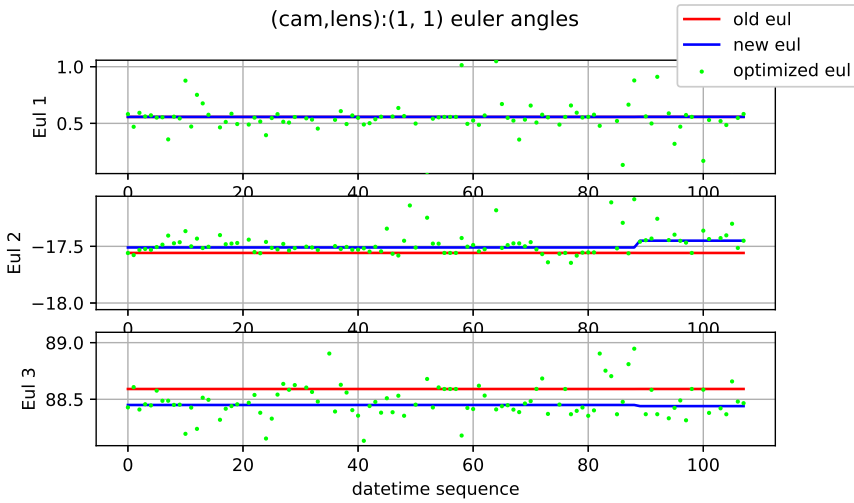


Figure 5.5: First angle optimization run, starboard cam

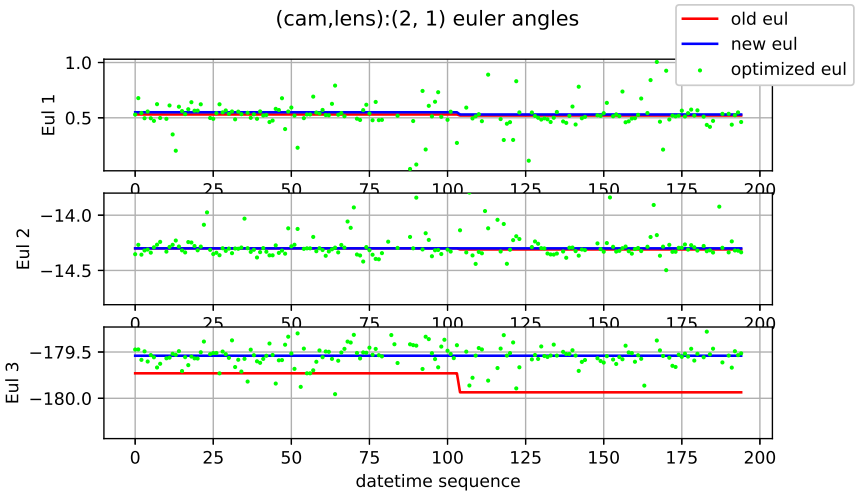


Figure 5.6: First angle optimization run, aft cam

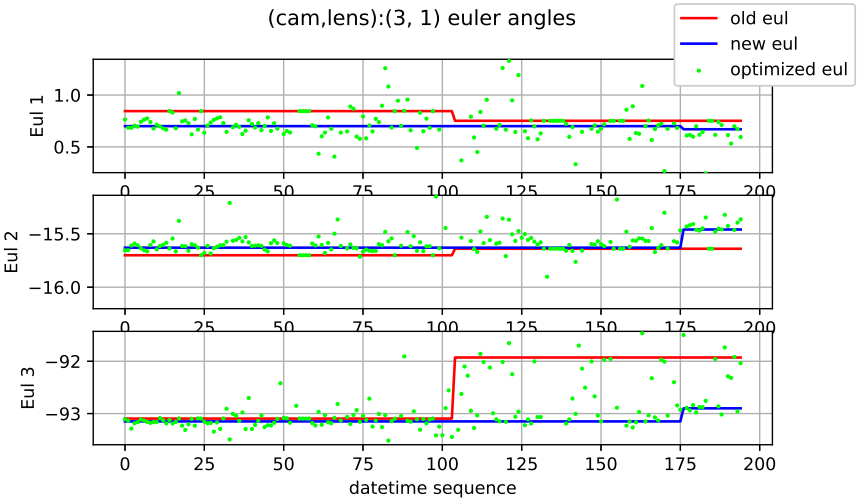


Figure 5.7: First angle optimization run, port-side cam

5.0.3.2 Second Angle Correction Run

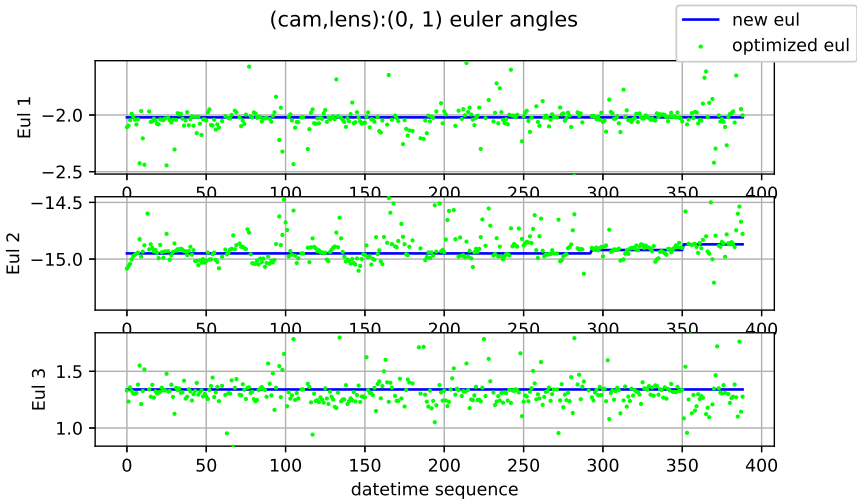


Figure 5.8: Second angle optimization run, front cam

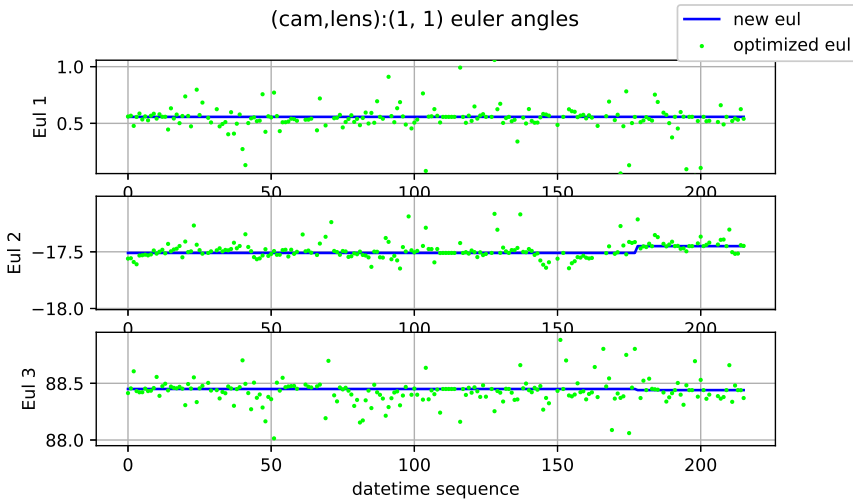


Figure 5.9: Second angle optimization run, starboard cam

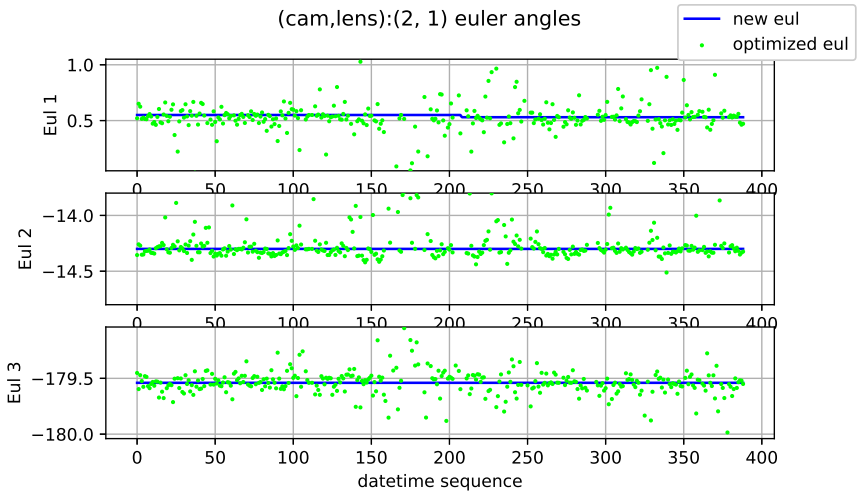


Figure 5.10: Second angle optimization run, aft cam

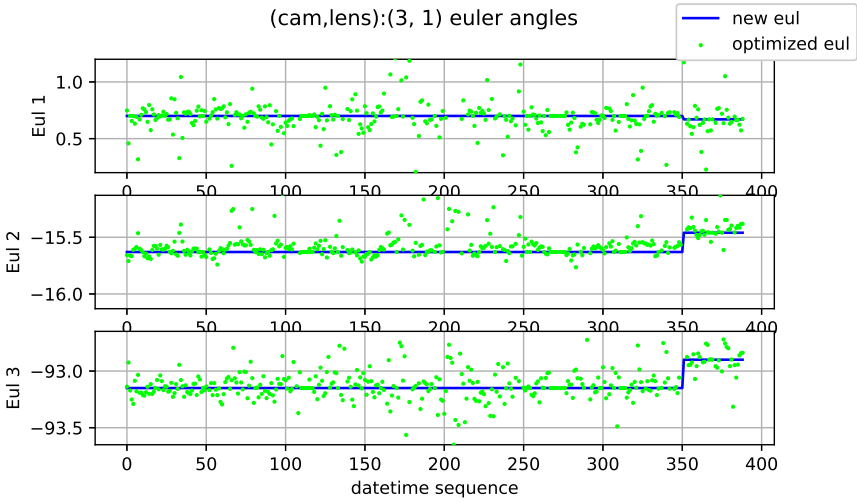


Figure 5.11: Second angle optimization run, port-side cam

Chapter 6

Semantic Segmentation

6.1 Creating an Image Segmentation Data Set

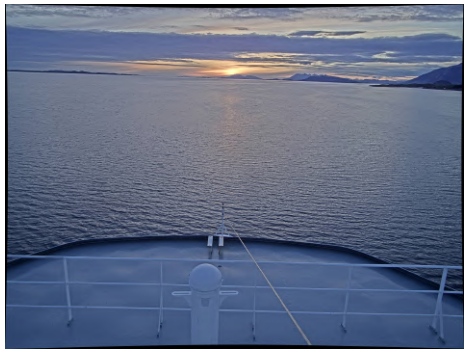
Training, validation and testing data for semantics segmentation is created by rendering terrain models using the rendering system described in chapter 4, then using the rendering as ground truth for the corresponding undistorted camera images. This yields a data set with three segmentation classes, each of which are encoded as a color. Since we have three classes, and RGB images have three color channels, each class is encoded in it's own color channel; red sky, green land and blue ocean. A list of datetimes for which the renderings are created is made by manually looking through the dataset selecting datetimes with good weather conditions and varying scenes. The updated camera angles calculated in chapter 5 are used for the rendering. As demonstrated in chapter 4 some of the generated 3D models have missing chunks or other problems. The models and renderings with obvious errors are discarded by sorting through the data manually. In particular bridges and smaller rock-formations in the water are often badly modelled, and thus provide somewhat unreliable training data even when the worst is discarded. 390 unique locations form the basis for the final dataset, for which images from 4 separate camera clusters are used. Some regions are missing the starboard camera images, as this cluster was disabled for a period of time. Only the middle cameras from each cluster are used, due to the side camera's inaccuracies as described in chapter 3.2.2.

Crop masks are created for each camera to crop out the visible parts of the ship, as well as the black areas from the image undistortion process. The front and aft images are much wider than high after the cropping, and are split into three square patches of 473x473, with some overlap. Starboard and port-side images are split into two 473x473 patches, also with some overlap. The data is stored as image patches with corresponding segmentation masks, still encoded as color images. This is illustrated in figure 6.1, where an image from the forward together together with a rendering is converted into 3 training samples consisting of 474x474 images with corresponding segmentation labels.

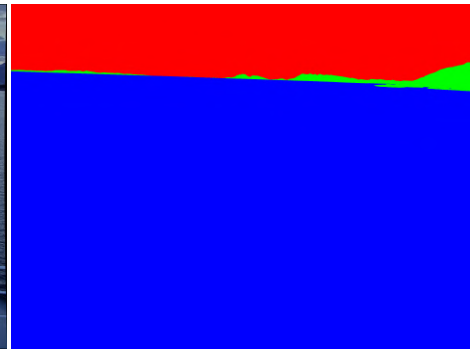
This yields a final dataset of ca. 3500 unique images total from those 390 geographical locations. The size of the dataset is later artificially expanded, as detailed in chapter

6.2.2. Upon creation the dataset is partitioned into training data (.8*.8=64%), validation data(.8*.2=16%) and testing data (.8=20%), stored in separate folders. As the names suggest the training data is meant to be used for training the net, the validation data is for validating the network's performance during training, and the final performance is tested using the test data. All image patches from a specific geographical region is strictly used as either training, validation or testing, to ensure ensure proper separation between training and testing data. It is possible that some images from separate regions see some of the same terrain, but this is assumed to be insignificant as the Norwegian coastline is very long, and the weather conditions would also likely be different causing the data to have little similarity. For the images in the dataset the majority of the pixels are labelled as ocean, followed by sky and at last land. The exact pixel label distribution is shown in table 6.1.

The data must be processed before it can be used by the segmentation network. The network uses arrays as input and output, not PNG image files. When the Data is to be used by the net, either for testing or training, the image and label-image is loaded using PIL, and converted to arrays using numpy. Furthermore the label-image is also decoded from colors to label probabilities. Since the labels are completely separated in their own RGB color channels the decoding is as as simple as dividing the array by 255, the max RGB value, to yield an array of binary masks for each class. The array of masks now represents a probability distribution between the different labels for each pixel. The decoding of the label-image is illustrated in figure 6.1.

