NTNU
Norwegian University of
Science and Technology

Rolls-Royce

Specialization project

Computer Vision in Ferry Docking Operation

Tord Sjeggestad Bjørnsen

Submission date: December 18, 2018
Supervised by: Dr. Annette Stahl, ITK
Dr. Øivind Kåre Kjerstad, Rolls Royce Marine

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Project Description

Introduction

This specialization project is done in collaboration with Rolls-Royce Marine. The project will be continued into a master thesis in the spring of 2019.

Main Goal

The main goal of the project is to explore methods that can be used in a positioning system on board typical "roll-on roll-off" (ro-ro) ferries as a redundant measurement to the GNSS system. The positioning system should be based as much as possible on the sensors that already exist on the ferries, with a special focus on the camera system. The system is a step on the way in the development of fully autonomous ferries, and it is supposed to be used in auto-docking situations.

Secondary Objectives

- Literature study on monocular computer vision methods, as well as pose estimation algorithms.
- Study the OpenCV documentation.
- Find intrinsic parameters using calibration techniques and the OpenCV library.
- Test different pose estimation algorithms from the OpenCV library.
- Present some suggestions for future work on the problem presented in the main goal.

Tasks

- Calibration of a monocular camera.
- Evaluate the reprojection error of the calibration.
- Use feature detection to find markers in images.
- Test different pose estimation algorithms.
- Evaluate the reprojection error, relative pose results and speed of pose estimation.
- Evaluate all results and develop some suggestions for future work.

Preface

This specialization project concludes my 9 th. semester at NTNU in Trondheim. The work on the project has been time consuming and difficult at times, but I have learned a lot and I look forward to continuing this work in my thesis next spring.

I want to thank my advisor Dr. Annette Stahl for helpful discussions and help during the semester. I also want to thank Rolls-Royce Marine and especially Dr. Øivind Kåre Kjerstad for contributing with helpful information and guidance before and during the project.

Abstract

Autonomy is becoming increasingly popular in the maritime industry today. Developing autonomous ferries for commercial use can lower the cost of running, building and maintaining ferries. This project is a study on potential methods that can be used in a computer vision system, as a redundant positioning system both on board ferries today and on future autonomous ferries. This report presents a study done on methods and systems that can be used in such a system.

The report presents different methods and algorithms. There are two areas within computer vision that has been given attention in this report, and that is pose estimation and camera calibration. Throughout the report we present methods, theories and give an implementation of both camera calibration and pose estimation testing. Within pose estimation we present six different methods and compare them against each other.

The pose estimation are tested on 18 different images, and the results show that there are big differences in performance. The results from this testing will be presented in different diagrams in the last part of the report.

The report finishes with a discussion on the results and a small section containing useful suggestions for future work on the subject.

Contents

List of Figures

Chapter 1

Introduction

This specialization project is done in collaboration with Rolls Royce Marine (hereby referenced as RRM). It is a concept study on using monocular computer vision (CV) in a supplementary system on board ocean vessels to identify and measure the distance to the quay where the vessel is supposed to dock. The measurement will be redundant with the GNSS signals, radar or other sensor data already in use on a lot of ships. Based on the results from the CV measurements it is possible to estimate the position and attitude of the vessel.

The GNSS positioning system used on board the vessels are quite good and is used as the main positioning system on board. It does however have a small uncertainty, which is okay in transit when there is nothing to bump into. In the docking situation however, the vessel can hit the dockside or car ramp with too much force if the error in the GNSS position estimate is large enough. By applying a positioning system based on computer vision in the last couple of meters, it is possible to optimize the position of the ferry based on both systems.

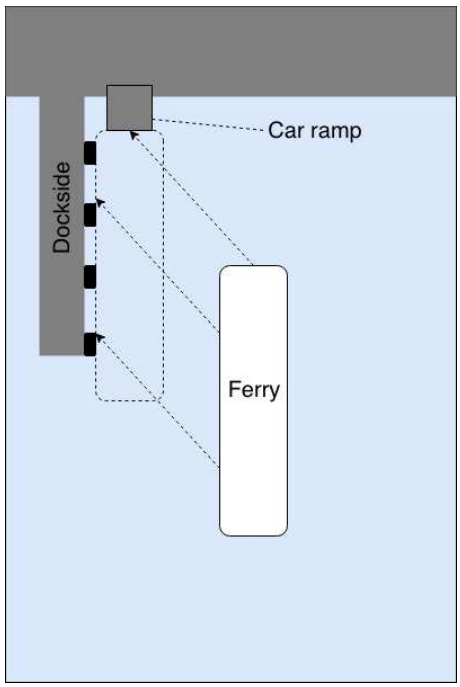

Figure 1.1: Illustration showing the docking situation from a top down perspective.

1.1 Motivation

The main goal of this project is to explore methods and systems that can be used to develop a new positioning system. The focus will be on the camera systems already installed on typical "roll-on roll-off" (ro-ro) ferries, as the Fjord1 vessel *MF Gloppefjord* in fig. 1.2. The cameras installed on board ferries are located both on the sides and in the front, and are directed towards the area between the ferry and the quay, as it is illustrated in fig. 1.3. The computer vision positioning system is

Figure 1.2: Fjord1 ferry *MF Gloppefjord* in transit between Anda and Lote. Image courtesy of Fjord1 [10]

supposed to be a redundant system to other positioning systems that are based on GNSS signals and radar.

RRM officially started working on autonomous ship solutions back in 2014. The development has been fast, and already in 2016 they sold their first *auto-crossing* system to Fjord1 [26]. The first commercial operation with the system was on *MF Gloppefjord*, a ferry that service the transit between Anda and Lote on the western coast of Norway. This system controls the ferry autonomously between the quays, and the captain only takes the control when it's docking to and leaving the quay. The system is vital for the ferry *MF Gloppefjord* as the ferry is one of the first electric ferries in the world, and the "auto-crossing" system makes the power usage as efficient as possible so that the charging time at the quay is as low as possible. In order to make the entire transit autonomous, the system needs to be supplemented with a system for autonomous docking. The first version of this system has now been developed, and on the third of December 2018 RRM demonstrated the system on the ferry *Falco*, see fig. 1.5, which made the worlds first fully autonomous ferry transit in Åbo Finland [31]. Although the demonstration was a success, there is still a lot of development that needs to be done before the system is ready for commercial use, which is estimated to be

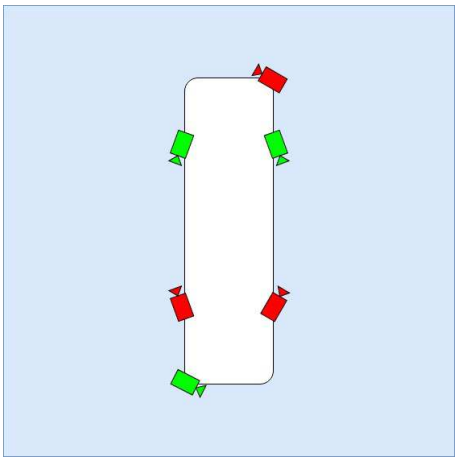

Figure 1.3: Illustration showing the camera locations on board a typical ro-ro ferry from a top down perspective. The color code represents what cameras that will be used in combination, they are based on what direction the ferry sails.

around 2021.

Although the "auto-docking" system has already been demonstrated as a working system, there are still a lot of challenges remaining. In order to make such a system interesting to shipowners it has to be economical enough so that they can make a profit on the investment. This is especially difficult on older ships with older equipment and propulsion systems that are difficult to configure. On the old vessels it is not desirable to make a lot of new installations, because it might be difficult to make it work together with the older equipment. In other words it is desirable to use the equipment already installed on the ferries in the system, so that it in theory just will be a software installation needed to make the ship ready for autonomous service.

Another big challenge for the use of such systems is redundancy. In order to operate completely autonomously the ferry has to be able to handle unexpected events such as a radar malfunction or loss of signal/error in the GNSS signals. This is especially a big challenge in the Norwegian fjords where the satellite signals tend to mirror off the mountain sides so that the GNSS gets confused. This behavior is called multipath, see fig. 1.4, because the GNSS receiver can receive multiple signals from the same source, one direct and other reflected off mountains or other objects[23].

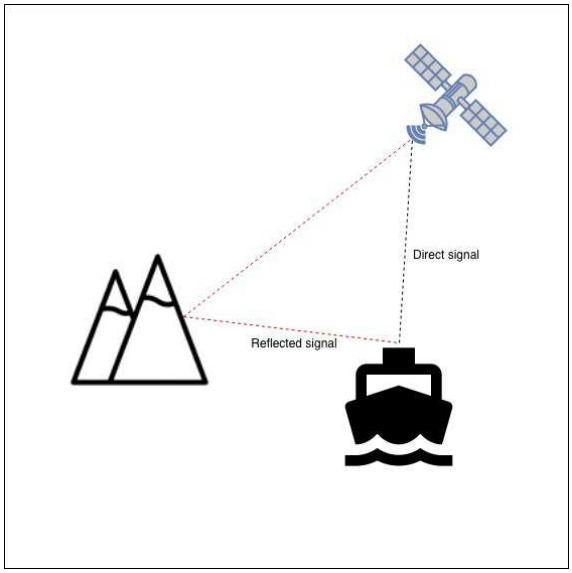

Figure 1.4: Illustration showing the GNSS Multipath phenomenon.

A system that is completely redundant to the traditional positioning systems on board the vessel is the camera system. The ferry has six cameras mounted around the ferry and they are used by the captain as aid when the ferry docks to the quay. Four of the cameras are mounted on the two sides of the ferry (two on each side) and the last two are mounted on the front and on the back, as indicated by fig. 1.3. As the ferry docks there are only two cameras in use; the front camera and one of the side cameras depending on what side of the ferry the camera is located. The red and green cameras in the figure indicates what cameras that are used together in a docking situation.

1.2 Contribution

This project explores methods that are needed for estimating the position of a typical ro-ro ferry. The project includes a simple camera calibration algorithm based on the *OpenCV* documentation. QR detection algorithms used in the project are based on the *ZBar* python library, which has built-in functions for detection of

Figure 1.5: The Finnish ferry *Falco* during the world's first fully autonomous ferry transit. Image courtesy of *The Engineer* [8].

QR- and bar-codes. Different PnP solvers are already implemented as functions in *OpenCV*, and are tested in this project. The results from the testing are evaluated, and the methods compared against each other.

A considerable literature study was done within the computer vision field. Different calibration algorithms was studied, but only one of them made it into the report. Choosing the right calibration algorithm is difficult because they all have different qualities, but the final choice was in this case the same as the one that is implemented in *OpenCV*. The reasoning behind this is that the implementation implementation in the library is already optimized and it would be very complicated to implement an algorithm from scratch.

The literature study also involved a considerable amount of pose estimation methods. A lot of different PnP algorithms were looked into, and six of them made it into the report. The ones that are presented in the report are all implemented in *OpenCV*, which was very useful when they were tested. The reason for not implementing any other methods is that the built-in functions gave very good results, and are very well optimized in the *OpenCV* library.

All implementations in this project has been written and rewritten many times. The implementations are inspired by examples and the documentation of the *OpenCV* library, in addition to other sources on the Internet. The final versions are based on a class structure, and the main reason for this is that it would be simpler to rewrite the code to C++ if it was needed.

Some parts of the project did not make it to the report. There was done a lot of research on visual odometry, and a lot of code was written to run a positioning system based on video input instead of still images. It turned out to be more difficult than expected to implement, and in the end it was decided that there was not enough time to finish the work. There can be many reasons to why the odometry did not work, but mainly the QR detection time was an issue. Also the

runtime of the rest of the systems was very high, and so no successful test of the system was done. It is suggested to replace the current camera with a camera with lower resolution, or use software to downscale the images although this might lead to longer runtime than just changing the camera. Rewriting the code to C++ might also solve the runtime issues, and this is presented as a suggestion for future work in the last chapter of the report.

RRM has contributed with information on the main goal of the project. In addition they have given detailed descriptions on the docking situation, camera setup and on ferry and quay configurations. They have also made a simulator available if the project got as far as to test any systems on a ferry simulator.

1.3 Structure

We have now introduced the project and some of its contents. We follow up with chapter 2 which contains related work within computer vision, and some introductions to autonomy and its history. Following comes the theory in chapter 3 where some mathematics behind the concepts presented in the related work will come. After that we will present the implementations we have worked on during the project, both the hardware used and the software in chapter 4. The results from the testing follows with a lot results from testing four different image sets in chapter 5. In chapter 6 we will discuss the results from the chapter before, and make some conclusions based on the work. We will also present some suggestions for future work on the subject. In the end we have placed the appendices, where all code, images and some runtime results are attached.

Chapter 2

Related work

This chapter contains literature references to related work within autonomous systems, sensors and computer vision. Some of the theory of the contents within this chapter will be presented in detail in chapter 3, which is the basis for the implementations presented in chapter 4.

2.1 Autonomous systems

We start this chapter by introducing a couple of autonomous systems and their history, in order to get some context within the field of autonomy. A system can be defined "autonomous" if it "can change it's behavior in response to unanticipated events during operation" [40]. The first autonomous systems were purely based on feedback control, which is the theory that founded the field of cybernetics. One of these early applications was done at *Johns Hopkins University Applied Physics Lab* in the 1960's when they made *the Beast* which was a mobile automaton that drove around in the hallways of the university. When the power got low it searched for a black socket in the hallway and was able to plug itself in to charge. The robot was purely cybernetic and did not use a computer, it was purely driven by a lot of transistors controlling analogue voltages, and it used sonar and photocell optics to navigate [37].

Autonomy has come a long way since the *Beast* was introduced. The most popular area for the use of autonomy is within the automobile industry. Companies like Über, Tesla, Waymo and many more are all working towards introducing the first autonomous car for commercial sale. It is however difficult to make this a reality as the security requirements are set very high, especially after Über had a fatal accident in March of 2018 [14]. Autonomy is also a hot topic within the maritime industry, and companies like RRM, Volvo Penta, Kongsberg Maritime, Wärtsila and others are all working on bringing autonomy to the marine market. Volvo Penta sticks out as the only company that has a main focus on the private yacht sector, and demonstrated a working automatic docking system in July of 2018 [25]. This was however just a prototype of the system, and it will take them a long time to make it ready for commercial sale.

2.1.1 Autonomous Vessels

As it is mentioned in the paragraph above there are many companies working on autonomy in the maritime industry, including RRM. Kongsberg Maritime, see fig. 2.1, is working in a collaboration with DNV-GL to launch the worlds first fully autonomous cargo ship, *Yara Birkeland*, by the first quarter of 2020 [30]. As mentioned above Wärtsila is also working towards autonomous ships, and as a step along the way they successfully tested remote controlling a ship in the north sea from San Diego [7] in August of 2017.

Depending on the type of vessel, the transit and service usually consists of three different modes: un-docking, transit and docking. Since un-docking basically is the inverse of a docking operation it is possible to reduce the problem into two modes, docking and transit. RRM developed and tested an autonomous transit system called *auto-crossing* that's able to control a ship between two endpoints without human interaction. This system is installed on *MF Gloppefjord*, see fig. 1.2, which transits between Anda and Lote, and RRM has made a deal with Fjord1 to install the system on 13 more ferries as well [26].

Figure 2.1: Yara Birkeland. Image courtesy of Kongsberg Maritime [22]

2.1.2 Autonomous Docking

One of the most precarious parts of a ferry transit is the docking operation. Docking refers to the last operation of the transit when a vessel is slowly approaching and finally connecting to the quay. Then the vessel stays in this position by thrusting against the dockside, connecting a rope or by some other connection between the vessel and the quay. A docking operation for a typical ro-ro ferry is illustrated in the introduction, see fig. 1.1, but other vessels may have a different approach to the operation. An aspect of the docking operation that makes it difficult is that it must hit the quay with as little force as possible in order to avoid damage on passengers, cargo, dockside, car ramp and ship. In order to achieve this, the ferry needs to be maneuvered precisely and slowly towards the quay until it hits, and is able to load off its cargo and passengers. Such an operation is difficult even for trained captains, and making a system that can achieve this autonomously will require sensors with very high precision and good redundancy to handle unexpected events.

The high precision requirement for an autonomous docking system makes it complicated to develop. As it was mentioned above, there are a lot of companies

working on automatic docking systems, and a lot of them has already demonstrated working prototypes. Volvo Penta's system relies heavily on sensors mounted on the quay, and cannot be used unless the dockside is configured for automatic docking [25]. Wärtsila has also made an automatic docking system, and in contrast to Volvo Penta's system this system does not require installation of sensors on the dock. Instead this system relies heavily on the the ships DP system based on GNSS and IMUs, making it very reliant on satellites to avoid drift in the position estimate [39]. As mentioned earlier, these systems are prototypes, and they still rely on a captain to take over if something unexpected should happen, which is why they are called automatic and not autonomous systems.

We have now presented some background on autonomous systems as well as some challenges in the development of autonomous docking systems. In order to develop the positioning system described in the problem description, we need to introduce and explore what positioning really is, as well as how position is parametrized.

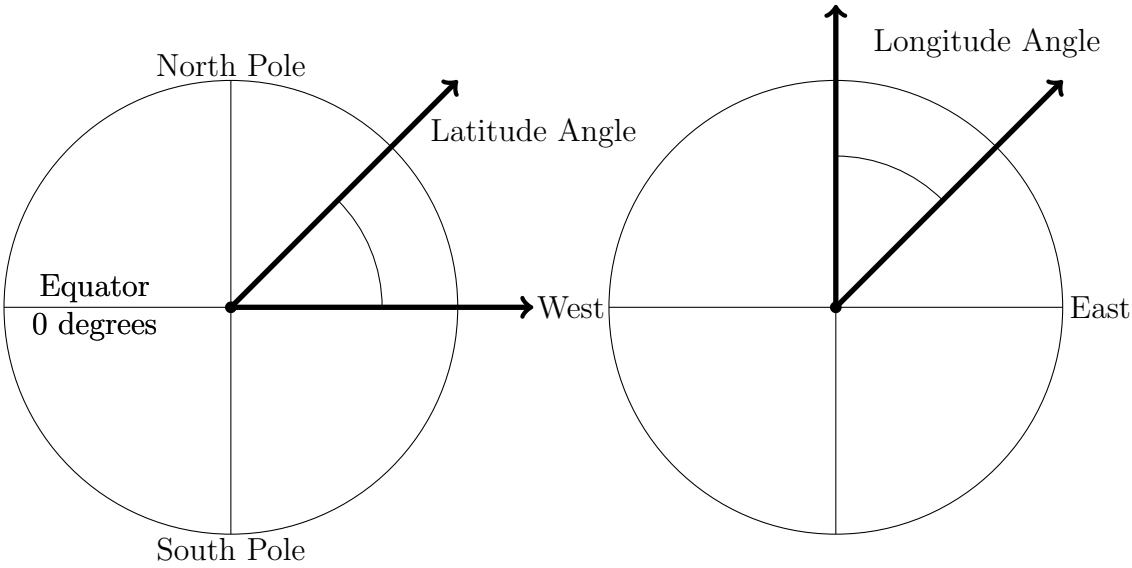

Figure 2.2: Illustration showing the earth coordinate system as it is described by latitude and longitude, where the circle represents the earth from different different perspectives. The height is the distance to the point from the earth center.

2.2 Positioning

Positioning is the process of determining and describing the position of an object with respect to a coordinate system. When navigating the earth it is most common to use an "earth-centered earth-fixed" (ECEF) coordinate system that rotates with the earth. A proper positioning system is able to estimate the position of an object based on sensor inputs. There are many ways of accomplishing this, but one of the oldest is the magnetic compass which was first used in the Han dynasty in China more than 2000 years ago [21]. The compass is not able to find the position on the

earth on its own, but it exploits the magnetic field around the earth to find the four four cardinal directions (north, south, east and west). Another more modern approach is to use satellites and a GNSS system that triangulate the position on the earth based on the information it receives from a minimum of three satellites [5]. The position in earth coordinates is given in latitude, longitude and height, as the illustration in fig. 2.2 shows.

In this project we assume that the position of the quay in earth coordinates is known. The goal is then to use CV methods to detect the quay using some kind of feature detector and estimate the position of the ferry based on that information and some known model of the quay. After that the position of the quay in the earth coordinate system and the position of the vessel in the quay coordinate system are both known. Then the position of the vessel in the earth frame can be found by a simple coordinate transformation, and a transformation from Cartesian coordinates to longitude, latitude and height.

The goal of the project is to explore methods that can be used to develop a positioning system as discussed above, mainly based on already installed sensors on board the vessels with a focus on the camera system. We will now explore different options of hardware that can be used in such a system, including an introduction of sensors and some potential markers.

2.3 Hardware

Hardware used on board vessels has to be certified to both national and international safety and regulatory requirements. These requirements are set by different agencies, one example is the European Maritime Safety Agency (EMSA) [29]. Some of the requirements are about redundancy, water resistance and material standards, which all make the system more robust, but it also make sensors and equipment more expensive. The installation will also be more complex because important systems as GNSS, Radar, etc. is required to have separated redundant cabling.

In this section different hardware components will be discussed. First we will discuss the sensors that we can use, and afterwards different markers that can be installed on the dockside will be presented.

2.3.1 Sensors

Making a positioning system requires to decide on what sensor inputs to use. Although a complete system probably will use a sensor fusion solution, it is interesting to see what the options are. For this project, the positioning system is supposed to be as redundant to the GNSS system as possible, hence it has to use other sensors to collect data. Choosing a sensor is not an easy task, and it is important to find the right balance between price and precision. We will now present a couple of different sensors both already installed and some good alternatives that could be installed.

Radar (Radio detection and ranging)

The radar is a very common sensor that is used a lot in the marine industry. The sensor is active and transmits bursts of radio waves in all directions by rotating 360 degrees. Then the receiver on the radar measures the time until a reflected signal returns to the sensor and calculates the distance to it. The radar can use this information to map the area around it quite well, and it is also used to detect other vessels in the vicinity.

An advantage of the radar is that it's very robust to disturbances. It can filter out small objects like birds, leafs and snowfall by setting a threshold on how big the returning radio waves has to be in order to be classified as an object. Since the radar is an active sensor it is also not very dependent on the weather conditions, and will work well even in changing light conditions and fog. In addition the radar is already installed on most vessels, so there will be no extra installation cost to the shipowner if it was used in a new positioning system.

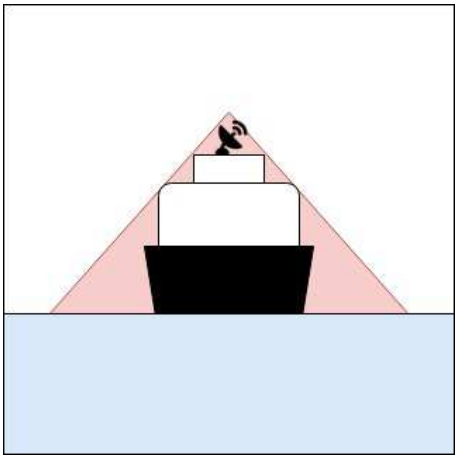

Figure 2.3: Illustration showing the ferry from the front, where the radar "dead zone" in the red triangle.

A ro-ro ferry as seen from the front is illustrated in fig. 2.3 where the red triangle represents the "dead zone" in the radar view. The radio waves from the radar will reflect of all surfaces, and is therefore dependent on a optical line of sight. Unfortunately this means that an object located behind another will not be detected by the radar. The area that cannot be seen is called the radar "dead zone", and a more detailed description of the phenomenon can be found in [43], where the curvature of the earth makes a big "dead zone".

The radar would be a great sensor to use for positioning because of its robustness against disturbances and changing weather conditions. The "dead zone" is however a big problem if the system is going to be used in docking situations. The reason for this is that the area between the quay and the vessel is inside the "dead zone", making it difficult to find the position of the ferry relative to the quay. One solution to this problem would be to install radar reflectors on specific locations on the dockside that does not fall in the radar "dead zone" during docking. The reflectors has to be quite high to avoid it, and the installation might be difficult as the reflectors must be completely still even in high winds. In addition, the system will not be able to work on quays that does not have the reflectors installed.

Camera

The camera is a passive sensor that use the available light in a scene to record an image of the environment in front of it. By taking a lot of images in series, the camera can also record a video. The sensor data, called images, can be combined with geometrical calculations and known information about the model to estimate

the position of the camera with respect to the scene. The geometrical information in the image is extracted using feature detection where some known features in the image is detected. This can be done either by a classical feature detector or by a more modern approach using neural networks.

