Master's thesis

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

Maxime Roedelé

GNSS-free maritime navigation using DEM data and monocular camera images

Master's thesis in Cybernetics and Robotics
Supervisor: Tor Arne Johansen
Co-supervisor: Kjetil Vasstein
June 2023

NTNU
Norwegian University of
Science and Technology

Maxime Roedelé

GNSS-free maritime navigation using DEM data and monocular camera images

Master's thesis in Cybernetics and Robotics
Supervisor: Tor Arne Johansen
Co-supervisor: Kjetil Vasstein
June 2023

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

NTNU
Norwegian University of
Science and Technology

Abstract

The potential threats to satellite based navigation have renewed the interest in GNSS-independent systems as a means to secure the continued service of critical infrastructure. To this end, a baseline framework for a GNSS-independent navigation system in the maritime sector is presented in this report. The proposed framework only utilizes a monocular camera and a digital elevation model of its surroundings to estimate a vessel's deviation from a predefined, intended trajectory. A Unity based simulator is further used to simulate the vessel and its surroundings, before an independent Python application is deployed to analyze the synthetic data. The pose estimation was done through minimization of the reprojection error between features in the terrain and a matching feature in the camera image plane. Image features from the intended and actual trajectories were extracted using morphological operations and an iterative depth-first search, before being matched through a sum of absolute differences energy measure. The feature in the 3D environment was found through a ray-tracing scheme with image features for reference. 3 different trajectories affected by varying degrees of noise were used to evaluate the systems performance. Positive improvements of the mean square error ranging from 1.5% to 12.8% were recorded on paths experiencing moderate to high disturbances, whilst negative contributions were recorded on paths with little to no noise. The various modules of the system were analyzed, verified and deemed adequate, with the exception of the matching algorithm for image features and the adaptive grid search algorithm for pose-estimation. Overall, the framework was deemed a promising prospect for future development, with the potential of furthering scientific solutions in the field of long-range visual odometry.

Sammendrag

Potensielle trusler rettet mot satellittbaserte navigasjonssystemer har vekket fornyet interesse for GNSS-uavhengige systemer, som et middel for å sikre fortsatt drift av kritisk infrastruktur. I tråd med dette presenterer denne rapporten et grunnleggende rammeverk for et GNSS-uavhengig navigasjonssystem for bruk i den maritime sektoren. Det foreslåtte rammeverket brukte kun et monokulært kamera og en digital høydemodell av fartøyets omgivelser for å estimere avviket fra en forhåndsdefinert, planlagt bane. En simulator laget i Unity ble videre brukt for å simulere fartøyet og dets omgivelser, mens en separat applikasjon i Python ble anvendt for å analysere den syntetiske dataen fra simulatoren. Posisjonsestimatet ble utført ved å minimere reprojeksjonsfeilen mellom et landemerke i terrenget og et tilsvarende landemerke lokalisert i bildeplanet til kameraet. Landemerker i bildeplanene til de to banene ble funnet ved hjelp av morfologiske operasjoner og et iterativt dybde-først-søk, før de ble forsøkt matchet ved hjelp av en sum av absolutte forskjeller. Landemerker i 3D terrenget ble funnet ved hjelp av en strålesporingsalgoritme, som benyttet landemerkene i ett av bildeplanene som referanse. 3 ulike baner, påvirket av varierende grader med støy, ble videre definert og brukt til å evaluere systemets ytelse. Positive forbedringer av den kvadratiske gjennomsnittsfeilen på mellom $1,5\%$ og $12,8\%$ ble registrert for baner som opplevde moderat til høye forstyrrelser, mens negative bidrag ble registrert hvor liten til ingen støy ble lagt til. De ulike modulene i systemet ble så analysert, verifisert og vurdert tilstrekkelige, med unntak av sammenligningsalgoritmen mellom landemerker i bildeplan og den adaptive grid-search algoritmen brukt for posisjonsestimering. Rammeverket ble alt i alt vurdert som et lovende fremtidsprospekt, med høyt potensial innen fremtidig, visuell posisjonsestimering til havs.

Acknowledgements

I would first like to express my sincerest gratitude to all those who have contributed to the successful completion of this thesis, be it professionally or personally. I doubt it would have seen the light of day without your continued support.

Even so, I would like to express my deepest gratitude to my supervisors Tor Arne Johansen and Kjetil Vasstein throughout this project, for their immense patience, endless advice and scientific contributions from start to finish. Without their help and guidance, through countless mails and hours of meetings over Microsoft Teams, this thesis would have looked a whole lot different. I hope to be able to work with you again on new projects some day!

A special thanks is also directed to my partner Lee and to my family. Without their reassurances, help and support in difficult times I'm not sure this work would have been completed at all. Knowing that I can always rely on you when things get tough means the world to me. I love you all!

List of mathematical symbols

$\vec{\omega}_i$	Intended position in the XZ plane at waypoint i
$\theta_{\vec{\omega}_i}$	Intended heading of the vessel at waypoint i
$\vec{p}_i$	Actual position in the XZ plane at waypoint i
$\theta_{\vec{p}_i}$	Actual heading of the vessel at waypoint i
$\mathcal{N}_w$	Number of columns in the semantic segmentations
$\mathcal{N}_h$	Number of rows in the semantic segmentations
$\mathrm{I}_{\vec{\omega}_i}$	Semantic segmentation of terrain captured at $\vec{\omega}_i$
$\mathrm{I}_{\vec{p}_i}$	Semantic segmentation of terrain captured at $\vec{p}_i$
$\mathrm{S}_{\vec{\omega}_i}$	Skyline contour extracted from $\mathrm{I}_{\vec{\omega}_i}$
$\mathrm{S}_{\vec{p}_i}$	Skyline contour extracted from $\mathrm{I}_{\vec{p}_i}$
$\mathcal{M}$	Number of features
$\mathcal{L}$	Length of all features
$\mathcal{K}$	Amount of possible features within a skyline contour
$\vec{f}_{\vec{\omega}_i}$	Matched image feature extracted from $\mathrm{S}_{\vec{\omega}_i}$
$\vec{f}_{\vec{p}_i}$	Matched image feature extracted from $\mathrm{S}_{\vec{p}_i}$
$\vec{R}_j$	3D ray used to find element $j, j \in [0\ \mathcal{L})$ in $\vec{f}_{DEM_i}$ matching element j in $\vec{f}_{\vec{\omega}_i}$
$\vec{f}_{DEM_i}$	Matched set of 3D features extracted from the dem
r	Resolution of the DEM
e	Reprojection error
$\hat{p}_i$	Estimated position in the XZ plane at waypoint i
$\theta_{\hat{p}_i}$	Estimated heading of the vessel at waypoint i
h	Height of the camera aboard the maritime vessel
$\mathbf{K}$	Intrinsic camera matrix
f	Focal length of the camera aboard the maritime vessel
f_x	Directional focal length of the camera along the horizontal axis
f_y	Directional focal length of the camera along the vertical axis
o_x	Camera principal point coordinate along the horizontal axis
o_y	Camera principal point coordinate along the vertical axis
FOV_H	Horizontal field of view of the camera
$\mathbf{E}$	Extrinsic camera matrix
$\mathbf{R}^w_{c\theta}$	Rotation matrix from a world coordinate system to a camera coordinate system with a general rotation θ
$\vec{t}^w_{c\theta}$	Translation vector from a world coordinate system to a camera coordinate system with a general rotation θ
E_N	Structuring element for morphological noise removal
E_E	Structuring element for morphological erosion of skyline contours
μ	Mean of Gaussian noise
ϕ	Variance of Gaussian noise

List of technical abbreviations

GNSS	global navigational satellite systems
GPS	global positioning system
APNT	alternative positioning navigation and timing
DoS	denial of service
DEM	digital elevation model
DOF	degree of freedom
AI	artificial intelligence
ML	machine learning
CV	computer vision
VO	visual odometry
USV	unmanned surface vehicle
SLAM	simultaneous localization and mapping
IMU	inertial measurement unit
FOV	field of view
SAD	sum of absolute differences
MSE	mean squared error
API	application programming interface

Table of Contents

1 Introduction

1.1 Motivation

In the maritime sector, global navigational satellite systems (GNSS) have been the de facto gold standards for navigation for the better part of a decade. Originally released to the public as the US military's global positioning system (GPS), satellite based navigation has rendered alternative positioning navigation and timing (APNT) systems mostly redundant [1] and become solely responsible for the efficiency of one of the most important socio-economic sectors in our modern world [2–4]. Every year, billions of tonnes of cargo and millions of passengers are transported over national and international waterways [5, 6], making the safeguarding and security of navigational integrity of joint, global interest.

With the breakout of the war of Ukraine in February of 2022, the known weaknesses of GNSS have become more apparent than ever [7, 8] and the prospect of denial of service (DoS) through jamming devices or spoofing of GNSS signals [9] is causing growing, international unrest. A study from 2019 with US government backing estimates economical losses of 5-14 billion USD given a 30-day GPS outage only in the maritime sector [10], whilst [11] highlights the possible catastrophic consequences of GNSS denial to critical infrastructure and personnel. As so, a global endeavour is now underway to limit the necessity of GNSS in a variety of sectors, including for maritime navigation.

For the sake of simplicity, a division of GNSS independent navigational systems in development into two distinct sub-categories is adopted: Signal dependent and signal independent systems. Signal dependent systems are defined as relying on the propagation of EM-waves in one form or another, whilst signal independent systems perform pose estimation and navigation without relying on external information. This report seeks to present an experimental framework for the latter, using a single monocular camera aboard a maritime vessel to estimate its position in known terrain.

1.2 Problem description

The proposed navigation problem is limited to maritime vessels following a predefined trajectory, in this case a piece-wise linear path consisting of n densly located *waypoints* $\vec{\omega}_i$, $i \in [0, n]$ in the 2D plane, dubbed the *intended path*. The vessel is further assumed to deviate from said intended path, yielding a separate behaviour defined as the *actual path*. The vessel is further assumed armed with one or more mounted, calibrated monocular camera(s) of which all parameters are known, as well as a digital elevation model (DEM) of the vehicle's surroundings. The camera's mounting height h on this vehicle is also assumed known, but is, for proof of concept, set to a negligible value $h \simeq 0$. Using only these tools, the problem is to estimate the vessel's deviation from the intended path, i.e the actual path, only using images captured from the camera(s) and the on-board DEM.

Figure 1 illustrates a simplified version of the problem: A vessel near waypoint i misses the intended position $\vec{\omega}_i$, ending up at a different position $\vec{p}_i$. This deviation is not immediately apparent to the vessel. The vessel uses synthetic data (the on-board DEM) to generate a *semantic segmentation* of the visible terrain given the assumption of being in the intended pose. This synthetic image is labeled $\mathrm{I}_{\vec{\omega}_i}$. Similarly, a semantic segmentation of the terrain in the actual state of the boat can be generated in real time from the camera. This image is labeled $\mathrm{I}_{\vec{p}_i}$. From these segmentation and the DEM, a set of features are extracted and used to optimize the 3 degree of freedom (DOF) pose to minimize the so-called *reprojection error*. A 6 DOF pose, which would be more realistic but significantly increases problem complexity, is omitted by the assumption of constant altitude and either negligible or compensated roll- and yaw-angles aboard the vessel. The actual pose at waypoint i then consists of 3 variables, the vessel's position in the local coordinate frame ($\vec{p}_{ix}$, $\vec{p}_{iy}$), as well as the clockwise bearing of the vessel in the 2-dimensional plane $\theta_{\vec{p}_i}$. However, given the nature of piece-wise linear paths, the intended and actual headings, $\theta_{\vec{\omega}_i}$ and $\theta_{\vec{p}_i}$, will be known for a linear segment if the previous and next positions are known, further reducing the complexity of the problem to a 2 DOF position estimation at waypoint i, $\hat{p}_i$.

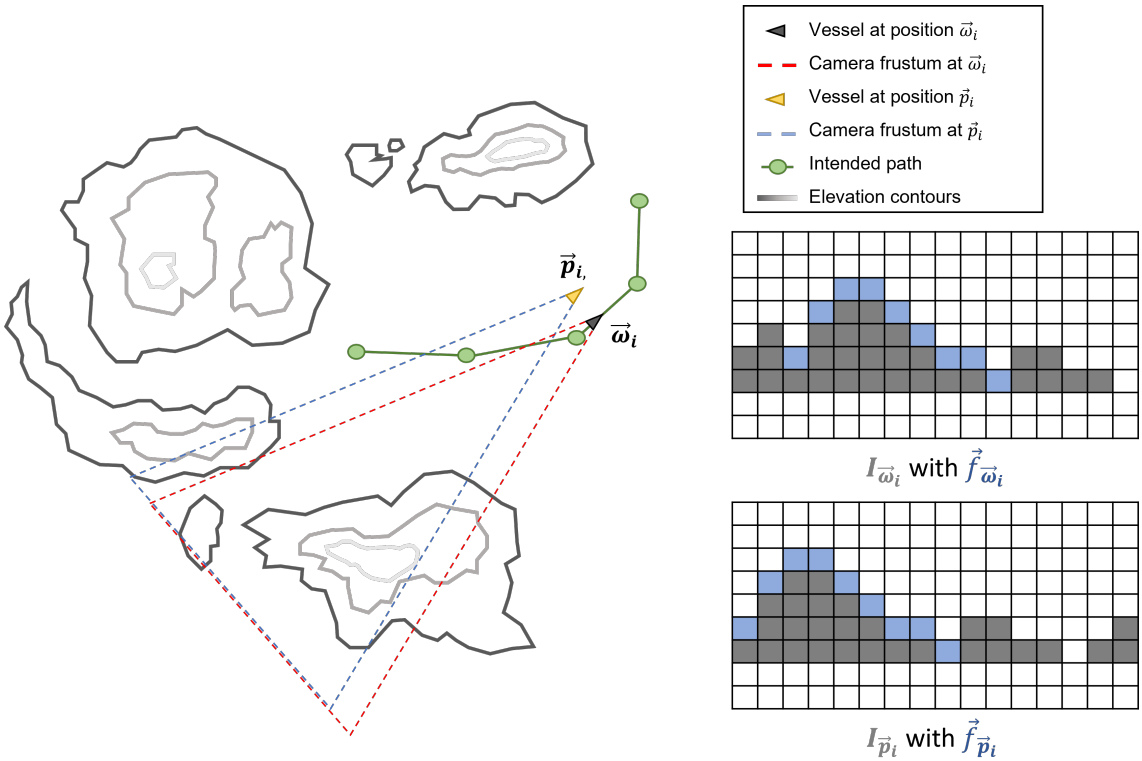


Figure 1: Simplified summary of the overall problem. Two semantic segmentations $\mathrm{I}_{\vec{\omega}_i}$ and $\mathrm{I}_{\vec{p}_i}$ are captured at waypoint i and used to estimate the actual position $\vec{p}_i$ deviating from the intended position $\vec{\omega}_i$ through optimizing the reprojection error.

At each i, an update of known a-priori knowledge is assumed to occur. That is, the boat is assumed to update its knowledge of $\vec{\omega}_{i-1}$ such that $\vec{\omega}_{i-1} = \hat{p}_{i-1}$. This is a necessary assumption when developing the presented estimation scheme.

For the purpose of testing and development, the images captured by the vessel are to be generated in a Unity based simulator environment and analysis of said images are to be performed post data acquisition in Python. Unity is an open world, cross platform game engine capable of generating highly realistic, physics based simulations, whilst Python is a high-level computer programming language, optimal for precise, easy proofs of concepts given the extensive support of libraries for advanced data modification and analysis. The images generated in Unity are to be on the form of *semantic segmentations*, that is, they are to contain pixel-level class labelling of the terrain visible from the vessel.

1.3 Related works

The rapid advances in artificial intelligence (AI), machine learning (ML) and computer vision (CV) over the last decade has laid a strong foundation for the research and development around visual odometry (VO). In many autonomous and/or robotics systems, the concept of being able to use one or more cameras to estimate an agents ego-pose whilst simultaneously providing object detection and tracking [12, 13] is alluring, avoiding the dependencies of GNSS and allowing for potentially cheap and precise navigation in challenging scenarios [14]. In the fields of autonomous cars [15–18], interplanetary rovers [19–22] and aviation [23, 24], VO occurs fairly frequently, emerging in standalone implementations [16, 22, 23] or as a part of larger sensor-fusions [17, 20, 24]. Generally, the approaches are divided in *knowledge based* and *learning based* approaches, utilizing geometric properties and ML models based on vast amounts of data, respectively [18]. Knowledge based VO, falling in line with the proposed framework in this paper, often use features between a set of images, like the 3-dimensional peaks proposed by Li Wei and Sukhan Lee [22], the lunar craters proposed by Larry Matthies et al. [21] or the complete features obtained by a monocular camera and matched

with LiDAR depth-estimates proposed by Johannes Graeter et al. [17]. Similar methods have proven to yield good results on everything from short- to long-range ego-pose estimations.

As [25] highlights however, similar work for marine deployment is rather lackluster. Although the focus has been increasing in recent years, VO is still in a somewhat experimental field for maritime vessels. The best results have primarily been reported for short-range ego-pose estimation, like the marker-based docking system employed by Volden et al. [26] and Cortes-Vega et al.'s unmanned surface vehicle (USV) controller based on monocular VO in urban waterways [27]. VO is more common, arguably more so than in other sectors, as part of a larger sensor fusion, where great successes are being reported in applications such as the autonomous passenger ferry Milliampere 2 [28], Liu et al.'s augmented reality based navigation system [29] and a promising navigation system for smart ships based on game theory proposed by Zhou et al. [30]. In such fusions, cameras are often deployed as part of a greater system for simultaneous localization and mapping (SLAM), allowing for both exploration and navigation in a variety of coastal areas.

Although major sensor fusions are seeing the most success in the maritime sector, researchers have also found interest in navigation and pose-estimation based on a limited amount of sensors; work that can prove interesting for more economic appliances or in larger fused frameworks. Ma et al. propose a feature based radar system, extracting coastlines and inferring with satellite data to determine the position of a USV [31], Jung et al. present a working sonar based particle filter performing navigation by underwater geophysical features [32] and Rogne et al. presented an initial framework for dead-reckoning based position estimation using multiple inertial measurement unit (IMU) [33]. [34, 35] attempt to use infrared cameras for various navigational services whilst Jungwook et al. use a typical marine radar to perform SLAM on coastal areas [36]. One should also note some of the significant contributions advances in AI present to a variety of these single-sensor applications, especially for monocular VO. The ever-improving YOLO labelling algorithms [37] and monocular depth estimation algorithms, such as the exceptional Boosting network [38] are promising steps in covering the many shortcomings of monocular cameras, rendering them proficient, standalone sensors.

1.4 Goals and contributions

This report seeks to present a baseline framework for long-range, knowledge-based, monocular VO aboard a maritime vessel following an intended trajectory in known terrain. A 2 DOF position in the XZ plane is estimated by the reprojection error, using synthetically generated semantic segmentations of terrain visible aboard the maritime vessel and abstraction to matching features in the DEM.

The main contributions can be summarized as:

- A baseline estimation framework based on reprojection errors and nonlinear optimization, including algorithms for feature extraction and matching given a set of semantic segmentations and a-priori information about the vessel's surroundings in the form of a DEM.
- A baseline 3D Unity simulator environment for consistent data generation and processing.
- A qualitative analysis of the algorithmic framework in light of experimental results.

1.5 Structure of the report

The report is divided into 7 sections in total. Section 2 provides an overview of the complete proposed framework, for the reader to familiarize him/herself with the intended workings of the system. Section 3 presents all deployed algorithms, with illustrations and pseudocode for reproducibility. Section 4 presents the proposed simulator environment and necessary considerations when working between Unity for data generation and Python for analysis. Section 5 presents the results of the framework applied to the synthetically generated data and Section 6 serves as a discussion of said results. Section 7 presents the final conclusion.

2 Overview of the proposed framework

The framework proposed in this report is designed to function as a standalone pose-estimator for a marine vessel following a predetermined trajectory in known terrain. Although the possible amount of cameras and their models or configurations are technically limitless aboard the vessel, it is here assumed a single, forward facing camera with a limited horizontal field of view (FOV), FOV_H, typically ranging between 60°and 90°to accommodate a pinhole-camera model. The vessel's intended trajectory is assumed to be piece-wise linear, connected between a set of n densely populated waypoints, and the surrounding terrain is assumed know on the form of a DEM with a fixed resolution and known spatial constraints. Figure 2 illustrates this intended behaviour on an arbitrary maritime vessel in a simple example environment consisting of several islands of differing height-profiles:

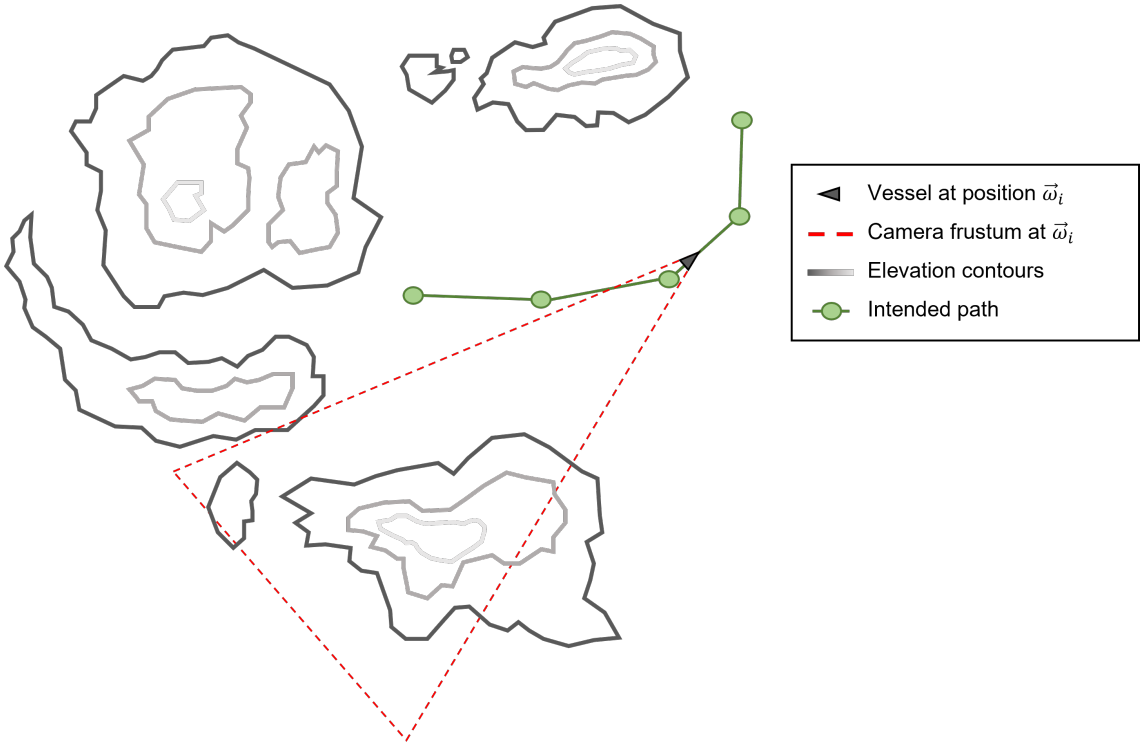


Figure 2: The assumed *intended* behaviour of a maritime vessel following a predetermined, piece-wise linear path. The vessels camera frustum is limited to $\text{FOV}_H < 90°$ to accomodate a pinhole camera model and the DEM of the surrounding terrain is assumed known.

By defining a local reference frame for the maritime vessel within this environment or using geospatial referencing of both waypoints and the vessel's pose it is possible to construct a variety of robust control-schemes guiding the vessel from one waypoint to another. Such a scheme will, however, be subject to a plethora of error-sources; a reality ever more prevalent for control schemes functioning out at sea. External, time dependent factors such as waves, winds and sub-surface currents are certain to cause deviations from the intended path of any maritime vessel, rendering the actual behaviour of the proposed maritime vessel more reminiscent of that presented in Figure 3. Note that at $i = 0$, $\vec{\omega}_0 = \vec{p}_0$, as initialization is assumed supervised and verified by human inspection.

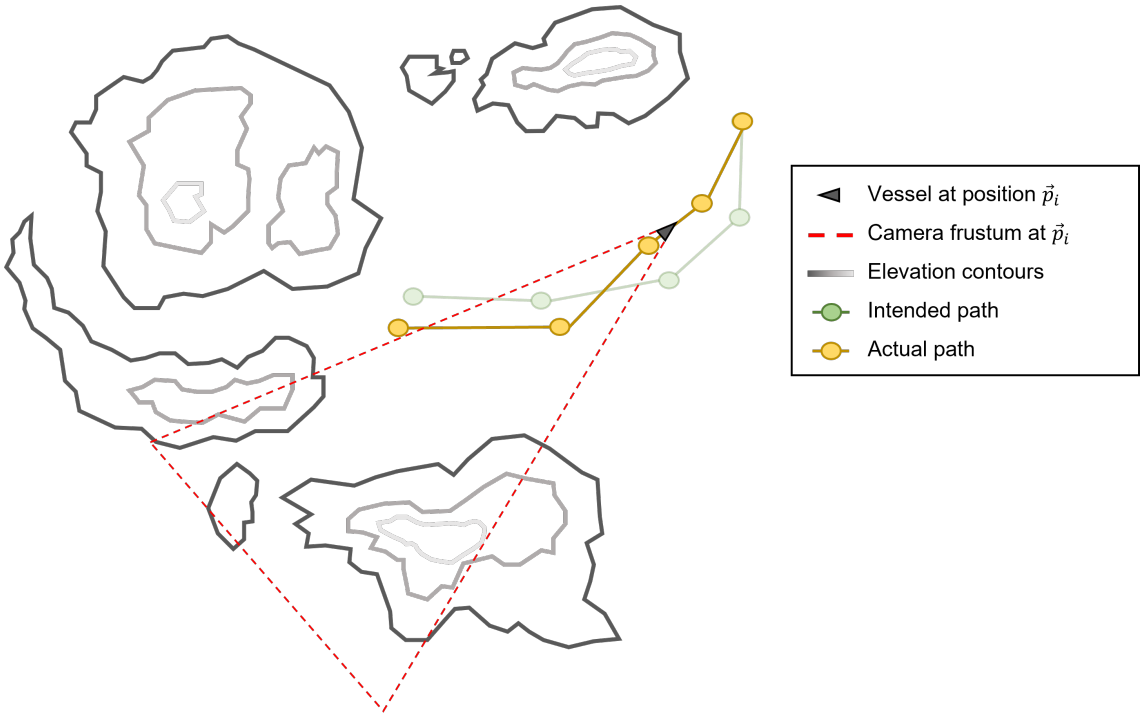


Figure 3: The assumed *actual* behaviour of a maritime vessel at each waypoint i. The intended behaviour is assumed disrupted by external factors, causing the vessel to drift away from the planned trajectory.

The proposed solution to estimate this deviation at each i is through the minimization of a well-poised optimization problem which has gained significant popularity within the scientific circles of CV the last decades: The reprojection error. The general concept is in many ways fairly simple: By assuming a rigid camera model and the surrounding terrain, a good estimate of the vessel's position $\hat{p}_i$ can be found through images taken at the intended and actual positions, respectively $\mathrm{I}_{\vec{\omega}_i}$ and $\mathrm{I}_{\vec{p}_i}$ for waypoint i. Figure 4 illustrate a set of plausible 16×9 images taken at the position of a maritime vessel in the intended and actual position illustrated in Figures 2 and 3 respectively. Note again how the headings $\theta_{\vec{\omega}_i}$ and $\theta_{\vec{p}_i}$ of the camera in both positions are solely dependent on the linear connection between the current and previous positions along both paths. The discretization on these images is significant for ease of conceptual illustration, but will not affect the algorithmic approach.

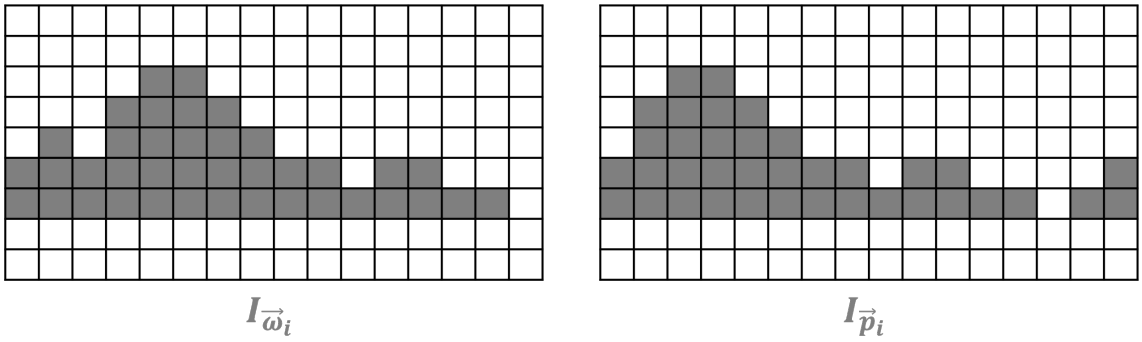


Figure 4: Possible 16×9 images capturing the shape of terrain from a maritime vessel with position and heading similar to those in Figure 2 and 3 in the illustrated terrain.

Here, both $\mathrm{I}_{\vec{\omega}_i}$ and $\mathrm{I}_{\vec{p}_i}$ are semantic segmentations of the terrain visible from the camera aboard the vessel at waypoint i, along the intended and actual path respectively. This means that all pixels in the image are assigned a class label: ***Gray*** pixels illustrating labeled terrain and *white*

pixels denoting anything else. Although this can be extended to classify and isolate more objects, like the ocean, vessels and obstacles, here it serves as a way to threshold the terrain from its surroundings. These semantic segmentations are what allow us to extract features with common CV techniques, enabling the knowledge-based estimation scheme. They are also becoming more and more accessible for real-time deployment, for example through the YOLO algorithms [37].

Given the importance of features in the proposed framework, they warrant a more thorough description. To formulate the nonlinear reprojection error, a set of $\mathcal{M} \geq 3$ features are required. The first $\mathcal{M}-1$ such features are located within $\mathcal{M}-1$ camera views, or image coordinates. For simplicity, these are denoted *image features*. These image features vary from one application to another, but always tend to highlight noticeable areas within the image. They can consist of one or more pixels encompassing an area of distinct illumination, geometry, edges ect. It is important that all these image features highlight **the same feature** across all $\mathcal{M}-1$ image features and are all of the same dimensions. For the proposed framework, there will be two image features $\vec{f}_{\vec{\omega}_i}$ and $\vec{f}_{\vec{p}_i}$ denoting a matching geometric area between the semantic segmentations $\mathrm{I}_{\vec{\omega}_i}$ and $\mathrm{I}_{\vec{p}_i}$. Figure 5 illustrate these for the running example. Both features will consist of $\mathcal{L}$ 2D pixel-coordinates denoting matching *hilltops* in the images, where hilltops are geometric, convex shapes in the terrain segmentation. $\vec{f}_{\vec{\omega}_i}$ and $\vec{f}_{\vec{p}_i}$ are highlighted as ***blue*** pixels in both images.

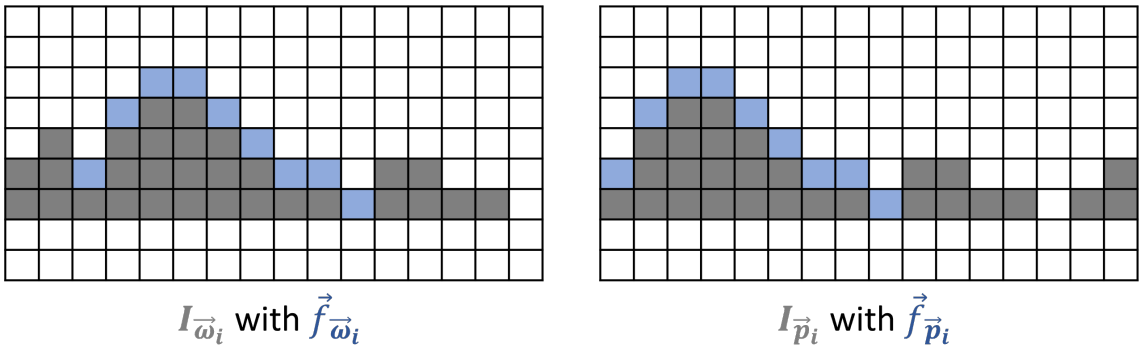


Figure 5: Example of extracted hilltop-contour from the images in Figure 4.

The last of the $\mathcal{M}$ required features is a set of $\mathcal{L}$ 3D coordinates in the DEM that are as close to a set of triangulated matches to the $\mathcal{M}-1$ image features as possible. In other words: The last feature $\vec{f}_{DEM_i}$ consist of the 3D world coordinates that, when projected onto all of the $\mathcal{M}-1$ camera views forming image features, align with said features for all views. Figures 6 and 7 attempt to illustrate briefly how $\vec{f}_{DEM_i}$ can be visualized and how it is related to matches in image features. Figure 6 shows that one of the $\mathcal{M}-1$ image features, in our case $\vec{f}_{\vec{\omega}_i}$, is used to locate $\mathcal{L}$ points in the DEM. Figure 7 builds on this, showing how the points in $\vec{f}_{DEM_i}$ have to align during projection into a pinhole camera model with the located image features, here $\vec{f}_{\vec{\omega}_i}$ and $\vec{f}_{\vec{p}_i}$.

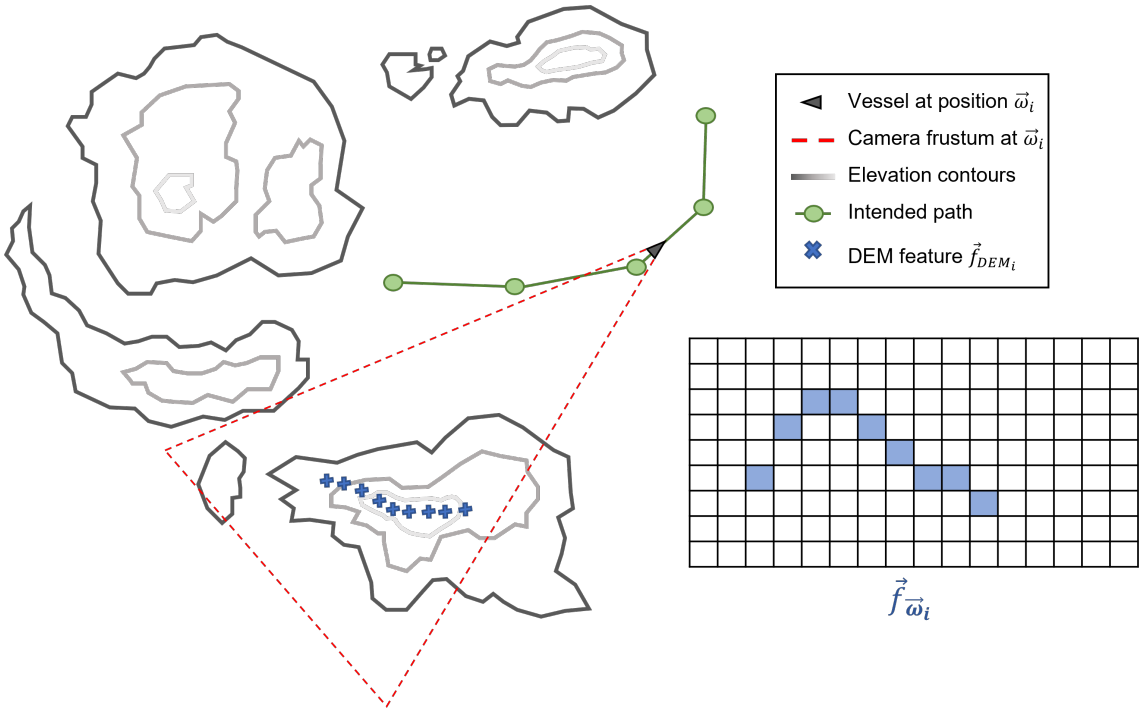


Figure 6: Example of $\mathcal{L}$ world points $\vec{f}_{DEM_i}$ found through a matching scheme with image feature $\vec{f}_{\vec{\omega}_i}$.

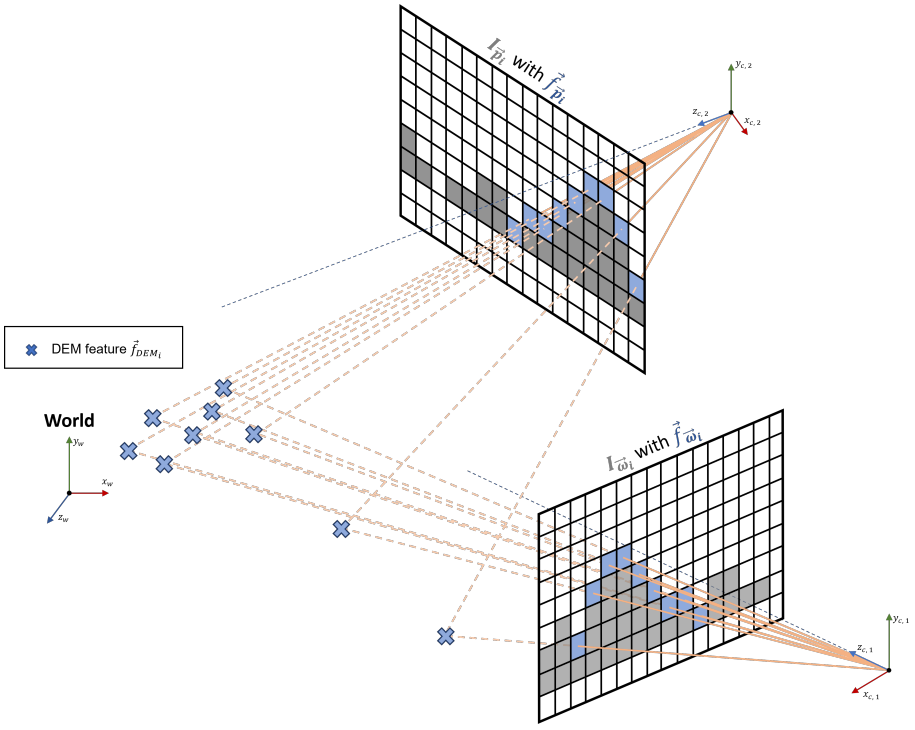


Figure 7: The alignment condition on features $\vec{f}_{DEM_i}$ to located image features $\vec{f}_{\vec{\omega}_i}$ and $\vec{f}_{\vec{p}_i}$. Note that the difference in heading between camera views is highly exaggerated. This also distorts the elements of $\vec{f}_{DEM_i}$.

All $\mathcal{M}$ features can be written on vector-form as in equation (1). $\vec{f}_I$ are here general image-features and $\vec{f}_{DEM}$ are DEM features:

$$\vec{f}_I = \begin{bmatrix} x_{I,0} & y_{I,0} \\ x_{I,1} & y_{I,1} \\ \vdots & \vdots \\ x_{I,\mathcal{L}-1} & y_{I,\mathcal{L}-1} \end{bmatrix}, \quad \vec{f}_{DEM} = \begin{bmatrix} x_{DEM,0} & y_{DEM,0} & z_{DEM,0} \\ x_{DEM,1} & y_{DEM,1} & z_{DEM,1} \\ \vdots & \vdots & \vdots \\ x_{DEM,\mathcal{L}-1} & y_{DEM,\mathcal{L}-1} & y_{DEM,\mathcal{L}-1} \end{bmatrix} \tag{1}$$

Finally, the optimization problem can be formulated using the 3 distinct features and a-priori information about the vessels position. The idea is, conceptually, quite simple: As we know the intended position $\vec{\omega}_i$ of a vessel at waypoint i, this can be used as an initial guess for the estimated position $\hat{p}_i$. The estimated heading, $\theta_{\hat{p}_i}$, is assumed implicitly known from the previous and estimated position along a piece-wise linear path, rendering it obsolete in estimation. From here, the 3D feature $\vec{f}_{DEM_i}$ is *reprojected* back to the camera on board the vessel at $\vec{\omega}_i$ and the distance between all $\mathcal{L}$ reprojected pixels and their corresponding matches in $\vec{f}_{\vec{p}_i}$ are measured numerically. This yields a numerical error e of the deviation from the actual position. Figure 8 attempts to show a visual representation of an unlikely, but illustrative, perfect estimation. The pose-estimator returns an estimation $\hat{p}_i$ of the actual deviation $\vec{p}_i$ after 2 iterations. Each iteration lessens the overall cost function by choosing intermediate positions who's reprojections of $\vec{f}_{DEM_i}$ are closer to $\vec{f}_{\vec{p}_i}$ than the last, eventually converging $\hat{p}_i = \vec{p}_i$ for waypoint i.

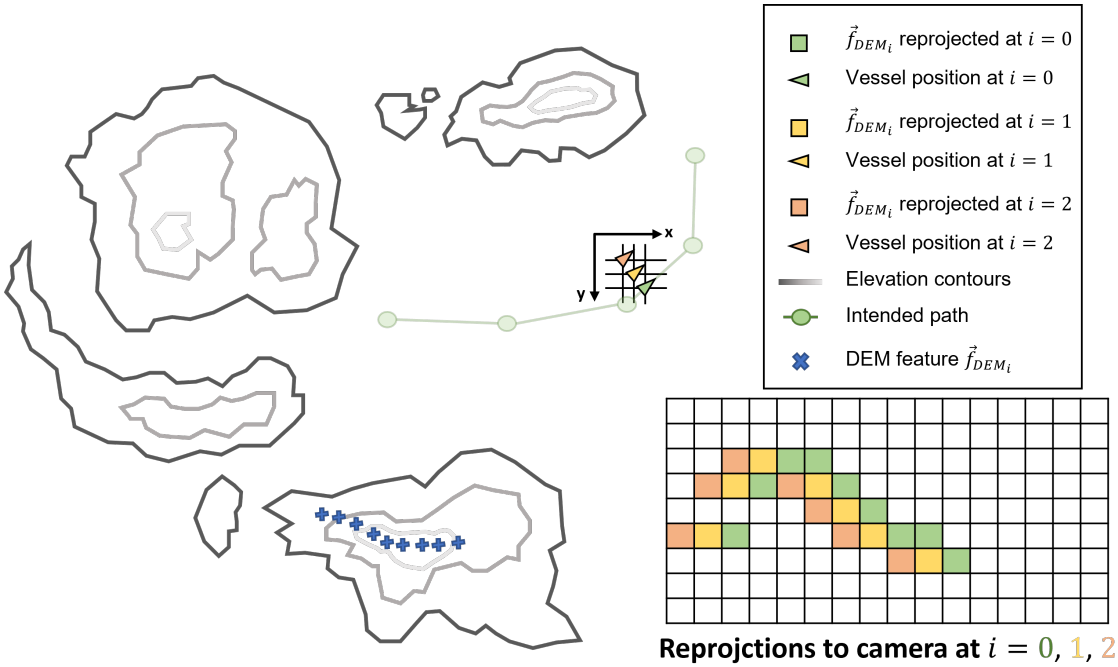


Figure 8: Example of plausible 3 DOF optimization, minimizing the euclidean distance between corresponding pixels of the reprojection of $\vec{f}_{DEM_i}$ and $\vec{f}_{\vec{p}_i}$.