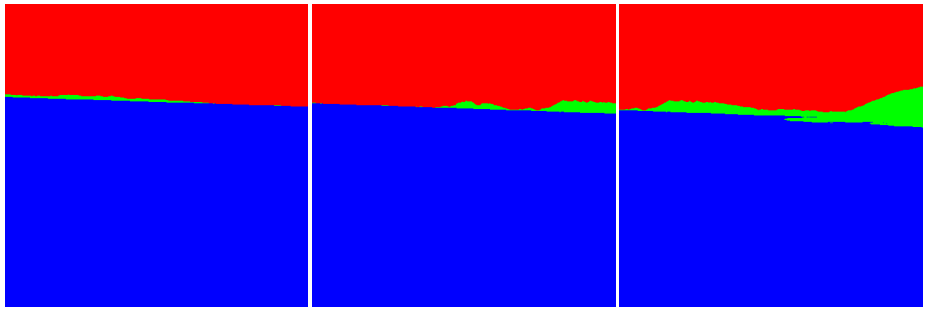


(a) camera image



(b) rendered ground truth segmentation label

(c) camera image patches



(d) segmentation label patches

Figure 6.1: converting camera image and rendering into multiple data samples

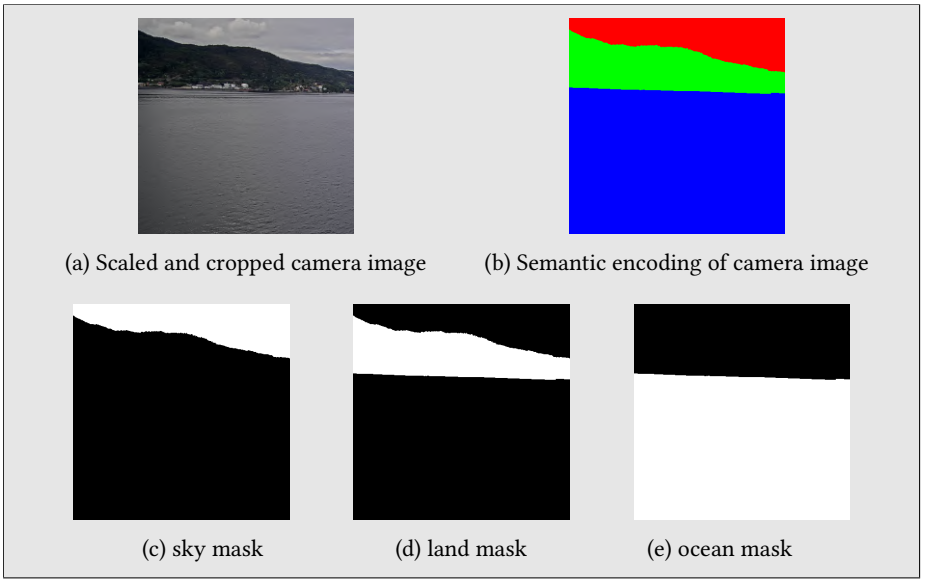


Figure 6.2: A data sample, where (a) is input and (c),(d),(e) are the binary masks for each class, decoded from (b)

Table 6.1: Total pixel label distribution in the dataset

Class ID	Pixel Distribution
0 (Sky)	26%
1 (Land)	16%
2 (Ocean)	58%

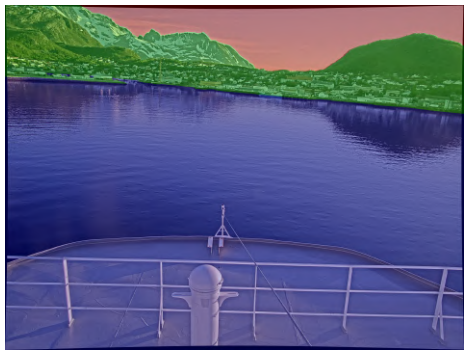
6.2 Semantic Segmentation Network

6.2.1 Architecture and Transfer Learning

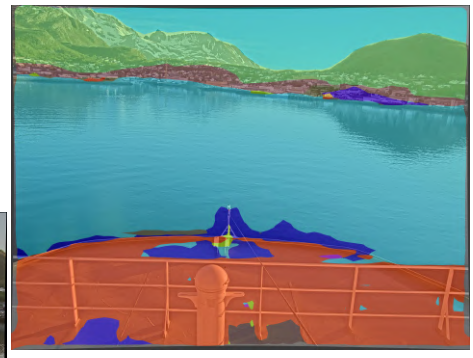
Pyramid Scene Parsing Network (PSPNet) is chosen as the architecture for the semantic segmentation network. PSPNet was presented in december 2016 and remains state of the art at the beginning of 2019. The idea of PSPNet is to process the input at multiple scales using an original pyramid-module, which can be attached to an existing feature extracting network, referred to as back-bone. The final layers also contain a dropout layers. It has gained popularity and existed long enough that it is well documented and implemented in several deep learning frameworks, including Keras. Keras with Tensorflow backend is chosen as the framework since it is good for quick prototyping and testing, widely used and well documented. Another architecture alternative that was heavily considered was ICNet, which is based upon PSPNet and is able to run much faster, at the cost of some accuracy. Since the system is not yet meant to run in real time, and is rather a research project to develop visual navigation methods, the most accurate net, PSPNet, is chosen. The original backbones of the network are ResNet-50 or ResNet-101. Using ResNet-50 backbone, a PSPNet with batch size 4 can be loaded onto a 12GB GTX1080 TI. Using ResNet-101 backbone only allows for batch size 1. A batch size of 1 would make the training drastically slower as a lower learning rate is needed to maintain stability. This could make the final performance worse as well. Therefore ResNet-50 is the chosen backbone. Furthermore the input and output size of the net is limited to 473x473, also out of memory concerns. These limitations when training PSPNet is a know problem in the semantic segmentation community, and it's apparent that the original author had access to some serious hardware. It is easy to find a PSPNet implementation with these backbones that are trained on some of the most popular semantic segmentation data sets, such as Cityscapes, ADE20K and VOC2012. I found weights for a PSPNet with backbone ResNet-50 trained on ADE20K, Cityscapes and VOC2012 respectively. The weights were converted to keras weights by Github user Karolmajek from the pytorch implementation by the original PSPNet author. The suitability for transfer learning is evaluated by using the weights to predict an image from this project's dataset, the creation of which is detailed in

chapter 6.1. The nets trained on Cityscapes and VOC2012 did not understand the scene at all, while the net trained on ADE20K gave some pretty decent segmentation results, as shown in figure 6.3. These results are very reasonable, when looking at what kind of images the datasets contain. ADE20K has many training samples resembling this project's dataset. Cityscapes' data does not much resemble this project's data, as it's images are from a car driving trough a city. Cityscapes does see a couple of trees, and this project's data does have some buildings, but that's as far as the overlap goes; the overall scenes are completely different. VOC2012 mainly has object labels such as bicycles, birds, boats, and bottles, so it's not surprising that it doesn't work here. ADE20K weights gave best performance due to some overlap in their dataset and that of this project. The segmentation is still rather inaccurate, and fails completely for land regions that are far away, so the weights cannot be used out of the box for this project. The weights are instead used as starting weights when training the dataset, drastically cutting the training time. ADE20K contains 120 classes, while the dataset used here only contains 3. Therefore the structure must also be modified somewhat. First the PSPNet structure is loaded, and the structure of the last layers is changed so as to only give three output channels (473x473x3). Then the name of the final layer is modified by adding the string "_custom". Now when the ADE20K weights are loaded with the by_name flag in keras, all layers are initialized with ADE20K weights, except the last layer which gets initialized with random weights.

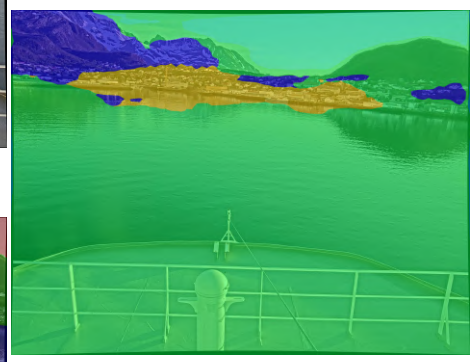
(a) camera image



(b) segmentation ground truth



(c) segmentation with ade20k weights



(d) segmentation with cityscapes weights

(e) segmentation with voc2012 weights

Figure 6.3: Comparing predictions using PSPNet with weights trained on various datasets

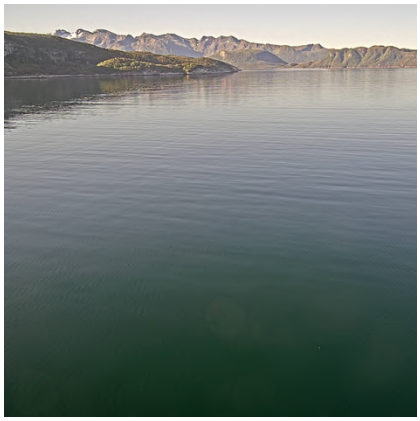
6.2.2 Training Setup

6.2.2.1 Initial Over-fit Test

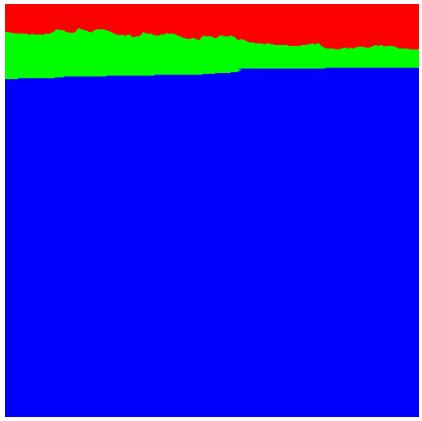
A simple test is performed by training the net on a single input-output pair. The input is just a black image, all 0 in all color channels. The desired output is all 0 in all output class channels, except for one arbitrary channel, which is all 1. This is just to test if the network is able to over-fit. If the network had not been able to over-fit, something must have gone very wrong. Here this test is passed, as the net reaches $loss \approx 0$ and $accuracy = 1$ in just a few batches of size 8, with learning rate 10^{-2}.

6.2.2.2 Random Data Augmentation

Data augmentation is a common technique in deep learning, and the original PSPNet paper places an emphasis on data-augmentation being important to avoid over-fitting and to generalize well. Random data augmentation artificially enriches the dataset by making random changes to each training sample before it gets passed through the network. The image is randomly rotated, zoomed, shifted, sheared and flipped horizontally. The augmentation values are specified as a range, and the augmentation values for each image are sampled uniformly from these ranges. When the transformed image does no longer covers the entire original (473,473) square, such as after being shifted, the missing values outside the augmented image boundary must be filled in. This is done by making a reflection of the image about the boundary, thus filling in the blank area with meaningful data. The exact same augmentation is performed on both the image and the semantic masks. An example of a pretty extreme augmentation is shown in figure 6.4. The exact values used for in the three final training runs are as follows. Zoom multiplier range 0.5-1.5, rotation range 25 deg, horizontal and vertical shift range 0.3, shear range 0.2 and horizontal flips set to true.

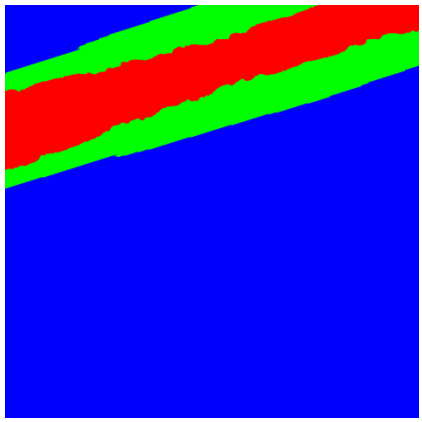


(a) original training input



(b) original training label

(c) augmented training input



(d) augmented training label

Figure 6.4: Example of random augmentation on training sample

6.2.2.3 Loss Function

The loss function is weighted categorical cross-entropy, a standard loss function for training semantic segmentation networks. Since the pixel label distribution is somewhat skewed, as per table 6.1 the classes are weighted by the inverse of their

representation in the data set. Land pixels are weighted heaviest, followed by sky and finally ocean, specific values seen in table 6.2. The main motivation for this however is that false negative for land could be very dangerous. It is obviously better for a ship to avoid some true ocean area it believes to be land, rather than sailing into some land area it believes to be ocean. We try to use the weighting to coax the net into predicting land if the image is ambiguous. Weighted categorical cross entropy is not implemented in keras, but someone made a forum post with a sketch for how such a function should look, which was used in this project. (https://forums.fast.ai/t/unbalanced-classes-in-image-segmentation/18289)

Table 6.2: Class weights for loss

Class ID	weight
0 (Sky)	4.3
1 (Land)	6.7
2 (Ocean)	1.6

6.2.2.4 Optimizer and Learning Rate

The optimizer-scheme is mini-batch stochastic gradient descent (SGD) with 0.9 nesterov momentum. An optimizer with adaptive learning rate was considered, but researchers at UC Berkeley found that adaptive methods generalize worse than SGD, even if the solutions show better training performance [51]. This comparative research did not test Adadelta, a robut extension of Adagrad [54], and so I was inclined to never the less do a training run with Adadelta. The advantage of an adaptive optimizer is that it requires minimal manual parameter tweaking. The results of training with Adadelta were much poorer than SGD, and are not looked further into. Suffice to say training was not as stable as SGD.
Many learning rates and learning rate schemes were tested for shorter training sequences, but variety in full-scale experiments is limited by the long training times; it takes more than 48 hours to do 500 training epochs with the current training data.

Three large large-scale training sequences are presented, one with a constant learning rate of 5e-4, and two sequences with step decay in the learning rate, starting at 1e-3 and 1e-4 respectively. The learning rate with decays by a $decay\_factor$ every $epoch\_wait$ epochs, as in eq. 6.1. In both runs the decay is 0.5 every 50th epoch. The relatively small batch size does not allow for as large a learning rate as bigger batches would. However since pre-trained weights are being used, a lower learning rate is not very problematic, as learning rate is often lowered towards the end of training anyways. The learning rates for these training runs were decided based on the baselines set in the many small-scale experiments. Furthermore it is in line with what is generally accepted as a range of normal learning rates for training deep neural networks. Graphs from the training process for each run, as well as a confusion matrices and total accuracy on the test data are shown in figures 6.5, 6.6 and 6.7. The comparison of each run is more briefly summarized in table 6.3, where it can be seen that the constant learning rate gave the lowest loss and highest pixel accuracy on the test data. due to the inaccuracies in what is used as ground truth in the dataset the loss and pixel accuracy is not a perfect measure of the real segmentation performance, though it gives a good indication. For both runs with step decay it is seems like the learning capabilities are killed rather quickly, as the loss and accuracy stops changing. This is does not happen in the constant learning rate run. Step decay was done to try to avoid over-fitting, but over-fitting turns out to not a problem when using constant learning rate. Had over-fitting become a problem this would have been seen by a ramping up in the validation loss. The images generated by running the test images through the final model checkpoint for the training runs is a qualitative measure of the model's performance. The final evaluation is not just done by comparing the numbers for loss and accuracy, since these can be misleading due to the imperfect dataset. Segmented test-set images are also compared, where it can be seen that the step decay runs in fact do perform worse, as they have more segmentation artifacts (obviously wrong segmentations), supporting the initial suspicion that learning rate is indeed killed off too early. Therefore the weights from the constant learning rate training run are the ones that will be used further on in this project, in the localization system, which is covered in chapter 7 and 8. When referring to the network in other sections of this

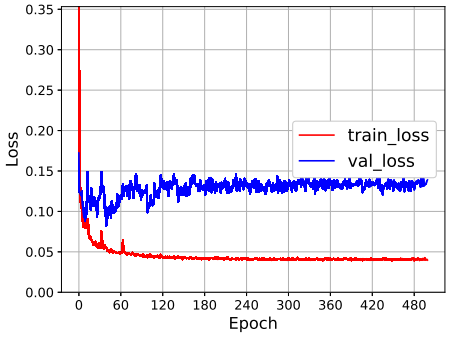
report, this means the network with weights from this training run. A qualitative evaluation of the network's performance on the test set is made at the end of this chapter, in section 6.2.3.