Just like the radar, the cameras are already installed on typical ro-ro ferries as it is described in fig. 1.3. An image of the ferry *MF Hjørundfjord* with the side cameras is depicted in fig. 2.4. These cameras are used as support for the captain when he cannot see over the edges of the vessel, a bit like the "dead zone" of the radar. Because the cameras are already installed there is no need for installation of sensors on board the vessel, and the cost of installation will be low. Another advantage is that if there is a need for installing hardware on the quay (some kind of marker), it will be simpler than for the radar as the cameras can see inside the "dead zone" of the radar. The maintenance of such a marker might however be more demanding as it will need to get rid of snow covering it. The cameras are also very easy to replace if they do not meet the required accuracy for the system.

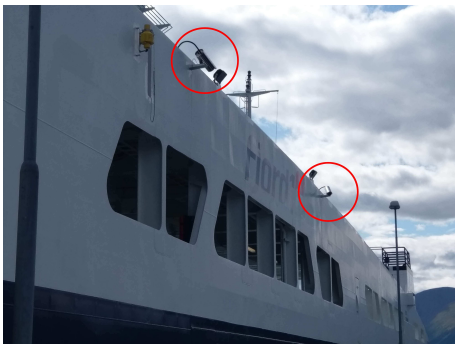

Figure 2.4: An image of the side of the Fjord1 ferry *MF Hjørundfjord* showing the location of the side cameras.

Cameras are highly dependent on the light conditions, and fog or heavy snow can cause problems for the sensor. A solution to this might be to use active markers such as red fog-lights or something similar. Today there are already installed a single red fog-light on the end of most docksides, which are used by the captain as navigation when the weather is bad. Such an active marker can also get quite warm, and hence melt the snow that covers it in the winter.

Lidar (Light detection and ranging)

The lidar has become a very popular active sensor within autonomy. It has been especially well received in the automobile industry in the development of autonomous cars. It works almost like a radar, but instead of sending radio waves it sends bursts of light. Both the radar and the lidar rotates 360 degrees to map the entire area around them, but the range of the radar is better than the lidar. The light from the sensor reflects of any surface, and the resulting output is a point cloud of the area around it. Just like the radar it uses the time of flight to determine the distance to each of the points in the point cloud.

This sensor has a lot of similar characteristics with the radar. It works in almost all weather conditions, and by some software post processing even thick snow can be eliminated from the point cloud [19]. There is no need for extra installations on the quay because the light rays from the lidar will reflect from any surface.

This is however a bigger investment to install on board the vessel. The lidar sensor(s) has to be installed on the ship at appropriate positions to catch the entire area around the vessel, including the radar "dead zone". This means that there

has to be installed multiple lidars (at least one in the front and one in the back), to cover the entire area around the vessel. It is also important that the lidars are certified for maritime use, which make them quite a bit more expensive than regular lidars.

IR camera

As a final proposal we present a thermographic (or infrared) camera. The imaging technique works much like a regular camera, but instead of measuring the light intensity, it measures the heat radiation transmitted from the objects in the scene. Because it measures a physical property of the scene without sending any signal out, this is a passive sensor just like the regular camera.

In order to use the geographic information in the images to estimate the position and attitude of the ferry, there has to be some ind of markers on the quay. The markers has to have a higher temperature than the surroundings in order to stick out from the scene, and be detected by the IR camera. Since the temperature usually is changing a lot between winter and summer they also have to be thermostatically controlled. The markers can however keep off snow by having a positive temperature in the winter.

In addition to installing the markers, the camera must also be installed. Since the camera is not a regular sensor on a vessel, it needs to be installed on board and facing both towards the front and the back of the vessel. This camera also has to be certified for maritime use, which will make it more expensive than a regular IR camera.

Comparison

All sensors have both positive and challenging qualities. It is difficult to decide what sensor to use in a positioning system, but the project has a goal of making the system as easy and low cost as possible to make it attractive to shipowners even with older vessels. This is why the regular camera is a good choice as sensor in the system. It is already installed on most vessels, and it is a cheap sensor to upgrade if higher accuracy is required.

Some of the sensors presented will require some kind of marker or feature point as positioning reference. The camera is chosen as main sensor, which means there has to be some visual points on the quay that can be located by the CV system. This can either be some kind of marker, or features that already exist s on the quay. In the next section we present three possible markers for the CV system.

2.3.2 Markers

In order to locate the quay there has to be some markers or features on the quay for the CV system to detect, see section 2.4.1. It might of course be possible to use some features that already exists on the quay as markers but that will be difficult to generalize as different quays have different features. In the main front camera of the ferry, a good placement for he markers could be on both sides of the car ramp. Then the pose of the ship could be estimated relatively to the car ramp, and

subsequently the latitude, longitude, height and attitude. We will now present a few different possible markers that can be used in a CV based positioning system.

QR Codes

QR codes are distinct markers that can contain different types of data. It can contain almost anything, Internet urls, wi-fi-configurations, plain text, and much more, the marker in fig. 2.5 show the qr code containing the word "Left". In other words it is possible to store information about the position of the marker inside the marker itself. This is useful because then the marker can be installed on different places on different quays, without hard coding the position of all markers on different quays in the CV system.

Figure 2.5: A QR code with the text-string "Left".

One difficulty with the QR codes is that they need to be well maintained so that it does not loose information. If the code is changed or distorted the positioning system will not be able to use it. Also if there is snow it needs to be brushed off, and in fog the camera might not be able to detect the code.

Light

An active marker that could be used is a light. Similar to how a light house works, it can be located even in difficult weather conditions such as in fog or thick snow. It might also be able to melt snow that lands on top of it, making maintenance just changing the bulb/LED if it stops working. One weakness with using lights is that they do not have the same capability to store information as the QR code has. Differentiating it from other light sources is also a challenge, but by making the light turn on and off at a certain frequency it might be possible to solve this problem. By making different lights have different frequencies it might also be possible to differentiate the markers from each other. Another option is to have different colored lights indicating different points.

This is actually a marker that is already present on the edge of most docksides. These lights are used as a reference point by captains when they are docking during difficult weather and cannot see the quay properly. A possible light that can be used is the "FAA L-810", which is actually an obstruction light for airports shown in fig. 2.6.

Custom

As a fourth option we suggest making a custom marker. There are lots of different possibilities, but one option is to combine the two suggestions above. Another option is to make a simpler coded marker than the QR code, for example an $n \times n$ binary matrix (a simpler version of the QR code). The possibilities are endless, but

in order to work well during fog and thick snow it might be smart to use some kind of active marker.

We have now presented different hardware that can be used in a system for estimation of pose during a docking operation, including different sensors and markers. Choosing the correct hardware is difficult, especially since the system is supposed to work on old ferries as well as new, but also in different light- and weather conditions. The system also has to be as affordable as possible and with small maintenance costs. In addition to this RRM has expressed an interest in using the camera systems on board the ferries, and so this will be the sensor that will be used in the positioning system. Now that the hardware has been discussed we will present some software that is relevant to the system.

Figure 2.6: Obstruction light for airports. Image courtesy of Flight Light Inc. [17]

2.4 Software

In this software section of the report we will go through software techniques that can be used in a positioning system for a ferry. We start by introducing computer vision, an interdisciplinary field based on using computer systems and algorithms on images and videos to extract useful information about the scene in front of the camera. We begin this section by introducing computer vision and useful parts of the field for this project, then we will introduce *OpenCV* which is a well known programming library for computer vision.

2.4.1 Computer Vision

Computer vision is a huge field spanning multiple areas within science and technology. The goal in this project is to use computer vision methods to estimate the position. This includes detecting the quay, estimating the pose of the camera with respect to the quay and subsequently the pose of the ferry in the ECEF coordinate system. In order to use the geometrical information in an image we have to know the distortion parameters in the image as well as the intrinsic parameters of the camera. These parameters can be estimated using a calibration algorithm which is discussed next.

Camera Calibration

In order to extract correct information from the images taken by the camera, it has to be calibrated. Camera calibration is a procedure that can estimate the intrinsic parameters of a camera, see section 3.1.2, as well as the parameters of the distortion model, see section 3.1.3.

There exists lots of different methods and algorithms for camera calibration, including [1], [9] and [41] to mention a few. The methods can roughly be classified into two different categories; ”self-calibration” and ”photogrammetric calibration”

[45]. "Self-calibration" algorithms does not use any kind of calibration object, instead it is based on multiple images of a static scene. They assume that the camera has the same settings in all images, then the displacement between the images can be used to estimate the intrinsic and extrinsic parameters of the camera. In this project we will however use a "photogrammetric calibration" method instead, where we use a calibration object. In most cases this is a chessboard pattern as shown in fig. 3.4, but this varies depending on the method. The methods use the known information it has about the chessboard, the size of the squares and that the points are coplanar, to estimate the intrinsic parameters.

In this project we will focus on the method proposed by Z. Zhang in [45]. This is a "photogrammetric calibration" technique using a chessboard as seen in fig. 3.4, which uses a chessboard with known square sizes as calibration object. The algorithm estimates the intrinsic and extrinsic parameters of the camera, as well as the parameters of the radial distortion.

After calibrating the camera and finding an estimate of all the parameters of the camera model, see section 3.1.2, we have found the relationship between 3D object points in the scene and points in the image plane. In order to locate image points corresponding to feature points in the scene, a procedure called feature detection can be used.

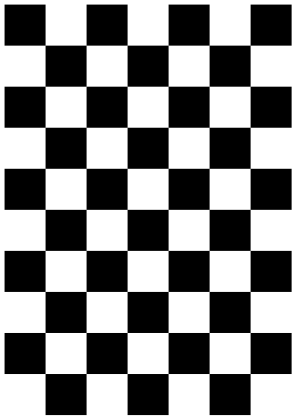

Figure 2.7: Calibration chessboard.

Feature detection

Features are pieces of information that are useful for solving tasks for certain applications. In computer vision context features are defined as an easily re-identifiable element of the scene. There are three typical features used in computer vision; edges, corners and ridges.

Edges are defined as an amount of points where there is a boundary between two image regions. According to R. Szelinski in [33] edges refer to "boundaries of objects" and "other kinds of edges correspond to shadow boundaries or crease edges, where surface orientation changes rapidly". Detecting edges in an image can be done by inspecting the image gradients, and there are many different algorithms for achieving this. The Canny Edge Detector was proposed by J. Canny in [2] in 1986, and is probably the most known edge detector today. The algorithm suggested by Canny is a multistage algorithm which takes advantage of the image gradients in an image. First it removes noise, this is because edge detection in general is susceptible to noise. This can be done by a applying a simple Gaussian filter. Next the algorithm finds the intensity gradient of the image. Lastly the algorithm scan the gradient image using non-maximum suppression, removing any pixels that may not constitute an edge. By performing a hysteresis thresholding on the image where the gradients below a lower threshold are discarded, the gradients above are classified as edges, and the ones in between are classified based on their connectivity with edges or non-edges. This leaves a binary edge image as the one in fig. 2.8.

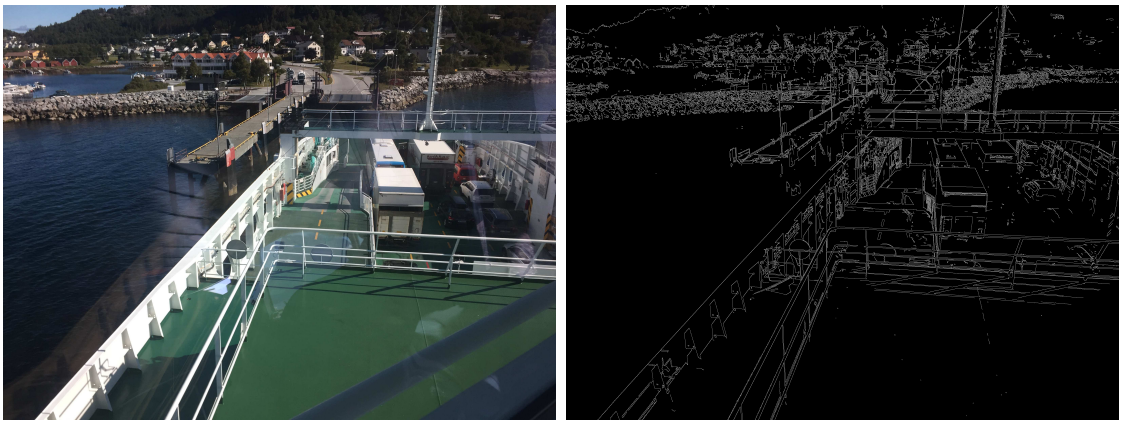

Figure 2.8: Canny edge detection used on an image taken from bridge of a ferry.

Ridges and edges are often used synonymously but are not actually the same. While edges are sharp borders between areas of high an low intensity, ridges are thin lines that are brighter or darker than the surrounding neighborhood. Thus the resulting binary image of the canny edges detector in fig. 2.8 is actually an image of ridges representing the edges in the original image. Both ridges and edges has a common problem, they both consists of multiple points and it is difficult to define a specific point along them. Hence they are not very useful in an operation to find feature point correspondences between two images or image and scene.

The most popular feature in computer vision is corners. They are uniquely defined as a single point, and are easy to find in multiple images of the same scene. This make them especially useful when point correspondences are to be found in different images, i.e. for epipolar geometry or feature tracking. Corners are easy to identify by evaluating the gradient, whatever direction one moves from the corner the gradient will change by a large amount. This is in contrast to edges or ridges where the gradient will stay more or less the same if we move along them. There are a lot of different corner detection algorithms, some of the most known are Movarec's corner detector, Harris' corner detector and "SUSAN" corner detector. All three algorithms work by the same basic principle of evaluating the gradients, but the most popular and well known is the Harris corner detector [13].

In order to perform point feature matching where the same feature is found in different images, we need some kind of descriptor that is unique for all points, in order to avoid false point matching. ORB (Oriented FAST and Rotated BRIEF) is a good descriptor based on BRIEF. It is very efficient and a good open source and free alternative to SURF and SIFT descriptors that both are protected by copyright [27]. The ORB descriptor use a FAST detector for detecting corners, augmented with a pyramid scheme for scale awareness and a Corner Harris filter for removing edges in the detected key-points. It uses a descriptor for the key-points called rBRIEF (rotated BRIEF), which is very efficient and robust, to describe the detected points. The ORB descriptor is very fast and outperforms both SIFT and SURF, and is therefore a good choice for real-time systems.

Feature detection is a big area in computer vision, and choosing detection method and the descriptor depends on the application; what is going to be de-

tected, and what information about the feature is important. In the event of QR code detection, as discussed in section 2.3.2, it might be useful to implement a pre-made detector, like the ones that are found in the library called "ZBar" [44].

By using feature detection it is possible to find either a marker or a specific feature in an image. In a positioning system the points will have a known longitude, latitude and height, so when the ferry has a known position relative to the quay a simple coordinate transformation gives the position in the world. In order to use the points that are detected in the image together with the points on the quay, and subsequently estimate the pose of the ferry, we move into another area within computer vision called pose estimation.

Pose Estimation

Pose estimation is the process of computing the pose of an object with a known structure, relative to a calibrated camera [32]. The problem of estimating the pose is a classical problem in computer vision, but also in photogrammetry where it is called "resection". The pose of an object refers to the position and orientation relative to a coordinate frame, often the camera coordinate frame.

Estimating the pose is done by connecting features between object and image. Features such as lines and corners that will be used for pose estimation has to be in a known position in the earth coordinate system, and they have to be easily identifiable in an image. Then the pose of the camera can be estimated by connecting the object points and the image points using the camera matrix containing the intrinsic parameters of the camera.

Initially the model has 6 degrees of freedom (6DOF) and the goal of the pose estimation is to reduce it to 0DOF. Introducing one single point correspondence between the image and the object reduces the degree to 4DOF, and introducing two reduces it to 2DOF. By introducing three point correspondences reduces the degree to 0DOF, which means that in order to estimate the pose of the camera a minimum of three points has to be known. However as R. Holt and A. Netravali show in [16] knowing three corresponding points does not necessarily give a unique pose, but can actually give up to four. They found that in order to generally get a unique solution four non-coplanar (Coplanar points are points that do not occupy the same surface or linear plane) points has to be known.

Estimating the pose is a nonlinear problem that must be solved with a nonlinear estimator. The solvers often minimize the mean-square error to find the best pose to fit the point correspondences (hence estimator). Assuming that the camera model is a perspective camera model (all image rays go through the center of projection) the problem is called the "Perspective-n-Point" (PnP) problem. If the camera model is non-perspective then it is called "Non-Perspective-n-Point" (NPnP), and it has to be solved with a NPnP solver like the one suggested by C. Chen and W. Chang in [4].

PnP Problem

The PnP, problem refers to the pose estimation problem based on n known point correspondences between image and scene. Each known point in the scene is de-

tected in the image plane, and the problem assumes that the position of the points in the scene is known or at least that the points have known locations with respect to each other. The problem itself is to find the rotation and translation of the camera so that the n scene points projected into the image plane, as will be presented in section 3.2.2, is as close to the detected image point as possible. This is called evaluating the reprojection error, and can be used both to validate calibration and pose estimation. This kind of pose estimation is called model based because the model, or relationship between the points in the scene, is known. The problem is nonlinear, and there has been proposed a lot of different solvers for it. Some of the possible solutions are presented below, and will be explained in more detail in section 3.3.1.

One solution to the problem is the Levenberg-Marquardt (LM) optimization algorithm presented by M. Transtrum, B. Machta and J. Sethna in [38]. This solver is iterative combining advantages of both Gradient-descent and Gauss-Newton methods. The LM steps are linear combinations of the Gradient-descent and Gauss-Newton steps based on an adaptive rule. The adaptive rule use Gradient-descent dominated steps until it reaches a "canyon" in the function, and then Gauss-Newton dominated steps along the canyon. The function that the algorithm optimizes over is the reprojection error which it wants to minimize, and the algorithm continues until it reaches convergence. When it has converged, it has found the pose that minimizes the reprojection error of the image points.

The CASSC algorithm solves the minimal PnP problem of estimating the pose based on three known control points. As it is described above, the P3P problem can have four solutions. The algorithm to solve the problem is presented by X.-S. Gao, X.-R. Hou, J. Tang and H.-F. Cheng in [12], which is an algebraic approach using Wu-Ritt's decomposition providing the first solution to the problem. Another P3P algorithm (AP3P) is presented by T. Ke and S. Roumeliotis in [18], and this algorithm is also algebraic. The main difference compared to CASSC is that AP3P calculates the attitude, and subsequently the camera position, instead of all parameters at once. This algorithm is supposedly faster and more accurate than other P3P algorithms because it avoids unnecessary calculations.

A solver for the general PnP problem is the Direct Least-Squares (DLS) method as presented by J. Hesch and S. Roumeliotis in [15]. The method "formulate a nonlinear least-squares cost function whose optimality conditions constitute a system of three third-order polynomials"[15]. Further it determines the roots of the system analytically by employing a multiplication matrix. The roots of the system give the minimum of the least-squares, and consequently the pose of the camera without iterating or making an initial guess of the parameters. One big advantage of this solver is the scalability, which is a result of the order of the polynomial that has to be solved is independent of the number of points, n.

There are many other solutions to the problem with advantages and disadvantages. Another non-iterative solver is the EPNP solver presented by V. Lepetit, F. Moreno-Noguer and P. Fua in [20], which is fast and require $n \geq 4$ points. The UPNP solver presented by A. Penate-Sanchez, J. Andrade-Cetto and F. Moreno-Noguer in [28] is another non-iterative solver. This is inspired by the EPNP solver, but in addition to estimate the pose, it also estimates the camera focal length, which

is useful if the camera is poorly calibrated.

Now that we have introduced the CV techniques, we need tools to implement and use the techniques on the computer. The next section introduces a computer vision library called *OpenCV* that come with a lot of already implemented functions.

2.4.2 OpenCV

OpenCV is a free open source computer vision library. It is written in optimized C/C++ but has interfaces for use in C++, python and Java. It is an efficient library that can utilize multi-core processing, which is essential in real time computer vision systems [35]. According to the developers, the library contains over 2500 built-in and optimized algorithms, and new algorithms are implemented frequently. The newest version of the library was released on the 20 th. of November this year (2018), but in this project a previous version will be used (v. 3.4.0). How the library is utilized in this project will be described in detail in chapter 4, and the main documentation of the language can be found in [36].

Chapter 3

Theory

This chapter presents the theory explored in the project. The theory is used as a basis for the systems implemented and described in section 4.2. The basis for most of the theory comes from the work presented in chapter 2.

3.1 Camera

A camera is a very simple passive sensor to capture visual information as it is described in section 2.3.1. It collects the light in a scene over a short time interval, and represent the information either as analogue film or digital as bytes. The camera mainly consist of three components; an imaging sensor to capture the information, a lens to collect the most amount of light and direct it towards the imaging sensor and a camera housing to make sure the imaging sensor is not influenced by other light sources. The imaging sensor can be either analogue or digital, but the digital sensor is by far the most used nowadays.

Figure 3.1: Image of Canon EOS 7D.

3.1.1 Digital Images

There are two types of digital imaging sensors; charge coupled device (CCD), and complementary metal oxide semiconductor (CMOS). Both of them consist of a grid of pixels that measures the amount of light in the scene, and they differ only in the way the light intensity is measured and converted into an imaging file [3]. How they work in detail is not important in terms of computer vision methods, and will therefore not be presented. Each pixel in the grid usually consist of three parts that measures the intensity of red, green and blue light, unless it is a greyscale sensor which only measures the total light intensity.

The image is saved as bytes of information about the color intensity. Each pixel gets a value between 0 and 255, which means that each color pixel requires one byte

to store the information. Assuming that the sensor is RGB means that one image takes 3 bytes times the number of pixels in the sensor to store the information of the image.

3.1.2 Camera Model

In order to use the information captured by the camera, a good model of the camera is needed. There has been developed a lot of different models for cameras with different usages. The two main groups of camera models are the central camera models and the non-central camera models, which differ only in the sense that the central camera models has an optical center and the non-central does not[32]. These are again divided into global, local and discrete models, where the parameters of the global models influence the entire image plane and the parameters of the local models affect different portions of the image. The discrete models sample the back-projection function at different points, which means that the model has a lot of parameters; one per sampled location.

The global central models are the most used because they are simple to use and describes the cameras in an intuitive way. There are many models in this category including affine models, two-plane model, models for Slit cameras, etc.., but the most common model is the perspective camera model. This model is based on the simple pinhole cameras that where used in the early days of photography, where the pinhole is in the center of projection. In a regular pinhole camera the imaging sensor would be behind the principle point, meaning that the image is projected upside down on the imaging plane. However to simplify the calculations based on the model, the imaging plane is moved in front of the principal point so that the image is the right way compared to the scene in front of the camera.

So what model should you choose? According to section 3.6 on page 84 in [32] "The general answer to these questions is: it depends...", but if the camera that is used is a regular perspective camera the easiest and most accurate is the perspective camera model, popularly called the pinhole camera model.

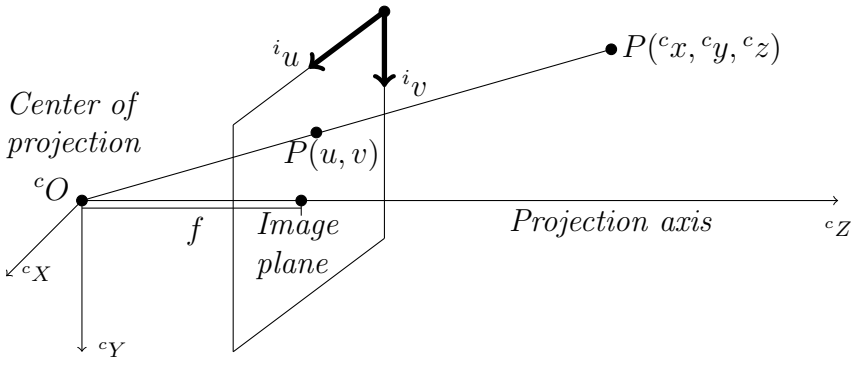

Figure 3.2: Illustration of the perspective camera model.

