2021•2022

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektromechanica

Masterthesis

Sensitivity analysis of camera localisation accuracy with respect to intrinsic parameters and environment model in simulation

PROMOTOR :

Prof. dr. ir. Eric DEMEESTER

PROMOTOR :

Dhr. Sergio PORTOLES DIEZ

Arno Molenaers, Pieter Vleeschouwers

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektromechanica

Gezamenlijke opleiding UHasselt en KU Leuven

UHASSELT

KU LEUVEN

UHASSELT

KU LEUVEN

2021•2022

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektromechanica

Masterthesis

Sensitivity analysis of camera localisation accuracy with respect to intrinsic parameters and environment model in simulation

PROMOTOR :

Prof. dr. ir. Eric DEMEESTER

PROMOTOR :

Dhr. Sergio PORTOLES DIEZ

Arno Molenaers, Pieter Vleeschouwers

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektromechanica

UHASSELT KU LEUVEN

Preface

This master thesis is the final puzzle piece of our master's degree, for Arno in Industrial Engineering sciences of Electronics and ICT Engineering, and for Pieter in Industrial Engineering sciences of Electromechanical Engineering and Technology completed at the University of Hasselt in combination with KULeuven. Our Master's thesis involved a sensitivity analysis for environmental and intrinsic camera parameters, and their influence on camera localization accuracy. We chose this topic because of the interest in the field of computer vision.

The performing of this master's study was an eventful, but educational process programming wise, as well as learning about the various factors that are important when performing localization using camera's. We could not have achieved this end result without the opportunity that was given to us by Intermodalics, That is why we would like to thank Dominick Vanthienen PhD and Sergio Portoles Diez PhD for the mentoring of the thesis. We would also like to thank our internal promoter prof. dr. ir. Eric Demeester for his advice and feedback.

In addition we would like to thank our parents for their endless support and help during the entire studies, as well as enabling the pursuit of this study. Not to forget we would like to thank our friends for the support and the friendship given in the past years.

Finally, we hope that in our further careers, we will still come into touch with computer vision and that we can use the experience and knowledge we have gained here.

Molenaers Arno and Vleeschouwers Pieter
June 2022

Contents

Preface **1**

List of tables **5**

List of figures **7**

List of explanatory terms **9**

Abstract **11**

Abstract in Dutch **13**

1 Introduction **15**
1.1 Context . . . 15
1.2 Problem statement . . . 16
1.3 Objectives and research questions . . . 16
1.4 Method . . . 18
1.5 Overview . . . 19

2 Literature study **21**
2.1 Theoretical background . . . 21
2.1.1 Camera pose estimation . . . 21
2.1.2 Perspective-n-Point . . . 22
2.1.3 Rotation conventions . . . 23
2.1.4 Camera model . . . 25
2.1.5 Thresholding . . . 27
2.1.6 AR-marker detection and implementation . . . 29
2.2 Related work . . . 31
2.2.1 Camera position estimation for UAVs using solvePnP with Kalman Filter . 31
2.2.2 Vision-based Docking of a Mobile Robot . . . 31
2.2.3 Hand-eye calibration using a single image . . . 31
2.2.4 Monocular Visual Odometry with Cyclic Estimation . . . 32
2.2.5 A Convolutional Network for 6-DOF Camera Relocalization . . . 32
2.2.6 Error Analysis and Optimization of Camera Calibration . . . 33
2.3 Conclusion . . . 33

3 Simulation frameworks **35**
3.1 Python simulation . . . 36

3.1.1 The simulation model . . . 36
3.1.2 Camera calibration . . . 38
3.1.3 Camera image acquisition . . . 38
3.1.4 Perspective-n-Point pose estimation . . . 44
3.1.5 Accuracy of the simulator . . . 45
3.1.6 Disturbance modelling . . . 46
3.2 Gazebo simulation . . . 47
3.2.1 The simulation environment . . . 47
3.2.2 Camera calibration . . . 48
3.2.3 ArUco-marker detector . . . 49
3.2.4 Transformation . . . 51
3.2.5 Pipeline of the Gazebo simulation . . . 52

4 Sensitivity analysis **55**
4.1 Python simulation results . . . 55
4.1.1 Camera parameters . . . 56
4.1.2 Specified camera image formation . . . 56
4.1.3 Camera pose estimation . . . 56
4.1.4 Error range determination . . . 58
4.1.5 Intel RealSense Tracking Camera T265 Results . . . 58
4.1.6 Intel Depth Camera D435 results . . . 61
4.2 Gazebo simulation results . . . 63
4.2.1 Simulation setup . . . 63
4.2.2 Intel RealSense Tracking Camera T265 Results . . . 63
4.3 Comparison between environments . . . 67

5 Conclusion **69**

Bibliography **73**

List of Tables

3.1 Pinhole camera parameters 38
3.2 Lens camera parameters 38
3.3 Perspective-n-Point pose estimation parameters 44
3.4 Camera test parameters 45
3.5 Camera error analysis parameters 46
3.6 Gazebo simulation details 47

4.1 Camera intrinsic properties 56

List of Figures

2.1 Visualization of the PnP problem . . . 21
2.2 Rodrigues-formula . . . 24
2.3 Principle of a pinhole camera . . . 25
2.4 Image acquisition visualization . . . 26
2.5 Cavity Array . . . 27
2.6 Light Cavities . . . 27
2.7 Color Filter Array . . . 27
2.8 Photosites with Color Filters . . . 27
2.9 Overview thresholding . . . 28
2.10 Cyclic BA algorithm . . . 32

3.1 Overview of simulator use-case . . . 35
3.2 Design of the virtual test environment . . . 36
3.3 Classes in the virtual test environment . . . 36
3.4 Processing pipeline of the simulation . . . 37
3.5 Field of view visualisation in the simulator . . . 39
3.6 Pinhole camera image plane with projections . . . 40
3.7 Lens camera image plane with projections . . . 41
3.8 Projection grid coordinate system . . . 42
3.9 NDC coordinate system . . . 43
3.10 Pixel grid coordinate system . . . 43
3.11 Visualization of the PnP problem in the simulation . . . 44
3.12 Gaussian noise probability distribution function . . . 46
3.13 Gazebo inputs and attributes . . . 48
3.14 Overview ArUco-marker detector . . . 49
3.15 Thresholded image . . . 49
3.16 Virtual camera image with detected marker . . . 50
3.17 Visualization of rotations in camera coordinate system . . . 51
3.18 Pipeline for initializing the Gazebo simulation . . . 52
3.19 Overview of localization calculations in the simulation . . . 53

4.1 Scene of the room with the camera and the markers . . . 55
4.2 Processing pipeline of the error analysis simulation . . . 57
4.3 Error on camera localization with deviation on camera matrix range test . . . 58
4.4 Error on camera localization for T265 with camera matrix error . . . 59
4.5 Error on camera localization for T265 with distortion coefficient error . . . 60
4.6 Error on camera localization for D435 with camera matrix error . . . 61

4.7 Error on camera localization for D435 with distortion coefficient error 62
4.8 Error on localization for marker size 0.09 m and 0.18 m 63
4.9 Error on localization for marker size 0.36 m and 0.45 m 64
4.10 Error on localization with camera matrix error 65
4.11 Error on localization with distortion coefficient error 66

List of explanatory terms

Pose		The combination of an object's location and orientation is referred to as its pose.
Perspective-n-Point	PnP	Perspective-n-Point is known as the challenge of predicting the pose of a calibrated camera given a collection of n 3D points in the environment and their associated 2D projections in the picture.
Field of view	FOV	The field of vision (FoV) is the area of the visible world that is seen at any determined moment.
Visual Simultaneous Localization and Mapping	VSLAM	Refers to the operation of locating and orienting a sensor relative to its environment while simultaneously mapping the environment around that sensor.
Degrees of freedom	DOFs	The number of dimensions that defines the pose of an object within the given coordinate system.
Robot operating system	ROS	Robot operating system is a set of software libraries and tools that can be used to build robot applications
ArUco-markers		Library of square shaped markers with a binary ID
Intrinsic camera matrix	K	[3x3]-matrix that represents the camera model
Exentric camera matrix	E	[3x4]-matrix that defines the pose and orientation of the camera
Array of distortion coefficients	D	Array that represents the distortion model of the camera, parameters K1 - K6 and P1 - P2
Translation vector	tvec	Vector containing the translations
Rotation vector	rvec	The most compact representation of a rotation matrix

Abstract

Visual pose estimation is widely researched within computer vision, with several application areas. A wide variety of cameras can be used to estimate the camera pose in a room, adapted to the environment. In this research on behalf of Intermodalics located in Leuven, two simulators are designed to investigate the effect of the camera intrinsics and environment on the localization. One simulator was developed in Python, in this simulator the calculation of the camera pose with the PnP-problem is solved using the extracted image points in the camera's field of view, with corresponding known 3D world points. These reference points in combination with the intrinsic camera parameters are used to calculate the camera pose. The second simulation was developed using ROS, a camera pose estimate is determined by means of a virtual camera and markers. Some world camera aspects, such as geometry distortion, depth of field effect, and imperfections caused by image sensor characteristics were considered. For the pose determination, the pose of markers is estimated in a marker detector and transformed to determine the camera location. An experimental campaign was carried out to retrieve accurate results on the localization error by using the simulators. These results showed that change on the camera parameters affects multiple rotations and/or translations in pose determination. If there is an unprecedented error on the camera calibration, it is complex to identify the exact camera parameter to which this is attributable.

Abstract in Dutch

Visuele posebepaling is breed onderzocht binnen computervisie met verscheidene toepassingsgebieden. Vele verschillende soorten camera's kunnen gebruikt worden om de pose in een ruimte te schatten. In dit onderzoek, zijn twee simulatoren ontworpen om het effect van de cameraparameters en de omgeving op de lokalisatie te onderzoeken. Eén simulator werd ontwikkeld in Python, waarbij de berekening van de camerapose met het PnP-probleem wordt opgelost, door gebruik te maken van de beeldpunten in het zichtsveld van de camera, met corresponderende gekende 3D-wereldpunten. Deze referentiepunten in combinatie met de intrinsieke cameraparameters worden gebruikt om de camerapose te berekenen. De tweede simulatie is ontwikkeld met behulp van ROS, een camerapose schatting wordt bepaald door middel van een virtuele camera en markers met gekende grootte. Enkele wereldcameraaspecten, zoals geometrievervorming, scherptediepte-effect, e n o nvolkomenheden veroorzaakt door beeldsensor-karakteristieken worden in rekening gebracht. Voor de bepaling van de 6D-pose wordt de pose van de markers geschat in een markerdetector en getransformeerd om de cameralocatie te bepalen. Experimentele simulaties zijn uitgevoerd om nauwkeurige resultaten te verkrijgen over de lokalisatiefout. Uit deze resultaten bleek dat verandering op de cameraparameters invloed heeft op meerdere rotaties en/of translaties bij de posebepaling. Indien er een ongekende fout op de camera kalibratie zit, is het complex om te achterhalen aan welke cameraparameter dit juist ligt.

Chapter 1

Introduction

1.1 Context

Within the framework of this master thesis, research is conducted on the implementation of markers in industrial applications. In the modern industrial society robotics has a growing field of application. By adopting robotics, a variety of tasks can be executed efficiently and precisely. To improve the technology, research is conducted on various aspects. The hardware and kinematics are well understood but still have a range of improvement in certain applications. However, the software on the other hand still has much margin for improvement. The robots transform from commonly used robots that follow a predefined path, to robots that visualize their surroundings and make decisions based on deducted information. SLAM-software in industrial applications can dynamically map the immediate environment of autonomous mobile robots. The created map can then be used to navigate the environment.

This research will be conducted within the company Intermodalics located in Leuven. The company specializes in robotics and Visual Simultaneous Localization and Mapping (SLAM)-software. Visual SLAM tries to do the localization and mapping based on camera images and the features recognized in these images. With further extensions, Intermodalics also integrates inertial information in the SLAM solution.

In this thesis, a study is performed to test the effects of calibration errors on the camera localization, as well as the influence of marker size and distance to the camera. To perform a localization features need to be detected in the camera image. These features can be key-point features deducted from recognized corners or marks in the image, or can be specifically placed AR markers [1]. AR markers are often known as fiducial markers. In position estimation, fiducial markers often have a fixed size and orientation with a binary code for identification. This way they can act as a fixed reference for camera pose estimation. A common example of fiducial markers are ArUco-markers. The analysis is carried out by building simulators that are able to simulate real camera's, using a localization algorithm. The localization algorithm in combination with a mapping algorithm would a form a SLAM system. For the simulation only the localization algorithm is necessary because the map is predefined and can be directly used. For one simulator predefined points will be used which resembles a true corner feature detector, but with predefined points so the mapping step can be ignored. The other simulator uses AR markers, that can be detected using marker finders. The marker finders define the id, pose of the marker, as well as its

four corner points. The marker corners are used as input points with known world coordinates, so the localization can be performed using a predefined map. The SLAM-software tested in this simulator will be used for camera pose estimation within the space, localizing objects or creating a map of the surrounding area.

1.2 Problem statement

The problem when using visual SLAM systems, is the wide variety of camera's that can be used. All of these camera's have different intrinsic parameters and distortion models depending on the lens and sensor. When testing the built SLAM-system it would be useful to test its accuracy using multiple different camera's and find the camera's where the algorithms are optimized.

When dealing with real world variability, it is difficult to determine which factors cause the deviations on the camera localization accuracy. Often various factors are out of control of the researcher working on the analysis, which increases the difficulty of creating repeatable results, and determining which factors improve the localization. When working in a simulated environment, every variation can be controlled exactly in combination with the influence on the localization accuracy.

1.3 Objectives and research questions

The objective is to be able to study the accuracy of the camera pose determination when adjusting the camera parameters. To have control over these parameters, there is a need for a simulator where a pose estimation of a camera is performed using markers. For this thesis research, a simulator is created in the Python and ROS environment to perform a sensitivity analysis on the camera parameters. This will be divided in two separate objectives where the focus will be on different aspects of the problem. The Python simulation will be developed to do a more theoretical analysis whereas the ROS environment uses a more realistic approach.

In the Python simulation, the idea is to perform a theoretical approach to camera pose estimation. Having as much control as possible over what happens during this process can be a great advantage for obtaining information such as the effects of the camera parameters on the different translations and rotations of the camera. In order to perform a pose estimation, a room needs to be simulated in which markers and a camera are placed. The camera must contain several characteristics to model projections on the image plane: a focus length, a resolution, a distortion model to emulate a lens, and a scaling factor. The camera requires coordinates and rotations to be positioned in a space. Similarly, markers need coordinates to be placed in a space and to be projected onto the camera's image plane. For the projections from the camera, a method must be provided that also takes into account the resolution of the camera. With these projections, a calculation should then be able to figure out the pose of the camera using only the intrinsic camera properties and the projections of the camera. The simulator should then be used to investigate the research question of what the effect is on the determination of the pose when errors are added to the process.

With these properties, any camera from the real world can then be simulated. To position the camera in a space, the camera will need coordinates and rotations. The markers will also need coordinates to represent them in a space and to project them onto the camera's image plane. For the camera projections, a method must be provided that also takes into account the resolution of the camera. With these projections, a calculation should then be able to retrieve the pose of the camera using only the intrinsic camera properties and the projections of the camera. Once this works, an important research question can be investigated, namely what the effect is on the pose estimation when camera parameters are adjusted.