After obtaining $\hat{p}_i$ at waypoint i, it is imperative to update the previous knowledge of the route, that is $\vec{\omega}_i = \hat{p}_i$. This implies an assumption that the maritime vessel estimates its deviation with some precision ($\hat{p}_i \simeq \vec{p}_i$), which yields the correct heading when computing the features of the next images: $\mathrm{I}_{\vec{\omega}_i+1}$ and $\mathrm{I}_{\vec{p}_i+1}$. Figure 9 illustrates how the intended path is updated when an estimation has been completed:

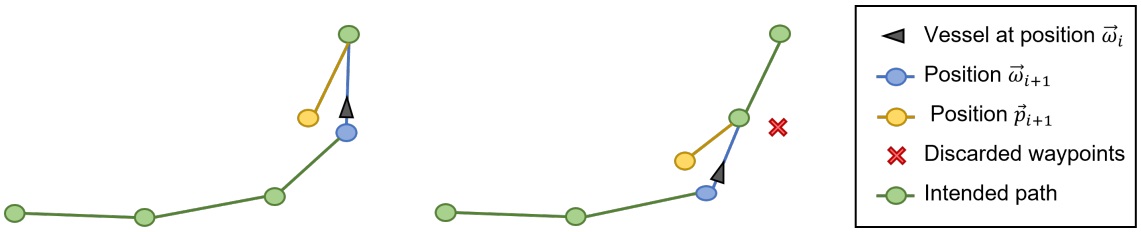


Figure 9: Update of a-priori information performed after each estimated waypoint.

3 Employed algorithms

This section is dedicated to the algorithms making up the presented framework. Each underlying subsection presents one or more algorithm(s) through a sequential description complete with figures and equations, before presenting a simplified pseudocode, working as both a summary and compressed template for repeatability.

3.1 Feature extraction and matching in camera images

The first subsection seeks to illustrate in detail the algorithms employed to extract and match features $\vec{f}_{\vec{\omega}_i}$ and $\vec{f}_{\vec{p}_i}$ in $\mathrm{I}_{\vec{\omega}_i}$ and $\mathrm{I}_{\vec{p}_i}$. Three algorithms will be presented to this end: One to accurately and consistently extract a skyline contour from semantic segmentations of terrain, one to extract predetermined features from such a skyline contour, and a final algorithm to match and align features between $\mathcal{M}-1$ camera views using the sum of absolute differences (SAD) algorithm:

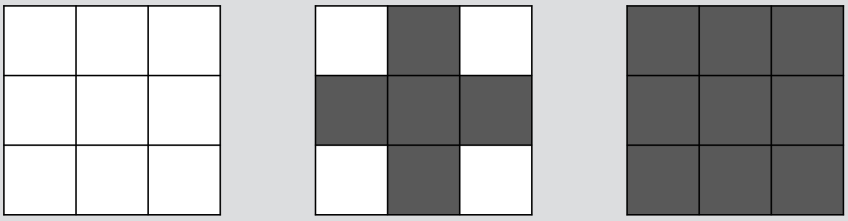
3.1.1 Extracting skyline contours from semantic segmentations

The skyline contour of a semantic segmentation of terrain is the highest pixel layer of the terrain intersecting visible sky. To extract this subset of the terrain, an algorithm based on morphological operations in image processing is employed. The full field of morphological operations, which is vast and relatively complicated, fall outside the scope of this report. However, a quick summary of the relevant operations employed by the algorithm follows below:

Morphological operations

The field of mathematical morphology encompasses numerous techniques applicable to the analysis and processing of various geometrical shapes, but in image processing it is more often than not synonymous with Binary Morphology on images, which presents a set of operations capable of extracting valuable information of shapes within a binary image. Denote a general, binary image as B. The methodology bases itself on a smaller, binary image known as a *structuring element*, denoted E, which is compared with the *neighborhood* around each pixel-center in B. The amount of overlap between binary values in B and E can be used to infer different bits and pieces of information, yielding a set of different operations.

Structuring elements: Examples of difference structuring elements E can be seen below. Note that this is a very small sample of possible elements, as they are not limited in size and contents and vary depending on the desired task:



0: 9
1: 0

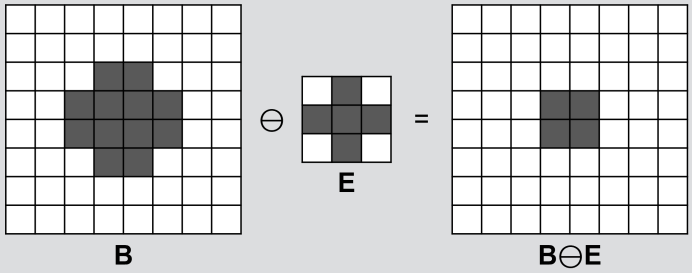
0: 4
1: 5

0: 0
1: 9

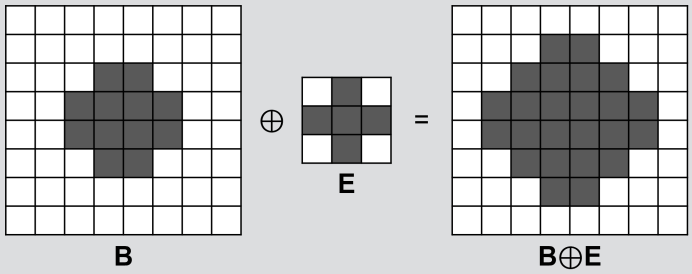
It should be noted that although the dimensions of all the above elements are the same, the amount of binary **True-False** values, or **1**'s and **0**'s, are vital to the output of the morphological operation. *Gray* elements are here binary **1**'s, whilst *white* elements are binary **0**'s, in line with the exemplary semantic segmentations in Section 2.

Binary Morphological Operations: In the subsequent framework, the following morphological operations were utilized:
The first two, *Erosion* and *Dilation*, are considered the most basic operations and are often used in combination to achieve more complex results:

- **Erosion ($\ominus$):** Erosion is performed by iterating a structuring element E over all pixels in the binary image B. The operation, denoted $B \ominus E$, results in the diminished circumference of all geometric shaped present in B. If, for a pixel $\mathbf{x}_i \in B$, all pixels in E that are **True** or **1** do not overlap pixels in B that are **True** or **1**, $\mathbf{x}_i$ is discarded. Although this can be described mathematically through set-theories, the concept is the easiest to grasp visually:



- **Dilation ($\oplus$):** Dilation is the inverse operation of Erosion, increasing the circumference of geometric shapes in B by iterating a structuring element E over all pixels $\mathbf{x}_i \in B$. If only one of the **True** values in E overlap a **True** value in B, the pixel $\mathbf{x}_i$ is also considered to be **True**. The operation, fittingly, has the opposite notation to the Erosion Operator $B \oplus E$ and can be visualized in a similar fashion:



The last two operations are composite operations, or rather operations made up of two or more basic operations:

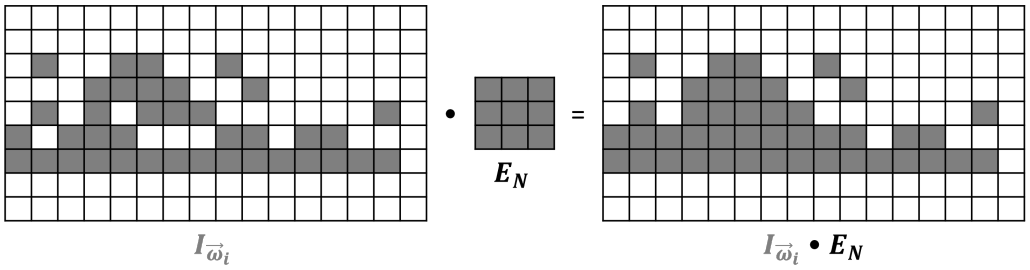
- **Opening:** A composite operation typically used to remove noise from *outside* larger geometric shapes in a binary image B. A structuring element E is first used to erode smaller shapes $B \ominus E$ before dilating remaining geometrical shapes to reconstruct their circumference $B \oplus E$. The composite operation is typically denoted:

$$B \circ E = (B \ominus E) \oplus E$$

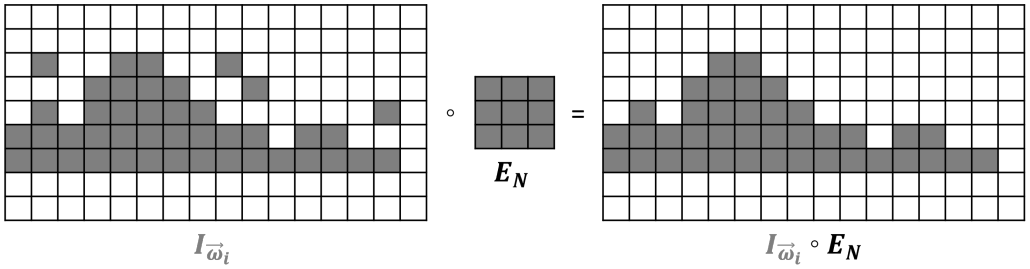
- **Closing:** A composite operation typically used to remove noise from *inside* larger geometric shapes in a binary image B. A structuring element E is first used to dilate the existing geometric shapes, filling minor discontinuities inside their hull $B \oplus E$, before eroding away the added circumference to reconstruct the original shapes of the geometric shapes $B \ominus E$. The composite operation is typically denoted:

$$B \bullet E = (B \oplus E) \ominus E$$

The following algorithm utilize these morphological operations in 5 fixed steps to consistently and precisely isolate the contour of a skyline in a semantic segmentation of terrain. The first two steps are attributed to reducing and removing noise potentially present in the segmented terrain images. Such noise can occur from a variety of external factors, such as scattering and weather phenomena, reflections of light on smoother surfaces, or internal phenomenons such as misclassifications by the employed segmentation algorithm or irregular bit-switches inside the camera encoder. Taking measures to remove noise, even when it is not present, is then an advisable step to improve the algorithms consistency throughout a wide range of possible scenarios. Figure 10 illustrate these two steps and their use of morphological opening and closing for noise-reduction/removal on the example terrain from Section 2:



(a) First step of the skyline contour extraction algorithm, designed to remove noise from inside visible terrain in the semantic segmentations through morphological closing.

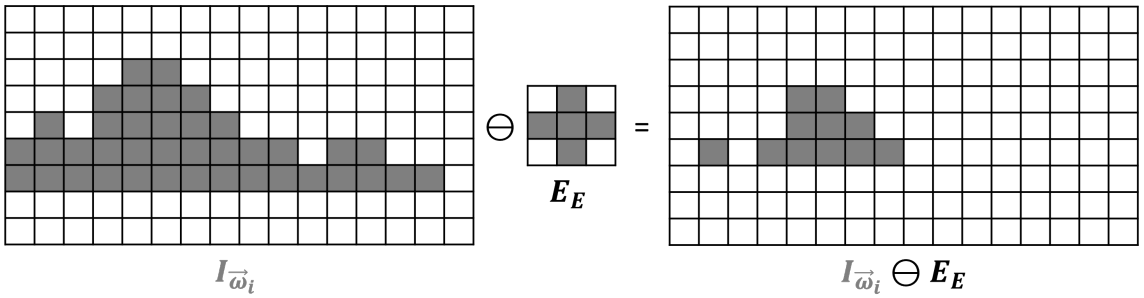


(b) Second step of the skyline contour extraction algorithm, designed to remove noise from outside terrain contours in the semantic segmentations through morphological opening.

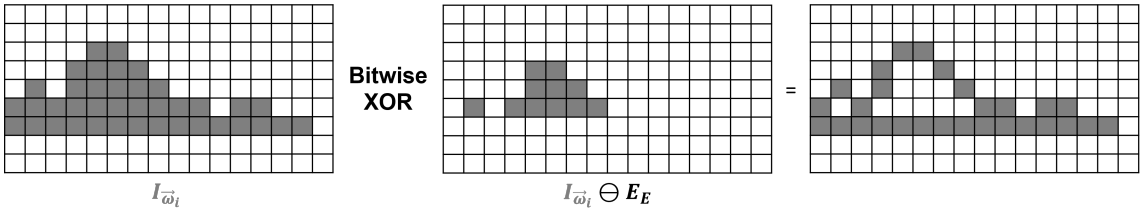
Figure 10: First two steps of the skyline contour extraction algorithm, designed to eliminate noise inside and outside visible terrain segmentations in a semantic segmentation.

Above, the structuring element E_N utilized in both operations is a 3×3 square matrix. Larger elements can be used and will eliminate greater instances of pixel-noise, but can affect the terrain by removing smaller parts of the segmented landmass in the same process. As noise tends to be smaller in nature and the ambiguity between larger noise and smaller landmasses grows with the size of the structuring element, this size was deemed the most reliable for the current application. In general the choice may vary based on a plethora of factors, such as the deployed camera parameters, expected external conditions and image-quality and aspect ratio, to mention a few.

The next two steps of the algorithm are illustrated in Figure 11 and seek to isolate the contour of the visible, segmented terrain by a masking scheme based on morphological erosion and bitwise operations:



(a) Step three of the skyline contour extraction algorithm, isolating the interior of the terrain contour through morphological erosion.



(b) Step four of the skyline contour extraction algorithm, extracting the contour of the terrain by masking the noise-reduced version of the skyline contour from steps 1-2 with the interior of the terrain contour from step 3 using a binary XOR operation.

Figure 11: Step three and four of the skyline contour extraction algorithm, isolating the contour of visible terrain in a semantic segmentation.

Notice the shape of E_E used in Figure 11a. This kind of cross-shaped 3×3 structuring element is used to ensure only one pixel on each side of a geometric shape is eroded away, contrary to the square 3×3 structuring element used in Figure 10a and 10b. These are more indiscriminate, especially around corners, and can result in elements further into the shape being discarded. If one such structuring element was utilized on the above $I_{\vec{\omega}_i}$, one would see the lone, leftmost pixel and all corner-pixels vanish, highlighting the importance of choosing suitable structuring elements when dealing with morphological operations.

The fourth step in Figure 11b outputs a complete contour of the terrain visible in the semantic segmentation, that is a skyline contour, distinguishable as the upper part of the contour, and a *coastline*: The intersection of the terrain with a planar surface, in this particular case the ocean. For the proposed framework, this coastline is unnecessary. Hence, the last step of the algorithm, shown in Figure 12, is a simple algorithmic procedure to remove the coastline, only keeping the skyline contour for feature extraction and matching:

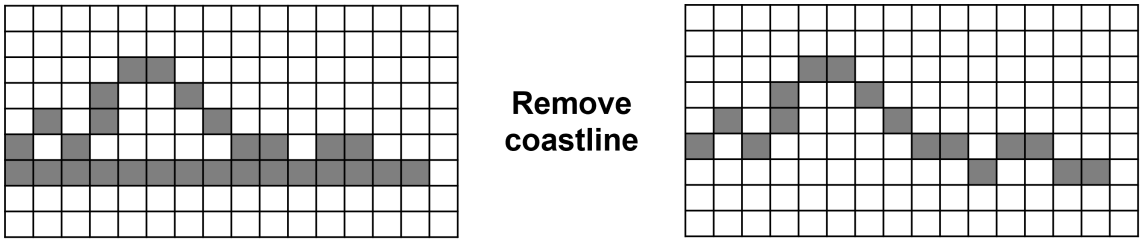


Figure 12: Step five of the skyline contour extraction algorithm, which erases the coastline from the complete terrain-contour.

The algorithm employed for this final step is fairly simple, comparing all pixels in the skyline of similar x-values and removing the one with the lowest y-value (assuming there are more than one). The reason removing the lowest pixel is preferred to only keeping the highest one is for cases in which a drastic gradient is present in the contour. The fourth column from the left is a good indication of such a scenario in the above figure. Here, only keeping the topmost-pixel

would render the skyline discontinuous, whilst removing the lowest one correctly outputs the target skyline contour.

A proposed pseudocode for the algorithm can be seen in Algorithm 1:

Algorithm 1 Extract Skyline Contour from Semantic Segmentation of terrain

Require: Semantic Segmentation of terrain I
  procedure SkylineContourExtraction(I)
    $I_b \leftarrow$ BinaryThreshold(I) ▷ Transform I into a binary image
    $\mathrm{E}_N \leftarrow$ StructuringElement((3 × 3), full)
    $I_b \leftarrow (I_b \bullet \mathrm{E}_N) \circ \mathrm{E}_N$ ▷ Remove noise
    $\mathrm{E}_E \leftarrow$ StructuringElement((3 × 3), cross)
    $I_{inside} = I_b \ominus \mathrm{E}_E$
    $C \leftarrow$ BITWISE_XOR(I_b, I_{inside}) ▷ Extract complete skyline contour
    for $x \in [0, \mathrm{len}(C_x)]$ **do** ▷ Remove cosatline
      $column\_values \leftarrow C[x, :]$
      C.Remove($[x, \mathrm{Min}(column\_values)]$)
    end for
    Return(C)
  end procedure

3.1.2 Feature extraction from skyline contours

With skyline contours extracted using Algorithm 1, the next step is to define an algorithm to extract features from one such skyline. Naturally, there are quite a few possible features one can choose to extract here, all the way from the single pixel, top-most extremities to the complete contour. In the proposed framework, a combination of these two are chosen: fully connected, continuous hilltops. These features are somewhat tougher to extract, being subject to quite a few limiting constraints, but are theoretically more robust where they are present. The top-most pixel extremities, or peaks visible in an image can often be ambiguous, as was the case in [39], whilst a complete skyline is bound to contain values on the edges of images. These can become a source of errornous matching from cameras with limited horizontal FOVs, due to the fact that hills or peaks past the image-border technically are unknown and can be misclassified as a result. Using heavily constrained hilltops should mitigate this error and subsequently related errors further down the pipeline. One should note that, as with single extremities, multiple hilltops can be extracted from the same skyline contour, given they satisfy the necessary constraints.

The constraints posed onto the extracted hilltops are as follows:

- Hilltops are defined to be sets of pixels forming convex hulls with negative semi-definite edges.
  - The convexity of the features imply that hills cannot be slopes, which would be highly ambiguous during matching of multiple hilltops.
- Hilltops cannot reach the edges of the images they are present in, as this can result in ambiguous or lacking features.
- Hilltops should be subject to minimum-thresholds on their widths t_w and heights t_h, ensuring more distinct features.
- No two hilltops, say H_1 and H_2, should have any overlapping pixels, that is $H_1 \cap H_2 = \emptyset$.

The proposed algorithm to extract a hilltop H from a skyline contour S is presented in Figure 13. The method contains a top-down, depth-first search of the supplied skyline contour, extracting one hilltop at a time and removing it from S by iterating from the top using a predefined *search area*. Extracted hilltops are kept if they adhere to the above conditions.

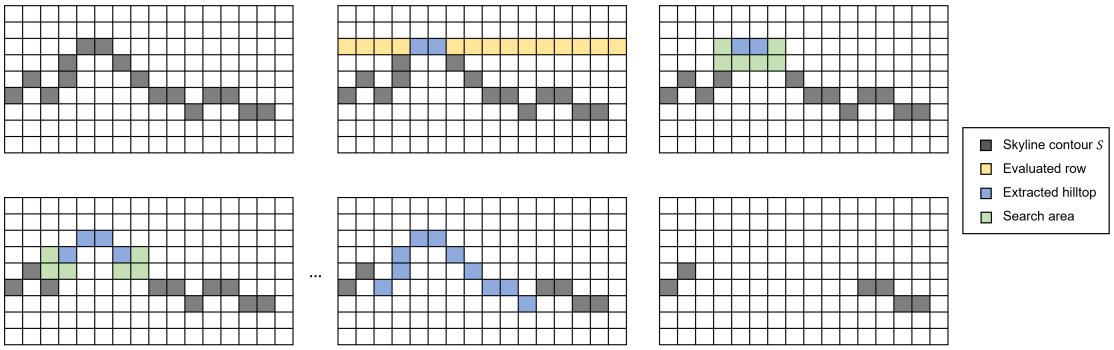


Figure 13: Algorithm to extract a hilltop H from a skyline contour S. One row of skyline elements are evaluated at a time. The algorithm divides them into smaller segments if clusters of elements occur at discontinuous column values. A *search area* moving down the contour in a negative semi-definite, depth-first fashion is then adopted, adding skyline elements to H until no further feasible elements can be located in either direction. Given the shape of the search area, the resulting hilltop H is guaranteed to be a convex hull. If H satisfies all conditions placed upon potential hilltops, it is removed from S and stored in a dictionary. The algorithm will then evaluate the next rows of S recursively, extracting new hilltops and deleting pixels as above, until no possible features remain.

A simplified pseudocode of the algorithmic procedure is presented in Algorithm 2. Given the many conditions on a hilltop and search-area specifics, this does not relay the precise extraction of hilltops as much as an overall workflow. For a full implementation, see the Python application programming interface (API) attached to this report.

Algorithm 2 Extract hilltop features from skyline contour

Require: Skyline contour S consisting of N pixels (x_i, y_i), $i \in [0, N)$
Require: Predetermined feature thresholds for minimum hilltop width t_w and height t_h
Require: Known image borders I_{min}, I_{max}
  procedure HILLTOPEXTRACTION(S, t_w, t_h, I_{min}, I_{max})
    $hilltops \leftarrow \{\}$
    for $row \in [S_{y,max}, S_{y,min}]$ **do**
      if nonzero(row) $= \emptyset$ **then**
        continue
      end if
      $row\_segments \leftarrow row$ divided into segments of continuous x-values
      for $segment \in row\_segments$ **do**
        Grow segment to a convex hull by adding descending, connected pixels present in s.
        $H \leftarrow$ Convex hull
        if $(H \cup I_{min} \neq \emptyset$ **or** $H \cup I_{max} \neq \emptyset)$ **then** ▷ Check if Hilltop overlap image borders
          continue
        end if
        if $H_{y,max} - H_{y,min} < t_h$ **then** ▷ Check if Hilltop is tall enough
          continue
        end if
        if $H_{x,max} - H_{x,min} < t_w$ **then** ▷ Check if Hilltop is wide enough
          continue
        end if
        $hilltops$.Add(H)
      end for
    end for
    Return($hilltops$)
  end procedure

3.1.3 Matching image features between multiple skyline contours

Utilizing Algorithms 1 and 2, it is possible to properly define a compound algorithm accepting two camera images of semantically segmented terrain, returning a set of optimally matching image features $\vec{f}_{\vec{\omega}_i}$ and $\vec{f}_{\vec{p}_i}$. The algorithm first finds a large subset of $\mathcal{K}$ possible hilltops in one image, before calculating which of these has the best match in the other. The similarity measure used is known as the sum of absolute differences (SAD) and is fairly known in the fields of signal processing and image processing:

Sum of Absolute Differences (SAD)

Often called Sum of Absolute Errors (SAE), SAD is an energy measure, like the famous Mean Square Error (MSE) and Mean Absolute Errors (MAE). The simplicity of the measure is often its biggest appeal, both from a theoretical and computational standpoint. It's use in image processing is predominantly to calculate the difference between different segments of images and its integration is quite straightforward [40] :

Suppose we have two images, one $N \times M, N > M$ image I_1 and a smaller $M \times M$ image I_2. To find the area in I_1 most similar to I_2, one can overlay I_1 with I_2 and calculate the deviation of overlapping elements between the two images. Summing these deviations for all pixels the $M \times M$ image, we get equation (2):

$$SAD = \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} |I_{1,(i,j)} - I_{2,(i,j)}| \tag{2}$$

Note that the above is a calculation in 2 dimensions. SAD is easily adaptable to any number of dimensions, like in this framework, where only 1 dimension is used. Calculating the SAD for all areas where I_2 can fully overlap I_1, we can find the best match to be the area of smallest error.

A quick visual example follows below to illustrate the concept in the case of two images with $N = 5 \& M = 3$. All areas not considered at the current iteration are marked "-":

12	7	9	1	5
10	8	6	3	7
9	7	4	4	10

I_1

10	2	4
5	5	6
4	3	8

I_2

2	5	5	-	-
5	3	0	-	-
5	3	4	-	-

i_0

-	3	7	3	-
-	3	1	3	-
-	3	1	4	-

i_1

-	-	1	1	1
-	-	1	2	1
-	-	0	1	2

i_2

For the three iterations $i_0 - i_2$ presented, the respective SAD values are $\text{SAD}_0 = 32$, $\text{SAD}_1 = 28$, $\text{SAD}_2 = 10$. Hence, the third area in I_1 is deemed the best match to I_2.

The presented algorithm works as follows: Given two skylines, $\text{S}_{\vec{\omega}_i}$ and $\text{S}_{\vec{p}_i}$ for the presented framework, $\mathcal{K}$ hilltops are extracted from $\text{S}_{\vec{\omega}_i}$ by Algorithm 2. All $\mathcal{K}$ hilltops are then matched with an area in $\text{S}_{\vec{p}_i}$, leaving $\mathcal{K}$ matching areas between $\text{S}_{\vec{\omega}_i}$ and $\text{S}_{\vec{p}_i}$. The match with the lowest total SAD score is determined the best and returned as the best possible image features $\vec{f}_{\vec{\omega}_i}$ and $\vec{f}_{\vec{p}_i}$. The matching between hilltops and skyline is done by a one-dimensional SAD measure, meaning

the error between two corresponding pixels is seen as the vertical difference between them. It is important to note that certain matches are seen as invalid, specifically those where the hilltop is matched with a discontinuous section of skyline. Figure 14 illustrates one iteration of the algorithm between a located hilltop H_0 from the intended skyline $S_{\vec{\omega}_i}$ and the actual skyline $S_{\vec{p}_i}$.

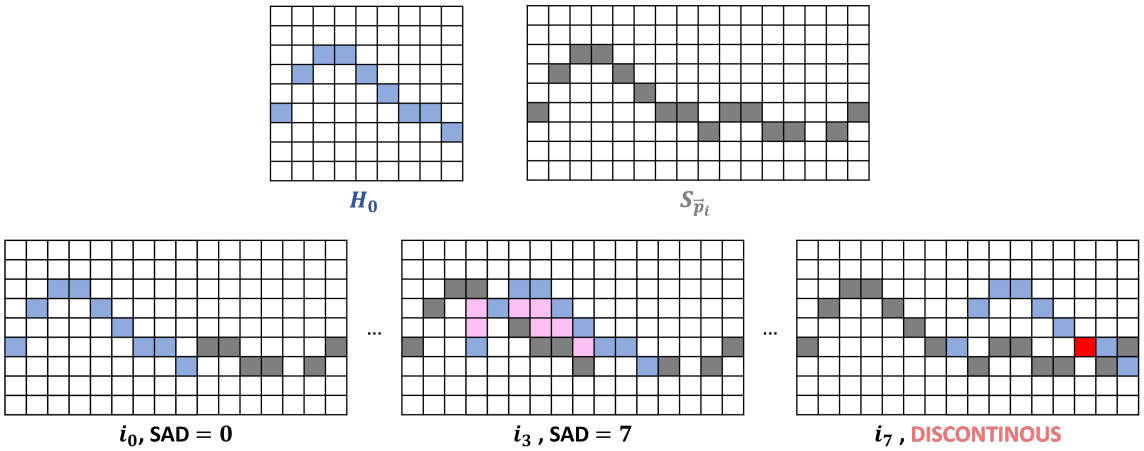


Figure 14: Matching algorithm deployed on an extracted hilltop H_0 from the intended skyline $S_{\vec{\omega}_i}$ and the actual skyline $S_{\vec{p}_i}$. The SAD between H_0 and $S_{\vec{p}_i}$ is calculated for each possible overlap, iterating from left to right. Highlighted iterations show a complete overlap in i_0, a partial overlap in i_3 and an infeasible matching in i_7. Infeasible matches occur where $S_{\vec{p}_i}$ is discontinuous in the considered area. The returned match between H_0 and $S_{\vec{p}_i}$ results in the smallest SAD.

An important preprocessing step here is to make sure the skylines are unique with regards to their column-values. This, at first glance, might seem contradictory to the removal of the coastline in Section 3.1.1, but should be considered a separate step entirely. Where it was imperative for the skyline to be as continuous as possible to successfully extract hilltops, it is more important when working with any similarity measures that all comparisons are unique and unambiguous. As the presented algorithm uses a one-dimensional SAD measure along each column, it is imperative that each column only has one value, although it might render the skylines or hilltops discontinuous. Figure 15 illustrates how this simple but important preprocessing is applied to the hilltop H_0 extracted from $S_{\vec{\omega}_i}$ and the entirety of $S_{\vec{p}_i}$ from the previous example.

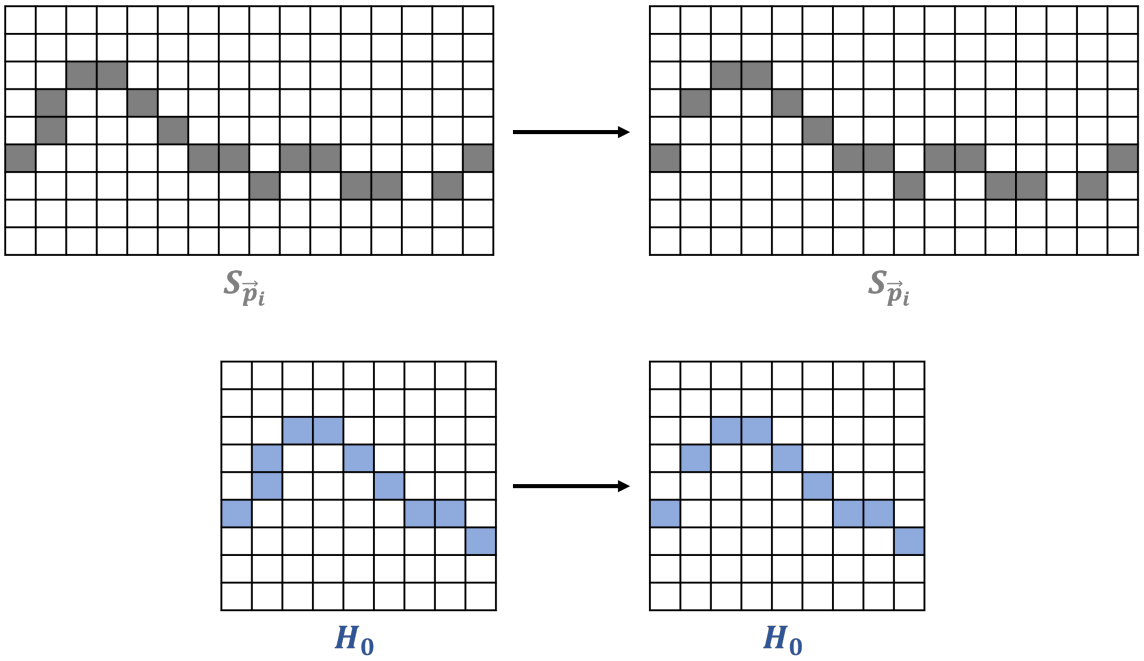


Figure 15: Necessary preprocessing of hilltop H_0 and skyline $S_{\vec{p}_i}$ before matching. If multiple elements exist for a column value x in either, the bottom value is removed to keep SAD computation unambiguous.

A simplified pseudocode for the presented algorithm can be seen in Algorithm 3:

Algorithm 3 Match Skyline Contours

Require: Two skyline contours: s_1 and s_2 consisting of N and M pixels.

```
procedure SkylineMatching(s_1, s_2)
    H ← HilltopExtraction(s_1, ...)                          ▷ Algorithm 2
    if H is empty then
        Terminate: No features in skyline.
    end if
    best_match_H ← []
    best_match_S2 ← []
    best_SAD ← ∞
    for H_i ∈ H do                          ▷ Iterate through all located Hilltops
        j ← 0
        while j ≤ len(s_2) − len(H_i) do    ▷ Iterate through overlaps of H_i and s_2
            s_{2,overlap} ← s_2[j : j + len(H_i), :]
            if s_{2,overlap} discontinuous then
                j ← first index after discontinuity
                Continue
            else
                SAD ← SAD(H_i, s_{2,overlap})
                if SAD < best_SAD then
                    best_SAD ← SAD
                    best_match_H ← H_i
                    best_match_S2 ← s_{2,overlap}
                end if
                j += 1
            end if
        end while
    end for
    return best_match_H, best_match_S2
end procedure
```

3.2 Locating skyline elements in DEM data

This subsection seeks to present the algorithm(s) employed in the framework to extract the feature from the DEM, $\vec{f}_{DEM_i}$. Due to the interwoven complexity of the algorithm, encompassing many other smaller sub-problems and their respective solutions, a divide and conquer approach is employed, dividing the problem into a set of smaller algorithms to simplify the final implementation. This makes the overall system more modular and easier to review in case of erroneous responses.

Before getting into the details, it is important to get a sufficient overview of the proposed solution. The idea is fairly simple: To extract a set of $\mathcal{L}$, 3D positions in world coordinates corresponding to all $\mathcal{L}$ pixels of all $\mathcal{M}-1$ image features. Although this seems daunting at first, it is important to remember that if one assumes the output of Algorithm 3 to yield $\mathcal{M}-1$ correctly corresponding image features of length $\mathcal{L}$, finding a set of 3D positions matching only one of the image features is equivalent to finding a set matching all. The presented algorithm does this for the feature $\vec{f}_{\vec{\omega}_i}$ captured at the intended position $\vec{\omega}_i$ of the vessel at waypoint i. This is the feature containing the most a-priori information, as one can assume both the position (and subsequently heading) of the vessel to be known. $\vec{f}_{DEM_i}$ will then contain $\mathcal{L}$ 3D positions corresponding to the $\mathcal{L}$ 2D pixels in feature $\vec{f}_{\vec{\omega}_i}$, and, by extension, $\vec{f}_{\vec{p}_i}$.

Figure 16 illustrates how $\vec{f}_{\vec{\omega}_i}$ is utilized in the algorithm to find $\vec{f}_{DEM_i}$ For each of the $\mathcal{L}$ pixels in $\vec{f}_{\vec{\omega}_i}$, a ray $\vec{R}_j$, $j \in [0\ \mathcal{L})$ is drawn from the intended position $\vec{\omega}_i$ through the pixel position on the image plane, into the DEM. The heading of each ray is calculated as a displacement from the known heading of the vessel, where the displacement comes from a division of the known camera frustum into $\mathcal{N}_w$ angular deviations. $\mathcal{N}_w$ is the total number of columns in image $\mathrm{I}_{\vec{\omega}_i}$ and the division is done by dividing the far render plane of the frustum into $\mathcal{N}_w-1$ equidistant lengths,

each representing a pixel on the image plane. Along each of these rays, a cross-section of the DEM is extracted through a 2D line-drawing algorithm and the point in this cross-section with the greatest angle is returned as the skyline-element in $\vec{f}_{DEM_i}$.

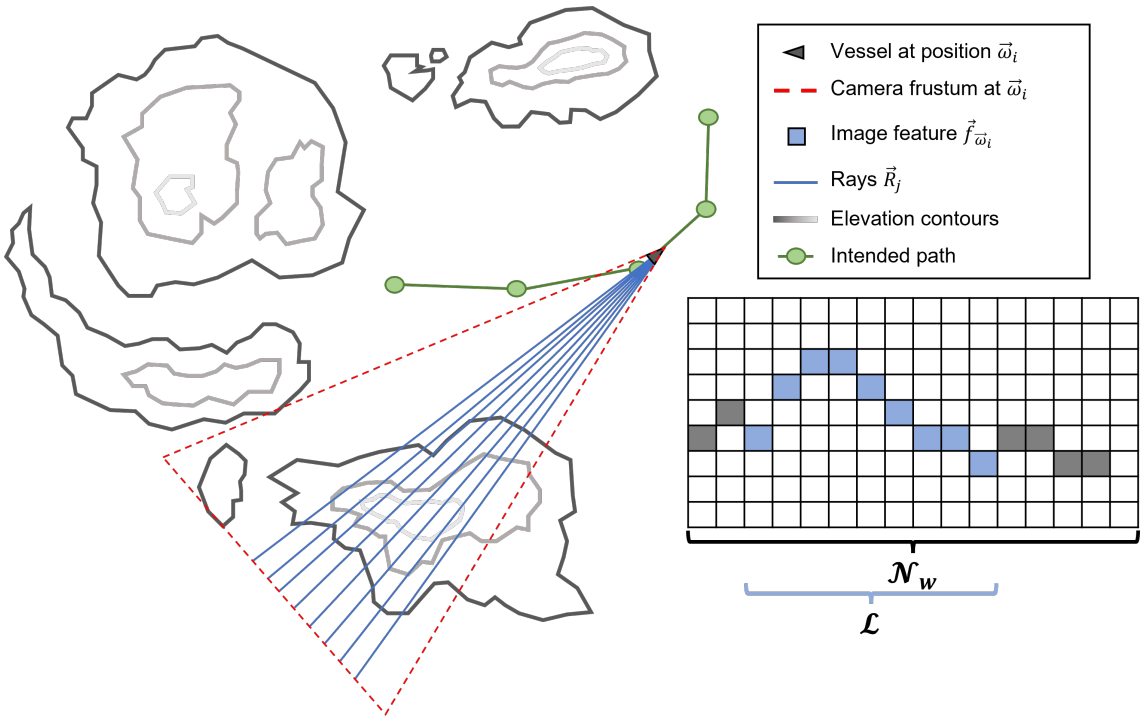


Figure 16: Overview of the DEM feature extraction algorithm. The feature $\vec{f}_{\vec{\omega}_i}$ consisting of $\mathcal{L}$ pixels, captured at $\vec{\omega}_i$, is projected into the DEM by $\mathcal{L}$ rays in the forms of 2D lines.

The simplified pseudocode of the complete algorithm can be seen in Algorithm 4. The rest of this subsection will be devoted to the modules making up this algorithm, namely an algorithm to divide the camera frustum into $\mathcal{N}_w$ rays, one to draw a ray $\vec{R}_j$ as a 2D line to the edge of the DEM and one to extract elements along $\vec{R}_j$ and add the correct feature to $\vec{f}_{DEM_i}$.

Algorithm 4 Extract $f_{i,DEM}$

Require: DEM I_{DEM}
Require: Image feature $\vec{f}_{\vec{\omega}_i}$ from intended skyline contour $S_{\vec{\omega}_i}$
Require: Intended position at waypoint i $\vec{\omega}_i$
Require: Actual position at last waypoint $i-1$ $\vec{p}_{i-1}$
Require: The horizontal span of the camera-frustum in degrees FOV_H
Require: The amount of columns in $I_{\vec{\omega}_i}$, $\mathcal{N}_w$
  procedure ExtractDEMFeature
    $dem\_feature \leftarrow []$
    $directional\_vector \leftarrow \vec{\omega}_i$ - $\vec{p}_{i-1}$
    $\theta_{\vec{\omega}_i} \leftarrow atan2(directional\_vector[1], directional\_vector[0])$ ▷ Calculate heading of vessel
    $angles \leftarrow DivideCameraFrustum(\mathcal{N}_w, FOV_H, \theta_{\vec{\omega}_i})$
    for j $\in(0, \mathcal{N}_w]$ **do**
      if j $\notin \vec{f}_{\vec{\omega}_i x}$ **then**
        continue ▷ Skip elements not in $\vec{f}_{\vec{\omega}_i}$
      end if
      $\alpha \leftarrow angles$[i]
      $\vec{R}_j \leftarrow DrawLine2D(\vec{\omega}_i$, ProjectRayToDEMBorder($I_{DEM}$, $\vec{\omega}_i$, α)) ▷ Draw $\vec{R}_j$
      $dem\_feature$.append(ExtractSkylineElementFromRay($\vec{R}_j$, I_{DEM})) ▷ Add $\vec{R}_j$ to $\vec{f}_{DEM_i}$
    end for
    return $dem\_feature$
  end procedure

3.2.1 Division of camera frustum

To find $\mathcal{L}$ rays matching each 2D pixel coordinate in $\vec{f}_{\vec{\omega}_i}$ it is important to understand how each ray will differ from one another. Primarily, each ray will have its origin on $\vec{\omega}_i$, face a different heading from other rays and terminate at the spatial border of the DEM. This subsection primarily deals with the headings of these rays, attempting to divide the **entirety** of the camera frustum's horizontal FOV FOV_H into $\mathcal{N}_w$ equal parts: each one corresponding to a column in the image $I_{\vec{\omega}_i}$. This is important, as not just the headings for the matching rays, but for all possible rays in the image are calculated. The reason for this will be explained, but introductory it should be sufficient to note that this is imperative to accurately compute the desired $\mathcal{L}$ rays.

The general strategy deployed is based on the geometrical interpretation of a triangular camera-frustum (for the pinhole camera model) and how it is geometrically similar to a smaller triangle that can be drawn from the camera center to the image plane. Figure 17 illustrates this, with a red triangle indicating the full camera frustum and a blue, similar triangle formed with the image plane a distance f from the camera center. It should be noted here that all rays are drawn at the same height h of the maritime vessel. This is intentional, as matching the height keeps the rays strictly horizontal, making them ideal for extracting cross-sections of the DEM at a later stage.

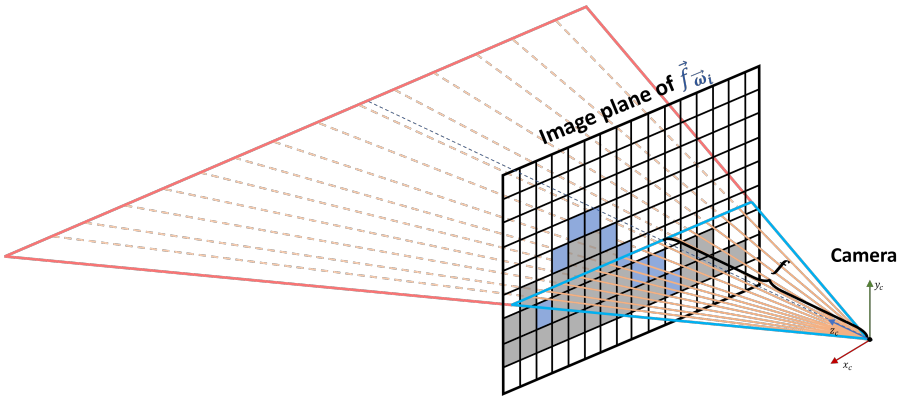


Figure 17: Geometrical interpretation of similar triangles within a camera frustum, fixed at the height of the camera h. The cross-section of the camera frustum at this height, drawn in red, goes from the camera center, through the borders of the image plane and reaches until the far-plane of the frustum. A geometrically similar triangle, shown in blue, can be formed with the image plane as the far side of the triangle. The image plane is imagined to be a length f, where f denotes the camera focal length, from the camera center.

Analyzing Figure 17 one can make a few important observations. Firstly, the two triangles formed both have a horizontal, angular span FOV_H at the image center. Secondly, the $\mathcal{N}_w$ angular headings calculated can be visualized as rays of their own, given that they are just headings in two dimensions from the camera center. As these intersect the image plane through the pixel centers of each column in $\text{I}_{\vec{\omega}_i}$, the distances between these rays on the far render plane have to be *equidistant*. This is an important assumption. Thirdly, the $\mathcal{N}_w$ rays drawn generate $\mathcal{N}_w{-}1$ smaller triangles with their own horizontal, angular spans from the camera center. These are denoted θ_m, where $m \in [1, \mathcal{N}_w]$ and are *not necessarily equivalent*. That is, $\theta_m \not\equiv \theta_{m+1}$. This is the reason all $\mathcal{N}_w$ headings have to be calculated, as subsequent angles are dependent on previous ones, which are not necessarily constant. Finally, given the geometric similarity between the two triangles, one should note that all angles θ_m are equivalent for any distance f, allowing it to be chosen arbitrarily.

With all of these observations, it is possible to discern an algorithmic approach based purely on geometric and trigonometric interpretations of triangles. Figure 18 illustrates a geometrically similar triangle formed within a camera frustum of horizontal, angular span FOV_H with $\mathcal{N}_w{=}\ 4$ headings prefixed by a letter d and $\mathcal{N}_w{-}1\ =\ 3$ angular deviations θ_m. In line with previous observations, one can safely normalize the focal length $f{=}\ 1$ and assume an equidistant partitioning of the one-dimensional distance between all headings, called l.