$$base\_lr(decay\_factor)^{floor(\frac{epoch}{epoch\_wait})} \tag{6.1}$$

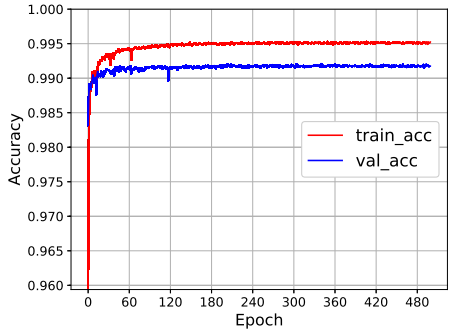
Table 6.3: Step decay vs constant learning rate

lrn_rate	test_loss	test_acc
1e-3 step	0.01577	99.460%
1e-4 step	0.01665	99.386%
5e-4 const	**0.01506**	**99.498**%

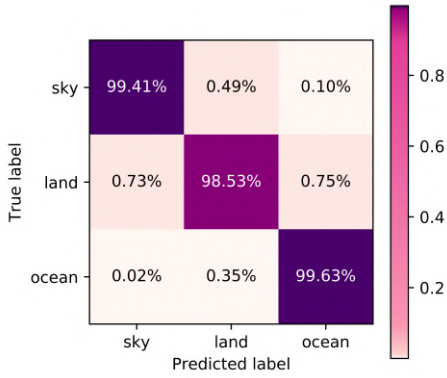
6.2.2.5 lrn rate 1e-3 with step decay



(a) Training and validation loss



(b) Training and validation accuracy



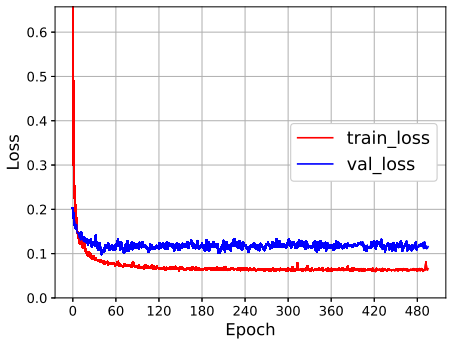
(c) confusion matrix with test data set

test loss	0.01577
test accuracy	99.460%

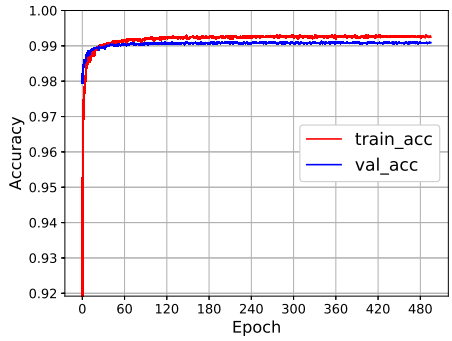
(d) loss and accuracy on test data set

Figure 6.5: training metrics and test metrics using model checkpoint epoch-500

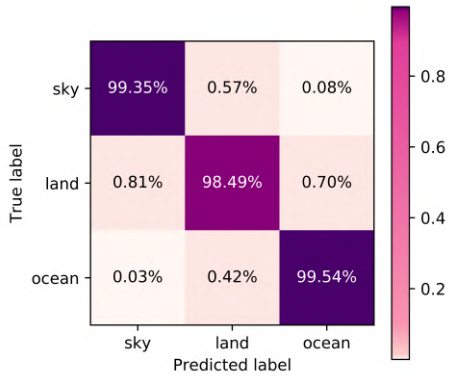
6.2.2.6 lrn rate 1e-4 with step decay



(a) Training and validation loss



(b) Training and validation accuracy



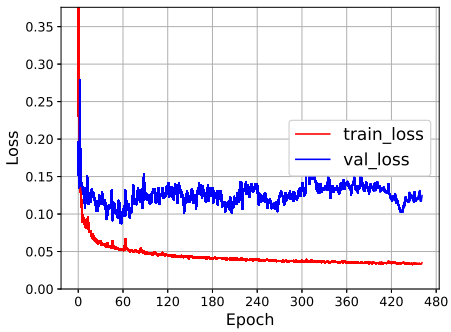
(c) confusion matrix with test data set

test loss	0.01665
test accuracy	99.386%

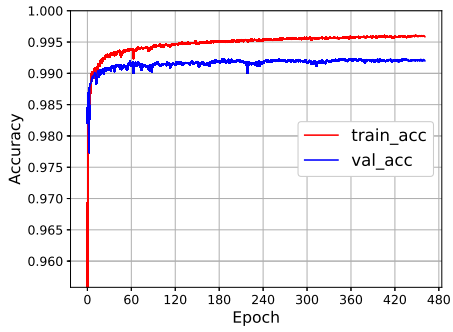
(d) loss and accuracy on test data set

Figure 6.6: training metrics and test metrics using model checkpoint epoch-500

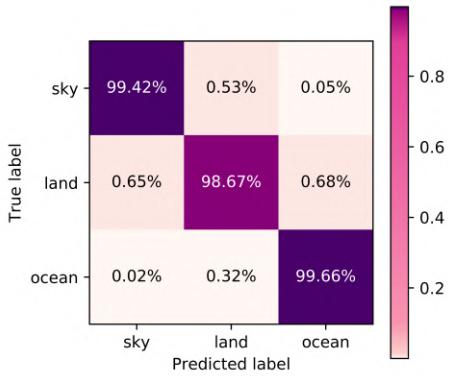
6.2.2.7 lrn rate 5e-4 constant



(a) Training and validation loss



(b) Training and validation accuracy



(c) confusion matrix with test data set

test loss	0.01506
test accuracy	99.498%

(d) loss and accuracy on test data set

Figure 6.7: training metrics and test metrics using the final model checkpoint epoch-462

6.2.3 Training Results

The network outputs very convincing segmentation labels for the images in the test-partition of the dataset. Some example segmentations performed by the network can be seen in fig 6.9, where the segmentation from the network is re-encoded as color and overlain with the input image. The network has generalized well from the examples

and often outputs segmentation labels that are more accurate than those stored in the dataset, illustrated in figure 6.8. The network predictions can however be bad for some scenarios, illustrated in 6.10. In particular, this includes images with bridges and small islands, which are often misrepresented in the dataset, as described in chapter 6.1, due to problems with the terrain models, as described in chapter 4.3. It became apparent that the network performed worse on images with darker lighting conditions, as well as images with extreme glare. Both of these scenarios are shown in figure 6.10. Some rare images have random artifacts for no apparent reason.

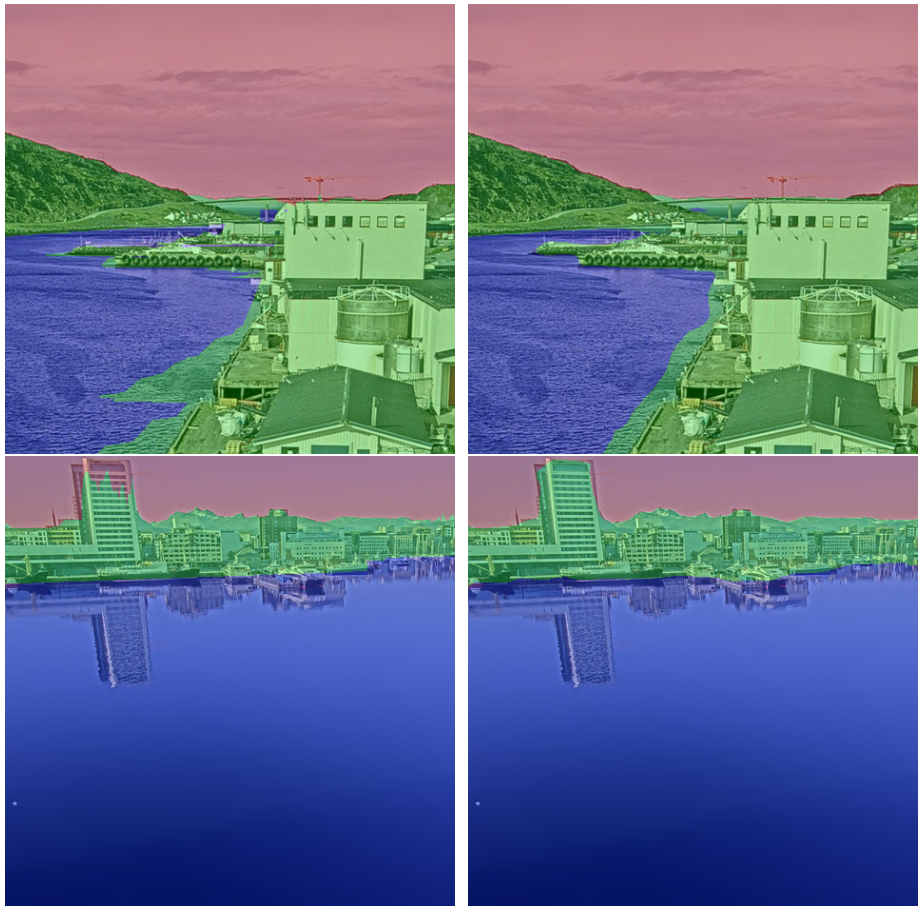


Figure 6.8: Dataset labels in left column, and network predicted labels in right column

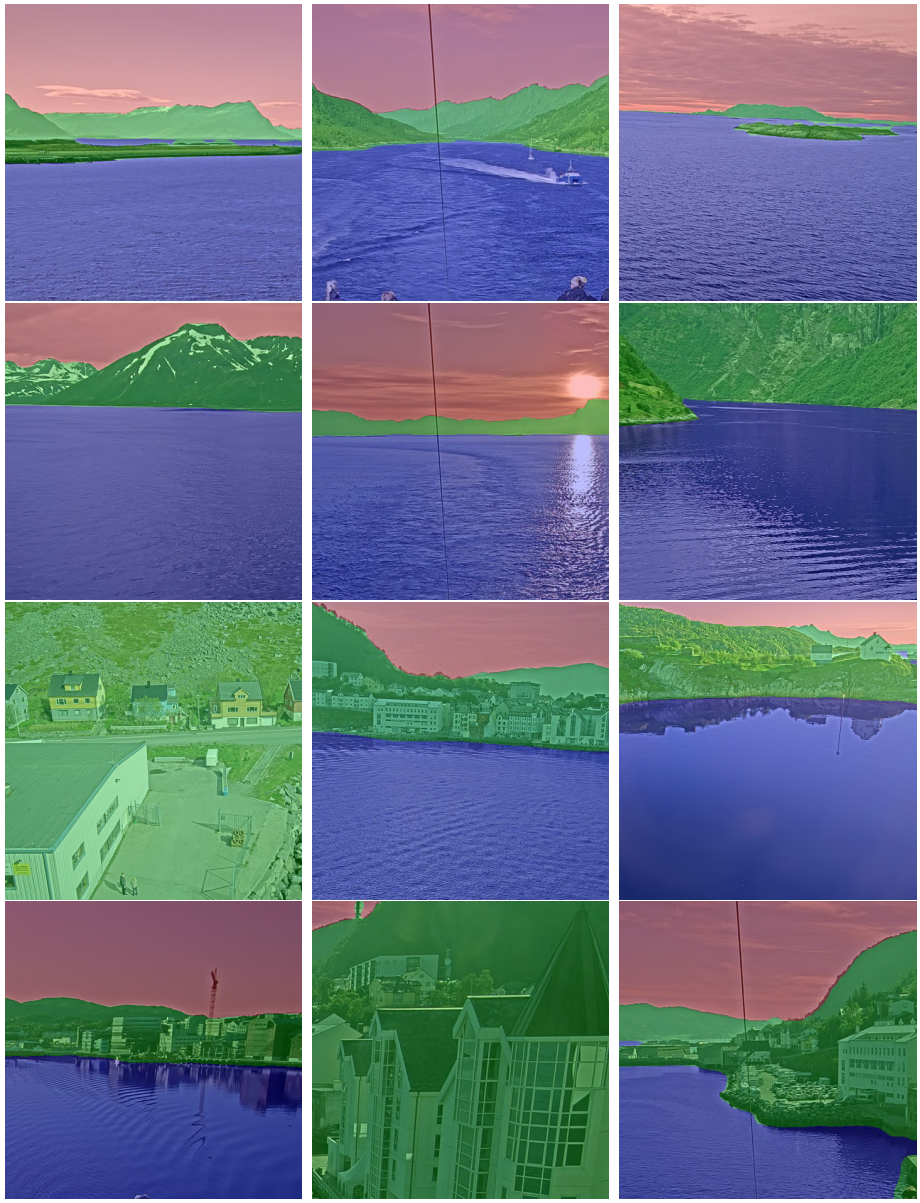
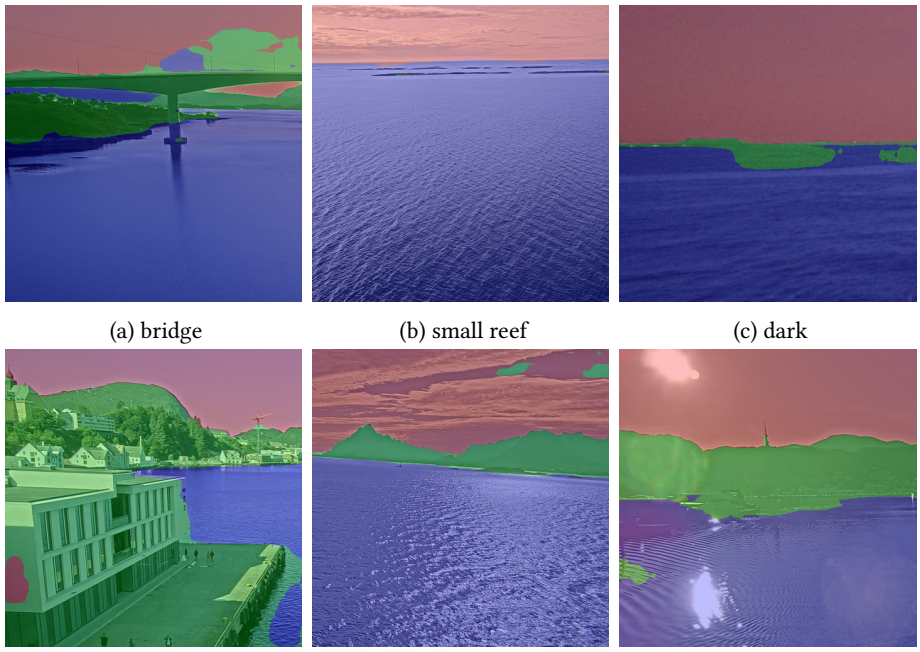


Figure 6.9: Some selected segmented test images



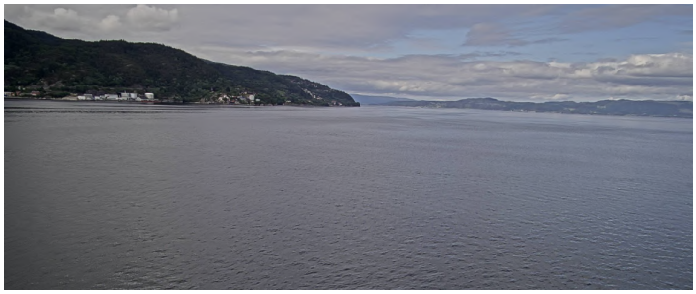
(a) bridge (b) small reef (c) dark

(d) artifact (e) artifact (f) glare

Figure 6.10: Bad segmented test images

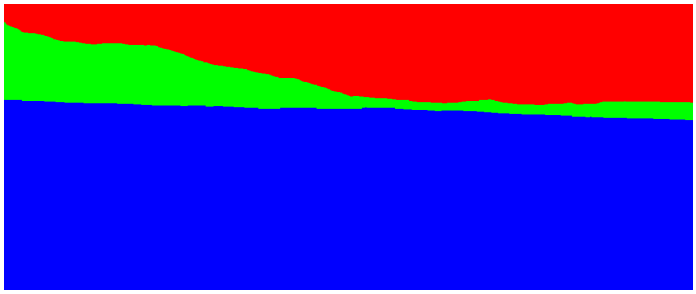
6.2.3.1 Sliding Prediction

When the net has given the class probabilities an image is constructed by associating each pixel color to the class with the highest probability. Multiple 473x473 patches can be extracted from an image and segmented individually, before being combined to a single prediction. For the overlapping sections the average probability of the patches is used.