For a more realistic approach when estimating the camera pose, there should be a simulator that also takes into account everything involved in a real camera pose estimation. This includes things like light incidence and different types of noise. The necessary objectives to achieve are:

- Using a simulation environment with a realistic approach
- Build a world with a virtual camera and various markers in any pose
- Markers should be automatically adjustable for marker detection and localization
- Camera has to represent real world camera's with their intrinsics and distortion model
- Accurately detect the markers in the image, and deduce their pose in the camera coordinate system as precisely as possible
- Implementation a calibration error, with a small deviation in the camera intrinsics or distortion coefficients.
- Calculate the camera pose using the transformation matrix that is generated from the detected markers.
- Deviations on the detected marker poses, as well as the true marker poses to analyse the error on the localization it causes.
- Estimation of the accuracy based on distance from the camera to improve the localization accuracy.

For both simulations a series of tests will be performed to verify the precision and repeatability of the pose estimation. The results of the error analysis of both simulators will be shown in graphs and discussed. The difference between the two simulators will also be compared. Finally, from these results, research questions will be answered.

Associated research questions are:

- Does a theoretic approach help in figuring out the effects of error implementations?
- Which errors cause the largest errors in pose estimation?
- Do all errors affect pose estimation?
- Can errors in real cameras be reduced with the acquired knowledge?

1.4 Method

The research started with a thorough search of useful sources. This research yielded a theoretical background, as well as various similar implementations that have already been performed. Various functions that have been found can be used or implemented.

The simulator was built in Jupyter [1], an open-source tool that makes it possible to easily merge Markdown text and executable Python source code on a single canvas known as a notebook. In order to create the simulator, research was done on similar Python projects involving the modeling of cameras. First, the room was visualized where the axes of the plot represent the dimensions of the room. The markers can be represented as simple points in the plot with their specific coordinates. Now, to model a camera that makes a camera projection of the markers, a pinhole model camera was first created. The pinhole camera is a simplified version of a lens camera. For this camera, important features are visualized to clearly understand what is happening. The image plane was placed at the focal length of the camera, the lines of the FOV were drawn and a line from the markers to the camera was drawn. The lines from the markers to the camera were important to establish whether the markers are in the FOV of the camera and consequently should be projected onto the projection plane. After forming the projections of the camera, the pose of the camera can be determined. For this, the correct conventions had to be found using a function from the OpenCV [2] library. Once a correct pose was found for the pinhole camera, this camera was extended to a lens camera where distortions are included. The lens camera and the pose estimation from its produced images are then used to create an automated simulation. In this simulation a chosen number of error points are implemented on the camera properties, and a pose is determined. These results are plotted on a graph to study the effect of the implemented errors.

At the beginning of the implementation of a simulation in ROS a clear foundation for the interaction of the different modules had to be established. This was necessary to create the virtual world in Gazebo, and the various models that exist within the simulation. To insert the models, various launch files where used. The Gazebo simulation initiates various services for every model, one of which is used to dynamically allocate the model in the simulation. Gazebo also provides plugins for various sensors, including a virtual camera which is used to emulate a real camera with its intrinsic and distortion parameters. This virtual camera is then placed in the simulation world and starts publishing the camera image, alongside the camera information. Following this step an ArUco-detector built for the drone application clover[3] is used. The ArUco-detector subscribes to the image provided by the virtual camera to detect and estimate the pose of the markers. The detector also uses the camera information to transform the two DOFs (u,v)-coordinates of the marker to a six DOFs pose in the camera coordinate system. To launch the world with its models a Python script is used that initiates the various nodes with the set parameters configured in the yaml-file. A second Python script is then used to retrieve the information provided by the simulation world to perform the localization of the camera, as well as relocate the various models when iterating over the various parameters of the simulation. The simulation results are then combined to a CSV-file for further analysis.

The final step is to compare the two simulators, and gain insights on the localization accuracy that can be achieved using various camera models in combination with different marker sizes and detection distances.

1.5 Overview

The chapter literature study consists of two main parts. The first part gives a theoretical background of the various aspects that are used in the transformation from 3D-image to the camera visual plane and calculations of pose estimation. The second part dives deeper in other related implementations of Visual based positioning. The key difference is that our implementation uses a virtual environment for testing, that will be validated in reality, while other research is directly tested in reality which makes the iterative process of changing parameters slower.

In Chapter 3: Simulation frameworks, the design of the two simulators will be elucidated. One version is built from scratch and coded in Python, while the other is built in ROS using a Gazebo virtual environment in combination with available packages.

In the fourth chapter a sensitivity analysis will be conducted. In this analysis the influence of various factors can be analyzed. One of the factors will be to implement a calibration error .

Chapter 2

Literature study

2.1 Theoretical background

2.1.1 Camera pose estimation

For this study, the camera pose estimation problem is explored. For this problem, the relative position and orientation of a camera is estimated using known landmarks. Several approaches to extrinsic camera parameter estimation (i.e. relative translation and rotation) have been presented during the last decades. Some relevant study's are presented in Section 2.2.

In the pose estimation problem, the goal is to minimize the reprojection error of 3D-2D point correspondences. A typical technique for pose estimation is Perspective-n-Point (PnP), this method is further discussed in Section 2.1.2 and an illustration is shown in Figure 2.1.

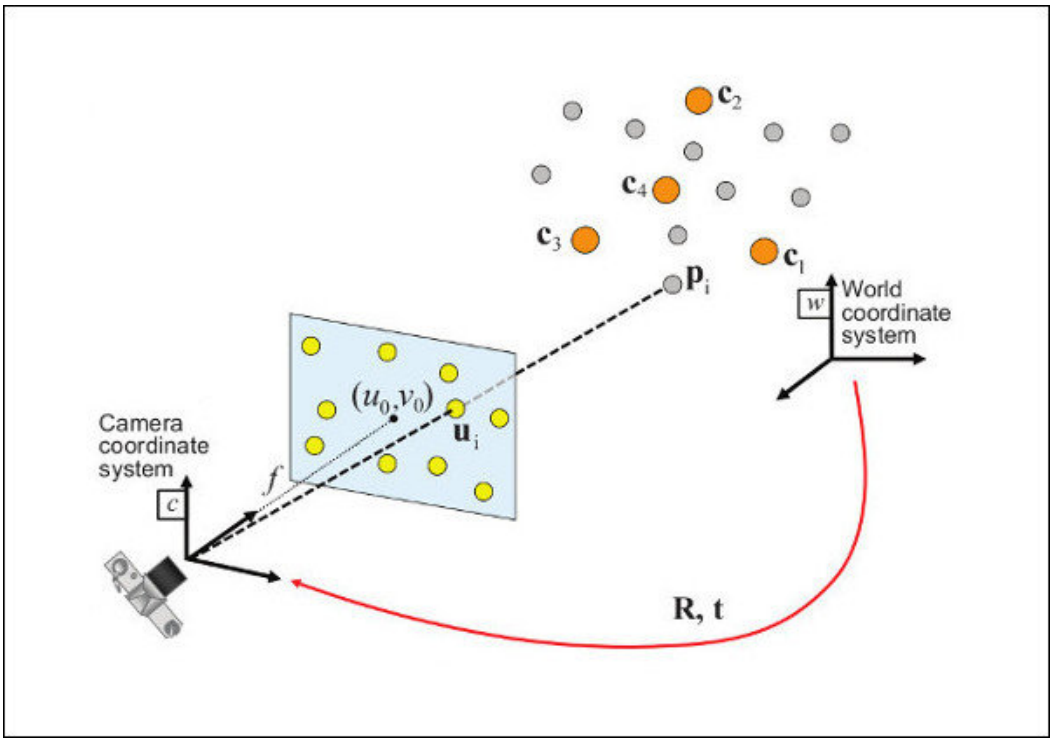


Figure 2.1: Visualization of the PnP problem [4]

2.1.2 Perspective-n-Point

Perspective-n-Point problem in camera pose estimation

In the beginning of the 1980s, the Perspective-n-Point (PnP) problem was presented by Fischler [5]. Up to now, PnP has been a fundamental problem in computer vision applications to determine the pose of a calibrated camera using n number of known 3D points from the 3D model and their corresponding 2D projections in the image. The camera posture consists of 6 degrees of freedom (DOF), which includes 3D rotation (roll, pitch, and yaw) as well as 3D translation of the camera relative to the environment. To tackle a PnP problem, information must be obtained from at least three pairs of related points. The two most common methods according to Fischler [5] to solve the PnP problem are Perspective-3-Point (P3P) and Efficient Perspective-n-Point camera pose estimation (EPnP).

The perspective projection model for cameras, in a PnP application is defined in the following formula:

$$s * P_i = K \begin{bmatrix} R & T \end{bmatrix} P_w \tag{2.1}$$

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.2}$$

where

P_w	homogeneous world point	$\begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$	(2.3)
P_i	homogeneous image point	$\begin{bmatrix} u & v & 1 \end{bmatrix}^T$	(2.4)
K	intrinsic camera parameters	$\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$	(2.5)
s	scale vector for image point		
R	3D rotation		
T	3D translation		

1. Perspective-3-Point

   If the number of points $n = 3$, the PnP problem is of minimal form. P3P will result in four real, geometrically feasible solutions for R and T. In the article [6] published by Gao, et al. a recent technique was developed for tackling the this problem.

2. Efficient Perspective-n-Point

   Lepetit, et al. introduced Efficient PnP (EPnP) in their paper [7], which solves the general PnP problem for $n \geq 4$. The alogrithm works by expressing each of the n points as a weighted sum of four virtual control points. The problem's unknowns are the coordinates of these control points. These control points are subsequently used to solve the camera's posture by rotating and translating these 3D points in their eigenspace, then solving the least squares formulation without regard to orthonormality. Despite the fact that EPnP

can yield excellent results, it still depends on unconstrained least squares estimation, where noise in the data can affect the result. It can also get stuck in local minima, especially when a small number of points are given.

3. Consistently Fast and Globally Optimal Solution to the Perspective-n-Point problem

   SQPnP, a fast and globally convergent non-polynomial PnP solver, was presented in the study [8] by Terzakis and Lourakis. SQPnP solves the PnP issue by conducting local searches in the neighborhood of particular feasible spots from which the global minima may be found in a few steps. SQPnP allows for a straightforward implementation that only requires standard linear algebra operations and is computationally inexpensive. Comparative investigations demonstrate that SQPnP performs competitively to state-of-the-art PnP solvers and reliably recovers the correct camera posture independent of the noise and the spatial arrangement of input data.

4. Random Sample Consensus

   If there are outliers in the collection of the corresponding points, PnP is prone to mistakes. To counteract this, Random Sample Consensus (RANSAC) as proposed by Fischler and Bolles [9] can be used in combination with other solutions to make the final camera pose solution more resistant to outliers. The algorithm is designed to work as follows, it attempts to obtain an optimal solution by estimating the underlying model parameters using the smallest possible number of data points.

2.1.3 Rotation conventions

Rotation matrices

The following three fundamental rotation matrices rotate vectors in three dimensions at an angle to the x-, y-, or z-axis, using the right-hand rule to determine the alternates.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix} \tag{2.6}$$

$$R_y(\theta) = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \tag{2.7}$$

$$R_z(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

From these basic rotation matrices eqs. (2.6) to (2.8), any possible rotation can be elaborated. The product is a representation of an extrinsic rotation having Euler angles: yaw (α), pitch (β) and roll (γ), about the axes x, y, z. The rotation product is presented in equation (2.9).

$$R = R_z(\alpha)R_y(\beta)R_x(\gamma)$$

$$= \begin{bmatrix} cos(\alpha) & -sin(\alpha) & 0 \\ sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos(\beta) & 0 & sin(\beta) \\ 0 & 1 & 0 \\ -sin(\beta) & 0 & cos(\beta) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\gamma) & -sin(\gamma) \\ 0 & sin(\gamma) & cos(\gamma) \end{bmatrix} \tag{2.9}$$

$$= \begin{bmatrix} cos(\alpha)cos(\beta) & cos(\alpha)sin(\beta)sin(\gamma) - sin(\alpha)cos(\gamma) & cos(\alpha)sin(\beta)cos(\gamma) + sin(\alpha)sin(\gamma) \\ cos(\alpha)sin(\beta) & sin(\alpha)sin(\beta)sin(\gamma) + cos(\alpha)cos(\gamma) & cos(\alpha)sin(\beta)sin(\gamma) - cos(\alpha)sin(\gamma) \\ -sin(\beta) & sin(\beta)cos(\gamma) & cos(\beta)cos(\gamma) \end{bmatrix}$$

Rotation vector and translation vector

In the geometry, a rotation in three dimensions can be expressed as a mathematical transformation using a variety of conventions. A rotation vector is the most compact representation of a rotation matrix (equation 2.10). analogously, for the translations there exists a translation vector called tvec (equation 2.11).

$$R_{vec} = \begin{bmatrix} R_X & R_Y & R_Z \end{bmatrix} \tag{2.10}$$

$$T_{vec} = \begin{bmatrix} T_X & T_Y & T_Z \end{bmatrix} \tag{2.11}$$

Rodrigues' rotation formula

Rodrigues' rotation formula provides an efficient technique to calculate the rotation matrix R. This rotation matrix corresponds to a rotation by an angle θ along a fixed axis described by the unit vector $k = (k_x, k_y, k_z)$ in R^3. An illustration is given in Figure 2.2. Where k is a unit vector defining the rotation axis, and v is a vector rotating arbitrarily about k over an angle θ. Using Rodrigues' rotation formula, vector v can be composed in its components parallel and perpendicular to k, and only the perpendicular component can be rotated. [10]

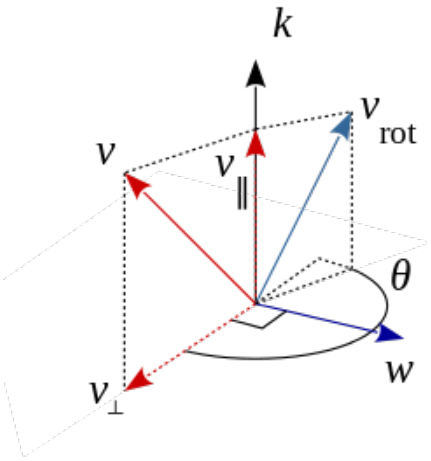


Figure 2.2: Rodrigues-formula [10]

2.1.4 Camera model

Pinhole camera model

The pinhole camera model is the simplest type of camera that can be realized in the real world. The principle of a pinhole camera is presented in Figure 2.3. This model represents the mathematical relationship between the coordinates of a point in three-dimensional space and its projection onto the picture plane, where the camera aperture is specified as a point and no lenses are required to concentrate light. For this model, since there is no lens, this gives a result with no geometric distortions. Another outcome is that no blurred objects are created due to finite apertures.

