Master's thesis

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

Mathias Thoresen Paasche

Real-time 360-degree bird's eye view for the operator of milliAmpere2

Master's thesis in Cybernetics and Robotics
Supervisor: Edmund F. Brekke
Co-supervisor: Øystein K. Helgesen
June 2023

NTNU
Norwegian University of
Science and Technology

Mathias Thoresen Paasche

Real-time 360-degree bird's eye view for the operator of milliAmpere2

Master's thesis in Cybernetics and Robotics
Supervisor: Edmund F. Brekke
Co-supervisor: Øystein K. Helgesen
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

Preface

The work presented is the result of a Master thesis written in the 10^{th} semester of the Master of Science program at the Norwegian University of Science and Technology, NTNU, Department of Engineering Cybernetics. The thesis is related to the AUTOFERRY project, which is about the concept of small autonomous passenger ferries in urban areas as a more flexible and environmentally-friendly alternative to bridges or manned ferries, and to the Center for Research-Based Innovation SFI AUTOSHIP on autonomous ships for safe and sustainable operations.

I would like to thank my supervisor Associate Professor Edmund F. Brekke and advisor Dr. Øystein Kaarstad Helgesen for their time, feedback and interesting discussions during this work. In addition, I would like to thank Ph.D. Candidate Nicholas Dalhaug for helping me in the attempt to calibrate the cameras on *milliAmpere2*.

The abstract of this thesis has been accepted to The International Conference on Maritime Autonomous Surface Ships (ICMASS) in Rotterdam, $8^{th} - 9^{th}$ of November 2023. The submitted article based upon this thesis can be found in the appendix A.1.

Mathias Thoresen Paasche
Trondheim, 9.6.2023

Abstract

In the evolving domain of autonomous marine operations, accurate perception and representation of the surrounding environment is crucial for safe and effective execution. This thesis addresses this issue by developing and testing a near real-time 360 degrees bird's eye view system for the situations where the ferry, *milliAmpere2*, has to be manually controlled by a local operator onboard. The goal was to aid the operator during the critical phase of docking, by displaying the surrounding area of the ferry from a bird's eye view. The bird's eye view was made by using inverse perspective mapping on the undistorted images from the 8 cameras onboard.

Implemented in `Python`, the system aimed to run in real-time, necessitating a run-time of less than 200*ms* between each bird's eye view image. During the code optimization phase, this goal was reached. However, the slower CPU onboard *milliAmpere2* and the overhead of accommodating ROS2 node-structure, combined with the impact of other processes running concurrently on the ferry's computer, resulted in borderline real-time performance during the live experiment.

Despite the system's shortcomings, such as inaccurate calibration causing image artifacts, and an oversized image of *milliAmpere2* in the center of the IPM image, the operators found the system to be a "useful additional assistance" during the docking process. Nevertheless, the system needs further refinement before it is ready for continuous real-world deployment.

Sammendrag

Innen det voksende feltet autonom maritim navigasjon, er presis oppfatning og representasjon av det nærliggende miljøet avgjørende for sikker og effektiv drift. Denne masteroppgaven bidrar til å løse dette problemet gjennom å utvikle og teste et system som produserer et "360 graders fugleperspektiv"-bilde i opp imot sanntid, for situasjoner der fergen, *milliAmpere2*, må styres manuelt av en lokal operatør ombord.

Målet var å hjelpe operatøren i den kritiske fasen der fergen legger til kai, ved å vise området rundet fergen fra et fugleperspektiv. Fugleperspektivet ble laget ved å bruke "inverse perspective mapping" på de korrigerte bildene fra kameraene ombord.

Systemet, som er implementert i `Python`, var designet for å kjøre i sanntid og måtte derfor ha en kjøretid på mindre enn 200*ms* mellom hvert fugleperspektivbilde. Dette ble oppnådd under kode optimaliseringen fasen. På tross av måloppnåelsen, førte den tregere CPU-en om bord på *milliAmpere2*, sammen med den ekstra belastningen fra ROS2 nodestrukturen og påvirkningen fra andre kjørende prosesser på fergens datamaskin, til at kjøretiden var i grenseland for sanntidskjøring under selve eksperimentet.

Selv med systemets svakheter tatt i betraktning, som unøyaktig kamera kalibrering og et for stort bilde av *milliAmpere2* i midten av fugleperspektivbilde, syntes operatørene at systemet var et "nyttig verktøy" for å legge til kai. Likevel krever systemet videre forbedringer før det er klart for bruk i vanlig drift.

Contents

Acronyms

2D two-dimensional. 7, 13

3D three-dimensional. 7, 10

AAI advanced array indexing. 46, 49, 65

CNN convolutional neural network. 3, 4

CPU central processing unit. 4

FOV field of view. 23, 25, 34, 36, 37, 44, 46, 49, 67–71, 75, 77

GPU graphics processing unit. 4, 78

IPM inverse perspective mapping. 2–4, 7, 10, 12, 23, 24, 27, 31, 34, 36, 37, 39, 40, 42, 44, 45, 47, 49, 51, 53–55, 57, 58, 61, 63, 64, 66–68, 71–73, 75, 77, 78

kd k-dimensional. 14–16

NTNU Norwegian University of Science and Technology. iii, 1, 2, 53

RGB red, green, blue. 28, 40, 42–44

ROS robot operating system. v, vii, 39, 40, 51, 53, 54, 57–60, 66

Chapter 1

Introduction

1.1 Motivation

In recent years, there has been a growing demand for sustainable transportation in cities. However, in cities with rivers and channels, building bridges can be challenging due to various environmental and logistical factors. When the municipality of Trondheim proposed to build a bridge across one of the canals in Trondheim in 2016, researchers at the NTNU set in motion a project to find alternatives to bridges. The result was the prototype *milliAmpere* (hereby called *milliAmpere1*), and the full-scale autonomous passenger ferry *milliAmpere2* [1], see Figure 1.1.

Figure 1.1: An image of *milliAmpere2* from the side. Image provided by Øystein Kaarstad Helgesen.

With transportation of people, safety requirements are high. Therefore, it is possible to manually take control over *milliAmpere2* from the controls onboard [2]. However, the operator's view fore and aft of the ferry is obstructed by the

ramp, as shown in Figure 1.2. To improve the operator's situational awareness, especially during the critical operation of docking to shore, a bird's eye view image created from the images taken by the onboard cameras was proposed.

In my project thesis, a proof of concept of a top-down view using inverse perspective mapping (IPM) was presented. In this master thesis, the further development of this research will be presented.

(a) The image shows an overview of the operator at the controls of the ferry with the ramp in view.

(b) The image provides the view of the operator at the controls looking forwards.

Figure 1.2: The figure illustrates how the operator's forward view of the ferry is obstructed by the ramp. The same holds true for the aft view of the ferry.

1.2 Background information

In the last years, there has been an increasing interest for urban ferries in many coastal cities. Within this revival of urban ferries, the concept of autonomous electrical passenger ferries has received recognition as a cost-effective, environmentally friendly and flexible transport alternative [3]. In several countries, Norway, Finland and the Netherlands, autonomous ferry projects have been tested and demonstrated [1]. One of these projects comes from NTNU, Trondheim, Norway, and consists of two autonomous passenger ferries: the prototype *milliAmpere1*, see Figure 1.3, and the full-scale *milliAmpere2*. The project has been used extensively in research on marine autonomy by master and PhD students at NTNU, such as [4], [5] and [6].

In [7] a novel system for multi-camera maritime tracking is proposed. It describes how detected measurements given by bounding boxes in images can be converted to measurements on the sea plane, which can be used for tracking. This is achieved with the combination of information from the navigation system and statistical modelling of the geometric transformations.

Figure 1.3: An image of *milliAmpere1* from the side. Image provided by Øystein Kaarstad Helgesen.

1.3 Related work

One of the essential parts to achieve the desired 360-degree bird's eye view, is inverse perspective mapping (IPM). IPM is a technique that transforms images from the perspective view created by a camera to a bird's eye view. Most research on IPM has been done within the automobile industry. The earliest research was directed towards improving car safety, especially automatic lane tracking and obstacle avoidance.

In [8] an automatic lane tracking system was presented. The system used IPM to reverse the perspective view created by the camera, to improve detection of lane markers. This method was further improved upon by [9]. They introduced a multimodal IPM, by combining images from a camera and a laser range finder. Using the data from the laser range finder, all pixels that were above the road could be removed, and thereby lower the amount of pixels that had to be processed by the IPM algorithm. Another improvement to the classic IMP algorithm was proposed by [10] in 2019. They included a convolutional neural network (CNN) to sharpen the IPM images. This is especially useful for objects that are far away, due to lower pixel density per object.

Alongside the development of automatic lane tracking, researchers have also researched how IPM can be used in obstacle avoidance. Using the distorted images of cars and objects produced by the IPM transformation, [11] was able to create a robust method for generic obstacle detection and collision warning with no prior knowledge of the obstacle. It was also able to detect partially covered objects.

Although IPM has received some attention within the maritime industry, the amount of research in this sector remains lower than within the automobile industry. [12] presented a path planning and navigation method for autonomous vessels using CNN and IPM. Each part of the image is categorized by the CNN as "open water", "partial open water" or "not water". Thereafter, the categorized image is transformed to a bird's eye view using IPM to make the path planning easier.

A similar method is proposed by [13] to obtain $3D$ ship detection and tracking. Each image is processed by a CNN, which detects ships and boats, and places a bounding box and segmentation mask over each detection. Using the lower edge (the one closest to the camera) of the segmentation mask, they calculate the distance to the ship using IPM. To get the height of the ship, they use the geometry of a pinhole camera, and they set the depth of the ship to be similar to the largest ship in the harbor.

One of the main challenges with IPM, is the progressively lower pixel density for objects further away from the camera. Interpolation can decrease the effect of this problem by filling the empty pixels in the objects with *rgb*-values from surrounding pixels.

The research field of image interpolation has taken place in many different industries: Medical imaging, remote sensing, target detection and recognition, radar imaging, forensic science and surveillance systems [14]. In addition to covering a large group of different industries, a large amount of different interpolation methods has been developed. [15] compared 7 different interpolation methods for medical imaging in 1999, covering "traditional interpolation methods", such as nearest neighbor, linear interpolation and Gaussian interpolation with different kernel sizes.

In recent years, more complex and modern interpolation methods have been developed. One of these are *Super-Resolution*, which are able to enhance low-resolution images or video frames by increasing their spatial resolution [14]. This technique has also been combined with deep learning: [16] proposed in 2016 an end-to-end mapping between the low and high resolution images using a deep CNN. Compared to a simpler method, bicubic interpolation, and to more advanced interpolation methods, it achieves a better peak singal-to-noise ratio and produces sharper edges in the image.

To be able to create the bird's eye view images in real-time, the IPM image pipeline has to process 8 images in less than $200ms$. This requires a powerful CPU. In the last decades, the performance of a CPU has increased some 10000 times, without a substantial power consumption increase. The information technology industry has grown to rely on this constant increase in performance, making it an expectation and a necessity for many applications.

However, as stated by [17], the era of faster and better individual CPUs has come to an end due to physical limitations of power consumption and heat dissipation. To continue the growth in industry, new approaches to software and parallelism have to be incorporated. This is further supported by [18], describing how Intel introduced a new "optimization" phase in their CPU development cycle. Another approach to continue the growth is to use a dedicated GPU instead of a CPU. [19] demonstrated this through developing the `Python` library (`pyFFS`) for computing fast Fourier series and interpolation, achieving improvements in the order of one magnitude better compared to CPU.

1.4 Problem description and main contributions

To develop a fully functional system from the proof of concept system presented in my project thesis, several challenges and problems need to be addressed. One of the primary objectives of this system is to attain real-time functionality, which demands significant optimization of the proof of concept code. Additionally, another significant challenge lies in adapting the code to run on the onboard system of *milliAmpere2*. This requires system and code restructuring to ensure compatibility to the existing autonomy system onboard. By successfully addressing these challenges, an operational bird's eye view system can be tested by the operators in a live, real-world situation.

This thesis' main contributions are:

- Exploration and implementation of interpolation and optimization methods in Python.
- Implementation of 360-degree bird's eye view that is able to run in close to real-time on the autonomous passenger ferry *milliAmpere2* with the full autonomy system running simultaneously.
- Testing of 360-degree bird's eye view system with operator on *milliAmpere2* in a comparable setting to the intended use case of the ferry.
- Attaining valuable feedback from the operators following the live experiment.

1.5 Outline

Chapter 2 is a theory chapter that provides an overview of the camera model, distortion and undistortion, inverse perspective mapping and interpolation. Chapter 3 outlines the image processing pipeline, including the recorded dataset used in the first part of the thesis. The code optimization techniques used to reduce the processing time of the pipeline, and the achieved run-times will be presented and discussed in Chapter 4. In Chapter 5 the experimental setup of the live experiment will be introduced. The results from the experiment are shown in Chapter 6. Run-times, images and video results are included. In Chapter 7 a discussion around the code optimization and the results from the experiment. Lastly, a conclusion and future work will be presented in Chapter 8.

Chapter 2

Theory

In this chapter, the theory behind camera models, image distortion and undistortion, inverse perspective mapping, and image interpolation will be explained. These topics are essential for understanding the image processing pipeline presented in Chapter 3. Camera models are important, since the geometry behind IPM assumes the images are taken with a camera that follows the pinhole camera model. Image distortion and undistortion techniques enables us to correct image distortions made by the camera lens to satisfy the prior assumption. The concept of inverse perspective mapping is the essential theory to be able to transform the camera images into a bird's eye view. Finally, understanding image interpolation is required to explain how empty pixels in the IPM image are filled with relevant values.

Parts of this chapter have been taken from my project thesis, with some minor corrections. These are Section 2.1, Section 2.2, Section 2.3, Section 2.4.1.

2.1 Camera model

A camera is a mapping between the 3D world and a 2D image [20]. There are different models describing cameras, each with their own camera matrix containing their different properties. In this thesis, the primary camera model is the central projection camera, which is a specialization of the general projective camera. Among projective camera models, there are two major classes: cameras with a finite center and cameras with the center at "infinity". The main distinction between these two models is that the camera with center at infinity is able to maintain parallel lines, whereas the finite center, will not maintain parallel lines.

One of the finite camera models, is the pinhole model. It maps a point in 3D space onto a 2D plane. This plane is often called the image plane or the focal plane, and is represented by the frame $\mathcal{F}_I$. The position of the point $\mathbf{x}_I$ is determined by the intersection between the image plane and the drawn line from the point $\mathbf{X}_C$ to the camera center through the image plane.

The camera center is also the origin of the camera frame $\mathcal{F}_c$. From Figure 2.1, one can observe that there is triangle similarity that can be used to calculate the

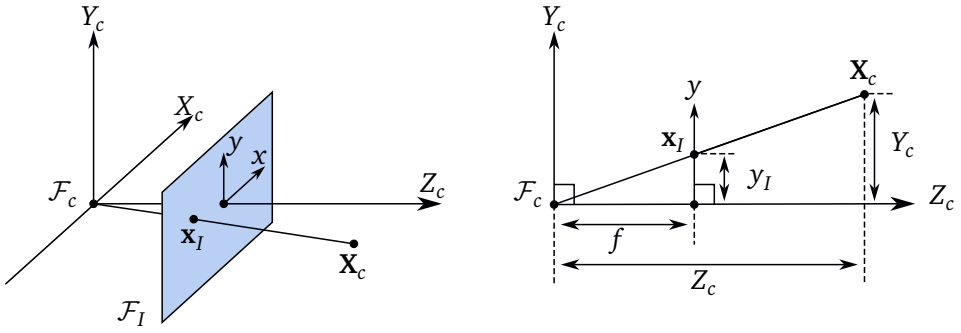


Figure 2.1: On the left: Illustration of the relative position of the camera frame $\mathcal{F}_C$ and the image plane $\mathcal{F}_I$. On the right: The geometry of the pinhole camera view from the side, showing the triangle similarity used to place a point $\mathbf{X}_C$ onto the image plane.

x- and y-position on the image plane, given position of $\mathbf{X}$. To make the notation more explicit, $\mathbf{X}$ will be written as $\mathbf{X}_c$. The relationship between $\mathbf{x}_I$ and $\mathbf{X}_c$ is given by

$$\mathbf{x}_I = \begin{bmatrix} x_I \\ y_I \end{bmatrix} = \begin{bmatrix} f\frac{X_c}{Z_c} \\ f\frac{Y_c}{Z_c} \end{bmatrix} \tag{2.1}$$

To make further calculations easier, homogenous coordinates can be introduced as $\tilde{\mathbf{x}}$. Then Equation 2.1 can be rewritten to

$$\tilde{\mathbf{x}}_I = \begin{bmatrix} x_I \\ y_I \\ 1 \end{bmatrix} = \begin{bmatrix} f & & & 0 \\ & f & & 0 \\ & & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{2.2}$$

and with the concise form

$$\tilde{\mathbf{x}}_I = \left[\begin{array}{c|c} \mathbf{K} & \mathbf{0}_{3\times 1} \end{array} \right] \tilde{\mathbf{X}}_c \tag{2.3}$$

Normally a point will not be represented in $\mathcal{F}_c$, but rather in the world coordinate frame, $\mathcal{F}_w$. These two frames are related with a rotation and a translation. To transform a point from the $\mathcal{F}_w$ to $\mathcal{F}_c$, one would translate the offset distance between the origin of $\mathcal{F}_c$ and origin of $\mathcal{F}_w$, represented by $\mathbf{C}$, and rotate to align the axes of the two frames with the rotation matrix $\mathbf{R}$. This can be written as Equation 2.4, and can be seen in Figure 2.2, where $\mathbf{t} = -\mathbf{R}_{cw}\mathbf{C}$. The subscript $_{CW}$ informs that the transformation is from $\mathcal{F}_W$ to $\mathcal{F}_C$.

$$\mathbf{X}_c = \mathbf{R}_{cw}(\mathbf{X}_w - \mathbf{C}) \tag{2.4}$$

Equation 2.4 expanded to homogenous coordinates will give the matrix multiplication

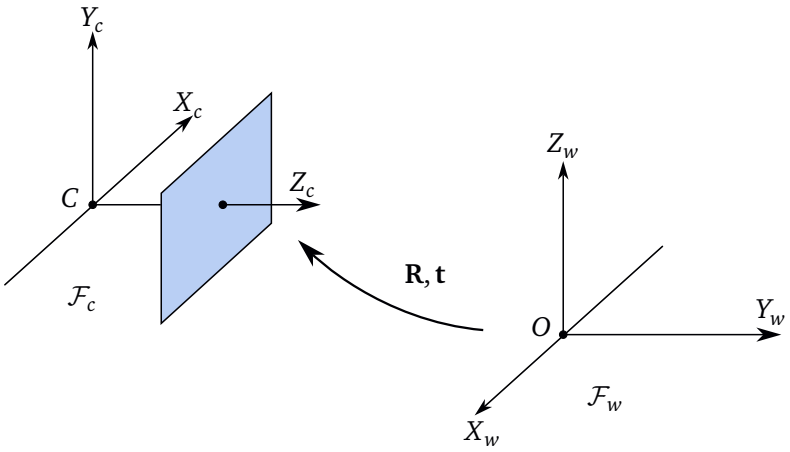


Figure 2.2: The transformation between the world and the camera coordinate frame.

$$\tilde{\mathbf{X}}_c = \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \left[\begin{array}{c|c} \mathbf{R_{cw}} & \mathbf{t} \\ \hline \mathbf{0}_{1\times 3} & 1 \end{array}\right] \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{2.5}$$

Combining Equation 2.5 and Equation 2.2 leads to the formula

$$\tilde{\mathbf{x}}_I = \mathbf{K}\left[\begin{array}{c|c} \mathbf{R_{cw}} & \mathbf{t} \\ \hline \mathbf{0}_{3\times 1} & 1 \end{array}\right] \tilde{\mathbf{X}}_w \tag{2.6}$$

To get the pixel coordinate, $\mathbf{u} = [u, v]^\intercal$, in the image, the intrinsic camera matrix $\mathbf{K}$ has to be modified. First, the origin of $\mathcal{F}_I$ has to be offset. Usually the origin in an image is in the top, left corner, see Figure 2.3. In addition, scaling and skew has to be added.

$$\mathbf{K} = \begin{bmatrix} s_x f & s & s_x p_x & 0 \\ & s_y f & s_y p_y & 0 \\ & & 1 & 0 \end{bmatrix} = \begin{bmatrix} \alpha_x & s & x_0 & 0 \\ & \alpha_y & y_0 & 0 \\ & & 1 & 0 \end{bmatrix} \tag{2.7}$$

The numbers s_x and s_y gives the amount of pixels per unit (often mm) for the image. s is for the skew, which normally is 0. p_x and p_y is the principal point offset, which moves the origin from the center of the image to the top, left corner. It is also important to note that the v-axis is the opposite direction of y-axis. This is only due to convention.

This produces the final equation

$$\mathbf{u} = \mathbf{P}\tilde{\mathbf{X}}_w \tag{2.8}$$

where

$$\mathbf{P} = \begin{bmatrix} \alpha_x & s & x_0 & 0 \\ & \alpha_y & y_0 & 0 \\ & & 1 & 0 \end{bmatrix} \left[\begin{array}{ccc|c} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{2.9}$$

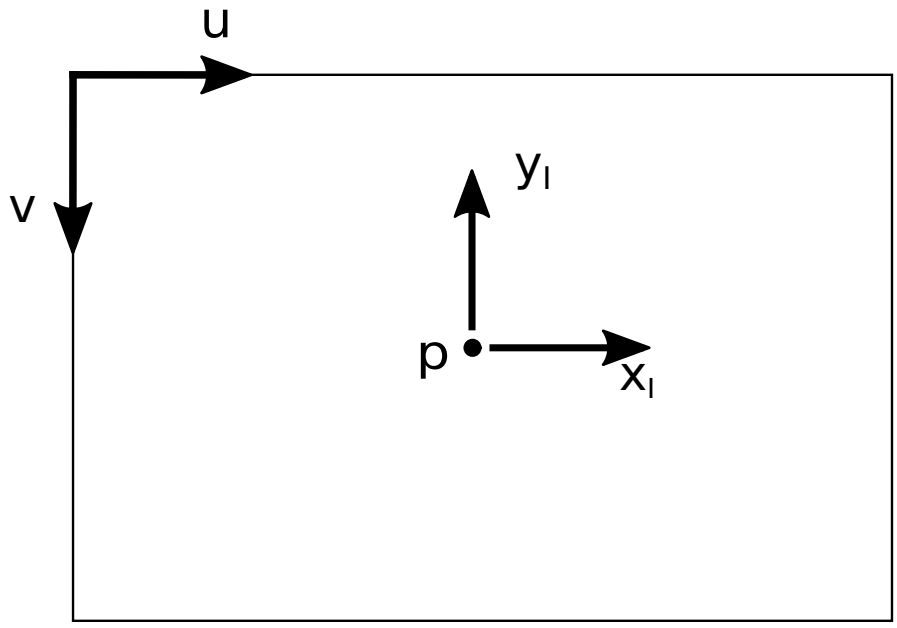


Figure 2.3: An image showing the image plane, with the principal point p, the axes of $\mathcal{F}_I$ in the middle. In the top left are the offset origin, with the two axes showing the orientation of the coordinate frame used when describing pixel position.

2.2 Distortion and undistortion

Li *et al.* described the task of undistortion as:

> *An indispensable pre-processing step for image-based 3D reconstruction or photogrammetry tasks is undistortion, which is employed for correcting the non-linear projection of the surface points of objects onto the image plane due to lens distortion* [21].

The non-linear projection from lens distortion is a deviation from the ideal projection in the pinhole camera model [22] this thesis is based upon. Therefore, it is necessary to undistort the images before IPM is done. Undistortion is often

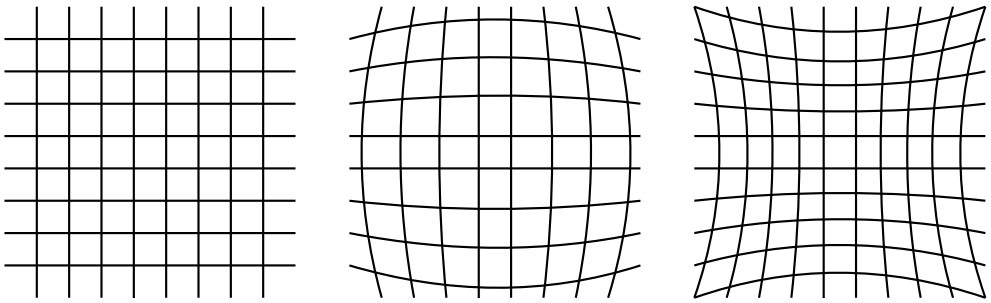


Figure 2.4: Examples of different lens distortions on a rectangular grid (left): barrel distortion (middle) and pincushion distortion (right)[22].

done in two steps; first a pixel position transformation, and then an interpolation of the new pixel coordinates to the image grid.

The first step is often completed with the use of Brown-Conrady model [22]. It takes the undistorted image coordinates and transforms them into distorted image coordinates. If only second order radial distortions are included from the model, the equations would be

$$u_d = u_n\left(1 + k_1 r^2 + k_2 r^4\right) \tag{2.10a}$$
$$v_u = c_n\left(1 + k_1 r^2 + k_2 r^4\right) \tag{2.10b}$$

If tangential distortions are also included, such as in `OpenCV`'s implementation [23], then Equation 2.10 can be extended to

$$u_d = u_n\left(1 + k_1 r^2 + k_2 r^4\right) + \left(2p_1 u_n v_n + p_2\left(r^2 + 2u_n^2\right)\right) \tag{2.11a}$$
$$v_u = v_n\left(1 + k_1 r^2 + k_2 r^4\right) + \left(2p_2 u_n v_n + p_1\left(r^2 + 2v_n^2\right)\right) \tag{2.11b}$$

where (u_d, v_d) are coordinates in the distorted image, (u_n, v_n) are the coordinates in the undistorted image and $r = \sqrt{(u_n - p_x)^2 + \left(v_n - p_y\right)^2}$. The numbers p_x and p_y are the coordinates of the principal point, while k_1 and k_2 are radial distortion coefficients and p_1 and p_2 are tangential distortion coefficients. If the value of k_1 is negative, then we typically have barrel distortions, and if k_1 is positive, we typically have pincushion distortions [22]. Examples of the two radial distortions can be seen in Figure 2.4. Tangential distortions are deviations where the principal point is moved from the center of the image.

With the coordinates $\mathbf{u_n}$ and $\mathbf{u_d}$, the second step of undistorting the image, interpolation, can be conducted. How interpolation work will be explained in section 2.4.