(a) whole image

(b) overlapping image patches



(c) prediction for whole image

Figure 6.11: Sliding evaluation of image

Chapter 7

Localization Using Semantically Segmented Camera Images

7.1 Algorithm overview

The purpose of the algorithm is to calculate a 6DoF pose in a virtual replica of the real world that best aligns virtual camera images (ch. 4) with the semantic segmentation of the camera images (ch. 6), illustrated in a concept drawing in figure 7.1. This is done by iteratively optimizing the ship's virtual pose to minimize the point-to-line distance between virtual image points and corresponding camera image points. It is important to not that edges are only matched with other edges of the same semantic type, illustrated in figure 7.2. A high-level view of the localization system's flow is seen in figure 1.1. The pose estimation at each time step is initialized with the optimized pose from the last time-step, but the optimization is otherwise not constrained by earlier pose estimation values. No filtering techniques are used because the time limits forced restrictions on the scope of the project. A justification however is that this more clearly exposes the weaknesses of the core algorithm, the segmentation-to-model tracking, as each second in a localization sequence is vulnerable.

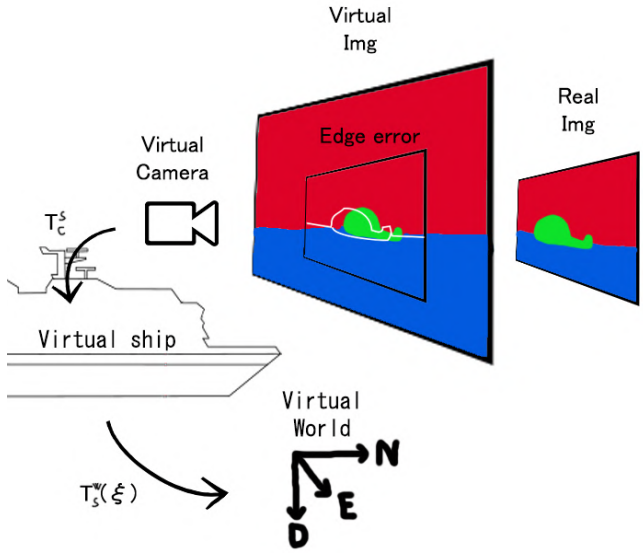


Figure 7.1: Illustration of the concept of comparing real images with a virtual world

Simplified high-level pseudo-code for the algorithm is shown in algorithm 1, which is explained in text in the following paragraphs. The algorithm starts with some initialization before progressing to a main loop. The real implementation of this algorithm is ca. 300 lines in python, not including the modules it uses, such as the rendering and the image segmentation. Each cam iteration also iterates over the different types of semantic edges, handling them in separate lists. Here they are summarized as one edge-list to make the pseudo-code brief enough to fit on one page, but the algorithm still conveys the core principle. The pseudo-code is high level, and references functions that are described in more detail in other sections of this chapter, such as edge extraction and processing, the projection and backprojection, the jacobian calculation and the optimal step calculation.

In chapter 5 projective ICP was used to align edges in the model rendering with canny edges in the real image to estimate the camera orientation. Now the edges between different semantic regions in a rendering is aligned against semantic segmentations of multiple camera images to estimate the full ship pose. The semantic labels of the edges are respected, distinguishing between sky-land, sky-sea and land-sea transitions. These edges are aligned against edges with corresponding labels in the segmentation image, not just using any detected edges as was done in chapter 5. Furthermore the error measure that is being minimized here is a point-to-line distance. The previously used point-to-point error can cause extremely slow convergence since when a line is aligned with a another line the optimization is penalized for moving the lines parallel to each other, since the distance being measured is between specific points on the line. The point-to-line error however does not penalize such movement, allowing faster convergence in regions with sparsely featured contours. The point-to-line distance for two points requires the definition of a line through one of the points. This is defined with a normal vector to that line. Here the line is defined for the edges in the semantic segmentation image, since the edges in the rendering will be moved around, and would require recalculation of the normal vectors. The normal vectors are kept spatially consistent by smoothing the edges the normal vectors are based on. This ensures that the edges that are jagged due to the pixelation do not yield normal vectors with alternating directions. The calculation of edge-points and normals is detailed

in 7.2.1. The render-edges are back-projected to 3D coordinates using the rendered depth-map and each camera's intrinsic matrix. Then the iterative process of aligning the points is started, the main loop.
For each ICP-step, in stead of re-rendering the edges, the initially calculated and back-projected render-edges are instead transformed using the pose-change calculated thus far, which is initialized as eye(4) and updated at the end of each iteration. This is explained in ch. 7.2.2, but is summarized by that reprojecting existing points in stead of rendering new ones saves a lot of time. We discard the points that are projected outside of the area equivalent to the crop-area for each camera, removing the ship structure and the undistortion artifacts described in ch. 3.2.2. The remaining reprojected render-edges' nearest neighbours among the camera segmentation edges are calculated for each camera and for each edge-type. This is sped up by representing the camera segmentation edges as KD-Trees using the python package scipy. The image segmentation edges are the same each iteration, so the trees need only be created once. Point correspondences that are more than 10 pixels away from each other are discarded. Now that the iteration's point correspondences are decided the residual errors (point-to-line distances) and jacobians for these errors for each camera can be calculated. The error functions being minimized for each camera are all expressed in the ship frame. Since the functions are in the same space they can be jointly optimized by just concatenating all the camera's residual errors and jacobians. Then, as described in 7.2.3, analytical gauss-newton is used to find the optimal change in ship pose that jointly minimizes the residual errors. This optimal change in pose is parameterized on-manifold, using lie algebra, and once calculated it is converted to an SE4 rigid transformation matrix, using 7.7 with openCV's rodrigues formula. This is more stable over multiple multiplication operations than the approximation 7.6, which causes the determinant of the rotation matrix to explode when multiplied with itself some tens of times. The optimal step is used to update the overall pose-change thus far, which is where the initial render-points will be transformed to next iteration. When the loop is done the calculated pose change is combined with the original ship pose T_w^s to express the new improved ship pose.

A type of photometric error minimization, sum of squared differences between model image and segmentation was considered, and the jacobian for this problem was even formulated. But it was painstakingly slow to calculate this jacobian, since it uses all pixels of two large 2D images, in addition to having to backproject all the pixels in one image, even those whose movement doesn't affect the error. Using ICP there are much fewer points being used in the calculation, just the contours, whose movement actually affect the error. It could be made faster with GPU-programming framework like CUDA, since jacobians relating to each point-correspondence are independent from each other. Furthermore though, using simple SSD there is no convergence for objects that do not already overlap, which is not a problem with ICP it minimizes the distance to the nearest point. This can be remedied by converting the images to truncated signed distance maps (distance to contours), but this does not seem too different from straight contour tracking with ICP, just a lot more pre-processing. A region based method would however give more weight to large regions, not just regions with a long contour, which could be argued gives a more intuitive or fair matching criteria.

Algorithm 1 Refine ship pose from segmented camera images

Require: $shipPose, camSegs, camPoses, intrinsics$

  for all $cam \leftarrow cams$ **do** # Also loops skyland, skysea, landsea edges
    $segEdges[cam], segNormals[cam] \leftarrow get\_edges\_and\_normals(camSegs[cam])$
    $segEdgesTree[cam] \leftarrow makeKDTree(segEdges[cam])$
    $img[cam], depth[cam] \leftarrow render(shipPose, camPoses[cam], intrinsics[cam])$
    $edges \leftarrow get\_edges(img[cam])$
    $edges3DInit[cam] \leftarrow backproject(edges, depth, intrinsics[cam])$
  end for
  $T \leftarrow eye(4)$
  for $i := 0$ **to** $n\_iterations$ **do**
    $resErrList = []$
    $jacobiansList = []$
    for all $cam \leftarrow cams$ **do** # Also loops skyland, skysea, landsea edges
      $T_s^w \leftarrow SE4(shipPose)$
      $T_m^s \leftarrow SE4(camPose[cam])$
      $T_c^m \leftarrow SE4(internal.viewAngles)$
      $T_w^c \leftarrow (T_c^m)^{-1}(T_m^s)^{-1}(T_s^w)^{-1}$
      $edges3D \leftarrow transform(edges3DInit[cam], (T_c^w)^{-1} T T_c^w)$
      $edges \leftarrow project(edges3D, intrinsics[cam])$
      $idx \leftarrow segEdgesTree[cam].query(edges, internal.outlier\_thresh)$
      $targets \leftarrow segEdges[cam][idx]$
      $normals \leftarrow segNormals[cam][idx]$
      $resErrList.append(normals^T(targets - edges))$
      $jacobianList.append(jacobian(normals, edges3D, intrinsics[cam], T_c^w))$
    end for
    $\Delta\xi \leftarrow get\_step(jacobianList, resErrList)$
    $T \leftarrow exp(\Delta\xi)T$
  end for
  return $T_w^s T$

7.2 Algorithm details

7.2.1 Calculating edges and normal vectors

Image contour points are used in the ICP-process, and so these edges are extracted from the render-image and the semantically segmented camera image. The alignment process respects the semantic labels of the regions associated with the contours. This means that sky-land, sky-sea and land-sea edges are only matched within their semantic groups, and must therefore be calculated and handled individually. These different semantic edges are illustrated in figure 7.2. With three classes there are three types of edges; sky-land, sky-sea and land-sea.
The shared edges for two classes in the rendered image is extracted by calculating by using the sobel operator to create a mask of edges between the classes. Once the semantic edge masks are created, a simple call to np.argwhere with each mask gives a list of pixel-coordinates for each edge pixel. This is repeated for all three class combinations. The process for the camera segmentation image is similar, but includes some extra steps. The process starts with using a gaussian blur on each class mask before calculating each pixel's normal vector using sobel. The sobel derivative used to create normal values does not yield normalized vectors. This fact is used in the thresholding process to find the strongest edges. Because we used a gaussian blur before the edge detection, the detected edges have had their normals smoothed. Slightly smoothed normals better represent the contours as sobel is only a 3x3 filter, making it sensitive to changes in just a single pixel.

(a) rendered labels (b) sky-land edges (c) sky-sea edges (d) land-sea edges

Figure 7.2: Semantic edge extraction for tiny image

Algorithm 2 Extract transition edge pixels between two semantic masks

Require: $mask1, mask2$

$gx \leftarrow convolve(img1, sobelX)$

$gy \leftarrow convolve(img1, sobelY)$

$edgeMask \leftarrow (gx \textbf{or} gy) \textbf{and} img2$ # logical, not bitwise

return $np.argwehere(edgeMask)$

Algorithm 3 Extract transition edge pixels and smoothed normal vectors between two semantic masks

Require: $mask1, mask2$

$img1 \leftarrow gaussian_blur(img1)$

$gx \leftarrow convolve(img1, sobelX)$

$gy \leftarrow convolve(img1, sobelY)$

$edgeMask \leftarrow (gx \textbf{or} gy) \textbf{and} img2$ # logical, not bitwise

$normals \leftarrow [gx[edgeMask], gy[edgeMask]]$

$normals\_norm \leftarrow norm(normals)$

$edges \leftarrow numpy.argwhere(edgeMask)[normals\_norm > threshold]$

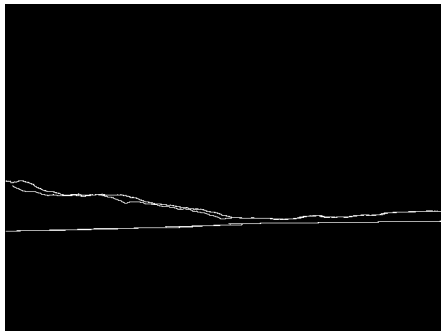
$normals \leftarrow normals[normals\_norm > threshold]$

return $edges, normals$

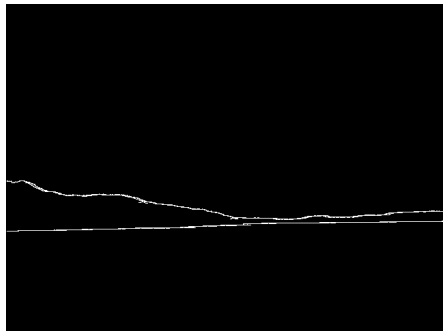
7.2.2 Justification for the Edge Reprojection Scheme

ICP requires multiple steps before converging, due to it's approximate iterative nature. One of the most costly parts of the algorithm is the calculation of edges, and so to save time edges are reused by transforming them to the new pose at the start of every iteration instead of rendering new edges at the new pose. This assumes that the points representing the contours remain approximately the same as you move around. This is of course not true when moving far, and so the edges must eventually be re-rendered. This is illustrated in figure 7.3, where the estimate rendering starts at pose 0, 150m from the true position. The first optimization run, using the transformed edges rendered at pose 0, converges with an error of ca. 40m, at pose 1. Even though

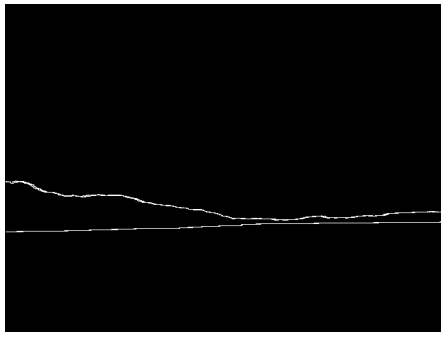
the edge-images (b) and (c) look similar at pose 1, when the edges are re-rendered this corrects the edge-inaccuracies and using transformations of these new edges allows the second pass to continue converging towards ground truth, converging ca. 4m from the true position, at pose 2. ICP is after all inherently an iterative algorithm, so as long as approximating contours using transformed edges reduces the pose error, that is all that's required. This scheme manages to drastically cut down the number of rendering and edge extraction procedures and thus saves a lot of time. In the current implementation two passes of ICP with 15 iterations each would take 10 times as long if re-rendering at each internal iteration of the ICP. With re-rendering convergence requires slightly fewer total iterations due to accurate edges at each step, but not few enough to compensate for the rendering overhead. Keep in mind that the rendering, which uses the GPU, is in itself very fast, it is the edge extraction and pre-processing which is slow, as it is implemented in python and just uses the CPU.