The pinhole camera model can be represented as equation (2.12):

$$X_p = DK_0EX_w = \begin{bmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} X_w \tag{2.12}$$

where

f_x, f_y	camera focal lenghts
c_x, c_y	image principal points
K	intrinsic camera parameters
E	camera extrinsic parameters
X_w	world point
X_p	projection point

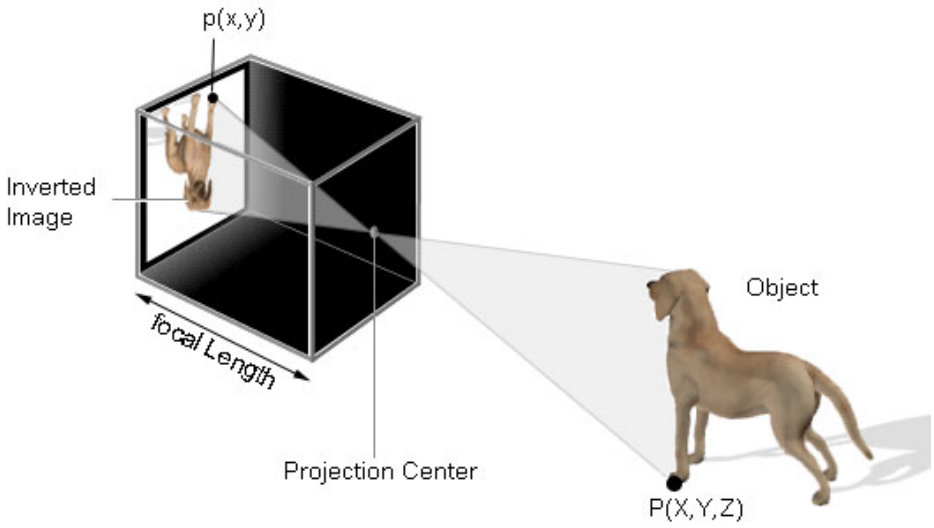


Figure 2.3: Principle of a pinhole camera [11]

Source: Adapted from [12], Figure 2.2

Image acquisition

Extracting an image from a camera is also referred to as image acquisition. In this process, light waves reflected from an object are converted into an electrical signal by a network of sensors sensitive to light energy. To create a digital image with these signals, a conversion from the continuous analog signals to a digital format needs to be done. Figure 2.4(a) shows a continuous image that is being converted to its digital form. This conversion requires two steps: sampling and quantization. Sampling is the process of transforming the electrical signal in the camera's sensors into a numerical sequence. For example the signal AB shown in Figure 2.4(b), is sampled by taking equally spaced samples, which is represented in Figure 2.4(c). After this, the intensity values are discretized to the corresponding bit resolution, this is also termed quantization. The digital resulting samples are shown in 2.4(d) as white squares.

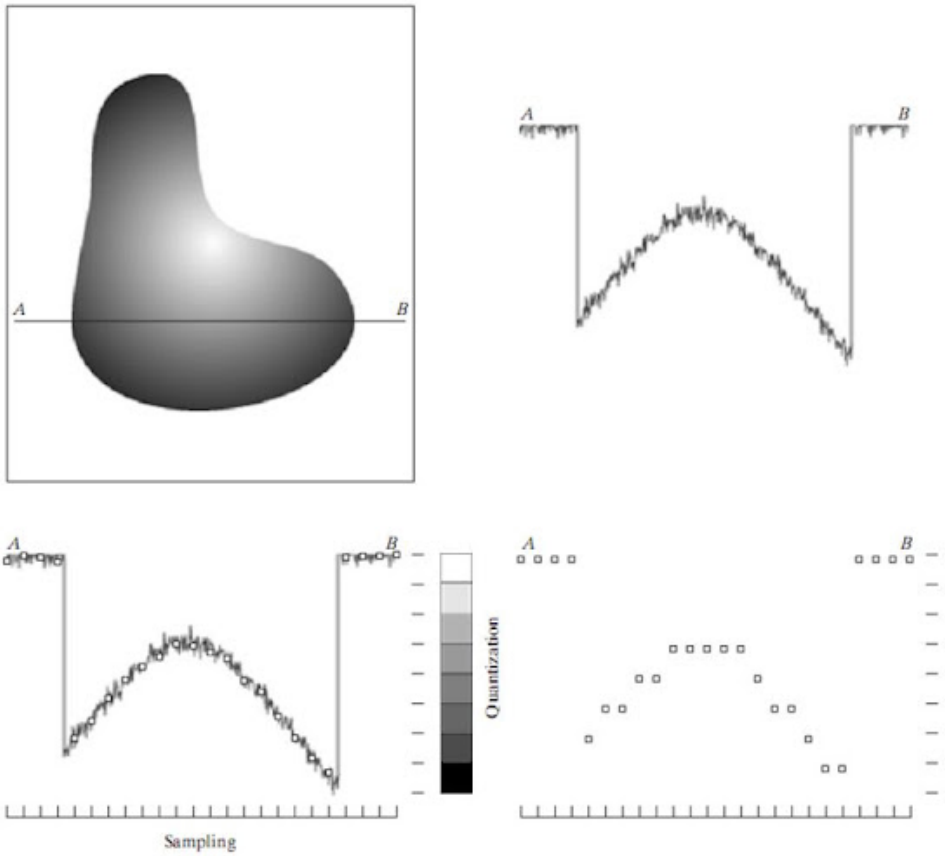


Figure 2.4: Image acquisition: (a) Continuous image. (b) Intensity variations along line AB. (c) Sampling and quantization. (d) Digital scan line [13]

Digital camera sensors

To capture a picture, a digital camera utilizes an array of millions of tiny light apertures or photosites shown in Figure 2.5. The moment an image is captured each of them is exposed to catch photons and store them as an electrical signal. When the exposure is complete, the camera covers each of these photosites and attempts to determine how many photons illustrated in Figure 2.6 went into each cavity by detecting the intensity of the electrical signal. The signals are subsequently quantized as digital values with precision dictated by bit depth.

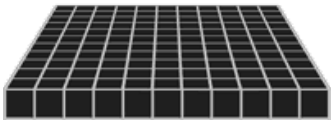


Figure 2.5: Cavity Array [14]

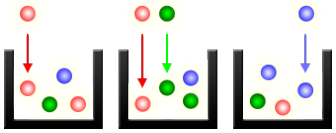


Figure 2.6: Light Cavities [14]

However, since these sensors do not determine how much of each color they contain, the above image would produce only grayscale images. To create color images, a filter must be placed over each cavity that allows only specific shades of light to pass through like shown in Figures 2.7 and 2.8. Almost all modern digital cameras can only capture one of the three primary shades in each chamber, so about two-thirds of the incoming light is ignored. As a result, to have full color at each pixel, the camera must approximate the other two primary colors.

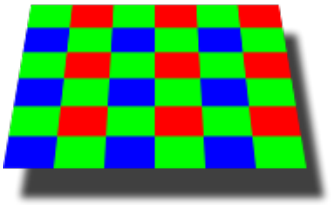


Figure 2.7: Color Filter Array [14]

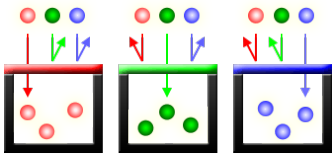
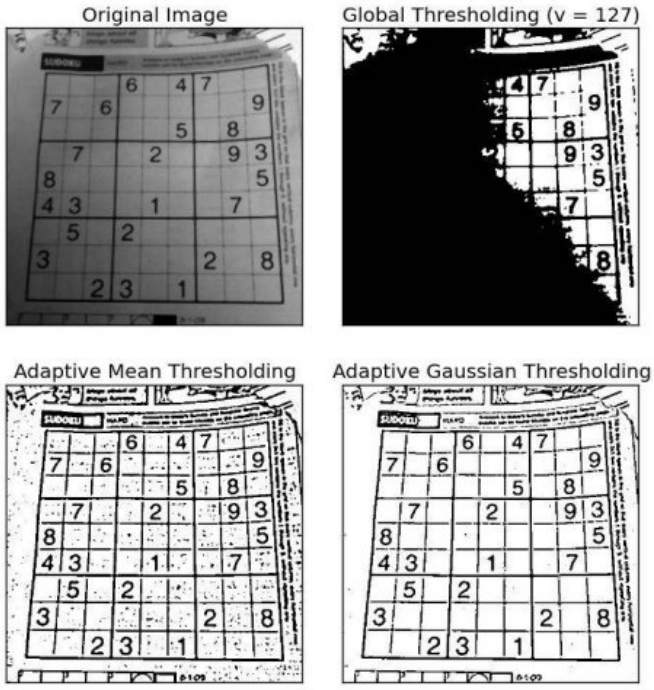


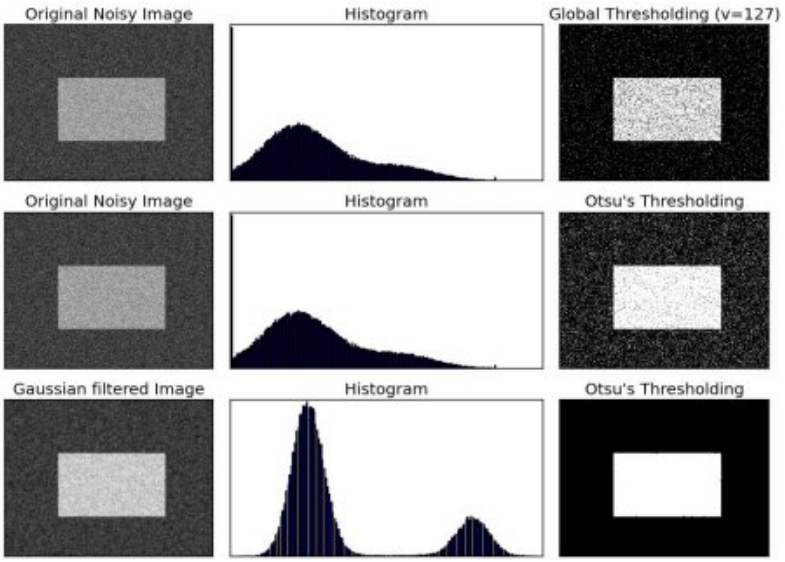
Figure 2.8: Photosites with Color Filters [14]

2.1.5 Thresholding

Binary images form an important aspect in computer vision, as it is fairly simple to detect edges from light to dark spots or reverse in an image. Thresholding can help in this aspect as it can perform image binarization on an RGB or monochrome image using a specified threshold value. Any pixel above the threshold value will return white (foreground) and when lower than the threshold it will return black (background), separating light and dark areas, and creating a more simple and effective image. The important aspect in thresholding is how the threshold value is determined. An overview of global, adaptive and Otsu thresholding is given in Figure 2.9.



(a)



(b)

Figure 2.9: Overview thresholding: (a) Global vs. adaptive method, (b) Otsu's method [15]

For this review Otsu's method will be examined, as well as an adaptive thresholding method. Otsu's method determines an image threshold based on the histogram of the total image, whereas the adaptive thresholding algorithm determines a pixel based threshold by using a small area around it [15]. Otsu's binarization method is based on a classification approach which searches for the threshold to minimize the intra-class variance [16]. An extension of Otsu is also implemented, where it uses the maximum between class variance as the ideal threshold value. Otsu's method improves by filtering the image with a Gaussian kernel to cancel out some noise. The difficulty for the adaptive thresholding method is to determine an appropriate size of window or subimage in combination with the local threshold depending on the variation in that specific window[17].

2.1.6 AR-marker detection and implementation

Marker type

There exist various types of markers, each with their own way of detecting them. The general method comes down to a corner or line detector to verify the position and orientation, but these are chosen to work better in specific lighting conditions and circumstances like occlusion or rotation.

1. Monochrome square planar markers

   The first and most common type of marker are square shaped, various libraries are developed to create and detect these. The markers are detected by using a global threshold of which the outline can be fitted by four line segments. Some of these markers use binary codes using black and white variations to represent an ID. A first library is ARToolkit [18], this was originally created for visual AR-conferencing [19]. The square shape with an identifier in the middle became the base for other implementations. ARToolkit allowed any shape to be placed as the identifier, which made it easily distinguishable for the users but decreases robustness. Unlike VisualCode or binARyID where the center is divided in a number of smaller squares to represent bits, and create patterns that are more robust to rotation and lighting conditions. ARTag then introduced a gradient-based method to detect the edges of the marker while keeping a similar marker. This change led to increased tag detection and detection under partial occlusion. AprilTag [20] continued this path by analysing gradient patterns and implementing a quad extraction method. In this manner non-intersecting lines can be estimated to complete the marker when it is partially occluded. ArUco markers try to reduce processing power by not including pre-created markers, but only including markers of your choice and thus having a smaller library. Another advantage of ArUco markers is their robustness to motion blur at the cost of being more computationally expensive than AprilTag. A final library called STag utilizes a square border with a circular pattern in the center. By using this type of marker the homography can be refined by using elliptical fitting. This provides a better localization then markers only using quad detection.

2. Monochrome circular markers

   Larics markers are simple circular markers with a line outside of the circles to determine the orientation [21]. Inside the marker a binary ID is formed using radial lines. The double circle improves the marker detection, as a double elliptical fitting can be utilized. Concentric contrasting circle (CCC)-markers are the simplest kind of markers. It can be used as a reference-point or to identify the object that has a marker, but has no ID, so no extra information can be provided. These markers aren't used as much as they became deprecated. Instead, more recent examples with an ID are chosen. An example of circular markers with an ID are TRIP-markers, these encode a message using ternary numbers and feature a bullseye in the middle. These markers are robust in different lighting circumstances. A marker that is better fit for precision, thus utilized for object measurement or camera calibration is called RuneTag [22]. This marker consists of two or three rings consisting of dots, representing the ID. This marker is robust against occlusions, while being very precise to detect. A disadvantage of this marker is that it can't be detected rapidly, thus isn't fit for many robotics applications.

3. Multicolor markers

   In previous parts monochromatic markers have been discussed, but multi-color markers have also been tested in packages such as ChromaTag. These tags have the advantage of increased detectability and limits false positives at the cost of worse detection rates at longer distance or steep angles. A next multicolor variant FourierTag uses grayscale to construct a radially symmetric ID. This gives the advantage of longer range detection, and detection grade descents slowly unlike other types that are either detected or lost.

4. Light polarization markers

   A final marker type PolarTag [23] makes use of light polarization to detect invisible tags. These markers can encode up to forty bits of data while being robust to ambient light and occlusion at steep angles and longer distances. The downside of these markers is in the fact that they are harder to manufacture, while different types can be printed with a regular printer.

2.2 Related work

2.2.1 Camera position estimation for UAVs using solvePnP with Kalman Filter

Using solvePnP and the Kalman filter (KF), the research [24] presented by D. H. Lee et al. offers an approach for getting camera coordinates for unmanned aerial vehicles (UAVs). The data and motion of a circle gets accurately detected and monitored using the KF. The SolvePnP method then uses the detected circle as an image point and is being used to obtain the translation and rotation vectors from the world coordinate system's reference. The retrieved data can be used to calculate the location of UAVs, and the findings of the experiment show that the position can be calculated using just the camera. The suggested algorithm's strong performance is proved by experimental findings.

2.2.2 Vision-based Docking of a Mobile Robot

A recent study [25] by A. Kriegler and W. Ober discussed the problems of LiDARs in combination with docking stations used with robots for determining robot position. The docking serves for localization and navigation with the robots. But these systems incorrectly detect dynamic obstacles. Therefore, a vision-based framework is proposed in this paper. Here, the SolvePnP [4] algorithms from OpenCV are implemented to estimate the camera position using the detection points and intrinsic parameters. These experiments have shown that using SolvePnP gives systematically inaccurate pose estimates in the x-axis for their results. Therefore, the pose estimates cannot be used for docking the robot. The vulnerability of the SolvePnP algorithms is addressed and insights are given to circumvent similar problems in future applications. These insights may be helpful for creating a simulation.

2.2.3 Hand-eye calibration using a single image

R. A. Boby presented a paper [26] that proposes a hand-eye calibration with the Efficient Perspective-n-Point (EPnP) camera pose estimation method suitable for a camera of an industrial robot, using only a single image. A method is proposed to detect the object and determine its pose using a minimum number of images taken from a given camera position. In the idealistic case, this involves a single image. But if only a single edge is captured, an additional image is taken after the camera is aligned with the identified edge. The 3D information about the edges, gathered from the calibration data, was then used to calculate the position of the object so that it could be picked up by a robot.