2.3 Inverse Perspective mapping

Inverse perspective mapping is the problem of obtaining the world coordinate position of a point from a pixel in an image [9]. This is an underdetermined problem, meaning there are fewer equations than unknowns. In this problem, there are three unknowns, X_w, Y_w, Z_w, and two knowns, u, v. To solve this, the plane where the pixels are projected onto has to be assumed. In this thesis, the model is assumed to use the sea as the plane, Π_{sea}. The equation for an arbitrary plane is given by

$$aX + bY + cZ + d = 0 \tag{2.12}$$

To address the IPM problem, [9] extended the non-homogenous version of Equation 2.1, $\mathbf{u} = \mathbf{K}(\mathbf{R}_{CW}\mathbf{X}_W + \mathbf{t})$, to incorporate the constraints from the plane.

$$\begin{bmatrix} u \\ v \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & 0 \\ p_{21} & p_{22} & p_{23} & 0 \\ p_{31} & p_{32} & p_{33} & 0 \\ a & b & c & d \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \\ 0 \end{bmatrix} \tag{2.13}$$

The d-variable can be moved to the translation vector and the Equation 2.13 can be rearranged to:

$$\begin{bmatrix} -t_x \\ -t_y \\ -t_z \\ -d \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & 0 \\ p_{21} & p_{22} & p_{23} & 0 \\ p_{31} & p_{32} & p_{33} & 0 \\ a & b & c & 0 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} - \begin{bmatrix} u \\ v \\ 1 \\ 0 \end{bmatrix} \tag{2.14}$$

Then the pixel coordinate vector, $\mathbf{u}$, can be included in the $\mathbf{P}$:

$$\begin{bmatrix} -t_x \\ -t_y \\ -t_z \\ -d \end{bmatrix} = \underbrace{\begin{bmatrix} p_{11} & p_{12} & p_{13} & -u \\ p_{21} & p_{22} & p_{23} & -v \\ p_{31} & p_{32} & p_{33} & -1 \\ a & b & c & 0 \end{bmatrix}}_{\mathbf{A}} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{2.15}$$

Lastly Equation 2.15 can be rearranged to solve for $\mathbf{X}_w$:

$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & -u \\ p_{21} & p_{22} & p_{23} & -v \\ p_{31} & p_{32} & p_{33} & -1 \\ a & b & c & 0 \end{bmatrix}^{-1} \begin{bmatrix} -t_x \\ -t_y \\ -t_z \\ -d \end{bmatrix} \tag{2.16}$$

This solution is valid as long as $\mathbf{A}$ in Equation 2.15 is invertible.

Another way to solve the IPM, is to make use of the method described by Hartley and Zisserman [20]. It is a less general method than the previous, and requires the plane to always be at a height of zero meter. To get the pixel position

u from a point $\mathbf{X}_w$ on a plane that is aligned with XY-plane in $\mathcal{F}_w$, Equation 2.17 can be used.

$$\begin{bmatrix} u \\ v \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & t_x \\ p_{21} & p_{22} & p_{23} & t_y \\ p_{31} & p_{32} & p_{33} & t_z \\ a & b & c & d \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{2.17}$$

Since the transformation is to the plane $Z_w = 0$, $a = b = c = Z_w = 0$, and the Equation 2.17 can be reduced to:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & t_x \\ p_{21} & p_{22} & t_y \\ p_{31} & p_{32} & t_z \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ 1 \end{bmatrix} \tag{2.18}$$

Rearranging Equation 2.18 solves the inverse perspective mapping problem instead of finding the pixel coordinate given a point $\mathbf{X}_w$ in $\mathcal{F}_w$.

$$\begin{bmatrix} X_w \\ Y_w \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} p_{11} & p_{12} & t_x \\ p_{21} & p_{22} & t_y \\ p_{31} & p_{32} & t_z \end{bmatrix}^{-1}}_{\mathbf{P}'} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{2.19}$$

For this to be a valid solution, $\mathbf{P}'$ has to be invertible.

2.4 Interpolation

2.4.1 Interpolation methods

Interpolation is the process of estimating the intermediate values in a signal at continuous positions from a set of discrete samples [24]. This process can be employed in multiple dimensions. However, in this thesis, the focus will be on 2D interpolation, which is relevant to image interpolation. The three most commonly used interpolation methods are nearest neighbor, bilinear (also called linear), and bicubic.

Nearest neighbor interpolation involves determining the value of a pixel based upon the four nearest pixels. The value of the pixel with the shortest distance to the target pixel will be the chosen values, as shown in Equation 2.20.

$$\text{Value}(P) = \min(\text{dist}(A), \text{dist}(B), \text{dist}(C), \text{dist}(D)) = \min(a, b, c, d) \tag{2.20}$$

An example of nearest neighbor interpolation is illustrated in Figure 2.5, where dist(A) would be the minimum distance, and therefore A would be the value of P.

Bilinear interpolation also considers the four closest pixels values to the target pixel, but weights them based on their distance from the target pixel. The values

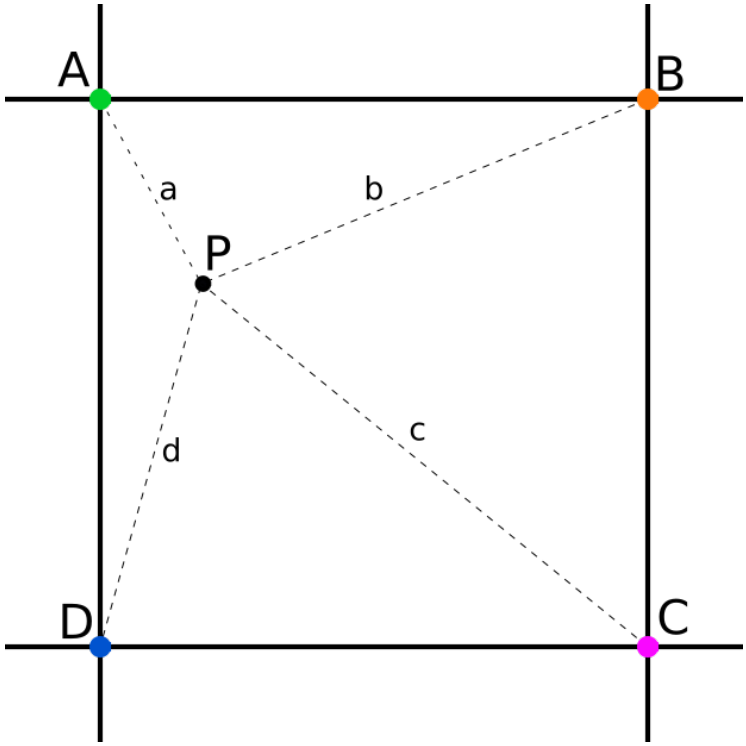


Figure 2.5: Interpolation: Nearest neighbor method.

are first interpolated linearly in the horizontal direction using Equation 2.21, and then in the vertical direction using Equation 2.22, as shown in Figure 2.6.

$$E = a * A + (1 - a)B \tag{2.21a}$$
$$F = a * D + (1 - a)C \tag{2.21b}$$

$$P = b * F + (1 - b)E \tag{2.22}$$

2.4.2 KD-Tree

A kd-tree is a type of binary tree used for efficient search and retrieval of points in high-dimensional space. It is particularly useful for range search and nearest neighbor search operations. The name "kd" stands for "k-dimensional," referring to the fact that the tree can be constructed for any k number of dimensions.

The structure of a kd-tree is formed by recursively dividing the space containing the data points into two subspaces at each level of the tree. The split is made perpendicular to the chosen axis. For example, if the points are in two dimensions, the axis can be chosen as the x-axis at one level and the y-axis at the next level.

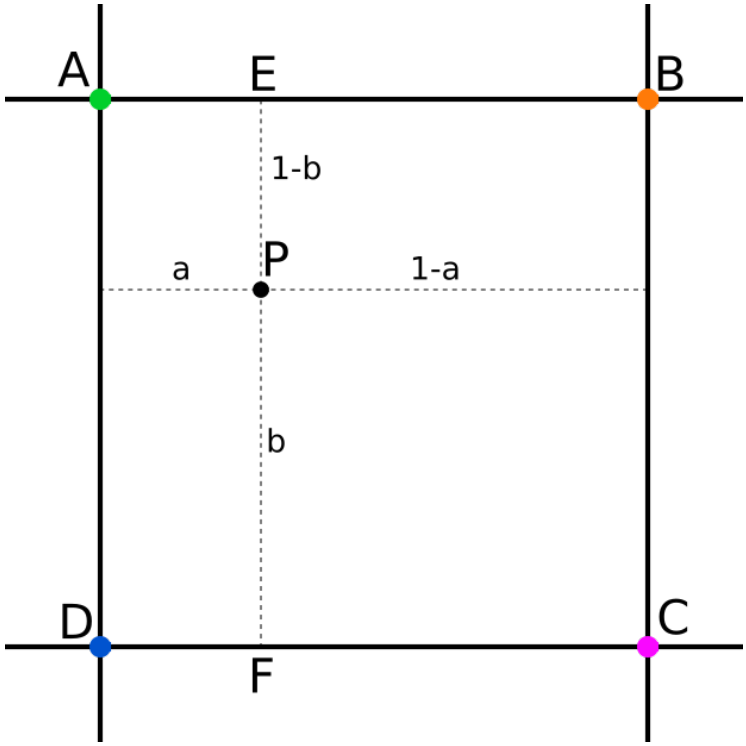


Figure 2.6: Interpolation: Bilinear method.

Data points in the "left" subspace are represented by the left subtree, while data points in the "right" subspace are represented by the right subtree. The terms "left" and "right" represent the two different sides of the split space. The terms could also have been "A" and "B", or "upper" and "lower", depending on the splitting axis and the number of dimensions in the space. However, "left" and "right" were chosen, since they are intuitive in two-dimensional space.

All data points in the left subtree have a lower value than the splitting axis value. For example, if the splitting axis is x and its value is 10, then all data points in the left subtree have a x-value of less than (or equal to, depending on splitting method) 10. The opposite is true for the right subtree. There, all the data points will have a x-value greater than (or equal to) 10. The process of dividing the space into subspaces is repeated recursively until there is only one point in each subspace. There is also the possibility of having 0 points in a subspace, depending on the splitting method [25].

An example of the method described above, can be seen in Figure 2.7. A set of points $S = \{(10, 10), (52, 15), (35, 25), (12, 38), (55, 42), (20, 50), (2, 68), (30, 58)\}$, is show in a 2-dimensional space. It is made into a kd-tree by splitting on the point closest to the middle of axis of choice in the subspace. First, it is split on x-axis, then y-axis, then lastly, x-axis again. The splitting line passes through the splitting point.

There are many ways to decide the splitting point and the axis to split upon,

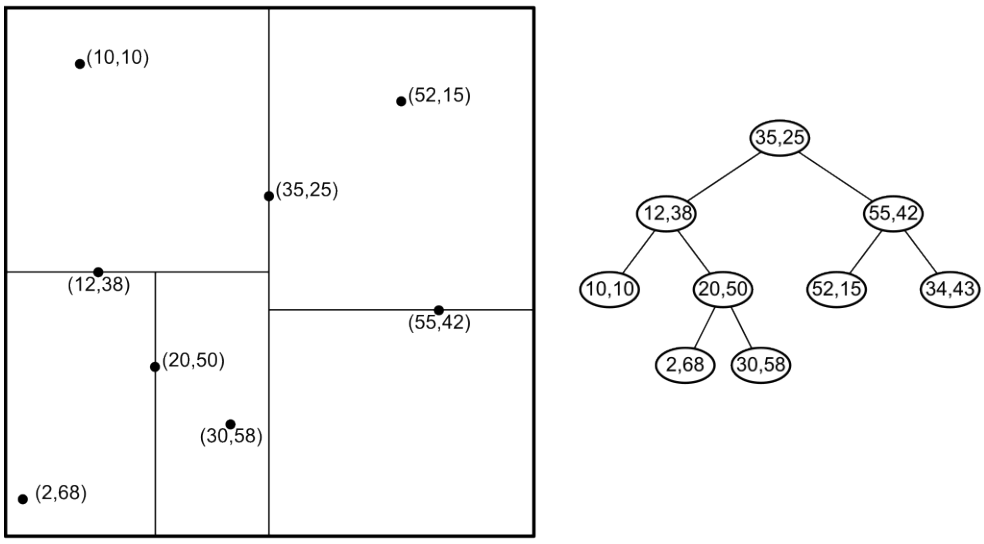


Figure 2.7: Illustration of the final product, showing the subdivision of the space on the left based on the closest point to the middle of the splitting axis. The kd-tree representation is depicted on the right.

from the method described above, to using the median value of the points in the space. All methods come with different pros and cons. One of the deficencies of the two mentioned methods are their handling of points clustered along an axis. For example, if data points are highly clustered along the x-axis, the splitting methods will create a deeper tree than necessary.

A better method would be to use the sliding-midpoint splitting method [24]. The advantage of the sliding-midpoint strategy is that it can result in a more balanced tree, with fewer levels and better search performance.

Sliding-midpoint splitting method uses a kd-tree with slight change in structure. Instead of having the coordinates of a point in each node, the method stores the axis it splits upon and the axis split value in each node. Only the leaf nodes contain data points. Moreover, it uses a midpoint rule as the first step of splitting a subspace. The idea behind the midpoint rule is to split the longest axis of the subspace in half. However, if this creates a trivial split (meaning one side has 0 data points), it will slide the splitting line towards the data points until it encounters a data point. It will then make the split on the other side of the data point, see Figure 2.8, to avoid a trivial split. This method guarantees no trivial splits. Nonetheless, it does not guarantee a balanced tree, which may result in a deeper tree than expected. In practice, the tree is rarely deeper than the depth of a balanced kd-tree [24].

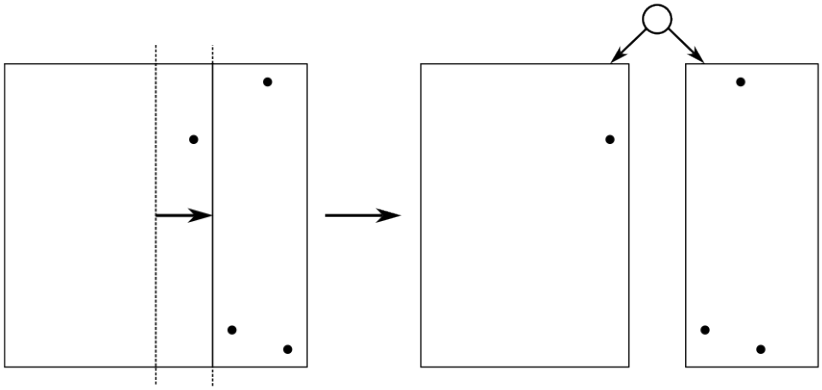


Figure 2.8: Illustration showing the sliding midpoint rule in action, where the splitting line is adjusted to the opposite side of the closest data point, effectively avoiding trivial splits.

2.4.3 Delaunay triangulation

A set of triangles are called a triangulation. This triangulation can be with or without any restrictions of how the triangles are located or shaped. However, most of the time restrictions are desired, to be certain some properties are preserved [26].

To create a triangulation, a finite set of points is used:

$$P = \{p_i\} \in \Omega, i = 1, ..., N. \tag{2.23}$$

Ideally, the domain Ω should be the convex hull of the set. A triangle in a triangulation is made up of three points p_i, p_j, and p_k which correspond to the vertices v_i, v_j, and v_k respectively. A single triangle t_{ijk} in a triangulation Δ is spanned by the vertices v_i, v_j, and v_k. A set of triples I_Δ is used to represent all the triangles in the triangulation Δ. Each triple $\mathbf{i} = (i, j, k) \in I_\Delta$ refers to the triangle t_{ijk} for some integer i. See Figure 2.9 for an illustration of a triangulation of a set of points.

While triangulations can be made up of any collection of triangles, certain families of triangulations are more relevant in practice. One of these families, where Delaunay triangulations are a family member, meet the following requirements [26]:

1. No triangle t_{ijk} in a triangulation Δ is degenerate, that is, if $(i, j, k) \in I_\Delta$, then p_i, p_j and p_k are not collinear.
2. The interiors of any two triangles in Δ do not intersect, that is, if $(i, j, k) \in I_\Delta$ and $(\alpha, \beta, \gamma) \in I_\Delta$, then

$$Int\left(t_{ijk}\right) \cap Int\left(t_{\alpha\beta\gamma}\right) = \emptyset$$

3. The boundaries of two triangles can only intersect at a common edge or at a common vertex.

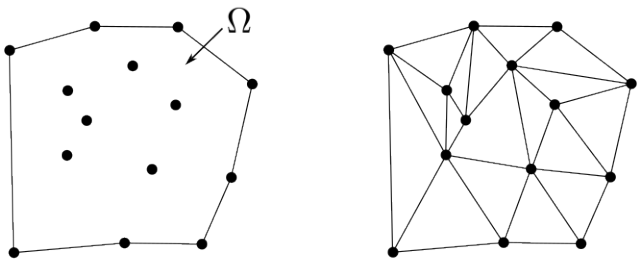


Figure 2.9: The graphic displays how a domain Ω (to the left) with a set of points can be triangulated. An example of the triangulation can be seen to the right.

4. The union of all triangles in a triangulation Δ is equal to the domain over which the triangulation is defined, that is
$$\Omega = \cup t_{ijk}, (i,j,k) \in I_{\Delta}$$
5. The domain Ω must be connected.
6. The triangulation shall not have holes.
7. If v_i is a vertex at the boundary $\partial\Omega$, then there must be exactly two boundary edges that have v_i as a common vertex. This implies that the number of boundary vertices is equal to the number of boundary edges.

If a triangulation Δ fulfills the four first requirements, then it is *valid*. If the last three requirements also are fulfilled, then the triangulation is *regular*.

For many applications, including interpolation, some form of "optimization" of triangle shape are sought after. Especially, flat and elongated triangles are undesired. Instead, an optimal triangulation consists of triangles that are near equiangular. Different criteria have been proposed to obtain such an optimal triangulation. One of these are the criteria of obtaining triangles with the *smallest maximum (MinMax)* angle. Another similar criteria, is the *largest minimal (MaxMin)* angle. Given the *MinMax* criteria, a set of points P can be triangulated in two different ways, and still satisfy the criteria. Nonetheless, all triangulations are not equally good. In Figure 2.10, it can be observed that the left triangulation consists of more poorly shaped triangles compared to the right triangulation. As a result, the right triangulation is considered more desirable.

To obtain a more precise definition of a Delaunay triangulation for the *MaxMin* criteria, an *indicator vector* $I\left(\Delta^k\right) = \left(\alpha_1, \alpha_2, ...\alpha_{|T|}\right)$ can be made, where Δ^k is a possible triangulation of a point set P, α_i is the smallest interior angle in each triangle t_{ijk} in Δ^k and $|T|$ is the number of triangles in Δ^k. If all triangulations are assumed to have the same border, then there is a fixed amount of triangles $|T|$ for all Δ^k. Arranging each indicator vector $I\left(\Delta^k\right)$ in a non-decreasing order,

$$I\left(\Delta^k\right) = \left(\alpha_1, \alpha_2, ...\alpha_{|T|}\right), \alpha_i \leq \alpha_j, i < j.$$

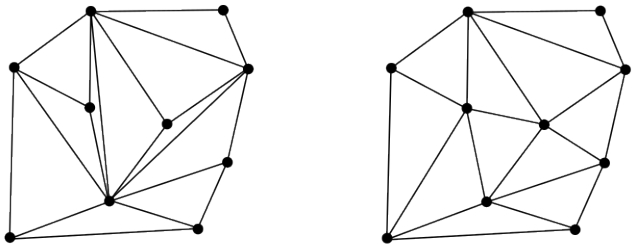


Figure 2.10: Comparison of two triangulations of a set of points P. The left triangulation consists of more poorly shaped triangles than the right triangulation. Therefore, the right triangulation is considered more desirable.

By lexicographically[1] ordering all indicator vectors $I\left(\Delta^k\right)$ in a non-decreasing order, the lexicographically largest indicator vector represent a triangulation that is considered the *optimal triangulation* and is "better" than the other triangulations.

This gives one definition of a Delaunay triangulation for *MaxMin* angle criteria:

> *A triangulation that is optimal in the sense of the MaxMin angle criterion and which is defined on the convex hull of a point set is called a Delaunay triangulation.*[26]

However, there are other ways to define Delaunay triangulations that are, from a practical point of view, more useful than the definition above. A commonly used definition is the based upon geometry, and more specifically circumcircles of the triangles in the triangulation Δ. A circumcircle is, as the name suggests, a circle around something. In this case, it is a circle enclosing a triangle, see Figure 2.11. A flat and elongated triangle (triangle on the right in Figure 2.11) produce a larger circle than a close to equiangular triangle (triangle on the left in Figure 2.11).

Using the circumcircle, a new definition of a Delaunay triangulation can be made:

> *A Delaunay triangulation Δ of a set of points P in the plane is a triangulation where the interior of the circumcircle of any triangle in Δ contains no point from P.* [26]

A larger example of Delaunay triangulation with circumcircles can be seen in Figure 2.12.

[1] A vector $\mathbf{v^a}$ is lexicographically larger than a vector $\mathbf{v^b}$ if for some integer m, $v_i^a = v_i^b$ for $i = 1, ..., m-1$, while $v_m^a > v_m^b$ [26].

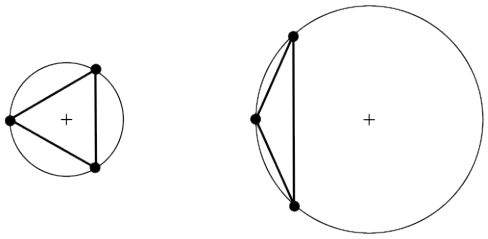


Figure 2.11: An illustration showing how the shape of the triangle affects the circumcircle of the triangle. The right triangle, which is flat and elongate, creates a larger circle compared to the left triangle, which is close to equiangular.

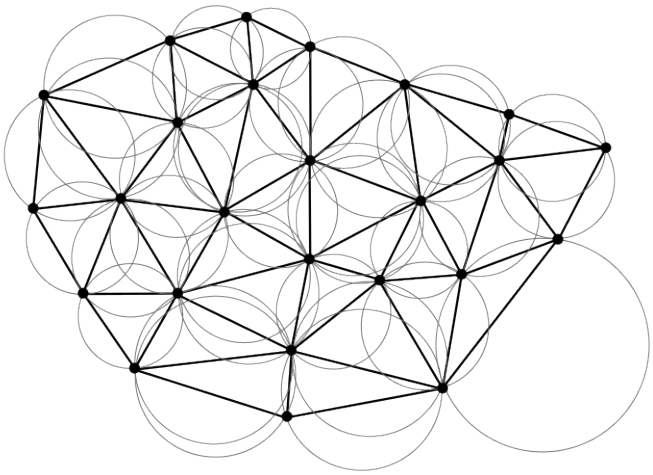


Figure 2.12: An illustration of a Delaunay triangulation with circumcircles.

2.4.4 Barycentric coordinates

Barycentric coordinates are a mathematical system used to specify the position of a point relative to the vertices in a space. This space is often of a triangle in a two- or three-dimensional space. Higher dimensions are possible, but not as common.

Consider a simpler space than a triangle: A segment represented by two points A and B. The position of an arbitrary point P on the line can be represented by

$$P = A + t\,(A - B) = (1 - t)A + tB, \tag{2.24}$$

or given $a + b = 1$,

$$P = aA + bB. \tag{2.25}$$

Point P is on the segment if, and only if, $0 \leq a \leq 1$ and $0 \leq b \leq 1$. For a segment, the barycentric coordinates for A and B are respectively $(1, 0)$ and $(0, 1)$. *Bary* comes from Greek, and means weight. In the previous example, a and

b was the weights of point A and B, positioned on the points. The point P can be expressed by the ratio $a : b$.

Going back to the two-dimensional triangle, the position of a point P inside a triangle t_{ABC} is expressed as a weighted sum of the positions of the three vertices A, B, and C, where the weights a, b, and c are proportional to the distances between the point P and the three vertices of the triangle. The barycentric coordinates of the vertices A, B, and C are respectively $(1,0,0)$, $(0,1,0)$ and $(0,0,1)$. A point P, with the barycentric coordinates (a,b,c), is inside, or on the edge of, the triangle if and only if, $0 \leq a,b,c \leq 1$ and $a+b+c = 1$. This gives the parametrization with vertex A as the origin, Equation 2.26a, and the general parametrization, Equation 2.26b.

$$P = A + b\,(B-A) + c\,(C-A) \tag{2.26a}$$

$$P = aA + bB + cC \tag{2.26b}$$

Undeniably, barycentric coordinates contains redundancy, since there are three components to describe a position on a surface (only two components are needed). However, this is kept for the reason of symmetry.

An example of a triangle and different points inside or on the edge of a triangle can be seen in Figure 2.13. If any of the weights a, b, and c are larger than 1 or negative, then the point P is outside the triangle. If one of the weights are 0 and the two other is equal to 1, then the point is on the edge opposite of the vertex represented by the weight equal to 0. And lastly, if one weight is equal to 1, then the point is the vertex represented by the weight equal to 1.

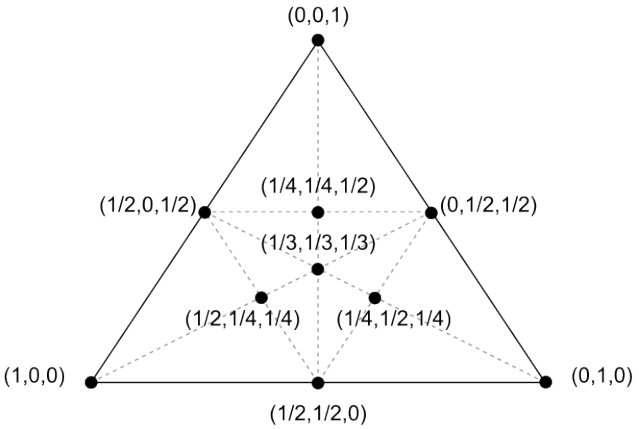


Figure 2.13: A triangle with a variety of its barycentric coordinates printed on.

Chapter 3

Image processing pipeline and recorded dataset

In this chapter a description of the camera equipment, the frames and coordinate systems, and the different steps in the image processing pipeline used to make the IPM images is given.

Parts of this chapter have been taken from my project thesis, with modifications to conform to the change from *milliAmpere1* to *milliAmpere2*. These are Section 3.1, Section 3.2, Section 3.3.2 and Section 3.3.4.

3.1 Equipment

On *milliAmpere2,* there are installed 8 electro-optical cameras, 4 facing front and 4 facing aft. All the cameras are of the type *FLIR Blackfly S 50-S5C*, see Figure 3.1a, equipped with a $6mm$ lens, and capture images in $1224px \times 1024px$ at a rate of $5Hz$. The images are recorded in raw Bayer format. Each camera has a field of view (FOV) of 77.8°, and the covered area can be seen in Figure 3.2. However, due to the cameras' positions and orientations, full near-range coverage is not achieved due to blind zones on the port and starboard sides.

3.2 Frames and coordinate systems

To create the IPM image, multiple reference frames are required. In this approach, two primary frames were utilized, namely the camera frame $\mathcal{F}_c$ and the vessel frame $\mathcal{F}_v$. In addition, two image coordinate frames are needed: The image frame of the camera $\mathcal{I}_c$, and the image frame of the IPM image $\mathcal{I}_{IPM}$. Each camera frame $\mathcal{F}_c$ and image frame $\mathcal{I}_c$ can be specified to each of the 8 individual cameras on the ferry. Their names are, from front port to starboard aft position: fp_p, fp_f, fs_f, fs_s, ap_p, ap_a, as_a and as_s, and the position of the 4 cameras in front are pointed at by red arrows in Figure 3.1b. In the names of the cameras, the "f" and "a" stands for "front" and "aft", while "p" and "s" stands for "port" and "starboard".

(a) An image of the *FLIR Blackfly S 50-S5C* camera used on *milliAmpere2*. Image courtesy of Edmund Optics, Inc [27].

(b) An image of *milliAmpere2* with red arrows pointing at 4 of the camera positions. Photo taken from the thesis proposition web page, hosted by AUTOFERRY [28].

Figure 3.1

3.3 Image Processing Pipeline

The main steps of the image processing pipeline are presented in Figure 3.3. The steps will be further explained in the sections below.

Two notes regarding the image processing pipeline: 1) The images are demosaiced (transformed from raw Bayer format to a full color, normal image format) before the undistortion step. 2) The undistortion method presented below, is relevant for code optimization phase in Chapter 4. During the live experiment, the autonomy system on *milliAmpere2* was responsible for the undistortion. This is illustrated in Figure 3.3 by the box named *External system*.

3.3.1 Distortion and undistortion

To get the undistorted image from each camera, `OpenCV`'s function `undistort()` was used. The distorted image, the intrinsic camera matrix K_i and a vector d_i containing the distortion coefficients for camera i are given as the input to the function. The information was read from calibration files, named *name*`_calib.yaml`, where "*name*" is replaced by the name of each camera. The values of the distortion coefficients used can be seen in Table 3.1. An example of a distorted and undistorted image can be seen in Figure 3.4.