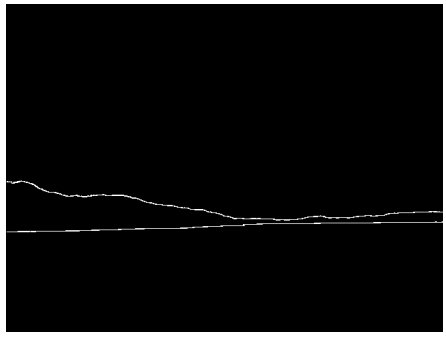
(a) camera edges and initial render edges at pose 0



(b) camera edges and reprojected render edges at pose 1



(c) camera edges and initial render edges at pose 1



(d) camera edges and reprojected render edges at pose 2

Figure 7.3: Iterative pose estimation, where both camera edges and render edges are shown. The camera edges are static

7.2.3 Analytical Gauss-Newton

This section describes the error being minimized at each step of the ICP algortihm, and how the optimal step that minimizes this error is calculated.
The error, $E(\xi)$,is a sum of all point correspondences' squared residual errors, $r_i(\xi)$, where i denotes a specific correspondence. Two corresponding points are from the segmented camera image, x_i', and the rendered image, x_i, respectively. The residual

error for two corresponding points is a point-to-line distance between a line through the 2D camerapoint defined by a normal vector n_i, and the 2D renderpoint that gets backprojected to 3D, rigidly transformed through a change in the ship pose, and then re-projected back into into 2D. projection and back projection is defined in 7.8 and 7.13 respectively. This rigid transformation is what gets optimized to minimize the error. The rigid transformation is parameterized on-manifold with lie algebra, denoted ξ, eq. 7.3, and can be converted to an SE4 rigid transformation matrix, eq 7.4. T_w^c and T_c^w are used to change the frame the points are referenced to between the virtual camera frame, c, and the virtual world-frame, w, more on this in 4.3. The jacobian for the residual errors, 7.17, is calculated analytically, and used to calculate a an optimal gauss-newton step for the rigid transformation in the ship frame that minimizes the error for the current point correspondences, eq. 7.18. Since the error corresponding to each camera is a function of the same variable, a rigid transform in the ship frame, the errors can be jointly optimized by simply concatenating all their point correspondences' residual errors and jacobians, and using these two larger arrays to calculate a single gauss-newton step that accounts for all camera alignments simultaneously. This assumes that dataset's stored values for the camera's poses relative to the ship are perfect. As was described in chapter 5 the angles are not perfectly correct, but as chapter 8 shows the localization still works well.

$$\boldsymbol{\xi}^* = arg\min_{\boldsymbol{\xi}} E(\boldsymbol{\xi}) = arg\min_{\boldsymbol{\xi}} \sum_i (\mathbf{n}_i^T(\mathbf{x}_i' - proj(T_w^c exp(\hat{\boldsymbol{\xi}}) T_c^w back(\mathbf{x}_i, d_i))))^2 \tag{7.1}$$

$$\begin{aligned} r_i(\boldsymbol{\xi}) &= \mathbf{n}_i^T(\mathbf{x}_i' - proj(T_w^c exp(\hat{\boldsymbol{\xi}}) T_c^w back(\mathbf{x}_i, d_i))) \\ &= \mathbf{n}_i^T(\mathbf{x}_i' - \pi(\mathbf{K}(T_w^c exp(\hat{\boldsymbol{\xi}}) T_c^w \tilde{\mathbf{z}}_i)_{3x1})) \end{aligned} \tag{7.2}$$

$$\xi = \begin{bmatrix} \mathbf{w} \\ \mathbf{v} \end{bmatrix} = [\omega_1, \omega_2, \omega_3, v_1, v_2, v_3]^T \in \mathbb{R}^6 \tag{7.3}$$

$$\hat{\xi} = \begin{bmatrix} \hat{\mathbf{w}} & \mathbf{v} \\ \mathbf{0} & 0 \end{bmatrix} = \begin{bmatrix} 0 & -\omega_3 & \omega_2 & v_1 \\ \omega_3 & 0 & -\omega_1 & v_2 \\ -\omega_2 & \omega_1 & 0 & v_3 \\ 0 & 0 & 0 & 0 \end{bmatrix} \in \mathfrak{se}(3) \tag{7.4}$$

$$\Delta T = exp(\hat{\xi}) \in \mathbb{SE}(3) \tag{7.5}$$

$$exp(\hat{\xi}) \approx \mathbb{I}_{4x4} + \hat{\xi} \tag{7.6}$$

$$\Delta T = \begin{bmatrix} rodriguez(\mathbf{w}) & \mathbf{v} \\ \mathbf{0} & 0 \end{bmatrix} \tag{7.7}$$

Projection of a 3D point Z in the camera frame into pixel coordinates, using camera intrinsic matrix K.

$$proj(Z) = \pi(KZ) \tag{7.8}$$

$$\pi(Z) = \begin{bmatrix} \frac{z_1}{z_3} \\ \frac{z_2}{z_3} \end{bmatrix} \tag{7.9}$$

$$\frac{\partial \pi(Z)}{\partial Z} = \begin{bmatrix} \frac{1}{z_3} & 0 & -\frac{z_1}{(z_3)^2} \\ 0 & \frac{1}{z_3} & -\frac{z_2}{(z_3)^2} \end{bmatrix} \tag{7.10}$$

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{7.11}$$

$$K^{-1} = \begin{bmatrix} \frac{1}{f_x} & 0 & -\frac{c_x}{f_x} \\ 0 & \frac{1}{f_y} & -\frac{c_y}{f_y} \\ 0 & 0 & 1 \end{bmatrix} \tag{7.12}$$

Back-projection of a 2D point X with depth $D_m(X)$ uses the inverste of the camera matrix K.

$$back(X, D_m(X)) = D_m(X)K^{-1} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = \begin{bmatrix} D_m(X)\frac{x_1 - c_x}{f_x} \\ D_m(X)\frac{x_2 - c_y}{f_y} \\ D_m(X) \end{bmatrix} \tag{7.13}$$

The definition of the jacobian is expressed as follows.

$$\begin{aligned} \mathbf{J}i = \left.\frac{\partial r_i(\xi)}{\partial \xi}\right|_{\xi=\mathbf{0}} &= -\mathbf{n}_i^T \frac{\partial \pi}{\partial \mathbf{K}(exp(\hat{\xi})\tilde{\mathbf{z}}_i)_{3x1}} \mathbf{K} \left.\frac{\partial (exp(\hat{\xi})\tilde{\mathbf{z}}_i)_{3x1}}{\partial \xi}\right|_{\xi=\mathbf{0}} \\ &= -\mathbf{n}_i^T \frac{\partial \pi}{\partial \mathbf{K}((\mathbb{I}_{4x4} + \hat{\xi})\tilde{\mathbf{z}}_i)_{3x1}} \mathbf{K} \left.\frac{\partial ((\mathbb{I}_{4x4} + \hat{\xi})\tilde{\mathbf{z}}_i)_{3x1}}{\partial \xi}\right|_{\xi=\mathbf{0}} \\ &= -\mathbf{n}_i^T \frac{\partial \pi}{\partial \mathbf{K}\mathbf{z}_i} \mathbf{K} \frac{\partial ((\mathbb{I}_{4x4} + \hat{\xi})\tilde{\mathbf{z}}_i)_{3x1}}{\partial \xi} \end{aligned} \tag{7.14}$$

The equations for the jacobian-factors are solved individually

$$\begin{aligned} \frac{\partial \pi}{\partial \mathbf{K}\mathbf{z}_i} = \frac{\partial \pi}{\partial \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} z_{i,1} \\ z_{i,2} \\ z_{i,3} \end{bmatrix}} &= \frac{\partial \pi}{\partial \begin{bmatrix} f_x z_{i,1} + c_x z_{i,3} \\ f_y z_{i,2} + c_x z_{i,3} \\ z_{i,3} \end{bmatrix}} \\ = \begin{bmatrix} \frac{1}{z_{i,3}} & 0 & -\frac{f_x z_{i,1} + c_x z_{i,3}}{(z_{i,3})^2} \\ 0 & \frac{1}{z_{i,3}} & -\frac{f_y z_{i,2} + c_x z_{i,3}}{(z_{i,3})^2} \end{bmatrix} &= \begin{bmatrix} \frac{1}{d_i} & 0 & -\frac{x_{i,1}}{d_i} \\ 0 & \frac{1}{d_i} & -\frac{x_{i,2}}{d_i} \end{bmatrix} \end{aligned} \tag{7.15}$$

$$
\begin{aligned}
\frac{\partial(T_w^c(\mathbb{I}_{4x4}+\hat{\xi})T_c^w\tilde{\boldsymbol{z}}_i)_{3x1}}{\partial\boldsymbol{\xi}} &= \left(T_w^c\frac{\partial\begin{bmatrix}1 & -\omega_3 & \omega_2 & v_1\\ \omega_3 & 1 & -\omega_1 & v_2\\ -\omega_2 & \omega_1 & 1 & v_3\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}z_{i,1}^{(w)}\\ z_{i,2}^{(w)}\\ z_{i,3}^{(w)}\\ 1\end{bmatrix}}{\partial\boldsymbol{\xi}}\right)_{3x1}\\
&= \left(T_w^c\frac{\partial\begin{bmatrix}z_{i,1}^{(w)}-\omega_3 z_{i,2}^{(w)}+\omega_2 z_{i,3}^{(w)}+v_1\\ \omega_3 z_{i,1}^{(w)}+z_{i,2}^{(w)}-\omega_1 z_{i,3}^{(w)}+v_2\\ -\omega_2 z_{i,1}^{(w)}+\omega_1 z_{i,2}^{(w)}+z_{i,3}^{(w)}+v_3\\ 1\end{bmatrix}}{\partial\begin{bmatrix}\omega_1 & \omega_2 & \omega_3 & v_1 & v_1 & v_3\end{bmatrix}^T}\right)_{3x1}\\
&= \left(T_w^c\begin{bmatrix}0 & z_{i,3}^{(w)} & -z_{i,2}^{(w)} & 1 & 0 & 0\\ -z_{i,3}^{(w)} & 0 & z_{i,1}^{(w)} & 0 & 1 & 0\\ z_{i,2}^{(w)} & -z_{i,1}^{(w)} & 0 & 0 & 0 & 1\end{bmatrix}\right)_{3x1}
\end{aligned}
\tag{7.16}
$$

The individual factors of the jacobian are combined to fully express the jacobian.

$$
\begin{aligned}
\mathbf{J}_i = &- \begin{bmatrix} n_{i,1} & n_{i,2} \end{bmatrix} \begin{bmatrix} \frac{1}{z_{i,3}} & 0 & -\frac{f_x z_{i,1} + c_x z_{i,3}}{(z_{i,3})^2} \\ 0 & \frac{1}{z_{i,3}} & -\frac{f_y z_{i,2} + c_x z_{i,3}}{(z_{i,3})^2} \end{bmatrix} \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} T_w^c \\
&\cdot \begin{bmatrix} 0 & z_{i,3}^{(w)} & -z_{i,2}^{(w)} & 1 & 0 & 0 \\ -z_{i,3}^{(w)} & 0 & z_{i,1}^{(w)} & 0 & 1 & 0 \\ z_{i,2}^{(w)} & -z_{i,1}^{(w)} & 0 & 0 & 0 & 1 \end{bmatrix} \\
= &- \begin{bmatrix} n_{i,1} & n_{i,2} \end{bmatrix} \begin{bmatrix} \frac{f_x}{z_{i,3}} & 0 & -\frac{f_x z_{i,1}}{(z_{i,3})^2} \\ 0 & \frac{f_y}{z_{i,3}} & -\frac{f_y z_{i,2}}{(z_{i,3})^2} \end{bmatrix} T_w^c \begin{bmatrix} 0 & z_{i,3}^{(w)} & -z_{i,2}^{(w)} & 1 & 0 & 0 \\ -z_{i,3}^{(w)} & 0 & z_{i,1}^{(w)} & 0 & 1 & 0 \\ z_{i,2}^{(w)} & -z_{i,1}^{(w)} & 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned} \tag{7.17}
$$

To minimize the error, 7.1, the jacobians and residual errors for each point correspondence are all concatenated and used to jointly calculate the optimal gauss-newton step in the parameterized rigid transformation.

$$
\xi* = -(\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T r(\mathbf{0}) \tag{7.18}
$$

Chapter 8

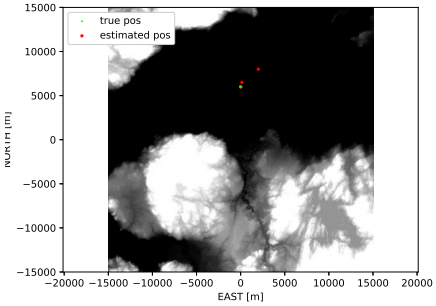
Localization Experiments and Results

8.1 Artificial Localization: Trondheim Fjord

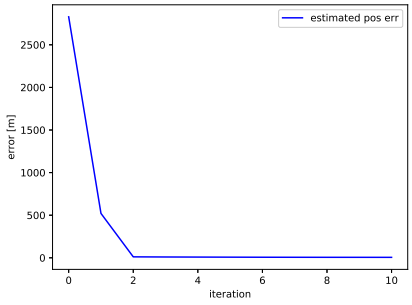
For this test the localization uses simulated segmentation images by rendering at ground truth. This way errors in the 3D model, camera intrinsic errors, camera angle errors and errors in the segmentation are non-existent, and we can learn the upper bounds for accuracy and convergence speed. Three test are done, one where the ship is static far away from land, one where it is static close to land, and one the ship is moving along a path. In the static tests the ship's estimated attitude and position is initialized relatively far away from the true pose, and int he moving test it is initialized with the true pose in the first step. The plots and graphs related to position are the most interesting, and so the estimated attitude angle plots are omitted. Suffice to say the attitude converges at approximately the same rate as the position. The values are however mentioned if they are interesting. The localization process for each time-step consist of two iterations of rendering the estimate and doing ICP with reprojected points for 15 internal iterations. It is done this way because when doing pose estimation with real data, we want to make sure that the estimate for each time step converges, therefore two passes are needed to mostly negate the effect of the reprojection scheme, as discussed in section 7.2.2.

8.1.1 Static Far from Land

The true pose of the virtual ship is set to be somewhere the middle of the Trondheim Fjord looking straight north with no pitch or roll. The estimate is initialized with an offset of 2km in both the north and east directions. The roll pitch and yaw initialized at a +1 deg offset. The localization converges after around 5 iterations to a $4m$ absolute error in the position, and less than 0.005 deg absolute error for each roll, pitch and yaw angle.