The perspective camera model is illustrated in fig. 3.2. The model assumes a linear relationship between a 3D point in the scene and a corresponding 2D point in

the image coordinate system. This linear relationship is dependent on five parameters called the intrinsic parameters. The parameters are f_x and f_y representing the focal length in pixel dimensions in the horizontal and vertical direction respectively (assuming that the pixels are square the focal lengths are equal). s is a skew parameter that can represent non-rectangular pixels or synchronization errors in the image read-out, assuming square pixels gives $s = 0$. The last two parameters, x_0 and y_0, represents the position of the principal point in the pixel coordinate system, see fig. 3.2, which represents the origin of the normalized image coordinate system. The intrinsic parameters will later on be presented in the camera calibration matrix in section 3.2.2, where the linear relationship between the scene and the image plane is presented and are found using a camera calibration method as presented in section 3.1.4.

3.1.3 Distortion model

The perspective camera model presented in section 3.1.2 is usually called the ideal perspective camera model because it assumes no distortions in the image. Distortions are errors with respect to the real world, and are divided into many different categories including geometrical, chromatic aberration, spherical aberration etc. The distortions can be difficult to model precisely, but can be approximated by polynomials.

In this project the geometrical distortions are the most important, which consists of radial and tangential parts. Both parts are illustrated in fig. 3.3.

The radial distortion can be modeled as

$$\delta r = k_1 r^3 + k_2 r^5 + k_3 r^7 + \cdots \tag{3.1}$$

where r is the distance to the image point from the principal point, $r = \sqrt{x^2 + y^2}$. The tangential distortion is usually smaller, and a bit more complicated to model [6], but is included in eq. (3.4). After removing the the distortions the new image coordinates are given as

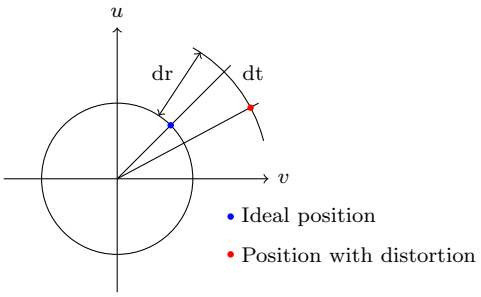

Figure 3.3: Illustration of radial (dr) and tangential (dt) distortions [42] on a single pixel.

$$u^d = u + \delta_u \tag{3.2}$$

$$v^d = v + \delta_v \tag{3.3}$$

and the displacement due to geometrical distortion is given by [6]

$$\begin{bmatrix} \delta_u \\ \delta_v \end{bmatrix} = \underbrace{\begin{bmatrix} u(k_1 r^2 + k_2 r^4 + k_3 r^6 + \cdots) \\ v(k_1 r^2 + k_2 r^4 + k_3 r^6 + \cdots) \end{bmatrix}}_{\text{radial distortion}} + \underbrace{\begin{bmatrix} 2p_1 uv + p_2(r^2 + 2u^2) \\ p_1(r^2 + 2v^2) + 2p_2 uv \end{bmatrix}}_{\text{tangential distortion}} \tag{3.4}$$

The parameters, k_1, k_2, k_3, p_1, p_2 in the distortion model, are called the distortion coefficients and can be estimated by a camera calibration algorithm.

3.1.4 Camera Calibration

A short introduction to camera calibration can be found in section 2.4.1. We will focus on the method proposed by Z. Zhang to find the intrinsic parameters as well as the extrinsic parameters of the calibration images, see fig. 3.4. The method also finds an estimate of the parameters describing the radial distortion of the image. We will now present the method in this text, but for the interested reader, the method is described in depth in [45].

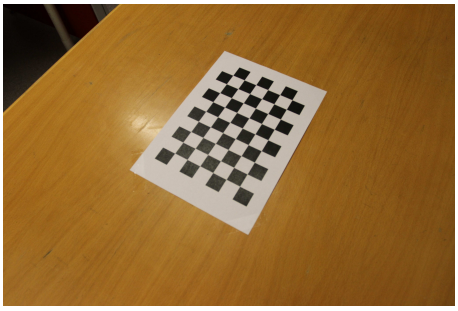

Figure 3.4: A typical calibration image of a chessboard.

Assuming the mathematical interpretation of the perspective camera model, as it is described in eq. (3.35) in section 3.2.2, and the augmented points ${}^{c}\tilde{\mathbf{P}} = ({}^{c}X, {}^{c}Y, {}^{c}Z, 1)$ in the camera frame and ${}^{i}\tilde{\mathbf{P}} = ({}^{i}u, {}^{i}v, 1)$ in the image plane (original points are ${}^{c}\mathbf{P} = ({}^{c}X, {}^{c}Y, {}^{c}Z)$ and ${}^{i}\mathbf{P} = ({}^{i}u, {}^{i}v)$). From section 2.2 in [45] we know that if we assume the model plane at ${}^{c}Z = 0$ we can relate a model point ${}^{c}\tilde{\mathbf{P}} = ({}^{c}X, {}^{c}Y, 1)$ with an image point with a 3×3 homography $\mathbf{H}$:

$$s \begin{bmatrix} {}^{i}u \\ {}^{i}v \\ 1 \end{bmatrix} = \mathbf{C} \begin{bmatrix} r_1 & r_2 & r_3 & t \end{bmatrix} \begin{bmatrix} {}^{c}X \\ {}^{c}Y \\ 0 \\ 1 \end{bmatrix} = \mathbf{C} \begin{bmatrix} r_1 & r_2 & t \end{bmatrix} \begin{bmatrix} {}^{c}X \\ {}^{c}Y \\ 1 \end{bmatrix} = \mathbf{H} \begin{bmatrix} {}^{c}X \\ {}^{c}Y \\ 1 \end{bmatrix} \tag{3.5}$$

where r_i represents the columns of the rotation matrix of the extrinsic parameters, and s is a scaling factor. From eq. (3.5) we have $\mathbf{H} = \mathbf{C}[r_1 \ r_2 \ t]$, where $\mathbf{H}$ is the homography. Appendix A in [45] explains one technique for estimating $\mathbf{H}$ based on maximum likelihood criterion. Minimizing the function

$$\sum_i ({}^{i}\mathbf{P}_i - {}^{i}\hat{\mathbf{P}}_i)^{\top} \Lambda_{{}^{i}\mathbf{P}_i}^{-1} ({}^{i}\mathbf{P}_i - {}^{i}\hat{\mathbf{P}}_i) \tag{3.6}$$

where

$${}^{i}\hat{\mathbf{P}}_i = \frac{1}{\bar{h}_3^{\top} {}^{c}\mathbf{P}_i} \begin{bmatrix} \bar{h}_1^{\top} {}^{c}\mathbf{P}_i \\ \bar{h}_2^{\top} {}^{c}\mathbf{P}_i \end{bmatrix} \tag{3.7}$$

will yield the maximum likelihood estimation of $\mathbf{H}$. $\bar{h}_i$ is in this context the i-th row of the homography $\mathbf{H}$. $\Lambda_{{}^{i}\mathbf{P}_i}$ is a covariance matrix of Gaussian white noise with 0 mean which is presumed to be corrupting the image point ${}^{i}\mathbf{P}_i = [{}^{i}u \ \ {}^{i}v]^{\top}$, this is usually assumed $\Lambda_{{}^{i}\mathbf{P}_i} = \sigma^2 \mathbf{I}_{2\times 2}$. Because of this assumption the minimization problem become a nonlinear minimization problem

$$\min_{\mathbf{H}} \sum_i \left\| {}^{i}\mathbf{P} - {}^{i}\hat{\mathbf{P}} \right\|^2 \tag{3.8}$$

This nonlinear minimization must be solved with a nonlinear solver, like the Levenberg-Marquardt Algorithm which is presented for solving the PnP-problem in section 3.3.1.

Based on the new estimate of the homography we can find constraints on the intrinsic parameters. By $\mathbf{H} = [h_1 \;\; h_2 \;\; h_3]$ and the following equation

$$[h_1 \;\; h_2 \;\; h_3] = \lambda \mathbf{C}[r_1 \;\; r_2 \;\; t] \tag{3.9}$$

with λ as an arbitrary scalar, the method in [45] finds the following equations (equations (3) and (4) in the paper)

$$h_1^\top (\mathbf{C}^\top)^{-1}\mathbf{C}^{-1}h_2 = 0 \tag{3.10}$$

$$h_1^\top (\mathbf{C}^\top)^{-1}\mathbf{C}^{-1}h_1 = h_2^\top (\mathbf{C}^\top)^{-1}\mathbf{C}^{-1}h_2 \tag{3.11}$$

where the fact that r_1 and r_2 are orthonormal is exploited.

Now that we have found the intrinsic constraints we can move over to the actual calibration itself. First we find a closed-form solution (see section 3.1 in [45]) by letting

$$B = (\mathbf{C}^\top)^{-1}\mathbf{C}^{-1} = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{12} & B_{22} & B_{23} \\ B_{13} & B_{23} & B_{33} \end{bmatrix} \tag{3.12}$$

$$= \begin{bmatrix} \frac{1}{f_x^2} & -\frac{s}{f_x^2 f_y} & \frac{y_0 s - x_0 f_y}{f_x^2 f_y} \\ -\frac{s}{f_x^2 f_y} & \frac{s^2}{f_x^2 f_y^2} + \frac{a}{f_y^2} & -\frac{s(y_0 s - x_0 f_y)}{f_x^2 f_y^2} - \frac{y_0}{f_y^2} \\ \frac{y_0 s - x_0 f_y}{f_x^2 f_y} & -\frac{s(y_0 s - x_0 f_y)}{f_x^2 f_y^2} - \frac{y_0}{f_y^2} & \frac{(y_0 s - x_0 f_y)^2}{f_x^2 f_y^2} + \frac{y_0^2}{f_y^2} + 1 \end{bmatrix} \tag{3.13}$$

where $\mathbf{B}$ is a symmetric matrix which can be defined by a 6D vector as

$$\mathbf{b} = \begin{bmatrix} B_{11} & B_{12} & B_{22} & B_{13} & B_{23} & B_{33} \end{bmatrix} \tag{3.14}$$

By following the suggestion of the paper and denoting the i-th column vector of $\mathbf{H}$ as $\mathbf{h}_i = [h_{i1}, h_{i2}, h_{i3}]^\top$ we get

$$\mathbf{h}_i^\top \mathbf{B} \mathbf{h}_j = \mathbf{v}_{ij}^\top \mathbf{b} \tag{3.15}$$

Now the constraints in eq. (3.10) and 3.11 can be rewritten as eq. (8) in [45]:

$$\begin{bmatrix} \mathbf{v}_{12}^\top \\ (\mathbf{v}_{11} - \mathbf{v}_{22})^\top \end{bmatrix} \mathbf{b} = 0 \tag{3.16}$$

Doing this with n images of the model in question gives the general equation

$$\mathbf{V}\mathbf{b} = 0 \tag{3.17}$$

$\mathbf{V}$ is a $2n \times 6$ matrix, so to solve this linear equation we need $n \geq 3$ images to estimate all five intrinsic parameters of the camera. If we assume that the skew-parameter is zero ($s = 0$), we can solve the equations with $n = 2$ images. If we

also assume that x_0 and y_0 are known, then we can actually estimate the last two parameters based only on a single, $n = 1$, image. When $\mathbf{b}$ is estimated, the camera matrix can be estimated. Then it is possible to find the extrinsic parameters of the camera (with respect to the calibration chessboard) by eq. (3.5).

Now that the intrinsic parameters are known the next step is to find the distortion parameters described in section 3.1.3. The method proposed by Z. Zhang in [45] only accounts for the radial distortion, because it dominates the tangential distortion, an assumption based on the paper by D. C. Brown [1]. The radial distortion in [45] is modeled by two parameters so that the distorted image coordinates are given by the following equations

$$\breve{u} = u + (u - u_0)[k_1(x^2 + y^2) + k_2(x^2 + y^2)^2] \tag{3.18}$$

$$\breve{v} = v + (v - v_0)[k_1(x^2 + y^2) + k_2(x^2 + y^2)^2] \tag{3.19}$$

where (u, v) are the ideal distortion-free image coordinates. In order to estimate the distortion parameters, the equations can be rewritten to matrix form as

$$\underbrace{\begin{bmatrix} (u - u_0)(x^2 + y^2) & (u - u_0)(x^2 + y^2)^2 \\ (v - v_0)(x^2 + y^2) & (v - v_0)(x^2 + y^2)^2 \end{bmatrix}}_{\mathbf{D}} \underbrace{\begin{bmatrix} k_1 \\ k_2 \end{bmatrix}}_{\mathbf{k}} = \underbrace{\begin{bmatrix} \breve{u} - u \\ \breve{v} - v \end{bmatrix}}_{\mathbf{d}} \tag{3.20}$$

The radial distortion parameters can now be estimated by taking the pseudo-inverse of $\mathbf{D}$ so that the equation for the parameters become [45]

$$\mathbf{k} = (\mathbf{D}^\top \mathbf{D})^{-1} \mathbf{D}^\top \mathbf{d} \tag{3.21}$$

Now that the initial estimates of all parameters, intrinsic, extrinsic and radial distortion, has been calculated, they will all be refined using a complete maximum likelihood estimation. This can be obtained by minimizing the following function [45]

$$\sum_{i=1}^{n} \sum_{j=1}^{m} \left\| {}^{i}\mathbf{P}_{ij} - {}^{i}\breve{\mathbf{P}}(\mathbf{C}, \mathbf{k}, \mathbf{R}_i, \mathbf{t}_i, {}^{c}\mathbf{P}_j) \right\| \tag{3.22}$$

where n represents the number of images in the calculation, m represents the number of points on the calibration object (corners in the chessboard), $\mathbf{C}$ represents the camera calibration matrix with the intrinsic parameters, $\mathbf{k}$ represents the radial distortion, $\mathbf{R}_i$ represents the rotation matrix of the camera with respect to the calibration object in the i-th image, $\mathbf{t}_i$ represents the translation of the camera in the same image and ${}^{c}\mathbf{P}_j$ represents the j-th 3D point on the calibration object. ${}^{i}\breve{\mathbf{P}}(\mathbf{C}, \mathbf{k}, \mathbf{R}_i, \mathbf{t}_i, {}^{c}\mathbf{P}_j)$ is the projection of the point ${}^{c}\mathbf{P}_j$ in image i. Solving this nonlinear minimization problem can be done by using a nonlinear solver like the Levenberg-Marquardt algorithm.

As a summary of the method, Z. Zhang in [45] presents the following procedure:

1. Print a pattern (chessboard) and attach to a planar surface;

2. Take images of the pattern from different orientations;

3. Detect the feature points in the images (chessboard corners);

4. Estimate the intrinsic and extrinsic parameters by using eq. (3.17) as described above;

5. Estimate the radial distortion parameters by eq. (3.21);

6. Refine the estimated parameters by complete maximum likelihood estimation by minimizing eq. (3.22);

A calibration procedure presented by *OpenCV* is loosely based on the method presented above. It begins by estimating the intrinsic parameters by using eq. (3.17) and the method described above, then it uses the intrinsic parameters to find the pose of the camera (extrinsic parameters) by solving a PnP problem, see section 2.4.1, and not by using the homography as proposed by [45]. The distortion is modeled a bit differently here as well, where the radial distortion is modeled with three parameters (can be extended to 6) and the tangential is not neglected but modeled by two parameters, p_1, p_2. The model of the distortion in eq. (3.4) is in this procedure modeled as:

$$\begin{bmatrix} \delta_u \\ \delta_v \end{bmatrix} = \begin{bmatrix} u(k_1 r^2 + k_2 r^4 + k_3 r^6) \\ v(k_1 r^2 + k_2 r^4 + k_3 r^6) \end{bmatrix} + \begin{bmatrix} 2p_1 uv + p_2(r^2 + 2u^2) \\ p_1(r^2 + 2v^2) + 2p_2 uv \end{bmatrix} \tag{3.23}$$

The parameters of the distortion model are all initialized to 0. This is to avoid the intricate calculations to find initial estimates, because the model is more complex than the model presented in [45], it is very difficult to find good initial estimates.

The last step of the procedure is to refine the parameters. This is done by running the Lavenberg-Marquardt optimization algorithm over the reprojection error in order to minimize it. The reprojection error is found by the following function

$$\sum_{i=1}^{n} \sum_{j=1}^{m} d({}^i\mathbf{P}ij, {}^i\hat{\mathbf{P}}(\mathbf{C}, \mathbf{k}, \mathbf{p}, \mathbf{R}_i, \mathbf{t}_i, {}^c\mathbf{P}_j))^2 \tag{3.24}$$

where d is a function calculating the euclidean distance between two points. ${}^i\hat{\mathbf{P}}(\mathbf{C}, \mathbf{k}, \mathbf{p}, \mathbf{R}_i, \mathbf{t}_i, {}^c\mathbf{P}_j)$ represents the projected image coordinates of the j-th model point ${}^c\mathbf{P}_j$ in the i-th image, and $\mathbf{p}$ are the parameters of the tangential distortion.

By running this kind of a procedure the parameters of the camera model with distortion are estimated. After calibration, the camera model stays the same unless the camera settings are changed, for example by increasing the focal length.

We have now introduced the camera. The perspective camera model has been presented together with a model of the distortion, and calibration procedures for estimating the parameters of the models. We will now use the parameters and model in single view geometry where we present some geometrical connections between a scene and its projection in the image plane.

3.2 Single View Geometry

In order to use the information in the image to perform a 3D reconstruction, there is a need for mathematical tools that can describe the projection of the scene into the image plane. Single view geometry is the common term for these mathematical tools, which are different based on what camera model that is used. In this project the perspective camera model described in section 3.1.2 is used, which assumes a linear relationship between the points in the 3D scene and the 2D image plane.

3.2.1 Image Plane

The image plane is illustrated in the perspective camera model in fig. 3.2. The plane has two different coordinate systems, the image coordinate system with origin in the top left corner and the normalized image coordinates with origin on the principle line only a simple translation along the camera z axis from the camera coordinate system with the origin in the principal point. The coordinate systems and the image plane is illustrated in fig. 3.5. The two coordinate systems have different units, the image coordinate system have pixels as unit while the normalized image coordinate system uses meters as unit.

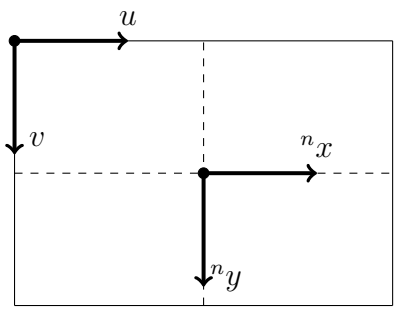

Figure 3.5: The image plane with the image coordinate system and the normalized image coordinate system.

3.2.2 Scene to Image Plane

The camera calibration matrix is used to project a point in the 3D scene into the 2D image plane. This is the inverse problem of projecting a point from the image plane into the scene, which is an important part of this project. However projecting points this way is important when calculating the reprojection error, which is important when calibration and pose estimation is evaluated.

Now the relationship between the scene coordinates and the image coordinates will be described. Assuming that a point in the camera coordinate system, ${}^{c}\mathbf{P} = ({}^{c}x, {}^{c}y, {}^{c}z)$ in the scene is projected into the projective plane by

$$ {}^{p}\mathbf{P} = ({}^{p}x, {}^{p}y, 1) = (\frac{{}^{c}x}{{}^{c}z}, \frac{{}^{c}y}{{}^{c}z}, \frac{{}^{c}z}{{}^{c}z}) = (\frac{{}^{c}x}{{}^{c}z}, \frac{{}^{c}y}{{}^{c}z}, 1) \tag{3.25} $$

In order to describe this point in the normalized image coordinate system triangle similarity is used, which can be rewritten in terms of the projective plane coordi-

nates, see also fig. 3.2 for the model,

$$\frac{{}^{n}x}{f} = \frac{{}^{c}x}{{}^{c}z} \implies {}^{n}x = f\frac{{}^{c}x}{{}^{c}z} = f\,{}^{p}x \tag{3.26}$$

$$\frac{{}^{n}y}{f} = \frac{{}^{c}y}{{}^{c}z} \implies {}^{n}y = f\frac{{}^{c}y}{{}^{c}z} = f\,{}^{p}y \tag{3.27}$$

Getting from normalized image coordinates to image coordinates is just a translation in the plane

$${}^{i}x = f\,{}^{p}x + x_0 \tag{3.28}$$

$${}^{i}y = f\,{}^{p}y + y_0 \tag{3.29}$$

The last step is to change the unit of the coordinates from meters to pixels, which is easily done with the pixel density of the imaging sensor. In horizontal direction the pixel density is given by $\rho_x = \frac{n_x}{s_x}$, and in vertical direction it is $\rho_y = \frac{n_y}{s_y}$. Where n_x, n_y, s_x and s_y are the number of pixels in the sensor and the physical size of the sensor in meters respectively.

$${}^{i}u = \rho_x f\,{}^{p}x + \rho_x x_0 = f_x\,{}^{p}x + c_u \tag{3.30}$$

$${}^{i}v = \rho_y f\,{}^{p}y + \rho_y y_0 = f_y\,{}^{p}y + c_v \tag{3.31}$$

The previous two equations can be written in matrix form as

$$\begin{bmatrix} {}^{i}u \\ {}^{i}v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_u \\ 0 & f_y & c_v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{p}x \\ {}^{p}y \\ 1 \end{bmatrix} = \mathbf{C}\,{}^{p}\mathbf{P} \tag{3.32}$$

Where the matrix $\mathbf{C}$ is called the camera calibration matrix. Formally this matrix is written as

$$\mathbf{C} = \begin{bmatrix} f_x & s & c_u \\ 0 & f_y & c_v \\ 0 & 0 & 1 \end{bmatrix} \tag{3.33}$$

where s is a skew parameter if the camera sensor is not rectangular, but this is usually not the case so it is set to 0. The camera calibration matrix contain the information from the calibration, and consists of the intrinsic parameters of the camera.

If we in addition relate the camera coordinate system to the world coordinate system we get the linear relationship between the scene and image plane. Assuming a point P described in the world frame: ${}^{w}P = ({}^{w}X, {}^{w}Y, {}^{w}Z)$, which can be augmented to include 1 as the last element: ${}^{w}\tilde{P} = ({}^{w}X, {}^{w}Y, {}^{w}Z, 1)$. Transforming this point to the camera coordinate system can be done by:

$${}^{c}\tilde{\mathbf{P}} = \begin{bmatrix} {}^{c}X \\ {}^{c}Y \\ {}^{c}Z \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} {}^{w}X \\ {}^{w}Y \\ {}^{w}Z \\ 1 \end{bmatrix} = [R\ t]\,{}^{w}\tilde{\mathbf{P}} \tag{3.34}$$

In this transformation, $[R\ t]$ are the extrinsic parameters of the camera in the world frame, describing the rotation and translation of the camera in the world coordinate system. By adding the results of eq. (3.32) into the results of eq. (3.34) we find the linear relationship between a point in the 3D scene and the projected point in the 2D image plane:

$$s\,{}^{i}\tilde{\mathbf{P}} = s\begin{bmatrix} {}^{i}u \\ {}^{i}v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & s & c_u \\ 0 & f_y & c_v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} {}^{w}X \\ {}^{w}Y \\ {}^{w}Z \\ 1 \end{bmatrix} = \mathbf{C}[R\ t]\,{}^{w}\tilde{\mathbf{P}} \tag{3.35}$$

where the constant s is a scale factor. Equation (3.35) is the mathematical interpretation of the perspective camera model described in section 3.1.2.

3.2.3 Image Plane to Scene

No that the forward projection from scene to image plane is found, it is easy to think that the inverse operation is the inverse of the calibration matrix. This is however not actually the case because this will only give the transformation from the pixel coordinates to the projective plane. In the scene this represent a camera ray, and in order to place the point one of the three scene coordinates has to be known.

In this project the goal is to estimate the pose of the camera compared to the quay. In order to achieve this pose estimation algorithms need some point correspondences between the image plane and the scene. This is called model based pose estimation, where the model in this case will be of the markers placed around the quay as discussed in section 2.3.2. The pose estimation is done by moving the pose of the camera so that the projection of the markers in the image plane is equal to the location of the detected markers in the image with a minimized geometrical error. The problem of estimating the pose of a camera based on n known points is called the "PnP problem", and as it is shown in the next chapter there are many different algorithms to solve it.

3.3 Pose Estimation

The pose of an object can be described by six parameters; three translations and three rotations, also called the extrinsic parameters of the camera. These six parameters of the camera can be estimated if the intrinsics and distortion parameters are known. In other words the results from the camera calibration in section 3.1.4 are used to calculate the extrinsic parameters of the camera by a spacial resection from known points in the scene [11]. Solving the PnP problem is one way of estimating the pose, and in the next section we will present different PnP solvers.