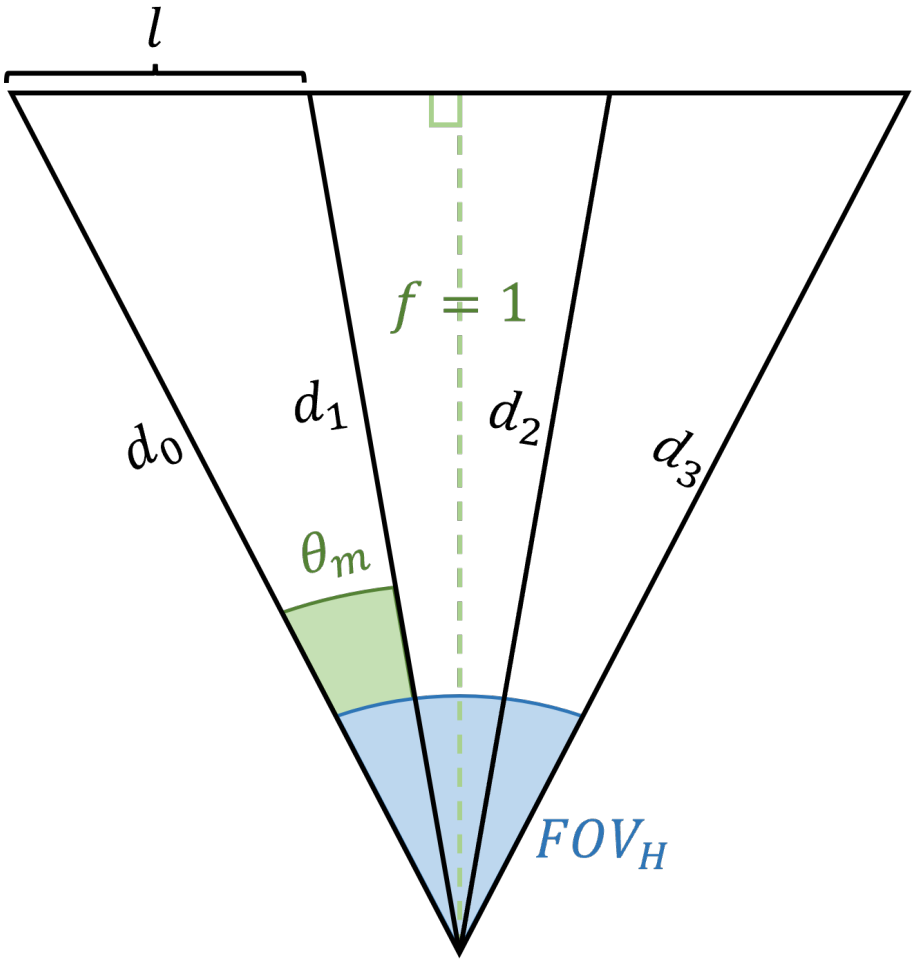


Figure 18: Overview of the camera frustum subdivision problem, with headings d, and angular deviations $\theta_m, m \in [1, \mathcal{N}_w]$. The camera has a total horizontal FOV FOV_H and the focal length f is normalized to 1. l is the equidistant, one-dimensional length between headings on the far plane.

The proposed subdivision makes each desired angle θ_m part of a smaller, not necessarily right, triangle. To isolate the angle, the generalized Pythagorean equation, also known as the *Law of cosines*, can be utilized:

$$c^2 = a^2 + b^2 - 2abcos(\theta) \tag{3}$$

Here, c is the opposite side of the angle θ, making a and b adjacent. To solve for θ_m in Figure 18, one can substitute $c = l$, $a = d_{m-1}$ and $b = d_m$ before solving for θ_m:

$$\begin{aligned} l^2 &= d_{m-1}^2 + d_m^2 - 2d_{m-1}d_m cos(\theta_m) \\ 2d_{m-1}d_m cos(\theta_m) &= d_{m-1}^2 + d_m^2 - l^2 \\ \theta_m &= cos^{-1}\left(\frac{d_{m-1}^2 + d_m^2 - l^2}{2d_{m-1}d_m}\right) \end{aligned} \tag{4}$$

When solving equation (4), it is desirable to find closed-form solutions to the expressions d_{m-1}, d_m and l. Using the fact that a camera frustum always forms an isosceles triangle and that the focal length f intersects the image plane orthogonally in the center, l can be calculated by dividing the entire length of the normalized image plane and simply divide it into $\mathcal{N}_w - 1$ equidistant parts. The horizontal length of the normalized image plane can be found as twice the opposing cathode of a right triangle formed by f and the frustum borders:

$$
\begin{aligned}
\mathrm{w}_{normalized\_image\_plane} &= 2 \cdot (tan(\frac{\mathrm{FOV}_H}{2})f) \\
&= 2 \cdot tan(\frac{\mathrm{FOV}_H}{2})
\end{aligned}
$$

Which enables the closed-form solution in equation (5):

$$
l = \frac{\mathrm{w}_{normalized\_image\_plane}}{\mathcal{N}_w - 1} = \frac{2 \cdot tan(\frac{\mathrm{FOV}_H}{2})}{\mathcal{N}_w - 1} \tag{5}
$$

The sides d_{m-1} and d_m of the triangle formed by an angle θ_m to the camera center can also be found by using Pythagoras on a right triangle formed by f. The tricky part for these is finding the far-side of the triangles, for which it is important to assess the direction one iterates through the lines within the frustum. In this report, this iteration is denoted to be *clockwise*, meaning d_m will be further to the right than d_{m-1} in Figure 18. Figure 19 then illustrates the proposed strategy to find d_{m-1} and d_m as the hypotenuses of their respective right-triangles for $m = 2$ in the example proposed in Figure 18:

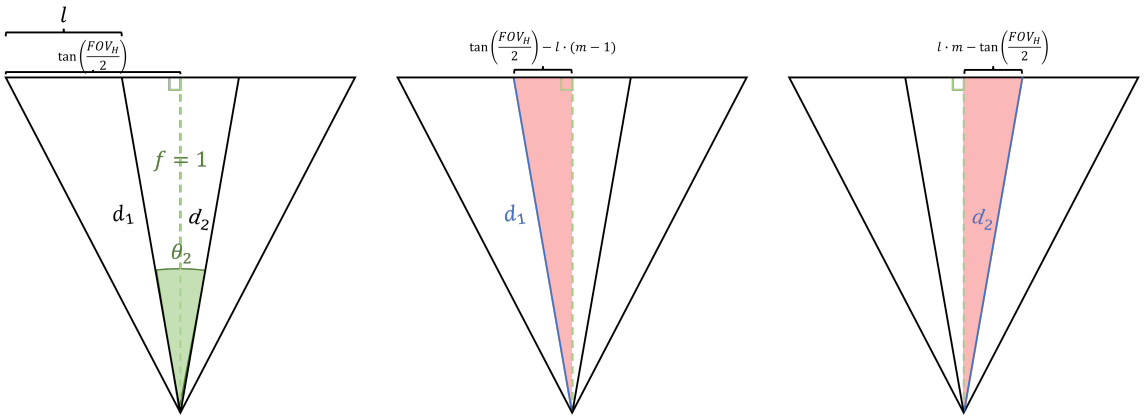


Figure 19: Algorithmic approach to calculate the sides d_{m-1}, d_m of a triangle m. Both sides are considered hypotenuses of their respective right-angle triangles formed with the normalized focal length f. The far cathode for each side is dependent on whether $l \cdot (m-1)$ and $l \cdot m$ are greater than or smaller than the half point of the opposing frustum side $tan(\frac{\mathrm{FOV}_H}{2})$. With all cathodes, d_{m-1} and d_m can be calculated by Pythagoras rule.

d_{m-1} and d_m are visibly subject to a condition: Whether they intersect the normalized image plane on the left or right side of its halfway point. Keeping in mind that this border-condition is the length $tan(\frac{\mathrm{FOV}_H}{2})$, a set of closed-form equations for the line of each side in triangle m can be calculated:

$$
d_{m-1} = \begin{cases} \sqrt{1 + (tan(\frac{\mathrm{FOV}_H}{2}) - l \cdot (m-1))^2}, & \text{if } l \cdot (m-1) \leq tan(\frac{\mathrm{FOV}_H}{2}) \\ \sqrt{1 + (l \cdot (m-1) - tan(\frac{\mathrm{FOV}_H}{2}))^2}, & \text{otherwise} \end{cases} \tag{6}
$$

$$
d_m = \begin{cases} \sqrt{1 + (tan(\frac{\mathrm{FOV}_H}{2}) - l \cdot m)^2}, & \text{if } l \cdot m \leq tan(\frac{\mathrm{FOV}_H}{2}) \\ \sqrt{1 + (l \cdot m - tan(\frac{\mathrm{FOV}_H}{2}))^2}, & \text{otherwise} \end{cases} \tag{7}
$$

Inserting Equations (7), (6) and (5) into (4), one can compute the $\mathcal{N}_w - 1$ angular deviations between all $\mathcal{N}_w$ headings in the frustum for an equidistant division of the horizontal length of the normalized image plane. Assuming the maritime vessel has the known heading $\theta_{\vec{\omega}_i}$ at $\vec{\omega}_i$, the

deviations can be added to this recursively to displace them correctly with respect to the vessels heading. A simplified psuedocode for the entire algorithmic procedure can be seen in Algorithm 5:

Algorithm 5 Divide camera frustum

Require: Predefined amount of subdivisions $\mathcal{N}_w$
Require: The horizontal span of the camera-frustum in degrees FOV_H
Require: The initial heading of the camera $\theta_{\vec{\omega}_i}$
  procedure DIVIDECAMERAFRUSUTM($\mathcal{N}_w$, FOV_H, $\theta_{\vec{\omega}_i}$)
    $angles \leftarrow []$
    $half\_fov \leftarrow \frac{\text{FOV}_H}{2}$
    $angles[0] \leftarrow \theta_{\vec{\omega}_i} - half\_fov$
    $l \leftarrow Calculate\_l()$
    for $m \in [1, N]$ **do**
      $l_{m-1} \leftarrow l \times (m-1)$
      $l_m \leftarrow l \times m$
      $d_{m-1} \leftarrow CaculateLine(l_{m-1}, half\_fov)$
      $d_m \leftarrow CaculateLine(l_m, half\_fov)$
      $angles[m] \leftarrow CosineLaw(l, d_{m-1}, d_m) + angles[m-1]$
    end for
    Return *angles*
  end procedure

3.2.2 Drawing a ray $\vec{R}_j$ in the DEM

Assuming the headings of the $\mathcal{L}$ desired rays $\vec{R}_j$ $j \in [0\ \mathcal{L})$ are extracted from Algorithm 5, the next step is to isolate the DEM along these rays, which will be imperative later on to extract a cross-section for evaluation. For this, two fairly simple subproblems require solutions, which should be covered for reproduceability and completeness.

Firstly, to isolate the DEM across a ray $\vec{R}_j$, the ray needs to be calculated as a set of points from which a cross-section can be retrieved. As described in Section 3.2.1, the rays are drawn perpendicular to the groundplane (as they intersect the image plane at height h), meaning a 2D line drawing algorithm should be sufficient to retrieve a set of 2D coordinates projecting this ray onto the DEM. The most well known algorithm is the original 1965 implementation of J. E. Bresenham [41], which holds up well to this day, but there are plenty other good implementation yielding similar results with various improvements. For this framework, an implementation by John Clark Craig of Medium is adopted [42]. His solution is based on Bresenhams algorithm, but utilizes some clever tweaking of the math to cover the many edge-cases in 2D line drawing in a shorter, albeit equally accurate and quick solution. The resulting lines are, for all intents and purposes, identical to those calculated by Bresenham; only experiencing shifts in a single pixel on diagonal segments.

Secondly: For any line-rendering algorithm, it is imperative to denote a start and end-point of which to draw the line between. In the presented implementation the initial point will always be the intended position $\vec{\omega}_i$, whilst the heading of ray $\vec{R}_j$, which can be denoted θ_j, is calculated in Algorithm 5. The end-point $\vec{p}_{end}$, however, proves more challenging. Although trivial on visual inspection, it is tricky to find $\vec{p}_{end}$ in a closed-form manner, given the many possible combinations of starting points, headings and spatial borders of the DEM. To this end, a fairly simple but efficient iterative algorithm is presented, localizing a sufficient approximation of $\vec{p}_{end}$ by recursively adding unit-vectors of $\vec{R}_j$, $\hat{r}_j$, onto a compound vector until a bound of the DEM is breached. Figure 20 illustrates this for an arbitrary starting point $\vec{\omega}_i$ and heading θ_j of $\vec{R}_j$:

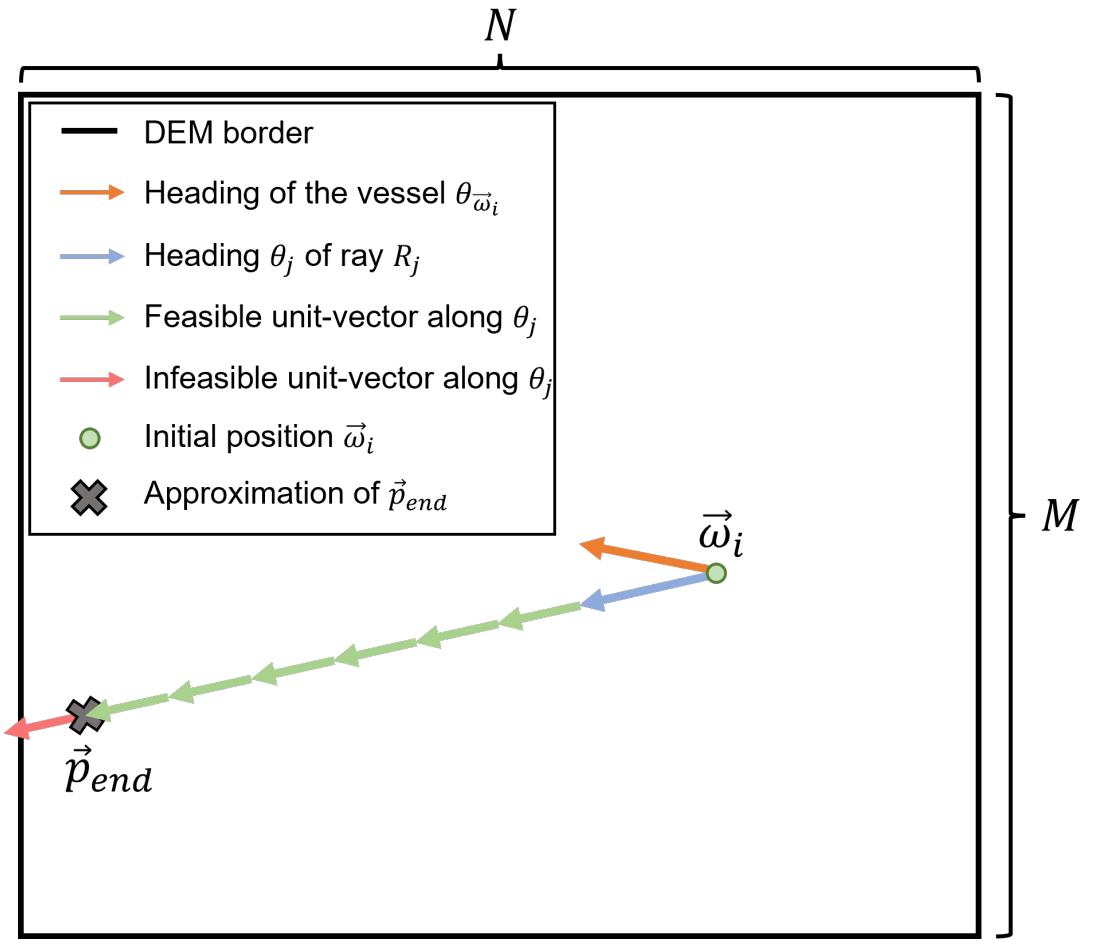


Figure 20: Algorithm to approximate the 2D intersection $\vec{p}_{end}$ of the ray $\vec{R}_j$ with the border of the DEM. A unit vector along the heading θ_j of $\vec{R}_j$, $\hat{r}_j$, is added onto $\vec{p}_{end}$ until $\vec{p}_{end}$ breaks one of the spatial constraints of the DEM. The last feasible value of $\vec{p}_{end}$ is returned as the approximated border-value.

A quick, simplified pseudocode for this solution can be seen in Algorithm 6

Algorithm 6 Approximate ray projection to DEM border

Require: DEM I_{DEM}
Require: Ray initial position $\vec{\omega}_i$.
Require: Ray heading θ_j.
  procedure ProjectRayToDEMBorder(I_{DEM}, $\vec{\omega}_i$, θ_j)
    $\hat{r}_j \leftarrow [cos(\theta_j), sin(\theta_j)]$ ▷ Unit vector in direction θ_j
    $\vec{p}_{end} \leftarrow \vec{\omega}_i$
    $ouf\_of\_bounds \leftarrow$ **False**
    while !ouf_of_bounds **do**
      if $\vec{p}_{end}$ outside borders of I_{DEM} **then**
        $\vec{p}_{end} -= \hat{r}_j$
        $ouf\_of\_bounds \leftarrow$ **True**
      else
        $\vec{p}_{end} += \hat{r}_j$
      end if
    end while
    return $\vec{p}_{end}$
  end procedure

3.2.3 Extraction of skyline element $\vec{f}_{DEM_i,j}$ along ray $\vec{R}_j$

Using Algorithm 6 and the proposed 2D line-drawing algorithm of Section 3.2.2, a cross section of the DEM can be extracted along $\vec{R}_j$ by masking the DEM with the resulting line. This will essentially isolate 2D coordinates along the groundplane of the DEM corresponding to the evaluated column in $\mathrm{I}_{\vec{\omega}_i}$. Using this cross-section, the 3D coordinate inside the DEM, corresponding to a 2D pixel on the matched hilltop $\vec{f}_{\vec{\omega}_i}$, can be isolated. The approached proposed in this report first reduces the dimension of the 3D cross section to two dimensions, of which the point with the greatest angle from origo, which equals $\vec{\omega}_i$, is returned. Figure 21 illustrates the proposed solution for a calculated ray $\vec{R}_j$ inside the DEM.

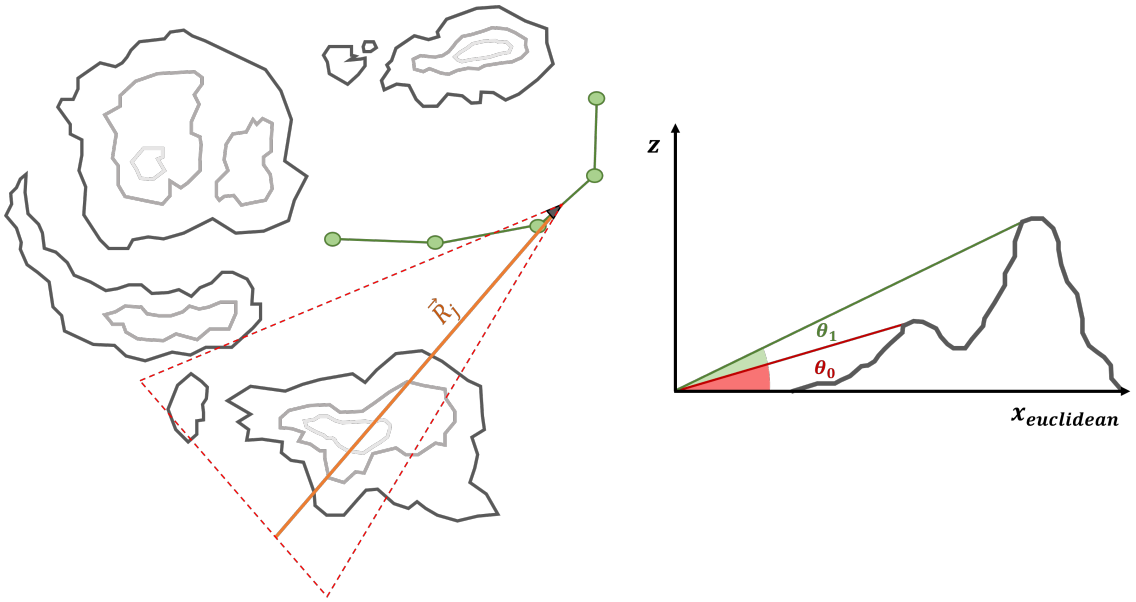


Figure 21: Algorithmic approach to find the DEM feature along $\vec{R}_j$, $\vec{f}_{DEM_i,j}$. A cross-section of the DEM is extracted along R_j and projected onto a 2D plane with horizontal axis $x_{euclidean}$: The euclidean distance from each 2D element of the DEM ground-plane to $\vec{\omega}_i$, and z: The altitude of each 3D element in the DEM. The angle from $\vec{\omega}_i$ to each element in the cross-section is calculated using atan2, and the heighest angle is returned.

The dimensional reduction in Figure 21 happens along the horizontal axis of the percieved cross-section. Instead of using the 2D ground-plane of the DEM, the euclidean distance, or L2-norm, from $\vec{\omega}_i$ to all points along $\vec{R}_j$ are calculated, allowing a 2D representation of altitudes along the DEM. With only 2 dimensions, calculating the angles all points form with $\vec{\omega}_i$ and the groundplane is trivial through the atan2-angular measure. The element with the greatest angle within the cross-section is considered the highest visible point from the maritime vessel at $\vec{\omega}_i$ and it's 3D coordinates inside the DEM are added to $\vec{f}_{DEM_i}$.

Algorithm 7 presents a simplified pseudocode for the proposed strategy:

Algorithm 7 Extraction of skyline element $\mathrm{f}_{i,DEM,j}$ from R_j

Require: Ray as a 2D line on the DEM floor plane L_{R_j}
Require: DEM data I_{DEM}
  procedure ExtractSkylineElementFromRay(L_{R_j}, I_{DEM})
    $DEM_{cross-section} \leftarrow \mathrm{I}_{DEM}[L_{R_j}]$
    $\vec{x} \leftarrow Euclidean(DEM_{cross-section}, \mathrm{I}_{DEM}[L_{R_j}[0]])$
    $\vec{z} \leftarrow DEM_{cross-section}[:, 2]$
    $\vec{\theta} \leftarrow atan2(\vec{z}, \vec{x})$
    return $DEM_{cross-section}[\mathrm{argmax}(\vec{\theta})]$
  end procedure

3.3 Posing the optimization problem: The Reprojection error

With all $\mathcal{M}$ matching features it becomes possible to pose the proposed optimization problem, designed to estimate the actual position of the vessel $\hat{p}_i$. As mentioned introductory in Section 2, this is reminiscent of the vessels 3 DOF pose along a piece-wise linear path, as $\theta_{\hat{p}_i}$ is known for a set of known points $\vec{\omega}_{i-1}$ and $\hat{p}_i$. The proposed solution is then to utilize a so-called reprojection error, the idea of which is, in many ways, fairly simple. As the sought after 2D position $\vec{p}_i$ is known to deviate from the intended position $\vec{\omega}_i$, the latter is used as the initial guess of the estimated position $\hat{p}_i$. During the first iteration of the optimization procedure, the 3D world coordinates $\vec{f}_{DEM_i}$ are projected to the image plane of the camera at position $\hat{p}_i = \vec{\omega}_i$, resulting in $\mathcal{L}$ 2D pixel coordinates. Assuming $\vec{\omega}_i \neq \vec{p}_i$ and $\vec{f}_{DEM_i}$ are perfect matches to the $\mathcal{L}$ 2D pixel coordinates in $\vec{f}_{\vec{p}_i}$, the reprojected coordinates will deviate from the ones in $\vec{f}_{\vec{p}_i}$ by a numerically quantifiable amount. This is the reprojection error e. The goal is then to update the 2D position of the vessel in the DEM, $\hat{p}_i$, such that e is minimized. As $\vec{f}_{\vec{p}_i}$ represents the reprojection of $\vec{f}_{DEM_i}$ at $\vec{p}_i$, this implies that $\hat{p}_i \simeq \vec{p}_i$ when e is minimized, effectively solving the pose-estimation.

Figure 22 attempts to illustrate this concept for the recurring example throughout this report. At the initial estimate $\hat{p}_i = \vec{\omega}_i$, the reprojection of $\vec{f}_{DEM_i}$ to the image plane clearly deviates from the target image feature $\vec{f}_{\vec{p}_i}$. This initial reprojection is illustrated by orange rays in the DEM and pixels on the image plane. At the position where the reprojection error e is minimized, that is where $\hat{p}_i \simeq \vec{p}_i$, the reprojection of $\vec{f}_{DEM_i}$ onto the image plane is equivalent to the target image feature $\vec{f}_{\vec{p}_i}$. This final reprojection is illustrated by green rays in the DEM and pixels on the image plane.

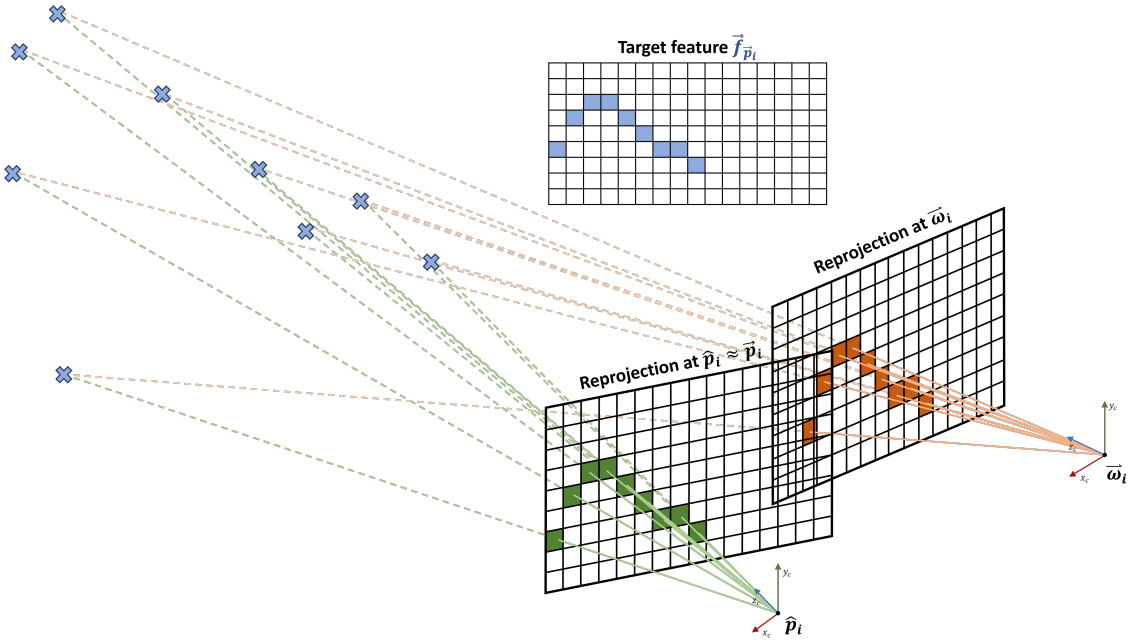


Figure 22: Conceptual idea of pose estimation utilizing the reprojection error. The 3D world coordinates $\vec{f}_{DEM_i}$, here illustrated as blue crosses, are projected to the image plane in its current position and compared with the target feature $\vec{f}_{\vec{p}_i}$. This numerical comparison is the reprojection error. If $\vec{\omega}_i \neq \vec{p}_i$, the initial reprojection at $\vec{\omega}_i$, illustrated by orange rays, will deviate from the target feature, prompting a translation of the image coordinate frame to a position with a better fit. Ultimately, this should end up at $\hat{p}_i$, where the reprojection, as green rays, match the target feature more closely.

The rest of this subsection is dedicated to defining a **general** closed-form, mathematical solution of the reprojection error e given a set of 3D world coordinates at any 2D position within a DEM. For the more specific solution proposed for the chosen implementation, the reader is directed to Section 4.3.2.

To correctly define the reprojection error e, a rigid understanding of the deployed camera model is imperative. Although many models exist for a plethora of cameras, lenses and purposes, a quick summary of the pinhole camera model, employed in this framework, is supplied below:

Pinhole camera model

The pinhole camera model, is a simple but common camera model assuming no lens but a small aperture opening where all light-rays from outside the camera are assumed to intersect before being projected onto an image plane. Generally, the pinhole camera model can be summarized by equation (8):

$$\vec{x}_I = \mathbf{P}\vec{x}_w \tag{8}$$

Here, $\vec{x}_w$ is a 3D world coordinate, $\mathbf{P}$ is the *camera projection matrix* and $\vec{x}_I$ is the 2D image coordinates resulting from projecting $\vec{x}_w$ by $\mathbf{P}$. This projection matrix can be expanded as in equation (9) to encompass two separate matrices describing the state of the pinhole camera.

$$\vec{x}_I = \mathbf{P}\vec{x}_w = (\mathbf{K}\cdot\mathbf{E})\vec{x}_w = (\mathbf{K}\cdot\left[\begin{array}{c|c}\mathbf{R} & \vec{t}\end{array}\right])\vec{x}_w \tag{9}$$

$\mathbf{K}$ is the 3×3 intrinsic matrix describing the internal parameters of a camera, often given on the form of equation (10). f_x and f_y are the focal lengths along the horizontal and vertical axes of the image plane in pixels, s is a skew factor and (o_x, o_y) denotes the *principal point* of the camera, also in pixels. The two focal lengths are used to map continuous values on the image plane to discrete pixels that are not necessarily square, hence the need for two values to describe the relation with each pixels aspect ratio. The skew factor denotes if pixels are skewed in either direction (in most cameras it's set to 0 by default) and the principal point denotes where the optical axis of a camera intersects its image plane.

$$\mathbf{K} = \begin{bmatrix} f_x & s & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \tag{10}$$

$\mathbf{E}$, in equation (9) is the 3×4 *extrinsic matrix*, which can be further reduced to a concatenation of a 3×3 rotation matrix $\mathbf{R}$ and a 3×1 translation vector $\vec{t}$.

An important note to make about the resulting image coordinates $\vec{x}_I$ is that the last element of the 3×1 vector is a scaling element denoted λ. To normalize the result to the discrete image plane consisting of pixel-values, often called the UV plane, the output is normalized with regards to the last element as follows:

$$u = \frac{\vec{x}_{I,x}}{\lambda}, \quad v = \frac{\vec{x}_{I,y}}{\lambda}$$

The reprojection error e can simply be defined as a sum of differences between the $\mathcal{L}$ projected elements of $\vec{f}_{DEM_i}$ at position $\hat{p}_i$ and the corresponding $\mathcal{L}$ elements in $\vec{f}_{\vec{p}_i}$, as in equation (11):

$$\begin{aligned} e &= \textstyle\sum_{k=0}^{\mathcal{L}} ||\mathbf{P}\vec{f}_{DEM_i,k} - \vec{f}_{\vec{p}_i,k}|| \\ &= \textstyle\sum_{k=0}^{\mathcal{L}} ||(\mathbf{K}\cdot\mathbf{E})\vec{f}_{DEM_i,k} - \vec{f}_{\vec{p}_i,k}|| \\ &= \textstyle\sum_{k=0}^{\mathcal{L}} ||(\mathbf{K}\cdot\left[\begin{array}{c|c}\mathbf{R}^w_{c\theta} & \vec{t}^{\,w}_{c\theta}\end{array}\right])\vec{f}_{DEM_i,k} - \vec{f}_{\vec{p}_i,k}|| \end{aligned} \tag{11}$$

One should note the extrinsic matrix $\mathbf{E} = \left[\begin{array}{c|c}\mathbf{R}^w_{c\theta} & \vec{t}^{\,w}_{c\theta}\end{array}\right]$, which specifies the change of base of the rotation matrix $\mathbf{R}^w_{c\theta}$ and translation vector $\vec{t}^{\,w}_{c\theta}$ from an arbitrary world coordinate system w to a general camera coordinate system c of a known heading θ. The choice of distance measure $||\cdot||$ further decides the shape and behavior of the error-function, hence presenting an important parameter for the overall performance of the pose estimator. In this framework, the *euclidean distance*, often called L2 norm, between corresponding points is adopted.

The final optimization problem can be formed by a minimization of equation (11) with regards to the estimated position $\hat{p}_i$, yielding an ideal pose-estimation $\hat{p}_i{}^*$ in equation (12):

$$\hat{p}_i{}^* = argmin_{\hat{p}_i} e \tag{12}$$

Again, note how, for a piece-wise linear path, the heading $\theta_{\hat{p}_i}$ is implicitly given by the estimated location $\hat{p}_i$ and is therefore not explicitly part of the above optimization.

3.4 Adaptive grid search to minimize the reprojection error

Given the non-linearity of equation (12), the deployment of a nonlinear solver is necessary to estimate the 2 DOF solution $\hat{p}_i{}^*$. For the sake of simplicity, this proof of concept limits this solver to a less advanced adaptive grid search, the theory of which will be covered briefly as its implementation is of secondary interest to the proposed optimization problem.

The adaptive grid-search is an improvement over the exhaustive grid search, which is often considered a brute-force approach for solving both linear and nonlinear problems. The method is used in optimization to alter the state of a system by a fixed step-length t at each iteration, slowly, but surely converging to the nearest local minima. The adaptive method makes this step-length a variable t_i, changing its value when an optimal for the current value is reached, before finding a new optima with the new length. In most cases, the initial length, t_0, is the largest value and subsequent lengths become smaller. This way, a crude approximation of the solution is first found using a large step length, before smaller steps are applied to get as close to the optimal solution as possible.

For the presented reprojection error in (11) the solution is adapted to finding a 2 DOF position estimate using a set of *neighborhoods*. These neighborhoods are a generalization of the step-length around the currently evaluated position, essentially denoting half the length of the cross-section of a square. The border of these square neighborhoods denote where the possible next position updates can be located. For each of these, $\vec{f}_{DEM_i}$ is reprojected back to the camera aboard the vessel, and the position resulting in the greatest decrease of the reprojection error is selected. When no new optimal value can be found with the current neighborhood, the cross-section is halved and the algorithm continues. This is done recursively until no better solution can be found or a number of iterations are surpassed. Figure 23 illustrates the proposed algorithmic approach. The neighborhood is initialized at position $\vec{\omega}_i$ with half a cross-section t_0, which is iteratively halved as the estimate improves. After 3 iterations, the method returns an estimate $\hat{p}_i$.

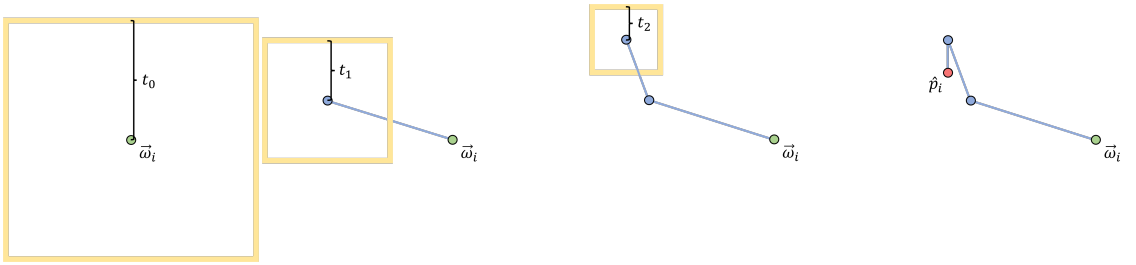


Figure 23: Example of the proposed, adaptive grid-search. After 3 iterations, the estimation starting in $\vec{\omega}_i$ finds a 2D location $\hat{p}_i$ optimizing the reprojection error (11). At each iteration, the cross-section of the square neighborhood is halved to more precisely hone in on an optimal solution.

4 Implementation

This section seeks to present the actual implementation of the proposed framework, as well as assumptions, considerations and decisions taken underway. The entire pipeline is highlighted for reference in Figure 24 serving as a template of the framework all the way from data acquisition to pose estimation. The system is split into 3 separate overarching modules, reflected in the division of the following section: Firstly, the acquisition and preprocessing of topological data for use in both the simulator- and analysis-environment is covered, with focus on how the conversions are aimed at staying 1:1 throughout the framework. Secondly, the process used to generate synthetic data through the Unity game engine is presented, building on previous work conducted in [39], and thirdly, the necessary considerations and adaptations to the algorithms presented in Section 3, taking place post-estimation, are described in detail.

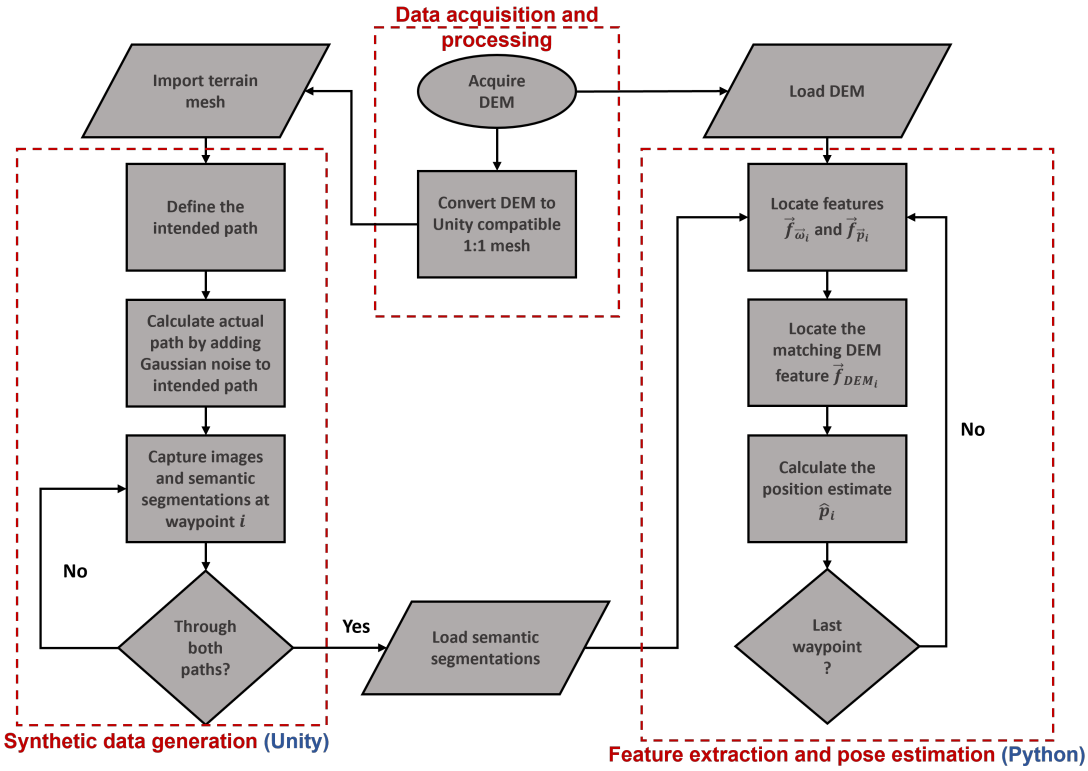


Figure 24: Pipeline of the proposed framework. Note the 3 overarching modules for data acquisition and processing, synthetic data generation, which takes place in Unity, and feature extraction and pose estimation, which takes place in Python.

Before delving into the specifics of the implementation, it is important to highlight a vital part of the presented pipeline: The disconnect between simulation and analysis. The idea is to present one simulator developed in the Unity Game engine and a framework for analysis developed independently in Python. As can be seen in the above pipeline, the task assigned to the Unity simulator was to generate synthetic imagery and/or semantic segmentations of a simulated maritime vessel in a marine environment, which were to be analyzed post-simulation by the software in Python. Although this simplifies the framework from a real-time simulator implementation, making it suitable for a proof of concept, the division between multiple development platforms introduces certain challenges, especially related to data-cohesion and replication.

4.1 Preserving DEM identicalness

As the Unity simulator was tasked with generating synthetic imagery and semantic segmentations of the terrain surrounding a maritime vessel, the question quickly became how elevation data should best be handled. For use within the Python application, the DEM could be utilized on its original format, typically as a .tif file, with its specific resolution, or meters per pixel, r. The Unity simulator, however, would require the DEM to be represented in the form of a 3D mesh: a model consisting of polygons built from a set of vertices, forming a 3-dimensional '*object*'. Such an object would then have to be generated through a conversion of the original DEM, keeping its resolution r identical and returning a 3D mesh which would be 1:1 with the raw data. Anything less precise than this would render the estimation framework infeasible, generating differences between the terrain data analyzed in Unity and Python independently. In other words: The same terrain, analyzed in different software at different stages in time, had to be identical after conversion.

The conversion adopted by this report is a manual one, going through 4 fixed steps to convert DEM data on a .tif format, into a .blend or .fbx file compatible with Unity. This was deemed a safer approach in comparison to using third-party plugins or software, as it allowed for full control of each step of the procedure rather than blindly trusting the output of a more efficient and simpler black-box conversion. Although a more cumbersome process, the final product came out very promising, with close to no visual discrepancies and a 1:1 scale upon a comparative analysis between Unity and Python. A DEM in raster-form, such as illustrated in Figure 25, is downloaded with an arbitrary precision r and size, illustrating the 3D topology of a landscape in the form of a height map:

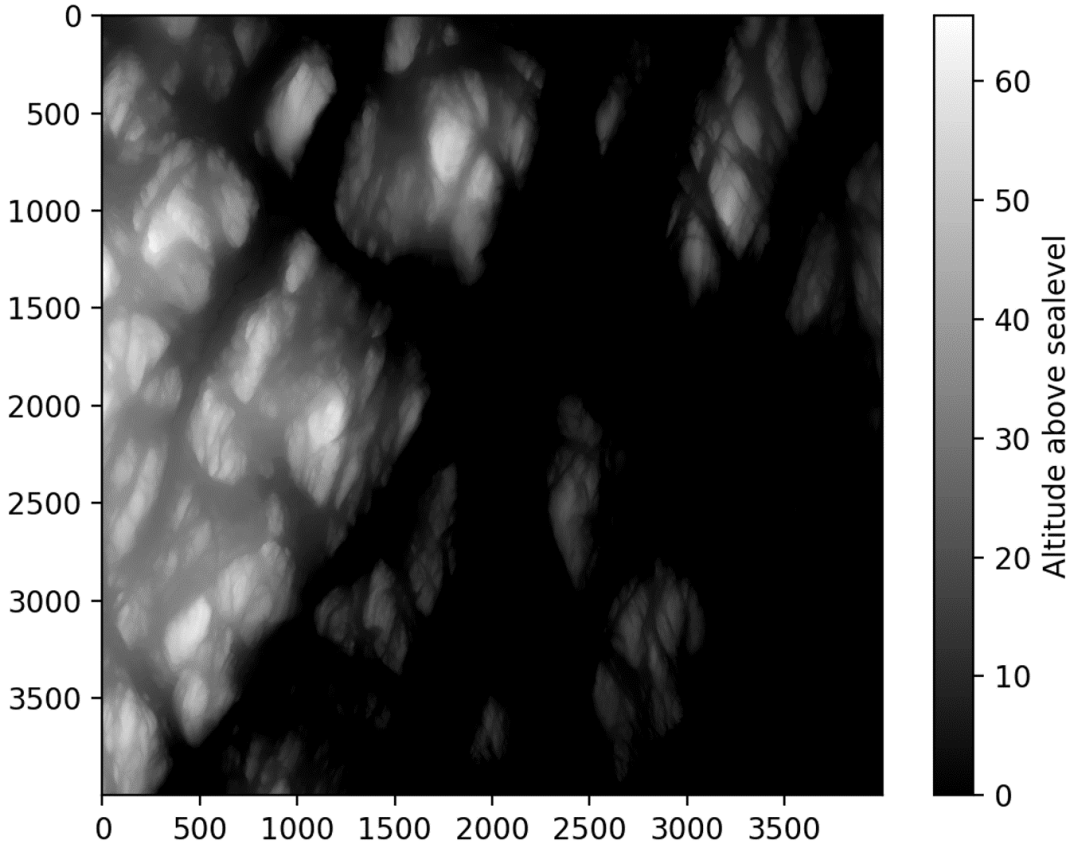


Figure 25: Elevation data embedded in a DEM, unaltered.