(a) position at start of each iteration

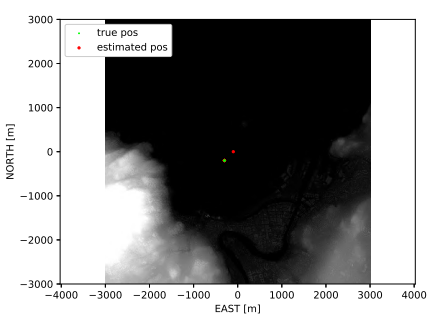


(b) absolute position error at start of each iteration

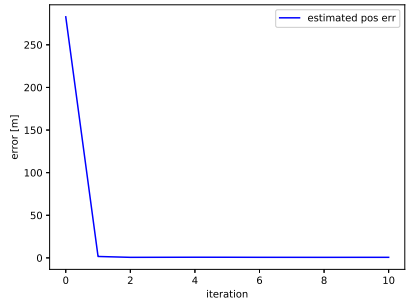
Figure 8.1: position and error plot for artificial static localization far from land

8.1.2 Static Close to Land

The true pose of the virtual ship is set to be close the shore at the center of the map, looking straight north with no pitch or roll. The estimate is initialized with an offset of 200m in both the north and east directions. The roll pitch and yaw initialized at a +1 deg offset. The localization converges after just 2 iterations to a $0.6m$ absolute error in the position, and less than 0.004 deg absolute error for each roll, pitch and yaw angle. Note that the position plot is 5x more zoomed in than for the test far from land.



(a) position at start of each iteration

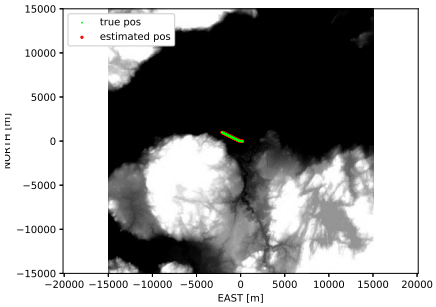


(b) absolute position error at start of each iteration

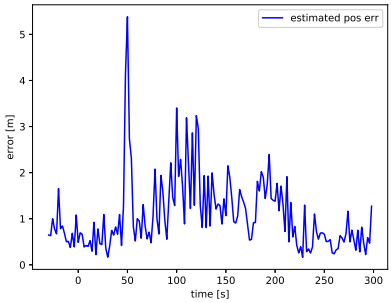
Figure 8.2: position and error plot for artificial static localization close to land

8.1.3 Moving

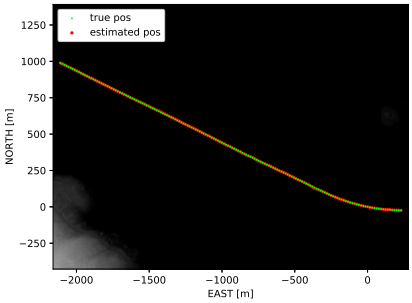
This test uses pose data from the real dataset as ground truth. The speed of the ship is about $15m/s$ in this sequence. The pose estimation achieves sub-meter accuracy for most of the sequence, over a kilometer from the closest point of land.



(a) position at each time step



(b) absolute position error at each time step



(c) position at each time step, zoomed in

Figure 8.3: position and error plot for the sequence

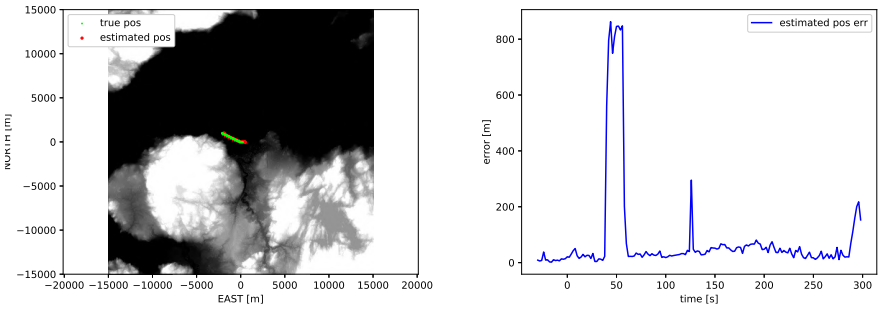
8.2 Real Localization: Trondheim Fjord Sequence

These localization sequences uses the real data from the dataset, and the segmentation network is used on real camera images at each time step. This is the same sequence as the artificial test. First each camera is used individually, and finally all cameras are used simultaneously.

8.2.1 Single Camera Tracking

8.2.1.1 Front

https://drive.google.com/drive/folders/1XQlq7MKUCJPH9exrF3fNQFMk85A6cExp
The localization fails and gets stuck in a local minima, but recovers after around 10 seconds.

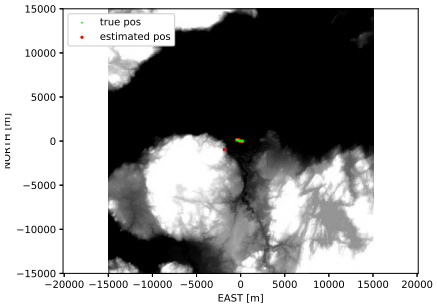


(a) True path compared with estimated path (b) absolute error in estimated path

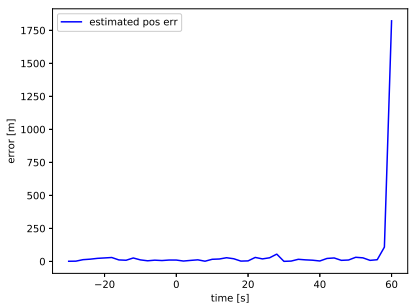
Figure 8.4: Tracking using front camera

8.2.1.2 Starboard

https://drive.google.com/drive/folders/1fPv4wAnEo-6sFbkE810Vb8lrkUBGAiN5
Here the localization failed hard, and so no video was created. The estimated position inside the mountain gave no visible points in the rendering, which the system cannot recover from.



(a) True path compared with estimated path

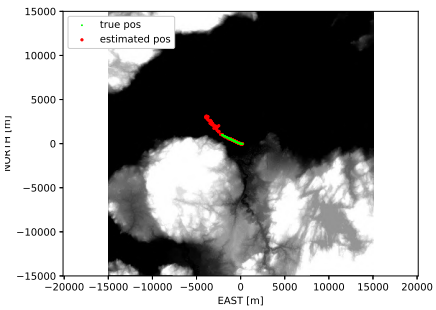


(b) absolute error in estimated path

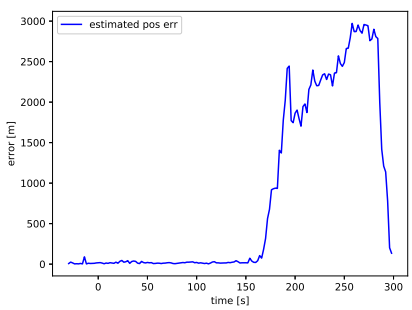
Figure 8.5: Tracking using starboard-side camera

8.2.1.3 Aft

https://drive.google.com/drive/folders/1xZYr0M3TzEzoWIJrnJsId_KMFkywyT4o
The localization fails spectacularly and gets stuck in a local minima as a small island exits the view, but manages to recover once the island is far enough away.



(a) True path compared with estimated path

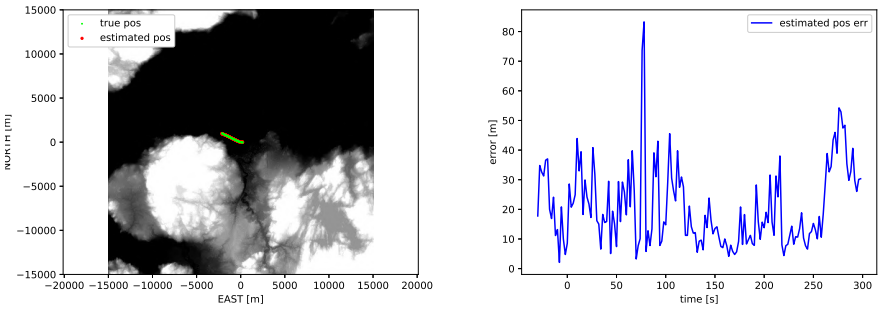


(b) absolute error in estimated path

Figure 8.6: Tracking using the aft camera

8.2.1.4 Port-side

https://drive.google.com/drive/folders/1OJkbG54Wk0QPDjEhxAzxalG7gwP9fIF2
This camera has the best view of the nearby terrain, and manages to localize the ship throughout the entire sequence, albeit with some noise.



(a) True path compared with estimated path

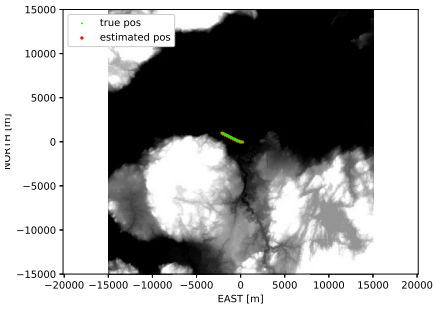
(b) absolute error in estimated path

Figure 8.7: Tracking using the port-side camera

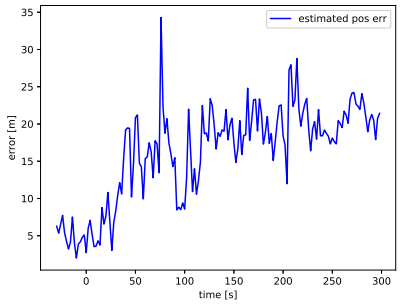
8.2.2 Multi-Camera Tracking

https://drive.google.com/drive/folders/1aSd98XqZtE7QBRDFcIzrYqACC3fpLhPn
Using all cameras simultaneously in the localization is much better than the combined localization of each single camera, and manages to consistently and relatively accurately localize the ship throughout the entire sequence. The error has a bias that is larger for the later part of the sequence. The first part has good view of a nearby island, and the error increases once this island moves out of view. This sequence has a known camera angle error of around 0.1 deg, causing perfect alignment of every camera to be impossible, seen in the trackblend-videos, by extension making perfect pose estimation impossible. The segblend videos also show that the segmentation generally is very good, but does not predict all contours perfectly accurate, which will also affect the segmentation-to-model alignment. Since segmentation and camera calibration is the only difference from the artificial localization, it is apparent that this

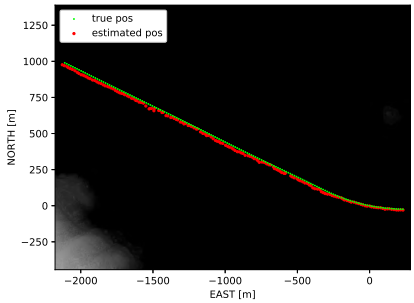
is causes the localization error.



(a) True path compared with estimated path



(b) absolute error in estimated path



(c) True path compared with estimated path, zoomed

Figure 8.8: Tracking using all cameras simultaneously

8.3 Multi-Camera Tracking Performance on Entire Test Set

8.3.1 Overview

Here the localization system is tested on data-sequences from all the regions in the test-set (see ch. 6.1), and the localization performance in each sequence is classified as good, ok or fail. Good means the system managed to accurately and consistenly estimate the pose, taking distance from land into consideration. Ok, means that there are a few small but visible misalignments causing error-spikes, which the system quickly recovers from. Fail means that the system makes very large errors in the localization for however long, or consistently makes obvious misalignments causing a constant significant error. As per chapter 3, normal ship velocity when moving can be up to 20m/s, so when the ship loses track it is very obvious, as the error explodes or grows linearly. Videos are created for each localization sequence, and are together with the error graphs inspected to classify the localization performance for a sequence. The videos are available here
https://drive.google.com/drive/folders/1TPLzuMLonLWutZzT2lJlendu7V0de2xc

Each sequence uses a minute of data, and two seconds pass between each localization, which each do two ICP passes, described in ch. 7. A single ICP pass could have been ran every second halving the start error at each time step, but a focus of these test is to stress the system to expose weaknesses. The first pose at step 0 is like the previous tests initialized with the ground truth pose. Longer sequences would have been preferred, but video creation creates a lot of overhead, and to simulate for all 70 sequences this late in the project compromises had to be made. The tracking results for all test regions is shown in table 8.1.

Table 8.1: Performance of all-cam tracking runs in test-data areas

Run Datetime	Result
2018-06-13_11-50-00	OK
2018-06-13_12-00-00	GOOD
2018-06-13_23-45-00	GOOD
2018-06-15_02-00-00	GOOD
2018-06-15_10-30-00	GOOD
2018-06-15_16-40-00	OK
2018-06-15_18-00-00	GOOD
2018-06-15_19-30-00	GOOD
2018-06-15_20-00-00	GOOD
2018-06-15_21-30-00	GOOD
2018-06-15_22-00-00	GOOD
2018-06-15_23-00-00	GOOD
2018-06-16_07-00-00	GOOD
2018-06-16_07-30-00	GOOD
2018-06-16_08-00-00	GOOD
2018-06-16_08-30-00	GOOD
2018-06-17_05-36-00	GOOD
2018-06-17_15-00-00	GOOD
2018-06-17_19-00-00	GOOD
2018-06-18_03-00-00	GOOD

Run Datetime	Result
2018-06-18_07-30-00	GOOD
2018-06-18_13-30-00	GOOD
2018-06-18_18-00-00	GOOD
2018-06-19_05-00-00	GOOD
2018-06-23_12-20-00	GOOD
2018-06-23_12-30-00	OK
2018-07-01_02-24-00	GOOD
2018-07-01_03-20-00	GOOD
2018-07-01_04-50-00	GOOD
2018-07-03_08-50-00	GOOD
2018-07-03_09-10-00	GOOD
2018-07-03_10-00-00	OK
2018-07-03_11-00-00	GOOD
2018-07-03_11-20-00	GOOD
2018-07-03_11-50-00	GOOD
2018-07-03_19-00-00	GOOD
2018-07-05_13-00-00	GOOD
2018-07-05_15-00-00	GOOD
2018-07-07_04-00-00	GOOD
2018-07-07_07-00-00	GOOD

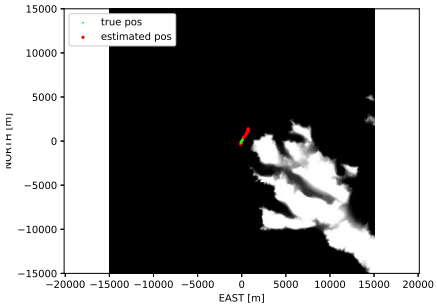
Run Datetime	Result
2018-07-07_08-00-00	GOOD
2018-07-11_03-00-00	GOOD
2018-07-11_04-30-00	GOOD
2018-07-11_07-30-00	GOOD
2018-07-11_14-30-00	GOOD
2018-07-11_15-00-00	GOOD
2018-07-11_15-30-00	GOOD
2018-07-11_16-30-00	GOOD
2018-07-14_12-00-00	GOOD
2018-07-14_19-00-00	GOOD
2018-07-14_19-30-00	GOOD
2018-07-25_10-00-00	OK
2018-07-25_11-00-00	GOOD
2018-07-26_02-00-00	GOOD
2018-07-26_02-30-00	GOOD
2018-07-26_04-00-00	FAIL
2018-07-26_07-00-00	GOOD
2018-07-26_08-00-00	GOOD
2018-07-26_12-30-00	OK
2018-07-26_18-00-00	GOOD