3.3.1 PnP

The PnP problem is explained in section 2.4.1, and we will in this section present some of the mathematical theory behind a couple of algorithms that solve the

problem. The solvers are all implemented in the *OpenCV* library, can are easily used by calling a "solvePnP" function with image points and scene points as input. An extra possibility presented by the OpenCV developers is to use the presented algorithms with a RANSAC scheme, where the points used in the algorithm is based on selection using the RANSAC algorithm. Such a solver is more robust to outliers than the original solver, but can be more inaccurate if the points are well defined. The RANSAC solver is most used when the points are influenced by a lot of disturbances. For more information on the implementation, the reader is advised to review the documentation [34].

A lot of the mathematical background on the PnP solvers is quite complex, and the methods that are used for solving the problems are outside the scope of this report. However all papers are cited in the proper sections, so for a more in-depth look at the algorithms the reader can find the information following the citations.

Iterative Levenberg-Marquardt (LM) Optimization

This method is based on both Gradient Descent (GD) and Gauss-Newton (GN) optimization. The algorithm combines the advantages of GD and GN by choosing the iterative step (popularly called the "LM step") to be a linear combination of the two methods. The algorithm is presented by M. Transtrum, B. Machta and J. Sethna in [38], and it iteratively solves the problem by minimizing the reprojection error. A purely analytical explanation of the algorithm is that the LM step is dominated by the GD until it finds a "canyon" in the function, and then the step will be dominated by GN until the function converges. The LM step is called adaptive because it changes based on how the function looks around the current iteration.

CASSC

The CASSC (Complete Analytical Solution with the assistance of Solution Classification) algorithm is an algorithm for the minimal PnP problem, P3P. In section 2.4.1 we presented how 3 points are the minimum number of points that give a finite number of possible poses, and this is why the P3P problem is called the minimal PnP problem. CASSC is a complete and robust algorithm solving the P3P problem, and was presented by X.-S. Gao, X.-R. Hou, J. Tang and H.-F. Cheng in [12]. The algorithm finds all the possible solutions to the P3P problem, and classifies them as possible or impossible results.

The paper presents two approaches; geometrical and algebraic. The geometric approach can give some criteria for the number of solutions which will be simpler than the algebraic approach, but it has to find the solutions using an algebraic approach in any case. The algebraic approach finds the first solution to the problem by Wu-Ritt's zero decomposition algorithm, which gives a triangular decomposition of the equation sets of the P3P problem. The decomposition gives the first analytical solution to the equation set. After this a complete solution classification to the P3P equation system is found from the decomposition. This classification means that there are found explicit criteria for the given problem to have one, two, three or four solutions. The CASSC algorithm combines the analytical solutions with the

criteria, which can be used to find the complete and robust numerical solutions to the P3P problem.

The mathematics supporting this algorithm is quite extensive. Especially finding the classification is very complex, and the reader is advised to read the paper, [12], for a complete description of the algorithm.

OpenCV has implemented this algorithm as a PnP solver. This implementation does however require four points, even though the solver is for P3P problems. The fourth point in this problem is used to narrow down the result to a single pose estimate after running the proposed CASSC algorithm.

AP3P

This is a method based on the paper by T. Ke and S. Roumeliotis [18]. They do not actually provide a name for their algorithm, so we will reference it by AP3P or (Algebraic P3P). The algorithm is a more efficient solver than the previously proposed CASSC algorithm, in fact they claimed that it only used 40% of the time of the current state of the art methods. The authors also claimed that the AP3P algorithm was both more robust and accurate than other methods.

This method starts by finding the attitude (orientation) of the camera, and then finds the translation (this is opposite to other methods like the CASSC presented above). The method first directly determines the camera's attitude by eliminating it's position and features' distances, resulting in a system of trigonometric equations. By employing an algebraic approach to the system it yields the rotation matrix of the camera, and then it's position.

This method is also implemented as a function on the "solvePnP" function in *OpenCV*. The implementation require four points in both image coordinates and 3D scene coordinates, the reason for requiring four points is the same as in the CASSC solver: to find the correct pose if multiple are found.

DLS

This is a Direct Least-Squares method for PnP based on the paper [15]. It is a solver that works for $n \geq 3$ points, based on solving a nonlinear least-squares cost function, eq. (3) in [15]

$$\{\alpha_i^*, {}^S_G\mathbf{C}, {}^S\mathbf{p}_G^*\} = argmin \sum_{i=1}^{n} \left\| \mathbf{z}_i - {}^S\bar{\mathbf{r}}_i \right\|^2 = \sum_{i=1}^{n} \left\| \mathbf{z}_i - \frac{1}{\alpha_i}({}^S_G\mathbf{C}\,{}^G\mathbf{r}_i + {}^S\mathbf{p}_G) \right\|^2 \tag{3.36}$$

$$\text{subject to } {}^S_G\mathbf{C}^\top {}^S_G\mathbf{C} = \mathbf{I}_3, \qquad det({}^S_G\mathbf{C}) = 1 \tag{3.37}$$

$$\alpha_i = \left\| {}^S_G\mathbf{C}\,{}^G\mathbf{r}_i + {}^S\mathbf{p}_G \right\| \tag{3.38}$$

where the cost function is the sum of the squared measurement errors. Finding all solutions of this problem is challenging because the cost function is nonlinear in the unknown quantities. The approach that is used in [15] is to relax the optimization problem described above, and manipulate the measurement equations in order to reduce the number of unknowns. The details of this procedure is described in sections 3.3-3.5 in [15].

The algorithm directly finds all possible poses based on any number of points, $n \geq 3$. This makes the algorithm very flexible, and the scalability is very good. The poses are calculated using an analytical approach, and it does not need any initialization.

EPnP

The EPnP algorithm is a fast and non-iterative solver based on the paper [20] for $n \geq 4$. This method expresses the n points as a weighted sum of four virtual control points

$$\mathbf{p}_i^w = \sum_{j=1}^{4} \alpha_{ij} \mathbf{c}_j^w, \quad \sum_{j=1}^{4} \alpha_{ij} = 1 \tag{3.39}$$

where α_{ij} are homogeneous barycentric coordinates, $\mathbf{p}$ are the 3D world coordinates for the points n and $\mathbf{c}$ are the virtual control points. The algorithm then solves for the coordinates of the control points in the camera referential instead. If there is a higher accuracy requirement the result can be optimized by running Gauss-Newton with the previous result as initialization.

UPnP

This method is inspired by the EPnP method, and is also non-iterative. It is based on the paper [28], and in addition to estimate the pose of the camera it also handles the focal length as an unknown. This means that the algorithm will work even on uncalibrated cameras provided that the location of the principal point in the image plane is known, and that the skew parameter is zero.

Chapter 4

Hardware and Software Implementation

In order to work with new methods and algorithms, they have to be tested and verified. In this chapter we present the implementations that was used in the project, as well as the hardware and software.

4.1 Hardware

This section introduces the reader to the hardware that was used to test the methods. The only hardware components that were used is a camera, a computer and markers.

4.1.1 Computer

The main computer used in the project was an Apple Mac-book Pro. It has a 2GHz Intel Core i7 processor and 8 GB RAM.

4.1.2 Camera

In order to test methods and implementations a single camera is used. The camera is a Canon EOS 7D digital single-lens reflex (DSLR) with a Canon EF 17-40mm lens attached. The camera has an 18 mega pixel APS-C CMOS image sensor with a size of 22.3x14.9mm. In order to keep the parameters of the camera model constant, the lens is kept at 17mm in all images both in the calibration and the pose estimation.

4.1.3 Markers

QR codes was used as markers in this project, see section 2.3.2. The QR codes contained simple text displaying information on what marker it was, and was printed with 81mm and 161mm sides. In all tests the QR codes was taped to a straight object like a wall or a table. All four corners were used as feature points in the algorithm.

4.2 Software

The software used in the project is mainly focused around the *OpenCV* library for python. The *OpenCV* library is simple to use and install in addition to containing a lot of built-in functions for computer vision tasks. We used python version 3.5.3 with version 3.4.0 of *OpenCV*.

4.2.1 Calibration

The calibration algorithm is based on the examples and implementation shown on the *OpenCV* documentation page [24]. As the documentation page suggests, a 9×6 chessboard was printed and used for calibration of the camera. There was taken 22 pictures of the chessboard from different angles, but only 20 was used for the calibration as two of the images was out of focus.

The implementation of the calibration can be found in section A.1. The calibration is done in the "main()"-function on line 91 which run the entire calibration procedure. The function first make an instance of the "cameraCalibration"-class and run the "calibrate"-function on the instance. This function iteratively finds the chessboard in all the calibration images, saves them as image points and associates them with object points. Then it runs the built-in function "calibrateCamera" which returns a translation vector containing the translation of the camera in all calibration images, a rotation vector containing the attitude of the camera compared to the chessboard in all images, the distortion coefficients for the distortion model introduced in section 3.1.3, the camera calibration matrix introduced in section 3.2.2 and a boolean that informs whether the calibration was successful. This information is then saved as parameters in the instance that was made in the main function. The main function continues, if the calibration is a success, by saving the calibration matrix and distortion coefficients in their own files so that they can easily be loaded and used by other functions. The last parts of the main function on line 98 and 99 calculate the reprojection error and illustrates the information in a figure. If the images with a high reprojection error are removed from the calibration, a recalibration is necessary so that the parameters are updated.

4.2.2 QR Detection

The QR detection implemented in this system is from the *ZBar* library [44]. The implementation itself is in the function "detect_QR_codes(self)" on line 186 in section A.2. The library has a lot of good functions for detection and reading of both bar codes and QR codes, which makes the implementation very simple. It consists of a "scanner" object that can run over the image, and return a container with all the codes detected within it. Each element in the container has the contents of the code and the position of all corners of the code making it easy to use all four corners as points for the pose estimation.

4.2.3 Pose Estimation

The pose estimation implementation is written to test different methods for PnP solving. The accuracy of the methods will be tested both visually and mathematically by measuring the reprojection error. The implementation can be found in section A.2, and the results are shown in section 5.2.

The pose estimation is implemented using a class for each image, and a subclass for each PnP solving methods. The PnP solvers presented in section 3.3.1 are already implemented in the OpenCV library, and are easily called using the function "solvePnP()" as it is done in line 173 in the code in section A.2. The output of the function is a return boolean indicating whether the estimation war a success, a rotation vector indicating the attitude of the camera and lastly a translation vector indicating the position of the camera in the scene coordinate system. The reprojection error is calculated by reprojecting the known scene points using the estimated pose of the camera, and then evaluating the projected points with the detected points in the image.

There are two methods that use only four points, and those are the two P3P solvers. In the testing were there are more than one marker, which gives more than four points, the implementation uses the two first points and the two last. This means that in the testing where we used two markers, two points from the first marker and two points from the last marker are chosen as reference points in the pose estimation.

4.2.4 Runtime Pose Estimation

The pose estimation runtime is tested by timing the pose estimators. A timing test is implemented in the function "pose_estimations_timing_test()" in line 237 in the code in section A.2. The PnP methods are each ran 100 times on the same points, and the runtime results are then shown in the same plot by the function "plot_timing_test_results()".

Chapter 5

Testing and Results

The methods and algorithms were tested on the components introduced in chapter 4 together with the implementations in software.

5.1 Camera Calibration

5.1.1 Camera Calibration Matrix and Distortion Coefficients

One of the results of the camera calibration is the calibration matrix described in section 3.1.4.

$$\mathbf{C} = \begin{bmatrix} 4058.49591 & 0 & 2575.64541 \\ 0 & 4057.71596 & 1722.26599 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

$$= \begin{bmatrix} 0.01745 & 0 & 0.01108 \\ 0 & 0.01749 & 0.00743 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.2}$$

The first matrix has pixels as unit, and the second matrix has meters as unit.

The distortion coefficients are the second result of the calibration error. These correspond to the radial and the tangential distortions presented in section 3.1.3.

$$d = [k_1, k_2, p_1, p_2, k_3] \tag{5.3}$$

$$= [-0.11582583, 0.13720204, 0.00113294, 0.00100731, -0.09993004] \tag{5.4}$$

5.1.2 Reprojection Error

The reprojection error measures how good a calibration is. Figure 5.1 show the result of the reprojection error calculation on the different images that were used for the calibration of the camera. To improve the calibration all images with a reprojection error above the mean error (above the black line) indicated with red bars in the chart was removed and the calibration was done again. The procedure reduced the mean reprojection error by almost 0.01.

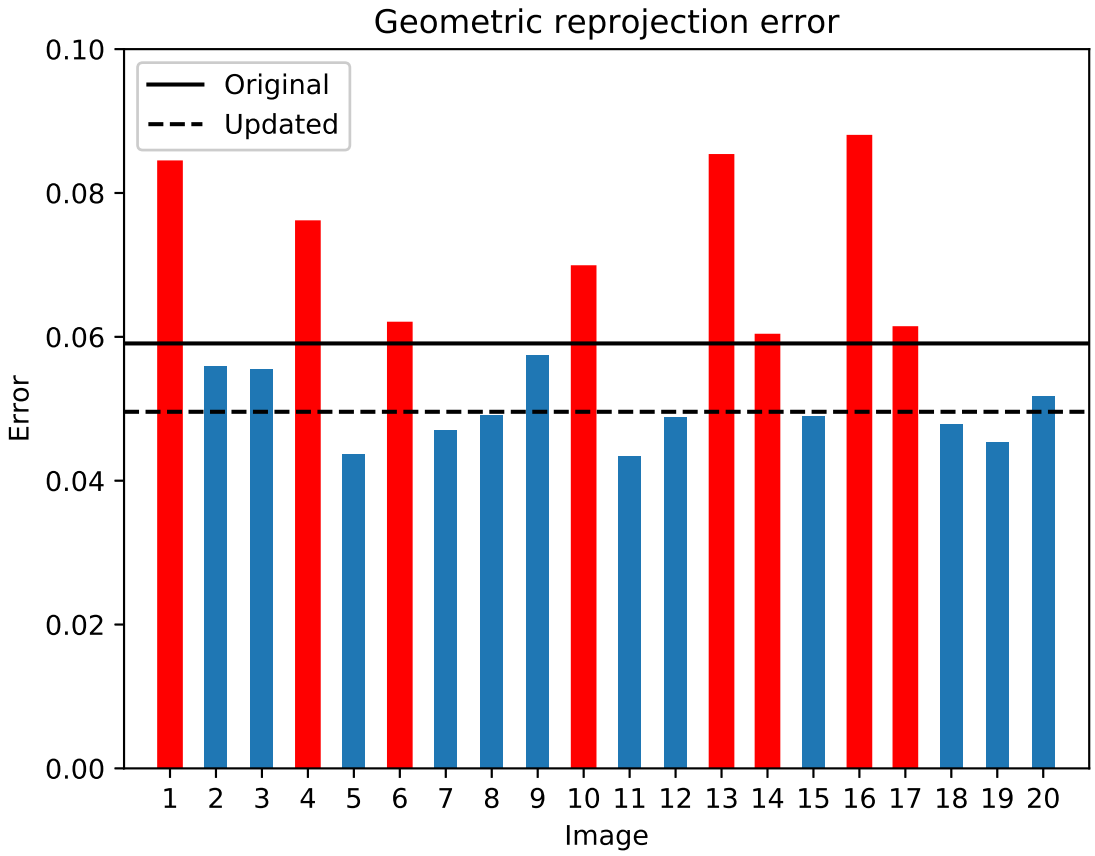

Figure 5.1: The reprojection error in the different images of the calibration.

5.2 Pose Estimation

Pose estimation was thoroughly tested using the implementation presented in chapter 4. The code for all testing presented in this section can be found in section A.2, and the images used are found in section A.3. The images are divided into four sets, where the marker combination is the same. The recorded results are the reprojection error of all methods on all images, the runtime of the estimation methods, the QR detection time and a illustration of the estimated and true poses.

In all image sets, the scene coordinate system is based on the floor. The y-axis is up along the wall, the x-axis is along the corner between wall and floor, and the z-axis goes along the floor perpendicular from the wall. The illustration of the estimated and true poses of the camera are shown for all the images, where the z-axis is along the perspective line, the y-axis points down from the camera and the x-axis points out to the right of the camera.

5.2.1 Image set number 1

In the first image set, there were five images (1.JPG-5.JPG). As it is presented in table 5.1, there was a single marker (with 81mm sides) in all images in this set. The QR code detector was not able to detect the marker in image 3. and 5, which are

the two images that are the furthest away from the marker itself. Hence there are no more results from these images below. Figures 5.2 to 5.7 show the results from the tests of images 1.JPG to 5.JPG. Finally figs. 5.8 and 5.9 show the combined reprojection errors and the runtime of the QR code detection respectively.

Image Name:	#Markers	#Detected Markers:	#1 Marker Size:	Camera Position:
1.JPG	1	1	0.081m	(0,0.85,1)
2.JPG	1	1	0.081m	(0,0.85,2)
3.JPG	1	0	0.081m	(0,0.85,3)
4.JPG	1	1	0.081m	(-1.5,0.85,2.5)
5.JPG	1	0	0.081m	(-1.5,0.85,4)

Table 5.1: Information on the first image set.

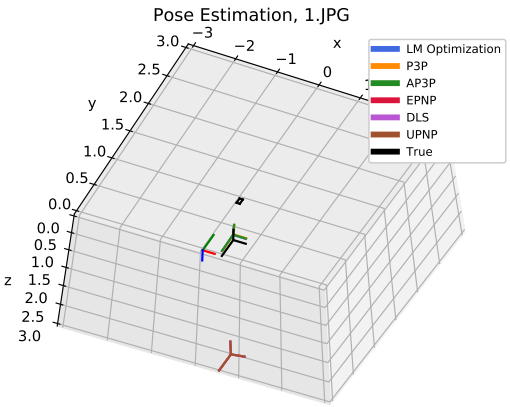

Figure 5.2: The estimated poses and true pose of image 1

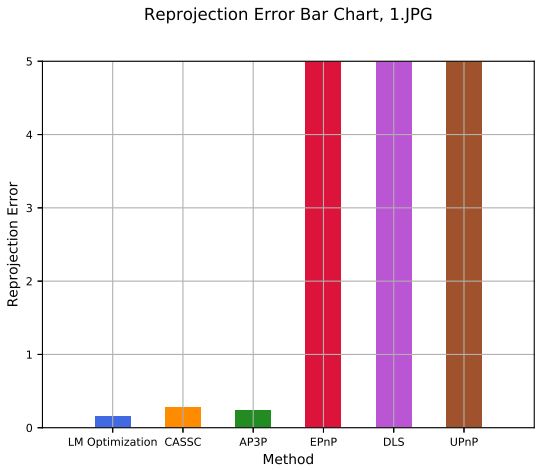

Figure 5.3: Reprojection error of pose estimation in image 1

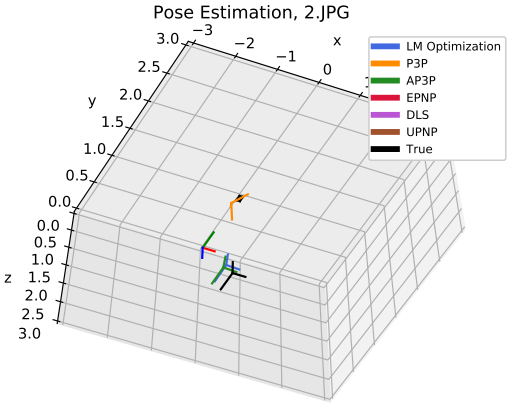

Figure 5.4: The estimated poses and true pose of image 2

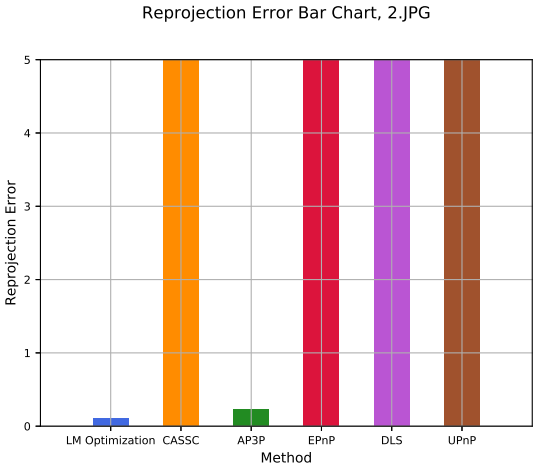

Figure 5.5: Reprojection error of pose estimation in image 2

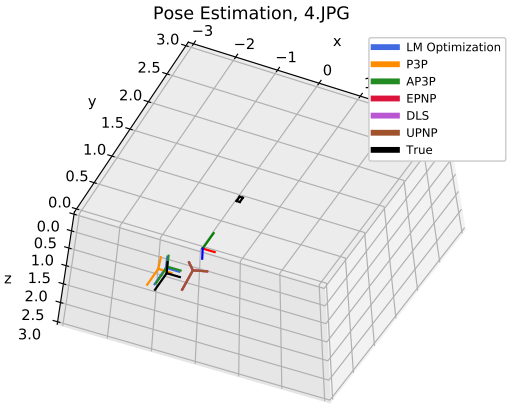

Figure 5.6: The estimated poses and true pose of image 4

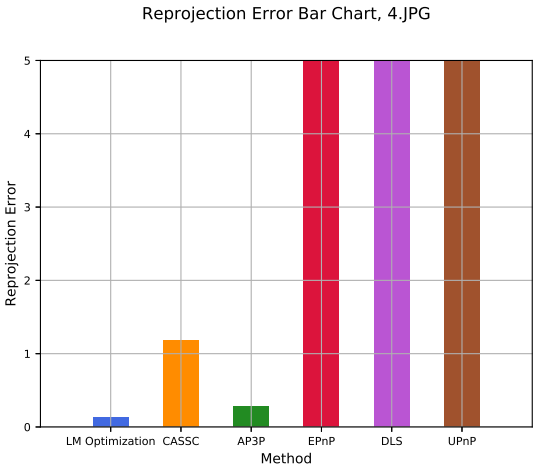

Figure 5.7: Reprojection error of pose estimation in image 4

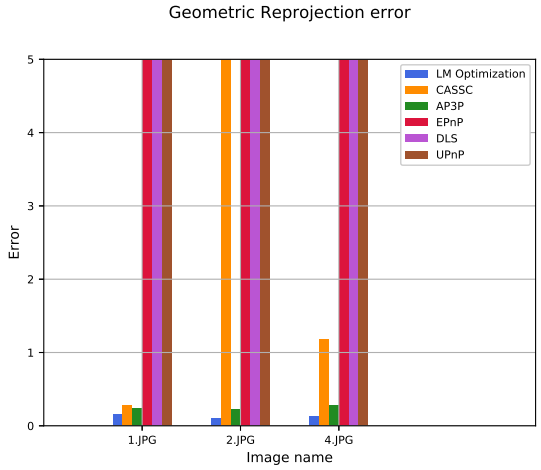

Figure 5.8: Comparing the reprojection error of all images and methods in image set 1.

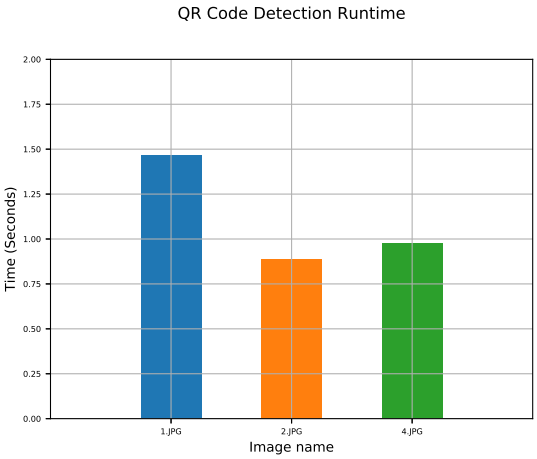

Figure 5.9: Runtime for the QR detection in all images in image set 1.

5.2.2 Image set number 2

The second image set contained images 6.JPG-9.JPG. In this set there was two markers in the scene, both of them had sides measuring 81mm, see table 5.2. Figures 5.10 to 5.17 show the test results of images 6.JPG-9.JPG, and figs. 5.18 and 5.19 show all projection errors compared and the QR code detection respectively.

Image name: Name:	#Markers	#Detected Markers:	#1 Marker Size:	#2 Marker Size:	Camera Position:
6.JPG	2	2	0.081m	0.081m	(0,0.85,2)
7.JPG	2	2	0.081m	0.081m	(0,0.85,3)
8.JPG	2	1	0.081m	0.081m	(-1.5,0.85,2.5)
9.JPG	2	1	0.081m	0.081m	(-1.5,0.85,4)

Table 5.2: Information on the second image set.