Efficient Perspective-n-Point

To perform the camera position estimation, the calibration parameters, identified grid angles in the 2D space of the image, and the 3D coordinates of the grid are sent to the EPnP camera pose estimation method, which is implemented in the 'SolvePnP' function from the OpenCV library [2]. This calculation results in the camera's pose. This information can be obtained from a single input image of the calibration grid, as opposed to multiple pose information as required by traditional calibration methods.

2.2.4 Monocular Visual Odometry with Cyclic Estimation

A new system camera pose estimation system is proposed in the paper [27] by F. I. Pereira et al. The new system, known as Monocular visual odometry (MVO) estimates the camera position and orientation based on images taken by a single camera. A Cyclic Estimation is made from the three most recently taken images. Results from this system show that an average translation error of 1.29 percent and an average rotation precision of 0.0029 degrees per meter are achieved.

Cyclic Estimation with Perspective-n-Point

The algorithm illustrated in Figure 2.10 for cyclic camera pose estimation works by using the point correspondences in the last three images. Three corresponding features are used to determine the distance to the camera using two triangulations [28]. Following with a Perspective-n-Point calculation to estimate relative pose of the camera. This solvePnP method uses Levenberg-Marquardt [29] nonlinear optimization where RANSAC is implemented to estimate the camera displacement.

function CYC_BA($T_{k,k-1}, U_{k-1,k,k+1}$)
  num_cyc $\leftarrow$ 0,
  rep $\leftarrow$ max_rep
  threshold $\leftarrow$ init_threshold
  $T_{k-1,k} \leftarrow Invert(T_{k,k-1})$
  while rep>threshold & num_cyc<max_cyc **do**
    $d_{k-1} \leftarrow triangulate(T_{k-1,k}, U_k, U_{k-1})$
    $T_{k+1,k-1} \leftarrow SolvePnP(U_{k-1}, d_{k-1}, U_{k+1})$
    $d_{k+1} \leftarrow triangulate(T_{k+1,k-1}, U_{k-1}, U_{k+1})$
    $T_{k,k+1} \leftarrow SolvePnP(U_{k+1}, d_{k+1}, U_k)$
    $d_k \leftarrow triangulate(T_{k,k+1}, U_{k+1}, U_k)$
    $T_{k-1,k} \leftarrow SolvePnP(U_k, d_k, U_{k-1})$
    $rep \leftarrow GetReproj(U_{k-1}, U_k, d_k, T_{k-1,k})$
    num_cyc, threshold $\leftarrow$ num_cyc+1, threshold + δ
  end while
  return $Invert(T_{k,k+1})$
end function

Figure 2.10: Cyclic BA algorithm

Source: Adapted from [27], page 3, Figure 3

2.2.5 A Convolutional Network for 6-DOF Camera Relocalization

Some CNN designs have been offered as a solution for automatically estimating extrinsic features of the camera. Based on this network, the authors of this solution[30], proposed an approach for regressing the 6-DOF camera pose from RGB images recorded by a moving camera. This approach is resistant to changes in illumination, motion blur, and other intrinsic factors of the camera. The output is specified by a 3D vector representing the camera location and a quaternion representing the orientation. The Euclidean distance is used as a loss function, and a weighting factor is used to keep the position and orientation errors on the same scale.

2.2.6 Error Analysis and Optimization of Camera Calibration

The study [31] of Zhang D. et al. explored the relationship between the calibration conditions and the accuracy of the calibrated TV camera parameters. The ideal calibration conditions are proposed. These conditions were obtained by applying an iterative process to calibrate the parameter values using the law of error propagation. In the case of the optimal calibration condition, the variance of the estimated 3D information was measured quantitatively. The result was that no rotation angle, no cordinate translation, and a depth ratio of 1.1 are the best calibration conditions for the camera.

2.3 Conclusion

In summary, many studies have been conducted for correctly determining a camera pose based on the PnP problem, where there are multiple possible ways to solve it. Furthermore, there are also other methods for determining the pose of a camera that do not work based on the PnP-problem. Research efforts are intended to search for the optimization of the camera pose determination, so it can be used in various applications. These include augmented reality, autonomous navigation etc, and tries to implement this as efficiently as possible.

Nevertheless, such a case study has not yet been conducted examining the influence of the camera parameters obtained with the camera calibration. Studies on camera modelling for a theoretical approach seem to be scarce. The studies mainly examine the method of pose determination, and focus less on the effects of the camera in this process.

In the framework of this thesis two simulators are developed with camera models for mode selection and parameter changes. This helps to simulate the impact of scenarios with errors on the camera parameters and applies this to a case study for pose determination. Moreover, it does so in the context of cameras specifically used within Intermodalics. In this way, this thesis integrates a solution to optimize the pose determination of their cameras, for use in future applications. Finally, with this thesis a general connection is explored between the pose estimation and the effect of errors on the camera parameters or in the environment.

Chapter 3

Simulation frameworks

A rough overview of the components of the localization and mapping system is visualized in Figure 3.1.

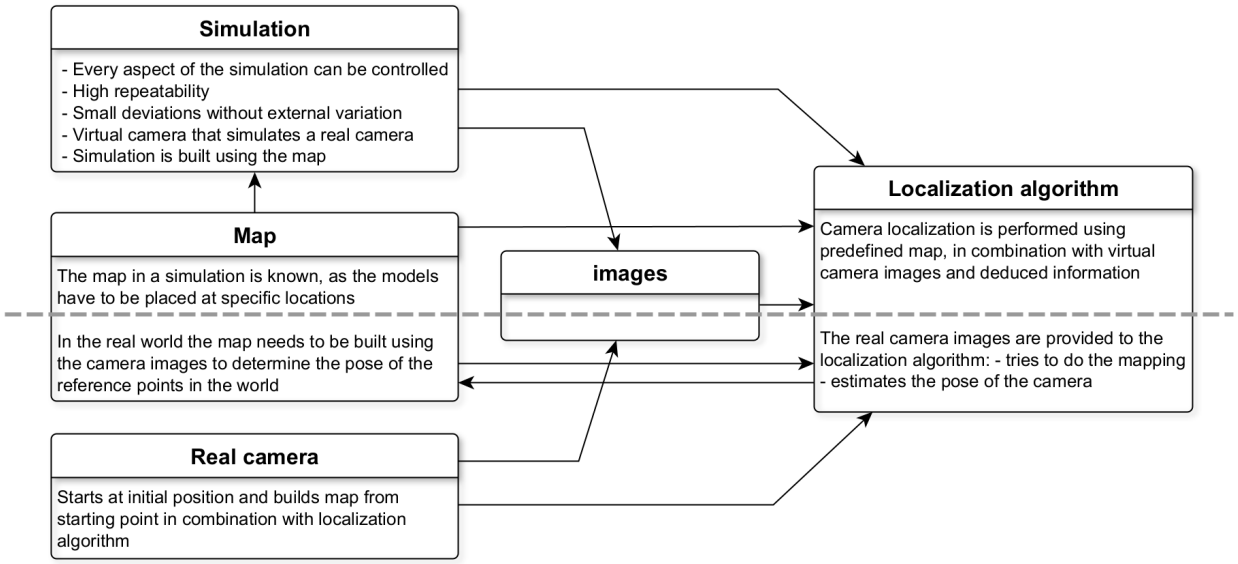


Figure 3.1: Overview of simulator use-case

Figure 3.1 shows the factors that the simulation could replace during the sensitivity analysis. Instead of using the real camera in a test setup, the simulator is setup with a predefined map. This predefined map simplifies the localization process as every pose of every model can be determined exactly. The exact poses determined in the map can be used to perform the camera localization, while an analysis with a real camera needs to detect and estimate the poses of the various models first and after building the map perform an accurate localization.

3.1 Python simulation

3.1.1 The simulation model

A simulation program containing two camera types was written in Jupyter [1]. In the notebook with the required self-written SimLib library, the camera pose is obtained according to the principle of the PnP problem. This requires the theoretical camera calibration matrix along with the coordinates of the projection points and the coordinates of the object points in the world reference system. To obtain the camera projections in the simulation, a 2D image plane of the camera must be formed which is explained in Section 3.1.3. This is accomplished using the given camera parameters and marker points in the simulation. An interesting question here is how the addition of errors or noise can affect the theoretical estimation of the relative position and/or rotation of the camera pose. In Figure 3.2, an example of a simulation is shown. The camera is represented by a black cross with the corresponding axes, the projection plane of the camera is displayed as a blue plane and the blue dots are the markers in the room. Figure 3.3 contains all the classes that are required for the simulation.

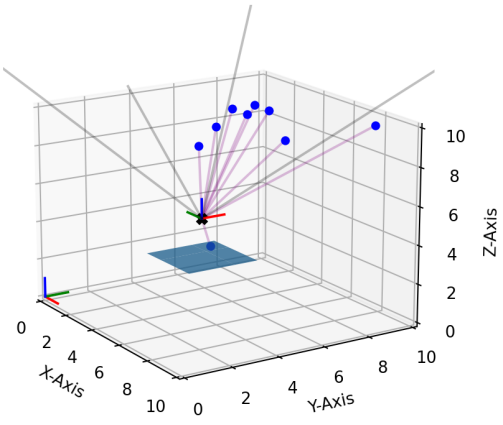


Figure 3.2: Design of the virtual test environment

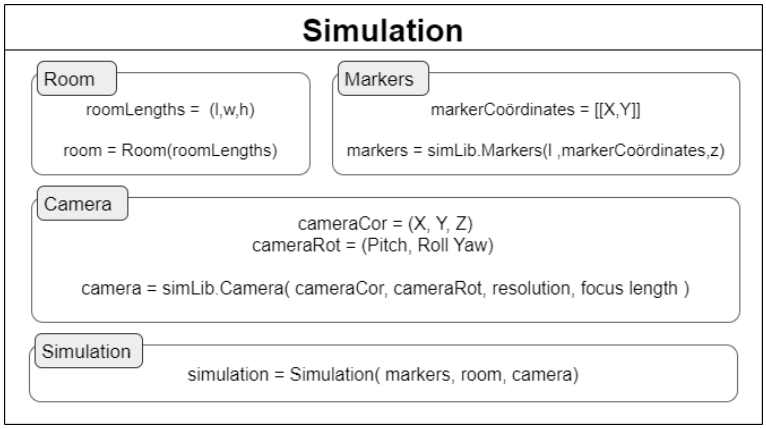


Figure 3.3: Classes in the virtual test environment

In the simulation environment, the chosen camera setup has to project the marker points in the current pose in order to accurately determine the camera location. All the marker points are mapped from their location on an image plane by the geometric approximation of a pinhole camera model. The localization of the camera depends only on the points in the field of view (FOV) of the camera. This makes the localization resilient to configuration changes, such as rotations of the camera where marker points appear and/or disappear. The only requirement is that a minimum of 4 marker points are required in the FOV to perform the localization. The processing pipeline of the localization is shown in Figure 3.4.

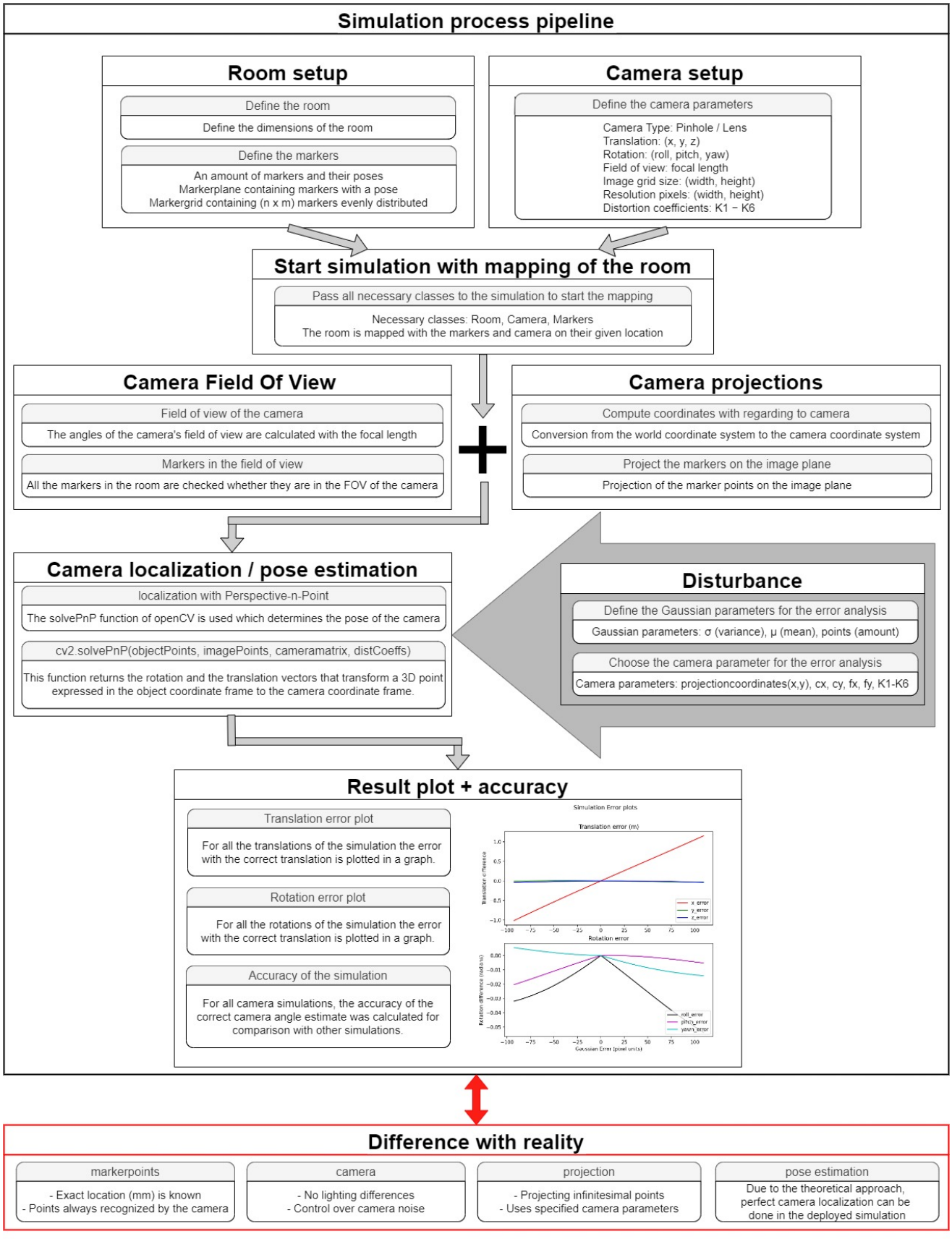


Figure 3.4: Processing pipeline of the simulation

3.1.2 Camera calibration

Camera calibration is used to determine the intrinsic properties of a camera. For this simulation, the user can define the intrinsic properties of the camera. Calibration of the camera is required to convert objects or points in the image into real-world coordinates. The creation of the calibration matrix is done when initializing the camera class in the manner shown in the following matrix:

$$camera.k = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

The camera extrinsic matrix [E] (3 x 4) is also created when the camera is initialized and resulting in the following equation:

$$camera.E = \begin{bmatrix} r_{camera1} & r_{camera12} & r_{camera13} & t_{camera1} \\ r_{camera21} & r_{camera22} & r_{camera23} & t_{camera2} \\ r_{camera31} & r_{camera32} & r_{camera33} & t_{camera3} \end{bmatrix} \tag{3.2}$$

In the case of lens cameras, the model is extended to include distortions. The distortion coefficients k_1, k_2, k_3, k_4, k_5, and k_6 are radial distortion coefficients. p_1 and p_2 are tangential distortion coefficients. The considered distortion coefficients are the following

$$camera.dist = (k1, k2, p1, p2, k3, [k4, k5, k6]) \tag{3.3}$$

3.1.3 Camera image acquisition

The camera in the simulation is able to form a 2D image from the 3D points in the environment that are within its FOV. For this image formation, a transformation is performed where the projection points are calculated using the calibrated camera. The camera parameters are used to make the projection on a image plane. The parameters of the 2 camera types are presented in Table 3.1 for the pinhole camera and in Table 3.2 for the lens camera.