3.3.2 Inverse Perspective Mapping

In practice, the IPM was solved with the function `georeference_point_eq()` which was provided by Øystein Kaarstad Helgesen [7]. It is presumed that the target plane Π_{sea}, has $Z = 0$ in accordance with the assumption made to derive Equa-

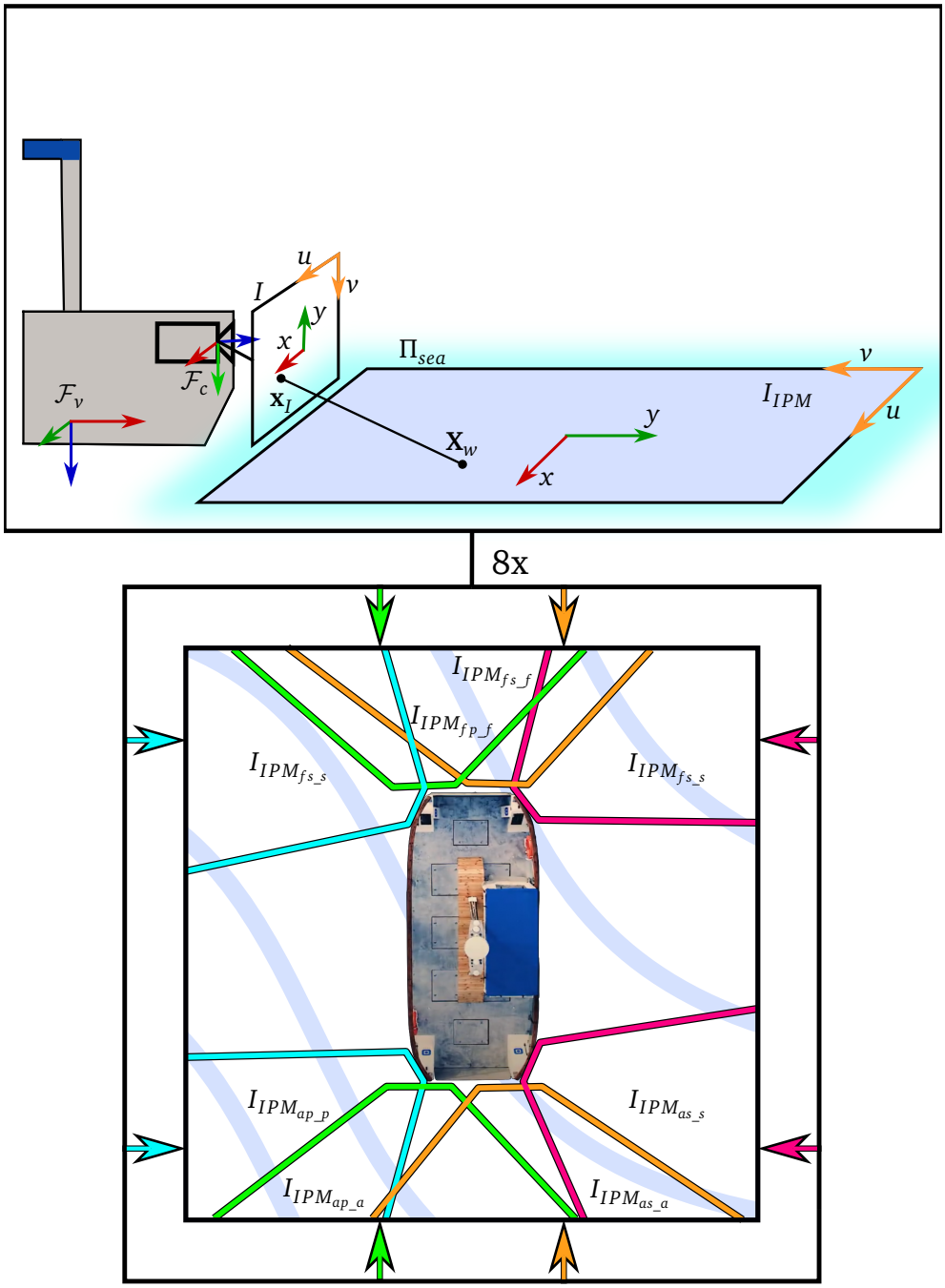


Figure 3.2: Illustration of each coordinate frame and how the bird's eye view is made by combining all 8 camera views. The upper part of the image shows the position of all the coordinate frames. The positions are not exactly as in the real world, but rather meant to convey where the coordinate frames are relative to each other. The bottom part illustrates how the 360-degree bird's eye view is split between the eight cameras, and how much each camera's FOV overlap the neighboring camera's FOV.

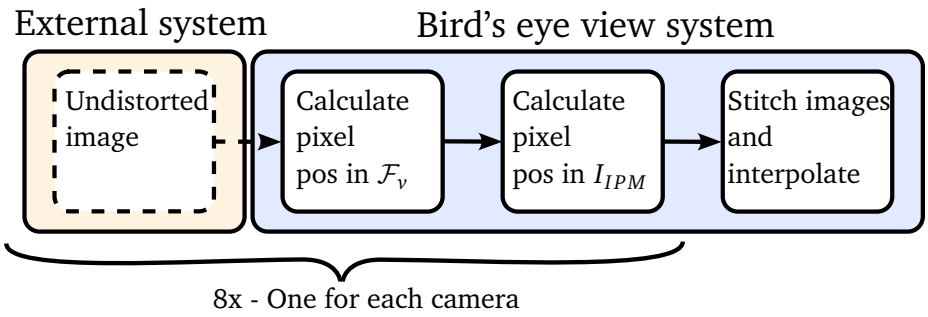


Figure 3.3: The main steps in the image processing pipeline.

(a) **(b)**

Figure 3.4: Showing image with the horizon visible to showcase distortion and undistortion: a) the distorted image; b) the undistorted image. Images are taken with *milliAmpere1*, since *milliAmpere2* did not have any footage of an unobstructed horizion.

tion 2.18. The function uses Equation 2.18 to find the X_v and Y_v. This is done by multiplying the last row of Equation 2.18, $[1 = p_{31}X_v + p_{32}Y_v + t_z]$, by u and v to get Equation 3.1a and 3.1b.

$$u = u(p_{31}X_v + p_{32}Y_v + t_z) \tag{3.1a}$$
$$v = v(p_{31}X_v + p_{32}Y_v + t_z) \tag{3.1b}$$

and substituting Equation 3.1a and Equation 3.1b into the two top rows in Equation 2.18. This gives Equation 3.2.

$$u(p_{31}X_v + p_{32}Y_v + t_z) = p_{11}X_v + p_{12}Y_v + t_x \tag{3.2a}$$
$$v(p_{31}X_v + p_{32}Y_v + t_z) = p_{23}X_v + p_{22}Y_v + t_y \tag{3.2b}$$

Table 3.1: All distortion coefficients vectors d_i used.

Camera name	d_i
fp_p	$[-0.018605, 0.042917, -0.001832, -3.870074e-05]$
fp_f	$[-0.018673, 0.047078, -0.001220, \quad 0.000400]$
fs_f	$[-0.022843, 0.049915, -0.002009, \quad 0.001255]$
fs_s	$[-0.021118, 0.051208, -0.002408, \quad 0.001388]$
ap_p	$[-0.015409, 0.043337, \quad 0.002489, \quad 0.001675]$
ap_a	$[-0.017589, 0.044218, \quad 0.001240, \quad 0.001164]$
as_a	$[-0.019606, 0.052677, \quad 0.001692, \quad 0.0007788]$
as_s	$[-0.012545, 0.038147, \quad 0.001048, -9.807762e-05]$

Rearranging to gather X_v and Y_v on one side:

$$(up_{31} - P_{11})X_v + (up_{32} + p_{12})Y_v = t_x - ut_z \tag{3.3a}$$
$$(vp_{31} - P_{21})X_v + (vp_{32} + p_{22})Y_v = t_y - vt_z \tag{3.3b}$$

For easier calculations, can Equation 3.3 be converted to matrix form:

$$\underbrace{\begin{bmatrix} up_{31} - P_{11} & up_{32} + p_{12} \\ vp_{31} - P_{21} & vp_{32} + p_{22} \end{bmatrix}}_{A} \begin{bmatrix} X_v \\ Y_v \end{bmatrix} = \begin{bmatrix} t_x - ut_z \\ t_y - vt_z \end{bmatrix} \tag{3.4}$$

Which can be solved as long as **A** is invertible.

$$\begin{bmatrix} X_v \\ Y_v \end{bmatrix} = \begin{bmatrix} up_{31} - P_{11} & up_{32} + p_{12} \\ vp_{31} - P_{21} & vp_{32} + p_{22} \end{bmatrix}^{-1} \begin{bmatrix} t_x - ut_z \\ t_y - vt_z \end{bmatrix} \tag{3.5}$$

After calculating the $\mathcal{F}_v$ coordinates for all pixels in the image, a filter removing all points with a distance of outside the range $min_x < X < max_x$ and $min_y < Y < max_y$ is applied. These distances were chosen after experimentation with different distances, and deemed to be the best tradeoff between visual quality and range. The distances that were used are given below.

max_x	$10m$
min_x	$-10m$
max_y	$10m$
min_y	$-10m$

3.3.3 Interpolation

Different methods were used to interpolate the IPM image. This is the step in the image processing pipeline that takes the longest time to complete, and therefore the part of the process that can be improved the most. Two different interpolation

methods from the `SciPy` library will be presented and explained in this section. Both methods are based "traditional interpolation methods": Nearest neighbor and linear interpolation.

Nearest neighbor interpolation

The `scipy.interpolation.NearestNDInterpolator()` (from now on called `NearestNDInterpolator()`) is based upon the method described by [25]. First, a kd-tree is constructed from the data points using the sliding midpoint splitting rule. The kd-space, in this thesis a $2D$ image, is iteratively split into "left" and "right" subspaces. Each subspace is a left or right child node in the kd-tree. If the child node is a leaf node, then it contains the data point in that subspace. If the child node is *not* a leaf node, then the node will contain the splitting axis and the value of where the split of the subspace takes place.

In Figure 3.5, an example of this process is illustrated. On the left is the data points (pixel coordinates) in the image, while on the right side the corresponding kd-tree is constructed. The nodes do not display the splitting axis value for simplification of the illustrations.

To find the nearest neighbor to a new data point, the algorithm first finds the approximately nearest neighbor. After that, it iterates through the tree, measuring if a new node or subspace is closer in distance than the so far closest data point. This continues until the algorithm can conclude that all other data points are further away than the so far closest data point. This is the value of the new data point (in this case, a pixel and its *RGB*-value). An example of the process is visualized and described in Figure 3.6.

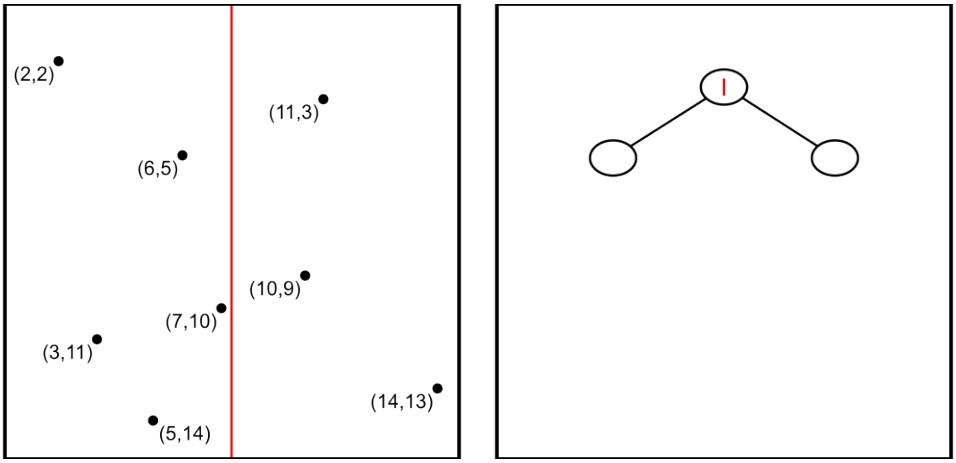
Linear interpolation

There are different types of linear interpolation algorithms. One of the commonly used algorithms are `scipy.interpolate.LinearNDInterpolator()` (from here on called `LinearNDInterpolator()`) [29]. The algorithm can be described in four steps:

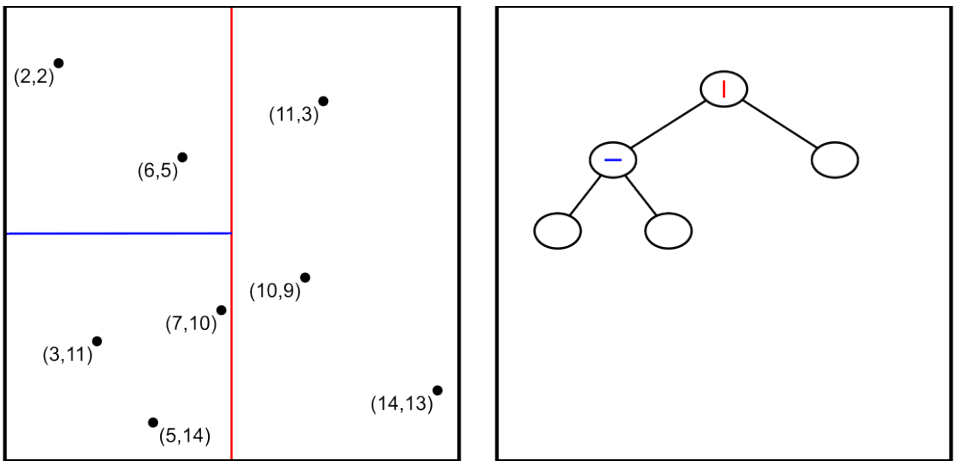
1. Triangulating the irregular input data (pixel positions) using Delaunay triangulation from the `scipy.spatial.qhull`.
2. For every point in the new grid, the triangulation is searched to identify the simplex (a triangle) that encompasses the point.
3. The barycentric coordinates of each new grid point are calculated relative to the vertices of the surrounding simplex.
4. An interpolated value is computed for each grid point using the barycentric coordinates as weights and the RGB-values at the three vertices of the enclosing simplex, performing linear interpolation.

An illustration of step 1-3 is presented in Figure 3.7. Step 4 is calculated with Equation 2.26b.

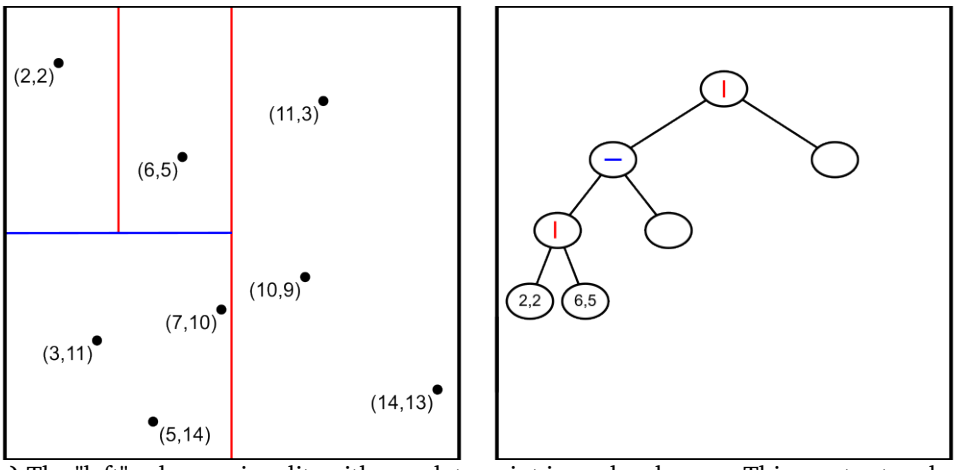
Figure 3.5: The figure illustrates the step-by-step process to make kd-tree from a space containing data points using the sliding midpoint rule.



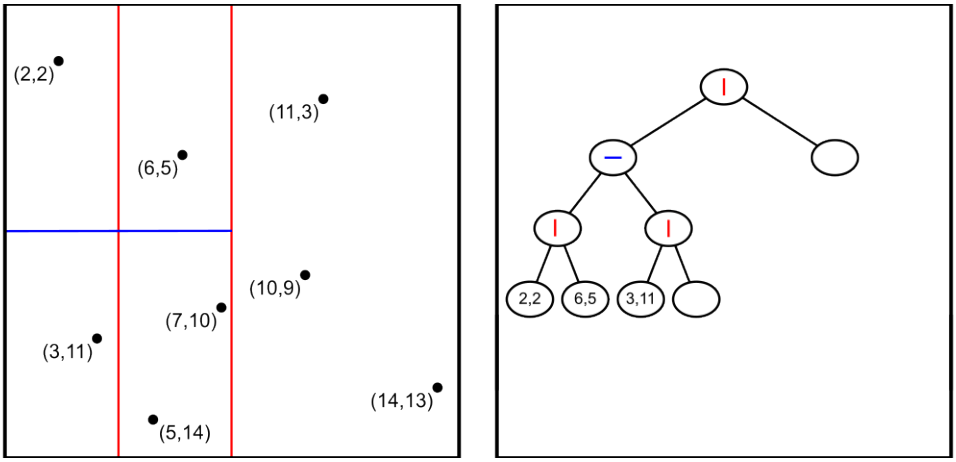
(a) First, the space is divided in the middle along the y-axis. The choice of first splitting axis is chosen randomly. The root node saves the slitting axis and splitting value (only the splitting axis is illustrated in the figure). A child node is made for each new subspace.



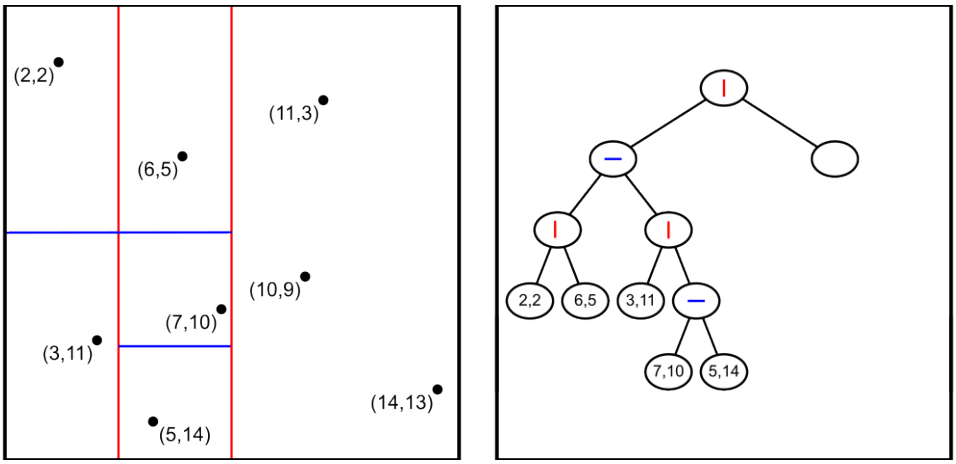
(b) The left node is further split along the x-axis and the splitting axis is saved in the node.



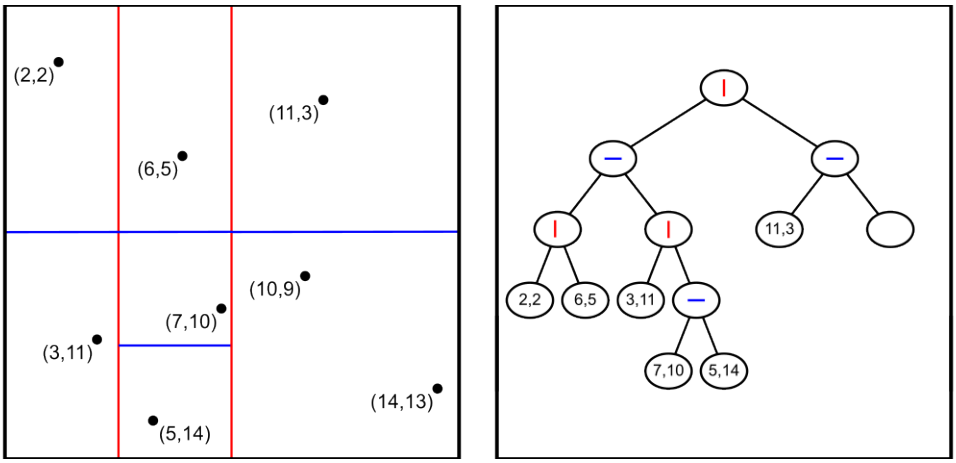
(c) The "left" subspace is split, with one data point in each subspace. This creates two leaf nodes, each with the values of one data point.



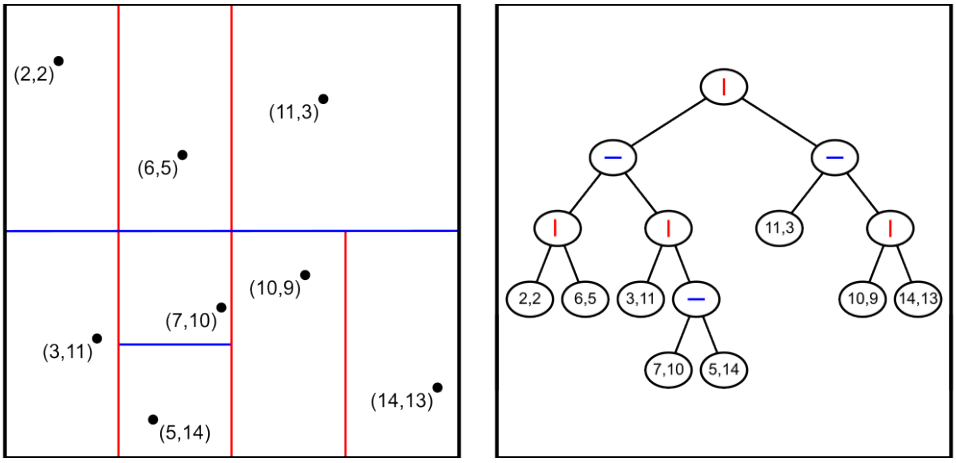
(d) As the last node has two leaf nodes and cannot be further divided, the algorithm ascends to the previous level and continues down the right path of the node. One subspace contains a single data point, which becomes a leaf node, while the other subspace still requires one more division.



(e) The "left" subspace is divided entirely into subspaces, each containing only one data point. The algorithm then returns to the root node and proceeds to iterate through the "right" subspace.



(f) The initial split along the x-axis results in a subspace with a single data point, forming a leaf node as the left child. Meanwhile, the "right" subspace requires further division.



(g) The final subspace, which initially contains multiple data points, is split into two subspaces, each with only one data point. This results in the creation of two leaf nodes, completing the construction of the kd-tree.

3.3.4 Stitching of images

Once the IPM has been applied to the images captured by all 8 cameras, the next step is to stitch them together. For each pixel in each image, the IPM provides the position of the corresponding point $\mathbf{X}_v = [X_v, Y_v, Z_v, 1]^\intercal$ in $\mathcal{F}_v$. These positions can be used to place all the pixels in their correct position in $\mathcal{F}_v$, and later be transformed to one combined IPM image. To make it easier to transform these positions into a image of arbitrary resolution, a normalization of all the points $\mathbf{X}_v$ was performed. Since $\mathbf{X}_v$ has $Z = 0$ on the plane Π_{sea}, the normalization matrix can be written as a 3×3-matrix.

$$\hat{\mathbf{X}}_v = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_v \\ Y_v \\ 1 \end{bmatrix} \tag{3.6}$$

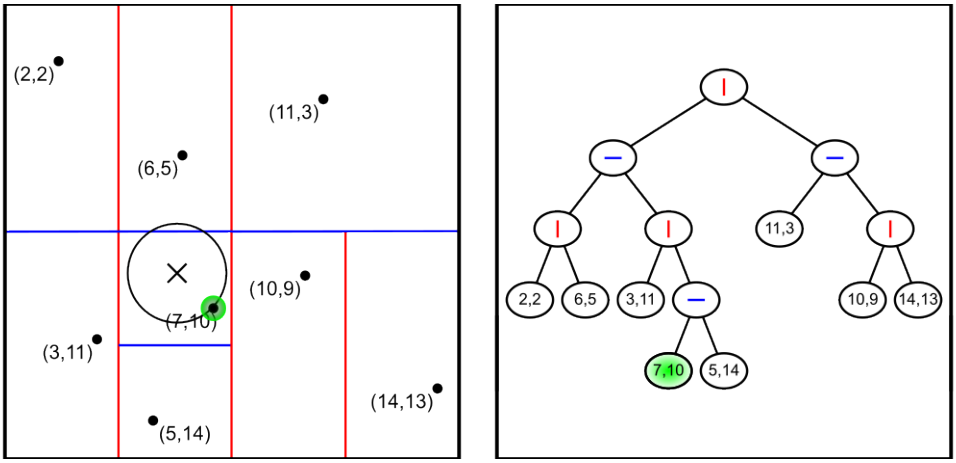
where

$$s_x = \frac{2}{max_x - min_x} \qquad t_x = -s_x min_x - 1$$
$$s_y = \frac{2}{max_y - min_y} \qquad t_y = -s_y min_y - 1$$

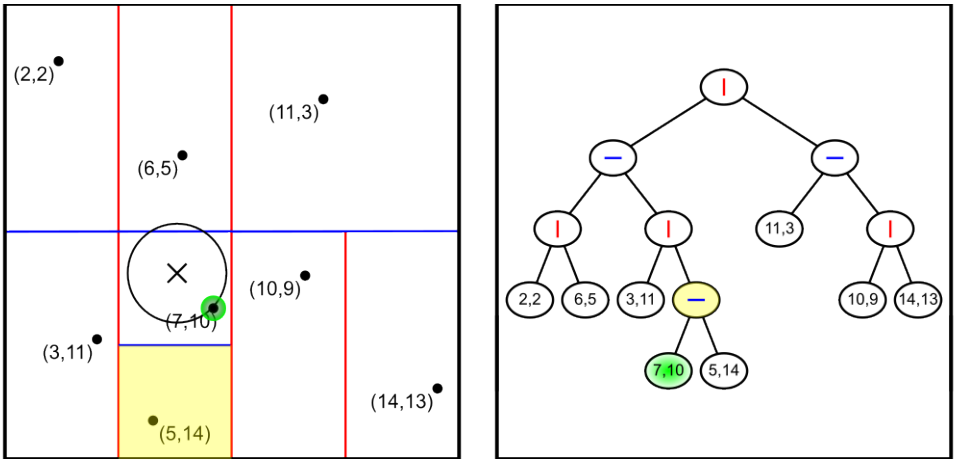
The normalization transforms $\mathbf{X}_v$ from the range $[-10m, 10m]$ to $\hat{\mathbf{X}}_v$ with the range $[-1, 1]$. The final step, to go from $\hat{\mathbf{X}}_v$ to $\mathbf{u}_{IPM}$, was to multiply $\hat{\mathbf{X}}_v$ with the "intrinsic" matrix $\mathbf{K}_{IPM}$, where res_x and res_y is the resolution of the I_{IPM}.

$$\mathbf{u}_{IPM} = \begin{bmatrix} u_{IPM} \\ v_{IPM} \end{bmatrix} = \mathbf{K}_{IPM}\hat{\mathbf{X}}_v = \begin{bmatrix} \frac{res_x}{2} - 1 & 0 & \frac{res_x}{2} \\ 0 & -\frac{res_y}{2} - 1 & \frac{res_y}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{X}_v \\ \hat{Y}_v \\ 1 \end{bmatrix} \tag{3.7}$$

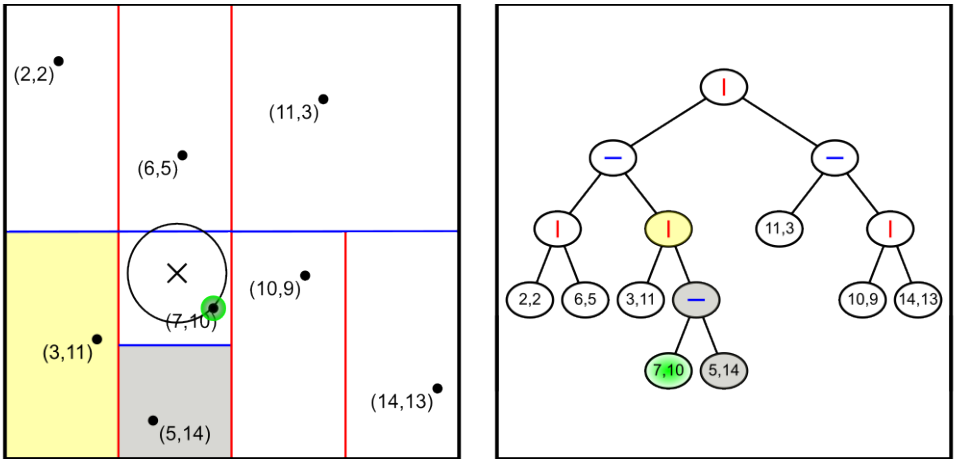
Figure 3.6: The figure illustrates the step-by-step process to find the nearest neighbor to the new data point (marked by an x).