At this point, the DEM is on the form of a simple raster image with encoded data in a separate file. To transform it into a 3D mesh with interconnected polygons between vertices, the popular open source geographic system QGIS [43] was utilized. Here, a java-based plugin called Qgis2threejs [44]

could be employed to import and convert geospatial data into 3D meshes through the well-known Javascript library threejs [45], before being exported for later use. Figure 26 shows the raster of Figure 25 converted to a 3D mesh inside the interactable interface of the plugin. Some things to note here is that the subsampling/smoothing of the mesh is set to the minimum value such as to retain the original dimensions and vertices of the DEM, whilst a manual polygon is used to appropriately define the borders of the DEM within QGIS. The result is exported as a .glb file; a standardized format used to export and send 3D models in the form of a binary file.

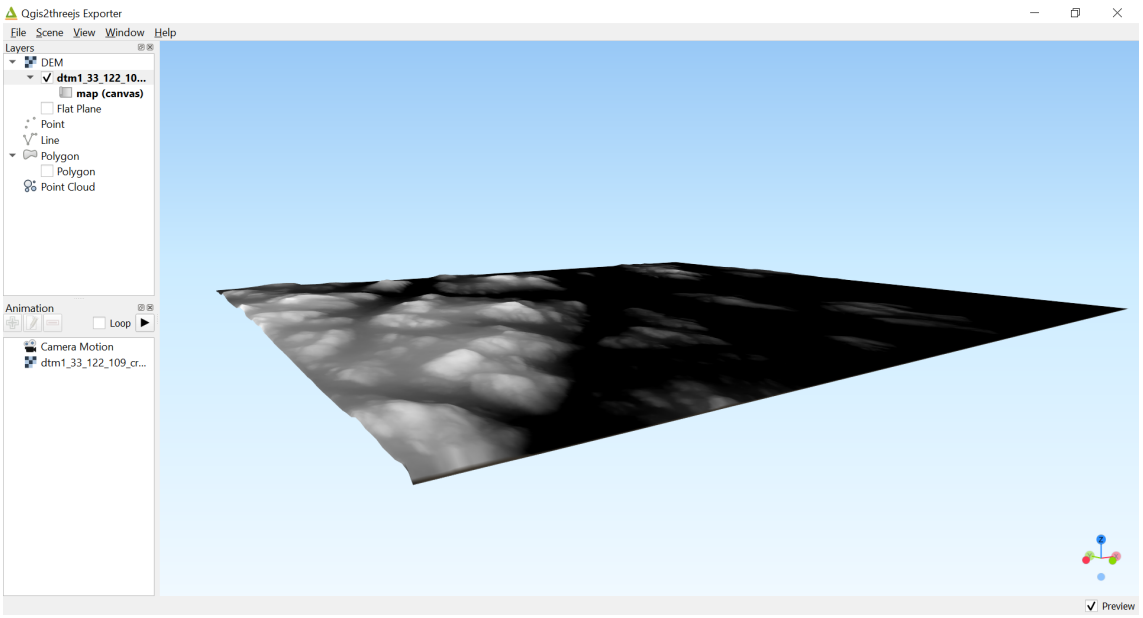


Figure 26: The raw raster data from Figure 25, imported into and visualized by the Qgis2threejs interface. Subsampling and smoothing of the raster data is set to the minimal value and the borders of the DEM are reinforced by the *Polygon* object.

The resulting .glb file is imported into Blender, a highly regarded open source software used for just about anything related to 3D models [46]. This is capable of importing .glb files and convert them into files suitable for use within the Unity game engine, such as .fbx or .blend files. Figure 27 shows the .glb file imported into blender as a registered 3D mesh, whilst Figure 28 shows the fully converted .blend mesh imported into unity with a simple 3D plane functioning as a makeshift substitution for a generated ocean:

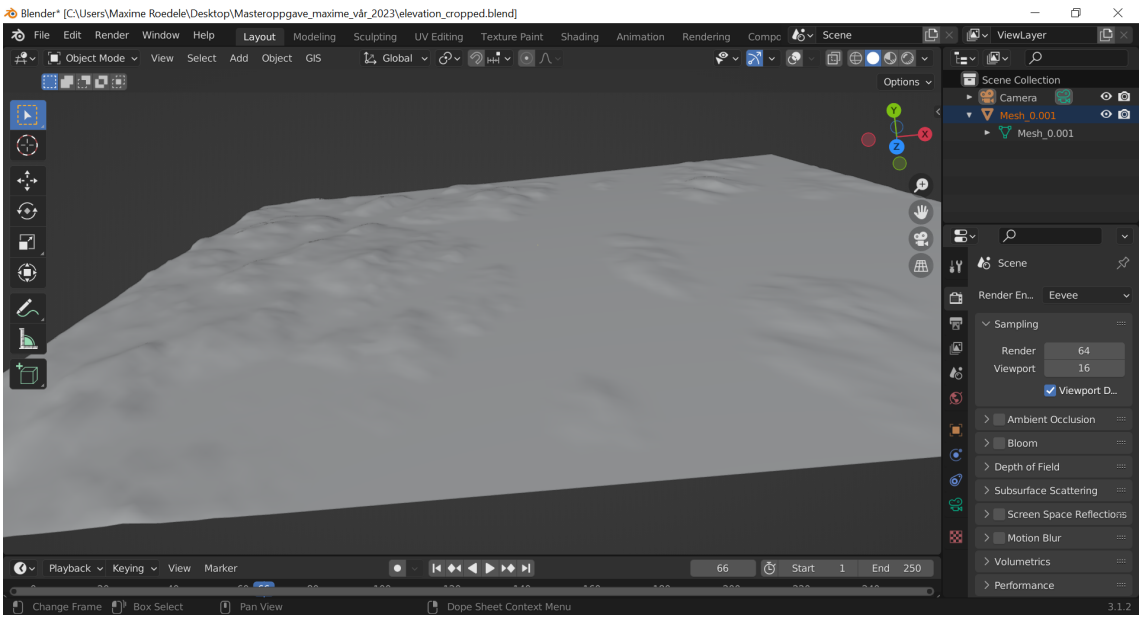


Figure 27: The .glb file of Figure 26 imported into Blender before being exported as a 3D model on the .blend format.

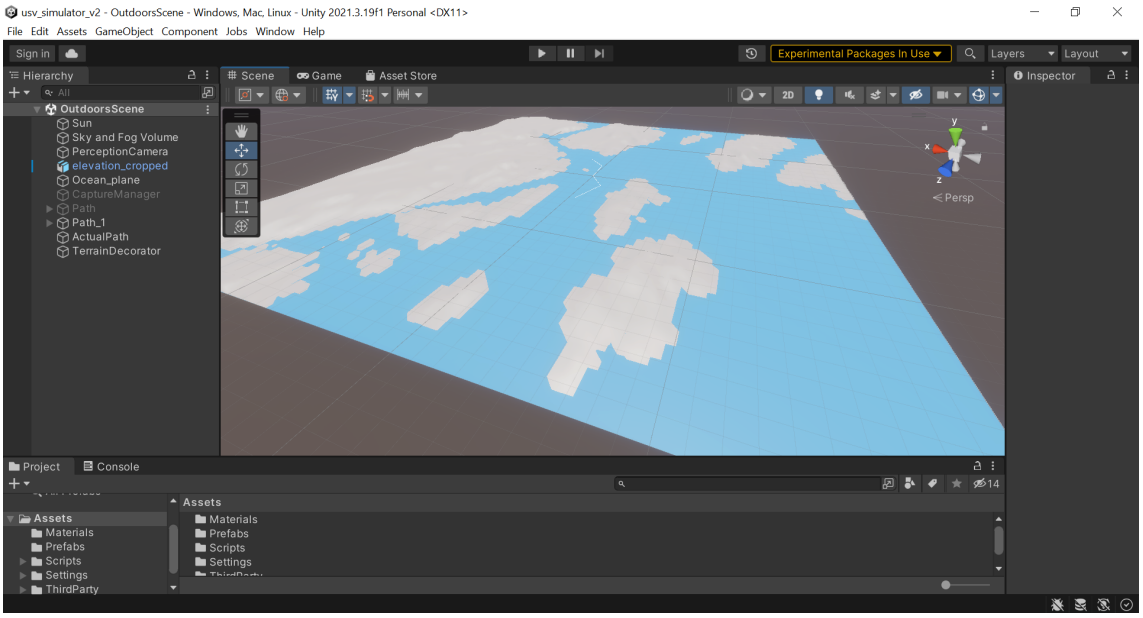


Figure 28: The fully converted .blend file of the raw raster data imported for use in Unity. A simple, 3D plane is inserted with a blue material to illustrate the ocean level.

Although this method seems promising upon inspection of the final results and the comparison with the original raster data, it should be noted that a conversion with this many steps can be prone to errors and could be necessary to revise in future implementations.

4.2 Synthetic data generation

The basic simulator environment deployed for this proof of concept is in many ways identical and/or similar to the one implemented in [39]. With the 3D terrain data located in the middle of the world coordinate system, a simple 3D plane slightly elevated above the actual sea-level

(∼5mm) is used to more clearly distinguish the ocean from the surrounding terrain. The reason for this slight increase in altitude is to avoid so called *z-fighting*, where different objects with similar distances to the camera begin to experience flickering during rendering due to near identical values in the depth-buffer. It should further be noted that the terrain is not curved as to fit the earths circumference, but is assumed to be planar in a smaller area. A basic environment from the default High Definition Render Pipeline (HDRP) is used, implementing a simple but efficient atmospheric lighting, and unnecessary post-processing effects are disabled on the camera.

The intended path of the maritime vessel is generated by the improved *Path Generator* component, stemming from [39], which is assigned to an empty GameObject in the scene intended to work as a parent-transform to all n waypoints along the path. These n waypoints are then instantiated as n child-objects of this Path Generator and are linked in the order of appearance within the Unity hierarchy. Figure 29 show the implemented setup of a random path in the hierarchy, with Unity Gizmos highlighting the path in the editor and a basic interface in the inspector:

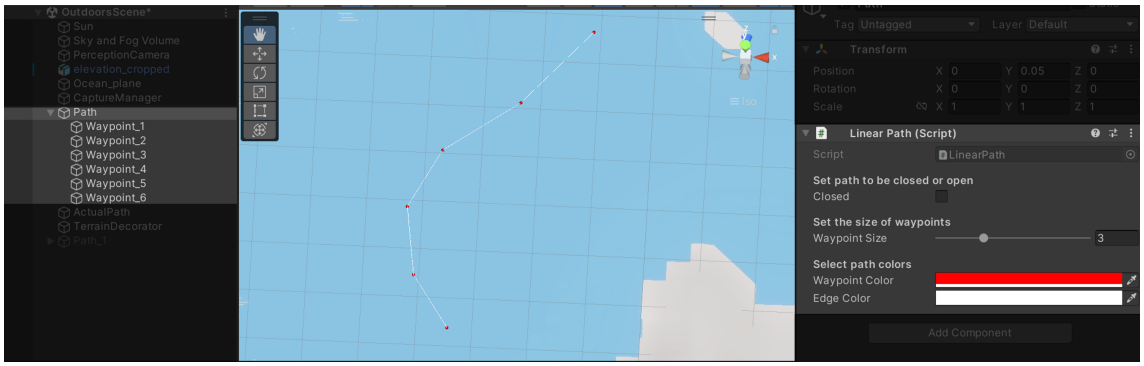


Figure 29: Intended path, as shown in the Unity editor. The path consists of 6 waypoints in the hierarchy on the left, under a parent gameobject. They are visible in the scene view in the middle of the image through a set of unity gizmos and the Path Generator component, named *Linear Path*, can be seen on the right.

Similarly to the above Path Creator component, the *Noisy Path Generator* is adopted from [39]. This component is used to generate an actual path deviating from the intended path by applying Gaussian noise $\mathcal{N}(\mu, \phi)$ to each intended position $\vec{\omega}_i$ at waypoint i. The mean value of the noise is set to $\mu = 0$ by default, whilst the variance ϕ can be set in the component window. Figure 30 illustrates how the tool looks and works inside the Unity editor. On the right, in the component window, the *Intended path* of which the component inherits the intended waypoints is assigned, as well as min and max values for ϕ. The Gaussian noise is further applied along a specified set of *dimensions*, here set to be the XZ plane. In the hierarchy on the left a new set of n waypoints are instantiated under the ActualPath parent transform, and a new path of blue vertices connected by purple edges can be seen in the scene view.

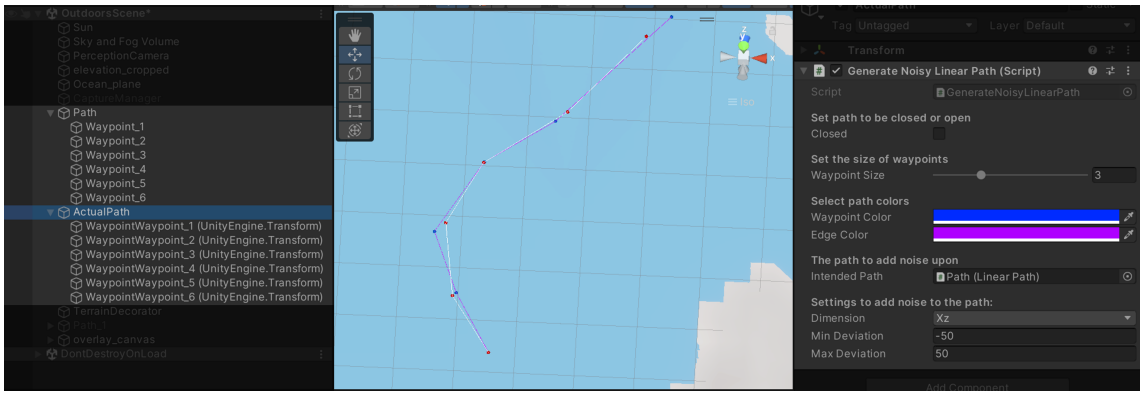


Figure 30: Actual path in the Unity editor. The intended path from which the waypoints are inferred, as well as the parameters of the applied Gaussian noise, are assigned in the Inspector on the right. A set of n actual positions $\vec{p}_i$ are instantiated in the hierarchy on the left and the new path is visible in the scene view.

To acquire the data along both paths, the Perception Camera package from Unity [47] was utilized exactly as in [39]. This allows for a quick and easy integration of a *Perception Camera* component, performing flawless semantic segmentations of objects in the world with correctly assigned *Segmentation Labels*. Although this is not necessarily a realistic segmentation, given how well the labeler works under all conditions such as with heavy fog, rain or powerful diffuse reflections of sun in the water, it is sufficient for a proof of concept of the proposed framework.

To streamline the collection of data and the export from both the Perception Camera and Path Generators, a component named the *Capture Manager* was instantiated. This was tasked with managing the cameras position and heading between all waypoints of an intended and actual path, requesting captures from the camera with sufficient delays to compensate for the shutter speed of a simulated physical camera and storing all relevant data at different locations. The storing of data itself was done through two coroutines: One implemented in the *Linear Path*-class inherited by both Path Generators, writing the positions of all waypoints in each respective path to a designed .txt file, and one in the Capture Manager writing the utilized intrinsic camera parameters to another .txt file for reproduction of the Unity camera in Python.

Finally, the Capture Manager also handles the update of the intended path presented in Figure 9. At each intended position $\vec{\omega}_i$, the previous intended position is updated to the previous actual location $\vec{\omega}_{i-1} = \vec{p}_{i-1}$, mimicking the update one could expect from an optimal pose estimator.

4.3 Data reproduction from Unity to Python

With data generated and exported from the Unity simulator, the remaining feature extraction and estimation is to be performed in a separate Python implementation, adopting and utilizing the algorithms presented in Section 3. This comes with a few challenges however, all introduced by the need to replicate parts of the simulator environment present in Unity in the adopted Python environments. In this report, the main libraries utilized in Python for image analysis and feature extraction were Numpy [48] for mathematical operations and OpenCV [49] of more specific, image based operations. Evidently, there are fairly large differences between the different environments, requiring some adjustments and adaptations to allow for consistent and accurate feature extraction and analysis. Amongst these challenges, the one constantly rearing its head proved to be the difference in world coordinate frames between the two platforms. Most mathematical or graphical libraries in Python utilize a right-hand coordinate system, whilst Unity opts for a left hand coordinate system, popular amongst some 3D software packages, but rather unusual in the grand scheme of things. Furthermore, it should be noted that matrix conventions (and hence most predefined matrices in Unity) change given the currently deployed graphics API, the OS the Unity installation is currently running on and whether the actual computation is performed on the GPU or CPU.

4.3.1 Reproducing the intrinsic matrix of Unitys Phyiscal Camera component

As of the publishing of this report, no satisfactory means exist in which to extract a fully functioning 3×3 intrinsic matrix $\mathbf{K}$ from a physical camera component through the Unity API. The closest are a few public methods capable of extracting projection matrices from either the CPU or GPU, which project 3D camera-coordinates into *normalized device coordinates*. Although similar at a glance, the resulting matrices do not yield a good generalization of camera parameters and result in poor performances when used to re-create said camera in Python. Hence, the method of replication presented in this report is a re-creation of the intrinsic matrix $\mathbf{K}$ from camera parameters readily available through the Unity API.

Referencing the matrix in equation (10), the directional focal lengths f_x and f_y have to be found in [pixels]. This is not directly available through the Unity API, but can be calculated through available parameters. More specifically, the focal length f [mm] denoting the distance from the camera center to the image plane, can be multiplied by a *pixel density measure* m_x and m_y, denoting the amount of pixels per physical millimeter on the *camera sensor* along the horizontal and vertical axes respectively. As the sensor dimensions s_x and s_y [mm] can be fetched from the physical camera component in Unity, as well as the width $\mathcal{N}_w$ and height $\mathcal{N}_h$ [pixels] of the resulting image plane, f_x and f_y can be found by equation (14):

$$f_x = m_x f = \frac{\mathcal{N}_w}{s_x} f \tag{13}$$

$$f_y = m_y f = \frac{\mathcal{N}_h}{s_y} f \tag{14}$$

The *principal point* (o_x, o_y) [pixels] can be found through public methods on the physical camera component in Unity, enabling the re-construction of the intrinsic camera matrix from Unity for use with Python libraries as per equation (15):

$$\mathbf{K} = \begin{bmatrix} \frac{\mathcal{N}_w}{s_x} f & 0 & o_x \\ 0 & \frac{\mathcal{N}_h}{s_y} f & o_y \\ 0 & 0 & 1 \end{bmatrix} \tag{15}$$

4.3.2 Reprojection of 3D world coordinates

When reprojecting the 3D world coordinates in $\vec{f}_{DEM_i}$ as in Section 3.3, the differences in coordinate frames between Unity and the Python libraries present a number of challenges when defining the camera extrinsic matrix $\mathbf{E}$. The parameters of $\mathbf{E} = \left[\begin{array}{c|c} \mathbf{R}^w_{c\theta} & \vec{t}^w_{c\theta} \end{array} \right]$ require careful crafting to match the change between world- and camera-coordinates as well as camera positions and headings along a piece-wise linear path. To find closed-form solutions to the above parameters, Figure 31 is supplied for geometric interpertations. Two coordinate systems w and c are defined for world- and camera-coordinates respectively, as well an arbitrary rotation of the camera θ, caused by the constraints on the linear path. A generic 3D point in world coordinates $\vec{p}_w$ is also defined for reprojection back to the camera's image plane. Notice how, as the reprojection is happening in Python, the world coordinate system is right handed, whilst the camera coordinate system, which is replicated from Unity, is left handed. Also notice how the representation of the camera extrinsics are happening in the XZ-plane, again replicating the original Unity environment.

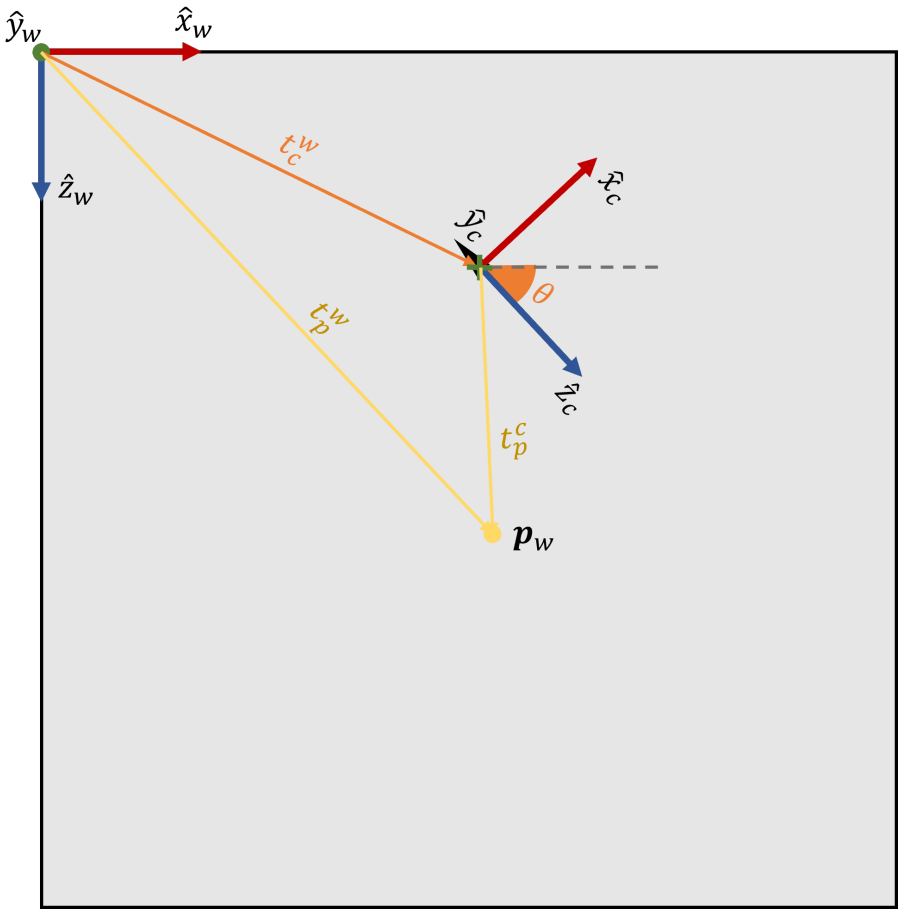


Figure 31: Geometric interpretation of the relationship between world- and camera-coordinate frames.

Figure 31 also contain a set of vectors intended to visualize the necessary steps to transform $\vec{p}_w$ into camera coordinates: $\vec{t}_p^{\,w}$ denoted the position of $\vec{p}_w$ in world coordinates, $\vec{t}_c^{\,w}$ denote the position of the camera in world coordinates and $\vec{t}_p^{\,c}$ denote the position of $\vec{p}_w$ in camera coordinates. Upon inspection of these vectors, one can come to the conclusion that $\vec{t}_p^{\,c}$ is the key to finding the parameters of $\mathbf{E}$, being the output of a 3D world coordinate transformed by said extrinsic matrix. Denoting the general rotation and translation necessary to go from w to c rotated by an angle θ as $\mathbf{R}_{c\theta}^{w}$ and $\vec{t}_{c\theta}^{\,w}$ respectively, $\vec{t}_p^{\,c}$ can be written in order of the other vectors and used to isolate $\mathbf{E}$ as in equation (16):

$$\begin{aligned}\vec{t}_p^{\,c} &= \mathbf{R}_{c\theta}^{w}(\vec{t}_p^{\,w} - \vec{t}_c^{\,w}) \\ &= \mathbf{R}_{c\theta}^{w}\vec{t}_p^{\,w} - \mathbf{R}_{c\theta}^{w}\vec{t}_c^{\,w} \\ &= \left[\begin{array}{c|c} \mathbf{R}_{c\theta}^{w} & -\mathbf{R}_{c\theta}^{w}\vec{t}_c^{\,w} \end{array}\right]\tilde{p}_w = \mathbf{E}\tilde{p}_w \end{aligned} \tag{16}$$

Note the use of *homogeneous coordinates* here for $\tilde{p}_w = \begin{bmatrix} x_w & y_w & z_w & 1 \end{bmatrix}^\intercal$. Equation (16) yields an expression directly comparable to $\mathbf{E} = \left[\begin{array}{c|c} \mathbf{R}_{c\theta}^{w} & \vec{t}_{c\theta}^{\,w} \end{array}\right]$, resulting in equation (17) for $\vec{t}_{c\theta}^{\,w}$:

$$\vec{t}_{c\theta}^{\,w} = -\mathbf{R}_{c\theta}^{w}\vec{t}_c^{\,w} \tag{17}$$

What remains is then to find the correct rotation $\mathbf{R}_{c\theta}^{w}$, which can be seen as a compound transformation between two dependent rotations: $\mathbf{R}_{c}^{w}$ and $\mathbf{R}_{c\theta}^{c}$.

$\mathbf{R}_c^w$ is the coordinate frame transformation from w to c, with $\theta = 0$. This yields an overlap between the unit-axes $\hat{x}_w$ and $\hat{z}_c$, resulting in the matrix

$$\mathbf{R}_c^w = \begin{bmatrix} 0 & 0 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$\mathbf{R}_{c\theta}^c$ can be found by a standard counter-clockwise 3D rotation matrix of θ about the camera y-axis, yielding:

$$\mathbf{R}_{c\theta}^c = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix}$$

Combining these two together into the compound rotation matrix $\mathbf{R}_{c\theta}^w$ can then be done by a simple matrix multiplication, shown in equation (18):

$$\begin{aligned} \mathbf{R}_{c\theta}^w &= \mathbf{R}_c^w \mathbf{R}_{c\theta}^c \\ &= \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} -sin(\theta) & 0 & cos(\theta) \\ 0 & -1 & 0 \\ cos(\theta) & 0 & sin(\theta) \end{bmatrix} \end{aligned} \tag{18}$$

As so, the extrinsic matrix of a camera in Unity can be replicated in Python by supplying the camera rotation θ, available from the linear path, and camera position in world-coordinates $\vec{t}_c^w$.

5 Experimental results

This section seeks to present the experimental results of the entire pose estimation framework. Firstly, the experimental setup is presented, introducing the external DEM and a set of paths on which synthetic data for a maritime vessel is generated. Afterwards, a selection of outputs of all modules within the framework are presented in order of introduction, before the output of the pose estimator and its performance over a plethora of different paths is illustrated in detail. Visual results are supplied throughout, as the performance of certain modules are best analyzed by human inspection. Numerical and statistical results are further supplied where they contribute meaningfully to the discussion.

For repeatability of the results, Table 1 presents the specific hardware testing was performed upon:

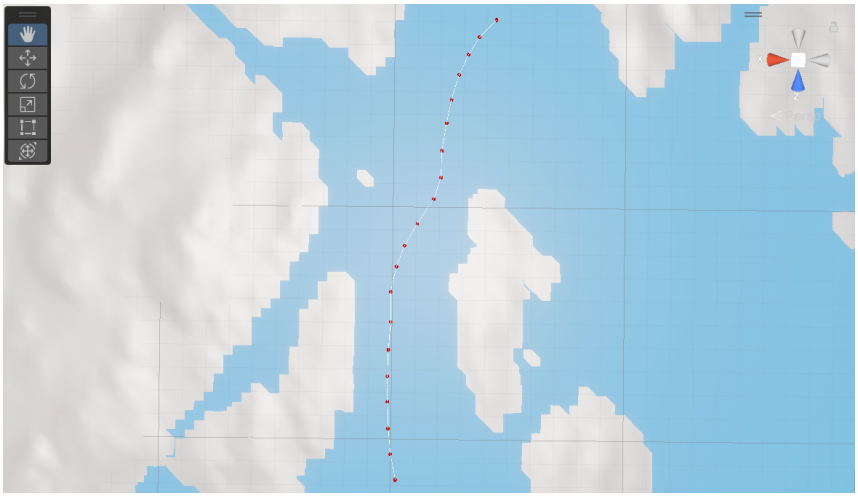
Component	Type
CPU	Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
GPU 1	Intel(R) UHD Graphics 620
GPU 2	NVIDIA GeForce MX150
Storage	SanDisk SD9SN8W256G1002 (SATA)
RAM	8,00 GB
OS	Windows 10 Home

Table 1: Hardware on which both testing and development was conducted.

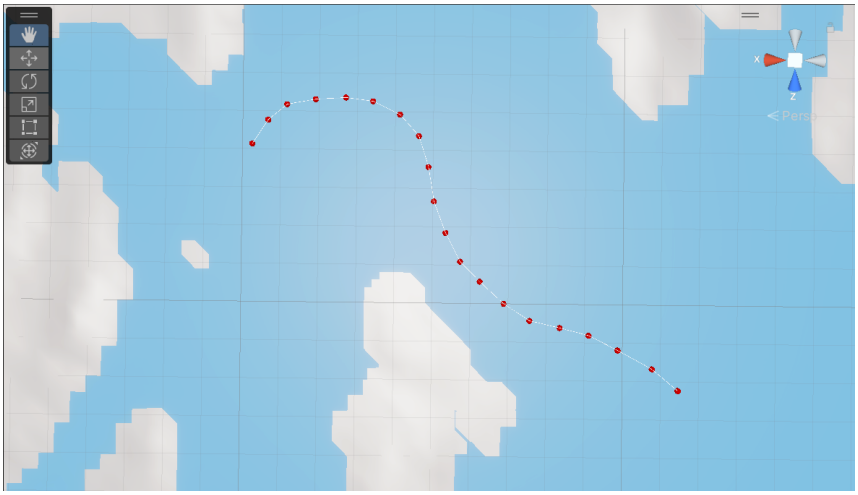
Furthermore, the complete API for the implemented framework can be found in this public GitHub repository, where continuous development might happen independent of the work presented in this thesis:
https://github.com/MaximeRoedele/API_master_thesis

5.1 Experimental setup

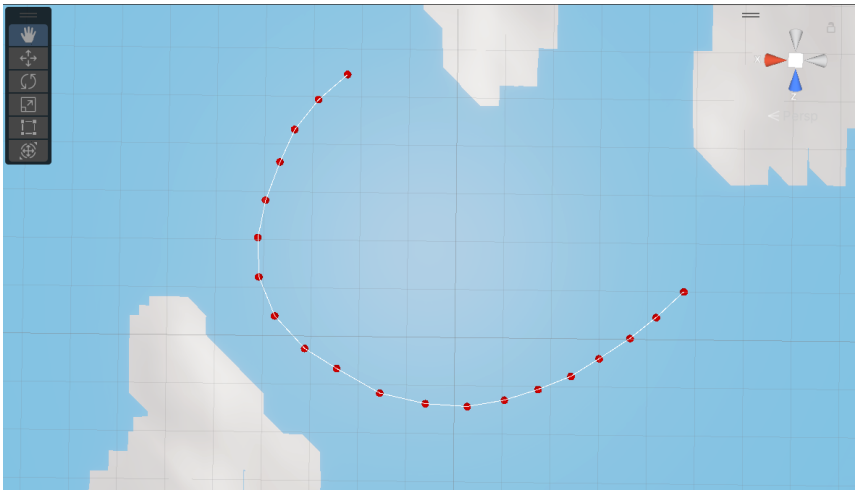
To evaluate the performance of the presented framework, 3 distinct intended paths of $n = 20$ waypoints each (total of 57 if one doesn't consider initial waypoints) were defined within the confines of a known DEM, acquired from the Norwegian Mapping Authority [50]. Said DEM represents a coastline outside of Nøtterøy in Norway and has an incredibly high resolution of $r= 1$. All 3 paths can be seen in Figure 32. The goal of using multiple, longer paths with a high number of waypoints was for the sake of a qualitative and quantitative analysis of the framework performance in as many scenarios as possible. Hence, the curvature of the paths, as well as their mean distances to shorelines, are noticeably different.



(a) Path 1: A straight line with minimal visible terrain.



(b) Path 2: A curving path with a mix between far-away and close terrain.

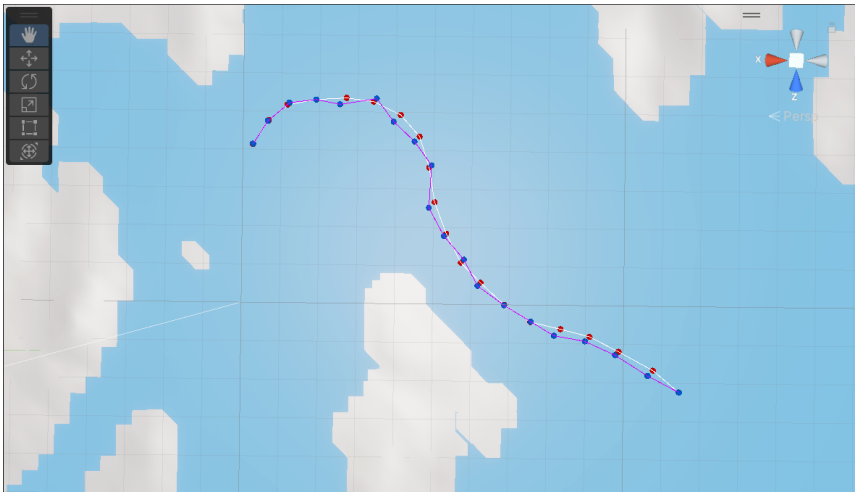


(c) Path 3: A curved path in mostly close terrain.

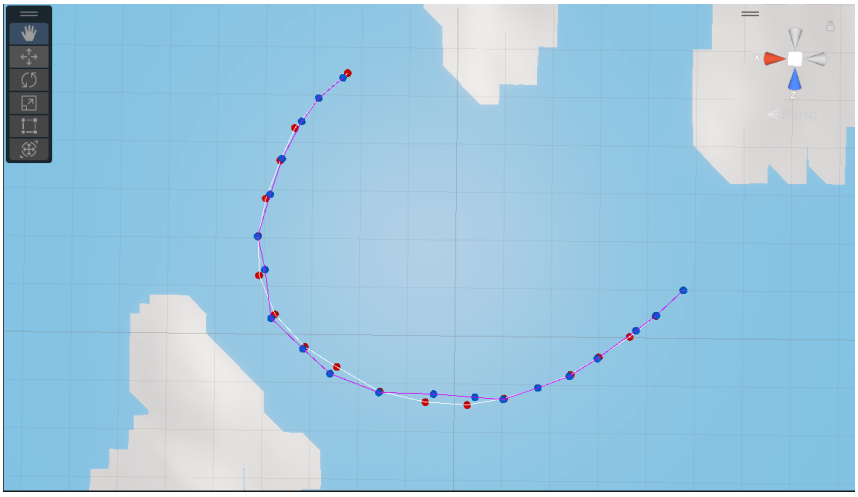
Figure 32: The 3 paths used to generate data for framework evaluation and module analysis.

The actual paths, generated by adding planar noise with $\mu = 0$ and $\phi = 35$m to the intended paths, can be seen in Figure 33:

(a) Path 1 with noise.



(b) Path 2 with noise.



(c) Path 3 with noise.

Figure 33: The 3 actual paths, overlayed onto the intended paths of Figure 32. They can be distinguished by the blue vertices and purple edges.

For reproduceability of the supplied results, the parameters of the simulated camera can be seen in Table 2:

Camera parameter	value
$\mathcal{N}_w$	1920
$\mathcal{N}_h$	1080
Near clipping plane	0.1
Far clipping plane	5000
Horizontal FOV, FOV_H	70°
Focal length f	25.70667 [pixels]
Skew	0
Optical center (o_x, o_y)	(960, 540)
Sensor size x	36 [mm]
Sensor size y	24 [mm]
Camera Gate Fit	*None*

Table 2: Camera parameters and their chosen values during data-generation.

Finally, on the analysis of results and provided statistics: To the authors knowledge there does not exist a well-established framework in which to analyse the performance of modules for the extraction and matching of visual features. This has to be done visually through inspection, although results can be ambiguous at times. For modules producing numerical outputs however, like the 3 DOF pose estimator, numerical values and deviations can be calculated to yield a directly measurable analysis.

5.2 Feature extraction and matching

This subsection will provide visual results of the presented modules used for feature extraction and matching.

5.2.1 Skyline contour extraction

Applying Algorithm 1 to semantic segmentations of terrain resulted in consistent skyline contours across as good as all captured images. Figures 34, 35 and 36 illustrate 3 examples encapsulating the algorithm performance in the most common scenarios: Figure 34 illustrate the extraction of a skyline contour from a secluded piece of terrain, Figure 35 illustrate the extraction of many independent contours and Figure 36 illustrate the extraction of one large, continuous contour spanning the entire camera frustum. Note here that the green pixels in the image are segmented sections of terrain, whilst red points illustrate the extracted skyline:

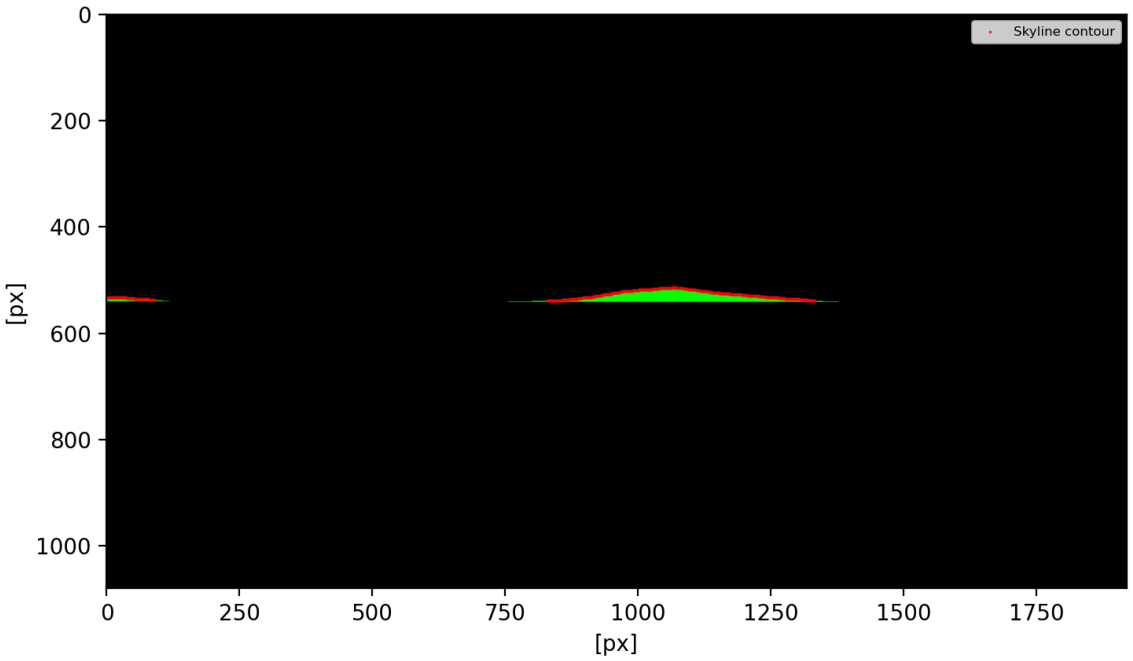


Figure 34: Skyline contour extracted at waypoint 18 along the actual path of path 1. Green pixels in the image are segmented sections of terrain, whilst red points illustrate the extracted skyline.

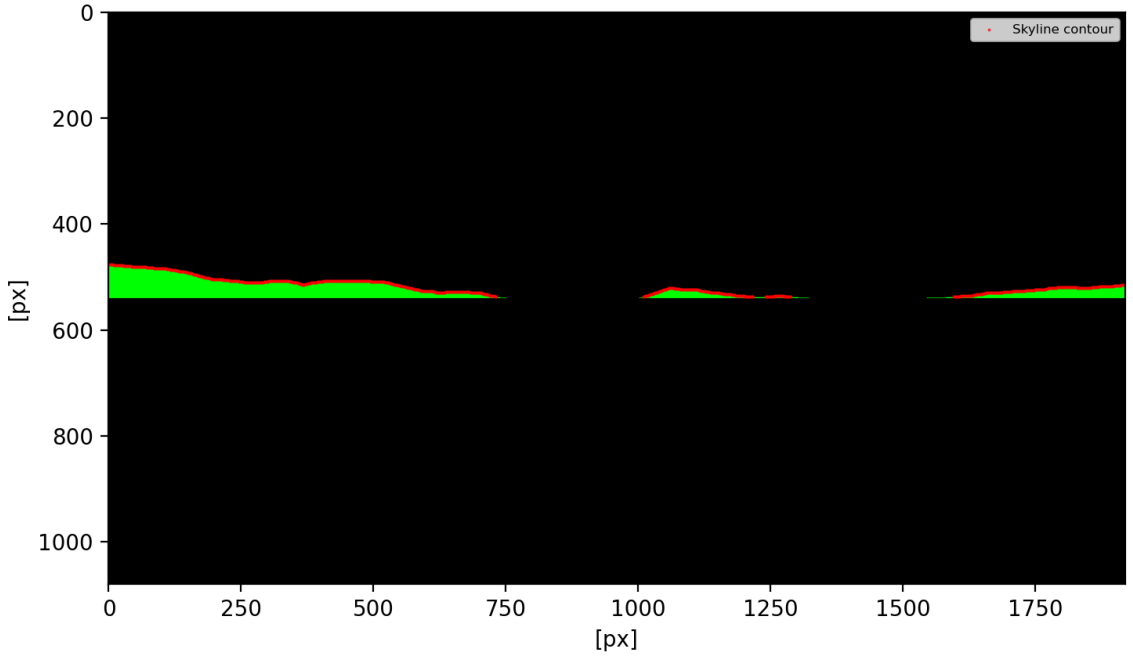


Figure 35: Skyline contour extracted at waypoint 11 along the intended path of path 2. Green pixels in the image are segmented sections of terrain, whilst red points illustrate the extracted skyline.

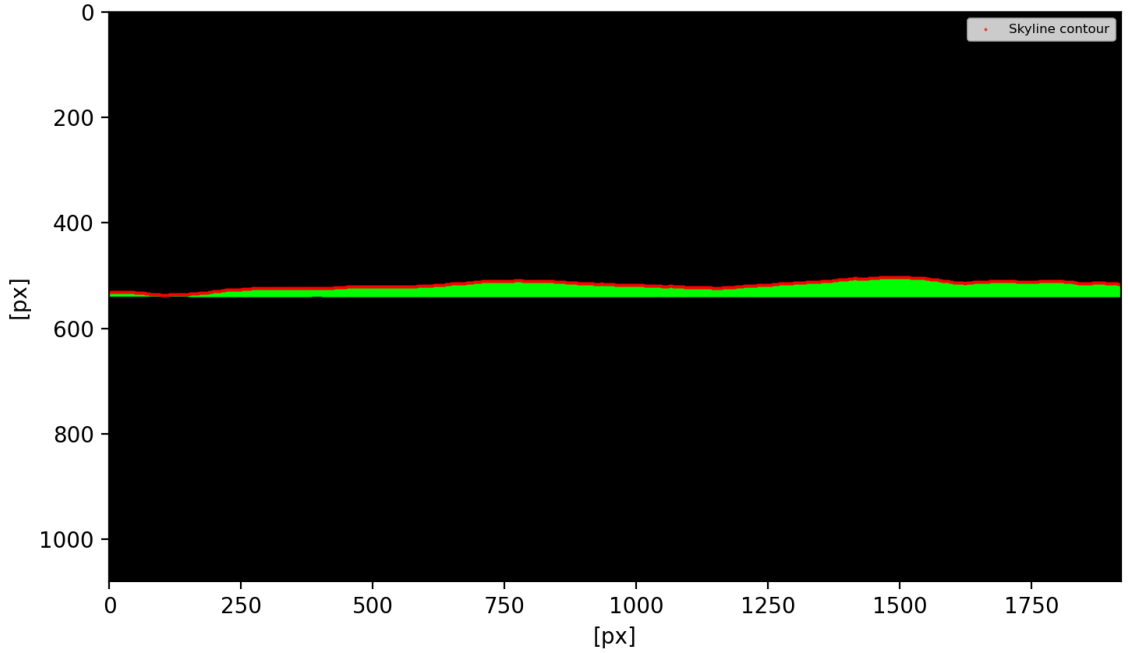


Figure 36: Skyline contour extracted at waypoint 6 along the actual path of path 3. Green pixels in the image are segmented sections of terrain, whilst red points illustrate the extracted skyline.