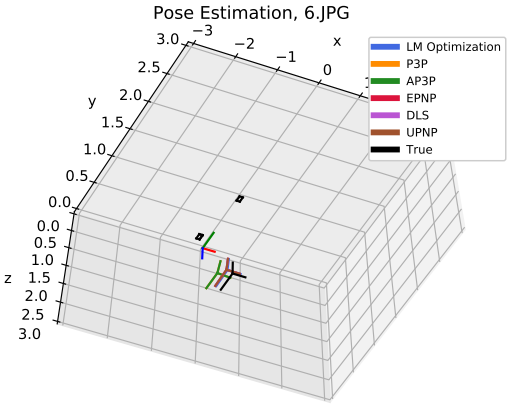

Figure 5.10: The estimated poses and true pose of image 6

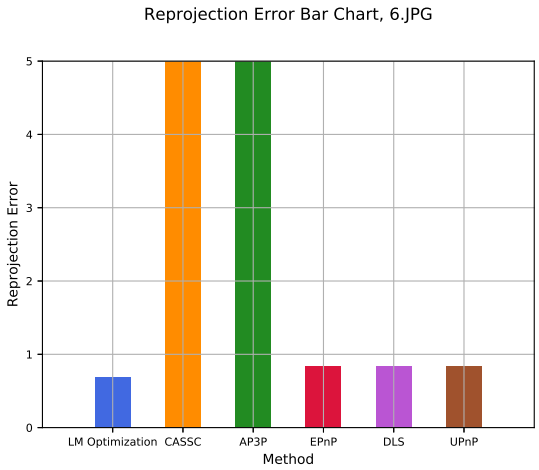

Figure 5.11: Reprojection error of pose estimation in image 6

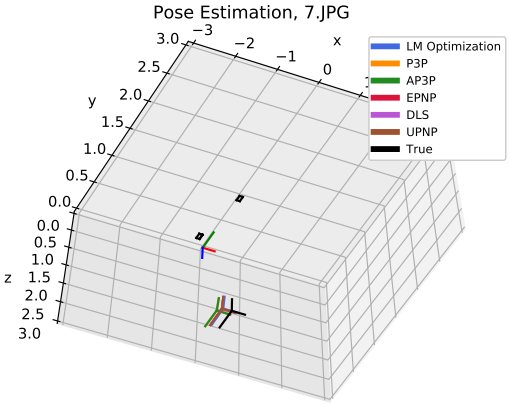

Figure 5.12: The estimated poses and true pose of image 7

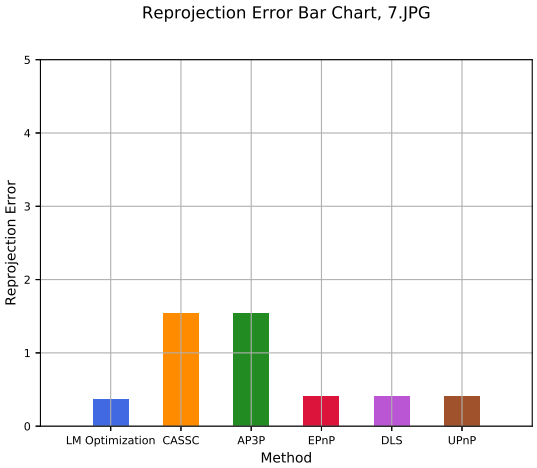

Figure 5.13: Reprojection error of pose estimation in image 7

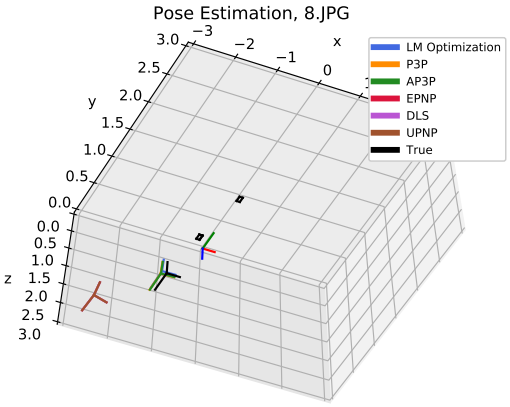

Figure 5.14: The estimated poses and true pose of image 8

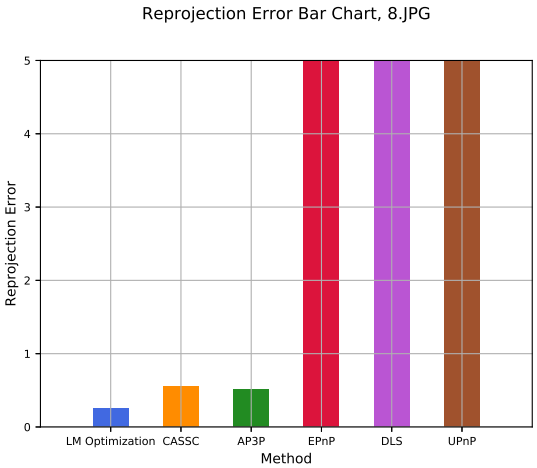

Figure 5.15: Reprojection error of pose estimation in image 8

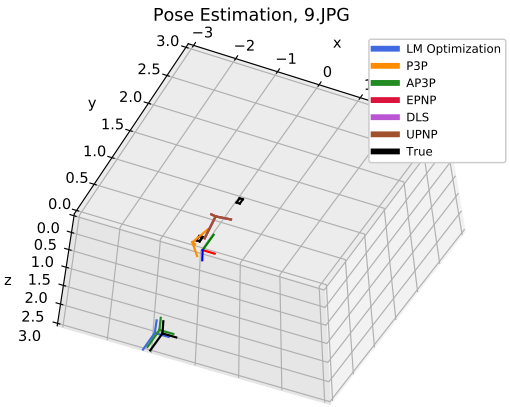

Figure 5.16: The estimated poses and true pose of image 9

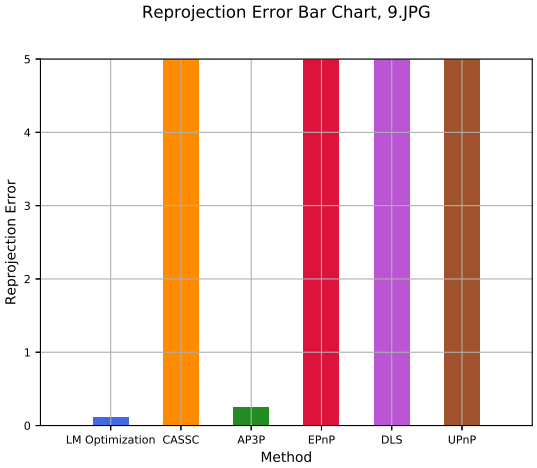

Figure 5.17: Reprojection error of pose estimation in image 9

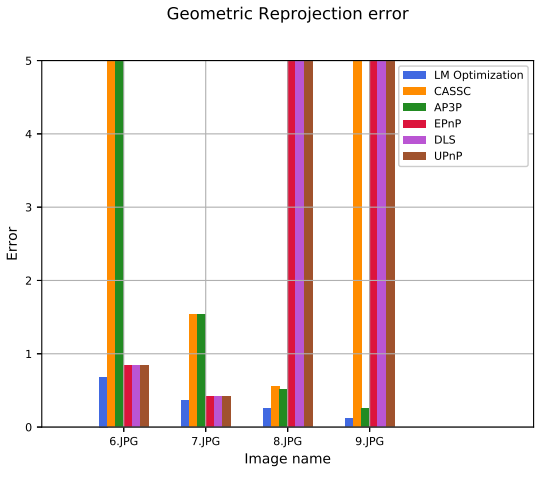

Figure 5.18: Comparing the reprojection error of all images and methods in image set 2.

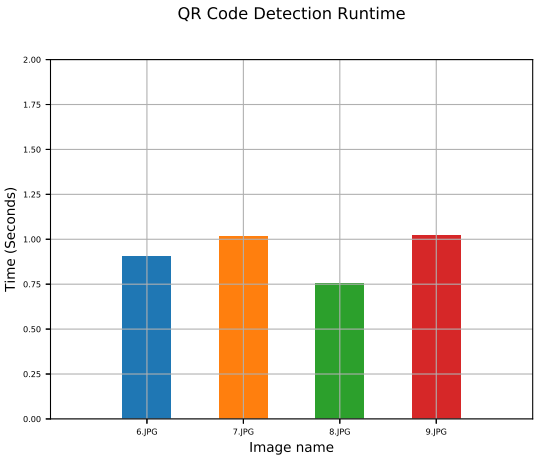

Figure 5.19: Runtime for the QR detection in all images in image set 2.

5.2.3 Image set number 3

Just like the first two image sets the information about the images can be found in table 5.3. The scene contained two markers, one with edge sizes 161mm and one with sizes 81mm, both of them can be seen in correct scale in fig. 5.26. Figures 5.20 to 5.29 show the image test results, and figs. 5.30 and 5.31 show the image reprojection errors compared and QR code detection runtime.

Image Name:	#Markers	#Detected Markers:	#1 Marker Size:	#2 Marker Size:	Camera Position:
10.JPG	2	1	0.161m	0.081m	(0,0.85,1)
11.JPG	2	2	0.161m	0.081m	(0,0.85,2)
12.JPG	2	2	0.161m	0.081m	(0,0.85,3)
13.JPG	2	2	0.161m	0.081m	(-1.5,0.85,2.5)
14.JPG	2	2	0.161m	0.081m	(-1.5,0.85,4)

Table 5.3: Information on the third image set.

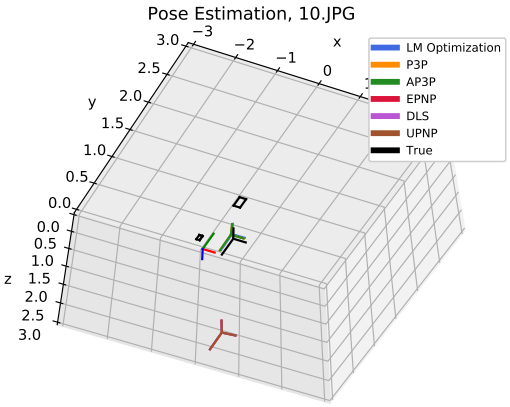

Figure 5.20: The estimated poses and true pose of image 10

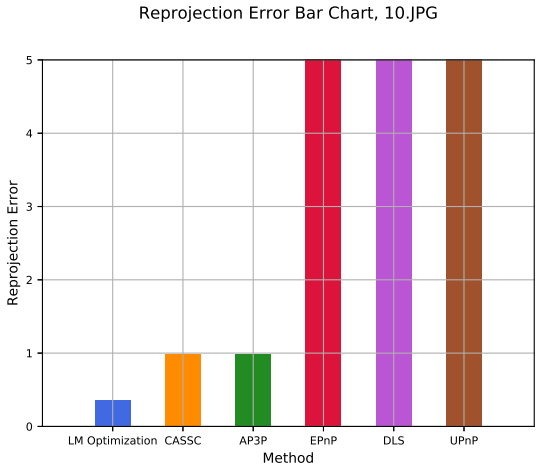

Figure 5.21: Reprojection error of pose estimation in image 10

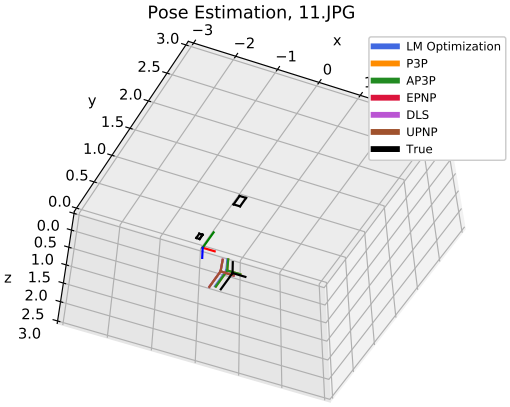

Figure 5.22: The estimated poses and true pose of image 11

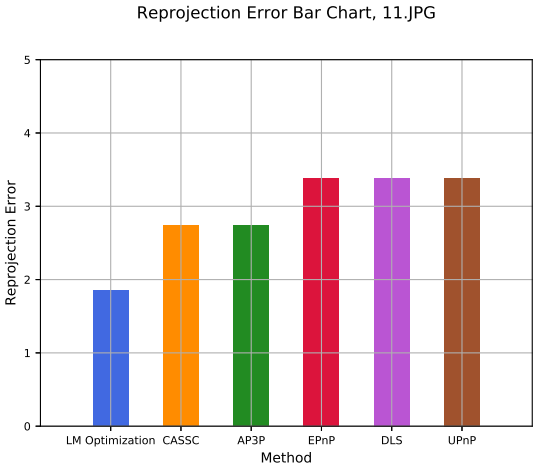

Figure 5.23: Reprojection error of pose estimation in image 11

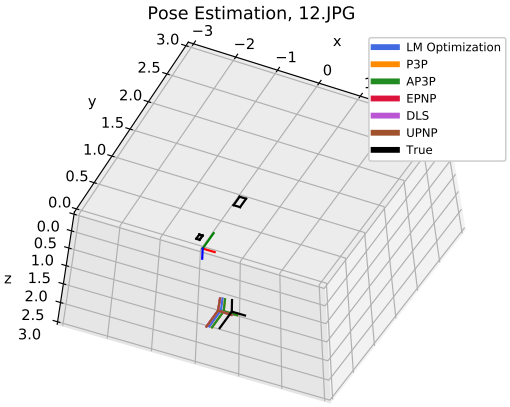

Figure 5.24: The estimated poses and true pose of image 12

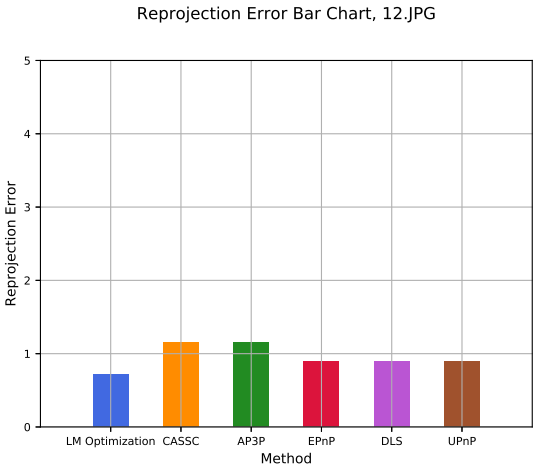

Figure 5.25: Reprojection error of pose estimation in image 12

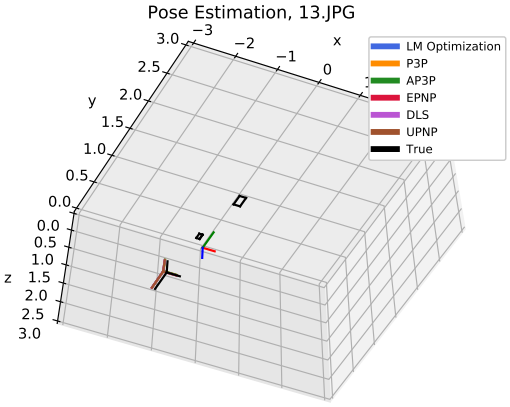

Figure 5.26: The estimated poses and true pose of image 13

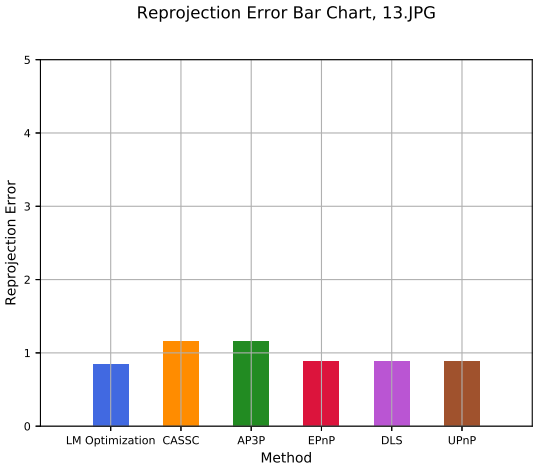

Figure 5.27: Reprojection error of pose estimation in image 13

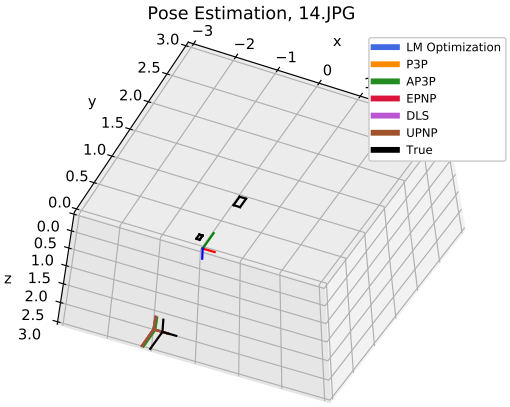

Figure 5.28: The estimated poses and true pose of image 14

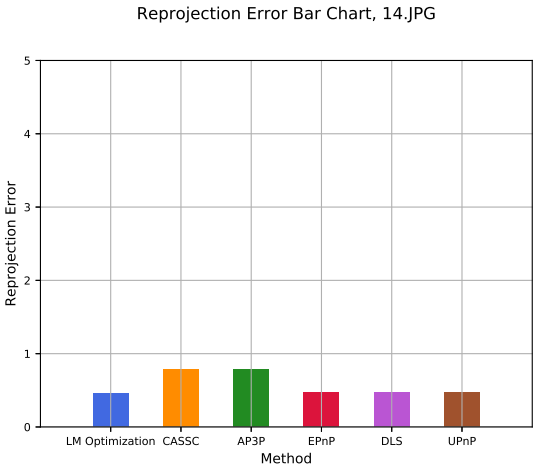

Figure 5.29: Reprojection error of pose estimation in image 14

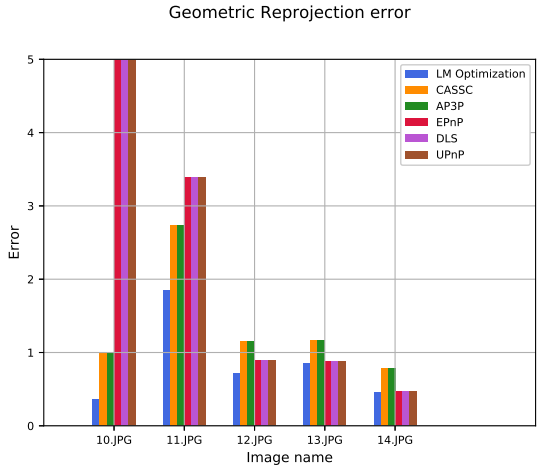

Figure 5.30: Comparing the reprojection error of all images and methods in image set 3.

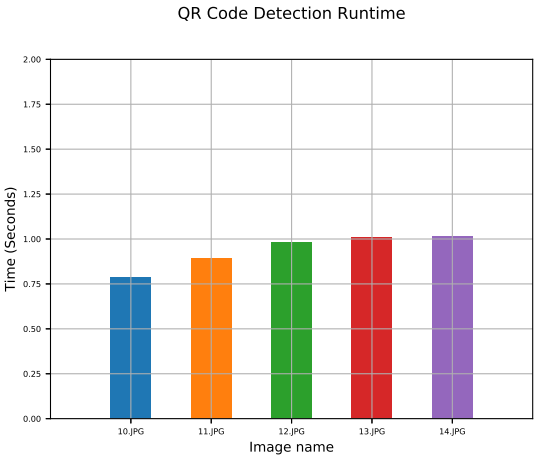

Figure 5.31: Runtime for the QR detection in all images in image set 3.

5.2.4 Image set number 4

The last image set is images 15.JPG to 18.JPG. The information in table 5.4 show the setup of the images, and the scene contained two markers with edge sizes 161mm. Figures 5.32 to 5.39 show the results of pose estimation, and the reprojection error in all images and QR detection runtime is shown in figs. 5.40 and 5.41.

Image name:	#Markers	#Detected Markers:	#1 Marker Size:	#2 Marker Size:	Camera Position:
15.JPG	2	2	0.161m	0.161m	(0,0.85,2)
16.JPG	2	2	0.161m	0.161m	(0,0.85,3)
17.JPG	2	2	0.161m	0.161m	(-1.5,0.85,2.5)
18.JPG	2	2	0.161m	0.161m	(-1.5,0.85,4)

Table 5.4: Information on the fourth image set.

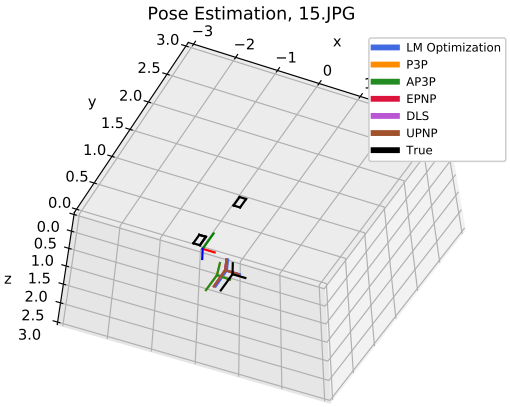

Figure 5.32: The estimated poses and true pose of image 15

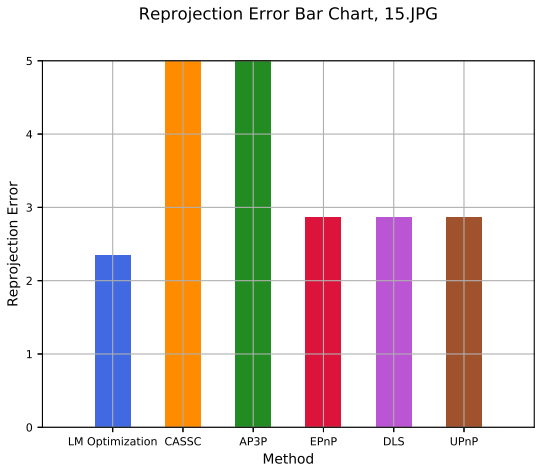

Figure 5.33: Reprojection error of pose estimation in image 15

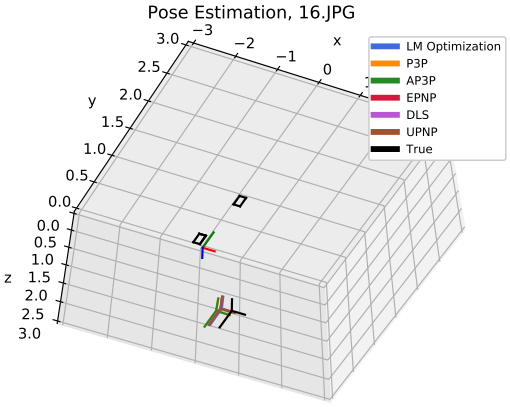

Figure 5.34: The estimated poses and true pose of image 16

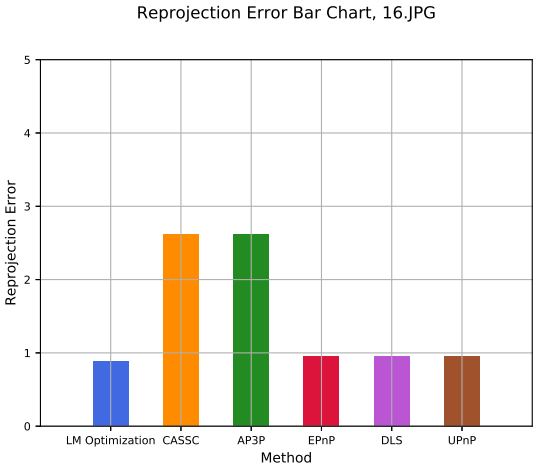

Figure 5.35: Reprojection error of pose estimation in image 16

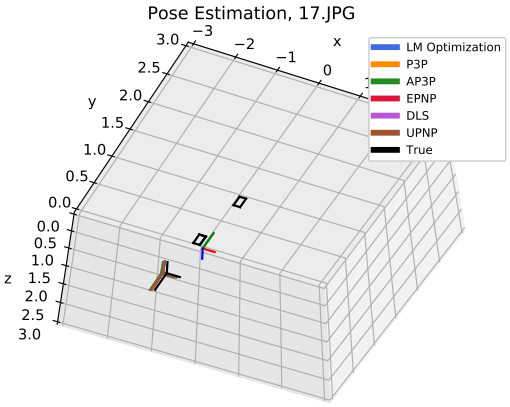

Figure 5.36: The estimated poses and true pose of image 17

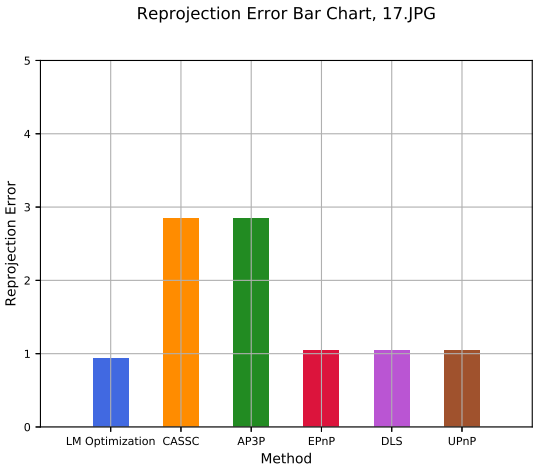

Figure 5.37: Reprojection error of pose estimation in image 17

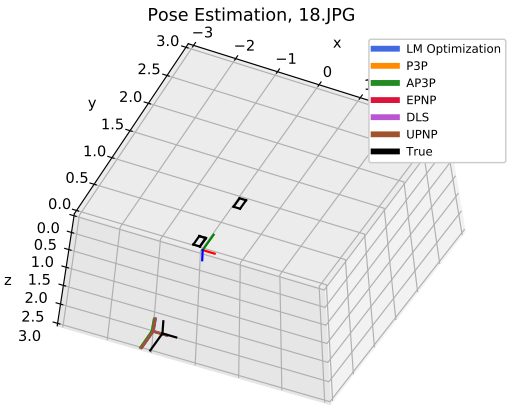

Figure 5.38: The estimated poses and true pose of image 18

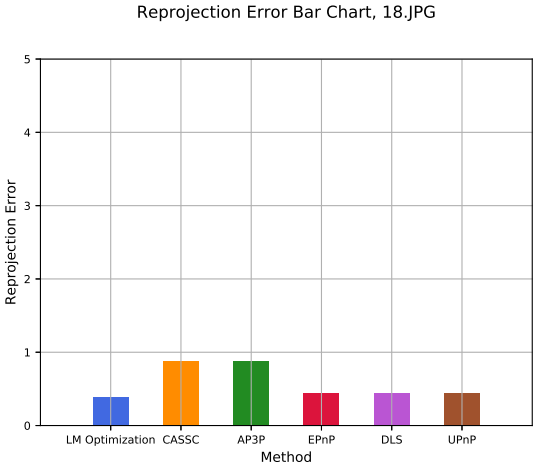

Figure 5.39: Reprojection error of pose estimation in image 18

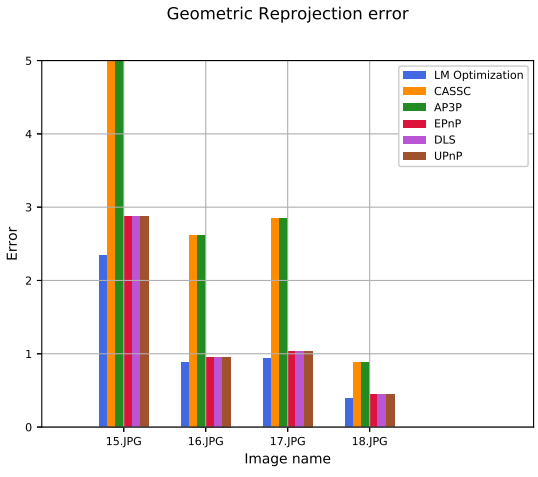

Figure 5.40: Comparing the reprojection error of all images and methods in image set 4.