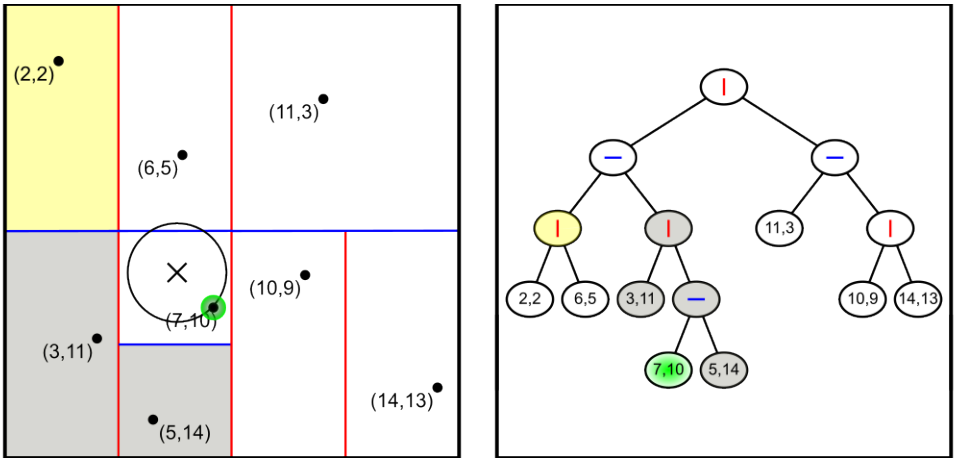
(a) In the first step, the approximately closest data point is found by looking at what data point "owns" the subspace the new data point (marked by an x in the figure) is located in. To illustrate the so far closest neighbor, the data point is colored green. The circle surrounding the new point indicates areas where it is possible to find closer points.



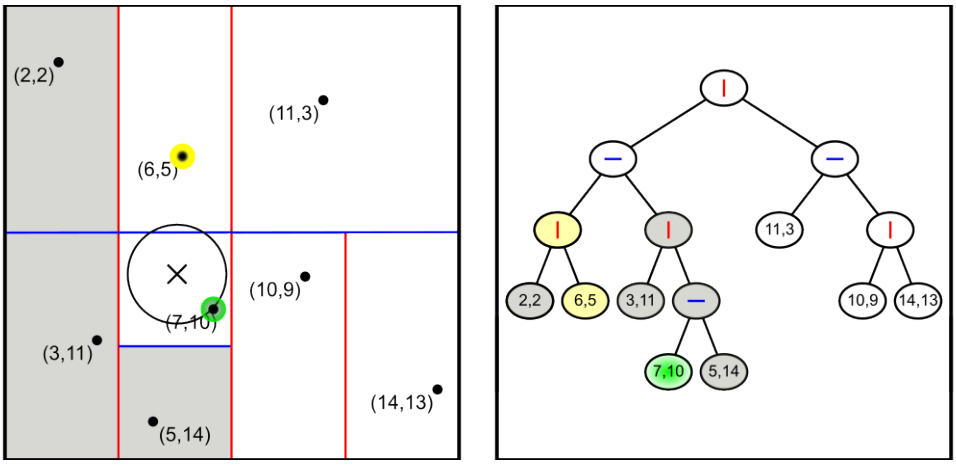
(b) Next, the distance to the so far closest data point is compared to the distance to the axis split value of the parent node. This comparison checks if it is necessary to check all points in the yellow shaded area.



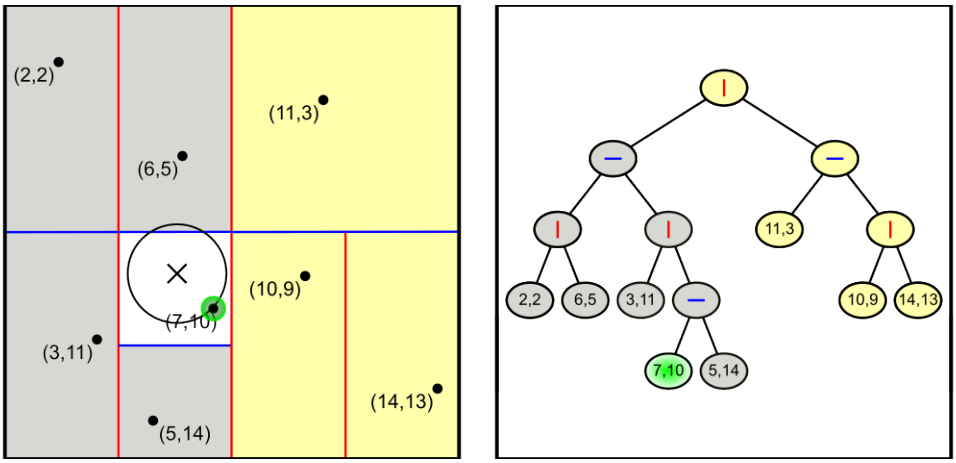
(c) Since the distance to axis split value was larger than that of the so far closest data point, the whole subspace is discarded. This is illustrated by being grayed out. In addition, since the subspace is discarded, there is no need to check the child nodes. The algorithm then continues up one level, and does the same check.



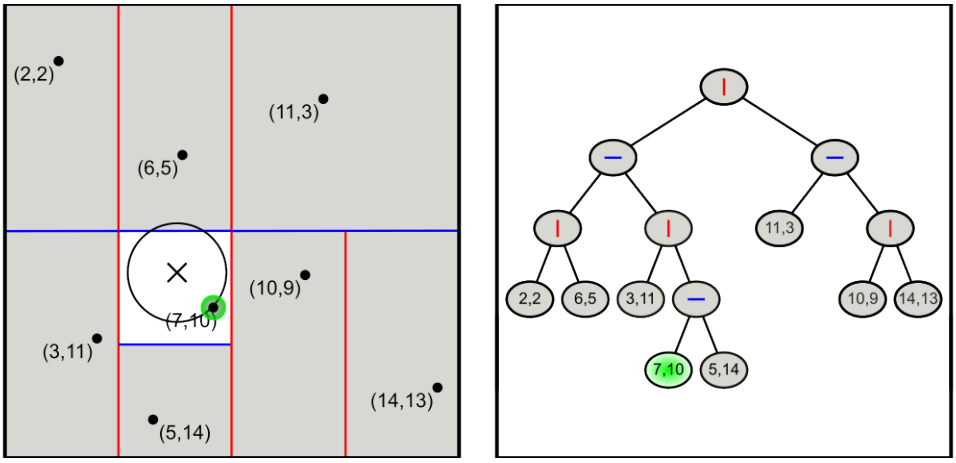
(d) Again, the area is discarded, and the algorithm goes one level up in the kd-tree. The next level node has an axis split value that can produce a data point that is closer than the so far closest data point. Therefore, the algorithm goes down one level in the new branch, and does a new distance check to the subspace shaded in yellow.



(e) Since the distance to axis split value was larger than that of the so far closest data point, the subspace is discarded. The other branch is then checked, and a distance check against the yellow colored data point is done, since it is possible it is the nearest neighbor.



(f) The distance to the data point is further than the so far closest data point, and therefore discarded. The last subspace (the whole right side) are also checked.



(g) Since the distance to the axis split value is larger than the distance to the so far closest data point, the whole right subspace is discarded. The algorithm has found the nearest neighbor, marked in green, to the new data point.

Given the cameras' FOV and position, some of the cameras observe the same area. Due to different viewing angles and local light conditions, this overlap creates unwanted noise in the IPM image, see image Figure 3.8a.

To remove the noise, a three-step process was made. To illustrate the process, the overlapping area between camera fp_f and fs_f is used. Point r_1 is the right most, closest pixel to the center of IPM image, while point r_2 is the rightmost and the furthest pixel from the center of the IPM image. Both r_1 and r_2 are pixels from camera fp_f. Point q_1 and q_2 are the same, but for camera fs_f and most left points instead of right. These 4 points create 2 vectors: **r** and **q**. The intersection of these two vectors can be found by solving $(r_2 - r_1)s + (q_1 - q_2)t = r_1 - q_1$, which can be rewritten to matrix form

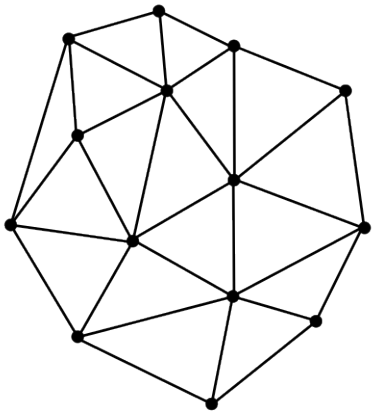
$$\underbrace{\begin{bmatrix} r_2 - r_1 & q_1 - q_2 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} s \\ t \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} r_1 - q_1 \end{bmatrix}}_{\mathbf{b}} \tag{3.8}$$

Equation 3.8 can be solved as a linear system of equations $\mathbf{Ax} = \mathbf{b}$ with `scipy.linalg.solve(A,b)`. The point of intersection between **r** and **q** can be expressed as $i = r_1 + (r_2 - r_1)[s, t]^\intercal$. Apart from the intersection point i, we also need to locate the midpoint m. This midpoint, denoted as m, is the midpoint of the vector $\mathbf{q_2 r_2}$. By connecting i and m with a line, we can form a left triangle L and a right triangle R, as shown in Figure 3.8b.

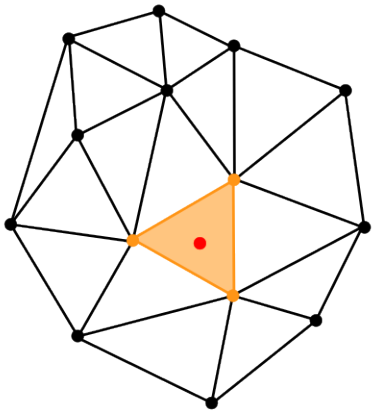
Utilizing these two triangles, we can proceed to eliminate pixels from the camera fp_f within the right triangle and pixels from the camera fs_f within the left triangle. Consequently, all pixels from the right camera that lie on the left side of the vector **im** will be removed, and similarly, pixels from the left camera on the right side of the vector **im** will be removed.

To check if a pixel is inside a triangle, barycentric coordinates were used. The

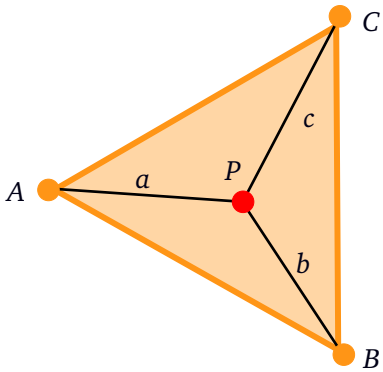
Figure 3.7: The figure illustrates the 3 first steps of `LinearNDInterpolator()`.



(a) Step 1: Triangulating the irregular input data using Delaunay triangulation.



(b) Step 2: Identify the simplex the new red point is located in.



(c) Step 3: Calculate the barycentric coordinates of the point relative to the vertices of the surrounding simplex.

pixel position and the triangle corner positions were transformed into barycentric coordinates, and the value of a, b and c were tested to be in the range $\left[0,1\right]$. Equation 3.9 was used to find a, b and c.

$$a = \frac{(y_2 - y_3)(x - x_3) + (x_3 - x_2)(y - y_3)}{(y_2 - y_3)(x_1 - x_3) + (x_3 - x_2)(y_1 - y_3)} \tag{3.9a}$$

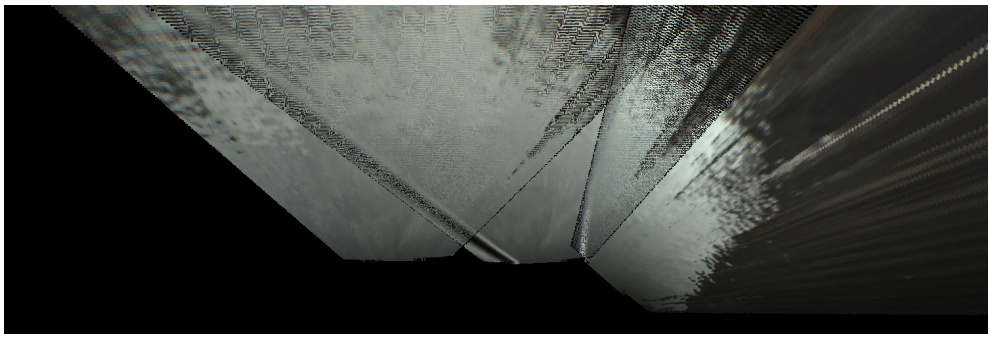
$$b = \frac{(y_3 - y_1)(x - x_3) + (x_1 - x_3)(y - y_3)}{(y_2 - y_3)(x_1 - x_3) + (x_3 - x_2)(y_1 - y_3)} \tag{3.9b}$$

$$c = 1 - a - b \tag{3.9c}$$

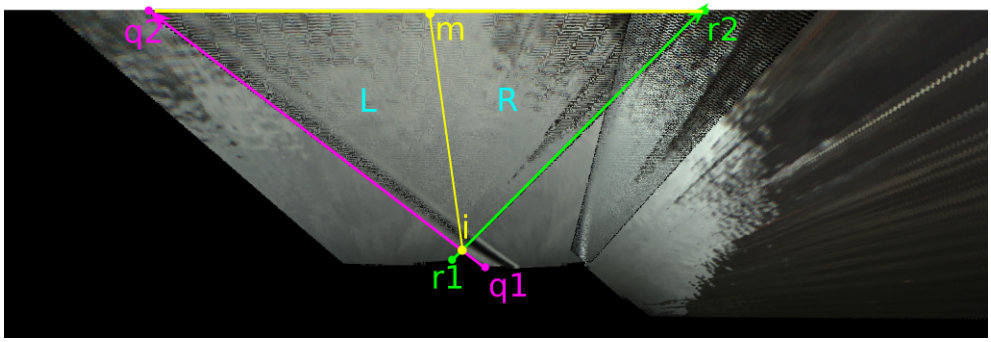
In Equation 3.9, x_1, x_2, x_3 and y_1, y_2, y_3 are the x- and y-values of the 3 corners of the triangle, while x and y represent the pixel. If the pixel were inside the triangle, it was removed from the IPM image.

The coordinates of points r_2 and q_2 depicted in Figure 3.8b do not correspond to their actual positions. In reality, these points were located far beyond the borders of the IPM image, yet aligned with the line connecting the illustrated points. This discrepancy arose because the distance limit was not applied to the pixels at this point in the process. The result of the removal of overlapping camera FOVs can be seen in Figure 3.8c.

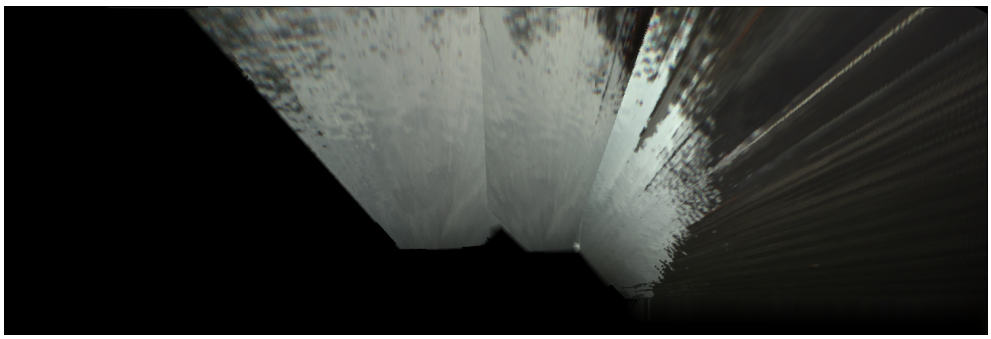
Figure 3.8: Showing the effect of removing the overlapping FOVs between front facing cameras. a) shows the unprocessed IPM image, while c) shows the same IPM image after camera overlapping areas has been resolved. b illustrates the position of the objects in the method described above. IPM image is a cropped version of frame 1 from `rosbag2_2022_10_09-08_02_54_80.db3`.



(a) A cropped IPM image showing the overlapping camera FOVs, noticeable by the presence of visual noise along the FOV edges.



(b) An illustration of where each part in the method to remove the overlap between cameras are positioned. Illustration is an example for the overlap between camera fp_f and fs_f.



(c) IPM image without overlapping camera FOVs, and therefore less visual noise.

Chapter 4

Code optimization

The final result of my project thesis was a video showing the bird's eye view of *milliAmpere1*. The code was not designed with any thought of efficiency; only to prove that the underlying method and technology was possible to use on the camera setup of *milliAmpere1*. However, for this thesis, one of the goals were to be able to produce the top-down view in real-time on *milliAmpere2*. As a result, the project thesis code needed to be adopted from ROS1 to ROS2, and made significantly faster. The change from ROS1 to ROS2 was necessary, since *milliAmpere2* use ROS2. Furthermore, the code also had to be modified to process images from 8 cameras, instead of the 5 cameras on *milliAmpere1*. Given that the cameras capture images at a rate of $5Hz$, a time of less than $200ms$ per IPM image was set as a goal.

This chapter is a complete description of the code optimization process. It includes the different methods and improvements to the code, their respective run-times, and a discussion around the results and observations.

4.1 Code optimization setup

For this thesis, the same computer as for the project thesis was used. The computer was set up with `Ubuntu 20.04` and an Intel Core i7-8700 CPU with a base clock frequency of 3.20GHz, max clock frequency of 4.60GHz, 6 cores and 12 threads. It was running `Python version 3.8.10`, `NumPy version 1.23.2`, `SciPy version 1.9.1` and `ROS 2 Foxy Fitzroy`. All image results made during the code optimization were made from the two files presented in Table 4.1.

The run-time data presented in section 4.2 were captured through running the `main()` function with `cProfiler`, a profiler library that measures elapsed time during all function calls. The reported run-times were the average run-time of the function `make_BEW()`, which contain the interpolation function that was used to produce the IPM image. The timer started after the `make_BEW()`-function had received all 8 undistorted images, and ended when the IPM image was made. It is important to note, that the presented run-time does not include reading the `ros`-messages, transforming the images from raw Bayer format to *OpenCV* format and

Table 4.1: Name and description of the files used in Chapter 4.

Filename	Information about the file
File 1: rosbag2_2022_10_06-13_50_07_0.db3	Initial file provided by advisor Øystein Kaarstad Helgesen. Contained images from 7 cameras. Images from camera fp_p is missing.
File 2: rosbag2_2022_10_09-08_02_54_80.db3	File provided by advisor Øystein Kaarstad Helgesen. Contained images from all 8 cameras.

undistorting the images. These functions were excluded from the measurement, because they were either handled by the autonomy system on *milliAmpere2* or the functions were not present in the version of the system running on the ferry. The initial run-time data were from the construction of 10 IPM images.

4.2 Optimization steps and results

The initial performance benchmark for this thesis was based on the runtime of the code from my project thesis, but used with ROS2. The `make_BEW()`-function was using `scipy.interpolation.griddata()`(from here on `griddata()`) with bilinear interpolation on $3000px \times 3000px$, and the run-time was $34s$. As mentioned earlier, this program was not written with efficiency in mind.

4.2.1 Exploration of interpolation methods

From this initial performance, different options regarding interpolation methods and a reduction of resolution were tested. The resolution was set to $1500px \times 1500px$ and using `griddata()` with *nearest neighbor* produced a run-time of $13.57s$. An important observation regarding the number of pixels that needed to be interpolated was made at this stage. In Figure 4.1 there is a large black area in the middle that is void of pixel data. All these pixels are supposed to be black, since there is no camera coverage here. However, since the pixels were empty instead of black, the interpolation function calculated a new RGB-value for all the empty pixels. This situation was as severe in my project thesis, since most of the IPM image were covered by the 5 cameras, see Figure 4.3.

Due to this discovery, an image with black pixels that filled the void area were added to the data given to `griddata()`. The image was made from a manually created mask, see the red area in Figure 4.2. With the same resolution as before and using `griddata()` with *nearest neighbor*, a significant improvement in run-time was achieved: $3.226s$.

After the prior improvement, a deep dive into how `griddata()` work behind the scenes was conducted. `griddata()` is only a wrapper-function for different `scipy.interpolation`-methods. In addition, the function can take a pre-calculated

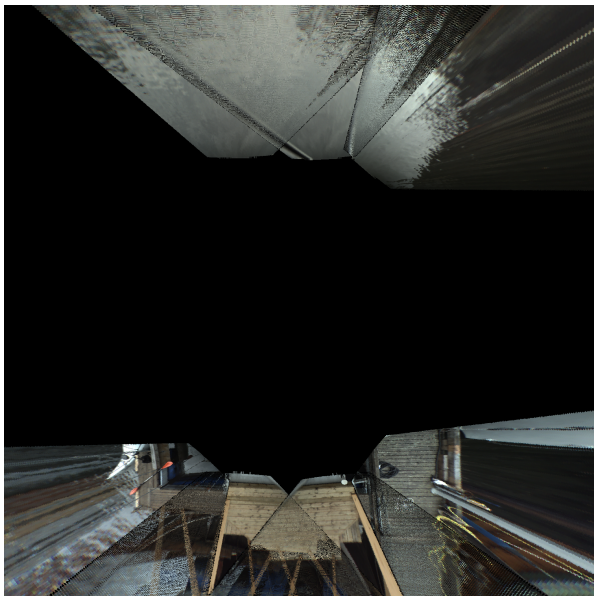


Figure 4.1: A bird's eye view image from the initial `ros`-file, and therefore missing camera fp_p. Made with `griddata()` and a resolution of $1500px \times 1500px$.

Figure 4.2: Showing the red mask used to make the black image to fill the void area in Figure 4.1.

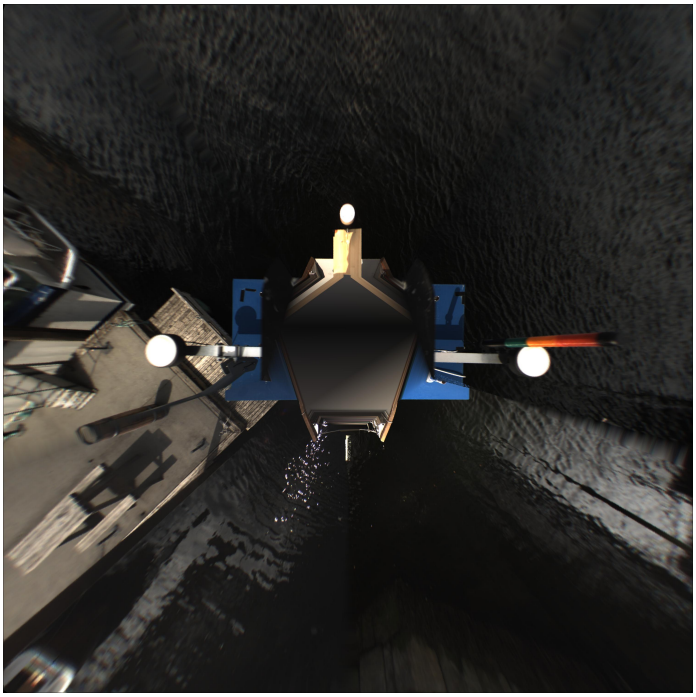


Figure 4.3: A bird's eye view image from my project thesis made by using `griddata()` with a resolution of $3000px \times 3000px$.

`scipy.spatial.Delaunay`-object, instead of the pixel positions. The pre-calculation does not affect the result, since one of the first steps of `griddata()` is to calculate the Delaunay triangulation. Furthermore, since the camera parameters are constant after the program initialization, all pixel positions from the IPM calculations are fixed as well.

With these three discoveries, pre-calculating the Delaunay triangulation of the pixel position and using `NearestNDInterpolator()` with a resolution of $1500px \times 1500px$ and the filled black area, a new time of $3.132s$ was achieved. The `scipy`-function `NearestNDInterpolator()` is the same function `griddata()` calls upon when the interpolation method is set to *nearest neighbor*.

To make the previous method, using `NearestNDInterpolator()`, work as desired, the interpolation step had to be split into one function call for each RGB-color. These 3 function calls were in the previous method, run in succession. To improve this, multiprocessing using the `python` library `multiprocessing` was tested. The attempt was to run the 3 function calls of `NearestNDInterpolator()` in parallel. The run-time was $5.801s$, which was higher than that of the serial method.

In the approach of utilizing `NearestNDInterpolator()` in succession, only the Delaunay triangulation is calculated beforehand in the initialization. However, the interpolation weights for each pixel would be possible to calculate once in the initialization, since the positions of the pixels in the IPM image are fixed. Only the RBG-values change from one IPM image to the next. During the design phase, the

approach of `LinearNDInterpolator()` from Section 3.3.3 was used as inspiration. The steps are repeated below.

1. Triangulating the irregular input data (pixel positions) using Delaunay triangulation from the `scipy.spatial.qhull`.
2. For every point in the new grid, the triangulation is searched to identify the simplex (a triangle) that encompasses the point.
3. The barycentric coordinates of each new grid point are calculated relative to the vertices of the surrounding simplex.
4. An interpolated value is computed for each grid point using the barycentric coordinates as weights and the RGB-values at the three vertices of the enclosing simplex, performing linear interpolation.

The first three steps can be completed during the initialization, leaving only the last step to be calculated during the execution. This method resulted in a run-time of 617.1*ms*, and a significant improvement towards the goal of 200*ms*.

A summary of the different methods and their run-times are presented in Figure 4.4 and Table 4.2.

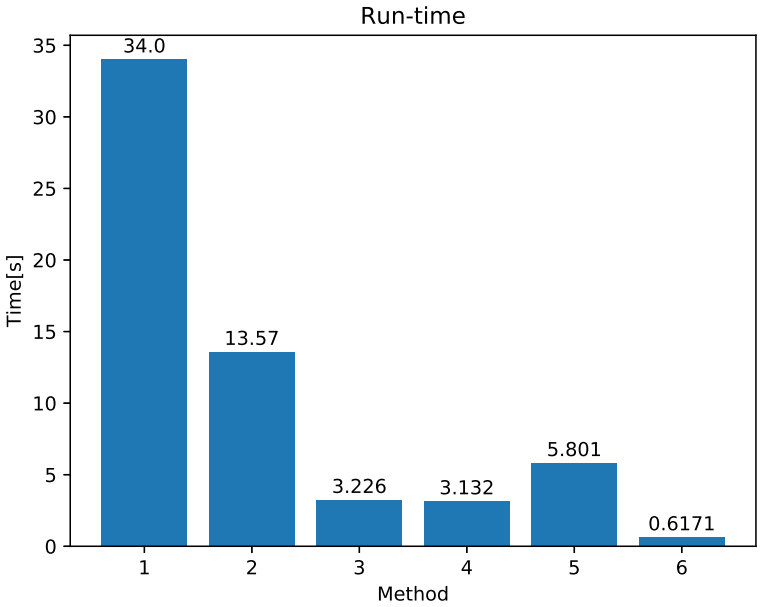


Figure 4.4: A summary of the different methods tested in Section 4.2.1. Description of each method can be found in Table 4.2.