Run Datetime	Result
2018-07-26_21-00-00	GOOD
2018-07-26_22-30-00	FAIL
2018-07-27_01-30-00	GOOD
2018-07-27_04-30-00	GOOD
2018-08-06_15-30-00	GOOD
2018-08-06_19-00-00	GOOD
2018-08-07_02-45-00	GOOD
2018-08-07_16-30-00	FAIL
2018-08-07_18-00-00	OK
2018-08-07_19-50-00	GOOD
2018-08-07_21-00-00	GOOD
2018-09-05_04-30-00	GOOD
2018-09-05_05-30-00	GOOD
2018-09-08_05-30-00	GOOD
2018-09-08_06-00-00	GOOD
2018-09-08_07-00-00	GOOD
2018-10-01_14-00-00	GOOD
2018-10-02_10-20-00	GOOD

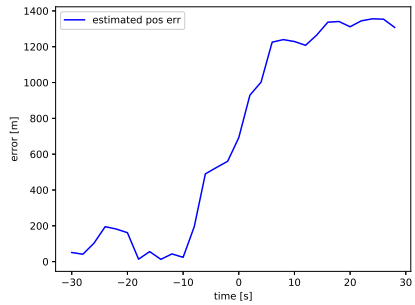
8.3.2 Localization Failures

8.3.2.1 Tracking Failure: 2018-07-26_04-00-00

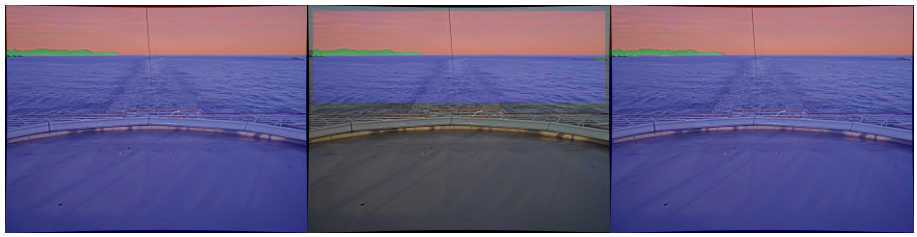
https://drive.google.com/drive/folders/1qo0nYpNseYYh3kEEG6LC096zW2KWVO9H
It's only really visible in the video, but a small island entering the simulated frame makes the pose estimation get stuck in a local minima, as it does not know what to do with the virtual island, as the real island has not entered the camera view. The island is allowed to enter the simulated frame because the starboard camera that actually has a good view of the nearby landmass is disabled. This allows the estimated position vary a lot more, as the only landmasses used in the pose estimation are very far away, requiring big movement to change visually. It also fails at recognizing a small island in the right side of the frame, both in the rendered ground truth and in the segmented image.



(a) True path compared with estimated path



(b) absolute error in estimated path

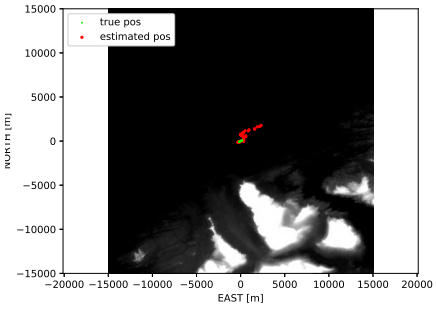


(c) ground truth (d) segmentation result (e) estimated pose

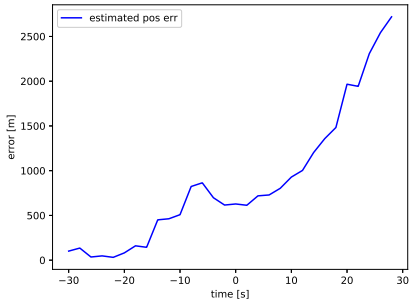
Figure 8.9: Data from the failed localization sequence 2018-07-26_04-00-00

8.3.2.2 Tracking Failure: 2018-07-26_22-30-00

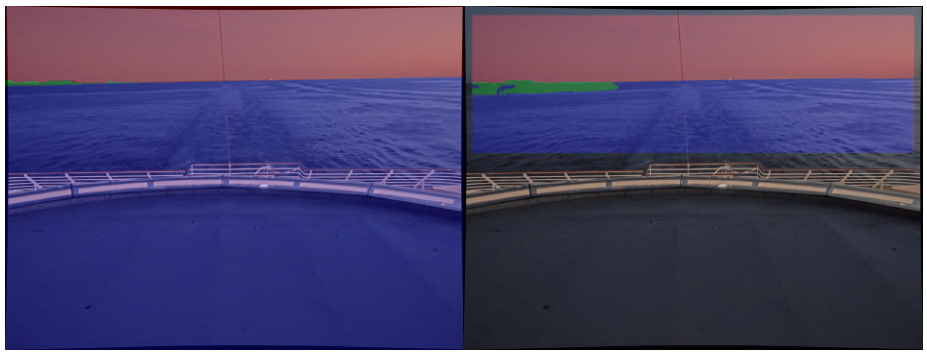
https://drive.google.com/drive/folders/1jPF7hpTr_LITxG5o7W87MxJa6gQy7g79
The cameras barely have some view of far away land despite being close to land due to the land-facing starboard camera being disabled for this sequence. This coupled with failed segmentation of one camera image makes the localization fail. The sun has almost gone down in this sequence, and dark lighting conditions are very sparsely represented in the training data for the segmentation network.



(a) True path compared with estimated path



(b) absolute error in estimated path



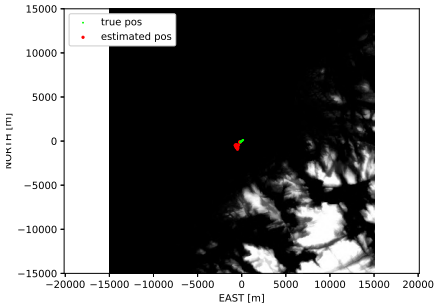
(c) ground truth

(d) segmentation network result

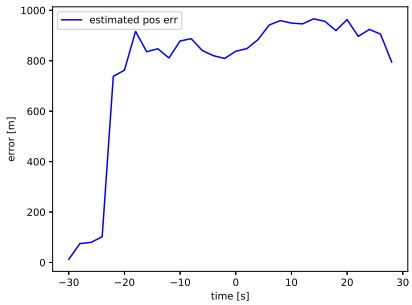
Figure 8.10: Data from the failed localization sequence 2018-07-26_22-30-00

8.3.2.3 Tracking Failure: 2018-08-07_16-30-00

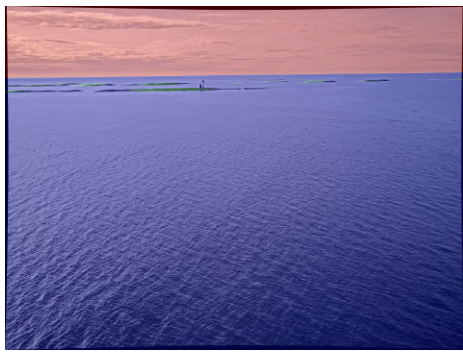
https://drive.google.com/drive/folders/1mj1OZkudKTtQXLBUScISvUBfvlISVIX-
In this tracking sequence, like in the other failed sequences, the missing starboard camera is the only one that is pointed towards any large land masses, while the other cameras barely have some view of very far away land. The port-side camera does in fact have direct view of many close small reef structures, but the segmentation network fails almost completely at labelling them as land.



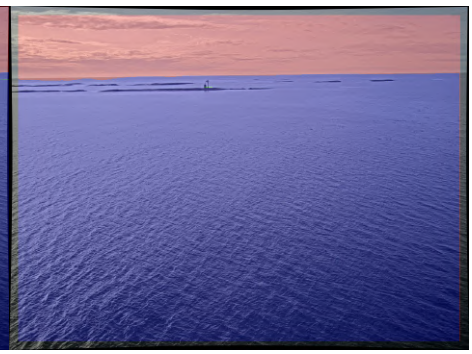
(a) True path compared with estimated path



(b) absolute error in estimated path



(c) ground truth



(d) segmentation network result

Figure 8.11: Data from the failed localization sequence 2018-08-07_16-30-00

Chapter 9

Conclusions and future work

9.0.1 Model and Rendering

The rendered ground truth images do not match the camera images perfectly. This in both due to errors in the 3D models, and errors in the camera angles. Some error in the camera mounting angle was measured and corrected, but part of the error was changing periodically, most likely due to warping of the ship hull as the sun heats the metal. The periodical error seemed to have an amplitude of 0.1 deg, and no further measures were taken to correct this, due to time constraints. When working with the data this is noticeable and important to take into consideration, and should be corrected if high-precision data is required. Since the angle error typically can change over the course of an hour, one could attempt angle correction at a frequency of every 5 minutes over a larger period of interest and store the results as a part of the dataset. Furthermore the calibration of the camera intrinsic parameters and lens distortion coefficients was found to be unsatisfactory for the side cameras in each camera cluster. The middle cameras are ok, and the only difference between the middle and side cameras is the angle at which they look through the shared protective glass dome. It is therefore likely that the 5-parameter lens distortion model does not have the expressive power to model the distortion caused by the tilted view through the glass

dome. Re-calibration using a more complex distortion model should therefore be considered.

Currently the model generation has some problems with incomplete source data, in particular reefs, islands and small ocean structures are very poorly modelled in some regions. Other regions also has huge chunks of missing DOM data, but this can be replaced using DTM data in stead. Using DTM data causes vegetation and man made structures to be unmodelled for those regions. It could be viable to use sentinel data from the Copernicus project in lieu of geonorge.

Certain objects are particularly prone to errors due to simply not being modellable using just a heightmap. One such example is bridges, there is no way of modelling the empty area beneath the bridge, which causes the bridge to either look like a block or be completely missing, depending on the source data. This could warrant the use of some other technique to model these objects, perhaps auto-generating bridge-models, which can then be aligned better by manual inspection.

Currently when models for neighbouring regions are created no information is shared about the overlapping area. Since separate maps must be created when the ship moves out of the center of a map this is a huge waste of space. Splitting the terrain into more modular chunks that can be transformed to conform to the moving ship should definitely be explored. A more modern but still simple LOD technique that would go well with this is chunked LOD. Perhaps if the model was created and stored in ECEF coordinates then the ship's pose could just be converted to ECEF for the rendering, and all maps could easily be created offline without the wasteful duplicate storage of the current implementation.

The current bottleneck of the model generation is the heightmap accessation, since the server needs to gather all the data, and then convert it to NED, which it does extremely slowly for some reason. The Seatex local map service is also slow, due to the conversion. The mesh generation only takes a second, but retrieving the heightmaps takes several minutes, making in the biggest bottleneck of the system by several orders of magnitude. Skipping or optimizing it could give high speed increases.

9.0.2 Semantic Segmentation Network

The segmentation network achieved around 99.5% pixel accuracy on the test-set, but due to the imperfection in what is used as ground truth for the evaluation much closer to 100% is practically impossible. When the network is used to evaluate an image it was trained on it is apparent that it has indeed learned to recreate the specific errors in the training data, the net generalizes well, but the training data errors are definitely a problem, in varying degrees. The network consistently fails for bridges as well as reefs that barely stick out of the water, which are particularly poorly represented in the dataset. Not just represented little, but rather represented incorrectly, and even with good generalization the network does learn to recreate this error, as it is consistently rewarded for it during training. Incorrectly labelling clearly visible small reefs as water is very dangerous for a navigation system relying on situational awareness, and so correcting this faulty data should be a top priority before any new net is trained. Many training runs have been done in parallel with improving in the dataset, and making the dataset bigger more accurate had a direct large impact on the performance. The only bad performance of the segmentation network is for images that are known to be unrepresented or even misrepresented in the training data. The prediction performed poorly in dark lighting conditions, even though structures were clearly visible to a human looking at the pictures. There are not many such images in the data set. Adding more varied lighting conditions to the dataset should therefore be done, but another a simple and effective measure would be to add brightness adjustment to the data augmentation during training, which was neglected due to reaching the internal deadline for work on the segmentation training pipeline. It should be a fairly simple fix, but would require some change to the current method of augmentation which applies the same seeded augmentation to both the image and the mask, which can not be done with an additional brightness adjustment to just the image. Other than adding brightness adjustment the training pipeline does not seem to be in dire need of any improvements, and so the focus should in stead be on improving and expanding the data set.

One important improvement would be to create a segmentation class for boats. Cur-

rently boats are ignored, and are just labelled according to what is behind them, water, land or sky. This often causes some confusion in the test images, but mostly the network correctly predicts what is behind the ship. Still ships that are close to the shore tend to, at the very least, warp the segmented shoreline somewhat. Adding ships to the training data can be done by rendering a color-encoded box in the virtual world using AIS data. AIS data would give both the location and approximate size of the ship, all the parameters needed for the box. Since the world model is in 3D the renderer handles occlusion as well, in case a ship is partially or completely behind some structure. Training samples could also be created by manually labelling data. Recognizing boats is not just important to not confuse the localization system through prediction errors, but could also aid in collision avoidance if a ship has a faulty AIS system, or no AIS at all. Other classes can also be added relatively easily, by using publicly available geotagged data, such as where buildings are located. This could be used to render buildings in a separate color from the general terrain, thus encoding buildings in the rendering. This could be used to then train the net to predict where building are in the image, creating additional information that can be used by the localization system.

9.0.3 Localization

During the experiments it was demonstrated that localization with multiple cameras is much more robust than tracking with single cameras, even when only one camera has close view of the land. The localization using real data was not as accurate as the simulation, which used perfect data. The main causes is that the segmentation of camera images is not perfectly accurate, and secondly the camera data is not perfect, in particular the camera angles, but also the calibration. Furthermore the localization test failed for 3/70 sequences, an in all of the failed test the only camera with good view of the main landmass was the starboard camera, which was disabled for these sequences. In addition to seeing little land, the little land that is seen is mislabeled by the segmentation network, which is again caused by bad training data. The localization can still be successful even if one of the cameras makes some occasional inaccurate

predictions. One such extreme case happened when the ship drove towards a bridge, and the front camera made wild predictions due to poor training on bridges. The other three cameras still kept the localization mere meters from ground truth over the whole sequence. In the test sequences all the failures could actually have been avoided if all the cameras were available like they were supposed to, and the segmentation network was trained without the faulty data.

It was inadvertently discovered how much the accuracy of the camera calibration affects the accuracy of the tracking. For an initial run which is not documented further the intrinsics for the forward camera was accidentally used as the intrinsics for all the cameras, even though their values differ. This caused the tracking to become very unstable with a big bias, and much less accurate even when the segmentation and the model was good. Erroneous camera angles also affect the localization accuracy. In areas where the camera angles are particularly inaccurate for a camera, this camera does not give a perfect model alignment for the same ship pose as the other cameras. This causes the localization to compromise at a pose that's not a perfect alignment any of the cameras, a pose that is near but not at the ground truth.

For some of the localization test the error spikes just as objects, such as small islands, are leaving the frame. This is since to the edge pair selection process knows nothing about points that are not seen in the current frame, causing some confusion when the camera segmentation points leave the frame but the render points remain. This effect is particularly large for these test since no filtering is performed. If the ship moves at 20m/s even perfect localization at each frame will start with a 40m error when estimating the pose every 2 seconds, meaning the initial rendering and segmentation image can be quite different, particularly for close terrain such as small islands.