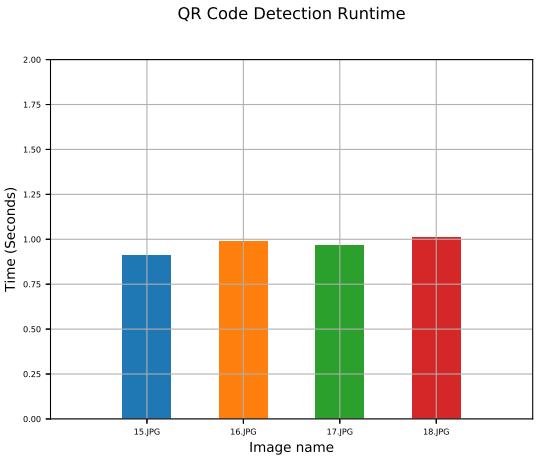

Figure 5.41: Runtime for the QR detection in all images in image set 4.

Runtime

The runtime of the PnP solvers was found from running all solvers 100 times. Each iteration was done on the same data, and the resulting chart can be seen in fig. 5.42. The runtime is found using the implementation described in section 4.2.4, and the code is found on line 237 in section A.2.

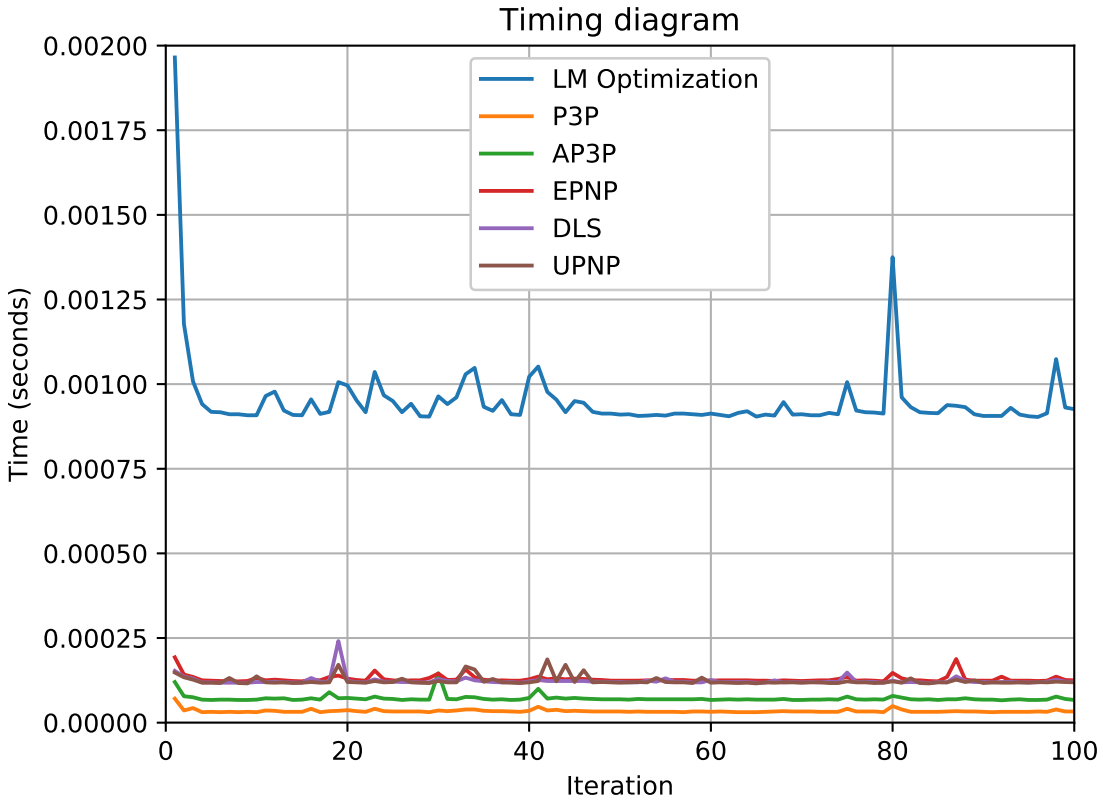

Figure 5.42: Timing of different PnP solvers during 100 iterations.

Chapter 6

Discussion

The contents of this last chapter is a discussion of the results in the project. In the last part of the chapter there are some suggestions for future work, that can be used when developing an actual positioning system for a vessel.

6.1 Hardware

In this section we will discuss the hardware used in the project.

6.1.1 Camera

The camera used in the tests in this project is too good for representing the cameras on board a vessel. The intention of using such a good camera was to get good results for testing the methods for pose estimation. The results from the testing show that the QR markers in the images can easily be detected, the only times where it was a problem was when the QR codes was too small and too far away from the camera. Making the QR codes big can solve problems of detecting them in cameras with low resolution. Images with lower resolution can also make the QR detection time in the images faster.

6.1.2 Computer

The computer is a standard UNIX-based computer. The processing power is average, installing a similar processing power on a ferry will be a reasonable investment.

6.1.3 Marker

As presented in section 4.1.3 we used QR codes as markers for the pose estimation. The codes are easy to detect as long as light conditions are good, and there are nothing blocking the view. For the pose estimation testing however, the QR codes works very well as they have four corners that can easily be used as four separate reference points.

One of the biggest disadvantages is resolution. In order to be detectable by the camera, it has to have a resolution so that the camera sensor can read it from an appropriate distance. This can be observed from the results in section 5.2 where

the bigger QR code is detected in all five images in image set 3, but only in the closest three images in image set 1.

Runtime is another issue. As it clearly comes from figs. 5.9, 5.19, 5.31 and 5.41, it takes a long time to detect a QR code (around 1 second on average). This is not acceptable even though the ferry is not supposed to move very quickly in the docking. This might be solved by changing language from python to C++, but this has yet to be verified. Another possible solution is lower resolution images, so there are less pixels to scan over for the QR detector.

6.2 Software

Now we will discuss the software used and tested in the project.

6.2.1 Python

The software in this project is written in python. It is a language that is easy to use and simple to understand, and it is therefore well suited for testing and developing new methods and algorithms. However the language is not very efficient, and the runtime is therefore quite high which is reflected especially in the QR detection section 4.2.2. This makes it not well suited for real-time applications such as in the autonomous docking operation.

It is suggested that further development of such a system is done in a more efficient language. A good alternative to python with the same advantages of the *OpenCV* package is C++, which will result in much lower runtime. Another advantage of C++ is that it is a language that is used a lot in the maritime industry, and it is also the language used by RRM on all of their systems. This makes implementation on board vessels easier, and the system can interact easy with the other systems on board.

6.2.2 Calibration

The implementation of the camera calibration is pretty simple. The code is found in section A.1, and the implementation is inspired by the examples in [24]. The "calibrateCamera()" on line 51 in the implementation is based on the theory in section 3.1.4, and can be called in the same way in both Python and C++. The implementation itself is quite efficient and easy to understand, and converting the entire code to C++ should be quite simple, especially since classes are used which is closer to C++ than regular python syntax.

6.2.3 Pose Estimation

The pose estimation implementation in section A.2 is written to test the pose estimation methods. It is not written as a positioning system, and even though some of the code can be used as inspiration in a fully optimized positioning system it cannot all be included. The code fulfills the purpose of testing different methods

against each other, and testing more methods can easily be done by adding them to the list.

6.3 Results

In this section we will discuss the results presented in chapter 5.

6.3.1 Camera calibration

The camera calibration results in section 5.1 are as expected. The calibration matrix in eq. (5.2) has approximately the same focal length in both horizontal and vertical directions, verifying that the pixels are indeed square, see end of section 3.2.2. The focal length is at 17mm, which also verifies the settings presented in section 4.1.2, and the skew element is 0 since the sensor is rectangular. The principal point is also very accurate as it is approximately half the length of the sensor size presented in section 4.1.2. The reprojection error presented in fig. 5.1 gives the same conclusion, as it is very low, and by removing some "bad" images with reprojection results above the mean, we were able to lower the reprojection error by 0.01 pixels. The results can be further improved by adding more images that results in a lower mean reprojection error. The distortion parameters shown in eq. (5.4) also seems to be quite accurate as they are very low. Clearly the tangential distortion, (p_1, p_2), is highly dominated by the radial distortion, (k_1, k_2, k_3), as the radial distortion parameters are many times larger than the radial distortion, see section 3.1.3. The low distortion is expected as the camera is a high-end consumer DSLR, expected to have close to no distortion.

6.3.2 Image set 1

The results from the first image set, see section 5.2.1, are not ideal, but they are however expected. The small size of the QR code makes it difficult to detect and decode from a distance. This is clear from table 5.1 where we can see that the code is not detected in the two images that were the furthest away from the marker. All methods are run on the images, but since the corners of the qr code obviously are coplanar, the three general PnP solvers does not work very well while the P3P solvers and LM Optimization work better, see figs. 5.3, 5.5 and 5.7. From the reprojection error in fig. 5.8 it is quite obvious that the LM-Optimization and the AP3P algorithms work the best, and the CASSC algorithm is more unstable. The poses does also seem to be quite accurate in figs. 5.2, 5.4 and 5.6 where they are illustrated together with the "true" pose. The deviation might be because the measurement of the "true" pose can have errors. This especially seems to be the case in fig. 5.4 where the results of AP3P and LM Optimization are almost equal, and the "true" pose is positioned a small deviation away from them.

6.3.3 Image set 2

From the second image set, the results can be found in section 5.2.2. In these images we had two markers, which in total make up for eight points. The QR codes are not coplanar, but are of the same size of 81mm, see information in table 5.2. As expected, the two images the furthest away does not detect the first QR code, which is the same as in the previous image set. However, since there are one QR code that is closer than the first one, it is able to find points in the to last images. The results are more or less expected, in the two images where both markers was detected the reprojection error from using the general PnP solvers as well as LM Optimization is very low, see figs. 5.11 and 5.13. In the last two images, there was only one code detected, which has coplanar points, and here the AP3P and LM Optimization solvers clearly handles the estimation best, see figs. 5.15 and 5.17. As a last observation, the poses illustrated in figs. 5.10, 5.12, 5.14 and 5.16 indicates again that the true pose of the camera might be poor measurements, and that the estimated poses actually are more accurate than the "true" pose. This is especially indicated in fig. 5.12 where almost all the estimated poses are placed on top of each other, and the "true" pose is offset by a small amount.

6.3.4 Image set 3

The results from the third image set can be found in section 5.2.3. As it is presented in table 5.3 there are two markers in the scene, with different sizes. In image 10.JPG there are only detected one code, which is reasonable since the other code is outside the image area, see fig. A.10. The first code is however bigger than it has been in the previous two image sets, but this does actually seem to lower the accuracy of the pose estimators, as the reprojection error is bigger in all methods when comparing fig. 5.3 and fig. 5.21. In the next four images both markers are detected, and the accuracy seem to be better than in the previous image set, at least if the reprojection errors presented in fig. 5.18 and fig. 5.30 are compared. Another observation is that the accuracy seem to get better when the camera is further away, which might indicate that the accuracy will be better the further away the camera is, until the code can no longer be detected. If this is the case, then the size of the QR code can be optimized depending on how far away the camera is supposed to be. As a final observation on the set the poses seen to indicate that the measurements done for the "true" pose are not accurate enough. It is clear from fig. 5.28 where all the reprojection errors are under 1 pixel, see fig. 5.29, that the "true" pose has an offset from the estimated poses and might actually be less accurate than the estimated poses.

6.3.5 Image set 4

Section 5.2.4 contains the results from image set 4. The scene contained two markers with size 161mm, which were both detected by the QR detection algorithm in all images, see table 5.4. Following the tendencies from the last image set, the accuracy seems to increase when the camera further away from the markers, until it is too far away to be detected at all. This is especially clear from the reprojection errors

shown in fig. 5.40. The "true" pose of the camera presented in the pose illustrations in figs. 5.32, 5.34, 5.36 and 5.38, has the same tendencies as in the previous image sets where it is offset from the estimated poses. Especially fig. 5.38 show that all estimated poses are more or less equal, while the "true" pose is shifted a bit away from them. The reprojection error in this image, see fig. 5.39, is very low, so it is very likely that the estimated poses are more accurate than the "true" pose.

One observation in all the image sets is that the QR detection is very slow. In figs. 5.19, 5.19, 5.31 and 5.41 the scanner uses around 1 second to detect the codes, regardless of the size of the code and the amount of codes in the image. This indicates that the amount of codes and the sizes does not matter for the efficiency of the QR detection. There are two possible reasons for this; the first is that the image quality is too good. Since the resolution in this project is very hight, the QR scanner has too many pixels to scan over. The other possibility is that the scanner is written in python which is not a very efficient language in itself, but the scanner implementation might also be inefficient. The solution might be to rewrite the code to C++ and check if the runtime is lower, or see if there are better and faster implementations for the detection. It might also be a solution use other markers that are easier to detect, or lowering the resolution of the camera itself. As a last suggestion the QR detector is run on the entire image only in the initialization, and after the initialization only check in a small region around the previously detected points. This is however only a solution if the vessel does not move so much that the detection area is very big.

6.3.6 Runtime PnP Methods

The runtime of the different methods is shown after running 100 times on the same input data in fig. 5.42, and they show a clear tendency. The LM optimization algorithm is much slower than the other methods which has approximately the same runtime. The results are verified again in figs. A.19 to A.34 in section A.4 which is the timing of the methods when they were used in the pose estimation of the image sets above.

6.4 Conclusions

In this project we have studied different possibilities for a position estimation system. Concluding from the related work in section 2.3.1 we decided to focus the work around the camera systems currently used on typical ro-ro ferries today, see fig. 1.3 for illustration of the camera placements. We also presented different possible markers for such a system in section 2.3.2, and for this project we chose to use the QR codes as they could be easily recognized by pre-made open source software. For an actual system on board a ferry we would advise to use custom markers, similar to QR but since it does not need to contain a lot of information, it can be made simpler than the QR code. Active markers would be the best because they can easily be detected in any weather conditions. In section 2.4.2 we also presented the *OpenCV* software, this is by far the most used computer vision library and it is easy to use and implement. Hence it is an obvious choice for this project.

From testing we see that the calibration functions implemented in *OpenCV* work very well. Clearly from the discussion in section 6.3.1 the calibration on the camera used in this project worked very well. The resulting camera matrix parameters are sufficiently close to the actual values given by the camera manufacturer.

After running pose estimation on image sets 1-4, we have multiple different implications. The most obvious is that the QR detection runtime is very high, see figs. 5.9, 5.19, 5.31 and 5.41. This needs to be reduced if the system is to run in real time on a vessel, and there are presented some possible solutions to this problem in the previous section. Another observation is that the markers used in such a system needs to be optimized in size with respect to the distance from the camera and the resolution of the image sensor.

The pose estimation algorithms give an interesting result. It seems as the LM Optimization algorithm has the most consistent low reprojection error, and is therefore the most accurate method of the ones presented. It works well both with 4 points and more, and even with coplanar points it works very well compared to the other methods. It seems to be an obvious choice for method in a pose estimation procedure. Consistently through all the images, the LM optimization algorithm gives the lowest reprojection error, and based on visual evaluation of the poses it fits very well with the "true" value. The downside to the LM optimization is that it is quite a lot slower than the other methods, which can be a problem in a very fast system. However the runtime is around 0.001 seconds, so it should really not be a problem for a real-time system.

6.5 Future Work

This project has mainly focused on an introduction to methods and algorithms that can be used in a positioning system based on computer vision. In order to use the methods described in this project, a system for tracking the position in multiple frames has to developed using some kind of visual odometry method. A complete system where all aspects of pose estimation, camera calibration and tracking also needs to be developed. As method for pose estimation it is suggested to use LM optimization, as it is consistently the most accurate estimator. It might also be a good idea to do some more accurate testing of the methods using more markers, and a more accurate measurement of the "true" position of the camera.

We suggest that the software is converted to C++ to lower runtime. C++ is a much more efficient language, and there might be a lot of runtime saved on converting the code. In addition C++ is the most used language in commercial maritime systems, and implementation of parts of a system developed for positioning will be easier if the code is converted.

The detection method has to be improved. This can be done by making a custom marker and making a detection method optimized for the marker. It is suggested that this is looked into in future work. Also the markers them selfs must be optimized to ensure high enough accuracy, and consistent detection. In order to get as realistic results as possible it is also suggested that the systems are tested on cameras that are more similar to the cameras used on the ferries.

Appendices

Appendix A

Code

A.1 Camera Calibration

```
1  import numpy as np
2  import cv2 as cv
3  import glob
4  import matplotlib.pyplot as plt
5
6  #Initialize directories and images
7  calibration_images_directory = "Calibration_images"
8
9  #Settings
10 criteria = (cv.TERM_CRITERIA_EPS + cv.
       TERM_CRITERIA_MAX_ITER, 30, 0.001)
11
12 class cameraCalibration:
13
14   def __init__(self):
15     self.cameraMatrix = None
16     self.distortionCofficients = None
17     self.ret = False
18     self.rvecs = []
19     self.tvecs = []
20     self.objectPoints = []
21     self.imagePoints = []
22     self.error_list = []
23
24   def calibrate(self):
25
26     objp = np.zeros((6*9,3), np.float32)
27     objp[:,:2] = np.mgrid[0:9,0:6].T.reshape(-1,2)
28
29     objpoints = []
30     imgpoints = []
```

```
31
32      images = glob.glob(calibration_images_directory+"/*.
           JPG")
33
34      for fname in images:
35        img = cv.imread(fname)
36
37        gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
38
39        ret, corners = cv.findChessboardCorners(gray, (9,6)
             ,None)
40
41        if ret == True:
42
43          objpoints.append(objp)
44
45          corners2 = cv.cornerSubPix(gray, corners, (11,11)
               , (-1,-1), criteria)
46          imgpoints.append(corners)
47
48      self.objectPoints = objpoints
49      self.imagePoints = imgpoints
50
51      self.ret, self.cameraMatrix, self.
           distortionCofficients, self.rvecs, self.tvecs = cv
           .calibrateCamera(objpoints, imgpoints, gray.shape
           [::-1], None,None)
52
53    def save_parameters(self):
54      np.savetxt('cameraMatrix.out', self.cameraMatrix,
           delimiter = ',')
55      np.savetxt('distortionCofficients.out', self.
           distortionCofficients  , delimiter=',')
56
57    def reprojection_calculation(self):
58      for i in range(len(self.objectPoints)):
59        imgpoints2, _ = cv.projectPoints(self.objectPoints[
           i], self.rvecs[i], self.tvecs[i], self.
           cameraMatrix, self.distortionCofficients)
60        error = cv.norm(self.imagePoints[i],imgpoints2, cv.
           NORM_L2)/len(imgpoints2)
61        self.error_list.append(error)
62
63    def make_figure(self):
64
65      fig,ax = plt.subplots()
```

```
66
67      ax.axhline(np.mean(self.error_list), c = "black")
68      ax.axhline(np.mean([i for i in self.error_list if i <
            np.mean(self.error_list)]), c = "black",
          linestyle="--")
69
70      pictures = range(len(self.error_list))
71
72      chart = plt.bar(pictures,self.error_list,width =0.5,
          tick_label = [str(i) for i in range(1,len(self.
          error_list)+1)])
73
74      plt.grid(False)
75
76      for i in range(len(chart)):
77        if chart[i].get_height() > np.mean(self.error_list)
            :
78          chart[i].set_color('r')
79
80      ax.set_xlim(-1,20)
81      ax.set_ylim(0,0.1)
82
83      ax.set_title("Geometric␣reprojection␣error")
84      ax.set_xlabel("Image")
85      ax.set_ylabel("Error")
86      ax.legend(["Original","Updated"])
87
88      plt.savefig("Calibration_reprojection_error.eps",
          format = "eps", dpi =2000, bbox_inches="tight")
89      plt.show()
90
91  def main():
92
93    calibration = cameraCalibration()
94    calibration.calibrate()
95
96    if calibration.ret:
97      calibration.save_parameters()
98      calibration.reprojection_calculation()
99      calibration.make_figure()
100
101   return 0
102
103 if __name__ == "__main__":
104   main()
```