4.2.2 Improving performance through problem-solving

After achieving a run-time of 617.1*ms*, down from the initial run-time of 34*s*, a major restructure of the code was undertaken. Two major problems were solved:

Table 4.2: Description of the different interpolation methods shown in Figure 4.4.

Method	Run-time	Description
1	34.0s	`griddata()` with linear interpolation and a resolution of $3000px \times 3000px$.
2	13.57s	`griddata()` with nearest neighbor interpolation and a resolution of $1500px \times 1500px$.
3	3.226s	`griddata()` with nearest neighbor interpolation, the empty, black area in the middle of the image filled with black pixels, and a resolution of $1500px \times 1500px$.
4	3.132s	`NearestNDInterpolator()` with the empty, black area in the middle of the image filled with black pixels, pre-calculated Delaunay triangulation, and a resolution of $1500px \times 1500px$. RGB-values interpolated in succession.
5	5.801s	`NearestNDInterpolator()` with the empty, black area in the middle of the image filled with black pixels, pre-calculated Delaunay triangulation, and a resolution of $1500px \times 1500px$. RGB-values interpolated in parallel with `multiprocessing` library.
6	0.6171s	Pre-calculated interpolation weights with linear interpolation, the empty, black area in the middle of the image filled with black pixels, and a resolution of $1500px \times 1500px$.

1) Due to the erroneous installation of the cameras on *milliAmpere2*, part of the images were covered up by the hull surrounding the cameras, see Figure 4.5a. This created visual artifacts/noise in the IPM image. The visual artifacts were especially prominent in areas with overlapping camera FOVs. This was because these areas were mainly covered by the edges of the images, where the hull was visible. The problem was solved by manually creating masks for each camera that removed all parts of the image containing walls, see Figure 4.5b.

2) In my project thesis, an oversight occurred regarding the omission of a distance check for pixels exceeding the desired maximum distance. Although this omission did not produce any visible errors due to `Python`'s image handling capabilities, it led to the inclusion of pixels outside the intended range in the interpolation function. As a result, the run-time was negatively affected due to additional calculations. To resolve this, a solution was implemented by performing a distance check for each pixel's position in the x- and y-direction, removing any pixels outside the specified range. The outcome of this check resulted in an additional mask that was later merged with the existing wall-mask.

Finally, by relocating the wall-mask generation, the distance check, and combining of the two masks to the initialization stage of the code, the run-time was

reduced to 338.8*ms*.

(a) **(b)**

Figure 4.5: Showing an arbitrary image from camera fs_f. In a) the hull surrounding the camera is visible on the left side. The mask to remove the hull from the IPM image is the area covered by red in b).

The next improvement of the code, was to add a function to address the areas of the IPM covered by two cameras, as described in Section 3.3.4. This improvement should not have a significant impact on the run-time, since it only removes a few pixels from being interpolated. The function is more important for the visual quality of the product. The new run-time was 320.7*ms*.

4.2.3 Expanding code from 7 to 8 cameras

In order to improve the code's functionality and prepare it for testing on *milliAmpere2*, an expansion was incorporated to accommodate images from all 8 cameras. The testing of this expansion was made possible by the availability of a new `rosbag`-file, see *File 2* in Section 4.1, which contained images captured by all cameras. Combined with the previous improvements, a new run-time of 355.8*ms* was achieved. An IPM image with all 8 cameras and correct handling of overlapping areas can be seen in Figure 4.6. The mask for filling the void area in the middle was also updated, see Figure 4.7. The increase in run-time was expected, since 1 more image had to be processed in the pipeline.

4.2.4 Final performance tuning

The run-time at this point in time was closing in on the goal of less than 200*ms* per IPM image. Improvements would therefore be in milliseconds, and small run-time variances per IPM image could affect the average run-time results in a greater degree. To be more certain the average run-time was accurate and representative, the number of IPM images the run-time data was collected from was increased from 10 to 100 images. On top of that, an extra, stand-alone test code was made

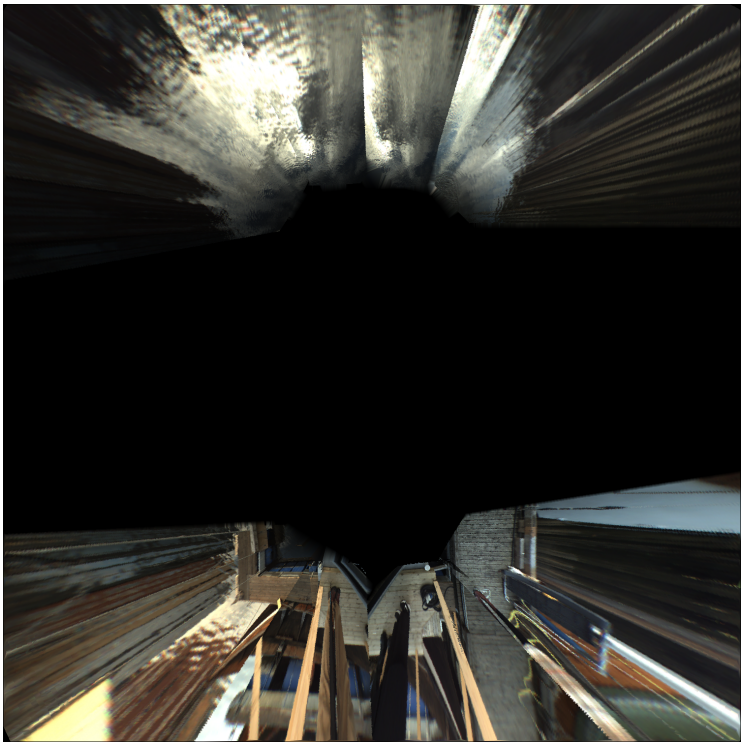


Figure 4.6: Image showing an arbitrary frame with the pier from *File 2*. All 8 cameras included. Overlapping camera FOVs are resolved.

for each function change. The extra test code was made to be as similar as possible to the main code, having the same input and the same output. The run-times from these test were collected from 500 function calls, and served as an indicator of the reasonability of the performance gain in the main code.

To find where potential improvements could be made, the profiler-file made by `cProfile` was investigated with `snakeviz`. Functions with a high cumulative run-time were investigated first, since the potential for large improvements were greater.

The first discovery was found in the function `interpolate()`. It used `numpy.take()`, which is easier to read and use than `NumPy`'s advanced array indexing (AAI), but much slower. From the stand-alone test, the change from `numpy.take()` to AAI showed a promising decrease of 15.8*ms* per function call. Since `interpolate()` contains one `numpy.take()`, and the function is called 3 times, the test indicated a decrease of around 47.4*ms*. After implementing the AAI instead of `numpy.take()`, the new run-time was 292.1*ms*. A decrease of 63.7*ms*, and better result than expected from the stand-alone test result.

With the new knowledge, that `numpy.take()` was slower than AAI, other `NumPy`-indexing functions were searched for. In the function `calculate_rgb_matrix_for_BEW()`, a `numpy.all()` was used. From the stand-alone test, a change from `numpy.all()` to AII resulted in a reduction of 16.61*ms* per function call. Since `calculate_rgb_-`

Figure 4.7: Image showing the new, updated red mask to create the black pixels to fill the void in the middle of the IPM image.

`matrix_for_BEW()` contains one `numpy.all()`, and is called 8 times during the IPM image production, the result from the external test indicated a decrease of 132.9*ms*. In the main program, the change resulted in a run-time decrease of 91.1*ms*, and a new run-time of 201.0*ms*. While the improvement was not as large as expected compare to the result of the stand-alone test, it still was a significant improvement in performance.

The last necessary improvement, to reach the goal of less than 200*ms* per IPM image, was to change all `numpy.flatten()` in the function `make_BEW()` to `numpy.ravel()`. Both functions collapse a `numpy`-array of N-dimensions into a one-dimensional array. The difference is that `numpy.flatten()` makes a copy of the given array and returns a flatten *copy* of it, while `numpy.ravel()` returns a flatten *view* of the given array. Since `numpy.flatten()` needs to allocate memory for the copy, it is slower than `numpy.ravel()`. From the stand-alone test, this difference was 3.96*ms* per function call. A total of 5 `numpy.flatten()` was called in `make_BEW()`, indicating a run-time improvement of 19.8*ms*. After implementation of `numpy.ravel()`, the run-time was decreased by 21.3*ms* to 179.7*ms*, and below the goal of 200*ms*.

An additional test was conducted to test how low the run-time could go while maintaining a reasonable resolution. A resolution of $1100px \times 1100px$ was tested, and the average run-time was 130.0*ms*.

A summary of all the improvement steps and their run-time can be seen in Figure 4.8. Since there were no visual improvements in the last code optimization part, the final image result can be viewed in Figure 4.6.

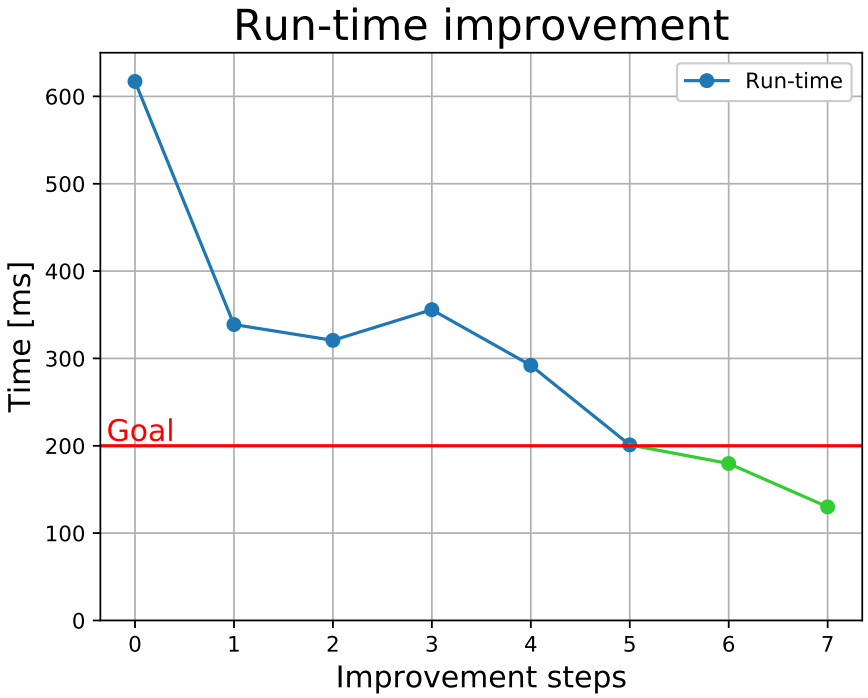


Figure 4.8: A summary of all the improvements, illustrated in a line chart, after method 6 from Section 4.2.1 was developed. A description of each improvement step and the respective run-time can be found in Table 4.3. Resolution was $1500px \times 1500px$ if not specified.

Table 4.3: The run-time and description of each improvement step presented in Figure 4.8.

Step	Run-time	Description
0	617.1*ms*	The best method (number 6) in Section 4.2.1 and the baseline for this part of the code optimization.
1	338.8*ms*	Wall-mask generation, distance check, and the generation of the final image-masks were moved to the initialization of the program.
2	320.7*ms*	Added function to fix overlapping FOVs in the IPM image.
3	355.8*ms*	Adopted code from 7 to 8 cameras.
4	292.1*ms*	Replaced `numpy.take()` with AAI in the `interpolate()` function.
5	201.0*ms*	Replaced `numpy.all()` with AAI in the `calculate_rgb_matrix_for_BEW()` function.
6	179.7*ms*	Replaced `numpy.flatten()` with `numpy.ravel()` in the `make_BEW()` function.
7	130.0*ms*	An extra test with the resolution lowered to $1100px \times 1100px$.

Chapter 5

Experimental setup

In this chapter, a description of the experimental setup, including the operational area, technical specifications of the ferry and its onboard systems will be given. Furthermore, the necessary code adaptations required to execute the code on *milliAmpere2* will also be presented.

5.1 Area of operation

The experiment was conducted on the 10^{th} of May 2023. It took place in Trondheim, Norway, in the canal between Fosenkaia and Ravnkloa as reflected in the red area in Figure 5.1. The black and blue path in Figure 5.1b is the approximate path driven by *milliAmpere2* during the experiment.

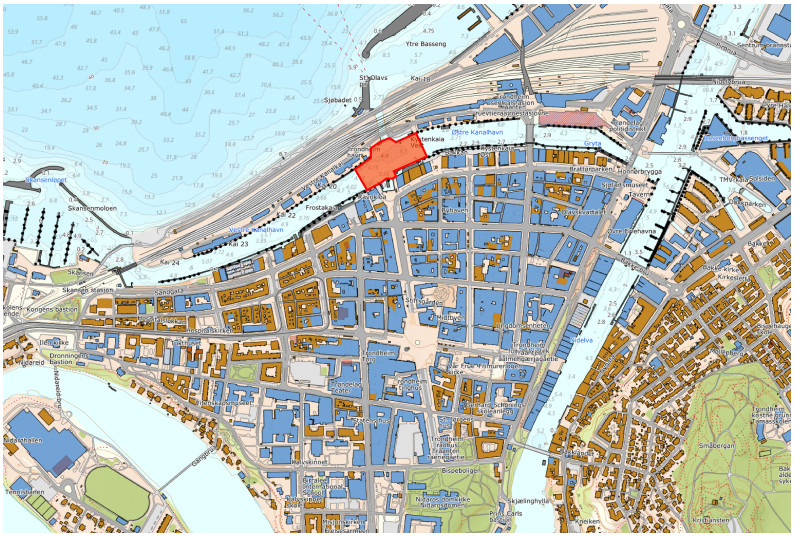
5.2 Technical specifications

The ferry *milliAmpere2* was used to capture images, and all 8 *FLIR Blackfly S 50-S5C* optical cameras were used. To run the program on the ferry, the computer owned by Zeabuz was used. The computer was set up with an AMD EPYC 7313P "MILAN" CPU with a base clock frequency of 3.0GHz, max clock frequency of 3.7GHz, 16 cores and 32 threads [30]. The code was in running in a *Docker* container on the computer using `Ubuntu 22.04` and `ROS 2 Humble Hawksbill`. Unfortunately, the `Python`, `NumPy` and `SciPy` versions were not written down, but following the required versions for `ROS 2 Humble` on `Ubuntu 22.04`, `Python version 3.10.13`, `NumPy version 1.21.5` and `SciPy version 1.10.1` can be assumed was used. The general specifications of *milliAmpere2* can be found in Table 5.1.

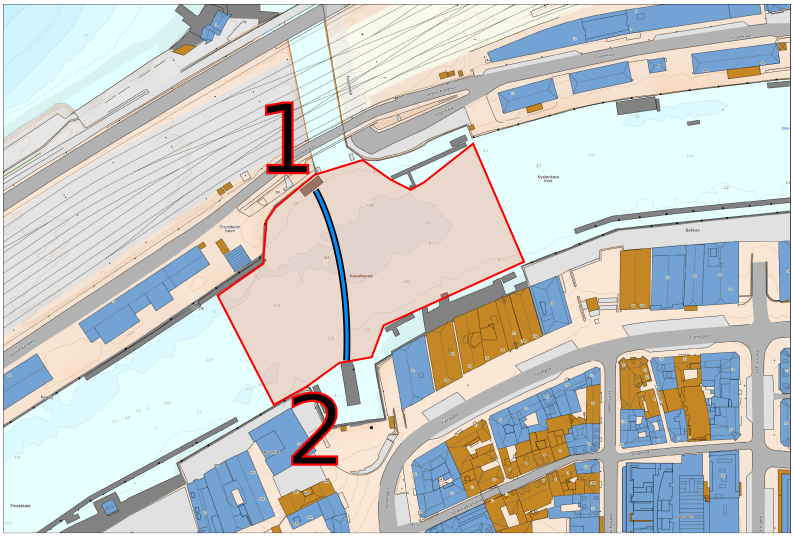
5.3 Plan of action

The ferry was operated locally by Egil Eide (project leader and builder of *milliAmpere2*) and Øystein Kaarstad Helgesen. In addition to the recording of the IPM images, a GoPro camera was set up inside the cockpit of the ferry. It recorded the

Figure 5.1: Map data is provided by the Norwegian Mapping Authority (©Kartverket).



(a) Map centered on Trondheim city center. The main experimental area used in this thesis is marked by the red polygon.



(b) A zoomed in view of the area of operation. The black and blue path is the approximate path taken by *milliAmpere2* during the experiment. Fosenkaia is marked by **1**, while Ravnkloa is marked by **2**.

Table 5.1: General specifications for *milliAmpere2* [2].

Specification	***milliAmpere2***
Length	$8.5m$
Beam	$3.5m$
Draft	$0.3m$
Max pax	12
Propulsion	4 azimuth thrusters
Operation speed	3 knots
Max speed	5 knots
Energy	Electric $48V$ DC
Batteries	Lead-Acid VR $48kWh$
Power	$4 \times 10kW$
Sensors	IR camera, camera, RADAR, LIDAR, ultrasonic

steering console and the operator. Lastly, a drone with a camera was operated by Marcus Lerfald (a fellow Master student at NTNU. The drone footage captured the ferry movements from the outside. Images from the different point of views will be presented later.

The plan for the experiment was to cross the canal two times, with a total of four dockings. The experiment had two main goals: To verify the real-time performance of the system on *milliAmpere2* while the onboard autonomy system was running, and to evaluate the system's usefulness for the operator, specifically during docking operations. The IPM images were recorded using the built-in recording in ROS2, `ros2 bag record`.

5.4 Code adoption to ROS2

In order to run the code on *milliAmpere2*, the code used in Chapter 4 was adopted to the ROS2 node-structure. Each *node* in ROS2 has one purpose, for example, undistorting images from cameras or making an IPM image. The nodes communicate over *topics*, where *messages* can be sent. A message could contain for example an image or text. A node can send messages through its *publisher* service and receive through its *subscribers* service.

To be able to communicate and function together with the autonomy system running on *milliAmpere2*, the code was adopted to run as a node called `BEW_maker` with the task to convert 8 undistorted images to one IPM image. The node was subscribed to all 8 camera topics, where the node received undistorted images. When an image from all cameras were received, the node ran the `make_BEW()`-function (the same function as used and timed in Chapter 4) to make the bird's eye view image, before it published the image on the topic *im_BEW*. An overview over the node and the topics used in the system can be seen in Figure 5.2.

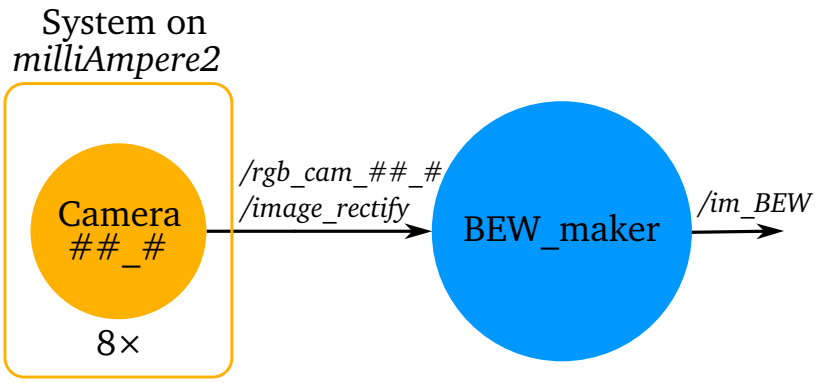


Figure 5.2: Illustration of the relevant nodes and topics used during the experiment on *milliAmpere2*. The blue circle, BEW_maker, is the node the bird's eye view images are made in. The images are published on the topic */im_BEW*. The orange circle represent the node for each camera, where *##_#* are replaced by camera name. On *milliAmpere2* there are 8 cameras, which are represented by the 8× below the node. From the camera nodes, the undistorted images are sent on the topics */rbg_cam_##_#/image_rectify*, which the BEW_maker-node are subscribed to. As for the node name, *##_#* are replaced by camera name.

In addition to adopting the code to a ROS2 node, a few new functions were added. Firstly, a logging-function was added. This function writes to a log-file with the setup: `camera_name;time_since_last_image;message`. Each time an image is received or published on a topic, it logs the camera name and time (in *milliseconds*) since the last image was received or published. The "`message`"-part is only used if the node detects there is a delayed camera, which will be explained below. For example, when an image is received from camera ap_p, it will log `rgb_cam_ap_a;240.916611;`.

Secondly, the code now continuously tracks how long it is since it last received an image from one of the 8 cameras. If the time since last image passes a set time limit, it will be written in the log: `rgb_cam_fp_p;0;Delayed. Using last image`. This time limit has to be set manually in the configuration file. When the time limit is passed, it will use the last image from that camera until it receives a new image. When it receives a new image from the delayed camera, it will, as normally, write the time since the last image was received `rgb_cam_as_s;33337.420608;`. This fail-safe system was implemented to increase the fault tolerance of the system in instances where one or more cameras fail to send images, while the others continue to work as intended.

Lastly, a bird's eye view image of *milliAmpere2* was placed in the center of the IPM image, see Figure 5.3. The position in the IPM image is manually set before code initialization, and the image of the ferry is added to all IPM images during the processing pipeline.

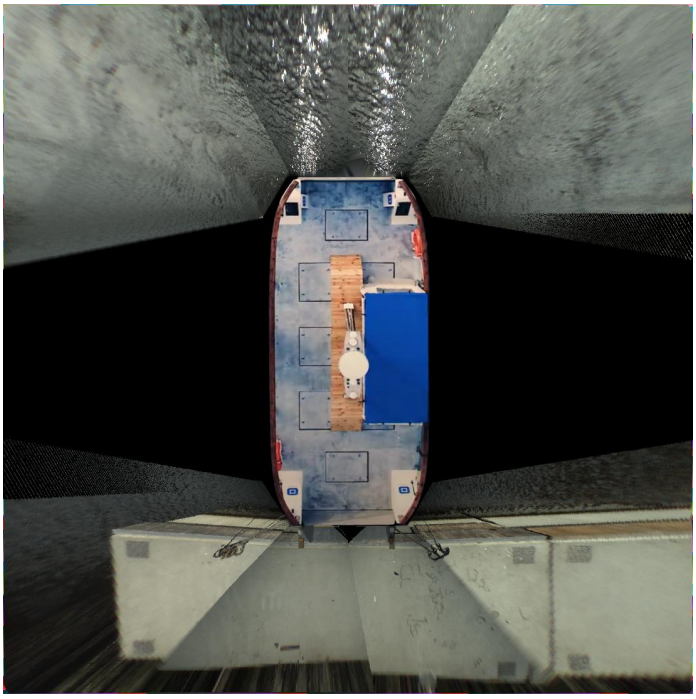


Figure 5.3: An image showing an IPM image with a cutout of *milliAmpere2* seen from a bird's eye view, added to the center of the image. The image of the ferry is manually positioned in the IPM image before code initialization, and the image of the ferry is added to the IPM images during the processing pipeline.

Chapter 6

Results

In this chapter, the results of the conducted experiment will be presented. The chapter includes a presentation of both the image results and the run-time results. The experiment was carried out following the proposed plan outlined in Chapter 5.

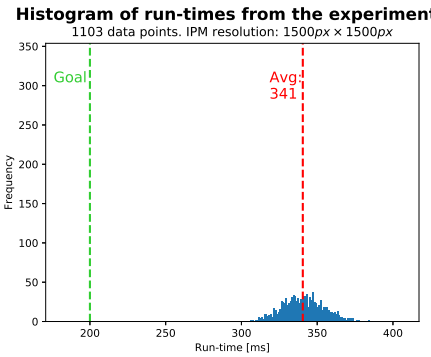
6.1 Run-time results

During the experiment, the log-function described in Section 5.4 was used to capture the run-time between each creation and publishing of the IPM image. Due to some startup errors in the communication system on the ferry during the first docking, a few starts and restarts of the program were needed. This created several log-files with a few different average run-times and distributions. However, the last log-file that was made contained the most run-times by a significant margin (a few of hundred vs 14752 data points) and therefore is the one used to represent the run-time of the program.

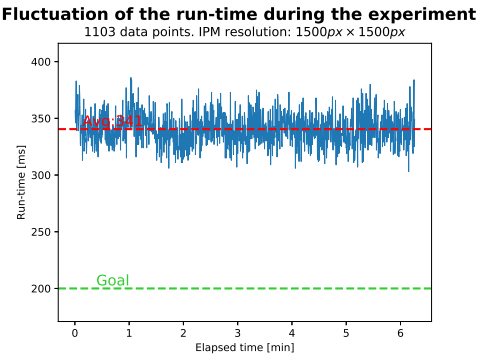
The run-times presented in this section is the run-time between each publishing of the IPM images on the topic *im_BEW*. This includes the `make_BEW()`-function, conversion between ROS2 messages and OpenCV image format and any other overhead in the system. Note, the final runtime presented in Chapter 4 was the run-time of only the `make_BEW()`-function. During the startup of the experiment, it was quickly observed that a resolution of $1500px \times 1500px$ was too high for real-time operation. With a resolution of $1500px \times 1500px$ the average run was $341ms$, see Figure 6.1. The resolution was lowered to $1100px \times 1100px$ for the rest of the experiment, and will be the resolution that is later discussed. The run-times were rounded to the nearest integer, and two outliers of $23075ms$ and $33135ms$ were removed from the data set.

The run-times are presented in Figure 6.2 and in Figure 6.3. In Figure 6.2 the 14750 run-times are presented in a histogram. Each bin represents a millisecond. The main density of data points are between $200ms$ and $265ms$, with less than 1% of the data points above $275ms$. The average run-time was $233.2ms$, and represented by a red dashed line in the histogram. The goal of $200ms$ is represented

Figure 6.1: IPM image resolution of $1500px \times 1500px$.



(a) A histogram showing the distribution of run-times to make the bird's eye view image during the experiment. Run-times are rounded to the nearest integer. The x-axis is the run-time in ms, while the y-axis shows how many times a runtime occurred. The goal of $200ms$ is shown as a dashed green line, while the average run-time is show with a dashed red line. The data is from the ROS2-logger function.



(b) A line graph showing the fluctuation of the run-time during the experiment. The x-axis is the elapsed time during the experiment in minutes, while the y-axis is the run-time in ms. The goal of $200ms$ is shown as a dashed green line, and the average run-time is show with a dashed red line. The data is from the ROS2-logger function.

by a green dashed line. From the profiler-file made by `cProfiler`, the average run-time of $233.2ms$ can be allocated to specific functions in the program, see Table 6.1.

In Figure 6.3, the fluctuation of the run-times during the experiment are displayed. The total time of the log-file is 57.3 minutes. As in Figure 6.2, the average run-time is displayed by a red dashed line and the goal by a green dashed line.

6.2 Video and image results

The visual result from the live experiment are presented in two ways. The main result is a video made with the footage captured during the experiment. The video shows three views: The bird's eye view, the drone view and the footage of the operator. The bird's eye view was recorded to a `rosbag`-file during the experiment, and therefore had to be extracted and converted into a *.mp4*-format. This was done in `Python` with the libraries `OpenCV` and `rosbags`. The frame rate of the video was set to match the average run-time.

The video shows two docking situations, where in the first one, the operator is docking the ferry using the bird's eye view on an external monitor. The video are available on YouTube, and must be watched to get the best impression of the results.