Table 3.1: Pinhole camera parameters

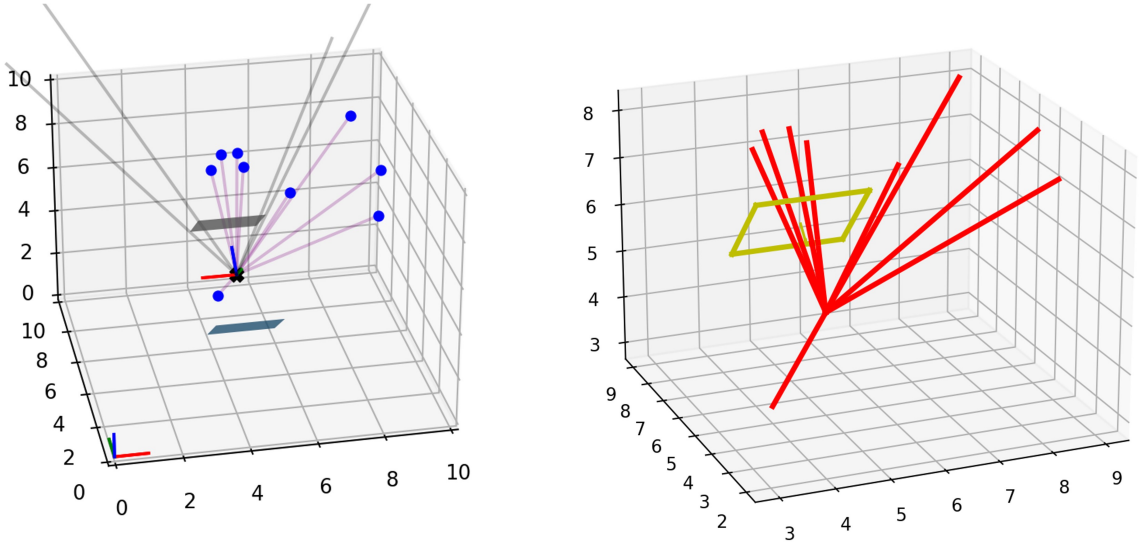
Camera:	
Translation	(x, y, z)
Rotation	(roll, pitch, yaw)
Field of view	focal length, aperture
Image grid	size (width, height)
Resolution	pixels (width, height)
Reference system:	
World coordinate system	Markers
Camera coordinate system	Projections

Table 3.2: Lens camera parameters

Camera:	
Translation	(x, y, z)
Rotation	(roll, pitch, yaw)
Field of view	focal length
Image grid	size (width, height)
Resolution	pixels (width, height)
Distortion coefficients	$k_1 - k_6, p_1, p_2$
Reference system:	
World coordinate system	Markers
Camera coordinate system	Projections

Field of view

If the coordinates of a point P are within the field of view of the camera and no object is located in front of the point, then the projection of that point is visible, otherwise the point is not visible and can be omitted. To verify in the simulator that the point is within the field of view of the camera, a plane is drawn that defines the field of view of the camera. This is called the FOV-plane, and is represented as a gray area in Figure 3.5(a) and as a yellow outline in figure 3.5(b).



(a) Field Of View of the camera

(b) Principle visualisation of the field of view

Figure 3.5: Field of view visualisation in the simulator

The FOV-plane is drawn using Algorithm 1. By drawing lines from the points to the camera, a check can be done whether there is an intersection between the line and the FOV-plane. In Figure 3.5(a) these are the purple lines and in Figure 3.5(b) these are the red lines. In the simulation program, the principle of the field of view can be visually rendered as shown in Figure 3.5(b). This allows an easy inspection whether the correct points are taken within the field of view of the camera. The calculation is explained in Algorithm 1.

Function: find_points_in_FOV()

Input: camera: parameters, room: dimensions
    Calculate the angles of view of the camera with the camera parameters
    Define the FOV-plane with the calculated angles
    for all points in the room:
        draw line from point to camera
        if intersection between line and FOV-plane:
            append point to list
    end for
Result: FOV_points: points in the FOV of the camera

Algorithm 1: Calculation for getting the points in the FOV of the camera

Projection coordinates pinhole model camera

To transform a 3D point into a 2D image point, using the pinhole camera model presented in Section (2.1.4), Algorithm 2 "camera.compute_coordinates_wrt_camera()" is applied to determine the transformation of coordinates to the image plane. This algorithm performs a transformation from the 3D world coordinate system to the camera coordinate system. The function takes as input the world points and converts them to points for the camera coordinate system with the extrinsic matrix E of the camera according to equation 3.5.

Function: camera.compute_coordinates_wrt_camera()

Data: points_w: worldPointsInFOV matrix, E: extrinsisc matrix (eq. 3.2)

The projection points (points_c) are calculated with a matrix multiplication of the extrinsic matrix (E) and the matrix containing the world points (points_w).

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{3.4}$$

Result: points_c: points in the camera coordinate system

Algorithm 2: World coordinates to camera coordinate system function

The next step is a conversion from the homogeneous coordinates in the camera coordinate system to 2D points on the image plane. This conversion is executed with the function compute_image_projection(). The resulting projection coordinates can be plotted on the camera image plane. An example of a plot obtained from a simulation is shown in Figure 3.6. These are not yet the final coordinates in the camera image, because the number of pixels on the image plane must still be taken into account. The more pixels there are, the more accurately the coordinates of the projections can be recorded. The detailed explanation is given in the upcoming section about pixelisation.

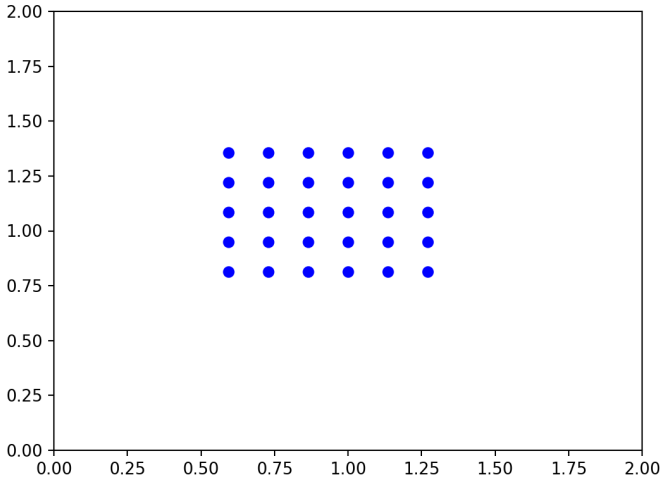


Figure 3.6: Pinhole camera image plane with projections

Function: compute_image_projection()

Data: points: points in the camera coordinate system, k: calibration matrix (eq. 3.1)

The function calculates the 2D projections of 3D points on the image plane, given the intrinsic camera parameters. This is calculated by doing a matrix multiplication of the intrinsic matrix (k), extrinsic and the matrix containing the points in the camera coordinate system (points_c) shown in equation 3.5.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} \tag{3.5}$$

Result: points_i: points on the image plane

Algorithm 3: Compute the image projection

Projection coordinates lens model camera

Real lenses experience distortion in their projections, with radial and tangential distortion being the 2 possible types of distortion. As a result, the aforementioned model is extended with an input vector of distortion coefficients as follows: $(k_1, k_2, p_1, p_2[, k_3[, k_4, k_5, k_6]])$ of 4, 5 or 8 elements. Where $k_1, k_2, k_3, k_4, k_5, k_6$ are radial distortion coefficients and p_1 and p_2 are tangential distortion coefficients. These distortion coefficients are taken into account when projecting the markers on the image plane of the camera. To determine these projection points the function projectPoints [32] of the OpenCV library, which uses E (eq. 3.2), k (eq. 3.1) and the distortion coefficients, is applied. The projection coordinates obtained from the function can be plotted on the camera image plane of the lens camera. An example is shown in Figure 3.7.

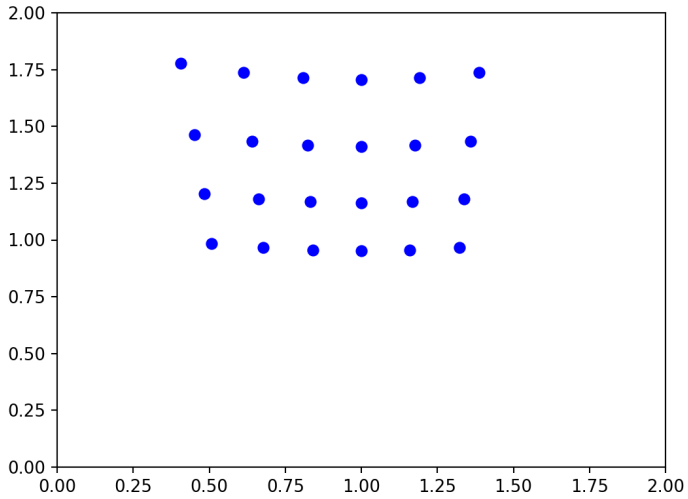


Figure 3.7: Lens camera image plane with projections

Pixelisation

A projection of a 3D marker will be represented as a point in the image. A point in a digital image is a pixel with a 2D coordinate, and these coordinates must be integers. The intensity of the pixels is not taken into account, but is set equal to zero intensity (no coordinate point) or maximum intensity (a 2D coordinate point). The coordinate system for representing the pixel points is also called the pixel grid. This coordinate system is illustrated in Figure 3.8 where a pixel in this coordinate system is one unit long for u and v. The projection coordinates defined in the orthogonal coordinate system must now be converted to the pixel grid coordinate system in terms of pixel coordinates. The change of the coordinate system is accomplished by first remapping the projection coordinates in the range of [0,1]. Since the size of the image grid is known, the normalised projection coordinate for u and v can be calculated as follows:

$$\begin{aligned} P_{u\_normalized} &= \frac{P_u + image\_width/2}{image\_width} \\ P_{v\_normalized} &= \frac{P_v + image\_height/2}{image\_height} \end{aligned} \tag{3.6}$$

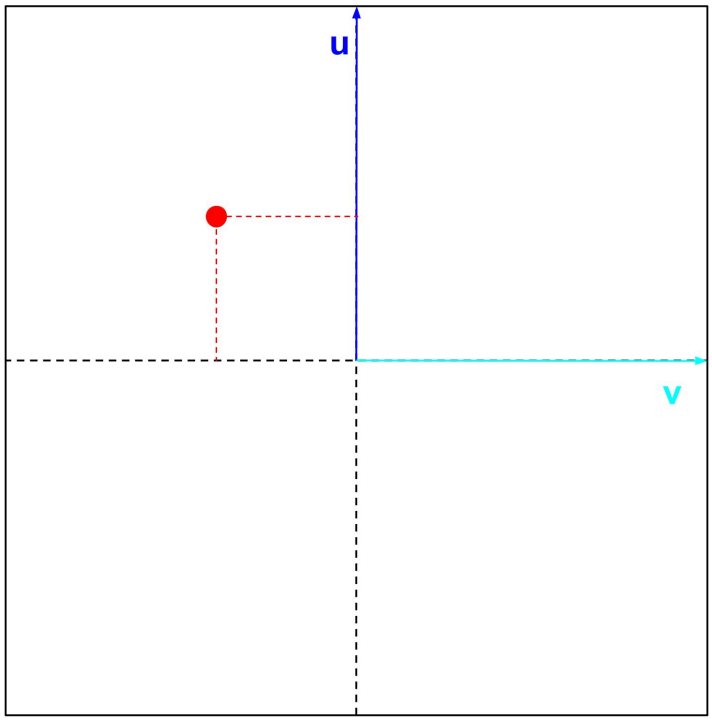


Figure 3.8: Projection grid coordinate system

Since the coordinates of projection points are now in the range [0,1], they are called normalized. These points can be considered in another coordinate system in which the points are defined after normalization. This is also known as the Normalized Device Coordinate system or NDC space. With the origin of the NDC coordinate system presented in Figure 3.9 being in the lower left corner of the axes system. It is important to note here that the coordinates are still real numbers, only within the range [0,1].

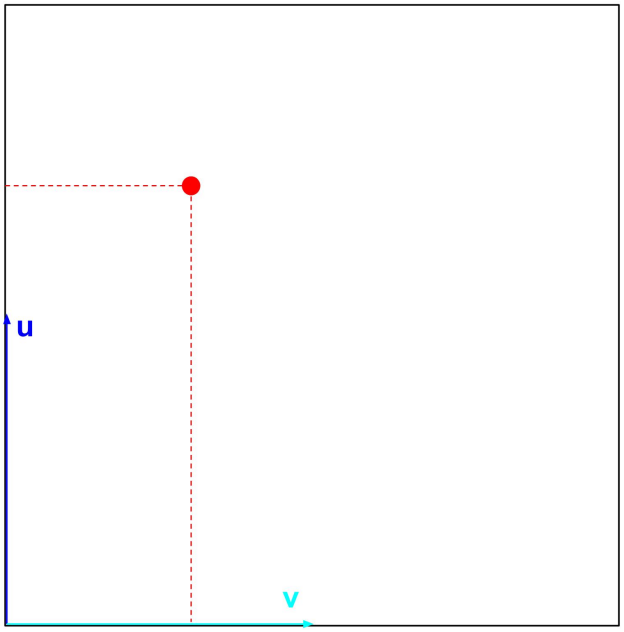


Figure 3.9: NDC coordinate system

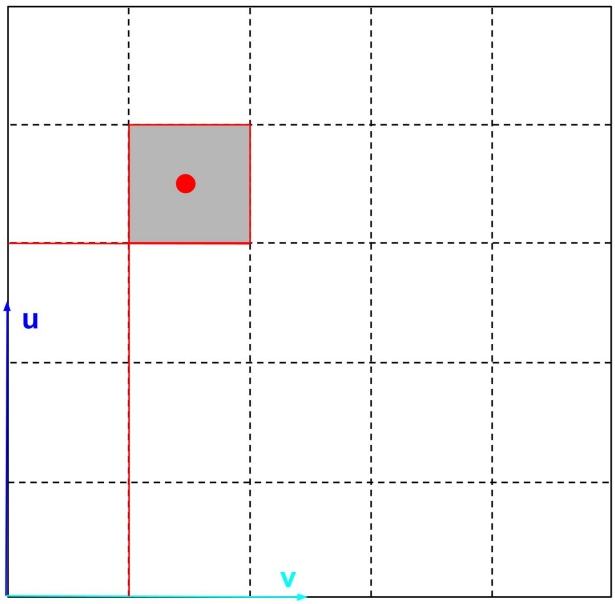


Figure 3.10: Pixel grid coordinate system

Finally, the normalized projected point u- and v-coordinates in the NDC space must now be multiplied by the pixel width and pixel height of the image, respectively. This is equivalent to a simple remapping of the range [0,1] to the range [0,Pixels in width] for the u-coordinate, and [0,Pixels in height] for the v-coordinate. And significantly, the pixel coordinates are integers and therefore rounding of the resulting numbers to the smallest next integer value is necessary (for doing that, the mathematical floor function can be used; this function rounds a real number to the smallest previous integer. For example, $\lfloor 2.4 \rfloor = 2$ and $\lfloor 2.8 \rfloor = 2$). The result returns the coordinates of the projected point defined in the pixel grid. The conversion is done with the following described formula:

$$\begin{aligned} P_{u\_grid} &= \lfloor P_{u\_normalized} * PixelsInWidth \rfloor \\ P_{v\_grid} &= \lfloor P_{v\_normalized} * PixelsInHeight \rfloor \end{aligned} \tag{3.7}$$

This results in the coordinates of a projection point in the pixel grid coordinate system as shown in Figure 3.10. By this method, the resolution is taken into account in the camera model for the projection of the points on the image plane.