A.2 Pose Estimation

```
1  import zbar
2  from PIL import Image
3  from PIL import ImageDraw
4  import time
5  import math
6  import settings
7  import numpy as np
8  import matplotlib.pyplot as plt
9  from mpl_toolkits.mplot3d import Axes3D
10 import cv2 as cv
11 import glob
12 from matplotlib.lines import Line2D
13 from matplotlib import rcParams
14
15
16 rcParams['axes.titlepad'] = 30
17
18 PnP_methods = [cv.SOLVEPNP_ITERATIVE,cv.SOLVEPNP_P3P,cv.
      SOLVEPNP_AP3P,cv.SOLVEPNP_EPNP,cv.SOLVEPNP_DLS,cv.
      SOLVEPNP_UPNP]
19
20 Color_list = ['royalblue','darkorange','forestgreen','
      crimson','mediumorchid','sienna']
21
22 #Configuration information
23 Image_directory = "Pose_estimation_images/"
24 QR_square_size_1 = 0.081
25 QR_square_size_2 = 0.161
26
27 Image_name_list_1 = [Image_directory+"1.JPG",
      Image_directory+"2.JPG",Image_directory+"3.JPG",
      Image_directory+"4.JPG",Image_directory+"5.JPG"]
28 QR_First_1 = [[0,1,0],[0,1-QR_square_size_1,0],[
      QR_square_size_1,1-QR_square_size_1,0],[
      QR_square_size_1,1,0]]
29 QR_Second_1 = None
30 True_poses_1 = [np.array
      ([[1,0,0,0],[0,-1,0,0.85],[0,0,-1,1],[0,0,0,1]]),np.
      array([[1,0,0,0],[0,-1,0,0.85],[0,0,-1,2],[0,0,0,1]]),
      np.array
      ([[1,0,0,0],[0,-1,0,0.85],[0,0,-1,3],[0,0,0,1]]),np.
      array
      ([[1,0,0,-1.5],[0,-1,0,0.85],[0,0,-1,2.5],[0,0,0,1]]),
      np.array
```

```
    ([[1,0,0,-1.5],[0,-1,0,0.85],[0,0,-1,4],[0,0,0,1]])]
31
32  Image_name_list_2 = [Image_directory+"6.JPG",
        Image_directory+"7.JPG",Image_directory+"8.JPG",
        Image_directory+"9.JPG"]
33  QR_First_2 = [[0,1,0],[0,1-QR_square_size_1,0],[
        QR_square_size_1,1-QR_square_size_1,0],[
        QR_square_size_1,1,0]]
34  QR_Second_2 = [[-0.5,0.48,0.5],[-0.5,0.48-
        QR_square_size_1,0.5],[-0.5+QR_square_size_1,0.48-
        QR_square_size_1,0.5],[-0.5+QR_square_size_1
        ,0.48,0.5]]
35  True_poses_2 = [np.array
        ([[1,0,0,0],[0,-1,0,0.85],[0,0,-1,2],[0,0,0,1]]),np.
        array([[1,0,0,0],[0,-1,0,0.85],[0,0,-1,3],[0,0,0,1]]),
        np.array
        ([[1,0,0,-1.5],[0,-1,0,0.85],[0,0,-1,2.5],[0,0,0,1]]),
        np.array
        ([[1,0,0,-1.5],[0,-1,0,0.85],[0,0,-1,4],[0,0,0,1]])]
36
37  Image_name_list_3 = [Image_directory+"10.JPG",
        Image_directory+"11.JPG",Image_directory+"12.JPG",
        Image_directory+"13.JPG",Image_directory+"14.JPG"]
38  QR_First_3 = [[0,1,0],[0,1-QR_square_size_2,0],[
        QR_square_size_2,1-QR_square_size_2,0],[
        QR_square_size_2,1,0]]
39  QR_Second_3 = [[-0.5,0.48,0.5],[-0.5,0.48-
        QR_square_size_1,0.5],[-0.5+QR_square_size_1,0.48-
        QR_square_size_1,0.5],[-0.5+QR_square_size_1
        ,0.48,0.5]]
40  True_poses_3 = [np.array
        ([[1,0,0,0],[0,-1,0,0.85],[0,0,-1,1],[0,0,0,1]]),np.
        array([[1,0,0,0],[0,-1,0,0.85],[0,0,-1,2],[0,0,0,1]]),
        np.array
        ([[1,0,0,0],[0,-1,0,0.85],[0,0,-1,3],[0,0,0,1]]),np.
        array
        ([[1,0,0,-1.5],[0,-1,0,0.85],[0,0,-1,2.5],[0,0,0,1]]),
        np.array
        ([[1,0,0,-1.5],[0,-1,0,0.85],[0,0,-1,4],[0,0,0,1]])]
41
42  Image_name_list_4 = [Image_directory+"15.JPG",
        Image_directory+"16.JPG",Image_directory+"17.JPG",
        Image_directory+"18.JPG"]
43  QR_First_4 = [[0,1,0],[0,1-QR_square_size_2,0],[
        QR_square_size_2,1-QR_square_size_2,0],[
        QR_square_size_2,1,0]]
```

```
44 QR_Second_4 = [[-0.5,0.48,0.5],[-0.5,0.48-
      QR_square_size_2,0.5],[-0.5+QR_square_size_2,0.48-
      QR_square_size_2,0.5],[-0.5+QR_square_size_2
      ,0.48,0.5]]
45 True_poses_4 = [np.array
      ([[1,0,0,0],[0,-1,0,0.85],[0,0,-1,2],[0,0,0,1]]),np.
      array([[1,0,0,0],[0,-1,0,0.85],[0,0,-1,3],[0,0,0,1]]),
      np.array
      ([[1,0,0,-1.5],[0,-1,0,0.85],[0,0,-1,2.5],[0,0,0,1]]),
      np.array
      ([[1,0,0,-1.5],[0,-1,0,0.85],[0,0,-1,4],[0,0,0,1]])]
46
47
48 def main():
49
50   First_QR = QR_First_2
51   Second_QR = QR_Second_2
52   Image_list = Image_name_list_2
53   True_poses = True_poses_2
54
55   pose_estimation_results = []
56   pose_estimation_timing_results = []
57   pose_estimation_reprojection_errors = []
58   QR_detection_timing_results = []
59   image_names_detected = []
60
61   timing_test = False
62
63   for i in range(len(Image_list)):
64     print("Running␣poseEstimator␣on:␣",Image_list[i])
65     opened_image = Image.open(Image_list[i]).convert('L')
66     poseEstimator = poseEstimation(image = opened_image,
         image_name=Image_list[i], QR_First = First_QR,
         QR_Second = Second_QR, True_pose = True_poses[i])
67
68     if timing_test:
69       poseEstimator.pose_estimations_timinig_test(
           print_result = True)
70
71     else:
72       poseEstimator.pose_estimations_accuracy_tests(
           print_result = True)
73       if len(poseEstimator.detected_QR_codes)>0:
74         pose_estimation_results.append(poseEstimator.
             estimated_poses)
75         pose_estimation_reprojection_errors.append(
```

```
            poseEstimator.reprojection_errors)
76          pose_estimation_timing_results.append(
              poseEstimator.timing_pose_estimation)
77          QR_detection_timing_results.append(poseEstimator.
              QR_detection_timing)
78          image_names_detected.append(Image_list[i])
79      del poseEstimator
80
81    if not timing_test:
82      display_reprojection_errors(
          pose_estimation_reprojection_errors,
          image_names_detected)
83      display_QR_detection_timing(
          QR_detection_timing_results, image_names_detected)
84
85  def display_reprojection_errors(reprojection_errors,
       image_name_list):
86    errors = [[reprojection_errors[i][j] for i in range(len
        (reprojection_errors))] for j in range(len(
        reprojection_errors[0]))]
87    fig , ax = plt.subplots()
88    ax.grid(True)
89    bar_width = 0.1
90    for i in range(len(errors)):
91      plt.bar([number+((i-2.5)*bar_width) for number in
          range(1,len(errors[i])+1)],errors[i], width =
          bar_width, color = Color_list[i])
92    ax.legend(["LM␣Optimization","CASSC","AP3P","EPnP","DLS
        ","UPnP"],fontsize = 8)
93    ax.set_title("Geometric␣Reprojection␣error")
94    ax.set_xlabel("Image␣name")
95    ax.set_ylabel("Error")
96    plt.xticks(range(1,len(reprojection_errors)+1))
97    ax.set_xticklabels([image_name[23:] for image_name in
        image_name_list])
98    for tick in ax.xaxis.get_major_ticks():
99      tick.label.set_fontsize(8)
100   for tick in ax.yaxis.get_major_ticks():
101     tick.label.set_fontsize(8)
102   plt.xlim(0,len(reprojection_errors)+2)
103   plt.ylim(0,5)
104   plt.savefig("Reprojection_error_all_images.eps",format
        = "eps", dpi =2000, bbox_inches="tight")
105   plt.show()
106
107 def display_QR_detection_timing(QR_detection_timing,
```

```
    image_name_list):
108   fig , ax = plt.subplots()
109   ax.grid(True)
110   for i in range(len(QR_detection_timing)):
111     plt.bar(i+1,QR_detection_timing[i],width = 0.5)
112   ax.set_title("QR Code Detection Runtime")
113   ax.set_xlabel("Image name")
114   ax.set_ylabel("Time (Seconds)")
115   plt.xticks(range(1,len(QR_detection_timing)+1))
116   ax.set_xticklabels([image_name[23:] for image_name in
        image_name_list])
117   plt.xticks(range(1,len(QR_detection_timing)+1))
118   for tick in ax.xaxis.get_major_ticks():
119     tick.label.set_fontsize(6)
120   for tick in ax.yaxis.get_major_ticks():
121     tick.label.set_fontsize(6)
122   plt.xlim(0,len(QR_detection_timing)+1)
123   plt.ylim(0,2)
124   plt.savefig("Timing_qr_detection.eps",format = "eps",
        dpi =2000, bbox_inches="tight")
125   plt.show()
126
127
128
129 class poseEstimation:
130
131   def __init__(self, image, image_name, QR_First,
        QR_Second, True_pose):
132     self.camera = 1
133     self.image = image
134     self.image_name = image_name
135     self.estimated_poses = []
136     self.timing_pose_estimation = []
137     self.detected_QR_codes = []
138     self.scene_points = []
139     self.image_points = []
140     self.QR_First = QR_First
141     self.QR_Second = QR_Second
142     self.True_pose = True_pose
143     self.QR_detection_timing = None
144     self.camera_matrix = None
145     self.distortion_cofficients = None
146     self.reprojection_errors = []
147
148   # Solver class
149   class method:
```

```
150
151     def __init__(self, method, image_points, scene_points
            , camera_matrix, distortion_cofficients):
152       self.method = method
153       self.image_points = image_points
154       self.scene_points = scene_points
155       self.camera_matrix = camera_matrix
156       self.distortion_cofficients =
             distortion_cofficients
157       self.estimated_pose = None
158       self.estimation_timer = 0
159       self.rvec = None
160       self.tvec = None
161
162     def estimate_pose(self):
163       if not len(self.image_points)==0:
164         estimation_start_time = time.clock()
165
166         if (self.method == cv.SOLVEPNP_P3P) or (self.
               method == cv.SOLVEPNP_AP3P):
167           self.image_points = np.concatenate((self.
                 image_points[:2],self.image_points[len(self.
                 image_points)-2:]),axis = 0)
168           self.scene_points = np.concatenate((self.
                 scene_points[:2],self.scene_points[len(self.
                 scene_points)-2:]),axis = 0)
169
170         image_points = np.ascontiguousarray(self.
               image_points[:,:2]).reshape((self.image_points
               .shape[0],1,2))
171         scene_points = np.ascontiguousarray(self.
               scene_points[:,:3]).reshape((self.scene_points
               .shape[0],1,3))
172
173         ret, self.rvec, self.tvec = cv.solvePnP(
               scene_points, image_points, self.camera_matrix
               , self.distortion_cofficients, flags = self.
               method)
174
175         if ret:
176
177           Rmat, jacobian = cv.Rodrigues(self.rvec)
178
179           T_cs = np.array([[Rmat[0][0],Rmat[0][1],Rmat
                 [0][2],self.tvec[0][0]],[Rmat[1][0],Rmat
                 [1][1],Rmat[1][2],self.tvec[1][0]],[Rmat
```

```
            [2][0],Rmat[2][1],Rmat[2][2],self.tvec
            [2][0]],[0,0,0,1]])
180
181           self.estimated_pose = np.linalg.inv(T_cs)
182
183         self.estimation_timer = time.clock()-
              estimation_start_time
184
185   # Supporting fucntions
186   def detect_QR_codes(self):
187     #Object that scans for the corners of a QR-code and
          measures the distance between them
188     scanner = zbar.Scanner()
189     #Use the scanner on the image to find the qr-code
          location
190     self.detected_QR_codes = scanner.scan(self.image)
191
192   def extract_image_and_scene_points(self):
193
194     for code in self.detected_QR_codes:
195       image_coordinates = []
196       scene_coordinates = []
197
198       if code.data == b'First':
199         for image_point in code.position:
200           image_coordinates.append(image_point)
201         for scene_point in self.QR_First:
202           scene_coordinates.append(scene_point)
203       elif code.data == b'Second':
204         for position in code.position:
205           image_coordinates.append(position)
206         for scene_point in self.QR_Second:
207           scene_coordinates.append(scene_point)
208
209       if len(image_coordinates) != 0:
210         for image_point in image_coordinates:
211           self.image_points.append((image_point[0],
                image_point[1]))
212         for scene_point in scene_coordinates:
213           self.scene_points.append((scene_point[0],
                scene_point[1],scene_point[2]))
214
215       del image_coordinates
216       del scene_coordinates
217
218     self.image_points = np.array(self.image_points,dtype
```

```
      = np.float32)
219     self.scene_points = np.array(self.scene_points,dtype
      = np.float64)
220
221   def load_intrinsics(self):
222     #If multiple cameras, a configuration is needed for
      different cameras
223     if self.camera == 1:
224
225       self.camera_matrix = np.loadtxt('../Camera␣
        Calibration/cameraMatrix.out', delimiter = ',')
226       self.distortion_cofficients = np.asarray([np.
        loadtxt('../Camera␣Calibration/
        distortionCofficients.out', delimiter=',')])
227
228   def reprojection_calculation(self, solver):
229
230     image_points2, _ = cv.projectPoints(self.scene_points
      , solver.rvec, solver.tvec, self.camera_matrix,
      self.distortion_cofficients)
231     image_points2 = np.array([point[0] for point in
      image_points2],dtype = np.float32)
232
233     error = cv.norm(self.image_points, image_points2, cv.
      NORM_L2)/len(image_points2)
234     self.reprojection_errors.append(error)
235
236   # Timing tests on PnP solvers
237   def pose_estimations_timinig_test(self, print_result =
     True):
238
239     self.detect_QR_codes()
240
241     if len(self.detected_QR_codes) != 0:
242
243       self.extract_image_and_scene_points()
244       self.load_intrinsics()
245
246       for iteration in range(100):
247
248         timers = []
249
250         for method in PnP_methods:
251
252           solver = self.method(method, self.image_points.
            copy(), self.scene_points.copy(), self.
```

```
                camera_matrix, self.distortion_cofficients)
253           solver.estimate_pose()
254
255           timers.append(solver.estimation_timer)
256
257           del solver
258
259         self.timing_pose_estimation.append(timers.copy())
260
261         del timers[:]
262
263       if print_result:
264         self.plot_timing_test_results()
265
266   def plot_timing_test_results(self):
267     fig , ax = plt.subplots()
268     ax.grid(True)
269
270     for iteration in range(len(PnP_methods)):
271       plt.plot(range(1,len(self.timing_pose_estimation)
             +1),[l[iteration] for l in self.
             timing_pose_estimation],color = Color_list[
             iteration])
272
273     ax.set_title("Pose␣Estimation␣Runtime")
274     ax.legend(["LM␣Optimization","CASSC","AP3P","EPnP","
        DLS","UPnP"],fontsize = 8)
275     ax.set_xlabel("Iteration")
276     ax.set_ylabel("Time␣(seconds)")
277
278     plt.xlim(0,100)
279     plt.ylim(0,0.002)
280
281     plt.savefig("Timing_test_"+str(self.image_name[23:len
        (self.image_name)-4])+".eps",format = "eps", dpi
        =2000, bbox_inches="tight")
282
283     plt.show()
284
285   # Accuracy tests on PnP solvers
286   def pose_estimations_accuracy_tests(self, print_result
       = True):
287
288     qr_timer_start = time.clock()
289     self.detect_QR_codes()
290     self.extract_image_and_scene_points()
```

```
291
292     if len(self.detected_QR_codes) >0:
293       self.QR_detection_timing = time.clock()-
             qr_timer_start
294       self.load_intrinsics()
295
296       for method in PnP_methods:
297
298         solver = self.method(method, self.image_points.
              copy(), self.scene_points.copy(), self.
              camera_matrix, self.distortion_cofficients)
299
300         solver.estimate_pose()
301
302         if solver.estimated_pose is not None:
303           self.estimated_poses.append(solver.
                estimated_pose.copy())
304         else:
305           self.estimated_poses.append(solver.
                estimated_pose)
306
307         self.timing_pose_estimation.append(solver.
              estimation_timer)
308
309         self.reprojection_calculation(solver)
310
311         del solver
312
313       if print_result:
314         self.display_estimation_accuracy_tests_results()
315     else:
316       print("No codes detected.")
317
318   def display_estimation_accuracy_tests_results(self):
319     # Print QR detection time
320     print("It took ",self.QR_detection_timing," seconds
          to detect ",len(self.detected_QR_codes)," QR codes
          in the image.")
321
322     # Display time it took to estimate pose
323     fig , ax = plt.subplots()
324     ax.grid(True)
325     for method in range(len(self.timing_pose_estimation))
          :
326       ax.bar(method+1,self.timing_pose_estimation[method
            ], width = 0.5,color = Color_list[method])
```

```
327     ax.set_title("Timing␣bar␣plot,␣"+str(self.image_name
            [23:]))
328     ax.set_xlabel("Method")
329     ax.set_ylabel("Time␣(seconds)")
330     ax.set_xticks(range(1,len(PnP_methods)+1))
331     ax.set_xticklabels(["LM␣Optimization","CASSC","AP3P",
            "EPnP","DLS","UPnP"])
332     for tick in ax.xaxis.get_major_ticks():
333       tick.label.set_fontsize(8)
334     for tick in ax.yaxis.get_major_ticks():
335       tick.label.set_fontsize(8)
336     plt.xlim(0,len(PnP_methods)+1)
337     plt.ylim(0,0.002)
338     plt.savefig("Timing_accuracy_test_"+str(self.
            image_name[23:len(self.image_name)-4])+".eps",
            format = "eps", dpi =5000)#, bbox_inches="tight")
339     #plt.close()
340     plt.show()
341
342     # Display reprojection error
343     fig , ax = plt.subplots()
344     ax.grid(True)
345     for method in range(len(self.reprojection_errors)):
346       ax.bar(method+1,self.reprojection_errors[method],
              width = 0.5,color = Color_list[method])
347     ax.set_title("Reprojection␣Error␣Bar␣Chart,␣"+str(
            self.image_name[23:]))
348     ax.set_xlabel("Method")
349     ax.set_ylabel("Reprojection␣Error")
350     ax.set_xticks(range(1,len(PnP_methods)+1))
351     ax.set_xticklabels(["LM␣Optimization","CASSC","AP3P",
            "EPnP","DLS","UPnP"])
352     for tick in ax.xaxis.get_major_ticks():
353       tick.label.set_fontsize(8)
354     for tick in ax.yaxis.get_major_ticks():
355       tick.label.set_fontsize(8)
356     plt.xlim(0,len(PnP_methods)+1)
357     plt.ylim(0,5)
358     plt.savefig("Reprojection_accuracy_test_"+str(self.
            image_name[23:len(self.image_name)-4])+".eps",
            format = "eps", dpi =2000, bbox_inches="tight")
359     plt.show()
360     #plt.close()
361
362     # Display actual pose estimates
363     figure = plt.figure()
```

```
364     ax = figure.add_subplot(111, projection='3d')
365     ax.set_xlim3d(-3,3)
366     ax.set_xlabel('x')
367     ax.set_ylim3d(0,3)
368     ax.set_ylabel('y')
369     ax.set_zlim3d(0,3)
370     ax.set_zlabel('z')
371     ax.view_init(120,-65)
372
373     #Scene coordinate system
374     x1 = np.array([0,0,0,1])
375     x2 = np.array([.3,0,0,1])
376     y1 = np.array([0,0,0,1])
377     y2 = np.array([0,.3,0,1])
378     z1 = np.array([0,0,0,1])
379     z2 = np.array([0,0,.3,1])
380
381     ax.plot([x1[0],x2[0]],[x1[1],x2[1]],[x1[2],x2[2]],
          color = 'red')
382     ax.plot([y1[0],y2[0]],[y1[1],y2[1]],[y1[2],y2[2]],
          color = 'green')
383     ax.plot([z1[0],z2[0]],[z1[1],z2[1]],[z1[2],z2[2]],
          color = 'blue')
384
385     #QR code position
386     x1_qr = self.QR_First[0][0]
387     x2_qr = self.QR_First[2][0]
388     y1_qr = self.QR_First[0][1]
389     y2_qr = self.QR_First[2][1]
390     ax.plot([x1_qr,x1_qr],[y1_qr,y2_qr],[0,0],color = '
          black')
391     ax.plot([x1_qr,x2_qr],[y2_qr,y2_qr],[0,0],color = '
          black')
392     ax.plot([x2_qr,x2_qr],[y2_qr,y1_qr],[0,0],color = '
          black')
393     ax.plot([x2_qr,x1_qr],[y1_qr,y1_qr],[0,0],color = '
          black')
394     if self.QR_Second is not None:
395       x1_qr = self.QR_Second[0][0]
396       x2_qr = self.QR_Second[2][0]
397       y1_qr = self.QR_Second[0][1]
398       y2_qr = self.QR_Second[2][1]
399       z1_qr = self.QR_Second[0][2]
400       z2_qr = self.QR_Second[2][2]
401       ax.plot([x1_qr,x1_qr],[y1_qr,y2_qr],[z1_qr,z2_qr],
            color = 'black')
```

```
402       ax.plot([x1_qr,x2_qr],[y2_qr,y2_qr],[z1_qr,z2_qr],
              color = 'black')
403       ax.plot([x2_qr,x2_qr],[y2_qr,y1_qr],[z1_qr,z2_qr],
              color = 'black')
404       ax.plot([x2_qr,x1_qr],[y1_qr,y1_qr],[z1_qr,z2_qr],
              color = 'black')
405       ax.plot([x2_qr,x2_qr],[y2_qr,y1_qr],[z1_qr,z2_qr],
              color = 'black')
406       ax.plot([x2_qr,x1_qr],[y1_qr,y1_qr],[z1_qr,z2_qr],
              color = 'black')
407
408     #Estimated poses
409     for i in range(len(self.estimated_poses)):
410       if self.estimated_poses[i] is not None:
411         x1_c = np.dot(self.estimated_poses[i],x1)
412         x2_c = np.dot(self.estimated_poses[i],x2)
413         y1_c = np.dot(self.estimated_poses[i],y1)
414         y2_c = np.dot(self.estimated_poses[i],y2)
415         z1_c = np.dot(self.estimated_poses[i],z1)
416         z2_c = np.dot(self.estimated_poses[i],z2)
417
418         ax.plot([x1_c[0],x2_c[0]],[x1_c[1],x2_c[1]],[x1_c
              [2],x2_c[2]],color = Color_list[i])
419         ax.plot([y1_c[0],y2_c[0]],[y1_c[1],y2_c[1]],[y1_c
              [2],y2_c[2]],color = Color_list[i])
420         ax.plot([z1_c[0],z2_c[0]],[z1_c[1],z2_c[1]],[z1_c
              [2],z2_c[2]],color = Color_list[i])
421
422     x1_t = np.dot(self.True_pose,x1)
423     x2_t = np.dot(self.True_pose,x2)
424     y1_t = np.dot(self.True_pose,y1)
425     y2_t = np.dot(self.True_pose,y2)
426     z1_t = np.dot(self.True_pose,z1)
427     z2_t = np.dot(self.True_pose,z2)
428
429     ax.plot([x1_t[0],x2_t[0]],[x1_t[1],x2_t[1]],[x1_t[2],
          x2_t[2]],color = 'black')
430     ax.plot([y1_t[0],y2_t[0]],[y1_t[1],y2_t[1]],[y1_t[2],
          y2_t[2]],color = 'black')
431     ax.plot([z1_t[0],z2_t[0]],[z1_t[1],z2_t[1]],[z1_t[2],
          z2_t[2]],color = 'black')
432
433     custom_lines = [Line2D([0], [0], color=Color_list[0],
            lw=4),
434             Line2D([0], [0], color=Color_list[1], lw=4),
435             Line2D([0], [0], color=Color_list[2], lw=4),
```

```
436             Line2D([0], [0], color=Color_list[3], lw=4),
437             Line2D([0], [0], color=Color_list[4], lw=4),
438             Line2D([0], [0], color=Color_list[5], lw=4),
439             Line2D([0], [0], color='black', lw=4)]
440
441
442     ax.set_title("Pose␣Estimation,␣"+str(self.image_name
            [23:]))
443     ax.legend(custom_lines,["LM␣Optimization","P3P","AP3P
            ","EPNP","DLS","UPNP","True"],fontsize = 8)
444     plt.savefig("Pose_estimation_"+str(self.image_name
            [23:len(self.image_name)-4])+".eps",format = "eps"
            , dpi =2000, bbox_inches="tight")
445     plt.show()
446     plt.close()
447
448
449
450 if __name__ == '__main__':
451   main()
```