In addition to the video presented above, multiple images from the video are

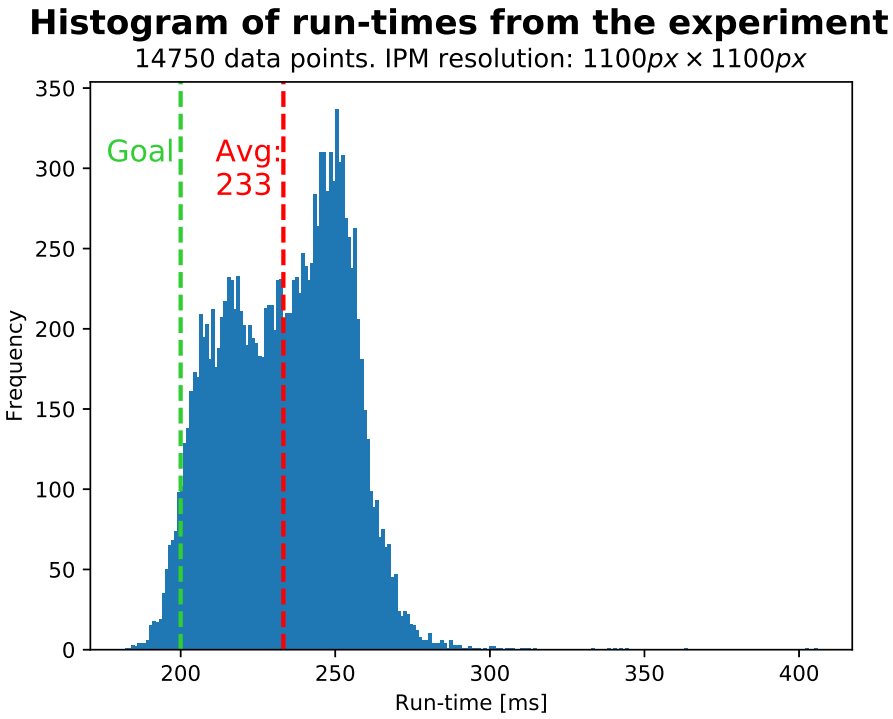


Figure 6.2: A histogram showing the distribution of run-times to make the bird's eye view image during the experiment. Run-times are rounded to the nearest integer. The x-axis is the run-time in ms, while the y-axis shows how many times a runtime occurred. The goal of $200ms$ is shown as a dashed green line, while the average run-time is show with a dashed red line. The data is from the ROS2-logger function.

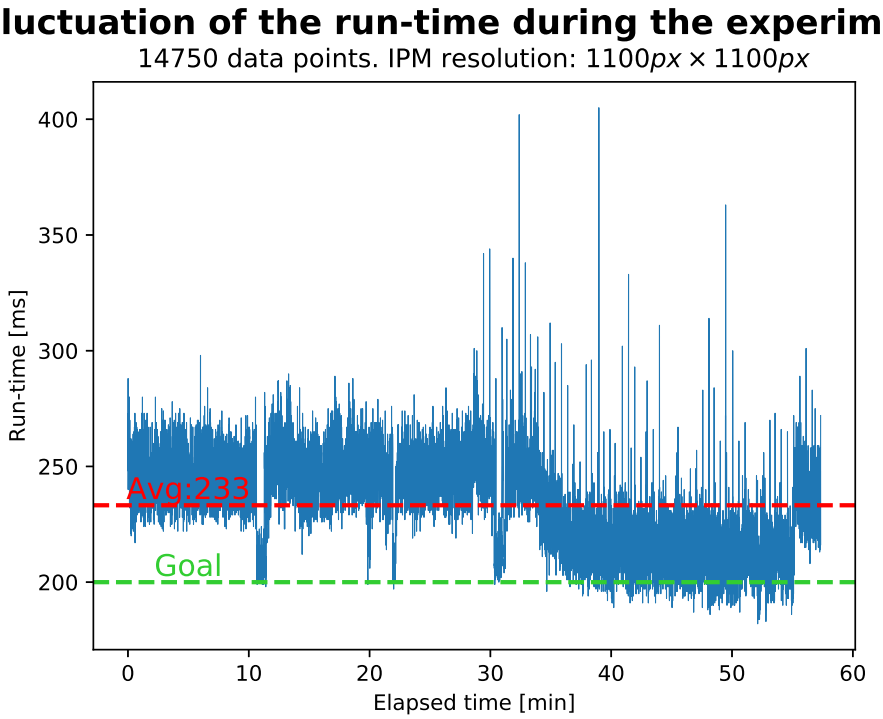


Figure 6.3: A line graph showing the fluctuation of the run-time during the experiment. The x-axis is the elapsed time during the experiment in minutes, while the y-axis is the run-time in ms. The goal of $200ms$ is shown as a dashed green line, and the average run-time is show with a dashed red line. The data is from the ROS2-logger function.

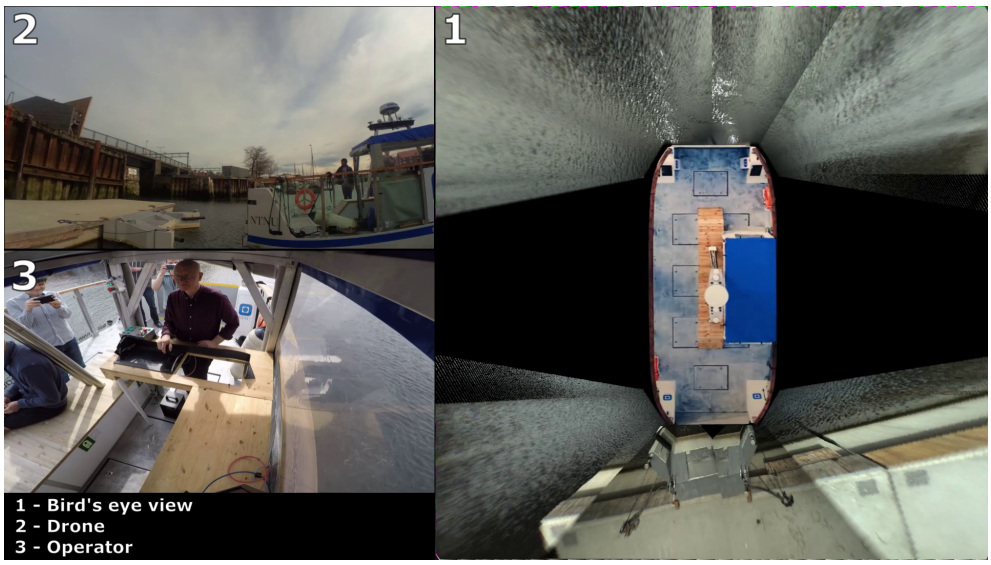
Function name	Time [*ms*]	Percentages of the average run-time 233.2*ms*
`make_BEW()`	179.9	77.14%
`cv_to_msg`	19.73	8.98%
`msg_to_cv` (8×)	8.36	3.80%
`logger` (9×)	9.324	4.24%
`publish`	2.507	1.14%
Sum	219.8	95.30%

Table 6.1: The table presents how the average run-time of 233.2*ms* is allocated to each function. The time of `msg_to_cv` and `logger` are multiplied by the number of times they are called during the making of 1 IPM image, respectively 8 and 9 times. Data from `cProfiler`-file with 14750 IPM images.

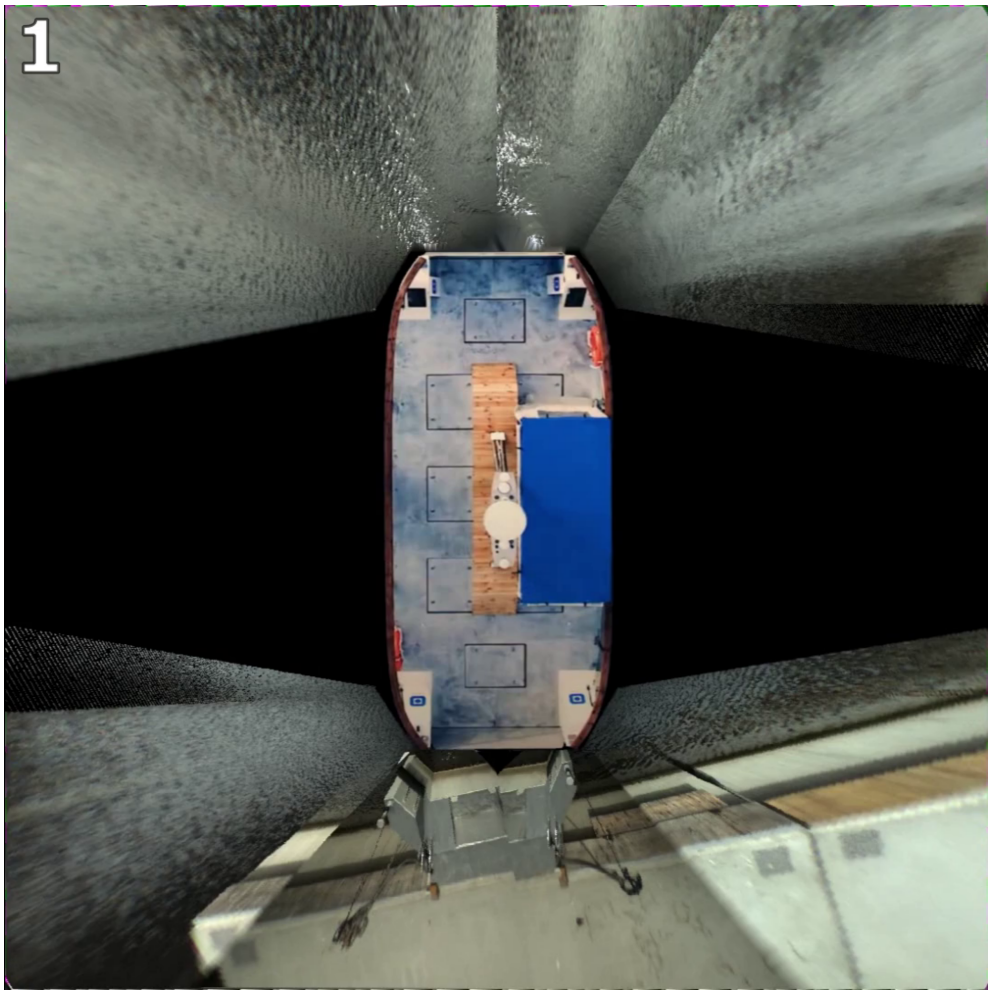
included in the thesis. A frame from the video can be seen in Figure 6.4a and the enlarged IPM image can be seen in Figure 6.4b. Figure 6.5 showcases how the operator (to the right) was able to see the IPM image on a monitor (to the left) while operating the ferry towards the dock in front.

During the first docking, the connection between the cameras and the computer disconnected due to an unknown error. This resulted in that the system did not receive any new images to make the IPM images. The effect of how the system handled the disconnection can be viewed in Figure 6.6. In Figure 6.6a the IPM image should have shown the dock, since the ferry is next to the dock (see view 2 in Figure 6.6a). However, the IPM image only displays open sea. One IPM image later, in Figure 6.6b, half the dock can be observed after the cameras started to reconnect to the computer. One more IPM image later, all the cameras had reconnected, and the IPM images shows the whole dock as expected, see Figure 6.6c.

Figure 6.4



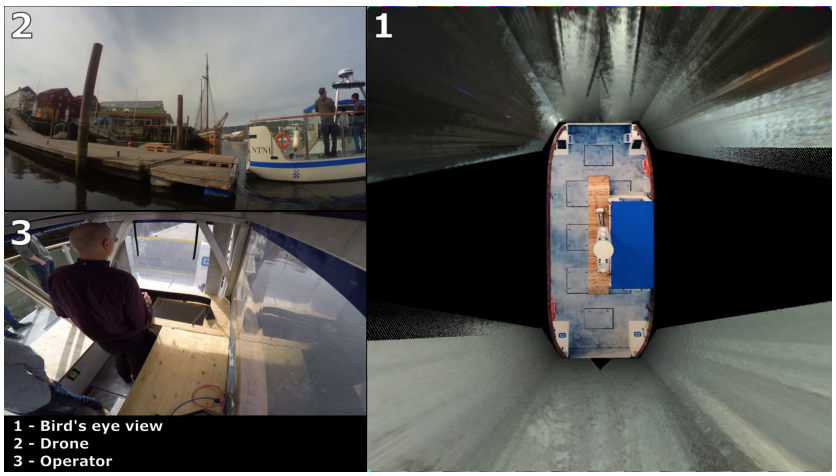
(a) Image is a frame from the video result. There are three views, the bird's eye view (1), the drone view (2) and the view of the operator (3). The image shows a docking situation where the operator can see the bird's eye view image in real-time on an external monitor in front of him.



(b) Figure shows a larger image of the bird's eye view from Figure 6.4a.

Figure 6.5: An image of how the operator (to the right) was able to see the IPM image on a monitor (to the left) while operating the ferry towards the dock in front.

Figure 6.6: Showing how the system handled the disconnection between the cameras and the computer during the first docking. The sub figures display 3 consecutive IPM images. The total time between Figure 6.6a and Figure 6.6c was about 233*ms*. During this time period, it can be observed in view 2 that the ferry did not move significantly.



(a) The IPM image shows open sea, instead of the dock in front of the ferry, due to a disconnection between the cameras and the computer.

(b) The IPM image now shows the right half of the dock, since the cameras started to reconnect to the computer.

(c) The whole dock can now be observed in the IPM image, as all cameras had reconnected.

Chapter 7

Discussion

In this chapter, discussion around the results and methods will be presented. The chapter will be divided into two parts: One for the code optimization and the run-times of both the live experiment and the pre-recorded setup, and one about the visual accuracy and operator usefulness in a real environment.

7.1 Run-time and optimization

One of the primary influencers of the run-time was the choice of programming language. This master thesis is based on the work done in my project thesis, which were written in `Python`. Given that there was an unknown factor of how a real-time version of that system would look like, and what the run-time limits of `Python` were, the choice were made to continue developing the 360-degree bird's eye view system in `Python`. The programming language was chosen, despite its reputation of being a slow and inefficient language, since it provides a solid platform for fast implementation and testing of code. A sought-after trait for programming languages used in research and development. The initial assumption was that a run-time of less than 200*ms* could be achieved, despite the reputation of Python. This was later proven to be true, at least during the code optimization phase.

To further improve the capabilities of `Python`, numerous options exist to boost its efficiency. Many of them were tested and used during the experimentation in Chapter 4. Among them were multiprocessing techniques and the utilization of `Python` libraries capable of executing code in faster programming languages such as `Fortran`, `C`, and `C++`. While these libraries offer a convenient `Python` interface with run-times comparable to faster languages, their performance did not achieve the desired run-time goal. As a result, the code was refined to use AAI and array transforming operations in a custom-made interpolation method.

Despite achieving a run-time below the desired threshold of 200*ms* during the code optimization phase, the performance was not matched while running on *milliAmpere2*. This is shown by the average run-time in Figure 6.2. There are several reasons why this was the case:

1) The CPU on *milliAmpere2* is 19.6% slower (3.7GHz vs 4.6GHz) than the computer used during the code optimization. This is assuming both CPUs were working at maximum clock frequency. Since the program is mainly executed on a single core, it can be assumed to have a direct increase of 24.3% to the run-time. This can be observed in the performance of the `make_BEW()`-function, which had an average run-time of 130.0ms with a resolution of $1100px \times 1100px$ during the code optimization phase. However, when running on *milliAmpere2* with the same resolution, the run-time increased to 179.9ms, reflecting a 38.4% increase that exceeded the estimated impact of a slower CPU.

2) The adaptation of the code from the code optimization phase into a ROS2 node-structure, in order to be able to run on *milliAmpere2*, resulted in increased overhead and subsequently higher run-time. From the data in Table 6.1, it is apparent that this overhead accounted for 22.86% of the average run-time. While certain functions were identifiable as sources of overhead, there were also unidentified sources that impacted the run-time. Notably, the recording of data from multiple sensors during the execution of the bird's eye view system had a significant impact on the run-time. A decision to only recorded the IPM images to the `rosbags` were taken to minimize this effect.

From an analysis of Figure 6.2, one can argue there is a presence of two density peaks, with one peak centered around 250ms and another around 215ms. This finding is further supported by the data in Figure 6.3, where there are two distinct periods. Prior to the 35-minute mark, the average run-time is around 250ms, while after 35 minutes, there is a noticeable reduction to approximately 215ms. Additionally, a closer analysis of Figure 6.3 reveals an increase in spikes with higher run-times occurring after the 30-minute mark. Lastly, there are two significant drops in run-times at approximately 11 minutes and 31 minutes. Determining the exact reasons behind these fluctuations and changes are challenging due to the lack of data regarding concurrent systems or potential changes in the recording of data. Nevertheless, it is evident that the performance of the 360-degree bird's eye view system is significantly impacted by other processes and programs running on the computer.

It is also worth highlighting the change in the way run-times were determined and presented in the results. Initially, in Chapter 4, the run-time was defined as the processing time of the `make_BEW()`-function. The decision was made, due to it being the only known part of the system that would later be executed on *milliAmpere2*. Moreover, it was also assumed to take the majority of the run-time, therefore reducing it would have the largest positive impact on the run-time during the experiment. During the live experiment, the focus was shifted to measuring the total time between each IPM image published on the *im_BEW* topic, since the frame rate during the experiment was more important than the run-time of an individual function.

Throughout the research and development process of the system, the documentation of each improvement and the resulting IPM image were of varying levels of quality. This was primarily due to the limited experience in document-

ing code optimization procedures and the broad variety of optimization options available. Furthermore, due to the pipeline-design of the code, the whole system had to work to be able to time the execution. This lead to some gaps in the optimization documentation. Nevertheless, the available documentation is sufficient to describe the major improvement steps that were taken.

Although the documentation was at times lacking during the initial part of the research, there was a notable increase in quality during the research outlined in Section 4.2.4. This was parts due to more experience, and parts due to a narrower scope of possible improvements, making it easier to track progress and maintain a well-structured approach. This can for instance be observed in the extra external tests were performed to validate the effectiveness of the implemented improvements.

Drawing from the insights of the `linearNDInterpolator()` description for the custom-made interpolation method turned out to be a smart choice. Not only did it speed up the development process by using an existing technique, but it also granted higher reliability and quality of the interpolation result. By utilizing a well-established approach, it saved time and effort that could be devoted to preparing for the live experiment. Moreover, this decision made image comparison easier throughout the development phase, especially during the testing of different interpolation methods.

7.2 Visual accuracy and operator usefulness in a real environment

Accurate camera calibration, both intrinsic and extrinsic, is of paramount importance in achieving optimal bird's eye view images. Camera calibration plays a crucial role in undistorting images, to ensure adherence to the pinhole camera model, and accurately transforming pixel positions from the image plane $\mathcal{F}_I$ to the sea Π_{sea}. Having a good calibration is therefore essential to produce accurate IPM image.

Unfortunately, there was not an accurate camera calibration available during the research. An attempt to calibrate the cameras were done before the experiment, but due to lack of access to the correct systems on *milliAmpere2* and little experience with the practical side of calibration, a new and more accurate calibration was not achieved. Instead, the CAD-model of *milliAmpere2* was used for the extrinsic parameters, and an older intrinsic calibration of the cameras.

The consequences of the inaccurate calibration can be observed in multiple examples of the bird's eye view. One example can be seen in Figure 7.1a, where the straight lines on the dock do not form a continuous straight line across the dock. Instead, they exhibit abrupt steps when crossing the edges of the cameras' FOV. This is visible across all three FOV edges. Furthermore, it can be seen in Figure 7.2b that the docking station (inside the red and white square) are duplicated across the cameras' FOV. The docking station should have appeared as one single object.

This duplication is most noticeable when the ferry is far away from the dock and gradually diminishes to non-continuous straight lines as the ferry approaches, as shown in Figure 6.4a.

To illustrate the difference of the camera calibration used in this thesis and an accurate one, a bird's eye view made with *milliAmpere1* are compared to the bird's eye view of *milliAmpere2* in Figure 7.3. The ferries are positioned at about the same distance from the same dock. In Figure 7.3a the straight lines on the dock are non-continuous, while they are continuous in Figure 7.3b. In addition, the edges between the cameras' FOVs are a lot less visible in the IPM image made with *milliAmpere1*. This probably also due to less brightness difference between the cameras on *milliAmpere1* when the image was taken.

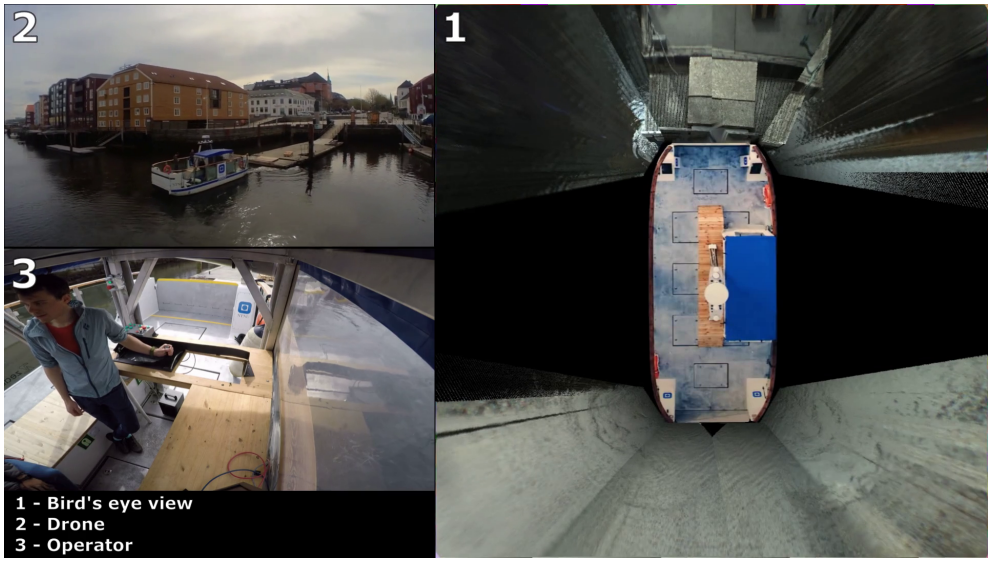
A few other notable image artifacts exist. There are two areas where the black fill area (made in Section 4.2.1) intersect the area covered the cameras, pointed at by the red arrows in Figure 7.4. This issue arose from opting for a slightly different extrinsic parameter file during the setup of the experiment, as it produced slightly straighter lines in front of the ferry. Unfortunately, due to time constraints, it was not possible to correct the problem prior to carrying out the experiment, since it had to be done manually. Automating the black fill area creation could have prevented these image artifacts.

Another area that could have been improved, was the division of the IPM into FOV-sections for each camera. The three-step process explained in Section 3.3.4 did not work as intended without manual intervention in adjusting the values of r_1, r_2, q_1, and q_2. It was attempted to make a universal function capable of handling all the different image overlaps, but it was difficult due to the irregularity of the input data. One of the factors contributing to this issue was the variation in the horizon position across each undistorted image, which the function used to decide the input data. In hindsight, it would have been better to set the positions of r_1, r_2, q_1 and q_2 manually right from the beginning, as it would have been a one-time job and would not have been affected by future calibrations. Furthermore, it would also have made it easier in the future to adjust the FOV-sections in the IPM image.

There is one physical limitation to the system on *milliAmpere2* that affects the quality of objects in the IPM image and the practical max distance that can be viewed: The camera placements. More precisely, the camera elevation. The cameras are placed at a height of about $1.2m$ above the sea, which is low compared to *milliAmpere1*, where the cameras' height is about $3.7m$. This is further supported by the results in my project thesis, where it was concluded that: The higher the camera placement, the better the bird's eye view. However, this system was designed to aid the operator in docking situations, where distances are relatively short. As answered by Øystein Kaarstad Helgesen in the questionnaire (which can be found in the appendix, A.2), the viewing distance was perfect for docking situations.

Furthermore, the camera position and rotation also influence how close to the ferry that can be observed. There is a blind zone within the last meter, which could

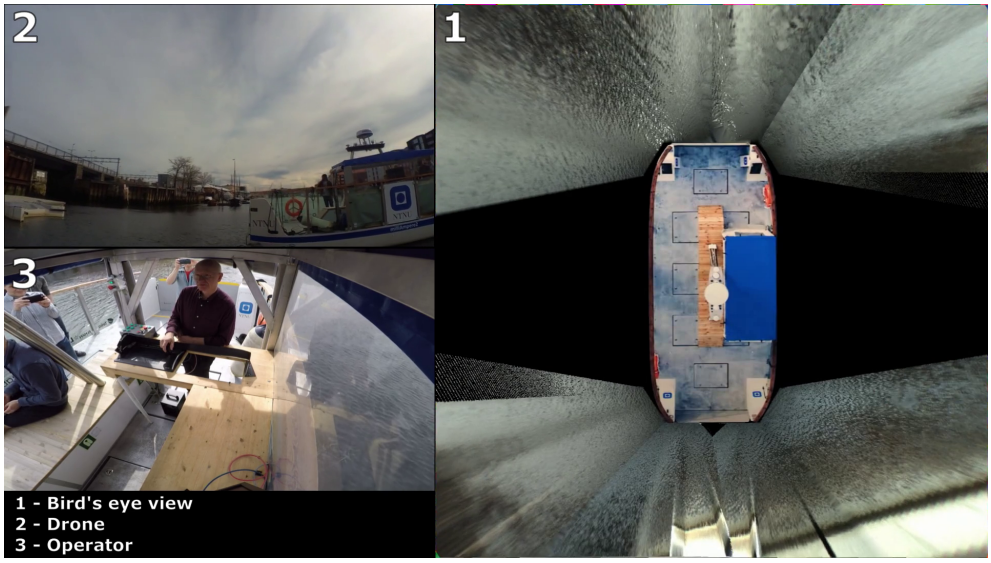
Figure 7.1



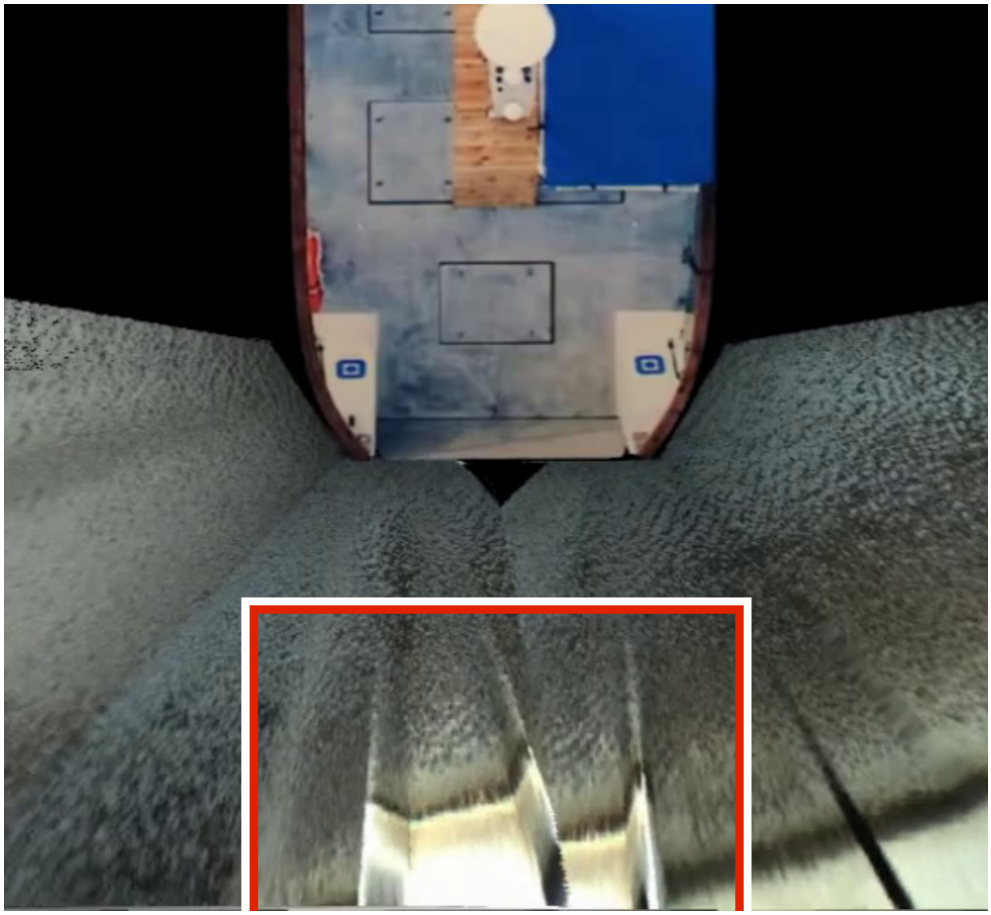
(a) Image is a frame from the video result. In view 1, one can observe that the straight lines on the dock does not align across the different camera FOVs. A closer view can be seen in Figure 7.1b.