Some attention should however be drawn to the Figure 37, where the noise-removal results in an ambiguous contour on the leftmost side of the larger landmass:

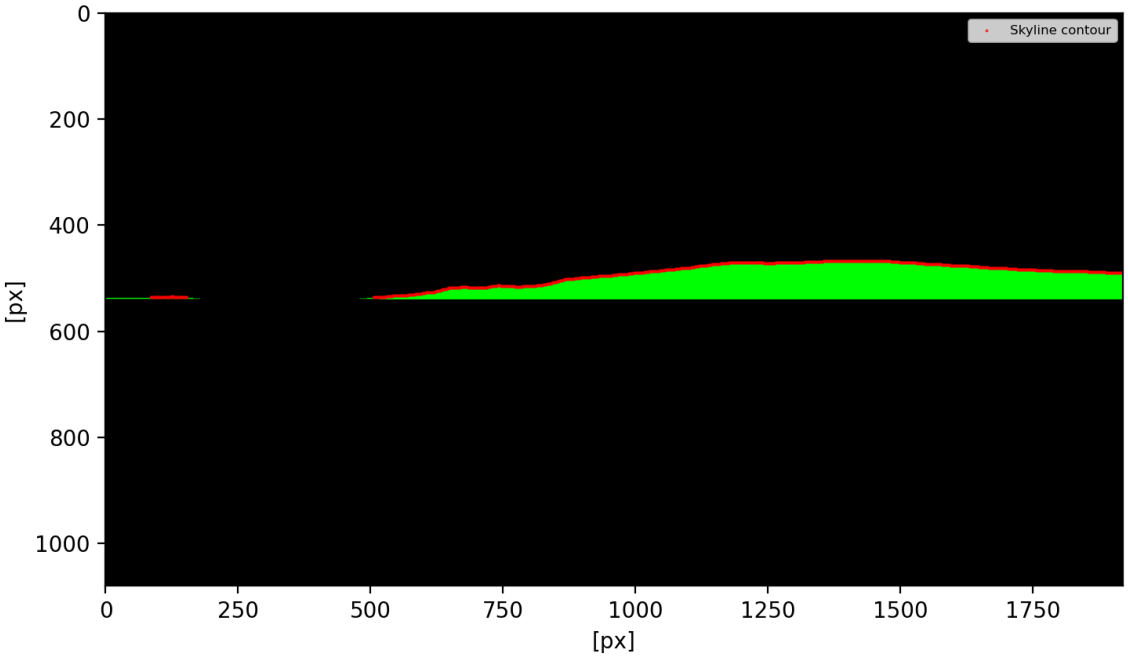


Figure 37: Skyline contour extracted at waypoint 18 along the intended path of path 3. Green pixels in the image are segmented sections of terrain, whilst red points illustrate the extracted skyline.

The performance of the algorithm is attempted illustrated in Figure 38, where the elapsed time of the algorithm for all waypoints of the intended and actual paths of path 2 are plotted, alongside the mean elapsed time. Path 2 was chosen because it represented the worst case scenario amongst

the three paths, having the most visible terrain on average.

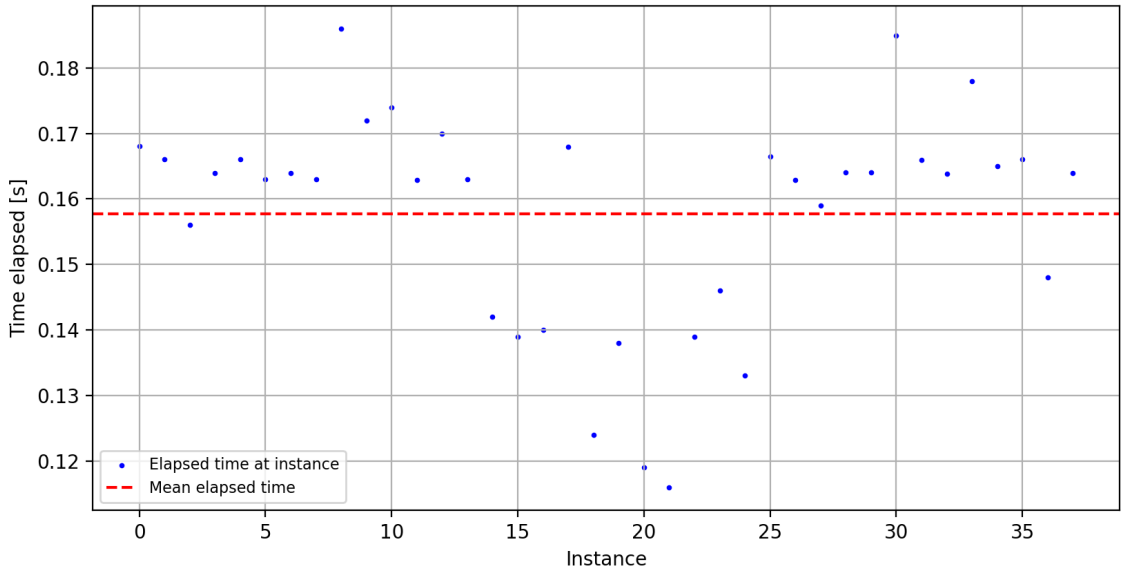


Figure 38: Time elapsed by the skyline contour extraction algorithm over all terrain segmentations of the intended and actual paths of path 2. The mean elapsed time was 0.16s.

5.2.2 Hilltop extraction from skyline contours

Algorithm 2 yielded $\mathcal{K}$ hilltops subject to the strict conditions in Section 3.1.2 for pretty much any contour where $\mathcal{K}$ hilltop(s) fulfilling said conditions exist. Figures 39 and 40 illustrate the algorithm output from a continuous skyline contour with multiple possible features, whilst Figures 41 and 42 illustrate the localization of features in very sparse of uniform contours, respectively. The height and width threshold used when generating these results were 5 and 25 pixels respectively. The subplots contain the input skyline contour as a set of green pixels on the left and all $\mathcal{K}$ extracted hilltops in different colors, for ease of distinction, on the right.

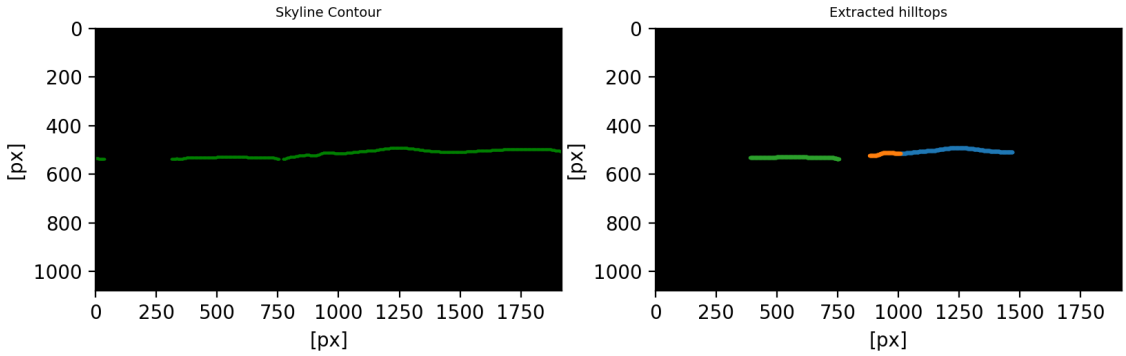


Figure 39: $\mathcal{K}$= 3 hilltops extracted from the abundant skyline contour of the intended path of path 1 at waypoint 1. The input skyline contour is represented as a set of green pixels in the left image, whilst all $\mathcal{K}$ extracted hilltops are represented by different colors in the right.

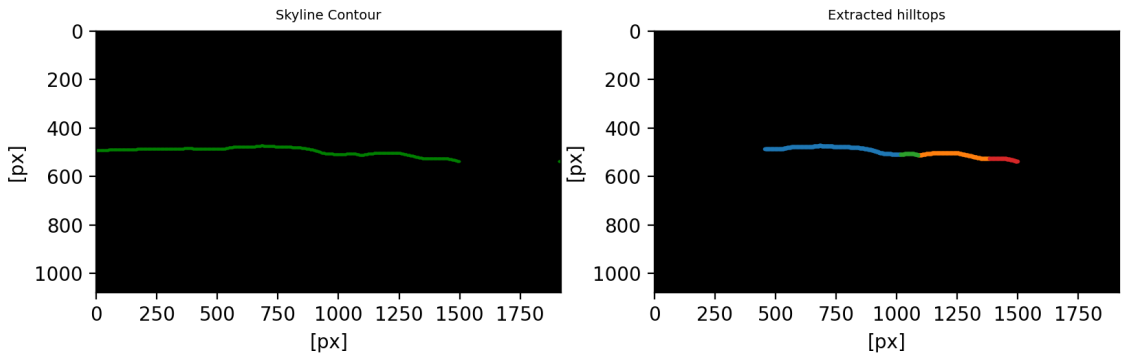


Figure 40: $\mathcal{K}$= 4 hilltops extracted from the abundant skyline contour of the intended path of path 2 at waypoint 13. The input skyline contour is represented as a set of green pixels in the left image, whilst all $\mathcal{K}$ extracted hilltops are represented by different colors in the right.

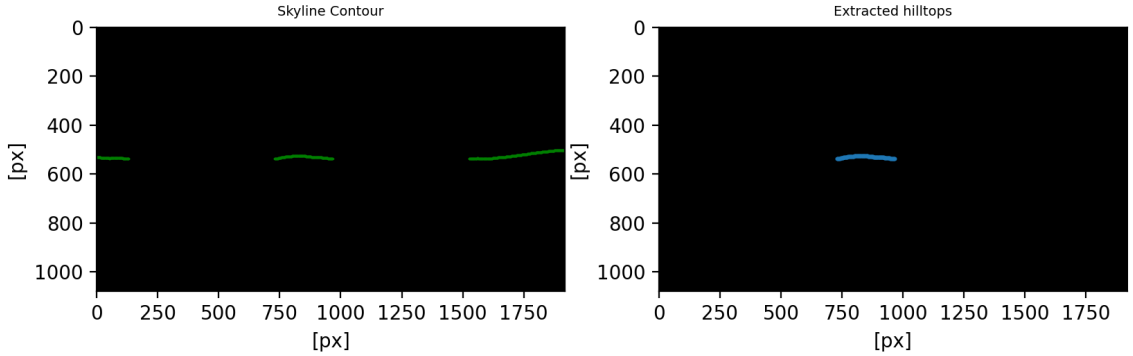


Figure 41: $\mathcal{K}$= 1 hilltop extracted from the sparse skyline contour of the intended path of path 1 at waypoint 14. The input skyline contour is represented as a set of green pixels in the left image, whilst all $\mathcal{K}$ extracted hilltops are represented by different colors in the right.

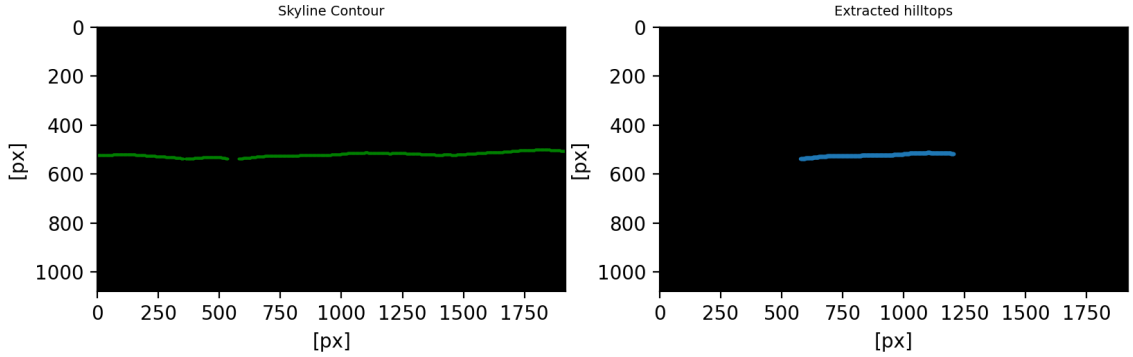


Figure 42: $\mathcal{K}$= 1 hilltop extracted from the very uniform skyline contour of the intended path of path 3 at waypoint 4. The input skyline contour is represented as a set of green pixels in the left image, whilst all $\mathcal{K}$ extracted hilltops are represented by different colors in the right.

At none of the 57 waypoints did the algorithm return infeasible or faulty hilltop-features, except in the cases where no hilltops matching the aforementioned conditions in Section 3.1.2 existed within the skyline contour.

Figure 43 illustrates the elapsed time of the hilltop extraction algorithm over all waypoints of the intended and elapsed time of path 2. As was the case in Figure 38, path 2 yielded the worst case result and is henceforth presented.

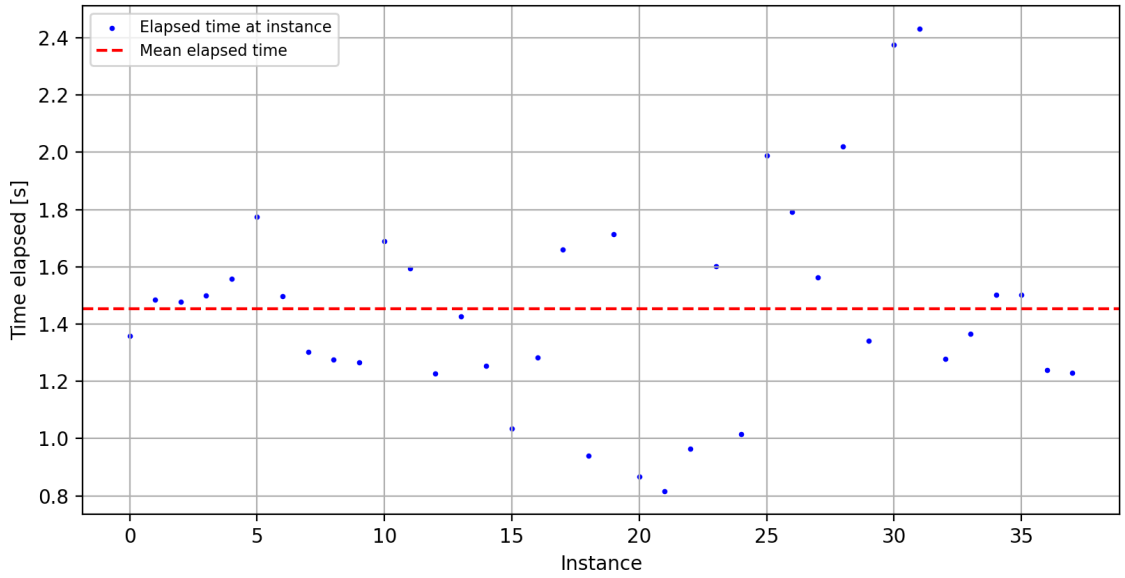


Figure 43: Time elapsed by the hilltop extraction algorithm over all skyline contours found along both the intended and actual paths of path 2. The mean elapsed time was 1.45s.

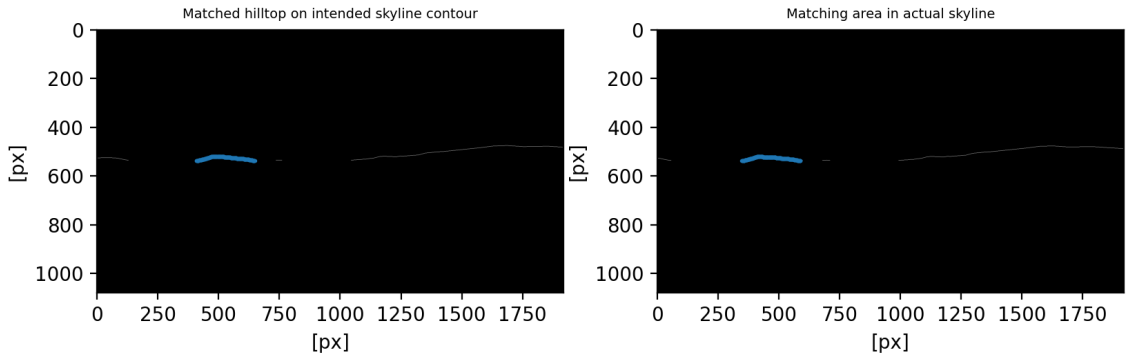
5.2.3 Matching image features between two skyline contours

Algorithm 3 proves to be the first module capable of generating suboptimal and even wrong results. For the sake of precision and transparency, Table 3 is supplied, illustrating how the algorithm-outputs of the 57 waypoints are divided between 5 categories. Correct matches, where the areas in either skyline are, in fact, matching, suboptimal matches, where the areas match but are visible inferior to other matches and ambiguous matches, where the correspondence of located matches is not necessarily clear. Furthermore, failed matches occur where no match between two skylines could be found and wrong matches are matches where two visibly different areas are considered similar. The division is done through visual inspection of the results.

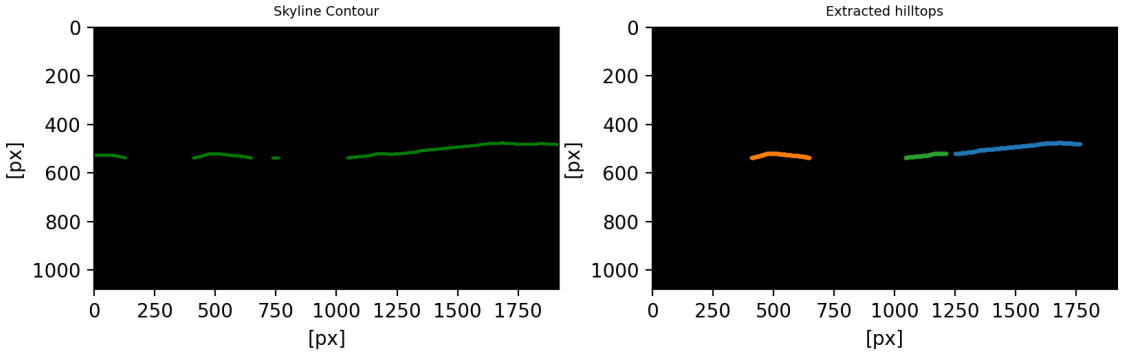
Path	Wrong	Failed	Ambiguous	Suboptimal	Correct	% feasible matches
1	4	1	0	2	12	$\sim 74\%$
2	4	0	1	1	13	$\sim 74\%$
3	1	0	1	0	17	$\sim 89\%$

Table 3: Overview of the performance of Algorithm 3 over the 57 considered waypoints from visual inspection of the results. The matches are classified as 1 of 5 categories: Correct matches are good matches between the two skylines, suboptimal matches are matches that are clearly inferior to other potential matches between the contours and ambiguous matches are matches which are hard to verify through visual inspection. Furthermore, Failed matches are the cases in which no matches can be procured whilst Wrong matches are matches of wrong areas between the two skylines. Feasible matches are finally the combination of correct or suboptimal matches.

To further supply the results in Table 3, visual examples are supplied for each respective category. Figures 44 and 45 illustrate a correct match, where the areas in both skylines correspond and the selected peaks are good with regards to physical parameters such as size, positioning and height. The topmost subfigure illustrate the final matching results between $\vec{f}_{\vec{\omega}_i}$ and $\vec{f}_{\vec{p}_i}$, whilst the lowermost subfigure illustrate the available hilltops found in $S_{\vec{\omega}_i}$:

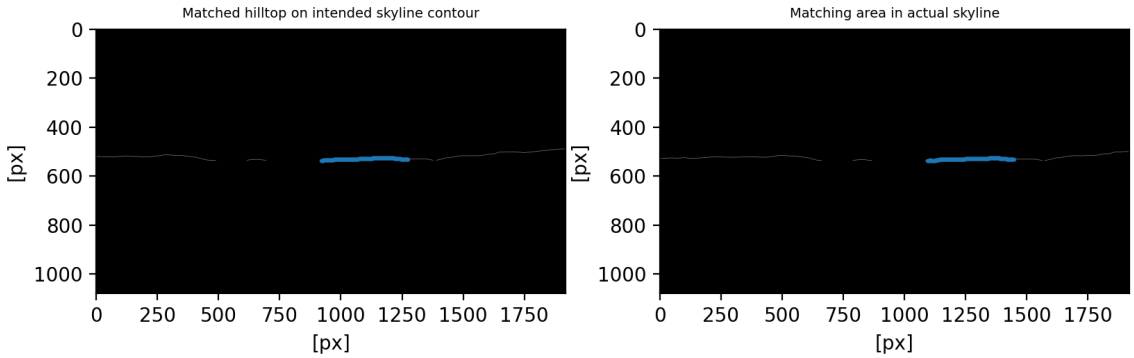


(a) Best image feature matches between the skyline contours captured along both the intended and actual paths.

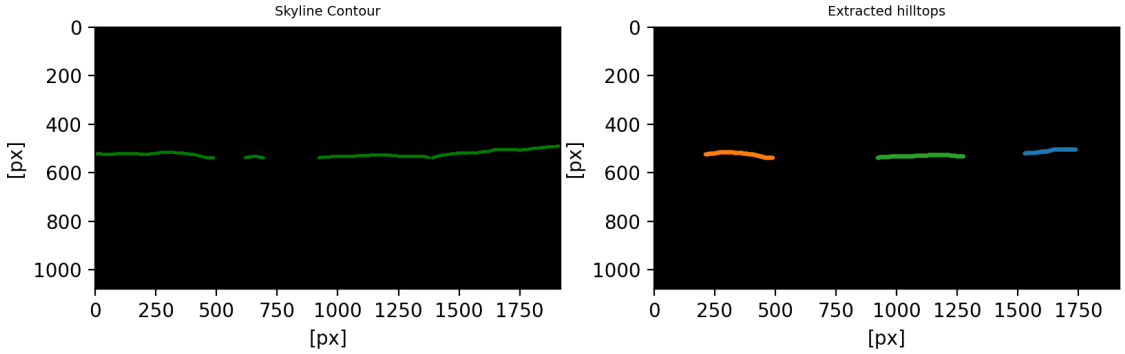


(b) $\mathcal{K}$= 3 hilltops extracted from the intended skyline in (a).

Figure 44: Waypoint 16 of path 3. Correct matching of image features in (a). The most prevalent hilltop in (b) used to find a close to optimal match. Note that in (b), the input skyline contour is represented as a set of green pixels in the left image, whilst all $\mathcal{K}$ extracted hilltops are represented by different colors in the right.



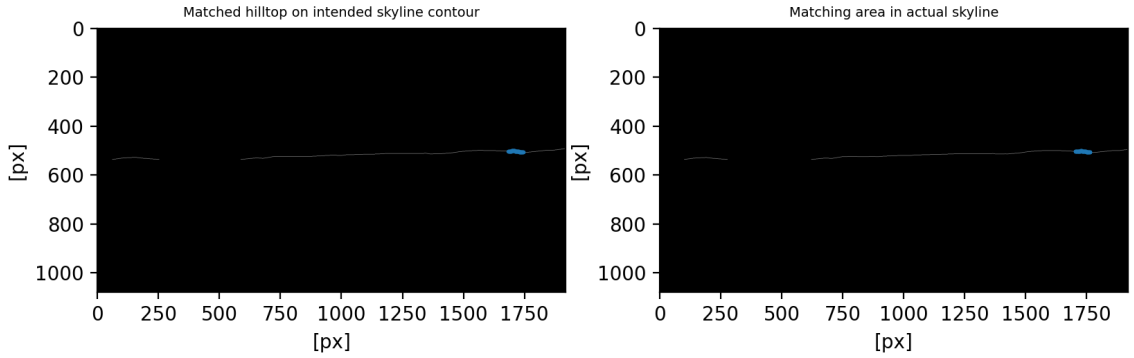
(a) Best image feature matches between the skyline contours captured along both the intended and actual paths.



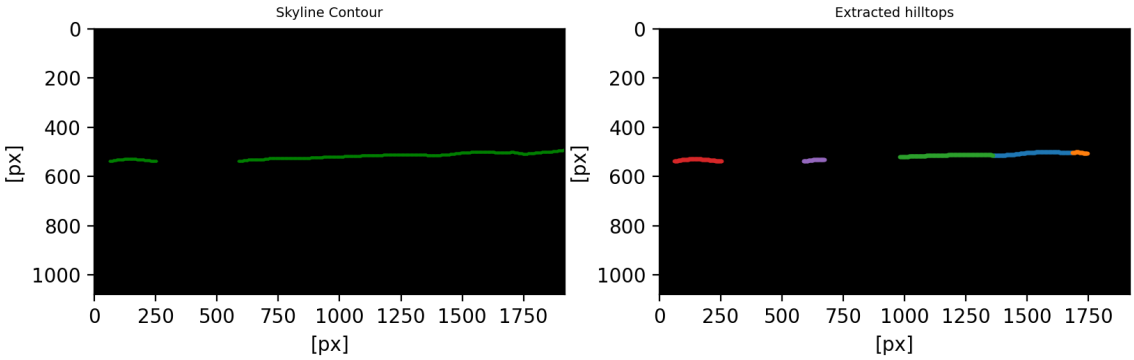
(b) $\mathcal{K}$= 3 hilltops extracted from the intended skyline in (a).

Figure 45: Waypoint 3 of path 1. Correct matching of image features in (a). The largest, most centered hilltop in (b) used to find a close to optimal match. Note that in (b), the input skyline contour is represented as a set of green pixels in the left image, whilst all $\mathcal{K}$ extracted hilltops are represented by different colors in the right.

Figure 46 and 47 on the other hand, illustrate a set of suboptimal matches. These are correct matches for all intents and purposes, but contain areas in the two skylines that are visibly worse than other options.

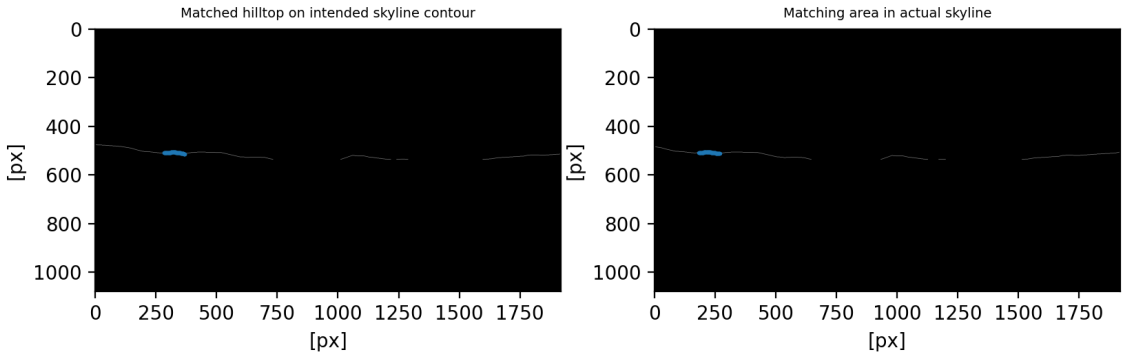


(a) Best image feature matches between the skyline contours captured along both the intended and actual paths.

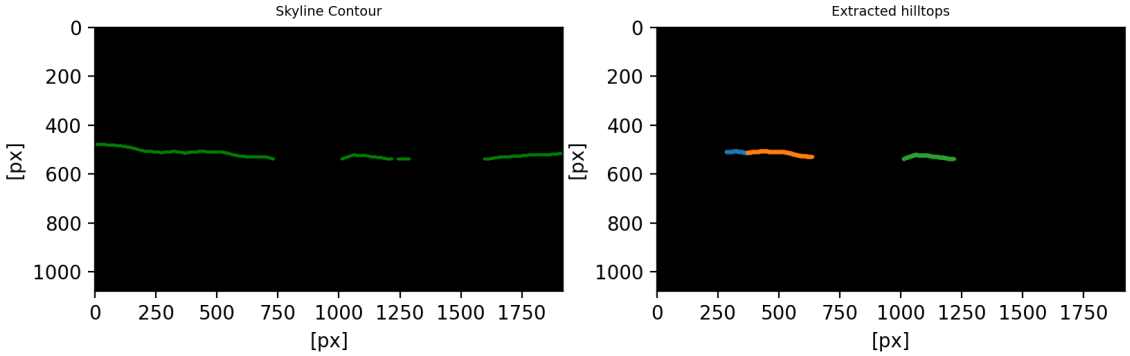


(b) $\mathcal{K}$= 5 hilltops extracted from the intended skyline in (a).

Figure 46: Waypoint 9 of path 1. A set of hilltops are found in (b), of which one the smallest, less centered and less distinct are returned as the located match in (a). The match is correct, but deemed suboptimal. Note that in (b), the input skyline contour is represented as a set of green pixels in the left image, whilst all $\mathcal{K}$ extracted hilltops are represented by different colors in the right.



(a) Best image feature matches between the skyline contours captured along both the intended and actual paths.



(b) $\mathcal{K}$= 3 hilltops extracted from the intended skyline in (a).

Figure 47: Waypoint 11 of path 2. A set of hilltops are found in (b). Arguably the worst of these, being the smallest, least centered and least distinct, is returned as an output of the matching procedure. Note that in (b), the input skyline contour is represented as a set of green pixels in the left image, whilst all $\mathcal{K}$ extracted hilltops are represented by different colors in the right.

Figure 48 and 49 further illustrate a set of ambiguous matches. These can be either correct or wrong, but due to the surrounding terrain and/or the shapes of the matches themselves, their classifications become uncertain upon visual inspection.

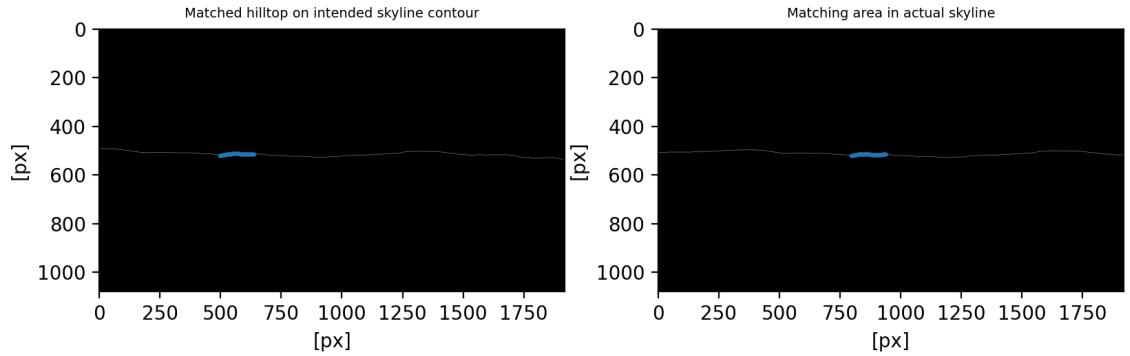


Figure 48: Waypoint 3 of path 2. The unknown, drastic shift in scenery between the two images puts the match into question: Is the actual match correct or on the wrong side of the little hill on the right?

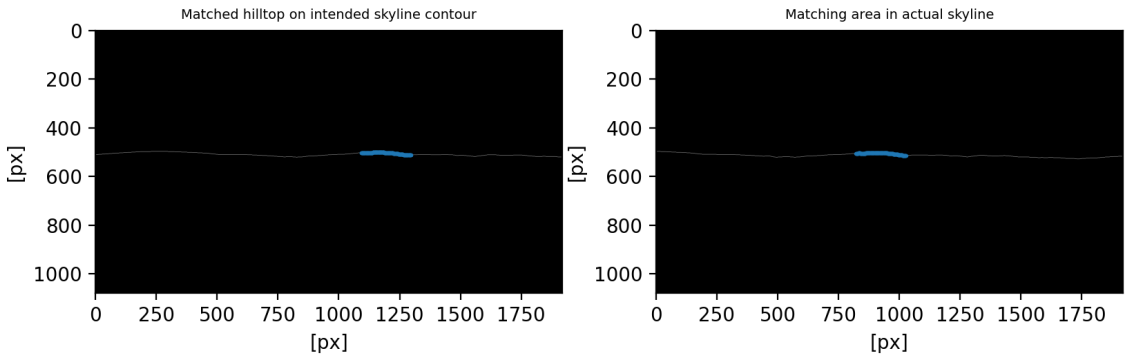


Figure 49: Waypoint 8 of path 3. The uniformity of the terrain makes it tough to distinguish whether or not the match is correct.

Figure 50, 51 and 52 further illustrate the three most common cases of wrong matches: The feature from the intended skyline goes out of frame in the actual skyline, there exist an area with better SAD score in than the actual matching area and feature from the intended skyline is greater than the exact same feature in the actual skyline.

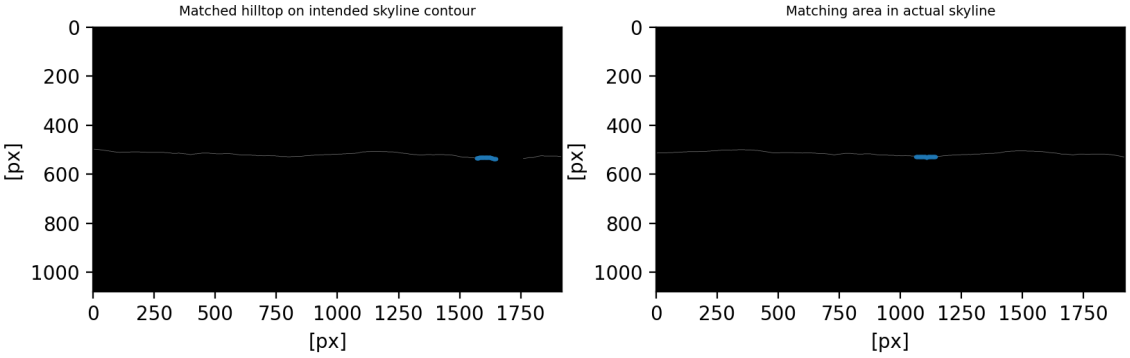


Figure 50: Waypoint 1 path 2. The hilltop in the intended skyline contour goes out of frame, yielding an incorrect match.

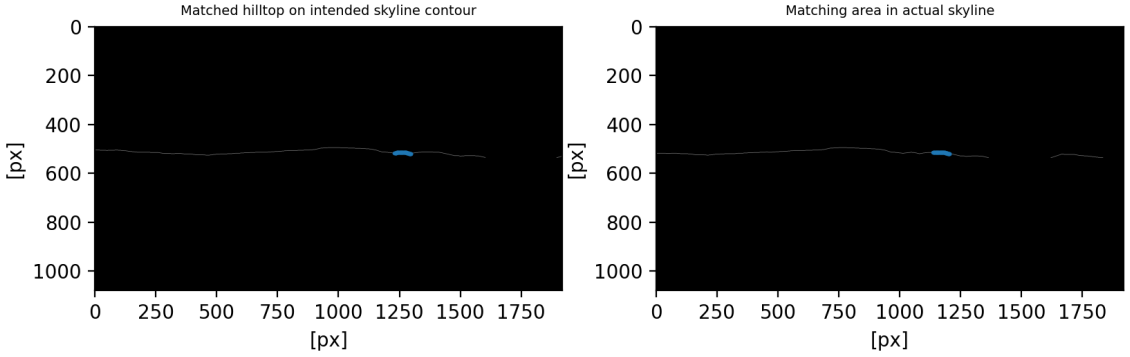


Figure 51: Waypoint 8 path 2. The hilltop in the intended skyline contour is matched with a different area than intended in the actual skyline contour, due to this receiving a better SAD score by chance.

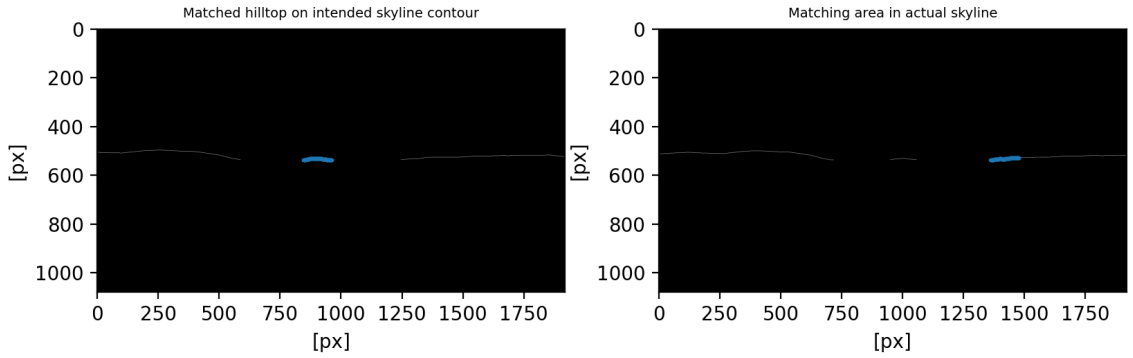
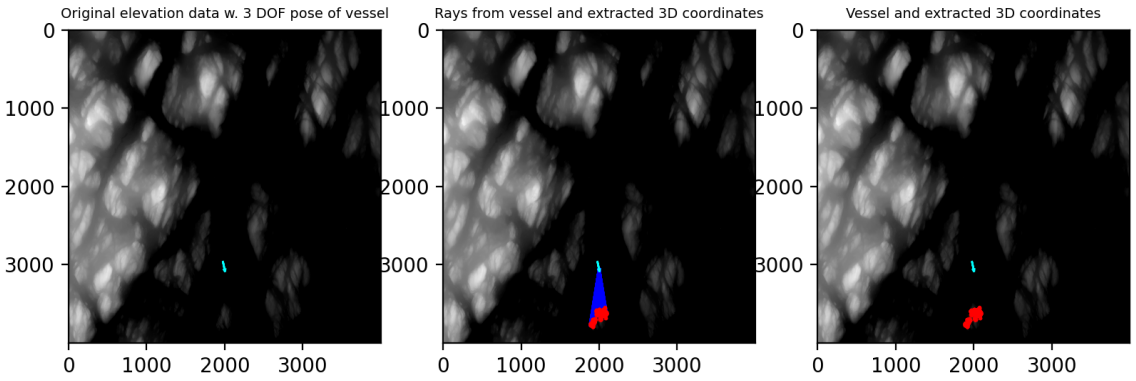


Figure 52: Waypoint 7 path 1. The hilltop in the intended skyline is matched with another area due to the same feature in the actual skyline contour containing fewer pixels. Above, the island in the first image is not matched correctly due to it being perceived as smaller in the actual skyline contour.

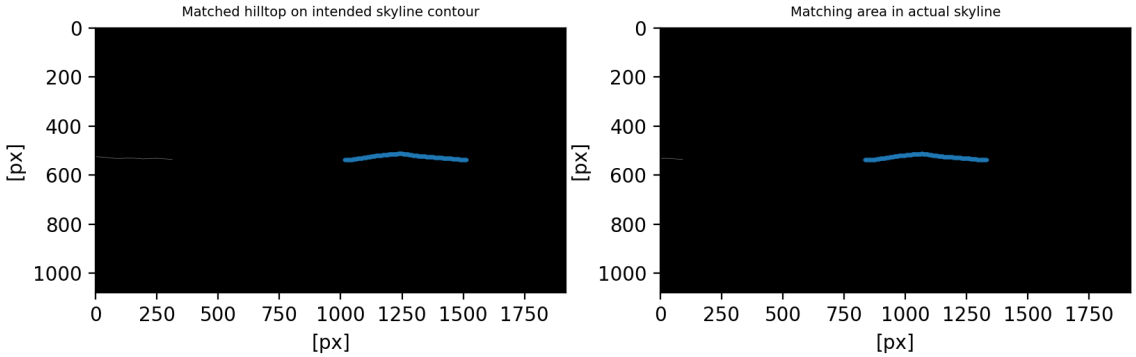
Finally, the image fails if no good matches are deemed to be within sigth. This happened only once for the 57 considered waypoints: At waypoint 19 of path 1, in which no features were visible.

5.3 Locating skyline elements in DEM

Algorithm 4 consistently yielded a set of $\mathcal{L}$ rays in the 2D plane at waypoint i, coinciding accurately with the pixels they stemmed from in $\vec{f}_{\vec{\omega}_i}$. Figures 53, 54 and 55 seek to illustrate the robustness of the algorithm and its respective outputs.

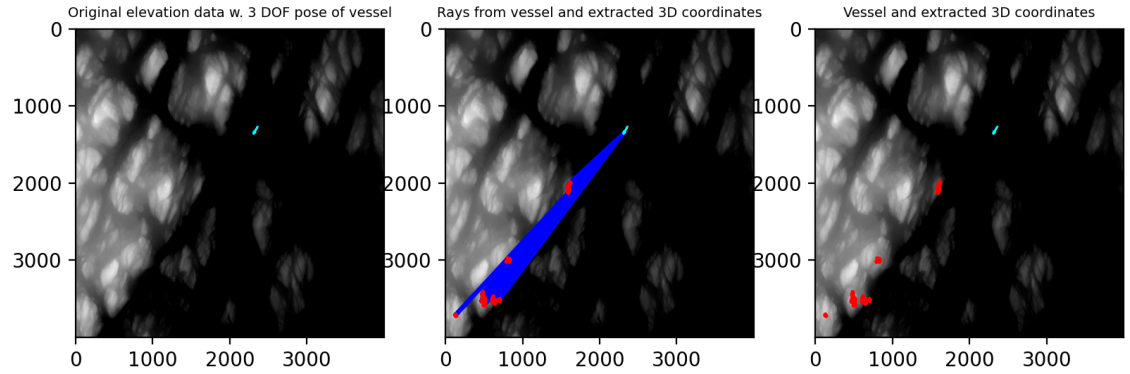


(a) The 3 DOF pose of the maritime vessel, rays in the 2D plane and resulting 3D coordinates of $\vec{f}_{DEM_i}$.

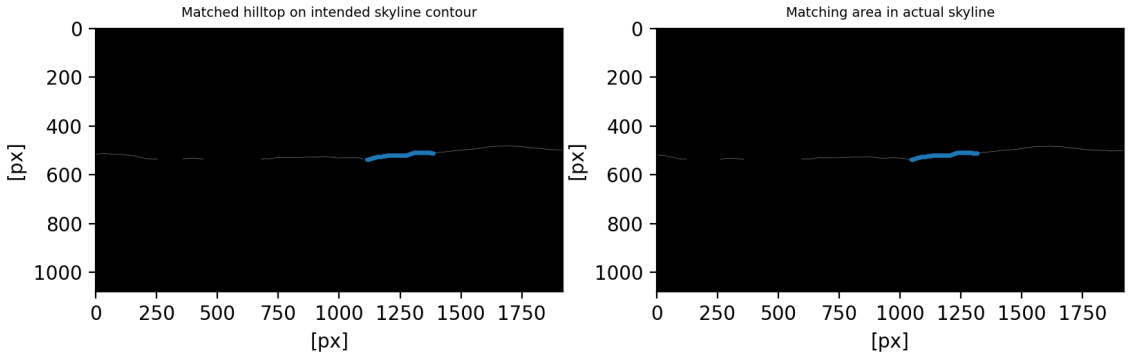


(b) Best image feature matches between the skyline contours captured along both the intended and actual paths.