One very natural extension of the system is to include a kalman filter in the localization system, smoothing out jittery localization and respecting the ship dynamics between time steps. Multiple faulty localizations in succession could still greatly impact the filtered value however. A more robust system, which is the basis of most modern visual localization system would be to use a graph based filtering approach. one such graph framework, GTSAM (Georgia Tech Smoothing And Mapping), lets you add custom factors to the smoothing process, proved you have the jacobian, meaning

we could add the segmentation-to-model alignment for different times as factors. This would basically be motion only bundle adjustment, very related to SLAM. It could therefore be interesting to go further and fuse feature point tracking into the localization, turning it into a full fledged SLAM system, where the segmentation-to-model alignment is used to correct drift in the system over time. The terrain model could even be used to initialize the location of feature points, and the segmentation masks out feature points that are undesired for tracking. If also fused with inertial measurements, there is the potential for a very robust system. No matter what filter system is chosen, one suggestion would be to model the camera angels to have a bias that can change over time, thus automatically compensating for the changing camera angles. Bias estimation is already common for IMUs, so why not for camera angles.

A huge weakness of the system right now is that it has no way of accounting for occlusions, such as clouds hanging low over mountains, causing the mountains to be classified as sky, which is not modelled. One idea could be to detect if the alignment for some region becomes poor, and gradually give lower weight in the optimization to consistently poorly aligned regions. Since the render depth is known, less weight could be given to regions far away, as they are more likely to be occluded. One could also consider temporarily changing the labels in the model, and in doing so modelling the clouds. This kind of weighing is more intuitive when the whole image is used for alignment, not just the label transitions, motivating further evaluation of such an approach, in which case GPU programming should be looked more into.

As of now the system is pretty slow, ca. 0.2Hz, as virtually no steps have been taken to optimize speed, apart from the re-projection scheme. Extracting the various edges is particularly slow, and so perhaps the edges could be created directly by the rendering engine, or at least be calculated using some GPU programming framework, like CUDA. Numpy, a python package which is used for almost all the maths in this project, is very slow when using big arrays, and just initializing a large array takes considerable time due to the complex datatypes it implements. Doing matrix math and loops in general is much slower in python than for instance in c++, and so a real-time implementation should definitely consider changing implementation language, or at least wrap some sections with c++.

References

[1] Adrien Angeli, David Filliat, Stéphane Doncieux, and Jean-Arcady Meyer. A fast and incremental method for loop-closure detection using bags of visual words. *IEEE Transactions on Robotics*, pages 1027–1037, 2008.

[2] Anil Armagan, Martin Hirzer, Peter M. Roth, and Vincent Lepetit. Learning to align semantic segmentation and 2.5d maps for geolocalization. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[3] C. Arth, C. Pirchheim, J. Ventura, D. Schmalstieg, and V. Lepetit. Instant outdoor localization and slam initialization from 2.5d maps. *IEEE Transactions on Visualization and Computer Graphics*, 21(11):1309–1318, Nov 2015.

[4] Georges Baatz, Olivier Saurer, Kevin Köser, and Marc Pollefeys. Large scale visual geo-localization of images in mountainous terrain. volume 7573, pages 517–530, 10 2012.

[5] Hernán Badino, Daniel Huber, and Takeo Kanade. Real-time topometric localization. In *2012 IEEE International Conference on Robotics and Automation*, pages 1635–1642. IEEE, 2012.

[6] Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In *Sensor Fusion IV: Control Paradigms and Data Structures*, volume 1611, pages 586–607. International Society for Optics and Photonics, 1992.

[7] Andrea Censi. An icp variant using a point-to-line metric. 2008.

[8] MWM Gamini Dissanayake, Paul Newman, Steve Clark, Hugh F Durrant-Whyte, and Michael Csorba. A solution to the simultaneous localization and map building (slam) problem. *IEEE Transactions on robotics and automation*, 17(3):229–241, 2001.

[9] J. Engel, V. Koltun, and D. Cremers. Direct sparse odometry. In *arXiv:1607.02565*, July 2016.

[10] J. Engel, T. Schöps, and D. Cremers. LSD-SLAM: Large-scale direct monocular SLAM. In *European Conference on Computer Vision (ECCV)*, September 2014.

[11] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[12] X. Gao, R. Wang, N. Demmel, and D. Cremers. Ldso: Direct sparse odometry with loop closure. In *iros*, October 2018.

[13] Marcel Geppert, Peidong Liu, Zhaopeng Cui, Marc Pollefeys, and Torsten Sattler. Efficient 2d-3d matching for multi-camera visual localization. *CoRR*, abs/1809.06445, 2018.

[14] Giorgio Grisetti, Rainer Kummerle, Cyrill Stachniss, and Wolfram Burgard. A tutorial on graph-based slam. *IEEE Intelligent Transportation Systems Magazine*, 2(4):31–43, 2010.

[15] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, et al. Recent advances in convolutional neural networks. *Pattern Recognition*, 77:354–377, 2018.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[17] Martin Hirzer, Clemens Arth, Peter M. Roth, and Vincent Lepetit. Efficient 3d tracking in urban environments with semantic segmentation. 09 2017.

[18] Martin Hirzer, Peter Michael Roth, Clemens Arth, and Vincent Lepetit. Pose determination with semantic segmentation, March 14 2019. US Patent App. 15/699,221.

[19] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Proceedings Visualization'98 (Cat. No. 98CB36276)*, pages 35–42. IEEE, 1998.

[20] Albert S Huang, Abraham Bachrach, Peter Henry, Michael Krainin, Daniel Maturana, Dieter Fox, and Nicholas Roy. Visual odometry and mapping for autonomous flight using an rgb-d camera. In *Robotics Research*, pages 235–252. Springer, 2017.

[21] Hirokazu Kato and Mark Billinghurst. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. In *Proceedings 2nd IEEE and ACM International Workshop on Augmented Reality (IWAR'99)*, pages 85–94. IEEE, 1999.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[23] Julien Li-Chee-Ming and Costas Armenakis. Uav navigation system using line-based sensor pose estimation. *Geo-spatial Information Science*, 21:1–10, 01 2018.

[24] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[25] Colin McManus, Ben Upcroft, and Paul Newman. Learning place-dependant features for long-term vision-based localisation. *Autonomous Robots*, 39(3):363–387, 2015.

[26] R. Mottaghi, M. Kaess, A. Ranganathan, R. Roberts, and F. Dellaert. Place recognition-based fixed-lag smoothing for environments with unreliable gps. In *2008 IEEE International Conference on Robotics and Automation*, pages 1862–1867, May 2008.

[27] Montiel J. M. M. Mur-Artal, Raúl and Juan D. Tardós. ORB-SLAM: a versatile and accurate monocular SLAM system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[28] Diederick C Niehorster, Li Li, and Markus Lappe. The accuracy and precision of position and orientation tracking in the htc vive virtual reality system for scientific research. *i-Perception*, 8(3):2041669517708205, 2017.

[29] David Nistér, Oleg Naroditsky, and James Bergen. Visual odometry for ground vehicle applications. *Journal of Field Robotics*, 23(1):3–20, 2006.

[30] Gabriel L Oliveira, Noha Radwan, Wolfram Burgard, and Thomas Brox. Topometric localization with deep learning. *arXiv preprint arXiv:1706.08775*, 2017.

[31] Simon JD Prince. *Computer vision: models, learning, and inference*. Cambridge University Press, 2012.

[32] Victor Prisacariu and Ian D. Reid. Pwp3d: Real-time segmentation and tracking of 3d objects. volume 98, 01 2009.

[33] Srikumar Ramalingam, Sofien Bouaziz, and Peter F. Sturm. Pose estimation using both points and lines for geo-localization. pages 4716–4723, 05 2011.

[34] Gerhard Reitmayr and Tom Drummond. Going out: robust model-based tracking for outdoor augmented reality. In *ISMAR*, volume 6, pages 109–118, 2006.

[35] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary R Bradski. Orb: An efficient alternative to sift or surf. Citeseer.

[36] Davide Scaramuzza and Friedrich Fraundorfer. Visual odometry [tutorial]. *IEEE robotics & automation magazine*, 18(4):80–92, 2011.

[37] K. Tateno, F. Tombari, I. Laina, and N. Navab. Cnn-slam: Real-time dense monocular slam with learned depth prediction. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6565–6574, July 2017.

[38] George Terzakis, Riccardo Polvara, Sanjay K. Sharma, Phil F. Culverhouse, and Robert Sutton. Monocular visual odometry for an unmanned sea-surface vehicle. *CoRR*, abs/1707.04444, 2017.

[39] Henning Tjaden, Ulrich Schwanecke, Elmar Schömer, and Daniel Cremers. A gauss-newton approach to real-time monocular multiple object tracking. *CoRR*, abs/1807.02087, 2018.

[40] Henning Tjaden, Ulrich Schwanecke, Elmar Schömer, and Daniel Cremers. A gauss-newton approach to real-time monocular multiple object tracking. *CoRR*, abs/1807.02087, 2018.

[41] Steffen Urban and Stefan Hinz. MultiCol-SLAM - a modular real-time multi-camera slam system. *arXiv preprint arXiv:1610.07336*, 2016.

[42] J. Ventura, C. Arth, G. Reitmayr, and D. Schmalstieg. Global localization from monocular slam on a mobile phone. *IEEE Transactions on Visualization and Computer Graphics*, 20(4):531–539, April 2014.

[43] L. von Stumberg, V. Usenko, and D. Cremers. Direct sparse visual-inertial odometry using dynamic marginalization. In *International Conference on Robotics and Automation (ICRA)*, May 2018.

[44] Lukas von Stumberg, Patrick Wenzel, Qadeer Khan, and Daniel Cremers. Gn-net: The gauss-newton loss for deep direct slam, 2019.

[45] Peng Wang, Ruigang Yang, Binbin Cao, Wei Xu, and Yuanqing Lin. Dels-3d: Deep localization and segmentation with a 3d semantic map. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[46] R. Wang, M. Schwörer, and D. Cremers. Stereo dso: Large-scale direct sparse visual odometry with stereo cameras. In *International Conference on Computer Vision (ICCV)*, Venice, Italy, October 2017.

[47] Eric W Weisstein. Euler angles. 2009.

[48] Eric W Weisstein. Euler angles. 2009.

[49] Matthew J Westoby, James Brasington, Niel F Glasser, Michael J Hambrey, and JM Reynolds. 'structure-from-motion'photogrammetry: A low-cost, effective tool for geoscience applications. *Geomorphology*, 179:300–314, 2012.

[50] Thomas Whelan, Stefan Leutenegger, R Salas-Moreno, Ben Glocker, and Andrew Davison. Elasticfusion: Dense slam without a pose graph. Robotics: Science and Systems, 2015.

[51] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. In *Advances in Neural Information Processing Systems*, pages 4148–4158, 2017.

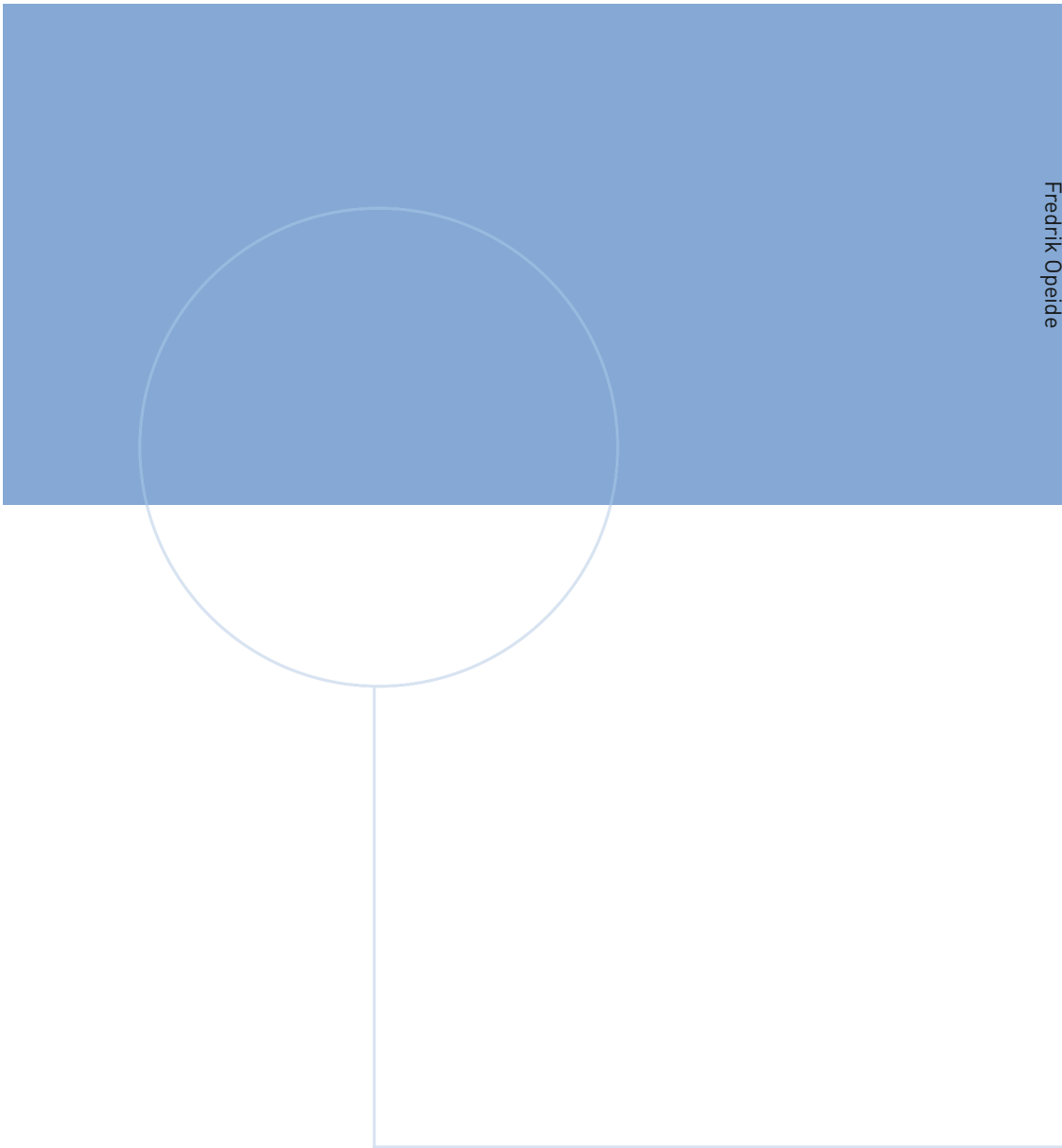
[52] Harald Wuest and Didier Stricker. Tracking of industrial objects by using cad models. *Journal of Virtual Reality and Broadcasting*, 4, 10 2007.

[53] N. Yang, R. Wang, J. Stueckler, and D. Cremers. Deep virtual stereo odometry: Leveraging deep depth prediction for monocular direct sparse odometry. In *eccv*, September 2018. "https://vision.in.tum.de/research/vslam/dvso".

[54] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[55] Hengshuang Zhao, Xiaojuan Qi, Xiaoyong Shen, Jianping Shi, and Jiaya Jia. Icnet for real-time semantic segmentation on high-resolution images. *CoRR*, abs/1704.08545, 2017.

[56] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. Pyramid scene parsing network. *CoRR*, abs/1612.01105, 2016.



Fredrik Opeide

NTNU
Norwegian University of
Science and Technology