(b) Cropped and zoomed in image of view 1. The image shows how the straight lines on the dock does not make a continuous straight line across the different camera FOVs.

Figure 7.2

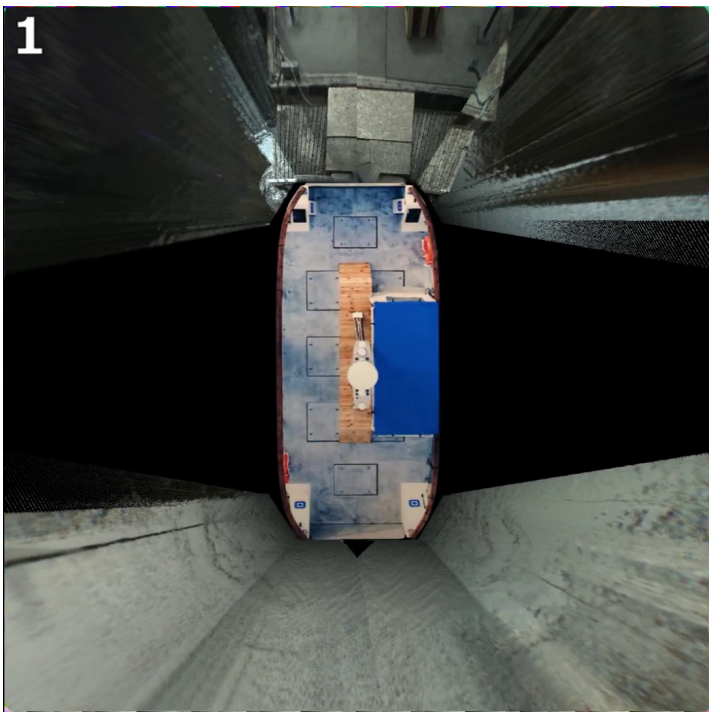


(a) The image is a frame from the video result. In view 1, one can observe that the docking station is duplicated in the FOVs of camera ap_a and as_a. A closer view can be seen in Figure 7.2b.

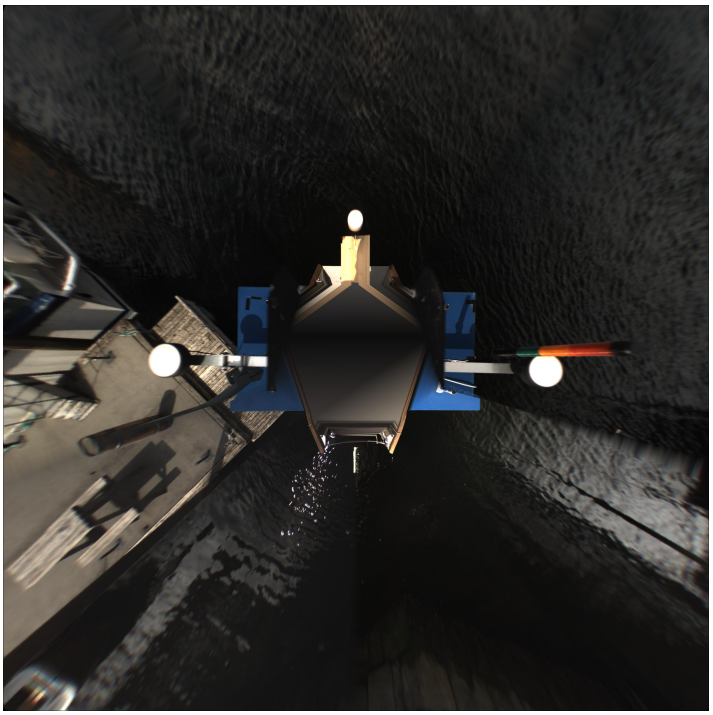


(b) A cropped and zoomed in image of view 1. The image shows that the docking station is duplicated in the FOVs of camera ap_a and as_a, instead of one single docking station.

Figure 7.3: Two images comparing the bird's eye view created with *milliAmpere1* and with *milliAmpere2*. In a) the straight lines on the dock are non-continuous, while they are continuous in b). Both images are taken of the same dock from the approximately same distance. In a) there are 3 FOV edges dividing the dock, while there is one FOV edge in b).



(a) IPM image created with *milliAmpere2*.



(b) IPM image created with *milliAmpere1*. Image taken from my project thesis.

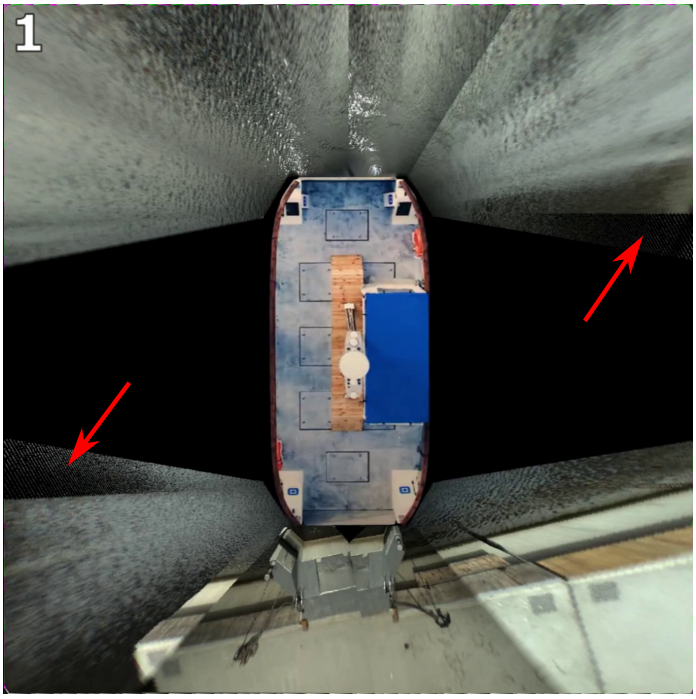


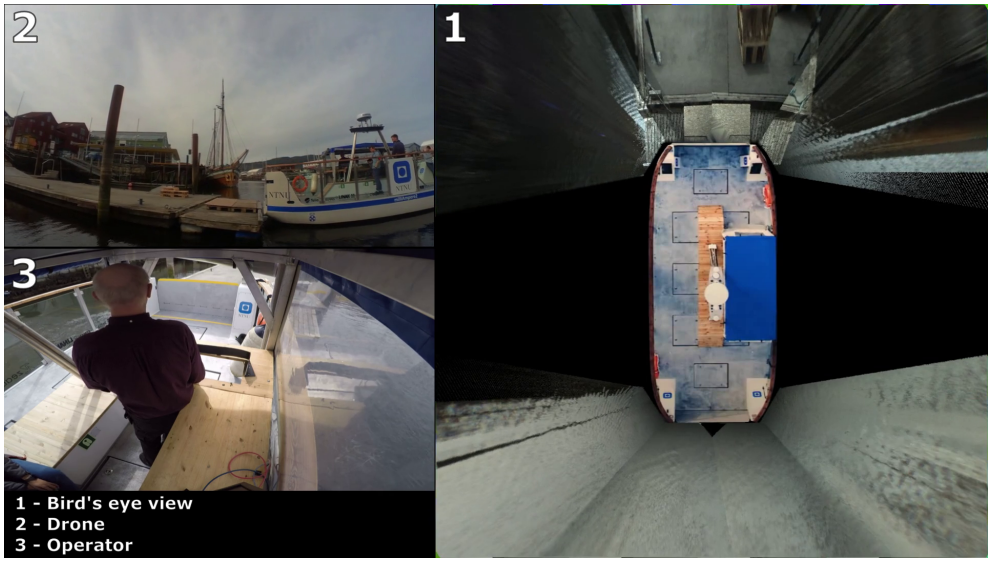
Figure 7.4: Image showing the area (pointed at by the red arrows) where the black fill area, made in Section 4.2.1, are overlapping part of the IPM image that is covered by images from the cameras.

have been problematic. However, according to Helgesen's feedback, this was not a concern as the operator at that point had already aligned the ferry correctly and could navigate the final distance. To provide a visual representation of what the operator can view during docking, an image from the view of the operator is shown in Figure 7.5. The ferry ramp covers the dock from the operator's view, whereas it is clearly visible on the monitor with the bird's eye view to the left.

Figure 7.5: An image showing the operator's view during docking. Clearly, it is difficult to observe the dock behind the ramp. However, on the bottom part of the monitor on the left-hand side of the image, the dock can be observed by the operator in the bird's eye view image.

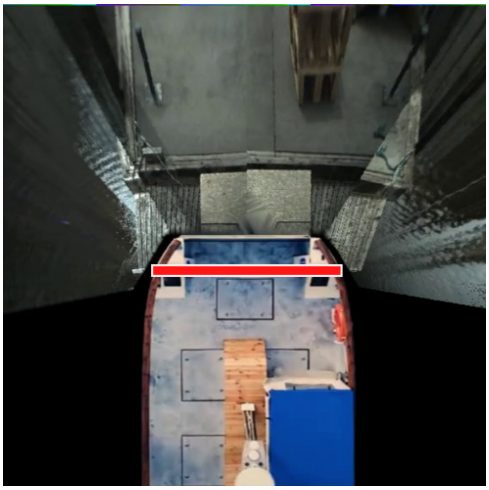
The bird's eye view also have two large blind zones on the port and starboard sides, as seen in Figure 6.4a. Nevertheless, this issue is considered minor or non-existent due to the ferry's docking procedure, which involves docking fore and aft. Furthermore, the operator has a significantly better view of the sides of the ferry compared to the fore and aft sections, thereby minimizing the need for assistance.

To further improve the operator's situational awareness, a top-down image of *milliAmpere2* was incorporated into the center of the IPM image. The positioning and size of the ferry were determined manually. However, this led to an inaccurate representation of the ferry's scale, as seen in Figure 7.6. Specifically, Figure 7.6c highlights the issue, with the red and white line indicating the dock's edge and illustrates how the ferry extends into the dock. To provide a visual reference of the actual distance between the ferry and the dock, a zoomed-in view from the drone perspective is presented in Figure 7.6b. These observations clearly indicate the need to scale down the size of the ferry in the image. Nonetheless, Helgesen expressed positive feedback, emphasizing that it was a "very positive" addition to the system.



(a) The image is a frame from the video result. In view 1, one can observe that the inserted image of *milliAmpere2* clips into the dock.

(b) A cropped and zoomed in image of view 2. The distance between the dock and the ferry can be observed inside the red and white square.



(c) A cropped and zoomed in image of view 1. The red and white line indicates where the edge of the dock is in reality. It is clear that the image of *milliAmpere2* is place incorrectly, as it clips into the dock.

Figure 7.6: The figure shows how the inserted image of *milliAmpere2* clips into the dock, even though the ferry in reality is right next to the dock.

Despite the program successfully handling the disconnection of the cameras during the first docking, there was no immediate response from the operator or other individuals on board. The fail-safe system wrote the correct error message to the terminal, but the terminal was not visible to the operator. A far more effective and visually intuitive solution would have been to highlight the areas of the IPM image that were not being updated, as demonstrated in Figure 7.7. This visual indication would have likely grabbed the operator's attention, with the red markings signaling that something was wrong. Even without knowing the specific error, the operator would have been incited to respond to the situation.

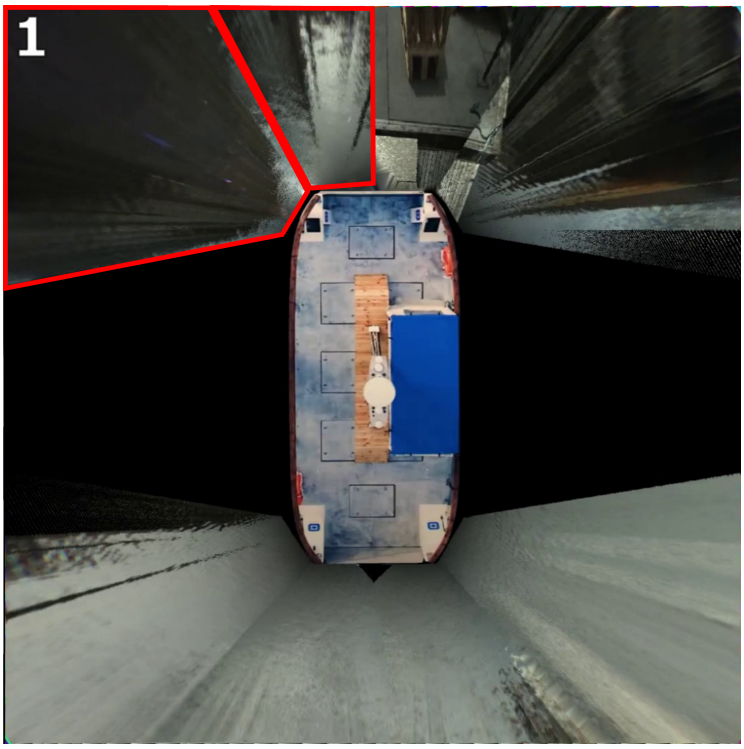


Figure 7.7: An image showing a better solution to alerting the operator that two cameras have not been updated. The camera FOVs that are surrounded by red are displaying old images of, while the other camera FOVs are updated.

Despite the run-time being on average above the real-time goal of 200*ms*, the system was still considered fast enough to provide useful aid. The slower speed at which the ferry navigates during docking mitigated the impact of the slightly longer run-time. This was also supported by Helgesen's feedback, where he answered that the refresh rate of the bird's eye view was deemed high enough to avoid a significant negative impact on docking. However, he also noted that higher refresh rates would be beneficial, and indicated that image synchronization could have improved consistency in the IPM images.

Lastly, it should be noted that the presented system can be utilized on any surface vessel running ROS2 with cameras, not only *milliAmpere2*. An important

feature if the system were to be commercialized. To successfully be utilized by another vessel, a few adjustments to the code, and new extrinsic and intrinsic parameter files are required.

Chapter 8

Conclusion and future work

8.1 Conclusion

In this thesis, a 360-degree bird's eye view system for *milliAmpere2* with borderline real-time performance was presented. Python was chosen as the programming language for its fast implementation and testing capabilities, despite its reputation for being slow. Initially, the desired run-time goal of less than 200*ms* was reached during the code optimization phase, due to efficient implementation of the interpolation function. Though, the run-time goal was not fully achieved in the live experiment with *milliAmpere2* due to a slower CPU and increased overhead from the full autonomy system running simultaneously.

Moreover, it can be concluded from the observed fluctuations in run-time that the run-time is significantly influenced by other processes running on the computer simultaneously. Throughout the course of the experiment, there were instances where the run-time approached 250*ms*, whereas in other instances, it neared 215*ms*, and close to the desired goal of 200*ms*.

The inaccurate camera calibration had a negative impact on the quality of the bird's eye view images. Straight lines appeared discontinuous across the FOV edges, and the dock was duplicated at far distances. The maximum view distance with reasonable quality was also limited by the height of the camera. However, from the feedback of the operator, the distance limitation was regarded as non-existent. The addition of a top-down image of the ferry in the center of the IPM image proved beneficial, although scaling down the size of the ferry image would improve the helpfulness.

Despite the shortcomings of the system, the operators found it "provided useful additional assistance" during the docking process. Further refinement should however be conducted before deployed in real use.

8.2 Future work

One potential improvement is rewriting the code in a faster programming language, such as `C` or `C++`, which has the potential to significantly enhance efficiency and achieve run-times well below the target of 200ms. Additionally, another area to explore is utilizing a dedicated GPU, known for its efficiency in performing interpolation tasks.

An additional area for improvement lies in optimizing the initialization process of the system. During the initialization, the program calculates the new pixel position in the IPM image for all the pixels in the 8 images taken by the cameras. Considering the high resolution of these images, the total number of pixel positions to be calculated exceeds 10 million, resulting in a high initialization time. Although the efficiency of the initialization was not prioritized in this thesis's, a faster initialization process would be advantageous for potential commercial applications. Utilizing libraries like `Numba` with `jit` could potentially yield significant improvements, due to the repetitive nature of the calculations in the initialization.

To improve the visual aspect of the system, the inclusion of an automatic camera calibration could greatly improve the accuracy and quality of the IPM image. Moreover, with auto-calibration of the cameras, an automatic system to generate the black fill area would be also be necessary. Additionally, having synchronized cameras would also improve the accuracy of the IPM image.

Lastly, to improve the utility of the bird's eye view for the operator, the addition of distance lines at intervals of 1m, 2m, and 5m for and aft of the ferry could be implemented. This could further aid the operator's distance estimation to objects in the vicinity.

Bibliography

[1] E. F. Brekke, E. Eide, B.-O. H. Eriksen, E. F. Wilthil, M. Breivik, E. Skjellaug, Ø. K. Helgesen, A. M. Lekkas, A. B. Martinsen, E. H. Thyri, T. Torben, E. Veitch, O. A. Alsos and T. A. Johansen, 'Milliampere: An autonomous ferry prototype,' *Journal of physics. Conference series*, vol. 2311, no. 1, p. 12 029, 2022, ISSN: 1742-6588.

[2] O. A. Alsos, E. Veitch, L. Pantelatos, K. Vasstein, E. Eide, F.-M. Petermann and M. Breivik, 'Ntnu shore control lab: Designing shore control centres in the age of autonomous ships,' *Journal of physics. Conference series*, vol. 2311, no. 1, p. 12 030, 2022, ISSN: 1742-6588.

[3] N. P. Reddy, M. K. Zadeh, C. A. Thieme, R. Skjetne, A. J. Sørensen, S. A. Aanondsen, M. Breivik and E. Eide, 'Zero-emission autonomous ferries for urban water transport: Cheaper, cleaner alternative to bridges and manned vessels,' 2019, ISSN: 2325-5897.

[4] Ø. K. Helgesen, K. Vasstein, E. F. Brekke and A. Stahl, 'Heterogeneous multi-sensor tracking for an autonomous surface vehicle in a littoral environment,' *Ocean engineering*, vol. 252, p. 111 168, 2022, ISSN: 0029-8018.

[5] E. H. Thyri, M. Breivik and A. M. Lekkas, 'A path-velocity decomposition approach to collision avoidance for autonomous passenger ferries in confined waters,' *IFAC PapersOnLine*, vol. 53, no. 2, pp. 14 628–14 635, 2020, ISSN: 2405-8963.

[6] A. Aurlien, M. Breivik and B. O. H. Eriksen, 'Multivariate modeling and adaptive control of autonomous ferries,' *IFAC-PapersOnLine*, vol. 54, pp. 395–401, 16 2021, ISSN: 24058963.

[7] Ø. K. Helgesen, A. Stahl and E. F. Brekke, 'Maritime tracking with georeferenced multi-camera fusion,' *IEEE Access*, vol. 11, pp. 30 340–30 359, 2023. DOI: 10.1109/ACCESS.2023.3261556.

[8] A. Muad, A. Hussain, S. Samad, M. Mustaffa and B. Majlis, 'Implementation of inverse perspective mapping algorithm for the development of an automatic lane tracking system,' in *2004 IEEE Region 10 Conference TENCON 2004*, vol. A, Piscataway NJ: IEEE, 2004, 207–210 Vol. 1, ISBN: 0780385608.

[9] M. Oliveira, V. Santos and A. D. Sappa, 'Multimodal inverse perspective mapping,' *Information fusion*, vol. 24, pp. 108–121, 2015, ISSN: 1566-2535.

[10] T. Bruls, H. Porav, L. Kunze and P. Newman, 'The right (angled) perspective: Improving the understanding of road scenes using boosted inverse perspective mapping,' *arXiv.org*, 2019, ISSN: 2331-8422.

[11] C. D. Prakash, F. Akhbari and L. J. Karam, 'Robust obstacle detection for advanced driver assistance systems using distortions of inverse perspective mapping of a monocular camera,' *Robotics and autonomous systems*, vol. 114, pp. 172–186, 2019, ISSN: 0921-8890.

[12] M. K. Plenge-Feidenhans'l and M. Blanke, 'Open water detection for autonomous in-harbor navigation using a classification network,' in *IFAC PapersOnLine*, vol. 54, Elsevier Ltd, 2021, pp. 30–36.

[13] D. Griesser, D. Dold, G. Umlauf and M. O. Franz, 'Cnn-based monocular 3d ship detection using inverse perspective,' in *Global Oceans 2020: Singapore – U.S. Gulf Coast*, 2020, pp. 1–8.

[14] V. Bannore, *Iterative-Interpolation Super-Resolution Image Reconstruction : A Computationally Efficient Technique* (Studies in Computational Intelligence), 1st ed. 2009. Berlin, Heidelberg: Springer Berlin Heidelberg : Imprint: Springer, 2009, vol. 195, ISBN: 3-642-00385-0.

[15] T. Lehmann, C. Gonner and K. Spitzer, 'Survey: Interpolation methods in medical image processing,' *IEEE transactions on medical imaging*, vol. 18, no. 11, pp. 1049–1075, 1999, ISSN: 0278-0062.

[16] C. Dong, C. C. Loy, K. He and X. Tang, 'Image super-resolution using deep convolutional networks,' *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 2, pp. 295–307, 2016, ISSN: 0162-8828.

[17] S. H. Fuller and L. I. Millett, 'Computing performance: Game over or next level?' *Computer (Long Beach, Calif.)*, vol. 44, no. 1, pp. 31–38, 2011, ISSN: 0018-9162.

[18] J. Newman. 'Moore's law stutters: Intel officially puts "tick-tock" cpu release cycle on hiatus | pcworld.' (Feb. 2016), [Online]. Available: `https://www.pcworld.com/article/420217/moores-law-stutters-intel-officially-puts-tick-tock-cpu-release-cycle-on-hiatus.html` (visited on 29/03/2023).

[19] E. Bezzam, S. Kashani, P. Hurley, M. Vetterli and M. Simeoni, 'Pyffs: A python library for fast fourier series computation and interpolation with gpu acceleration,' *SIAM journal on scientific computing*, vol. 44, no. 4, pp. C346–C366, 2022, ISSN: 1064-8275.

[20] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge: Cambridge University Press, 2004, ISBN: 9780521540513.

[21] W. Li, W. Huang, M. Breier and D. Merhof, 'Have we underestimated the power of image undistortion?,' vol. 2016, pp. 2946–2950, 2016, ISSN: 1522-4880.

[22] O. Stankiewicz, G. Lafruit and M. Domański, 'Multiview video: Acquisition, processing, compression, and virtual view rendering,' in *Academic Press Library in Signal Processing: Image and Video Processing and Analysis and Computer Vision*, vol. 6, 2018, pp. 3–74, ISBN: 9780128119006.

[23] G. Bradski, 'The opencv library,' *Dr. Dobb's Journal*, vol. 25, no. 11, pp. 120–125, 2000, ISSN: 1044-789X.

[24] W. Burger and M. J. Burge, *Digital Image Processing: An Algorithmic Introduction Using Java* (Texts in Computer Science), 2nd ed. 2016. London: Springer London, Limited, 2016, ISBN: 9781447166832.

[25] A. Moore, 'Efficient memory-based learning for robot control,' Carnegie Mellon University, Pittsburgh, PA, Tech. Rep., Nov. 1990.

[26] Ø. Hjelle and M. Dæhlen, 'Triangles and triangulations,' in *Mathematics and Visualization*, ser. Mathematics and Visualization 9783540332602, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–21, ISBN: 354033260X.

[27] Edmund Optics Inc., *Flir blackfly s bfs-u3-50s5c-c camera | edmund optics*, Apr. 2019. [Online]. Available: https://www.edmundoptics.com/p/bfs-u3-50s5c-c-usb3-blackflyreg-s-color-camera/41351/ (visited on 24/04/2023).

[28] AUTOFERRY. '360 degrees view for the operator of milliampere 2.' (2022), [Online]. Available: https://autoferry.github.io/sf/2022/04/02/360view/ (visited on 09/01/2022).

[29] The SciPy community, *Scipy.interpolate.linearndinterpolator — scipy v1.10.1 manual*, 2023. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.LinearNDInterpolator.html (visited on 02/03/2023).

[30] AMD, *Amd epyc™ 7313p*, Dec. 2020. [Online]. Available: https://www.amd.com/en/products/cpu/amd-epyc-7313p (visited on 25/05/2023).

Appendix A

Additional Material

A.1 The paper submitted The International Conference on Maritime Autonomous Surface Ships (ICMASS) 2023 in Rotterdam, based on this thesis

Real-time 360 degrees view for the operator of *milliAmpere 2*

M T Paasche[1], Øystein Kaarstad Helgesen[2], Edmund Førland Brekke[1]

[1]Department of Engineering Cybernetics, NTNU, O.S. Bragstads plass 2D, Elektroblokk D, Trondheim, Norway
[2]Zeabuz AS, Skippergata 14, Trondheim, Norway

E-mail: mathiatp@stud.ntnu.no, oystein.helgesen@zeabuz.com, edmund.brekke@ntnu.no

Abstract. In the evolving domain of autonomous marine operations, accurate perception and representation of the surrounding environment are crucial for safe and effective execution. This paper addresses this issue by developing and testing a near real-time 360-degree bird's eye view system for the situations where the ferry, *milliAmpere 2*, has to be manually controlled by a local operator onboard. The goal was to aid the operator during the critical phase of docking, by displaying the surrounding area of the ferry from a bird's eye view. The bird's eye view was made by using inverse perspective mapping on the undistorted images from the 8 cameras onboard. The system was implemented in `Python`, and aimed to reach a run-time of less than $200ms$. This goal was reached during the initial phase of the work. However, during live testing, only near real-time performance was achieved. Despite some shortcomings, the operators found the system to be a "useful additional assistance" during the docking process.

1. Introduction

The growing demand for sustainable transportation in cities has posed challenges for building bridges over rivers and channels. Trondheim's municipality faced similar obstacles when proposing a bridge construction project in 2016. As a result, researchers at NTNU embarked on a project to find alternative solutions, leading to the development of the *milliAmpere 2*, an autonomous passenger ferry.

Inverse perspective mapping (IPM) is a well-studied technique in the automotive industry for lane tracking and obstacle avoidance [1, 2, 3, 4]. While IPM has been primarily explored in that domain, its application in the maritime industry is still emerging. Some notable studies have investigated the use of IPM in different maritime contexts.

In [5], a path planning and navigation method for autonomous vessels using a convolutional neural network (CNN) and IPM is presented. A similar method is proposed in [6] to obtain $3D$ ship detection and tracking.

One of the main challenges with IPM is the progressively lower pixel density for objects further away from the camera. Interpolation can decrease the effect of this problem by filling empty pixels with *rgb*-values from surrounding pixels.