Figure 53: Waypoint 18 of path 1. The image feature from the intended skyline contour $\vec{f}_{\vec{\omega}_i}$ in (b) serves as the base for the resulting DEM feature $\vec{f}_{DEM_i}$ in (a). It is evident that the isolated island on the intended skyline is accurately extracted in the DEM.

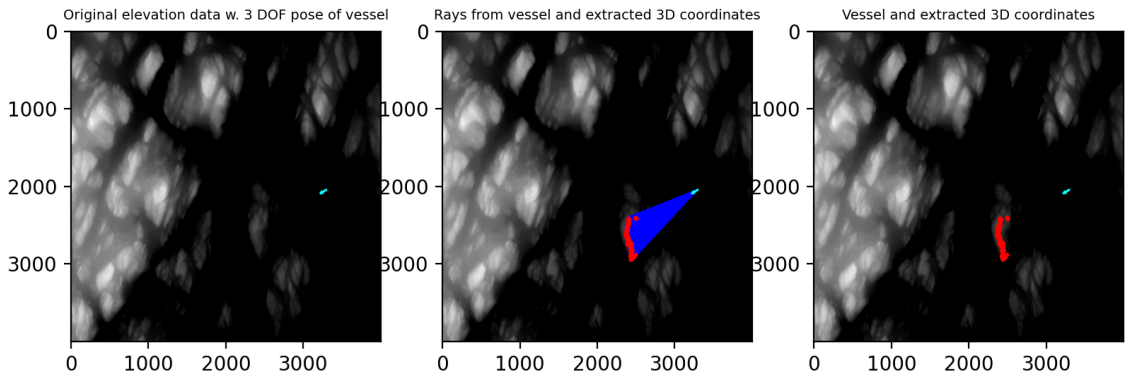


(a) The 3 DOF pose of the maritime vessel, rays in the 2D plane and resulting 3D coordinates of $\vec{f}_{DEM_i}$.

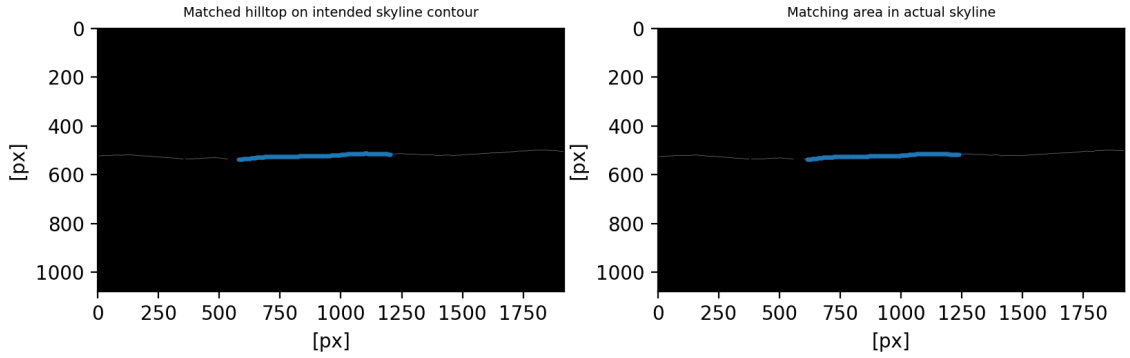


(b) Best image feature matches between the skyline contours captured along both the intended and actual paths.

Figure 54: Waypoint 2 of path 1. The algorithm evidently considers vertices at all depths along the rays, yielding $\vec{f}_{DEM_i}$ in (a) as a diverse set of the most probable matching coordinates in the image feature of the intended skyline contour in (b).



(a) The 3 DOF pose of the maritime vessel, rays in the 2D plane and resulting 3D coordinates of $\vec{f}_{DEM_i}$.



(b) Best image feature matches between the skyline contours captured along both the intended and actual paths.

Figure 55: Waypoint 4 of path 3. The algorithm is evidently capable of finding $\vec{f}_{DEM_i}$ for wider images.

Figure 56 illustrates the elapsed time taken by Algorithm 4 on all waypoints along the intended path of path 3, which was the slowest of the three.

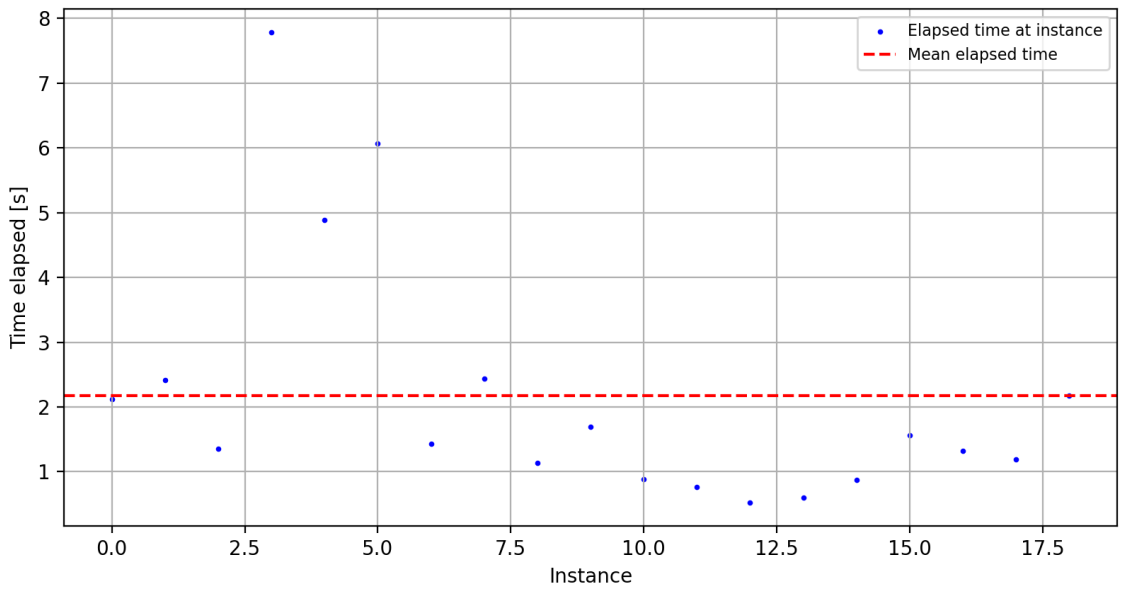
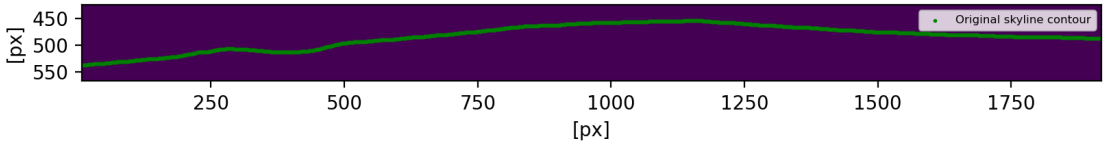


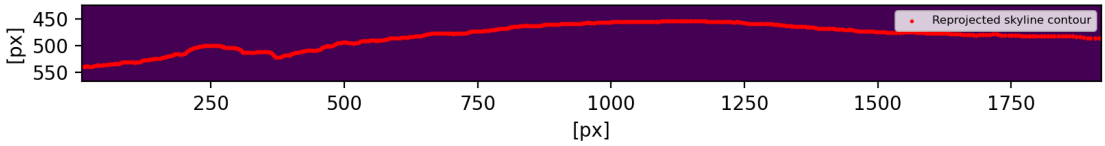
Figure 56: Time elapsed by the dem extraction algorithm on all image features $\vec{f}_{\vec{\omega}_i}$ along the intended path of path 3. The mean elapsed time was 2.17s.

5.4 Reprojection of 3D world coordinates

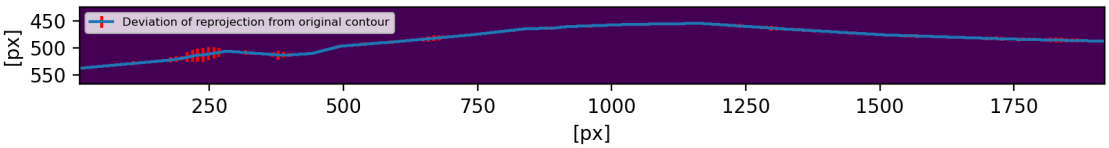
The results of the proposed solution from Sections 3.3 and 4.3.2 yield consistent and accurate results, which also validate the 3D features $\vec{f}_{DEM_i}$ from Section 5.3.

Figures 57, 58 and 59 illustrate the robustness of the reprojection scheme by reprojecting the entire skyline contour $S_{\vec{\omega}_i}$ back to the image plane. This can be done by using Algorithm 4 on the entirety the skyline, $\vec{f}_{\vec{\omega}_i} = S_{\vec{\omega}_i}$, allowing for a general analysis of the reprojection across the entire image plane. Special attention should be placed on the supplied error plots, illustrating the vertical deviation between original and reprojected skyline contours.



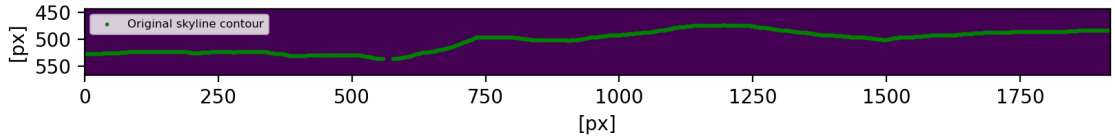


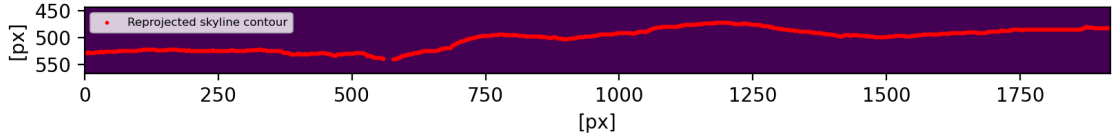
(a) Original and reprojected skyline.



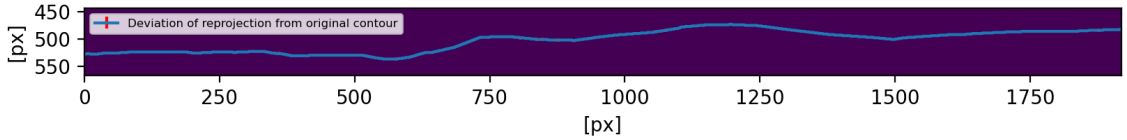
(b) Errorbar plot showcasing the deviation between the two contours at every 10'th pixel.

Figure 57: Reprojection of the entire intended skyline contour $S_{\vec{\omega}_i}$ at waypoint 19 of path 3. (a) illustrates the two skylines side by side. (b) illustrates the vertical deviation at 10 pixel intervals between the contours.



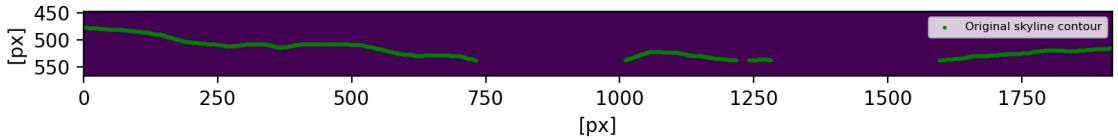


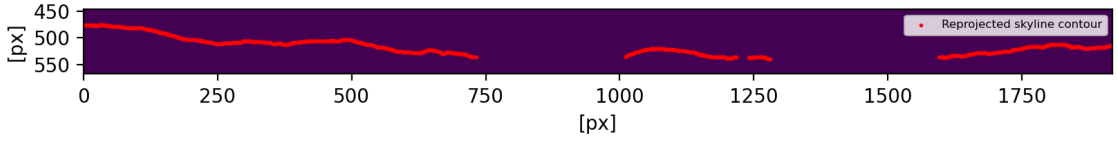
(a) Original and reprojected skyline.



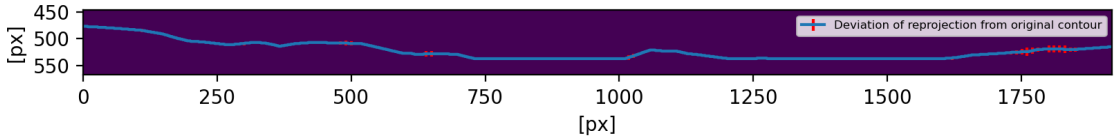
(b) Errorbar plot showcasing the deviation between the two contours at every 10'th pixel.

Figure 58: Reprojection of the entire intended skyline contour $S_{\vec{\omega}_i}$ at waypoint 18 of path 2. (a) illustrates the two skylines side by side. (b) illustrates the vertical deviation at 10 pixel intervals between the contours.





(a) Original and reprojected skyline.

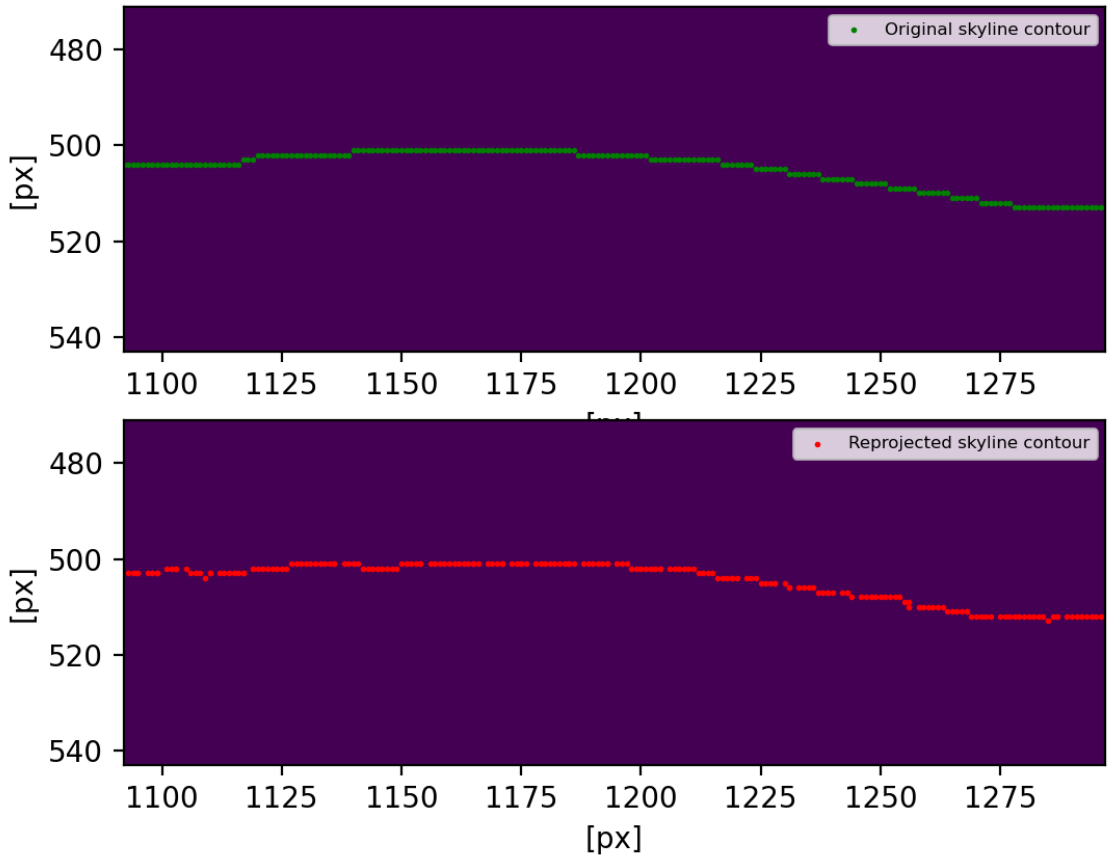


(b) Errorbar plot showcasing the deviation between the two contours at every 10'th pixel. Notice that *NULL*-values are defaulted to $y = 0$.

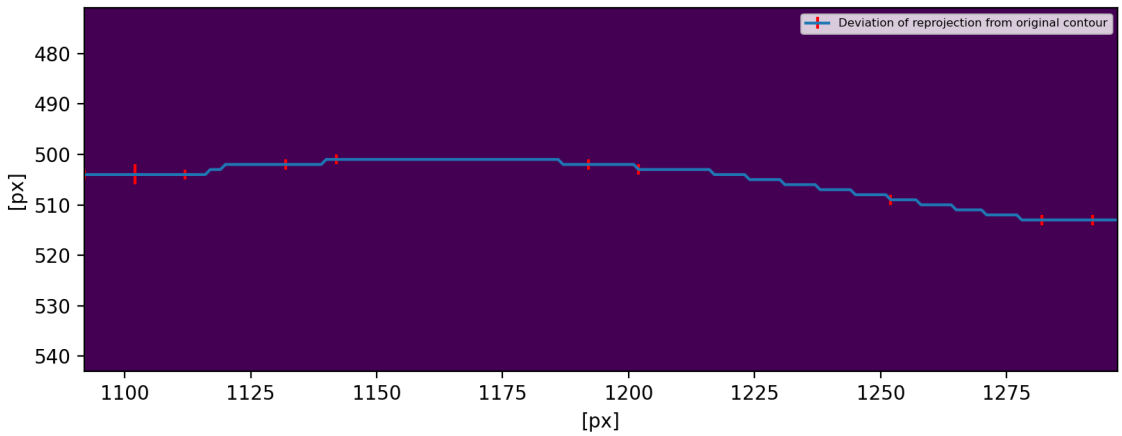
Figure 59: Reprojection of the entire intended skyline contour $S_{\vec{\omega}_i}$ at waypoint 11 of path 2. (a) illustrates the two skylines side by side. (b) illustrates the vertical deviation at 10 pixel intervals between the contours.

Figures 61 and 60 further illustrate the reprojection of selected features $\vec{f}_{\vec{\omega}_i}$. It should be noted that wrong matches, as defined in Section 5.2.3, do not affect the reprojection of $\vec{f}_{DEM_i}$ found

from $\vec{f}_{\vec{\omega}_i}$.

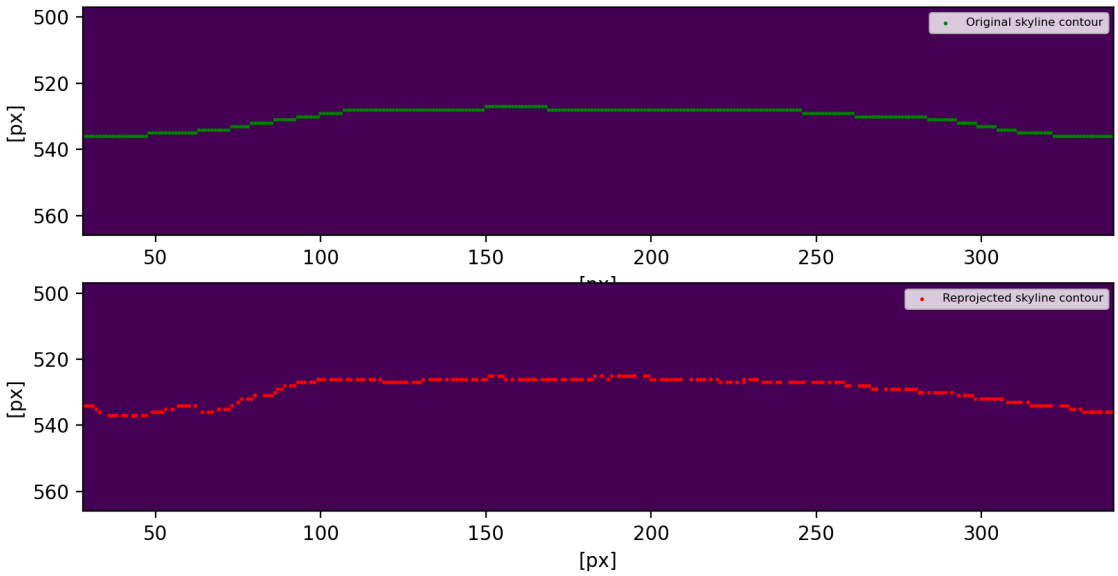


(a) Image feature $\vec{f}_{\vec{\omega}_i}$ and reprojected DEM feature $\vec{f}_{DEM_i}$.

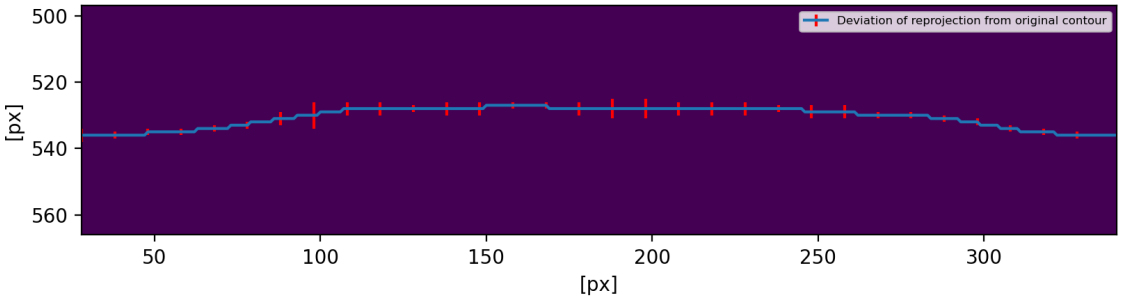


(b) Errorbar plot showcasing the deviation between the two features at every 10'th pixel.

Figure 60: Reprojection of the DEM feature at waypoint 8 of path 3. (a) illustrates the image feature $\vec{f}_{\vec{\omega}_i}$ and reprojection of $\vec{f}_{DEM_i}$ side by side. (b) illustrates the vertical deviation at 10 pixel intervals between features.



(a) Image feature $\vec{f}_{\vec{\omega}_i}$ and reprojected DEM feature $\vec{f}_{DEM_i}$.

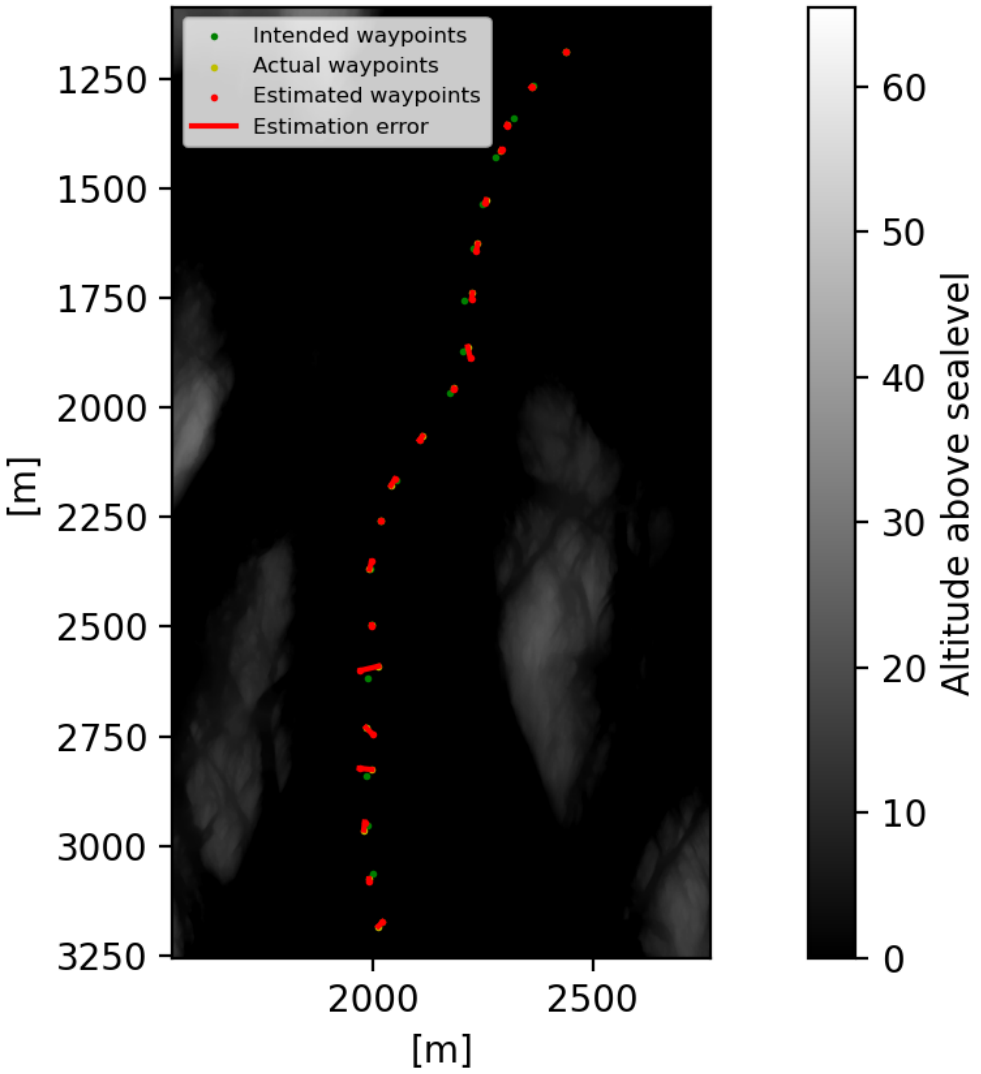


(b) Errorbar plot showcasing the deviation between the two features at every 10'th pixel.

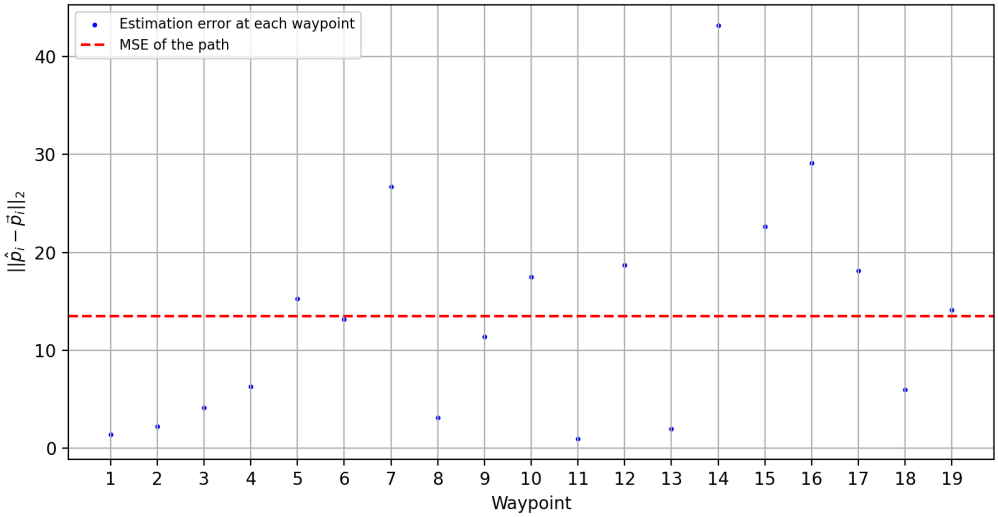
Figure 61: Reprojection of the DEM feature at waypoint 13 of path 1. (a) illustrates the image feature $\vec{f}_{\vec{\omega}_i}$ and reprojection of $\vec{f}_{DEM_i}$ side by side. (b) illustrates the vertical deviation at 10 pixel intervals between features.

5.5 Pose estimation

The results of the adaptive grid-search pose-estimator of Section 3.4, with $t_0 = 32$, for all paths can be seen in Figures 62, 63 and 64, including the mean squared error (MSE) recorded between all n waypoints. The topmost subplots contain all three positions, $\vec{\omega}_i$, $\vec{p}_i$ and $\hat{p}_i$, for each waypoint, as well as a line to connect each $\hat{p}_i$ to it's respective ground-truth $\vec{p}_i$. The second subplots illustrate the estimation error in euclidean distance for all waypoints of the presented path compared to the total path's MSE:

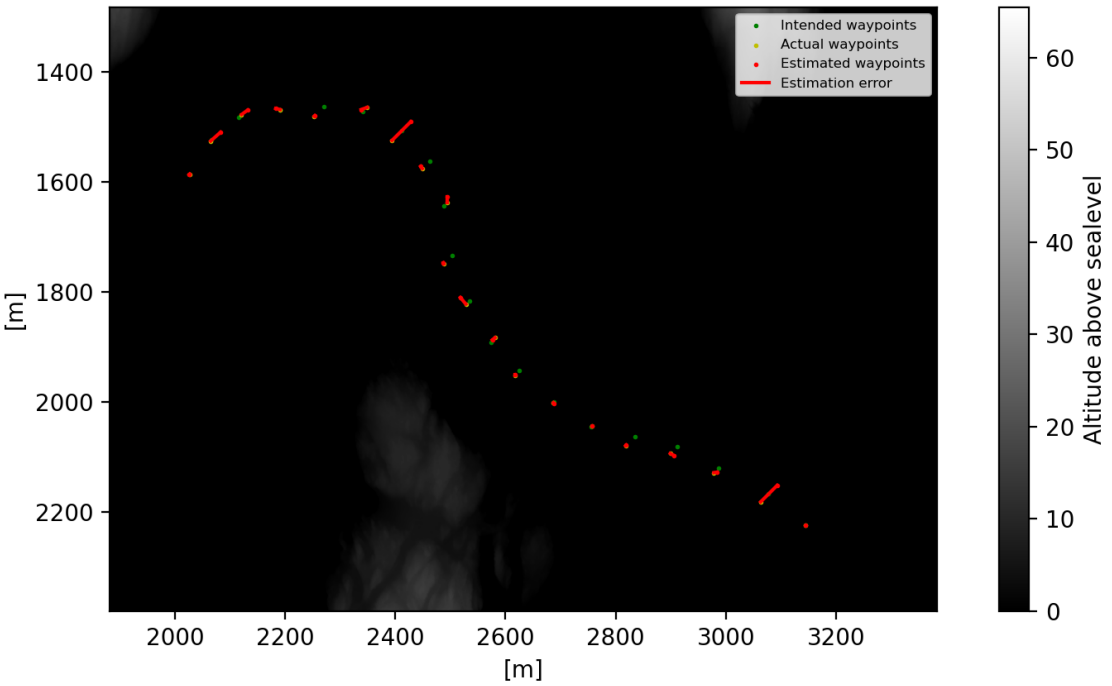


(a) Results of position estimations for the entirety of path 1.

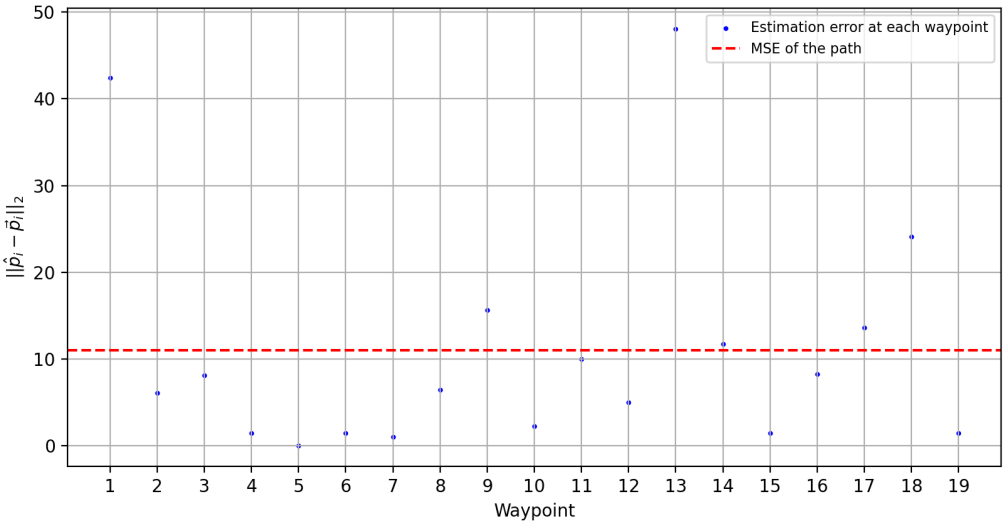


(b) Estimation error in euclidean distance at each waypoint of path 1. MSE = 13.48

Figure 62: The estimated points of path 1 compared to the actual and intended path in (a), with the estimation error in euclidean distance between each waypoint in (b).

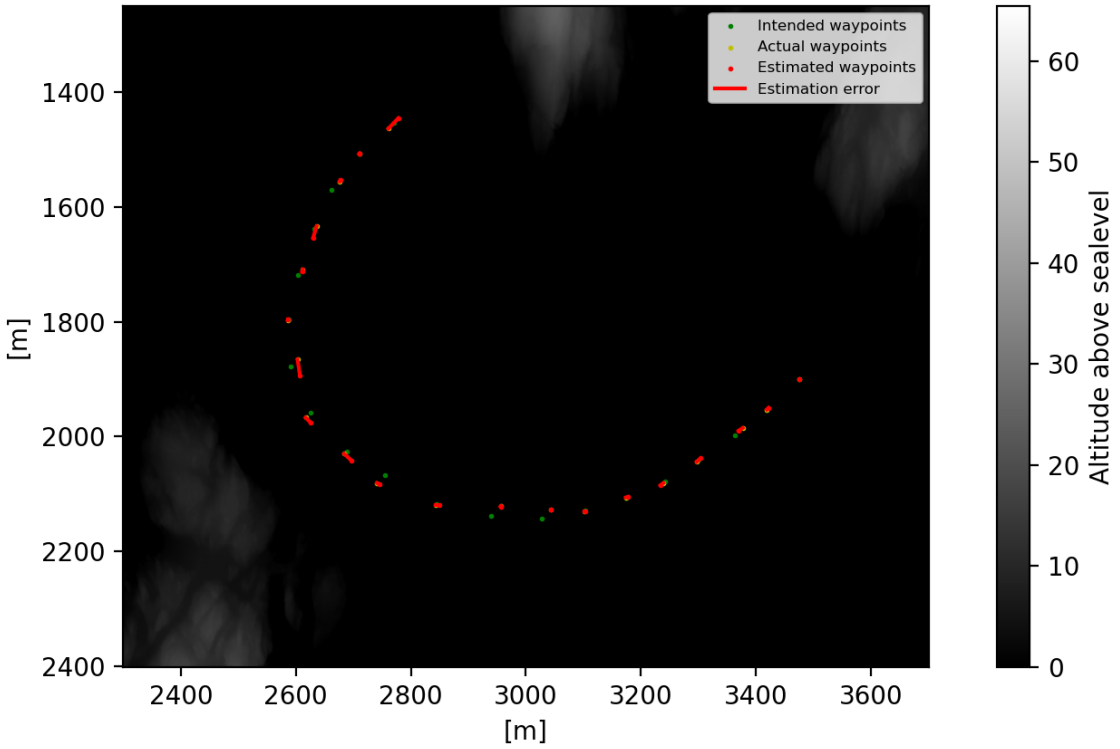


(a) Results of position estimations for the entirety of path 2.

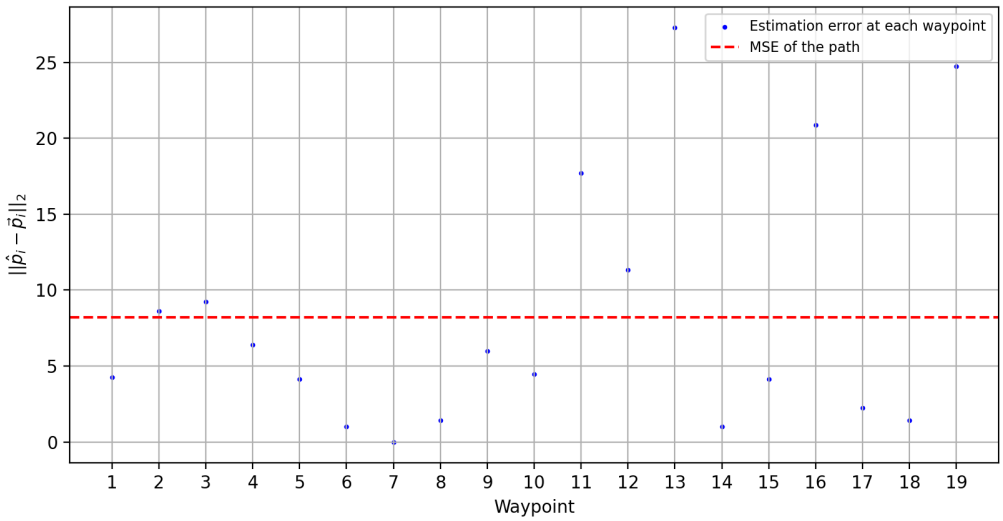


(b) Estimation error in euclidean distance at each waypoint of path 2. MSE = 10.96

Figure 63: The estimated points of path 2 compared to the actual and intended path in (a), with the estimation error in euclidean distance between each waypoint in (b).



(a) Results of position estimations for the entirety of path 3.

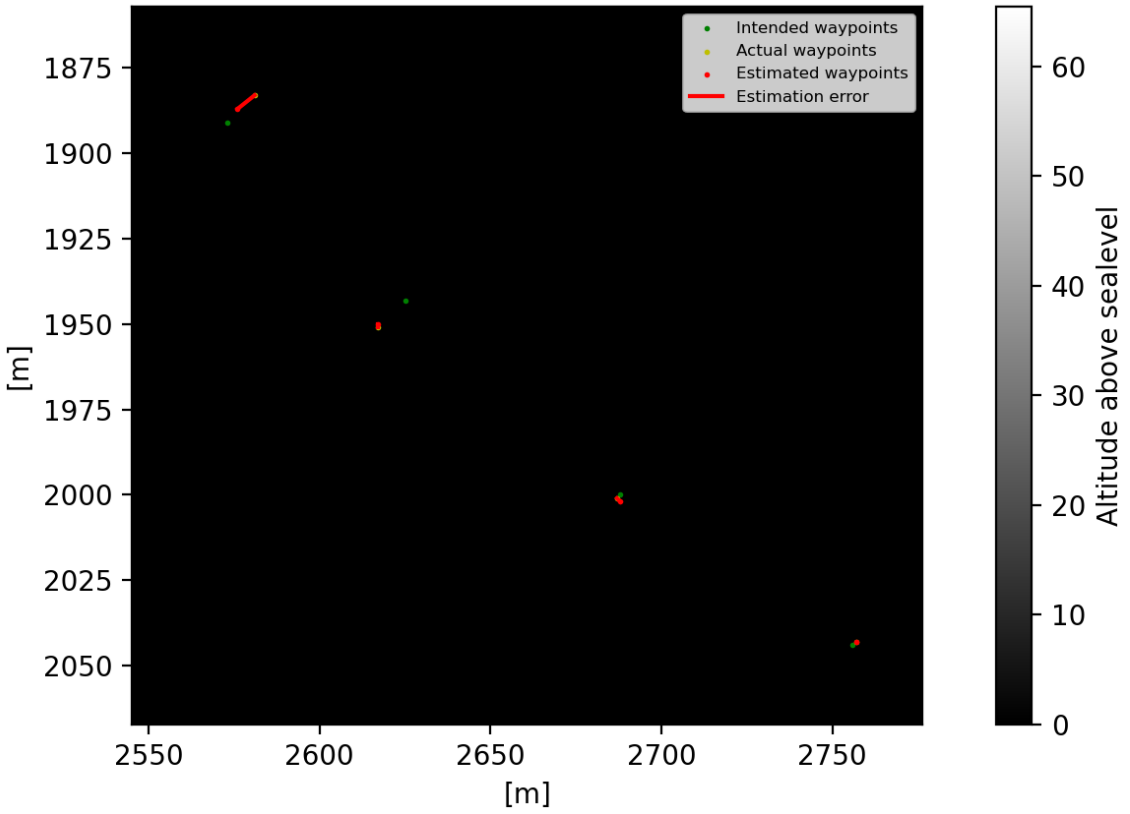


(b) Estimation error in euclidean distance at each waypoint of path 3. MSE = 8.22

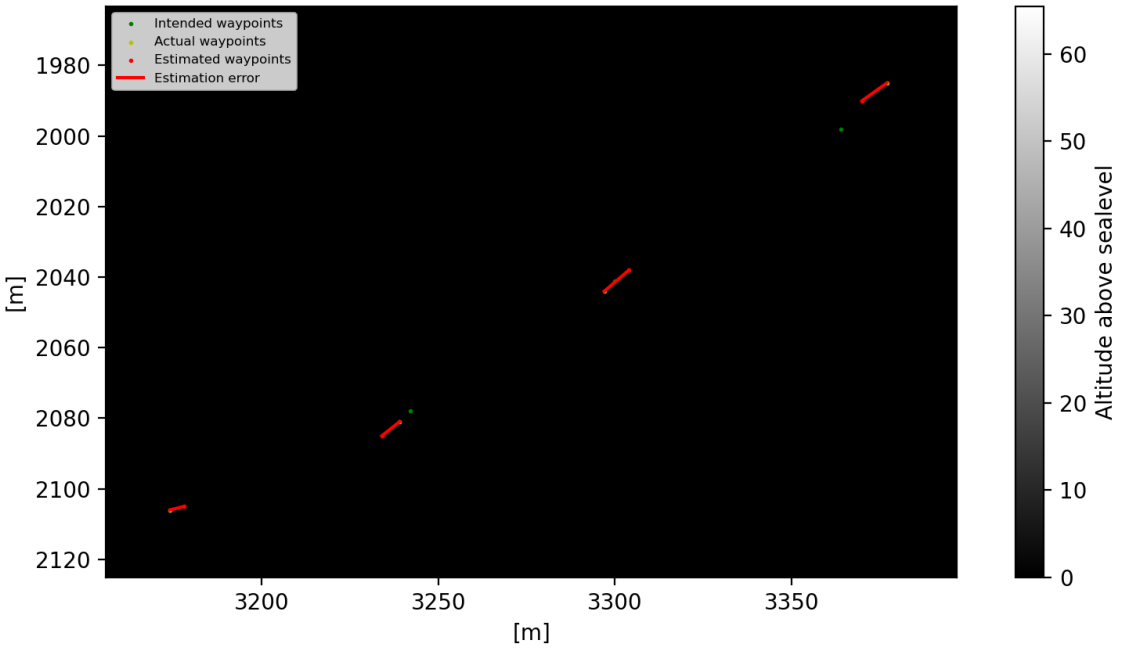
Figure 64: The estimated points of path 3 compared to the actual and intended path in (a), with the estimation error in euclidean distance between each waypoint in (b).

For a better understanding of the results, a closer look at known waypoints and their optimizations are warranted. Firstly, a set of waypoints with known correct matching between image-features, as defined in Section 5.2.3, are analyzed. Figure 65 illustrate close-ups of two sets of waypoints in paths 2 and 3, whilst Figure 67, 66 and 68 summarize the general estimation results through a select few of these waypoints. This is done by looking at the iterations of the estimation procedure with respects to both the position and reprojection of $\vec{f}_{DEM_i}$, from the initial position $\vec{\omega}_i$ to the final estimation $\hat{p}_i$, before comparing them to the target values for position $\vec{p}_i$ and reprojection $\vec{f}_{\vec{p}_i}$. The initial position $\vec{\omega}_i$ and reprojection of $\vec{f}_{DEM_i}$ are illustrated by green pixels. The iterations towards the final estimation are visualized by a color-gradient from green to yellow, with yellow

being the final estimation. The target position $\vec{p}_i$ and reprojection $\vec{f}_{\vec{p}_i}$ are (red):



(a) $\vec{\omega}_i$, $\vec{p}_i$ and $\hat{p}_i$ for $i \in [5, 8]$ of path 2.