3.1.4 Perspective-n-Point pose estimation

The Perspective-n-Point (PnP) computation problem consists of calculating the rotation and translation vectors that transform a 3D point expressed in the object coordinate frame to the 2D camera coordinate frame. The SolvePnP function and related functions are from the OpenCV library [2] and are described in the Section 2.1.2 Perspective-n-Point of the literature review. These functions are used for estimating the camera position. The parameters required for pose estimation are listed in Table 3.3.

Table 3.3: Perspective-n-Point pose estimation parameters

World parameters:
Markers in the FOV (3D)
Camera parameters:
The image projections (2D)
The intrinsic camera matrix (K)
The distortion coefficients of the camera (K1 - K6, P1-P2)

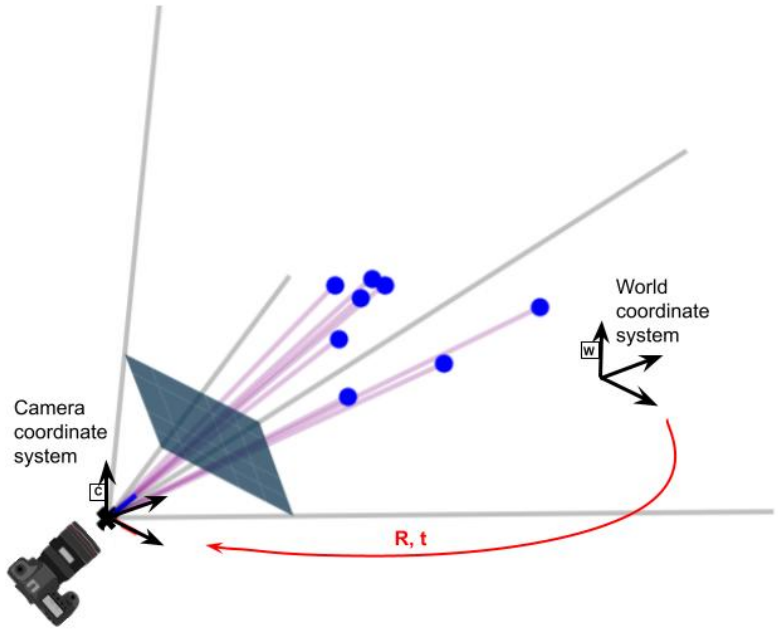


Figure 3.11: Visualization of the PnP problem in the simulation

The estimated pose determined by the SolvePnP method is thus composed of the vectors of rotation rvec (eq. 2.10) and translation tvec (eq. 2.11) that allow translating a 3D point stated in the world coordinate system into the coordinate system (Figure 3.11):

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{camera1} & r_{camera12} & r_{camera13} & t_{camera1} \\ r_{camera21} & r_{camera22} & r_{camera23} & t_{camera2} \\ r_{camera31} & r_{camera32} & r_{camera33} & t_{camera3} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{3.8}$$

Rodrigues' rotation formula

The PnP calculation of OpenCV returns a translation vector tvec and a rotation vector rvec obtained in the form of equation (2.10) where the axis–angle representation is used. This rotation vector must be transformed to find out the rotations around the axes. Using the Rodrigues function from OpenCV the rotation vector is converted to the rotation matrix. From this rotation matrix the angular rotations around the axes can then be derived. The algorithm for the computation of the Euler angles is described in algorithm 4.

Function: rvec_to_Euler_angles(rvec)

Data: input: rvec
output: Euler angles
Transform rvec to the rotation matrix: cv2.Rodrigues(rvec);
Convert rotation matrix to Euler angles: rot_matrix_to_euler(R);
Result: Euler angle rotations

Function: rot_matrix_to_euler(R)

Data: input: R
output: Euler angles
Convert rotation matrix to rotation angles
if angle > 360 **then** angle = angle%360;
(if the angle is greater than 360 the result is the remainder of the division of 360)

Algorithm 4: Rotation vector to Euler angles

3.1.5 Accuracy of the simulator

Simulator translation tests

To verify that the simulator is working properly, several tests are performed to determine the accuracy of the camera localization. First, different locations were tested for the camera. The camera contains the parameters defined in Table 3.4. The translation of the camera is tested for every possible camera translation with 0.5 m in between. Only if the position of the camera is correctly estimated down to 1mm and to 0.001° then the pose estimate is only considered correct.168 simulations were run with correct localization in 99.4% of all tests. Only for the camera coordinates where x, y and z are respectively equal to 3.5, 1.5 and 0 a wrong pose is estimated is due to a miscalculation of the used SolvePnP function of OpenCV. This location should therefore be avoided when performing the sensitivity analysis.

Table 3.4: Camera test parameters

Camera:	
Translation	(2, 2, 1)
Rotation	(90°, 0°, 180°)
Focal length	250
Image grid	(300, 300) pixel units
Resolution	(500, 500) pixels
Reference system:	
Room dimensions	(4, 4, 2) meters
World coordinate system	10 markers

Simulator rotations tests
To test the localization accuracy for different rotations, the same default values are used for the camera described in Table 3.4. In an automated simulation, a large amount of possible camera rotations are now used to determine the camera pose. Again only if the position of the camera is correctly estimated down to 1mm and to 0.001° then the pose estimate is only considered correct. From the automated simulation, there were 7959 simulations for which there were enough points in the FOV of the camera to determine the pose. Of these, there were 4287 where the location of the camera was correctly ascertained. This is an accuracy rate of 53.86%. When performing the error analysis, first the desired rotation must be checked for a correct pose estimate.

Simulator remaining parameters tests
After the translation and rotation tests, all other camera parameters (f_x, f_y, c_x, c_y, K_1, K_2, K_3, K_4, P_1, P_2) were tested in turn. For all these parameters, the simulations gave no errors in the pose estimation of the camera using OpenCV's SolvePnP function. Therefore, this should not be taken into account when testing the sensitivity analysis. A note here is that a bound was taken on choosing the possible focus length since the light beams can continue infinitely in the simulation which is not realistic.

3.1.6 Disturbance modelling
A normal distribution or Gaussian distribution is used to model noise on the cameras in order to obtain an error on the simulation for the error analysis. This is called Gaussian noise, which is a statistical noise with a probability density function (PDF) as shown in Figure 3.12. To model the noise in the simulation, the parameters $N(\mu, \alpha)$ can be adjusted to determine the distribution of the Gaussian noise. Where the parameter α determines the amount of noise, and can be set by the user. The other parameter μ affects the mean of the errors that are added, if $\mu = 0$ the added errors will have a random deviation of $0 + \alpha$. Figure 3.11 shows that the probability of greater noise is rapidly decreasing as a result of the decreasing distribution in the graph. Additional noise effects, such as spatially and temporally varying contrast thresholds due to electronic noise, or limited bandwidth of the event pixels, are research questions that are not considered for this study.

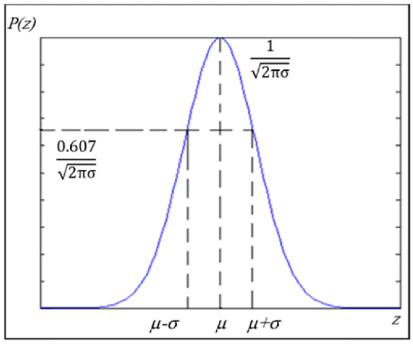


Figure 3.12: Gaussian noise probability distribution function

Table 3.5: Camera error analysis parameters

Camera parameters:	
focal lengths of the intrinsic camera matrix	(f_x, f_y)
principle point of the intrinsic camera matrix	(c_x, c_y)
distortion coefficients of the camera	$(k_1 - k_4, p_1, p_2)$
Projection points:	
X- and/or Y-coordinates of the projection points	p(X,Y)

For a chosen number, errors are extracted from the normal distribution or from the Gauss distribution. These errors can then be added to the desired camera parameter which are used in a simulation to perform an analysis. These results will be discussed in Chapter 4: Sensitivity analysis. For the simulation experiments, errors can be added to the following camera parameters or projections presented in Table 3.5.

3.2 Gazebo simulation

3.2.1 The simulation environment

To approach the problem from a more realistic perspective, a virtual world is built. This world uses plugins to replicate a virtual camera with its own intrinsic parameters and several ArUco-markers at various locations with known coordinates. The simulation will feature a marker detector to retrieve a six DOFs pose for every marker in the camera coordinate frame. These coordinates need to be converted to world coordinates. This can be done using the detected transformation matrix yielded from the ArUco-marker detector. The measured transformation can then be combined with the known virtual map to retrieve the camera location itself. This is a simplification to normal feature point detectors, as normally only 2 DOFs can be estimated from a camera image, which would be the *(u,v)*-coordinates in the camera image. For the simulation the same question arises as previously mentioned: "What is the influence of manually implemented errors on the pose estimation for the camera, as well as the influence of various distortion models?" For the Gazebo simulation the focus will be on implementing various distortion models on the camera while including errors, as well as the influence of the camera calibration on the detection accuracy depending on distance and size of the markers.

Table 3.6: Gazebo simulation details

Camera:	
Translation	(x, y, z)
Rotation	(roll, pitch, yaw)
Resolution	(width x height)
Field of view	fx, fy
Distortion/intrinsic factors	D, K
Lens distortion	
Reference system:	
World coordinate system	
camera coordinate system	
ArUco markers:	
marker size	40 - 400 mm
marker distance	1 - 20 m
marker orientation	yaw: 0° - 50°

The simulator is built in the virtual environment Gazebo. In this environment a plugin exists to simulate a virtual camera with various distortion models. The camera image is then published to a ROStopic where another subscriber, in this case the ArUco-marker finder, can read the image. The ArUco-marker finder script then detects the various markers within the given image and publishes an array of the detected markers with their pose in the coordinate system of the camera. With these coordinates the camera pose can be calculated. In Figure 3.13 an overview of the important aspects for the simulation is given. This is divided to separate the input parameters, the called services and output publishers.

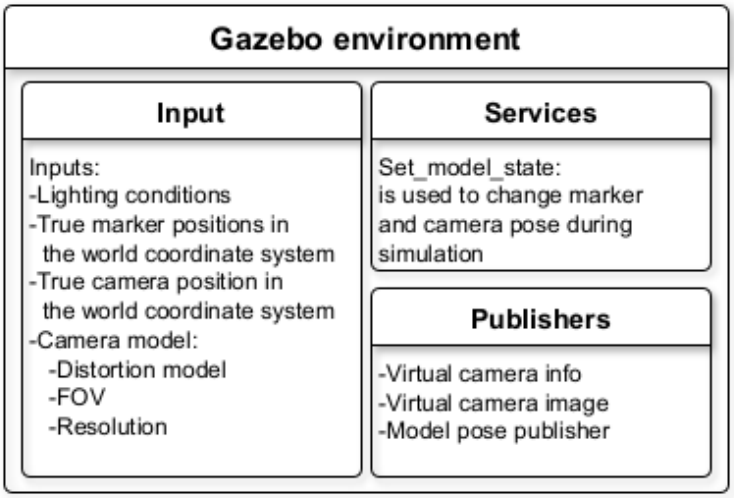


Figure 3.13: Gazebo inputs and attributes

Figure 3.13 shows the input parameters that need to be initialized to setup the environment. The camera with its parameters are created and the output publisher is started. This creates the virtual image stream, as well as publish the camera model info. The environment, because of its realistic nature, needs additional lighting. If lighting is not added, the detector will not be able to detect the markers in the image. For this simulation lighting conditions will not be taken into account, so enough light will be provided. Following this, the markers are placed in the environment at their initial poses. The service: "set model state" is initialized for every model in the simulation, when they are generated in the world. This service can be called to alter the pose of the model from the terminal, or a python script from the simulation itself. Finally for every model the pose is published using the model pose publisher. This is done so the pose of every model can be requested from different scripts.

3.2.2 Camera calibration

In reality a camera also needs to be calibrated to correct the image for its distortion model. This can be done manually by using a python script and a checkerboard with a fixed size of the squares. The calibration is necessary to retrieve the intrinsic camera matrix and distortion coefficients that will be used to eliminate perspective distortion on the image created by the camera. The perspective distortion needs to be compensated to be able to use the camera for most vision based localization applications. The virtual camera otherwise, is setup using a Gazebo plugin. The plugin can implement various cameras with different fields of view, as well as various distortion models and resolutions. These camera parameters can then be used to simulate various real camera models, as well as implement a calibration error by supplying an erroneous camera matrix to the visual application. With the known deviation on the camera matrix an error on the localization can be analyzed. The virtual camera image is published to a ROS-topic where different subscribers can use these images to perform a marker detection or other visual related tasks. The camera information publisher is started alongside the image to provide the camera matrix to the various modules.

3.2.3 ArUco-marker detector

The ArUco-marker detector takes a virtual or real camera image in combination with the camera specifications to accurately detect markers in the image, and give a pose estimate of the marker in the camera coordinate frame. An overview of the implementation in this simulator is given in figure 3.14.