The research field of image interpolation has taken place in many different industries: Medical imaging, remote sensing, target detection and recognition, radar imaging, forensic science,

Figure 1. An image of *milliAmpere 2* from the side.

and surveillance systems [7]. In addition to covering a large group of different industries, a large amount of different interpolation methods have been developed. [8] compared 7 different interpolation methods for medical imaging in 1999, covering "traditional interpolation methods", such as nearest neighbor, linear interpolation, and Gaussian interpolation with different kernel sizes. In recent years, more complex and modern interpolation methods have been developed. One of these is *Super-Resolution*, which is able to enhance low-resolution images or video frames by increasing their spatial resolution [7].

This work addresses the challenges of implementing a real-time 360-degree visualization system for use on the *milliAmpere 2* autonomous ferry. A proof-of-concept system is demonstrated on *milliAmpere 2* with operators in a scenario comparable to the ferry's intended use and evaluated based on both real-time performance and operator feedback. This paper is based upon the Master thesis [9] written by author 1.

2. Theory

2.1. Camera model

A camera is a mapping between the 3D world and a 2D image [10]. In this work, the primary camera model was chosen to be the pinhole camera model. It maps a point in 3D space onto a 2D plane. This plane is often called the image plane or the focal plane and is represented by the frame $\mathcal{F}_I$. The position of the point $\mathbf{x}_I$ is determined by the intersection between the image plane and the drawn line from the point $\mathbf{X}_c$ to the camera center through the image plane. From the geometry of the camera model, the relationship between a point $\mathbf{X}_c$ and $\mathbf{u}$ can be expressed in homogenous coordinates as

$$\mathbf{u} = \mathbf{K}\,[\mathbf{R}|\mathbf{t}]\,\tilde{\mathbf{X}}_c \tag{1}$$

2.2. Inverse Perspective mapping

Inverse perspective mapping is the problem of determining the world coordinate position of a point based on its pixel location in an image. This is an underdetermined problem, where X_v, Y_v, Z_v are unknown and u, v is known. In a maritime context, the ocean surface can be modeled as a flat plane with a known elevation where the pixels originate from. It is assumed the plane (Π_{sea}) has the elevation $Z_v = 0$. A method to solve the IPM is described by [10]. With the flat plane assumption, the solution is given by Equation 2.

$$\begin{bmatrix} X_w \\ Y_w \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} p_{11} & p_{12} & t_x \\ p_{21} & p_{22} & t_y \\ p_{31} & p_{32} & t_z \end{bmatrix}}_{\mathbf{P}'}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}. \tag{2}$$

For this to be a valid solution, $\mathbf{P}'$ has to be invertible.

2.3. Interpolation

Interpolation is the process of estimating the intermediate values in a signal at continuous positions from a set of discrete samples [11]. The three most commonly used interpolation methods are nearest neighbor, bilinear (also called linear), and bicubic. For image data, nearest neighbor interpolation involves determining the value of a pixel based on the four nearest pixels. The value of the pixel with the shortest distance to the target pixel will be the chosen value. Bilinear interpolation also considers the four closest pixel values to the target pixel, but weights them based on their distance from the target pixel.

3. Image processing pipeline

MilliAmpere 2 is equipped with 8 electro-optical cameras of the type *FLIR Blackfly S 50-S5C* with a $6mm$ lens. Images are provided at a rate of $5Hz$ with resolution $1224px \times 1024px$. Each camera has a field of view (FOV) of 77.8°. However, due to the cameras' positions and orientations, full near-range coverage is not achieved due to blind zones on the port and starboard sides.

3.1. Image Processing Pipeline

The main steps of the image processing pipeline are presented in Figure 2. In practice, the

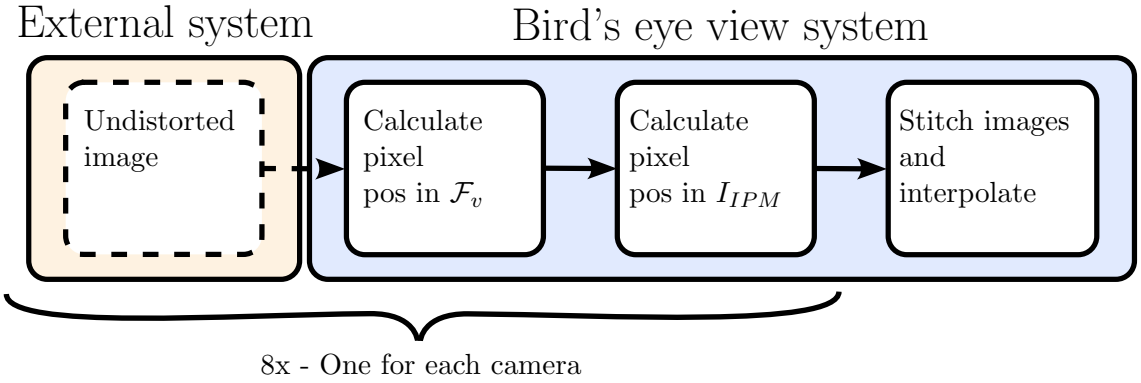


Figure 2. The main steps in the image processing pipeline.

IPM was solved according to [12]. The system solves Equation 2 to find X_v and Y_v, with the assumption that the target plane Π_{sea} has $Z = 0$.After calculating the $\mathcal{F}_v$ coordinates for all pixels in the image, a filter removed all points with a distance further out than $10m$ from the center of $\mathcal{F}_v$. The interpolation method used in this work is custom-made and inspired by [13]. The algorithm can be described in four steps:

(i) Triangulating the irregular input data (pixel positions) using Delaunay triangulation.

(ii) For every point in the new grid, the triangulation is searched to identify the simplex (a triangle) that encompasses the point.

(iii) The barycentric coordinates of each new grid point are calculated relative to the vertices of the surrounding simplex.

(iv) An interpolated value is computed for each grid point using the barycentric coordinates as weights and the RGB values at the three vertices of the enclosing simplex, performing linear interpolation.

After applying IPM to the images from all 8 cameras, the next step was to stitch them together. The IPM provides the position of each pixel in the captured images, represented as $\mathbf{X}_v = [X_v, Y_v, Z_v, 1]^\intercal$ in $\mathcal{F}_v$. To facilitate transformation into an image of arbitrary resolution, normalization of the points $\mathbf{X}_v$ was performed, mapping them the range $[-1, 1]$. The final step involved multiplying $\hat{\mathbf{X}}_v$ by the "intrinsic" matrix $\mathbf{K}_{IPM}$, where res_x and res_y denote the resolution of I_{IPM}, to obtain $\mathbf{u}_{IPM}$.

$$\mathbf{u}_{IPM} = \begin{bmatrix} u_{IPM} \\ v_{IPM} \end{bmatrix} = \mathbf{K}_{IPM}\hat{\mathbf{X}}_v = \begin{bmatrix} \frac{res_x}{2} - 1 & 0 & \frac{res_x}{2} \\ 0 & -\frac{res_y}{2} - 1 & \frac{res_y}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{X}_v \\ \hat{Y}_v \\ 1 \end{bmatrix} \tag{3}$$

Given the cameras' FOV and position, some of the cameras observe the same area. Due to different viewing angles and local light conditions, this overlap creates unwanted noise in the IPM image and was therefore removed.

4. Code optimization

To function in real-time operations, the system must be able to generate bird's eye view images at a rate no slower than the provided input images. For *milliAmpere 2*, the cameras capture images at a rate of $5Hz$ which imposes a run-time requirement of less than $200ms$ per IPM image. This section describes the equipment used during the first phase of the research and the optimization steps and methods used with their respective run-times. For detailed descriptions, see [9].

4.1. Code optimization setup

The computer used for this phase had `Ubuntu 20.04`, `ROS 2 Foxy Fitzroy` and an Intel Core i7-8700 CPU with a base clock frequency of 3.20GHz, max clock frequency of 4.60GHz, 6 cores, and 12 threads. The run-time data presented in section 4.2 were captured by using `cProfiler` on `main()` function. The presented run-times are the average run-time of the function `make_BEW()`, which produces the IPM image. The timer starts after the `make_BEW()`-function has received all 8 undistorted images, and ends when the IPM image is made. Overhead, such as undistortion, is excluded.

4.2. Optimization steps and results

The initial performance benchmark for this paper was bilinear interpolation with a resolution of $3000px \times 3000px$, and the average run-time was $34s$. During the initial phase of the research, multiple interpolation methods were tested and benchmarked.

The most important methods and their run-time are presented in Figure 3 where the first method was based on linear interpolation with a resolution of $3000px \times 3000px$. Method 2 reduced the resolution to $1500px \times 1500px$ and switched to nearest neighbor interpolation. In method 3, the center of the image was filled with black pixels and removed from the interpolation process. Method 4 introduced Delaunay triangulation which was combined with parallel processing for method 5. Finally, method 6 introduced the custom interpolation method described in section

3.1. This resulted in a run-time reduction from the original 34.0s using method 1 to 0.6s for method 6.

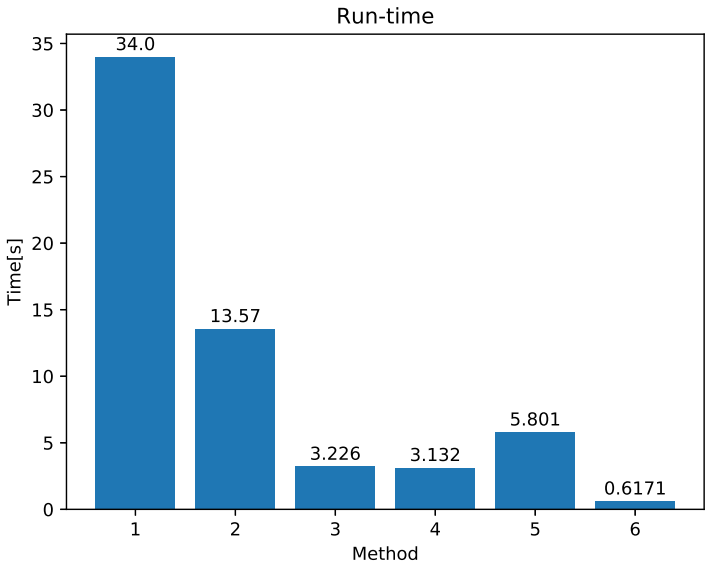


Figure 3. Run-times of the tested interpolation methods.

Additional optimization was then conducted to achieve the required run-time of < 200ms with results shown in Figure 4. From the baseline, step 1 moved the image masking and distance checking to program initialization. Step 2 removed overlapping pixels present in two cameras from the process. In step 3 the final 8th camera was introduced while step 4 optimized the indexing function in the interpolation. Step 5 did the same for the function that extracts relevant areas from individual images. Finally, step 6 introduced in-place processing for certain calculations, removing the previous copy-based methods. This reduced the run-time from the original 617ms to 179ms yielding real-time performance. Step 7 was then performed as an additional test with resolution reduced from $1500px \times 1500px$ to $1100px \times 1100px$, further reducing run-time to 130ms.

5. Experimental setup

The ferry *milliAmpere 2* was used to capture images, and all 8 *FLIR Blackfly S 50-S5C* optical cameras were used. The computer was set up with an AMD EPYC 7313P "MILAN" CPU with a base clock frequency of 3.0GHz, max clock frequency of 3.7GHz, 16 cores and 32 threads [14]. The code was running in a *Docker* container using `Ubuntu 22.04` and `ROS2 Humble Hawksbill`. The experiment took place on May 10, 2023, in the canal between Fosenkaia and Ravnkloa in Trondheim, Norway, and the system was tested with two operators. The experiment involved crossing the canal twice with four dockings and had two main goals: assessing the real-time performance of the system onboard *milliAmpere 2* and evaluating its usefulness for the operator during docking operations.

6. Run-time results

The run-times of the visualization system during live testing are presented in Figures 5 and 6 using a resolution of $1100px \times 1100px$ for the bird's eye view. The main density of data points

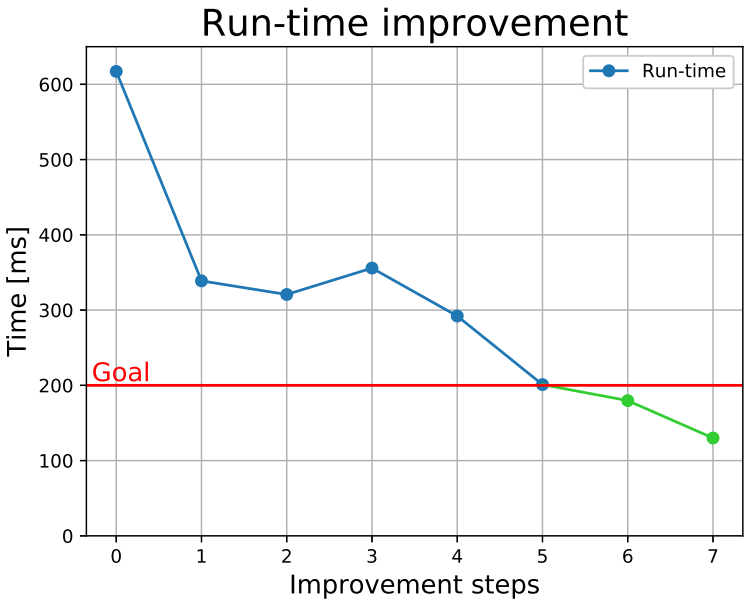


Figure 4. Run-times for the additional optimization steps based on method 6.

is between 200ms and 265ms, with less than 1% of the data points above 275ms. The average run-time was 233.2ms which is shown as a red dashed line in the figures.

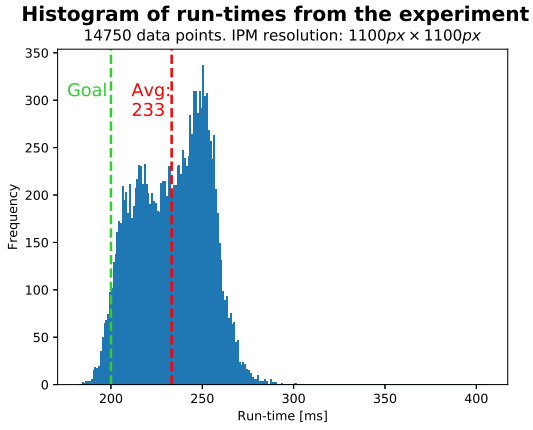


Figure 5. Run-time histogram from live experiment. Each bin represents a millisecond.

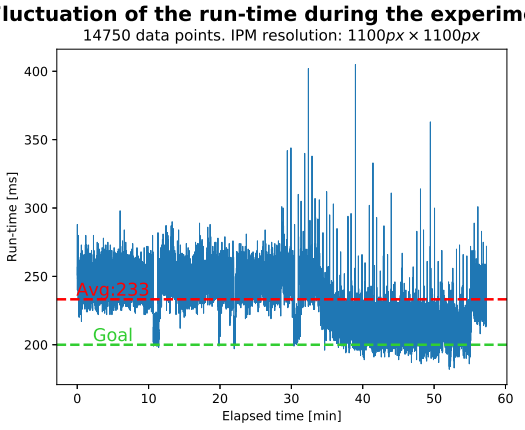


Figure 6. Line-graph showing run-times for the entire experiment.

For the sake of fast and convenient prototyping, the system was implemented in `Python`. At the onset of the project, we assumed that a run-time of less than 200ms could be achieved, which would be sufficient for the purpose of using the birds-eye view as a navigation aid. In the pre-recorded setting this was indeed achieved. In the real-time experiment, this did not hold, as shown by the run-time histogram in Figure 5. There are several reasons for this. The CPU clock speed of *milliAmpere 2* is about 20% slower than the computer used for code optimization and the integration of the code in the onboard ROS2 system resulted in increased overhead.

Analysis of Figure 5 reveals two peaks, centered around 250ms and 215ms. This is supported by the data in Figure 6, which shows two distinct periods. Prior to the 35-minute mark, the

average run-time is around 250ms, followed by a noticeable reduction to approximately 215ms thereafter. The exact reasons for these fluctuations and changes are challenging to determine due to limited data on concurrent systems and potential data variations. However, it is clear that the performance of the system is significantly affected by concurrent processes and programs. For further details, see [9].

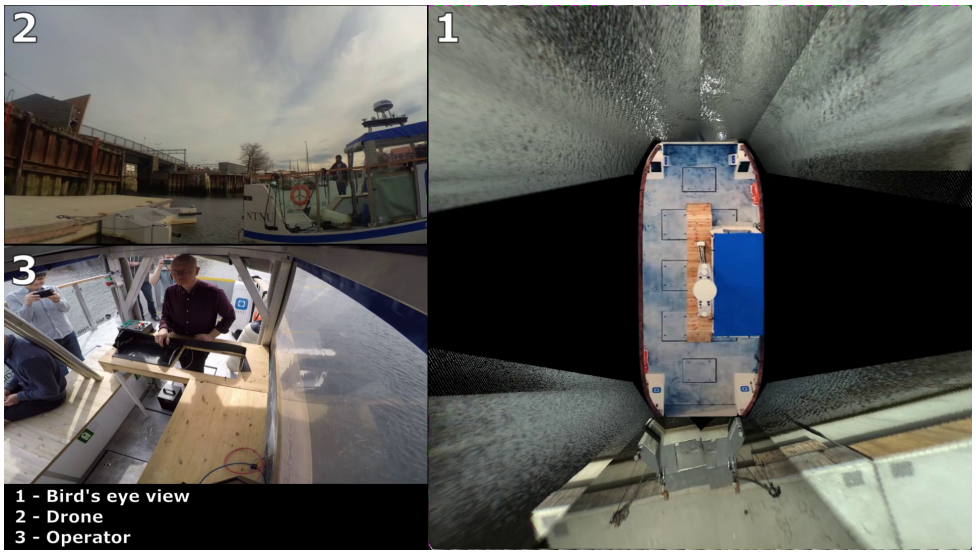


Figure 7. Bird's-eye view (1), drone footage (2), and operator footage (3). Bird's-eye view is shown on a monitor in front of operator.

7. Visual accuracy and operator usefulness

Accurate camera calibration, both intrinsic and extrinsic, is crucial for optimal bird's eye view images. Unfortunately, due to time constraints, precise calibration was not achieved during the research. Instead, the extrinsic parameters relied on the CAD model of *milliAmpere 2*. The consequences of this inaccurate calibration are evident in various bird's eye view examples. In Figure 7 for instance, the straight lines on the dock appear discontinuous, exhibiting abrupt steps when crossing the edges of the cameras' FOV. The position and size of the inserted ferry visualization were determined manually which resulted in slight inaccuracies in scale. Nonetheless, operator feedback emphasized that it was a very positive addition to the system.

Despite the run-time being on average above the real-time goal of 200ms, the system was still considered fast enough to provide useful aid. The slower speed at which the ferry navigates during docking mitigated the impact of the slightly longer run-time. The operator's questionnaire feedback indicated that the refresh rate of the bird's eye view was deemed high enough but that higher refresh rates would be beneficial in addition to camera synchronization. A video from the experiment with the birds-eye view and operator footage is available. [1]

8. Conclusion and future work

In this work, a 360-degree bird's eye view system written in `Python` for *milliAmpere 2* was presented. The run-time goal was not fully achieved on the ferry due to CPU speed and overhead in the autonomy system. Despite some shortcomings, the operators reported that the system provided useful additional assistance during the docking process.

[1] https://www.youtube.com/watch?v=aN2jtBRpnqk

Further refinement should however be conducted before the system is deployed for actual operations. Rewriting the code in a faster programming language like `C` or `C++` can significantly improve run-time. Additionally, leveraging a dedicated GPU for interpolation tasks can further optimize the system. Enhancing the visual aspect involves automatic camera calibration and synchronized cameras for improved accuracy. Adding distance lines at regular intervals can enhance the utility of the bird's eye view for operators, aiding their spatial awareness.

CRediT authorship contribution statement

Mathias Thoresen Paasche: Conceptualization, Methodology, Software, Formal analysis, Investigation, Data Curation, Writing - Original Draft, Visualization, Project administration.
Øystein Kaarstad Helgesen: Conceptualization, Resources, Writing - Review & Editing, Supervision, Project administration.
Edmund Førland Brekke: Conceptualization, Resources, Writing - Review & Editing, Supervision, Project administration.

References

[1] Muad A, Hussain A, Samad S, Mustaffa M and Majlis B 2004 *IEEE Region 10 Conference TENCON* vol A pp 207–210 Vol. 1
[2] Oliveira M, Santos V and Sappa A D 2015 *Information Fusion* **24** 108–121 ISSN 15662535
[3] Bruls T, Porav H, Kunze L and Newman P 2019 *arXiv.org* ISSN 2331-8422
[4] Prakash C D, Akhbari F and Karam L J 2019 *Robotics and autonomous systems* **114** 172–186 ISSN 0921-8890
[5] Plenge-Feidenhans'l M K and Blanke M 2021 *IFAC PapersOnLine* vol 54 (Elsevier Ltd) pp 30–36 ISSN 2405-8963
[6] Griesser D, Dold D, Umlauf G and Franz M O 2020 *Global Oceans: Singapore – U.S. Gulf Coast* pp 1–8
[7] Bannore V 2009 *Iterative-Interpolation Super-Resolution Image Reconstruction: A Computationally Efficient Technique* (*Studies in Computational Intelligence* vol 195) (Berlin, Heidelberg: Springer Berlin / Heidelberg) ISBN 9783642003844
[8] Lehmann T, Gonner C and Spitzer K 1999 *IEEE transactions on medical imaging* **18** 1049–1075 ISSN 0278-0062
[9] Paasche M T 2023 *Real-time 360 degrees view for the operator of milliAmpere2* Master's thesis
[10] Hartley R and Zisserman A 2004 *Multiple View Geometry in Computer Vision* 2nd ed (Cambridge University Press)
[11] Burger W and Burge M J 2016 *Digital Image Processing: An Algorithmic Introduction Using Java* 2nd ed (Springer Publishing Company, Incorporated) ISBN 1447166833
[12] Helgesen Ø K, Stahl A and Brekke E F 2023 *IEEE Access* **11** 30340–30359
[13] The SciPy community 2023 scipy.interpolate.linearndinterpolator — scipy v1.10.1 manual URL `https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.LinearNDInterpolator.html`
[14] AMD 2020 Amd epyc™ 7313p URL `https://www.amd.com/en/products/cpu/amd-epyc-7313p`

A.2 Operator questionaire

Questioner to the operators of milliAmpere2 during the experiment

Your name: *Øystein Kaarstad Helgesen*

How was it to dock the ferry with the aid of the bird's eye view (BEW) images on an external monitor?

It felt like the system provided useful additional assistance for positioning the ferry. Without this system the ramps had to be partially lowered to create better visibility from the operator station. With the system this was not required.

Did the BEW images make it easier to dock the ferry?

Yes, the images provided a better overview of the ferry position and distance relative to the dock than what was observable from the operator station.

Was the refresh rate of the BEW images high enough to not have a significantly negative impact when using the BEW images to dock the ferry?

Yes, but higher refresh rates would be beneficial. Synchronization of the cameras would also help to ensure consistency.

Did the image see far enough (in distance) in front (or rare) of the ferry, or should it have displayed objects that were further away?

For docking assistance the distance felt perfect.

How limiting was it to not be able to see directly in front of the ferry (the blind zones of the cameras), where the ferry contacted the dock?

Not that limiting, the ferry could be positioned quite accurately before the cameras lost view of the dock. Thrusting in from this position was not difficualt.

Was the inserted image of milliAmpere2 in the middle of the BEW image a positive, neutral, or negative addition to the system?

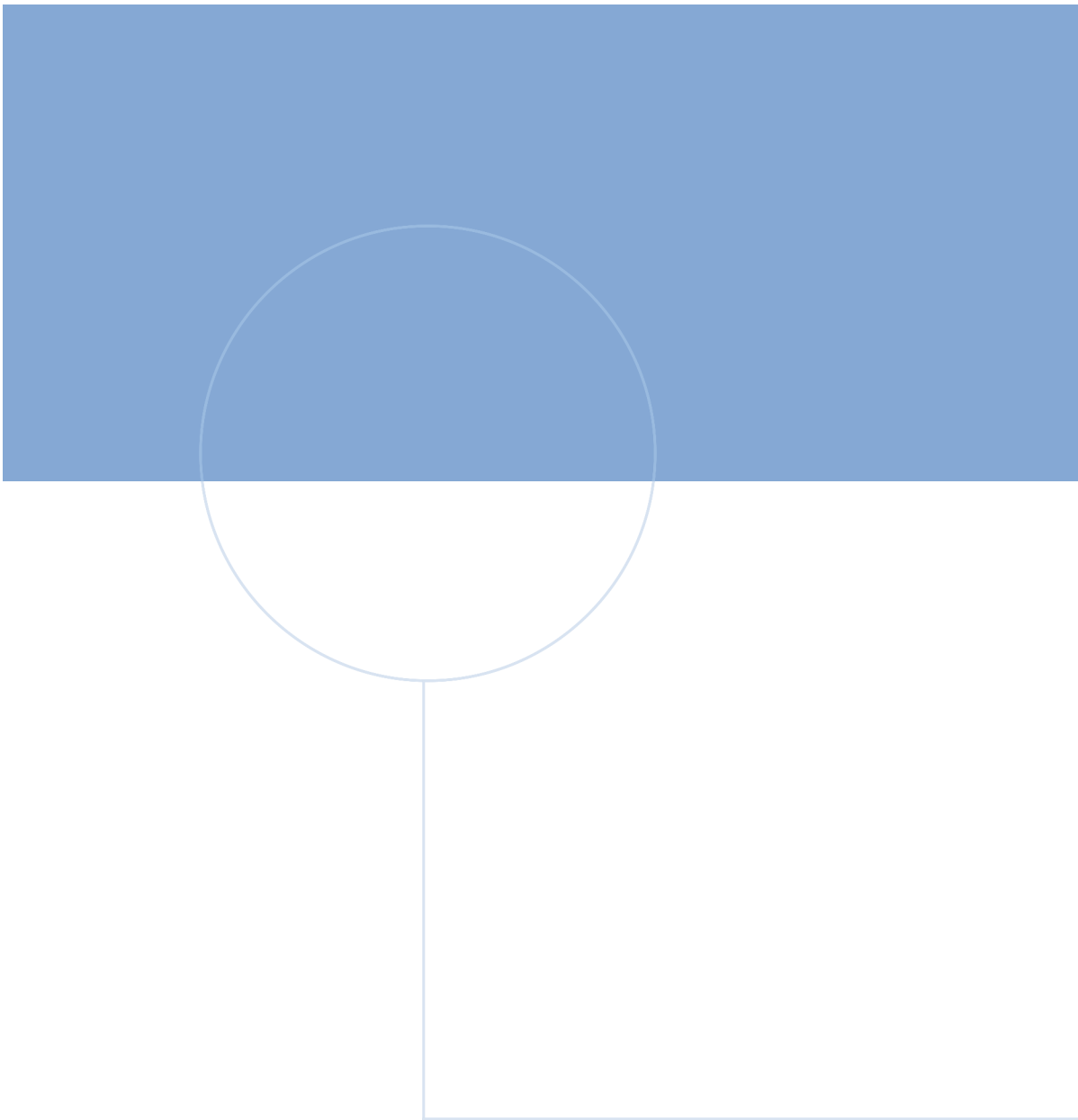
Very positive, it gave a much better sense of where in the image the ferry was located. The insterted image was slightly too large which made it a bit more difficult to estimate the distance to the dock.

Were the miss alignments of lines and objects across the different camera FOVs small enough to not have a significantly negative impact when using the BEW images to dock the ferry?

It was slightly harder to dock in the docking adapter on the Fosenkaia side which is much narrower, a better calibration would have made it appear more consistent. Otherwise no issues.

In your opinion, would it have helped to have lines indicating distance on the BEW images? For example, red lines at 1m, 2m, and 5m in front (or rare) of the ferry.

Yes, very useful, especially if the inserted image of milliAmpere could be scaled down to its correct size. This would make it much easier to estimate the distance to the dock in the final phase of docking.



NTNU

Norwegian University of
Science and Technology