(b) $\vec{\omega}_i$, $\vec{p}_i$ and $\hat{p}_i$ for $i \in [3, 6]$ of path 3.

Figure 65: Pose estimation of waypoints with correct matches between image-features.

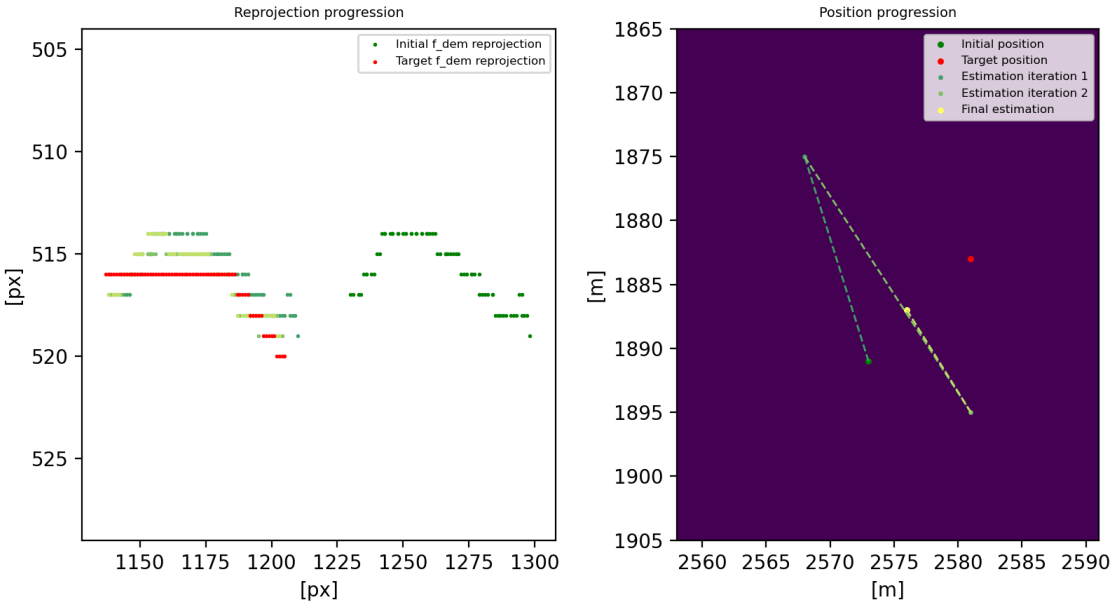


Figure 66: Pose-estimation for waypoint $i = 8$ of path 2. The estimation slowly converges closer to the actual solution, but misses due to the deviations between the image feature $\vec{f}_{\vec{p}_i}$ and the reprojection of $\vec{f}_{DEM_i}$

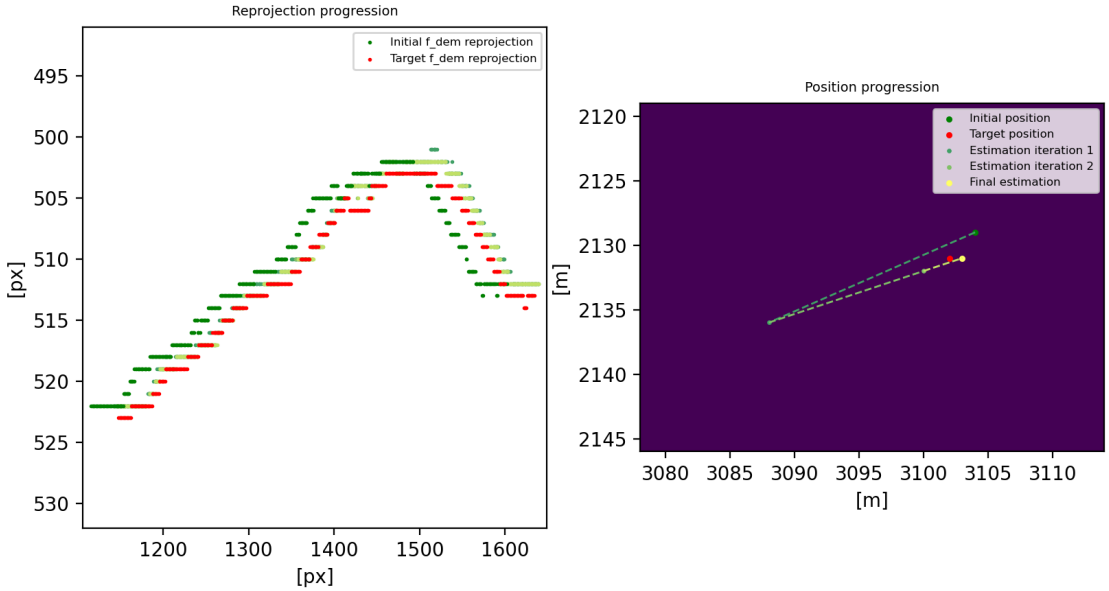


Figure 67: Pose-estimation for waypoint $i = 6$ of path 3. The estimation readjusts after a massive initial leap, successfully locating the target position up to reasonable degree.

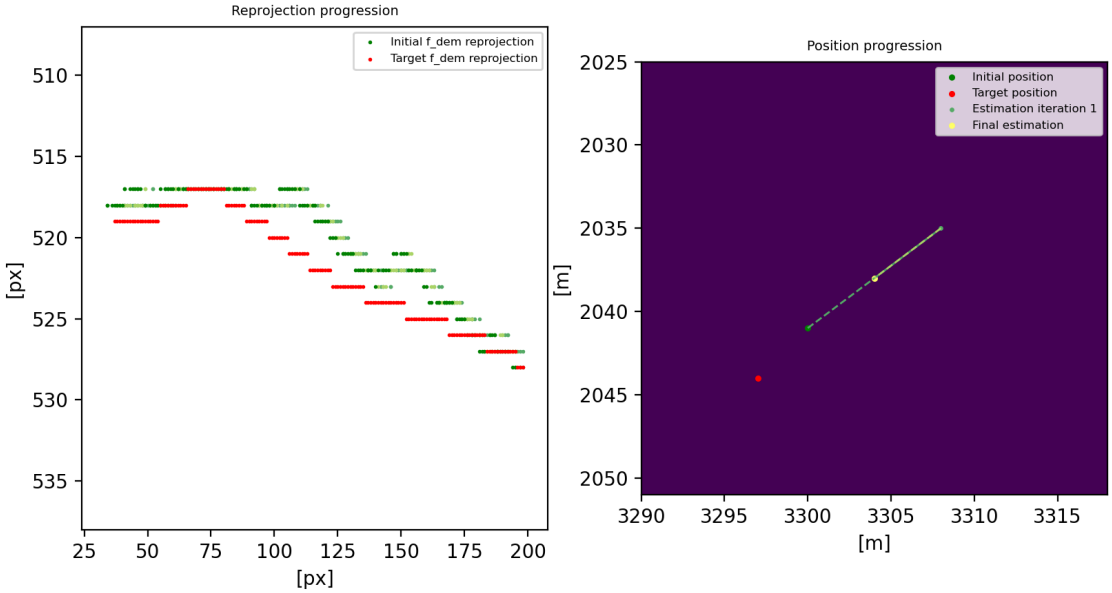
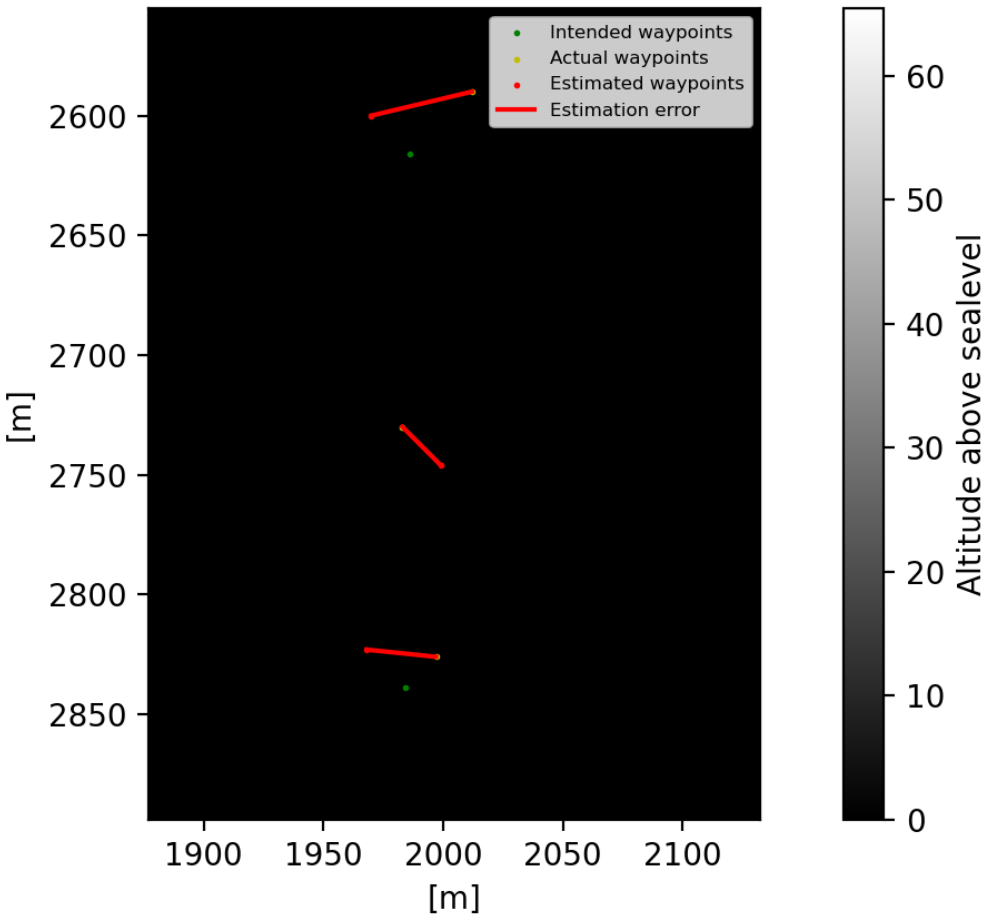
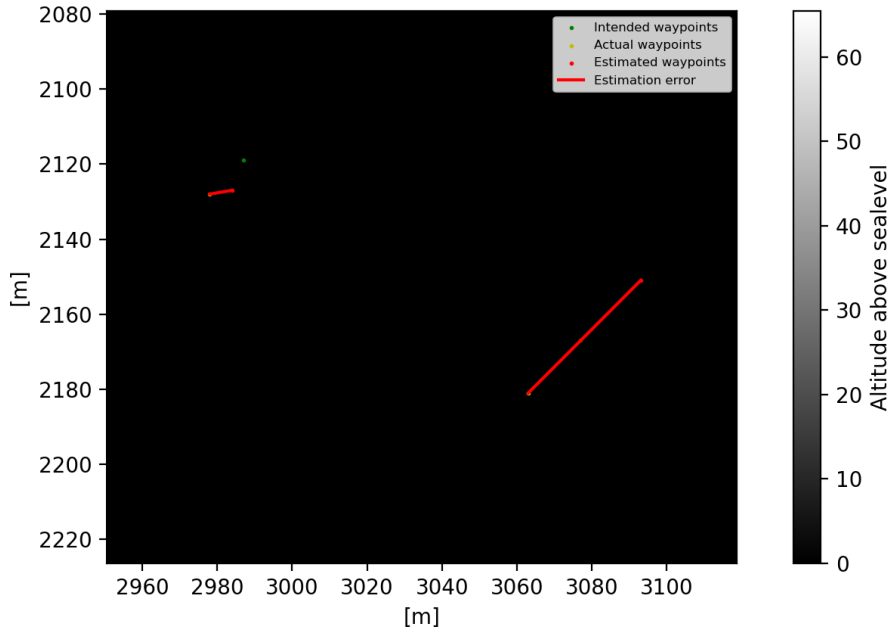


Figure 68: Pose-estimation for waypoint $i = 3$ of path 3. Note how the result deviates from the target position, although the feature is fairly well matched.

Similarly, one can analyze the results from a set of waypoints with known erroneous matches between image features to get an idea of the effect this has on the final estimation. Figure 69 illustrate two such sets from paths 1 and 2, whilst Figures 70 and 71 summarize the general estimation results for a selection of these waypoints:



(a) $\vec{\omega}_i$, $\vec{p}_i$ and $\hat{p}_i$ for $i \in [14, 16]$ of path 1.



(b) $\vec{\omega}_i$, $\vec{p}_i$ and $\hat{p}_i$ for $i \in [1, 2]$ of path 2.

Figure 69: Pose estimation of waypoints with incorrect matches between image-features.

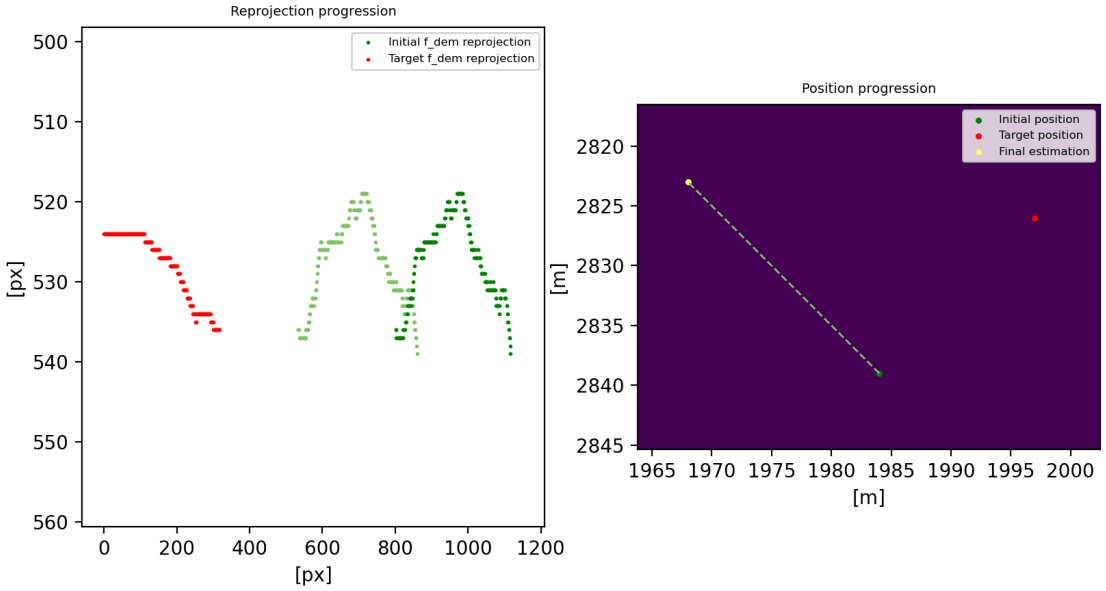


Figure 70: Pose-estimation for waypoint $i = 16$ of path 1. The estimation quickly diverges from the target position $\vec{p}_i$ to lessen the reprojection error to the incorrect match, but is not able to further converge.

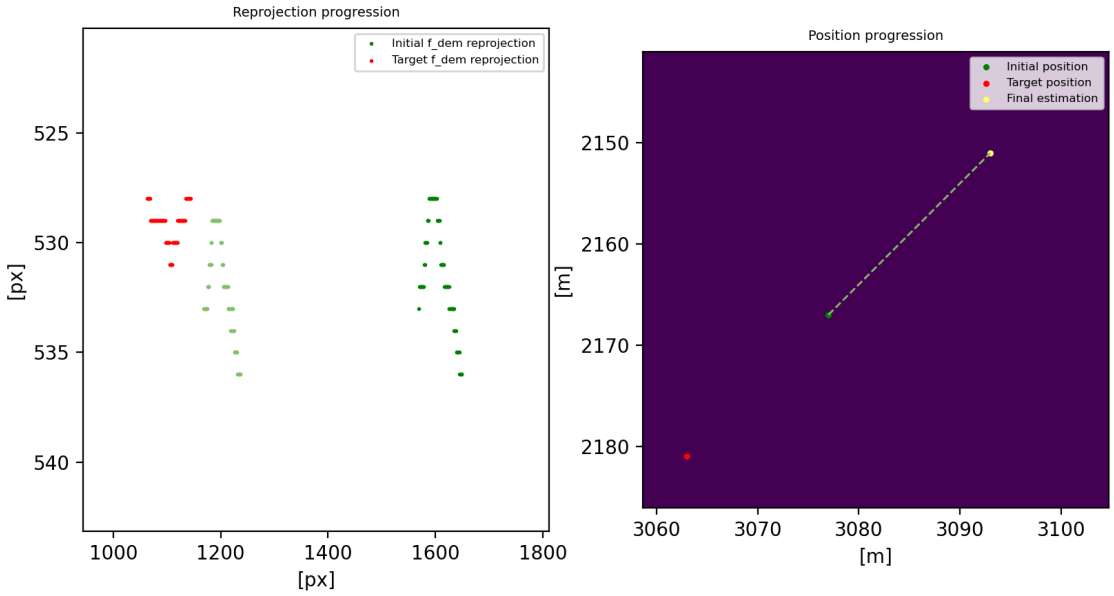


Figure 71: Pose-estimation for waypoint $i = 1$ of path 2. The estimated position $\hat{p}_i$ quickly diverges in the opposite direction of the target position $\vec{p}_i$, given the vastly different shape of image feature $\vec{f}_{\vec{p}_i}$ and reprojection of $\vec{f}_{DEM_i}$.

Finally, the elapsed time of the estimator on all waypoints of path 1 can be seen in Figure 72. Path 1 was chosen as it provided the worst results of the 3, illustrating the worst case scenario.

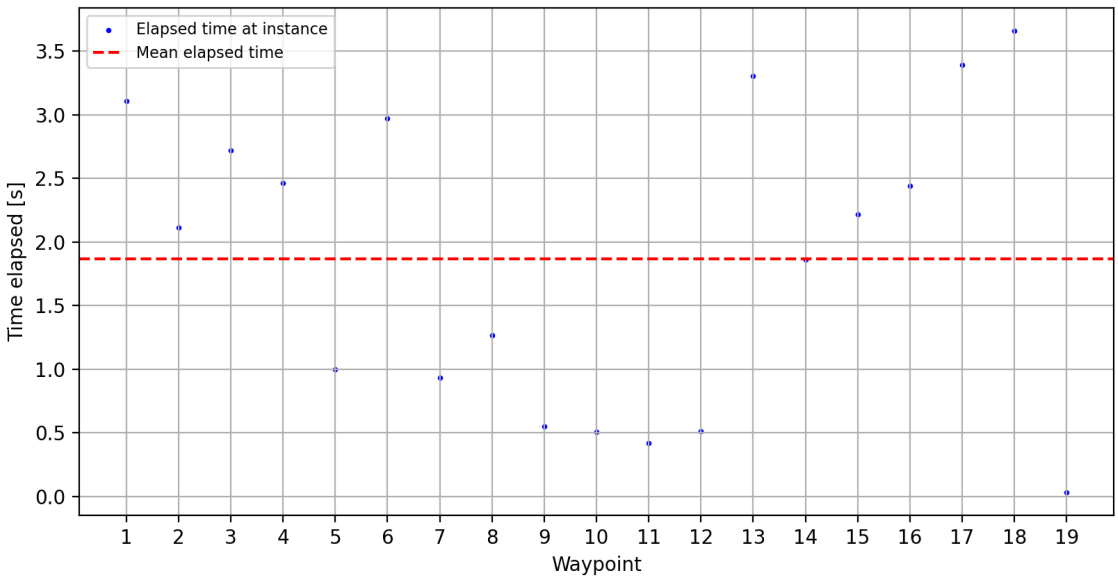


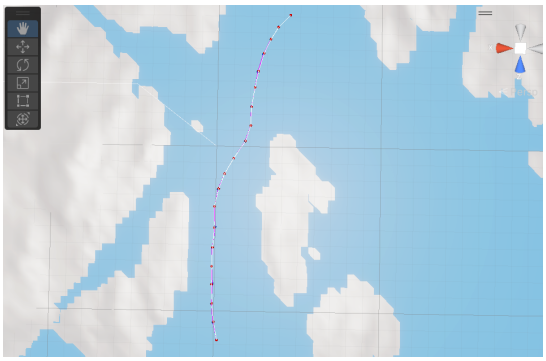
Figure 72: Time elapsed by the pose estimator at each waypoint of path 1. The mean elapsed time was 1.87s.

5.6 Estimator performance on trajectories with varying noise

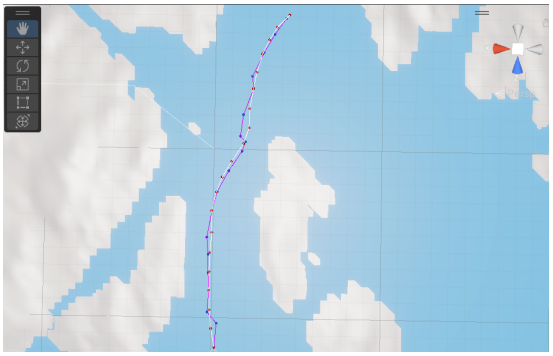
In the case of the presented pose estimator, a logical way to evaluate its performance is to analyze how much its estimation $\hat{p}_i$ is to prefer over a grossly simplified estimation, returning the intended position $\vec{\omega}_i$ at waypoint i, i.e., $\hat{p}_i = \vec{\omega}_i$. The performance can be analyzed through the mean squared error (MSE), which is calculated both when the estimator works as intended and in the more simplified state. For ease of notation, the simplified estimation can be denoted the *existing error*.

To further diversify and generalize the performance measures, the MSE of the estimator and existing errors are analyzed over a set of different values for the Gaussian noise generating the actual path. The mean μ is kept constant at 0 at all iterations, but the variance ϕ changes between a set of 4 values: 10, 70, 120 and 170, in addition to 35 used in the above results.

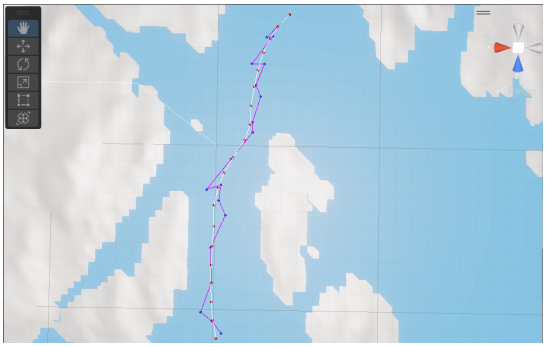
Figure 73 illustrates how path 1 is affected by the 4 additional values of ϕ and denotes existing errors between the intended and actual paths. Figure 74 illustrates the position estimations of path 1 for all values of ϕ, as well as the MSE of the subsequent estimations.



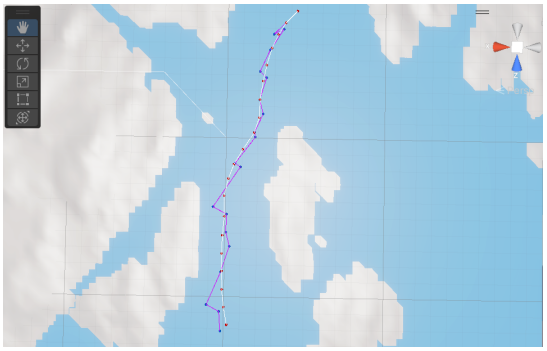
(a) Path 1 with ϕ= 10. Existing error (MSE): = 2.89.



(b) Path 1 with ϕ= 70. Existing error (MSE): = 31.41.

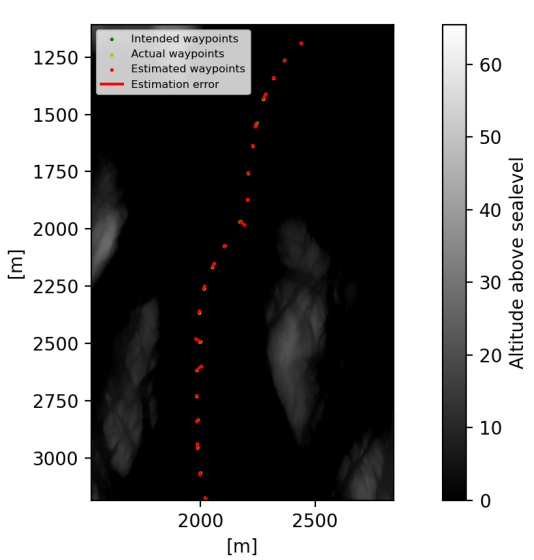


(c) Path 1 with ϕ= 120. Existing error (MSE): = 49.39.

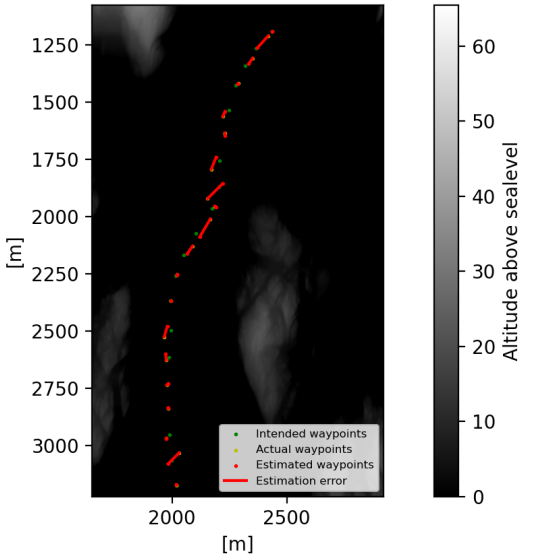


(d) Path 1 with ϕ= 170. Existing error (MSE): = 51.43.

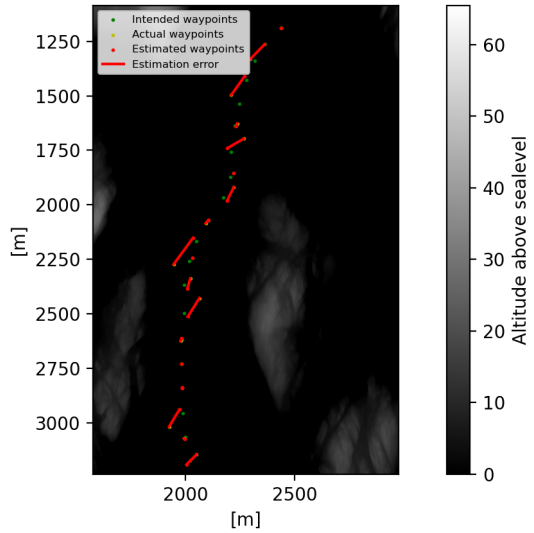
Figure 73: Path 1 with varying degrees of noise and the corresponding MSE between the intended and actual positions



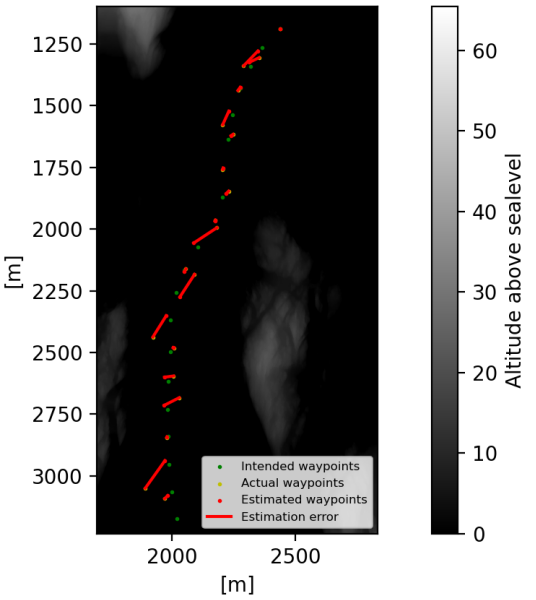
(a) Position estimation results for path 1 with ϕ= 10. Estimation MSE = 10.1



(b) Position estimation results for path 1 with ϕ= 70. Estimated MSE = 31.08.



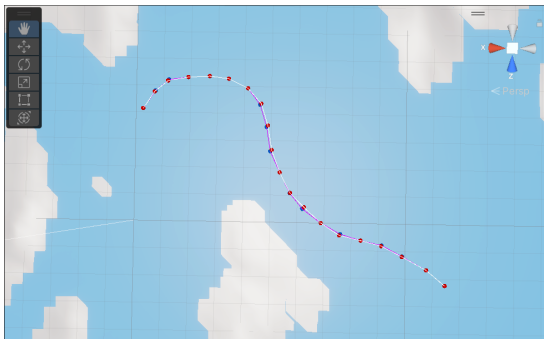
(c) Position estimation results for path 1 with ϕ= 120. Estimated MSE = 50.8.



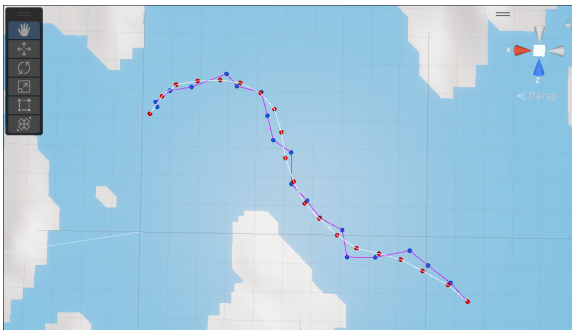
(d) Position estimation results for path 1 with ϕ= 170. Estimated MSE = 46.57

Figure 74: Position estimates for path 1 with varying degrees of noise and the corresponding MSE between estimated and actual values.

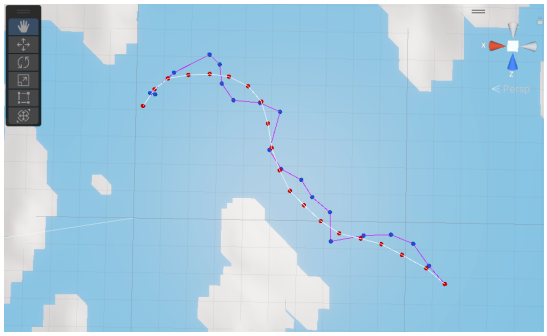
Figures 75 and 76 illustrate similar results for path 2 over all values of ϕ:



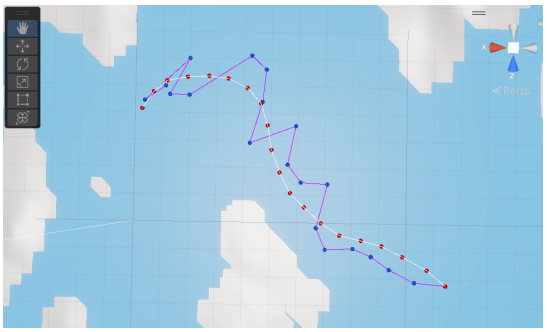
(a) Path 2 with ϕ= 10. Existing error (MSE): = 2.61.



(b) Path 2 with ϕ= 70. Existing error (MSE): = 25.08.

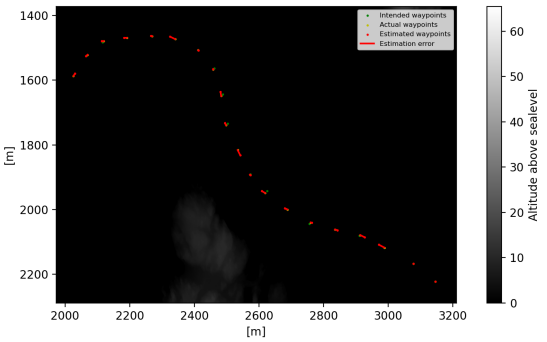


(c) Path 2 with ϕ= 120. Existing error (MSE): = 42.05.

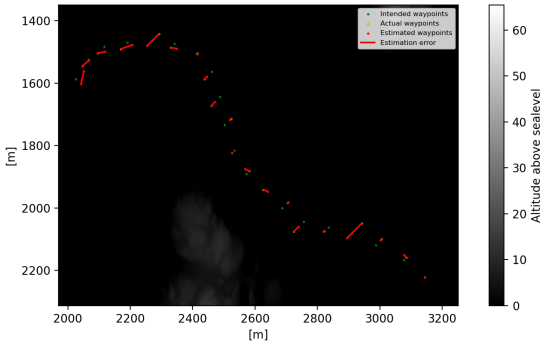


(d) Path 2 with ϕ= 170. Existing error (MSE): = 77.19.

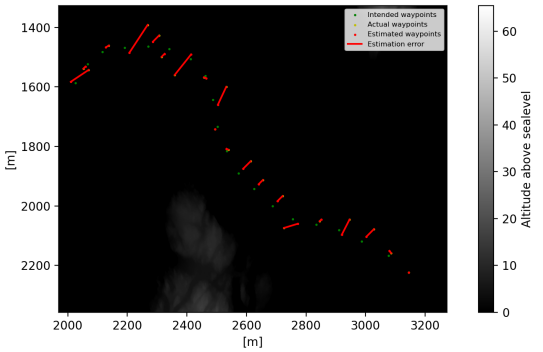
Figure 75: Path 2 with varying degrees of noise and the corresponding MSE between the intended and actual positions.



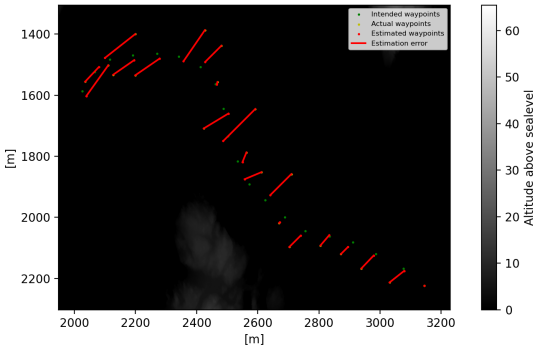
(a) Position estimation results for path 2 with ϕ= 10. Estimation MSE = 8.54



(b) Position estimation results for path 2 with ϕ= 70. Estimated MSE = 20.23.



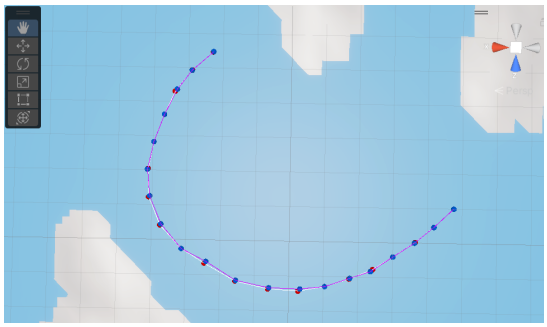
(c) Position estimation results for path 2 with ϕ= 120. Estimated MSE = 34.38.



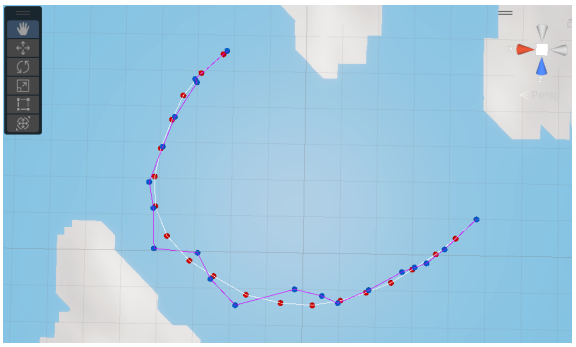
(d) Position estimation results for path 2 with ϕ= 170. Estimated MSE = 73.18

Figure 76: Position estimates for path 2 with varying degrees of noise and the corresponding MSE between estimated and actual values.

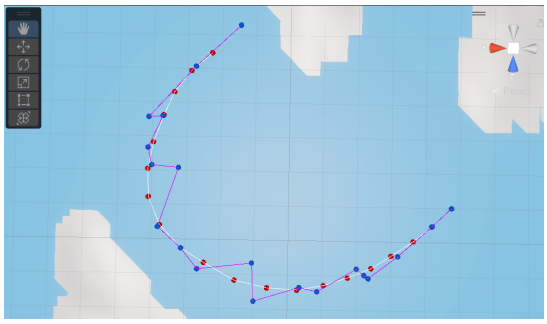
Finally, Figures 77 and 78 illustrate the results for path 3 over all values of ϕ:



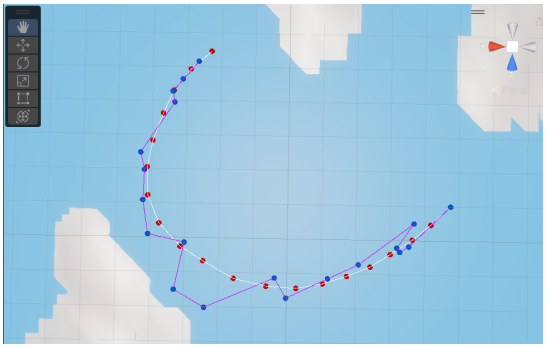
(a) Path 3 with ϕ= 10. Existing error (MSE): = 3.13.



(b) Path 3 with ϕ= 70. Existing error (MSE): = 26.80.

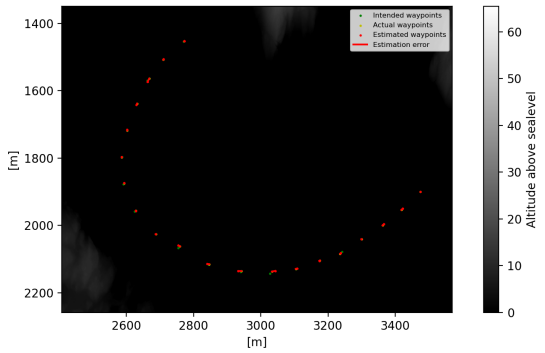


(c) Path 3 with ϕ= 120. Existing error (MSE): = 43.32.

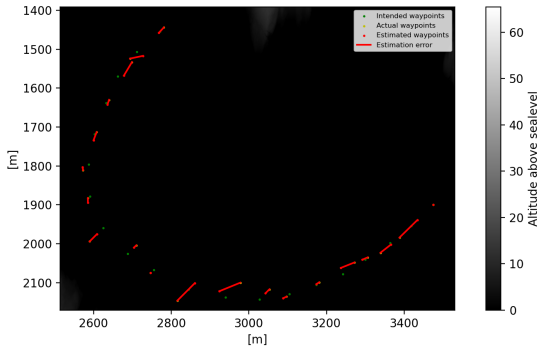


(d) Path 3 with ϕ= 170. Existing error (MSE): = 56.57.

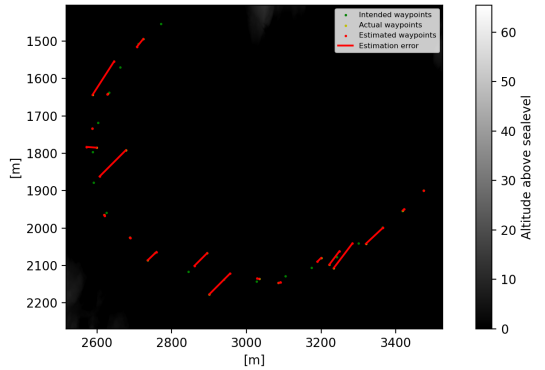
Figure 77: Path 3 with varying degrees of noise and the corresponding MSE between the intended and actual positions.



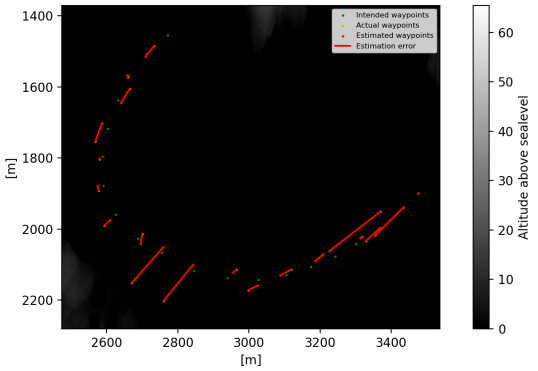
(a) Position estimation results for path 3 with ϕ= 10. Estimation MSE = 4.28



(b) Position estimation results for path 3 with ϕ= 70. Estimated MSE = 25.75.



(c) Position estimation results for path 3 with ϕ= 120. Estimated MSE = 35.73.



(d) Position estimation results for path 3 with ϕ= 170. Estimated MSE = 52.08

Figure 78: Position estimates for path 3 with varying degrees of noise and the corresponding MSE between the estimated and actual values.

The differences between the existing errors and estimation MSEs can then be compared to give a quantitative measure of the estimators performance over just assuming no positional change from the intended trajectory. Figure 79 illustrate this in a simple diagram, presenting the improvements of MSE measurements in % for all 3 paths and with all 5 values of ϕ. Positive values indicate an improvement over the existing error, i.e, the estimator performs better than simply estimating $\hat{p}_i = \vec{\omega}_i$. Figure 80 further elaborates on this in a new diagram showing the improvements of MSE with respect to each of the three paths. Note that ϕ = 10m is omitted to provide a generalized analysis to the 4 similar cases.

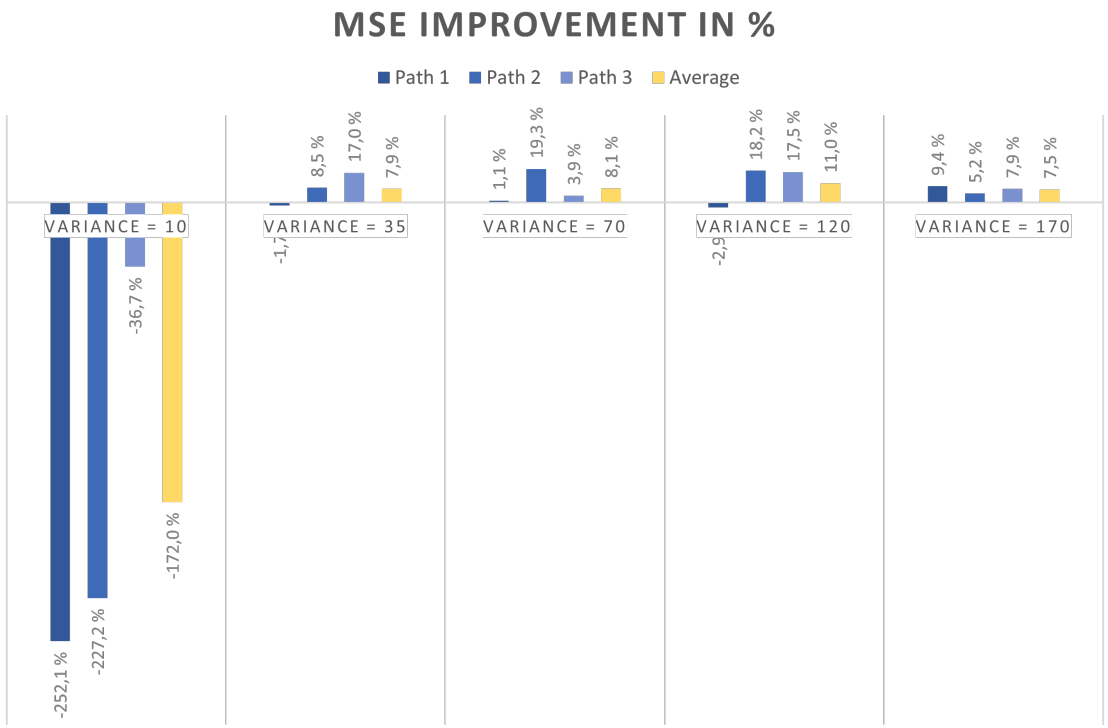


Figure 79: Diagram showing the % improvement of using the estimator, rather than the intended path, to make estimations. Noteably, the framework performs poorly on paths with little to no disturbances, but sees great improvements for noisier paths.