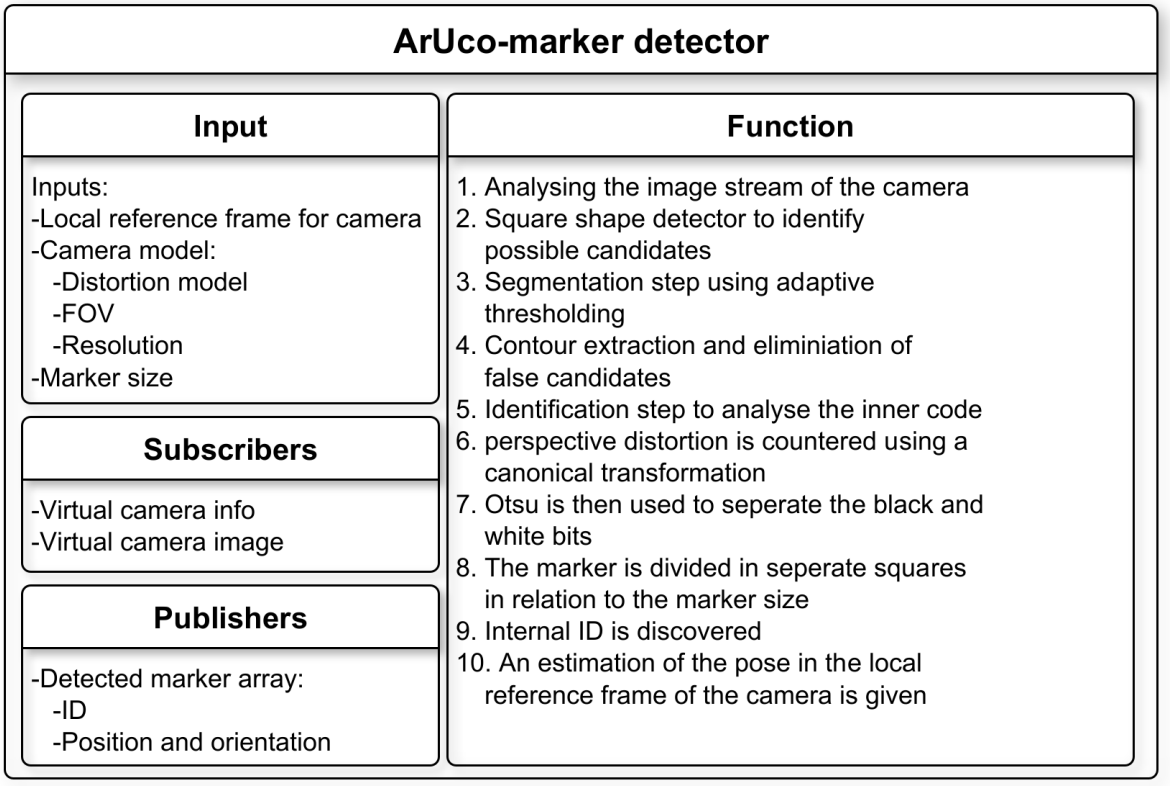
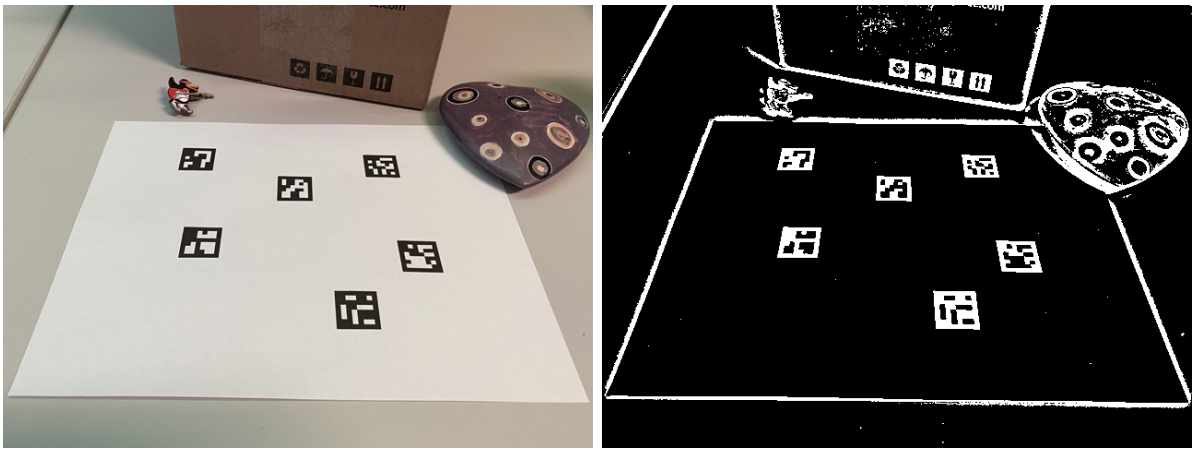


Figure 3.14: Overview ArUco-marker detector

The detection of marker candidates starts with an analysis of the image stream from the virtual camera. Thereafter, square shapes are identified as possible markers in the image. The candidate selection process starts with an adaptive thresholding step, this is visualized in Figure 3.15.



(a) (b)

Figure 3.15: Adaptive threshold implementation: (a) Original image, (b) Thresholded image [33]

Following this step, the contours that resemble a square shape are extracted from the thresholded image, and contours that do not approximate to a square shape or are not convex are discarded. Some extra filters are applied to remove contours that are too close to each other or too big or small.
After these steps the candidate list is assembled, and an analysis of the inner code is executed to verify if the selected candidates are actually markers. The identification process begins with a perspective transformation using the camera intrinsics to retrieve the canonical form of the marker candidate, after which a thresholding step using Otsu, explained in Section 2.1.5, is used to separate black and white bits. Otsu's method is a form of global thresholding, but avoids having to choose the value as it determines the threshold automatically from the image histogram. After the thresholding, a segmentation step is applied and the marker is then divided in a grid in relation to the marker size and amount of bits determined by the chosen library. Following this step the internal identification number is determined.
After the identification, the pose of the marker is calculated. For the calculation, the intrinsic camera parameters and distortion coefficients need to be known. In the simulation they are provided directly from the virtual camera. With the given parameters, an estimation of the marker pose is calculated. This can be done by estimating a scale factor of the marker in the image depending on the given marker size. Using this method an estimation of the distance of the marker to the camera can be made, this combined with the *(u,v)*-coordinates in the camera image give a reference position in the camera coordinate system. The orientation of the marker is estimated based on the warp on the angles of the square shape. To implement a calibration error and verify the deviation on the pose, a camera matrix with a small deviation can be provided. Finally, the marker detector publishes an array of the detected markers along with their pose to a ROS-topic as previously mentioned in the overview in Figure 3.14.

Figure 3.16 shows an image provided by the virtual camera with a marker that is detected within the frame.

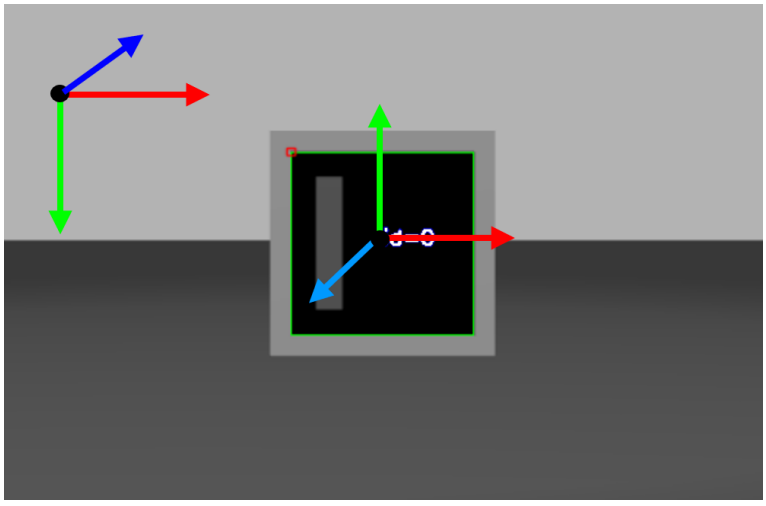


Figure 3.16: Virtual camera image with detected marker

As visible in Figure 3.16 the detected marker is overlayed with its estimated transformed coordinate system, provided within the local camera coordinate system. This image also shows the overlay provided by the detector, as it puts the ID as well as the detected borders of the marker in the image. Thanks to the overlay it can be easily verified if the correct contour was detected.

3.2.4 Transformation

To do an estimation of the pose for the camera, the detected marker transformation provided by the ArUco-marker finder needs to be transformed from the camera coordinate system to the global world coordinate system. For this transformation the camera coordinate frame is first rotated around the x-axis by 90°, followed by a rotation around the z-axis of 90°, this is visualised in Figure 3.17. By performing these rotations the camera coordinate frame aligns with the world coordinate system, when the camera is in zero position. This simplifies the following transformations, as it does not alter the accuracy.

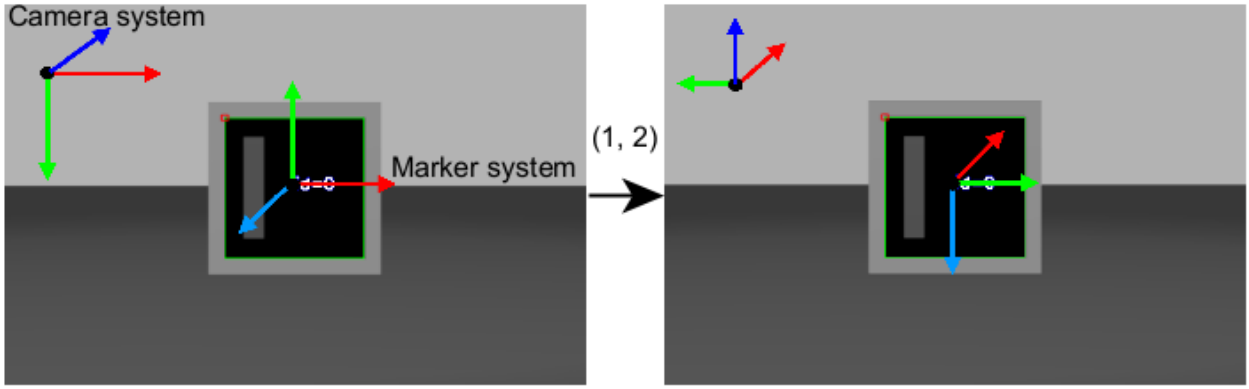


Figure 3.17: Visualization of rotations in camera coordinate system

The rotations, applied in Figure 3.17, have the purpose of aligning the camera frame to the world frame in orientation. This is done so the roll, pitch and yaw rotations in Gazebo are in accordance to the calculated orientation discussed in the following section. Firstly, the orientation parameters need to be converted from quaternions, which are used in Gazebo as well as in the marker detector, to the rotation matrix. This is done using Algorithm 5.

Function: quaternion_rotation_matrix()

Data: Input: qw, qx, qy, qz the four variables of a quaternion
Output: Rotation matrix

The rotation matrix is formed from the four quaternion pose variables
Result: [3x3] rotation matrix

Algorithm 5: Quaternion to rotation matrix

To calculate an estimation of the camera pose, the aligned marker transformation in the camera coordinate system needs to be transformed to the global coordinate system. The camera pose can be referred to as the transformation matrix of the world to the camera. This matrix can be calculated by multiplying the transformation matrix of the true pose of the markers in the world axis system, with the inverse of the detected transformation matrix of the marker in the camera axis system This is visualized in Equation 3.9.

$$ {}^{World}_{Camera}T = {}^{World}_{Marker}T * ({}^{Camera}_{Marker}T)^{-1} \tag{3.9} $$

3.2.5 Pipeline of the Gazebo simulation

Figure 3.18 gives an overview of how the different nodes and configurations interact with each other.

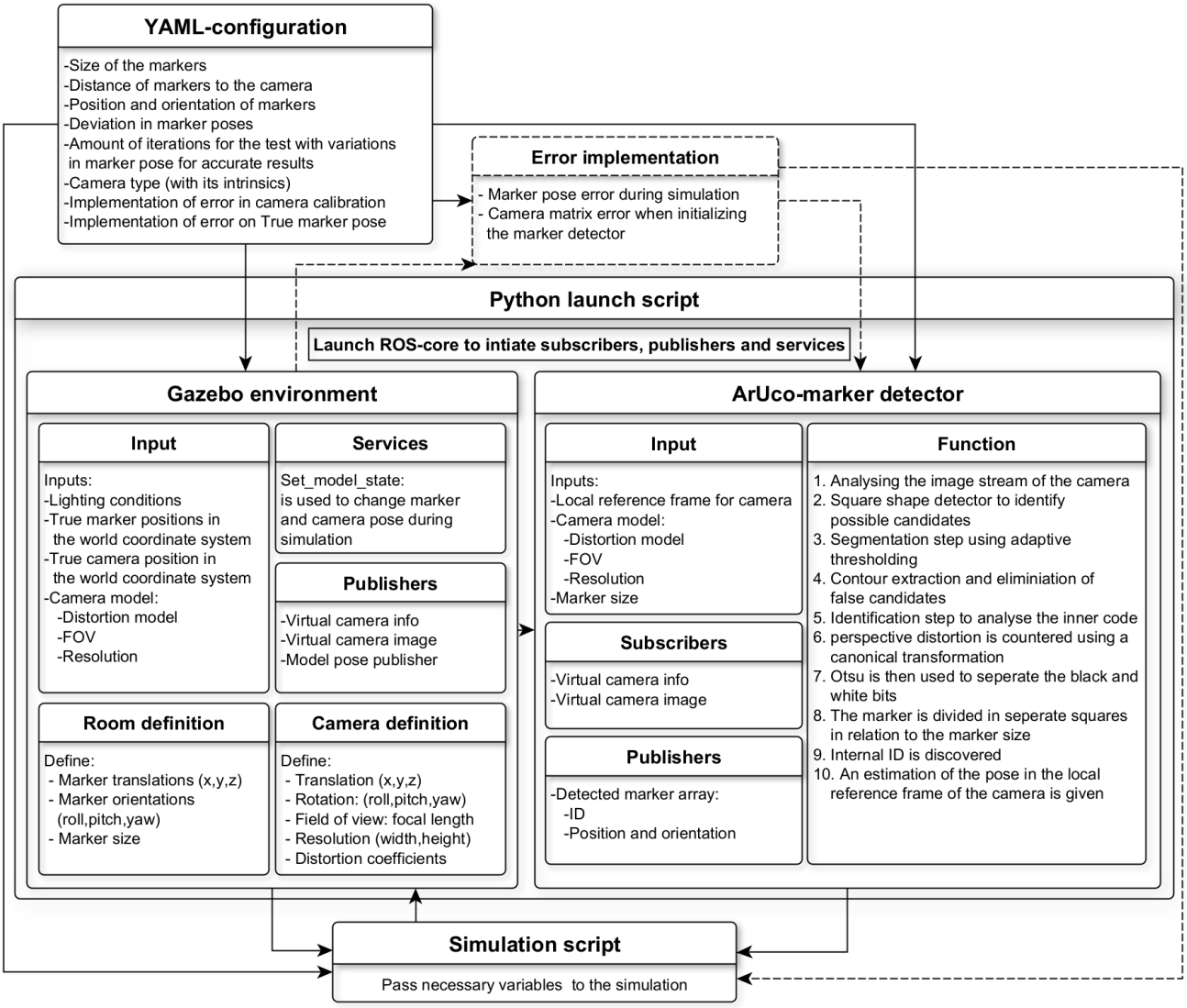


Figure 3.18: Pipeline for initializing the Gazebo simulation

The Gazebo simulation can be configured and started from a single yaml-file in combination with a Python launch script. All configuration variables can be entered in the yaml-file which will automatically be loaded and configured. The Python script launches ROS-core, and then distributes the configuration parameters to the different nodes, and initializes the necessary launch files to start the Gazebo environment with the markers and camera model. The ArUco-marker detector will be launched in a separate node in the same terminal. The various nodes communicate with the simulation script via various ROS-subscribers and publishers, the flow of information is visualized in Figure 3.18. The simulation script then uses this information to perform an analysis, an overview of the different steps in the simulation is given in figure 3.19.