A.3 Images

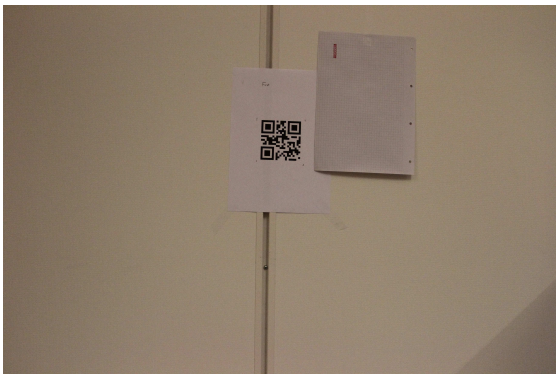

Figure A.1: Image 1

Figure A.2: Image 2

Figure A.3: Image 3

Figure A.4: Image 4

Figure A.5: Image 5

Figure A.6: Image 6

Figure A.7: Image 7

Figure A.8: Image 8

Figure A.9: Image 9

Figure A.10: Image 10

Figure A.11: Image 11

Figure A.12: Image 12

Figure A.13: Image 13

Figure A.14: Image 14

Figure A.15: Image 15

Figure A.16: Image 16

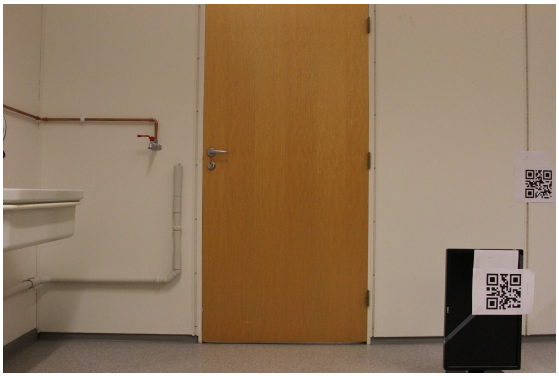

Figure A.17: Image 17

Figure A.18: Image 18

A.4 Runtime of PnP Methods

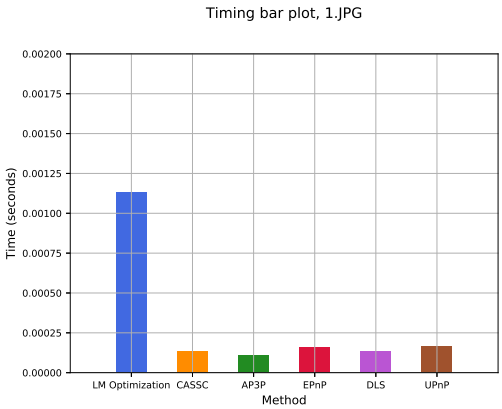

Figure A.19: Runtime of all methods ran on image 1

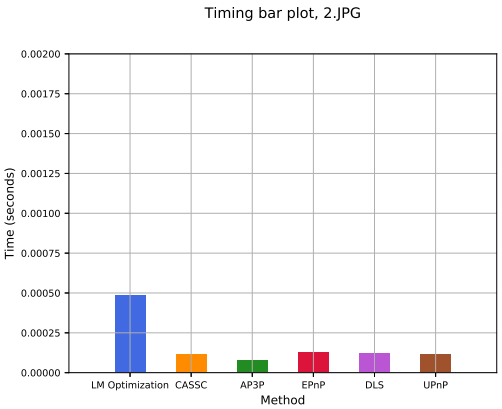

Figure A.20: Runtime of all methods ran on image 2

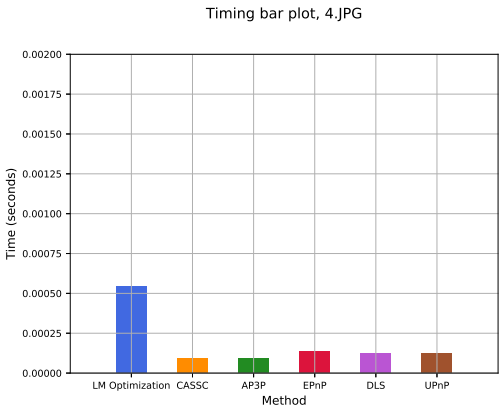

Figure A.21: Runtime of all methods ran on image 4

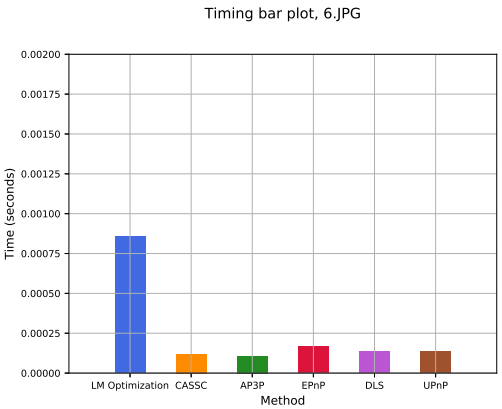

Figure A.22: Runtime of all methods ran on image 6

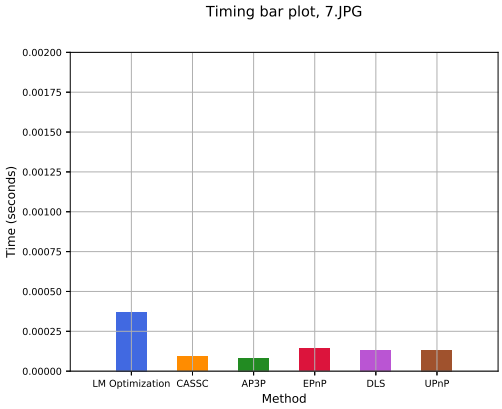

Figure A.23: Runtime of all methods ran on image 7

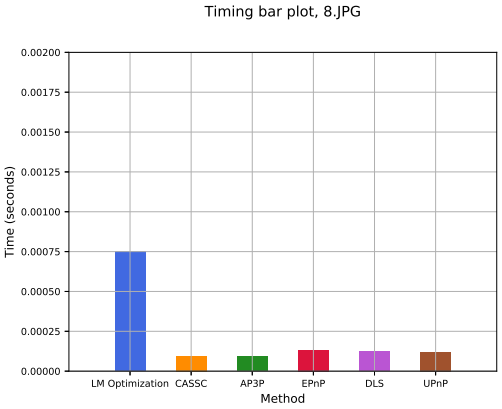

Figure A.24: Runtime of all methods ran on image 8

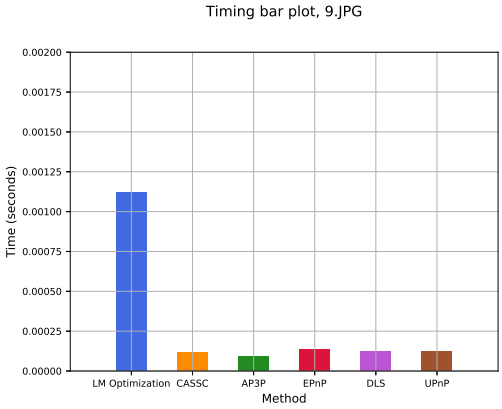

Figure A.25: Runtime of all methods ran on image 9

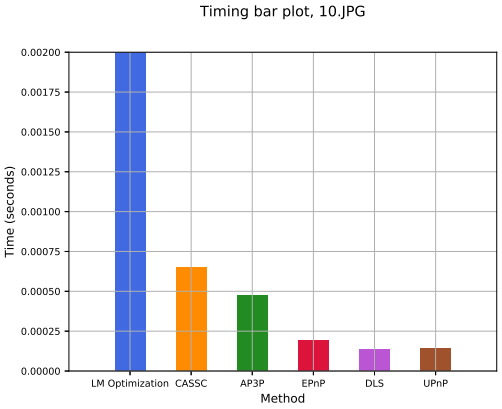

Figure A.26: Runtime of all methods ran on image 10

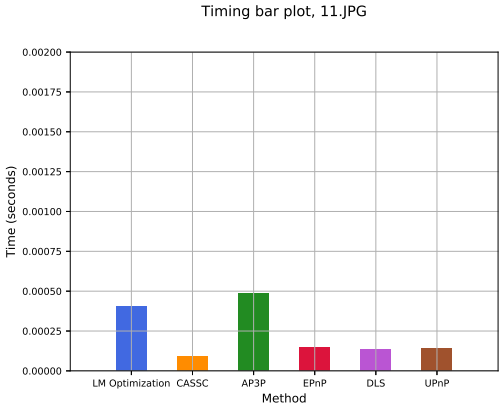

Figure A.27: Runtime of all methods ran on image 11

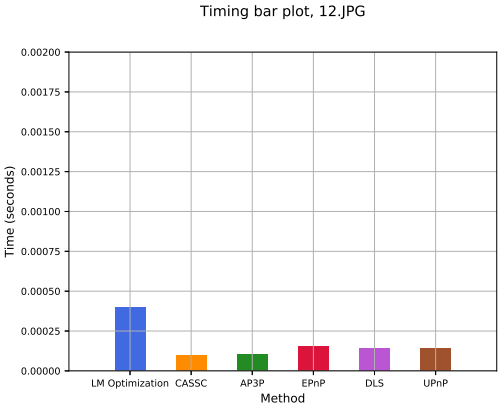

Figure A.28: Runtime of all methods ran on image 12

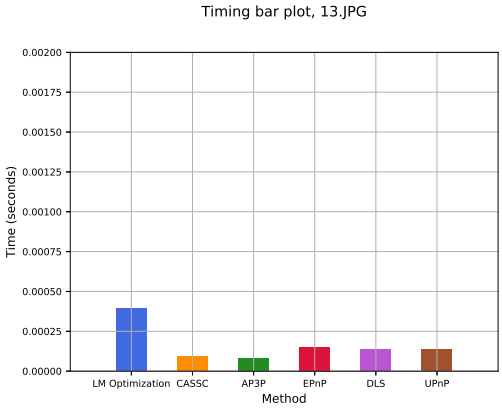

Figure A.29: Runtime of all methods ran on image 13

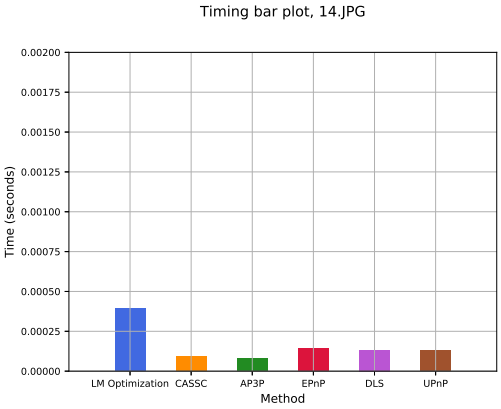

Figure A.30: Runtime of all methods ran on image 14

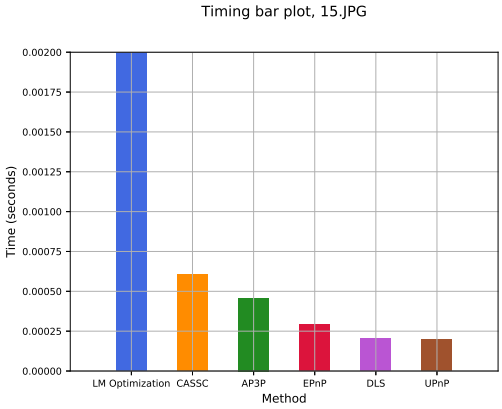

Figure A.31: Runtime of all methods ran on image 15

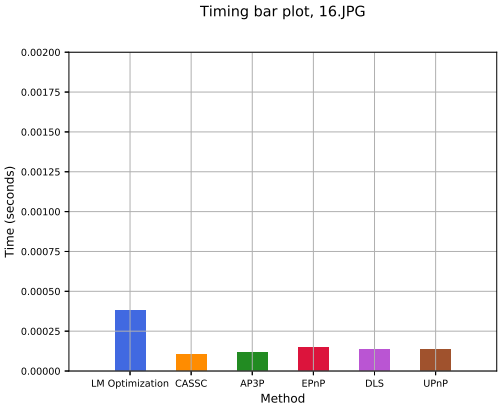

Figure A.32: Runtime of all methods ran on image 16

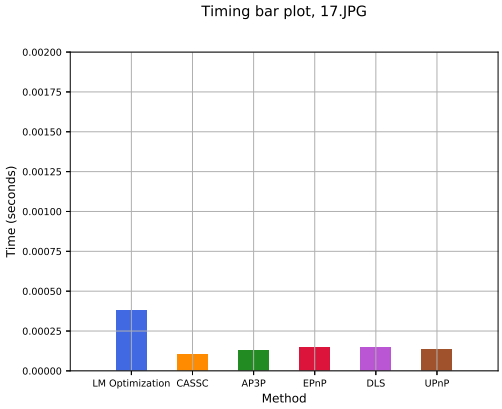

Figure A.33: Runtime of all methods ran on image 17

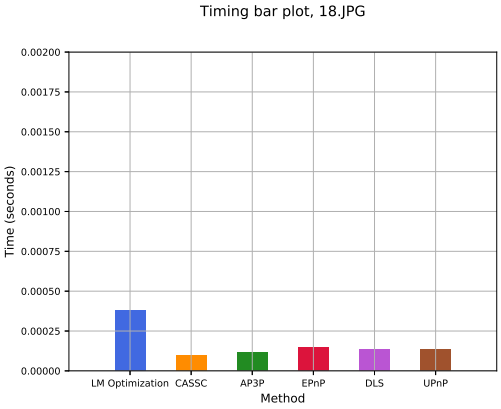

Figure A.34: Runtime of all methods ran on image 18

References

[1] Duane C. Brown. "Close-range camera calibration". In: *PHOTOGRAMMETRIC ENGINEERING* 37.8 (1971), pp. 855–866.

[2] J. Canny. "A Computational Approach to Edge Detection". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (Nov. 1986), pp. 679–698. ISSN: 0162-8828. DOI: 10.1109/TPAMI.1986.4767851.

[3] CPN Canon Professional Network. *Capturing the image CCD and CMOS sensors*. 2017. URL: https://cpn.canon-europe.com/content/education/infobank/capturing_the_image/ccd_and_cmos_sensors.do. (Accessed: 31.10.2018).

[4] Chu-Song Chen and Wen-Yan Chang. "On pose recovery for generalized visual sensors". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.7 (July 2004), pp. 848–861. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2004.34.

[5] Mahashreveta Choudhary. *How GNSS Works?* Jan. 3, 2018. URL: https://www.geospatialworld.net/blogs/how-gnss-works/. (Accessed: 08.12.2018).

[6] Peter Corke. *Robotics, Vision and Control : Fundamental Algorithms In MATLAB® Second, Completely Revised, Extended And Updated Edition*. eng. 2nd ed. 2017. Vol. 118. Springer Tracts in Advanced Robotics. 2017. ISBN: 3-319-54413-6.

[7] Wärtsilä Corporation. *Wärtsilä successfully tests remote control ship operating capability*. Mar. 14, 2018. URL: https://www.wartsila.com/media/news/01-09-2017-wartsila-successfully-tests-remote-control-ship-operating-capability. (Accessed: 07.12.2018).

[8] The Engineer. *Falco makes world's first autonomous ferry crossing*. URL: https://www.theengineer.co.uk/falco-autonomous-ferry-rolls-royce/. (Accessed: 17.12.2018).

[9] O. D. Faugeras, Q. -T. Luong, and S. J. Maybank. "Camera self-calibration: Theory and experiments". In: *Computer Vision — ECCV'92*. Ed. by G. Sandini. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 321–334. ISBN: 978-3-540-47069-4.

[10] Fjord1. *Våre Fartøy, MF GLoppefjord*. URL: https://www.fjord1.no/Om-Fjord1/Vaare-fartoey. (Accessed: 17.12.2018).

[11] Wolfgang Förstner and Bernhard P Wrobel. *Photogrammetric Computer Vision: Statistics, Geometry, Orientation and Reconstruction*. eng. Vol. 11. Geometry and Computing. Cham: Springer International Publishing, 2016. ISBN: 9783319115498.

[12] Xiao-Shan Gao et al. "Complete Solution Classification for the Perspective-Three-Point Problem". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 25.8 (Aug. 2003), pp. 930–943. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2003.1217599. URL: https://doi.org/10.1109/TPAMI.2003.1217599.

[13] Chris Harris and Mike Stephens. "A combined corner and edge detector". In: *In Proc. of Fourth Alvey Vision Conference*. 1988, pp. 147–151.

[14] Andrew J. Hawkins. *Uber Halts self-driving tests after pedestrian killed in Arizona*. URL: https://www.theverge.com/2018/3/19/17139518/uber-self-driving-car-fatal-crash-tempe-arizona. (Accessed: 07.12.2018).

[15] J. A. Hesch and S. I. Roumeliotis. "A Direct Least-Squares (DLS) method for PnP". In: *2011 International Conference on Computer Vision*. Nov. 2011, pp. 383–390. DOI: 10.1109/ICCV.2011.6126266.

[16] Robert J. Holt and Arun N. Netravali. "Camera Calibration Problem: Some New Results". In: *CVGIP: Image Underst.* 54.3 (Oct. 1991), pp. 368–383. ISSN: 1049-9660. DOI: 10.1016/1049-9660(91)90037-P. URL: http://dx.doi.org/10.1016/1049-9660(91)90037-P.

[17] Flight Light Inc. *Obstruction Lights and Controls — FAA L-810*. 2018. URL: https://flightlight.com/products/obstruction-lights-controls-faa-l-810/. (Accessed: 15.12.2018).

[18] Tong Ke and Stergios Roumeliotis. "An efficient algebraic solution to the perspective-three-point problem". English (US). In: 2017-January (Nov. 2017), pp. 4618–4626. DOI: 10.1109/CVPR.2017.491.

[19] Wave laboratory. *Real-time Filtering of Snow from Lidar Point Clouds*. Feb. 12, 2018. URL: http://wavelab.uwaterloo.ca/?weblizar_portfolio=real-time-filtering-of-snow-from-lidar-point-clouds. (Accessed: 09.12.2018).

[20] Vincent Lepetit, Francesc Moreno-Noguer, and Pascal Fua. "EPnP: An Accurate O(n) Solution to the PnP Problem". In: *International Journal Of Computer Vision* 81 (2009), pp. 155–166.

[21] William Lowrie. Cambridge University Press, 2007, p. 281. ISBN: 978-0-521-85902-8. URL: https://app.knovel.com/hotlink/toc/id:kpFGE00004/fundamentals-geophysics/fundamentals-geophysics.

[22] Kongsberg Maritime. *Autonomous ship project, key facts about YARA Birkeland*. 2018. URL: https://www.km.kongsberg.com/ks/web/nokbg0240.nsf/AllWeb/4B8113B707A50A4FC125811D00407045?OpenDocument. (Accessed: 18.12.2018).

[23] Novatel. *GNSS Error Sources*. URL: https://www.novatel.com/an-introduction-to-gnss/chapter-4-gnss-error-sources/error-sources/. (Accessed: 06.12.2018).

[24] OpenCV. *Camera Calibration*. 2018. URL: https://docs.opencv.org/3.4.3/dc/dbb/tutorial_py_calibration.html. (Accessed: 17.09.2018).

[25] Volvo Penta. *Volvo Penta unveils pioneering self-docking yacht technology*. 2018. URL: https://www.volvopenta.com/marineleisure/en-en/news/2018/jun/volvo-penta-unveils-pioneering-self-docking-yacht-technology.html. (Accessed: 26.10.2018).

[26] *Rolls-Royce to supply first automatic crossing system to Norwegian ferry company Fjord1*. 2016. URL: https://www.rolls-royce.com/media/press-releases/2016/18-10-2016-rr-to-supply-first-automatic-crossing-system-to-norwegian-ferry-company-fjord1.aspx. (Accessed: 05.12.2018).

[27] E. Rublee et al. "ORB: An efficient alternative to SIFT or SURF". In: *2011 International Conference on Computer Vision*. Nov. 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544.

[28] Adrián Peñate Sánchez, Juan Andrade-Cetto, and Francesc Moreno-Noguer. "Exhaustive Linearization for Robust Camera Pose and Focal Length Estimation." In: *IEEE Trans. Pattern Anal. Mach. Intell.* 35.10 (2013), pp. 2387–2400. URL: http://dblp.uni-trier.de/db/journals/pami/pami35.html#SanchezAM13.

[29] *Ship Safety Standards*. URL: http://www.emsa.europa.eu/implementation-tasks/ship-safety-standards.html. (Accessed: 08.12.2018).

[30] Asle Skredderberget. *The first ever zero emission, autonomous ship*. Sept. 1, 2017. URL: https://www.yara.com/knowledge-grows/game-changer-for-the-environment/. (Accessed: 07.12.2018).

[31] Tore Stensvold. *Verdens første helt autonome fergeseilas gjennomført - teknologien er 100 prosent klar*. 2018. URL: https://www.tu.no/artikler/verdens-forste-helt-autonome-fergeseilas-gjennomfort-teknologien-er-100-prosent-klar/452610. (Accessed: 03.12.2018).

[32] Peter Sturm et al. "Camera Models and Fundamental Concepts Used in Geometric Computer Vision". eng. In: *Foundations and Trends in Computer Graphics and Vision* 6.1-2 (2011). ISSN: 1572-2740. URL: http://hal.inria.fr/docs/00/59/02/69/PDF/sturm-ftcgv-2011.pdf.

[33] Richard Szeliski. *Computer Vision: Algorithms and Applications*. eng. Texts in Computer Science. London: Springer London, 2011. ISBN: 978-1-84882-934-3.

[34] Opencv dev team. *Camera Calibration and 3D Reconstruction*. 2014. URL: https://docs.opencv.org/3.0-last-rst/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#solvepnp. (Accessed: 03.12.2018).

[35] Opencv dev team. *OpenCV*. URL: https://opencv.org/. (Accessed: 14.12.2018).

[36] Opencv dev team. *OpenCV documentation*. 2014. URL: https://docs.opencv.org/3.0-last-rst/index.html. (Accessed: 27.08.2018).

[37] *The Hopkins Beast*. 1965. URL: http://www.frc.ri.cmu.edu/~hpm/talks/revo.slides/1960.html. (Accessed: 04.12.2018).

[38] Mark K. Transtrum, Benjamin B. Machta, and James P. Sethna. "Geometry of nonlinear least squares with applications to sloppy models and optimization". In: *Phys. Rev. E* 83 (3 Mar. 2011), p. 036701. DOI: 10.1103/PhysRevE.83.036701. URL: https://link.aps.org/doi/10.1103/PhysRevE.83.036701.

[39] Alexander Farnsworth (Wärtsila). *Look, Ma, No Hands! Auto-docking ferry successfully tested in Norway.* 2018. URL: https://www.wartsila.com/twentyfour7/innovation/look-ma-no-hands-auto-docking-ferry-successfully-tested-in-norway. (Accessed: 26.10.2018).

[40] David Watson and David Scheidt. "Autonomous Systems". eng. In: *Johns Hopkins APL Technical Digest* 26.4 (2005), pp. 368–376. ISSN: 0270-5214. URL: http://search.proquest.com/docview/29722879/.

[41] G. -. Wei and S. D. Ma. "A complete two-plane camera calibration method and experimental comparisons". In: *1993 (4th) International Conference on Computer Vision*. May 1993, pp. 439–446. DOI: 10.1109/ICCV.1993.378183.

[42] J. Weng, P. Cohen, and M. Herniou. "Camera calibration with distortion models and accuracy evaluation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.10 (Oct. 1992), pp. 965–980. ISSN: 0162-8828. DOI: 10.1109/34.159901.

[43] Christian Wolff. *Radar Line of Sight.* URL: https://www.geospatialworld.net/blogs/how-gnss-works/. (Accessed: 09.12.2018).

[44] *ZBar bar code reader.* 2011. URL: http://zbar.sourceforge.net/. (Accessed: 15.12.2018).

[45] Z. Zhang. "A flexible new technique for camera calibration". eng. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 22.11 (2000), pp. 1330–1334. ISSN: 0162-8828.