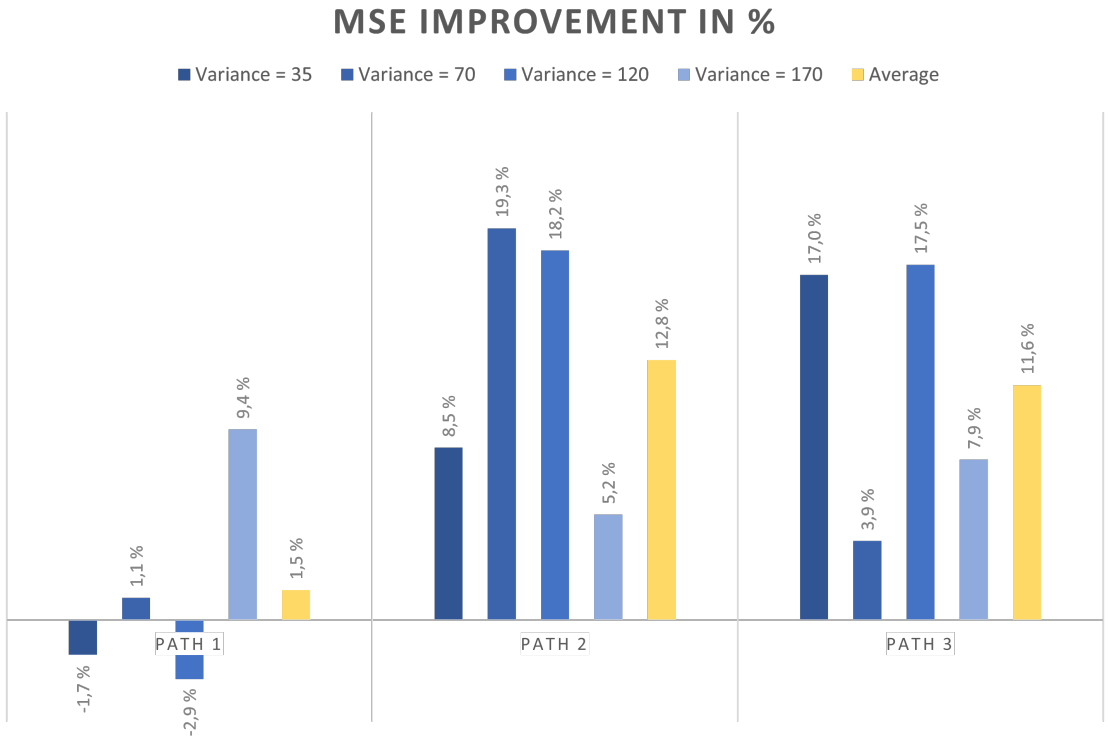


Figure 80: Diagram showing the % improvement of using the estimator, focusing on the improvements notices over the separate paths. $\phi = 10$m is omitted to give a better generalization of the results.

6 Discussion and future work

This section will seek to provide an in depth analysis of the results in Section 5 whilst providing suggestions for changes and improvements in future implementations.

Initially, the overall performance of the framework is analysed in depth, before the validity of both the simulator data and system architecture are discussed. Finally, the results of all the modules for feature extraction and matching are analyzed in more detail, with a strong focus on possible changes and improvements for future implementations.

6.1 Analysis of the complete estimation scheme and nonlinear problem

Upon inspection of the results in Section 5, it becomes clear from Figures 79 and 80 that the estimation framework generally performs quite well, albeit presenting room for a variety of improvements. Disregarding the absolutely abysmal results on the path with ϕ= 10m, the estimator seems to improve on the the so-called existing error by an average of $\sim 8.5\%$. For an initial proof of concept, these results should not be seen as too discouraging, given that the system performs fairly consistently over paths with adequate to high amounts of deviations. It is evidently, as shown in Figure 80, far superior on some paths rather than others, going from a 1.5% average improvement to a $12,8\%$ improvement and can produce results that are occasionally worse than the existing error. This can have a variety of different explanations and should be analyzed more rigorously, but the authors leading theory has to do with the average distance from to the nearest visible shoreline along each path. Path 2 and 3, on which the estimator far outperforms its results on path 1, are both situated significantly closer to the extracted features on average, something Figures 54 and 55a illustrate nicely. The possible errors of estimation over longer distances are then to be considered significantly more devastating to the system, as a pixel in the image space of the camera aboard the vessel will contain a lot more information, generalizing far more of the surrounding terrain, for features further away. This is, however, a hypothesis, and the focus should rather be turned to the visual and numerical results:

The elephant in the room should be addressed first: The paths of noise variance ϕ= 10m. Judging by the results in Figure 79, the estimator performs abhorrently on a path with little to no deviation, supporting an average deterioration of the MSE by almost 200&. This might be a little hard to believe when inspecting the complete estimated paths in Figures 74a, 76a and 78a, boasting the evidently smallest estimation errors for all 3 paths, but has a very simple explanation tied to an average error performed by the estimator. Quickly inspecting the estimation iterations on paths with $\phi \neq 10$, for example Figures 66 or 67 with $\phi = 35$, it becomes evident that the chosen adaptive grid search does not manage to correctly converge to the target solution $\vec{p}_i$, always introducing some deviation, if only of a few pixels. When no noise is present in the original path however, such deviations, although small as evident in Figures 74a, 76a and 78a, represent a massive increase over the existing noise, drastically inflating the underperformance of the system. One could argue that the choice of metric in Figures 79 and 80 exaggerates these errors drastically, but it is still an important illustration of the relative deterioration performance for such paths and should be used to conclude that the estimation framework has a negative effect on paths where little to no noise is expected.

It becomes important however to shift the focus from the special case where paths have little to no noise to the more general cases, where average to high levels of noise are present and the estimator performs better than the existing error on average. These are not only more realistic in a maritime environment, but the results here are significantly more promising, showcasing the future potential of the framework as a standalone or fused pose estimator. Figures 62, 63 and 64 compared with Figures 65a-71 illustrate, however, an assumption on which this analysis should take place. The vastly different performance of the framework when the matching between image features returns a correct or wrong match is staggering, and so the two cases should be analyzed separately so as to form as good an image as possible of the frameworks strengths and weaknesses along more plausible paths.

When the matching of image features return a correct set of matching areas $\vec{f}_{\vec{\omega}_i}$ and $\vec{f}_{\vec{p}_i}$, the results of the estimation framework are generally quite good, as visualized by Figures 65a and 65b. Upon comparison with Figures 69a and 69b, where the matches are wrong, and the numerical results of estimation errors for all waypoints in Figures 62, 63 and 64, one can safely conclude that correct matches produce far smaller errors than wrong matches. These remaining errors do, however, warrant discussion, elaborating to what degree they are the culmination of propagating errors from feature extraction and matching or the fault of the proposed estimation scheme. Figures 66 and 68 can be interpreted as advocating the former, showing how even when the iteratively updated position estimate results in reprojections approaching the target image feature $\vec{f}_{\vec{p}_i}$, the final estimation $\hat{p}_i$ deviates from $\vec{p}_i$. The shape of the reprojection $\vec{f}_{DEM_i}$ deviates, if only by a few pixels, from $\vec{f}_{\vec{p}_i}$, introducing an error in Figure 66 and even divergence in the wrong direction in Figure 68. The possible causes for these deviations are discussed in Section 6.7, but to summarize quickly: The errors most likely occur during data-generation, as slight differences between the DEM and constructed 3D mesh or discretization happening upon the capture of semantic segmentations. At the same time however, both figures, especially 66, also illustrate the shortcomings of the estimation-scheme, falling into a local minima deviating from the optimal solution at $\vec{p}_i$. This can occur from the lack of exploration in an adaptive grid search, where the first solution found is often regarded as the best solution, and warrants the exploration of other nonlinear solvers. Finally however, one should note Figure 67, which illustrates an example of a successful estimation. Although subject to a good match between the target feature $\vec{f}_{\vec{p}_i}$ and the reprojection of $\vec{f}_{DEM_i}$, as well as a probable local minima in the desired solution $\vec{p}_i$, such cases illustrate that the reprojection error is capable of rendering a proper optimization problem for the desired pose estimation. The perquisites for this to occur are just extremely sensitive to errors and would probably benefit from a more robust nonlinear solver.

Figures 69a and 69b, however, show how the optimization problem is rendered infeasible when wrong matches occur between image features. The selected examples of the estimation iterations in Figures 70 and 71 illustrate how the position of the vessel is driven drastically off course to try and compensate for the discrepancies between $\vec{f}_{\vec{p}_i}$ and the reprojection of a wrongly matched $\vec{f}_{DEM_i}$, producing a set of invalid estimations with staggering errors. The MSE diagrams in Figures 62, 63 and 64 illustrate this the most efficiently: Waypoints $14-16$ of path 1 and waypoints $1-2$ in path 2, illustrated in the Figures 70 and 71 respectively, result in sets of error-values eclipsing the MSEs along their respective paths, contributing to a significant deterioration of the overall results. For future implementations, the matching algorithm then becomes imperative to produce more consistent estimations and requires extra attention.

Overall, the estimation procedure can be significantly improved by remedying propagating errors from the feature extraction and matching as well as changing the nonlinear solver from the baseline adaptive grid search to a solution that is more robust in the face of local minima. A thorough discussion of the deployed modules dealing with features follow in the subsections below, but a quick discussion of possible optimization strategies should be mentioned here. As the presented grid-search solution is meant to function as a baseline for future work, a more sophisticated method, like the popular Levenberg–Marquardt algorithm applied to a nonlinear least squares formulation of the reprojection error, should be investigated to asses possible improvements of both the estimation errors and performance. At the same time, a thorough analysis of the multi-dimensional optimization space should be conducted, to validate whether or not a global optimization strategy would be to prefer if there exist a plethora of local minima around every desired solution. To this end, a set of global estimation schemes, such as the ever more popular Evolutionary algorithms, should be implemented an analyzed on a more sophisticated implementation.

6.2 Simulator validity and viability of synthetic data

Given the nature of VO applications, the data supplied by the monocular camera(s) are vital to the accuracy and performance of the final estimator. To this end, the use and necessity of the Unity game engine in generation of critical, synthetic data is brought under the lens, as well as the viability of the data presented in this report to a plausible, real world application.

The use of the Unity game engine can, at first glance, be considered fairly unnecessary. The extraction of the intended skyline contour $S_{\vec{\omega}_i}$ at waypoint i can be done entirely outside of Unity by applying Algorithm 4 to the entire camera frustum at a known position $\vec{\omega}_i$, whilst semantic segmentations of $I_{\vec{p}_i}$ are available through pre-trained deep neural networks, such as the aforementioned YOLOv8 algorithm [37], on the output of the on-board monocular camera. Furthermore, the presented data is both incomplete and unrealistic, utilizing machine-calculated semantic segmentations that would far exceed the precision of neural networks, while not taking into account external factors that should occur at sea. Although reasonable concerns, these should not discredit the value of deploying such a tool for either the presented application or future implementations.

For such a proof of concept, the use of perfect, semantic segmentations in an environment without the interference of waves, fog, illumination and weather-effects can be seen as a way to form a baseline understanding of the performance one can expect under optimal conditions. More than anything, one can imagine a decent performance under such conditions to present a minimum requirement of the system. One that, if exceeded, should entice adaptations and improvements to tackle more challenging scenarios with limiting factors. It is in such a development stage the worth of a fully customizeable simulator within a videogame engine becomes evident. Whereas the data produced for a proof of concept only suffices to produce a minimal reproduction of the real world, a full implementation can produce a plethora of realistic scenarios at any given time, complete with everything from realistic weather to believable interactions with the environment, which can be captured through realistic, replicable camera imagery without ever deploying a camera in the real world. Development and testing of more mature implementations can then be done entirely within the confines of a simulator, generating vast amounts of data for the training and validation of neural networks, enabling the development and later deployment of an improved system into the real world.

6.3 System architecture and design decisions

The overall system architecture, presented by the pipeline in Figure 24, should be taken critically into consideration. Although there is little visual evidence to suggest that the division into independent data generation and analysis has introduced any significant error to the system, one should not disregard the possibility of minor deviations occurring between the two. Notably, the results of the reprojections in Figures 57-61 could suggest that minor differences are being introduced upon generating the synthetic data within Unity. These small, but potentially deteriorating errors, could, however, also stem from a propagation of minor errors in previous modules. What can be discussed more explicitly, is the possible pros and cons of an alternative real-time estimation scheme and if this should be considered for future implementations. The greatest allure to a real-time system, for example in Unity, would be the possibility to use the exact same terrain data, without having to worry about conversions and identicalness, for both synthetic data generation and estimation. This would ensure that the estimation scheme is 1:1 by nature, independent of the deployed DEM, possibly yielding a more robust piece of software. It would also more closely resemble a software for active deployment onboard a physical vessel, more closely replicating the desired, final implementation. It should be noted, however, that the implementation of a real-time system would be significantly more cumbersome, and was hence avoided in order to prioritize the development of the presented modules and algorithms. To the authors knowledge, clever use of multi-threading and emulation of the software on-board a maritime vessel would be required to develop a satisfactory simulation, making a real-time implementation somewhat exuberant for such an initial proof of concept.

Within the chosen architecture, it should be mentioned that the generation of 3-dimensional meshes for the use in synthetic data generation could be done in more than one way. The method deployed in Section 4.1, manually converting external DEMs through a multi-step procedure, yields satisfactory results when compared to the raw data. One should note, however, that such a method could be subject to minor propagation of error if one or more of the conversion-steps produce the slightest amount of subsampling, smoothing of sharpening. A third-party plugin, ArcGIS, for the Unity game engine presents an alternative method for terrain replication. Deployed in [39], the method allows for faster, simpler generation of accurate terrain data anywhere on the planet, only

utilizing a set of global coordinates and a defined radius to infer terrain data directly from the cloud. Utilizing Unity's high precision framework to supply geospatial positioning unaffected by floating point rounding errors, the plugin is further optimized for the transformations of objects inside the world. Although ArcGIS proves a more than capable tool, the many features introduced a significant overhead that complicated certain parts of the proposed framework: Terrain generated by ArcGIS proves tough to manipulate and customize, positional data in a local coordinate frame prove a little tricky to extract due to the high precision framework and, the possibly the biggest deterrent, it proves fairly difficult to use custom terrain data inside the plugin. Although ArcGIS comes with default global terrain data, the precision required by the system, and by extension the required identicalness between terrain data, cannot be guaranteed without generating and analyzing data based on the same source.

6.4 Extraction of skyline contours

Section 5.2.1 illustrate the great performance of the skyline contour extraction algorithm proposed in Section 3.1.1. The results are consistent over all waypoints and the mean elapsed time of 0.16s per skyline contour from Figure 38 is highly performant. As there is little noise in the simulated images, it is tough to evaluate the performance of the integrated noise-removal, other than on the edges of segmented terrain, that are sometimes interpreted as minor disturbances. These are trimmed away, resulting in skyline contours not stretching out fully to the edge of the segmentations of terrain. This trimming of the visible terrain, which at first glance might seem undesirable, is not necessarily a bad thing. Quite on the contrary, removing ambiguous parts of the terrain could be considered a desirable trait, eliminating a potential source of errors in future matching of image features. In future implementations, rigorous testing should be performed on more complex sources of noise, especially those resulting from external factors. Smarter methods would most likely have to be adopted to extract skylines out of semantic segmentations heavily polluted by noise, the most logical of which would be some sort of deep neural network able to approximate skyline contours through dense vegetation, partial cloud cover and heavy precipitation. The presented method could then serve as a baseline for an algorithm designed to generate synthetic training data for such an algorithm, which would be useful for a possible real-world application as well.

A final thing to note about the presented algorithm is the possibility to keep or remove the coastline in the contours, as shown in Figure 12. Although this part of the contour proved uninteresting for this proof of concept, where 3-dimensional rotation of the boat was neglected, it could prove a valuable feature in more sophisticated implementations. If, for example, the system was to be extended to 6 DOF, the coastline could serve as a semi-horizontal line on which to align the segmented terrain for further analysis.

6.5 Hilltop extraction from skyline contours

The hilltop extraction algorithm performed well over all waypoints, returning $\mathcal{K}$ hilltop features meeting the strict requirements for each skyline contour defined in Section 3.1.2. These conditions are in many ways what makes the algorithm fairly unique, returning $\mathcal{K}$ robust features that, given good tuning of algorithm parameters, are guaranteed to be somewhat distinctive within the image. The algorithm has one potentially disruptive problem however: The required tuning of width and height-thresholds. Given the wide variety of possible skyline contours, the algorithm may yield exuberant amounts of hilltops in one contour and few to none in another if the same parameter values are applied in both cases. The algorithm does not adapt to the diverse terrain it might be facing and would then be unreliable as an out-of-the-box solution. Possible improvements then naturally include an adaptive version of this algorithm, perhaps using the vertical deviation from coastline to skyline and the amount of visible gaps in a contour to tune the parameters automatically, or a machine-learning driven approach, where the current solution might serve to produce vast amounts of training data under human supervision.

Lastly, the performance of the algorithm, although not terrible, could see significant improvements. The depth-first search methodology is sufficient for a proof of concept, estimating the pose post

data acquisition, but a mean elapsed time of 1.45s in Figure 43 renders the algorithm infeasible for a potential real-time implementation in the future.

6.6 Matching of $\mathcal{M}-1$ image features

As mentioned briefly above and judging by the results in Section 5.2.3, the matching and extraction of image features clearly has the greatest potential to cause critical failure during estimation. Although the occasional error occurring when image features are correctly matched between skyline contours primarily occur occur in other modules (see Section 6.1), the significant error posed by wrong or incorrect matches are solely the responsibility of the deployed matching algorithm. A very prevalent example of the negative contribution of such matches can be seen in Figure 69a, where the three consecutive, erroneous estimations, based on errorneous matches, dramatically increase the overall MSE in Figure 62b, resulting in the negative contribution of the estimator for ϕ= 35 in Figures 79 and 80. Another example can be seen in Figure 69b, where an incorrect match is produced at waypoint 1, visible in Figure 50, skyrocketing the estimation error manyfold above the MSE for path 2 in Figure 63b. Upon investigation of the results in Figures 62b, 63b and 64b it becomes evident that incorrect matches are the predominant cause of quickly diverging estimation errors, prompting the discussion of how more sophisticated algorithms could seek to improve on the one presented in this framework.

To this end, it becomes imperative to understand the possible scenarios in which erroneous matches may occur. Thankfully, inspection of the results in Section 5.2.3 yield some valuable insight:

- **Best match is not the correct one:** Figure 51 illustrates a difficult problem when dealing with energy based similarity measures: The SAD measure returns a different match to the correct area, simply due to it having a better overall score than the correct match. The transformation between the two images is just so that it invalidates the correct match by shifting the reprojection of the terrain, implicitly creating a better match through the deployed perspective transformation.
- **Image feature goes out of frame:** Figure 50 illustrate an example of an erroneous matching in part caused by the haunting limitations of monocular cameras with limited FOV. Between the two images, a sufficient transform finds place to eliminate a plausible image feature from $S_{\vec{\omega}_i}$ in $S_{\vec{p}_i}$ by moving it out of frame. However, another match, superior to other potential features, is found in the actual skyline and returned, resulting in a misclassification.
- **Isolated features have different sizes:** Figure 52 illustrate a case which, although seemingly extremely specific at first glance, produced the 4 wrong matches denoted for path 1 in Table 3. On the intended skyline $S_{\vec{\omega}_i}$, an isolated feature meeting all conditions is found. In $S_{\vec{p}_i}$, this same feature is located in the middle of the image, but is not returned as a match. This happens for a very simple reason: The transform between the two images render the corresponding area in $S_{\vec{p}_i}$ slightly smaller, making it an infeasible match, as per the definition described in Figure 14. The inverse of this is of course possible, with Figure 44a illustrating a similar image feature increasing in size between the two images, rendering it feasible.

An improvement to the presented algorithm would then have to take these cases into account to deliver a more reliable, consistent results. For image features getting out of frame, the author suggests that future implementations directly remedy the limiting FOV by using wide-angles images or even panoramic imagery, like in [22]. This would be both quicker to implement and yield more reliable results than developing a new algorithm for a camera of limited FOV, also covering cases where no features are visible within a slim camera frustum. In the case of isolated features having different sizes between one or more camera views, an option could be to develop a more adaptive algorithm, changing the requested width and height of matches through, for example, a rough estimation of rotation and depth between the images. Another option, which the author thinks holds more merit, is to consider replacing the energy based matching algorithm with a deep neural network, trained on a mix of data produced by the existing algorithm and, in the case of the respective errors, by human generation. Such a method would alleviate the complexity introduced

to the method by the declaration and tuning of adaptive parameters, whilst also handling the third error case, which occurs due to the deployment of an energy based similarity measure. When adopting SAD to find the most similar areas between two skyline contours, there is no guarantee that the returned area is necessarily the correct one. One can say that such methods are *blind*, not taking into account the neighborhood past the considered features, making them incapable of any form of spatial reasoning. A neural network would, in comparison, take the entire skyline into consideration, inferring from the experience of thousands of previous matches where the optimal sets of image features $\vec{f}_{\vec{\omega}_i}$ and $\vec{f}_{\vec{p}_i}$ could be located. It could potentially solve a plethora of problems introduced in the current algorithm, although requiring paramount amounts of training data at the same time.

One should also address the suboptimal and ambiguous matches denoted in Table 3, the explanations for which are fairly simple. The term ambiguous matches in Section 5.2.3 was deployed to explain a set of matches not easily verifiable through visual inspection. That is, they can both be wrong or correct, the author is just not capable of making an informed decision on the manner. The suboptimal matches are based on a problem with the adopted implementation, specifically with the SAD algorithm. Inspecting Figures 46 and 47, one can quickly conclude that all suboptimal matches seem to return the shortest hilltop features located by Algorithm 2 on $\mathrm{S}_{\vec{\omega}_i}$, although more prominent matches are clearly available. This is easily explained by equation (2), which does not, compared to for example an MSE measure, utilize a mean error between all points to define the similarity between two areas. SAD does not take the length of features into account, simply adding together the error of all points before returning this as the error of the evaluated matching. Longer features will then experience higher overall errors than existing shorter features in almost all cases, leading to the selection of suboptimal solutions. This is a major flaw in the implemented framework, but is easily remedied by introducing a different similarity measure, employing the mean of all combined errors. In the case of replacing the energy based similarity measure with a neural network all together, as mentioned above, the issue of both ambiguous and suboptimal matches would most likely be rendered redundant.

It is important to note, however, that the feature matching procedure is not completely without merit. Table 3 illustrate an overall mean accuracy of just about 79%, with Figures 44a, 45a and 55b showing a set of correct matches found between widely different, often close to uniform, skyline contours. The main contribution of errors to the pose-estimation scheme come from the significantly more infrequent errors, which are challenging to avoid when deploying a monocular camera with limited FOV. The errors produced when correct matches have been found between skylines have to be attributed to later modules.

6.7 The extraction and reprojection of the DEM feature $\vec{f}_{DEM_i}$

To the authors knowledge, the only way to numerically assess the quality of the outputs of Algorithm 4 is to analyze the reprojection of the resulting feature, $\vec{f}_{DEM_i}$, back to the reconstructed camera. If little to no errors are recorded between the original image feature and the reprojection of $\vec{f}_{DEM_i}$, one can safely assess that a plethora of modules and equations function as intended, whilst the opposite would indicate an error at any of these arbitrary locations. With the proposed interface, it is reasonable to assume that a good reprojection would validate the reconstruction of the Unity camera in equation (15), the reprojection of 3D parameters in equations (**??**) and (18), and the line rendering strategy to construct $\vec{f}_{DEM_i}$ from $\vec{f}_{\vec{\omega}_i}$ in Algorithm 4, as all are vital to the reprojection from equation (9).

The results highlighted in Figures 60a and 61a can then be seen as validations of the presented modules, illustrating the reprojections of the DEM features found at the respective waypoints and the deviation of this from the intended image feature $\vec{f}_{\vec{\omega}_i}$. The errors recorded are only in the range of a few pixels and always occur along the vertical axis of the skylines, indicating a slight difference in height between the terrain recorded in $\mathrm{I}_{\vec{\omega}_i}$ and the DEM. These slight errors can have a plethora of sources, like post-processing in Unity, discretization of images with limiting resolutions and possible smoothing on the 3-dimensional terrain mesh during conversion, but are evidently, by the sheer complexity of the compound problem and proximity of results, not caused by the reprojection framework. A few pixels difference are, however, sufficient to derail the estimation

significantly given the sensitivity of the features and corresponding error, so a thorough analysis should be performed on future implementations, especially of potential floating point errors of Algorith 4.

The results of the complete skyline reprojections in Figures 57, 58 and 59, where $\vec{f}_{DEM_i}$ is found by forming and evaluating $\mathcal{N}_w$ rays spanning the entirety of $\mathrm{S}_{\vec{\omega}_i}$, further illustrate the success of Algorithm 5 in subdividing the camera frustum into $\mathcal{L}$ angular deviations, as no skylines break the horizontal bounds of the image plane. The general consistency of the algorithm also works to credit the supplied line-drawing algorithm and Algorithm 6, used to approximate the intersection of a ray $\vec{R}_j$ with the edge of the DEM along the heading of the ray.

Finally, one can comment on the performance of Algorithm 4 by inspection of the results in Figure 56. Through comparison with path 3, which is where the diagram originates from, one should note how the fourth waypoint singlehandedly contributes to the mean elapsed time of the algorithm. The intended feature $\vec{f}_{\vec{\omega}_i}$ at this waypoint, as well as the extracted match $\vec{f}_{DEM_i}$ can be seen in Figure 55 and is the largest feature to be matched along all considered waypoints. Its significant computation time, as well as the hopeful eradication of suboptimal, short matches in future iterations, discussed in Section 6.6, implies that the presented extraction scheme will not be viable for real-time deployment. A plausible workaround for this, however, could be to generate $\mathcal{K}$ possible features $\vec{f}_{DEM_i}$ beforehand in simulator, only using images captured along the vessels intended path and uploading them to the vessel before deployment. This would allow the system to work in real-time, but would not capable of covering edge cases where the update illustrated in Figure 9 alters the intended course significantly. To ensure the consistency of such a solution, the monocular camera of limited FOV should, again, be replaced by a 360° panoramic camera as discussed in Section 6.6, capable of catching all possible features at a given location without fail.

7 Concluding remarks

This report presented a baseline framework for the 2DOF pose estimation of a maritime vessel deviating from a predefined, intended path. The vessel was only equipped with a limited, monocular camera and terrain data of its surroundings, relying on the comparison of synthetically generated images and the on-board camera to infer the vessel's position using the reprojection error. In this proof of concept, all data was generated inside of a Unity based simulation, resulting in machine-computed semantic segmentations of terrain, most likely more precise than more realistic segmentation schemes. Analysis and image processing was finally performed post data-generation in an independent Python script, implementing the bulk of the algorithmic framework.

The system was further tested on synthetic data. Although not ready for commercial deployment, either as a standalone estimator or as part of a greater sensor fusion, the system performed well for an initial proof of concept. The modules tasked with feature extraction in 2 and 3 dimensions yielded consistently good results on all waypoints, and the mathematical formulations used to reproject 3D world points in Python to a simulated Unity camera worked flawlessly. This, in turn, led to an implied well-posed optimization problem when image features were correctly matched, something that was not always the case. The algorithm tasked with matching the $\mathcal{M}-1$ features recorded wrong, failed or ambiguous matches at just about 21% of all waypoints, contributing drastically to the near constant error introduced by the baseline nonlinear solver. Overall, however, the system consistently yielded positive contributions compared to the existing mean squared error between intended and actual paths.

The general system performance, combined with the consistently good outputs of most modules and the well-posed optimization problem, provides evidence to conclude that the proposed framework presents significant potential as a future long-range pose estimator. From what was discussed in Section 6, integrating a more rigorous algorithm for the matching of image features and exchanging the proposed nonlinear optimizer for a more reliable, possibly even global, solution, could yield significant improvements. Coupled with the almost seamless integration of wide angle or even panoramic cameras into the modular framework, eliminating the many limitations of a smaller field of view, one could be looking at a consistent, accurate solution to a complex navigation problem in the future.

Bibliography

[1] *Marine*. United States government. URL: https://www.gps.gov/applications/marine/ (visited on 11th May 2023).

[2] European Commision. *The EUBlue Economy Report 2022*. 2022nd ed. European Union, 2022. ISBN: 978-92-76-52444-1. DOI: 10.2771/793264. URL: https://oceans-and-fisheries.ec.europa.eu/system/files/2022-05/2022-blue-economy-report_en.pdf.

[3] Raquel A. F Neves et al. «Socio-economic impacts of a maritime industrial development area (MIDA) model in Latin America: the case of the Açu Port-Industrial Complex». In: *WMU Journal of Maritime Affairs* 21.3 (2022). ISSN: 1654-1642. DOI: 10.1007/s13437-021-00261-z. URL: https://doi.org/10.1007/s13437-021-00261-z.

[4] G.S. Dwarakish and Akhil Muhammad Salim. «Review on the Role of Ports in the Development of a Nation». In: *Aquatic Procedia* 4 (2015). INTERNATIONAL CONFERENCE ON WATER RESOURCES, COASTAL AND OCEAN ENGINEERING (ICWRCOE'15), pp. 295–301. ISSN: 2214-241X. DOI: https://doi.org/10.1016/j.aqpro.2015.02.040. URL: https://www.sciencedirect.com/science/article/pii/S2214241X15000413.

[5] United Nations. *Review of Maritime Transport 2022*. 2022nd ed. United Nations Publications, New York, 2022, pp. 3–6. ISBN: 978-92-1-002147-0. URL: https://unctad.org/system/files/official-document/rmt2022_en.pdf.

[6] Eurostat. *Maritime passenger statistics*. Statistical Government of the European Union, 2022. URL: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Maritime_passenger_statistics&oldid=583930#Number_of_seaborne_passengers_only_partially_recovered_in_2021.

[7] Benoit Figuet et al. «GNSS Jamming and Its Effect on Air Traffic in Eastern Europe». In: *Engineering Proceedings* 28.1 (2022). ISSN: 2673-4591. DOI: 10.3390/engproc2022028012. URL: https://www.mdpi.com/2673-4591/28/1/12.

[8] European Union Aviation Safety Agency. *Safety Information Bulletin: Global Navigation Satellite System Outage Leading to Navigation / Surveillance Degradation*. 2022.

[9] Manuel Cuntz et al. «Jamming and Spoofing in GPS/GNSS Based Applications and Services – Threats and Countermeasures». In: *Future Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 196–199. ISBN: 978-3-642-33161-9.

[10] A.C. O'Connor et al. «Economic Benefits of the Global Positioning System (GPS).» In: (2019), pp. 9–31. URL: https://www.rti.org/sites/default/files/gps_finalreport.pdf.

[11] Tegg Westbrook. «The Global Positioning System and Military Jamming: The geographies of electronic warfare». In: *Journal of Strategic Security* 12 (2019). DOI: https://doi.org/10.5038/1944-0472.12.2.1720.

[12] Jingyi Liu et al. «Real-Time Monocular Obstacle Detection Based on Horizon Line and Saliency Estimation for Unmanned Surface Vehicles». In: *Mobile Networks and Applications* 26.3 (2021). ISSN: 1572-8153. DOI: 10.1007/s11036-021-01752-2. URL: https://doi.org/10.1007/s11036-021-01752-2.

[13] Zhengjun Qiu et al. «Vision-Based Moving Obstacle Detection and Tracking in Paddy Field Using Improved Yolov3 and Deep SORT». In: *Sensors* 20.15 (2020). ISSN: 1424-8220. DOI: 10.3390/s20154082. URL: https://www.mdpi.com/1424-8220/20/15/4082.

[14] Samira Badrloo et al. «Image-Based Obstacle Detection Methods for the Safe Navigation of Unmanned Vehicles: A Review». In: *Remote Sensing* 14.15 (2022). ISSN: 2072-4292. DOI: 10.3390/rs14153824. URL: https://www.mdpi.com/2072-4292/14/15/3824.

[15] Milad Ramezani and Kourosh Khoshelham. «Vehicle Positioning in GNSS-Deprived Urban Areas by Stereo Visual-Inertial Odometry». In: *IEEE Transactions on Intelligent Vehicles* 3.2 (2018), pp. 208–217. DOI: 10.1109/TIV.2018.2804168.

[16] Nan Yang et al. «D3VO: Deep Depth, Deep Pose and Deep Uncertainty for Monocular Visual Odometry». In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 1278–1289. DOI: 10.1109/CVPR42600.2020.00136.

[17] Johannes Graeter, Alexander Wilczynski and Martin Lauer. *LIMO: Lidar-Monocular Visual Odometry*. 2018. arXiv: 1807.07524 [cs.RO].

[18] Lucas R. Agostinho et al. «A Practical Survey on Visual Odometry for Autonomous Driving in Challenging Scenarios and Conditions». In: *IEEE Access* 10 (2022), pp. 72182–72205. DOI: 10.1109/ACCESS.2022.3188990.

[19] Fabio Cozman, Eric Krotkov and Carlos Guestrin. «Outdoor Visual Position Estimation for Planetary Rovers». In: *Autonomous Robots* 9 (Sept. 2000), pp. 135–150. DOI: 10.1023/A:1008966317408.

[20] Patrick Carle, Paul Furgale and Timothy Barfoot. «Long-range rover localization by matching LIDAR scans to orbital elevation maps». In: *Journal of Field Robotics* 27 (May 2010), pp. 344–370. DOI: 10.1002/rob.20336.

[21] Larry Matthies et al. «Lunar Rover Localization Using Craters as Landmarks». In: *2022 IEEE Aerospace Conference (AERO)*. 2022, pp. 1–17. DOI: 10.1109/AERO53065.2022.9843714.

[22] Li Wei and Sukhan Lee. «3D peak based long range rover localization». In: *2016 7th International Conference on Mechanical and Aerospace Engineering (ICMAE)*. 2016, pp. 600–604. DOI: 10.1109/ICMAE.2016.7549610.

[23] Albert S. Huang et al. «Visual Odometry and Mapping for Autonomous Flight Using an RGB-D Camera». In: *Robotics Research : The 15th International Symposium ISRR*. Ed. by Henrik I. Christensen and Oussama Khatib. Cham: Springer International Publishing, 2017, pp. 235–252. ISBN: 978-3-319-29363-9. DOI: 10.1007/978-3-319-29363-9_14. URL: https://doi.org/10.1007/978-3-319-29363-9_14.

[24] Ke Sun et al. «Robust Stereo Visual Inertial Odometry for Fast Autonomous Flight». In: *IEEE Robotics and Automation Letters* 3.2 (2018), pp. 965–972. DOI: 10.1109/LRA.2018.2793349.

[25] Yuwei Cheng et al. *Are We Ready for Unmanned Surface Vehicles in Inland Waterways? The USVInland Multisensor Dataset and Benchmark*. 2021. arXiv: 2103.05383 [cs.RO].

[26] Øystein Volden, Annette Stahl and Thor I. Fossen. «Vision-based positioning system for auto-docking of unmanned surface vehicles (USVs)». In: *International Journal of Intelligent Robotics and Applications* 6 (2022). DOI: 10.1007/s41315-021-00193-0. URL: https://doi.org/10.1007/s41315-021-00193-0.

[27] David Cortes-Vega, Hussain Alazki and Jose Luis Rullan-Lara. «Visual Odometry-Based Robust Control for an Unmanned Surface Vehicle under Waves and Currents in a Urban Waterway». In: *Journal of Marine Science and Engineering* 11.3 (2023). ISSN: 2077-1312. DOI: 10.3390/jmse11030515. URL: https://www.mdpi.com/2077-1312/11/3/515.

[28] Øystein Kaarstad Helgesen et al. «Heterogeneous multi-sensor tracking for an autonomous surface vehicle in a littoral environment». In: *Ocean Engineering* 252 (2022), p. 111168. ISSN: 0029-8018. DOI: https://doi.org/10.1016/j.oceaneng.2022.111168. URL: https://www.sciencedirect.com/science/article/pii/S0029801822005753.

[29] Ryan Wen Liu et al. «Intelligent Edge-Enabled Efficient Multi-Source Data Fusion for Autonomous Surface Vehicles in Maritime Internet of Things». In: *IEEE Transactions on Green Communications and Networking* 6.3 (2022), pp. 1574–1587. DOI: 10.1109/TGCN.2022.3158004.

[30] Zhijie Zhou et al. «A game theory-based fusion algorithm for autonomous navigation of smart ships». In: *Measurement* 216 (2023), p. 112897. ISSN: 0263-2241. DOI: https://doi.org/10.1016/j.measurement.2023.112897. URL: https://www.sciencedirect.com/science/article/pii/S026322412300461X.

[31] Hongjie Ma et al. «Radar Image-Based Positioning for USV Under GPS Denial Environment». In: *IEEE Transactions on Intelligent Transportation Systems* 19.1 (2018), pp. 72–80. DOI: 10.1109/TITS.2017.2690577.

[32] Jongdae Jung et al. «Navigation of Unmanned Surface Vehicles Using Underwater Geophysical Sensing». In: *IEEE Access* 8 (Jan. 2020), pp. 208707–208717. DOI: 10.1109/ACCESS.2020.3038816.

[33] Robert H. Rogne et al. «MEMS-based Inertial Navigation on Dynamically Positioned Ships: Dead Reckoning». In: *IFAC-PapersOnLine* 49.23 (2016). 10th IFAC Conference on Control Applications in Marine SystemsCAMS 2016, pp. 139–146. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2016.10.334. URL: https://www.sciencedirect.com/science/article/pii/S2405896316319218.

[34] David Cormack et al. «Joint Registration and Fusion of an Infrared Camera and Scanning Radar in a Maritime Context». In: *IEEE Transactions on Aerospace and Electronic Systems* 56.2 (2020), pp. 1357–1369. DOI: 10.1109/TAES.2019.2929974.

[35] Xuan Deng et al. «A Compact Mid-Wave Infrared Imager System With Real-Time Target Detection and Tracking». In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 15 (2022), pp. 6069–6085. DOI: 10.1109/JSTARS.2022.3192311.

[36] Jungwook Han, Yonghoon Cho and Jinwhan Kim. «Coastal SLAM With Marine Radar for USV Operation in GPS-Restricted Situations». In: *IEEE Journal of Oceanic Engineering* 44.2 (2019), pp. 300–309. DOI: 10.1109/JOE.2018.2883887.

[37] Ultralytics. *YOLOv8*. 2023. URL: https://github.com/ultralytics/ultralytics.

[38] S. Mahdi H. Miangoleh et al. *Boosting Monocular Depth Estimation Models to High-Resolution via Content-Adaptive Multi-Resolution Merging*. 2021. arXiv: 2105.14021 [cs.CV].

[39] Maxime Roedelé. «Surface vessel navigation in coastal environments using mono-vision camera sensor and 3D Digital Elevation Model». In: (2022).

[40] Iain E. G. Richardson. *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia*. 2003rd ed. United Nations, 2003, pp. 228–234.

[41] J. E. Bresenham. «Algorithm for computer control of a digital plotter». In: *IBM Systems Journal* 4.1 (1965), pp. 25–30. DOI: 10.1147/sj.41.0025.

[42] *We Must Draw the Line*. Medium. URL: https://jccraig.medium.com/we-must-draw-the-line-1820d49d19dd (visited on 13th Feb. 2023).

[43] *QGIS: A Free and Open Source Geographic Information System*. 2023. URL: https://qgis.org/en/site/.

[44] *Qgis2threejs*. 2023. URL: https://plugins.qgis.org/plugins/Qgis2threejs/.

[45] *Three.js – JavaScript 3D Library*. 2023. URL: https://threejs.org/.

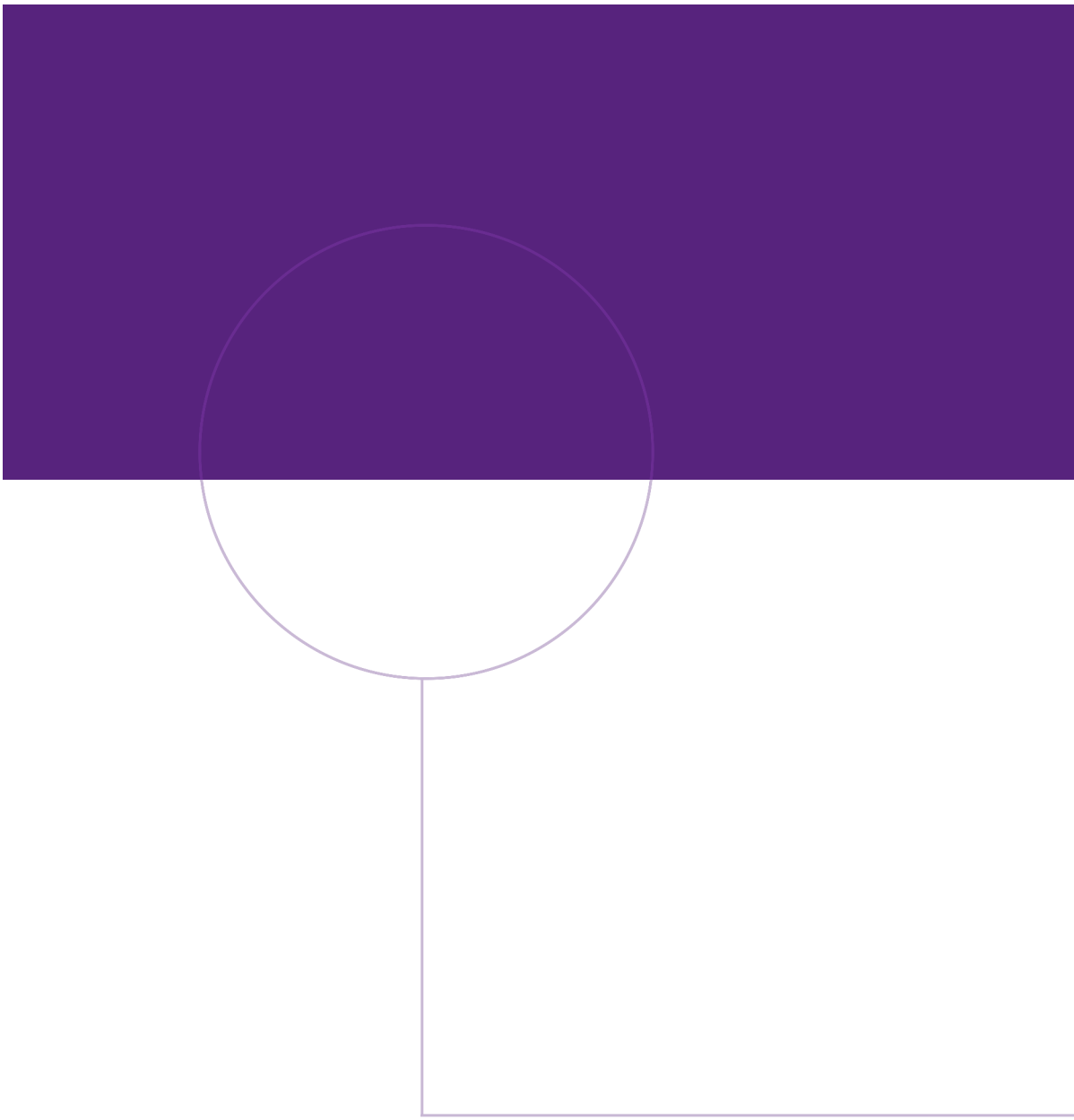
[46] *Blender*. 2022. URL: https://www.blender.org/.

[47] Steve Borkman et al. *Unity Perception: Generate Synthetic Data for Computer Vision*. 2021. DOI: 10.48550/ARXIV.2107.04259. URL: https://arxiv.org/abs/2107.04259.

[48] *NumPy*. 2023. URL: https://numpy.org/.

[49] *OpenCV*. 2023. URL: https://opencv.org/.

[50] *Norges Kartverk*. 2023. URL: https://www.kartverket.no/.



NTNU

Norwegian University of
Science and Technology