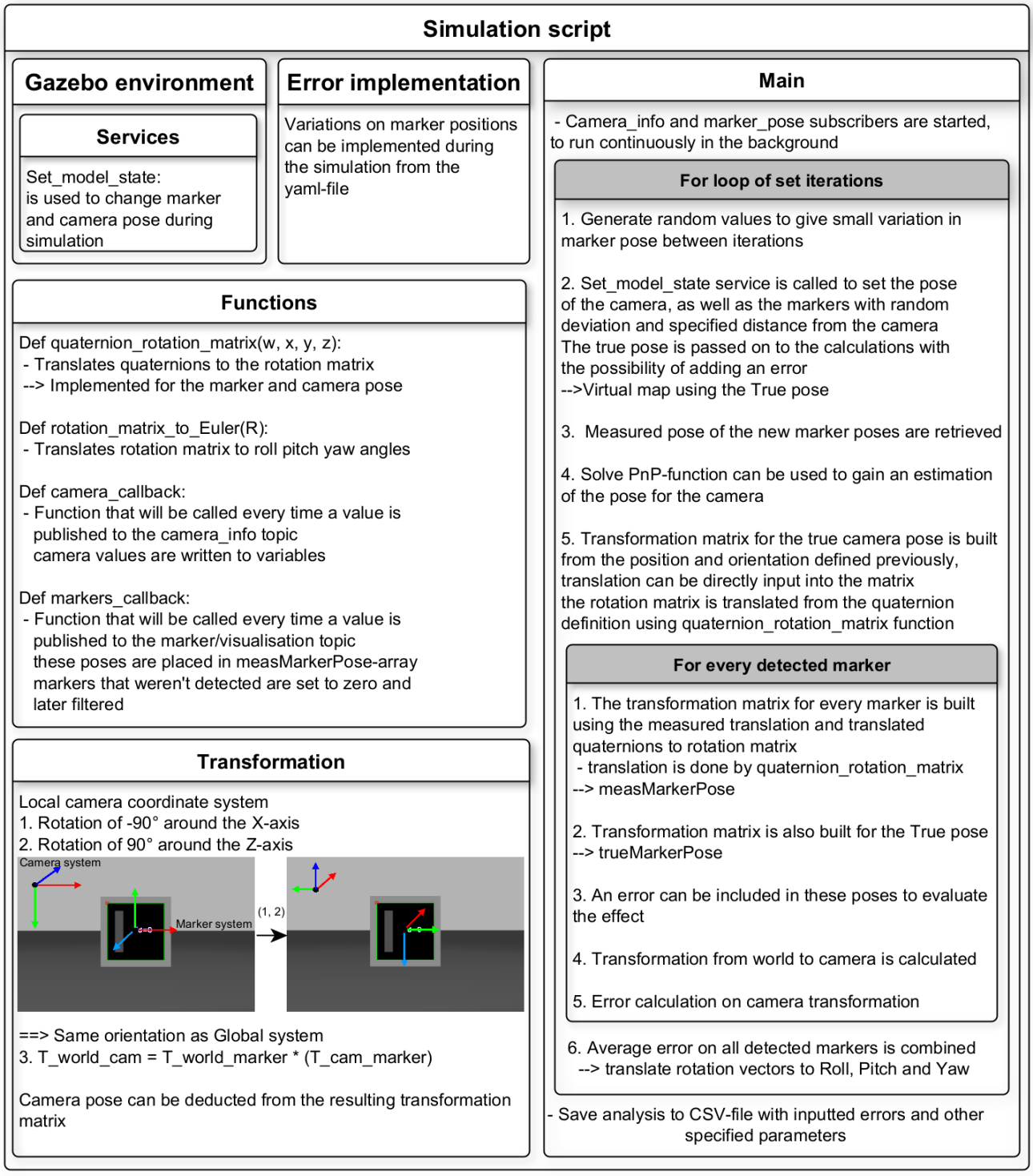


Figure 3.19: Overview of localization calculations in the simulation

As visible in Figure 3.19 the script contains various parts that run simultaneously. When the script is initialized two ROS-subscribers will be started, and they will run continuously in the background. Their accompanying functions will be called every time information is published to the subscribed topic. The next step in the script is to change the pose of the markers and camera to the desired pose according to the yaml-file. The position that the models are placed at will also be used as the true pose for further calculations, acting as a virtual map. Errors could be implemented in this map to analyze the effect on the localization. In the meantime the ArUco-marker finder has detected and published an array of the detected marker poses in

their new positions. These new poses will be used to establish the transformation matrix of the markers in the camera coordinate system. Using the true and measured transformation matrices the world to camera transformation matrix is deduced, following the steps mentioned in Section 3.2.4. The transformation matrix will then be divided in the (x,y,z)-position, and the remaining rotation matrix will be converted to Euler angles using Algorithm 6.

Function: rotation_matrix_to_euler()

Data: Input: [R]
Output: Roll, Pitch and Yaw in range (-180° - 180°)

The Euler angles are calculated from the given rotation matrix, if it is not singular
Result: The resulting Roll, Pitch and Yaw parameters

Algorithm 6: Convert rotation matrix to Euler angels

After estimating the camera pose, the error can be calculated for the six DOFs. An average error is calculated over the amount of iterations, this is done to give an accurate representation of the accuracy. Between iterations the markers are given a new pose that has a random deviation, so the accuracy is dynamically verified. The script then concludes the setup parameters and results in their respective csv-files to give an overview of the simulation.

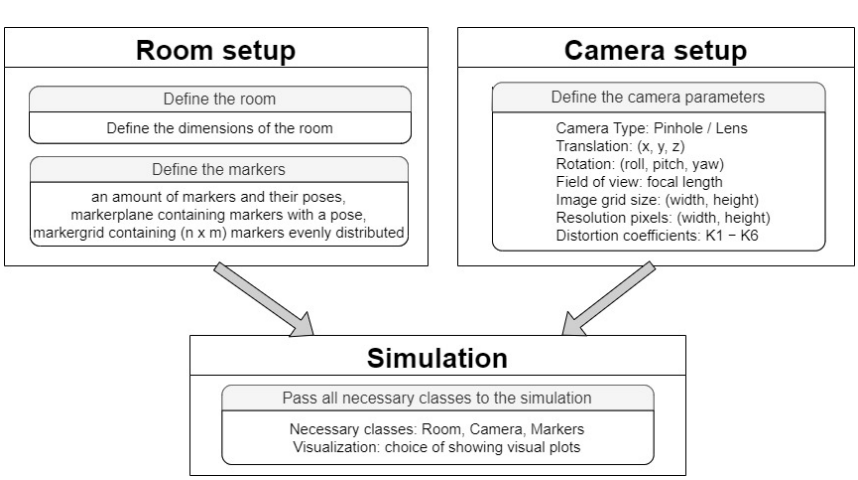
Chapter 4

Sensitivity analysis

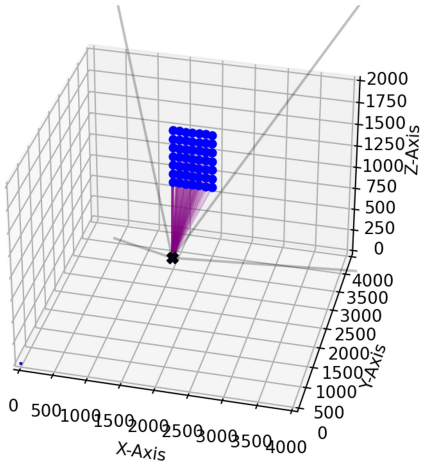
In this chapter, with the help of the proposed simulators, a sensitivity analysis is done on the simulated cameras and environment parameters, for the camera localization using the PnP-problem approach or marker detectors. The following simulations show the influence of errors on the camera images, and variations in the camera parameters. Different cameras will be used in the proposed simulators from Sections 3.1 and 3.2, respectively the python simulator and the Gazebo simulator. The results will be examined and discussed. Finally, the outcome of the two different simulators will be compared.

4.1 Python simulation results

To analyze the influence of adjustments of the camera parameters in the Python simulation, an identical scene is simulated as in Figure 4.1(b), with the same camera pose and markers but with adjustments on the different camera parameters and/or projection points as listed in Table 3.5. To calculate the error due to the additional noise, an error quantification can be obtained by comparing the camera pose estimate with the ground truth of the camera pose. These errors on the estimation of the camera pose are plotted and will be discussed in this chapter.



(a) Simulation setup



(b) Visualisation of the scene

Figure 4.1: Scene of the room with the camera and the markers

4.1.1 Camera parameters

The simulated cameras are the following: the Intel RealSense Tracking Camera T265 [34] and the Intel Depth Camera D435 [35]. By adopting the intrinsic properties of these cameras, a theoretical error analysis for these cameras can be performed. The intrinsic properties presented in Table 4.1 of these cameras were found online on question forums: ROS Answers [36] for the t265 and Github [37] for the d435. These intrinsic properties were determined using camera calibration.

Table 4.1: Camera intrinsic properties

	Intel T265 [34]	Intel D435 [35]
Fx	284.509100	322.282410
Fy	282.941856	322.282410
Cx	421.896335	320.818268
Cy	398.100316	178.779297
K1	-0.014216	-0.010342
K2	0.060412	0.081558
P1	-0.054711	-0.079618
P2	0.011151	0.0517975
Pixels	(640x360)	(640x360)

4.1.2 Specified camera image formation

After starting the simulation it is established which markers in the room are in the FOV of the camera, the markers that meet this condition are merged into a list. How this condition is verified is explained in Section 3.1.3. With this list of markers, a projection image is going to be formed that will be used later in the simulation for pose estimation. Using the specified intrinsic camera parameters, the marker points are projected onto the image plane of the camera. One key note is that there must be a projection of at least 4 markers and their corresponding 3D coordinates must be known.

4.1.3 Camera pose estimation

To estimate the camera pose, the OpenCv's solvePnP function [4] is used. This requires the marker points, image points, intrinsic matrix and the distortion coefficients of the camera to solve the pose. The working principle of this calculation is explained in Section 3.1.4. The result of this calculation is a translation and rotation vector of the camera. To perform the error analysis errors are added to the parameters in Table 3.5 which are used by the solvePnP function [4]. The errors are created according to the description in Section 3.1.6 these errors represent real noise or calibration errors. The entire simulation process performed for the error analysis is illustrated in Figure 4.2.

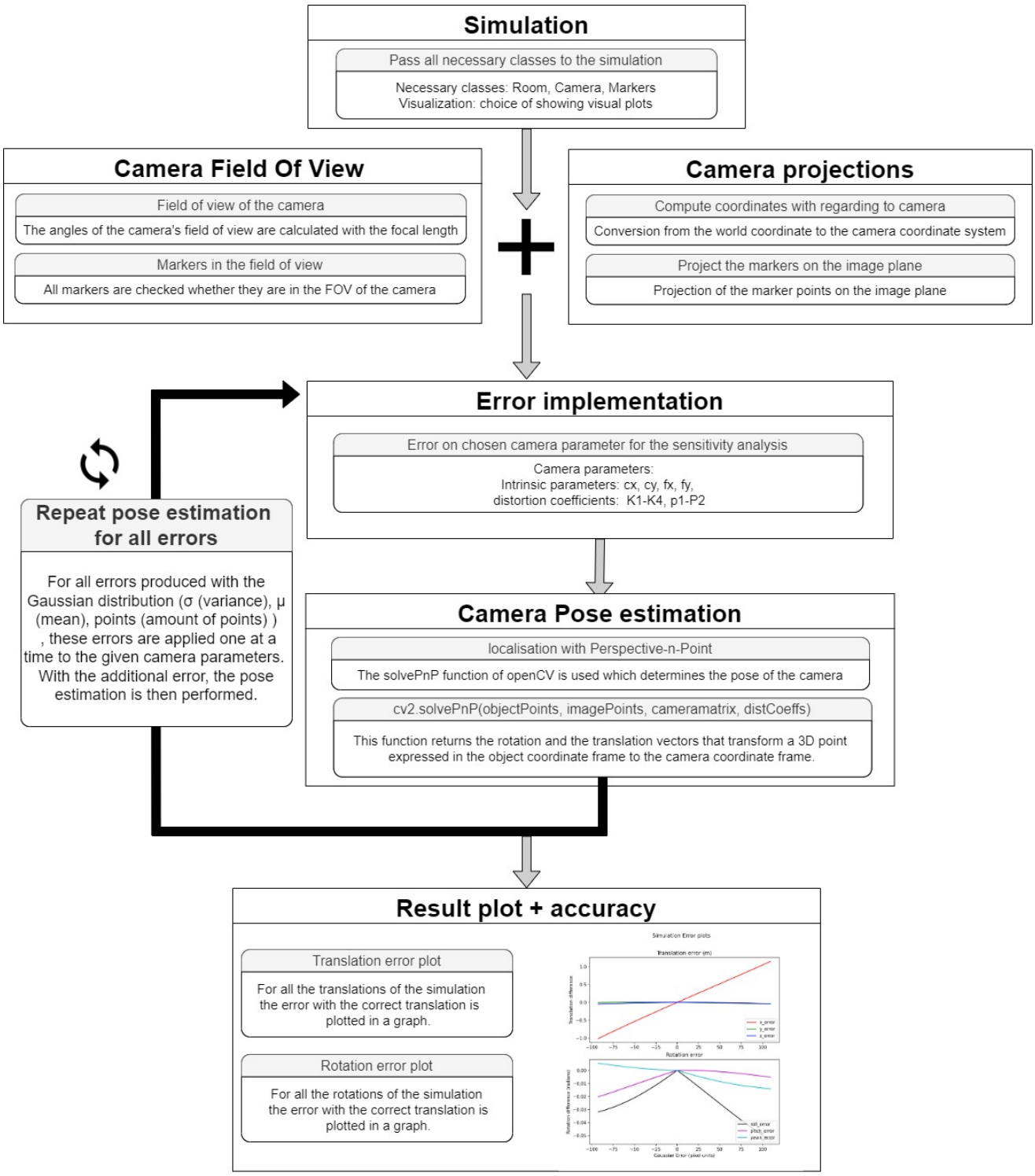
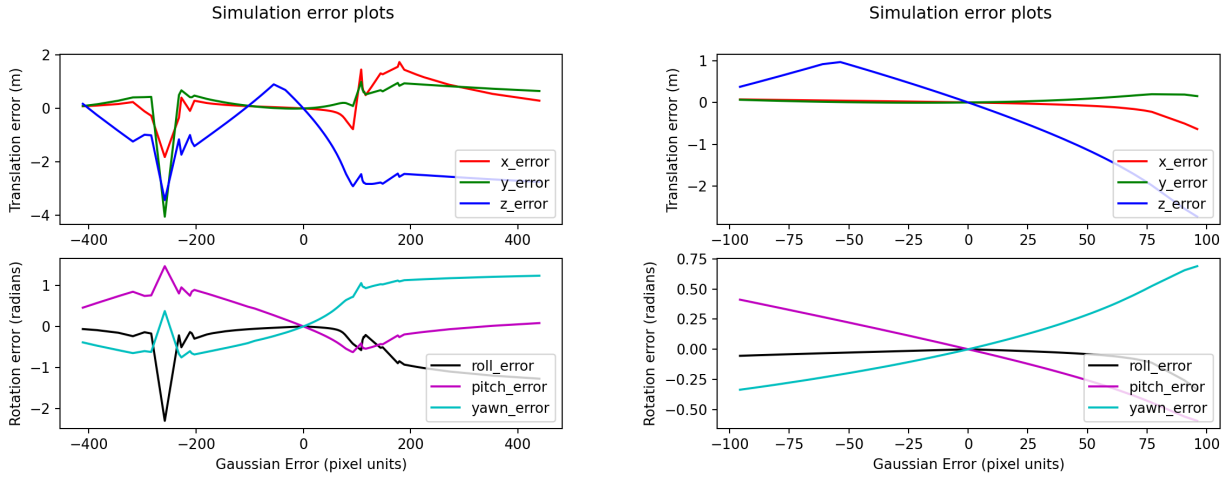


Figure 4.2: Processing pipeline of the error analysis simulation

4.1.4 Error range determination

To produce graphs in order to perform the error analysis, the range of the graph is an important criteria. To determine the optimal criteria, different ranges were first tested and plotted. It can be seen that if a range is taken that is twice as large as one of the focal lengths or as that of a principle point, the errors will look like those plotted in Figure 4.3(a). In this figure errors are added to the focal length f_x. If the error has the same value as the focal length and the used focal length for the camera becomes 0 resulting in erroneous calculations in the SolvePnP function for the pose estimate, this can be seen at the large peak around -250 pixel units. But because in reality there will never be errors of equal size on the calibration, the range of the graph is taken between -100 pixel units and +100 pixel units as shown in Figure 4.3(b). This value is about half of the focal length of the cameras used and is therefore a realistic value for the analysis.



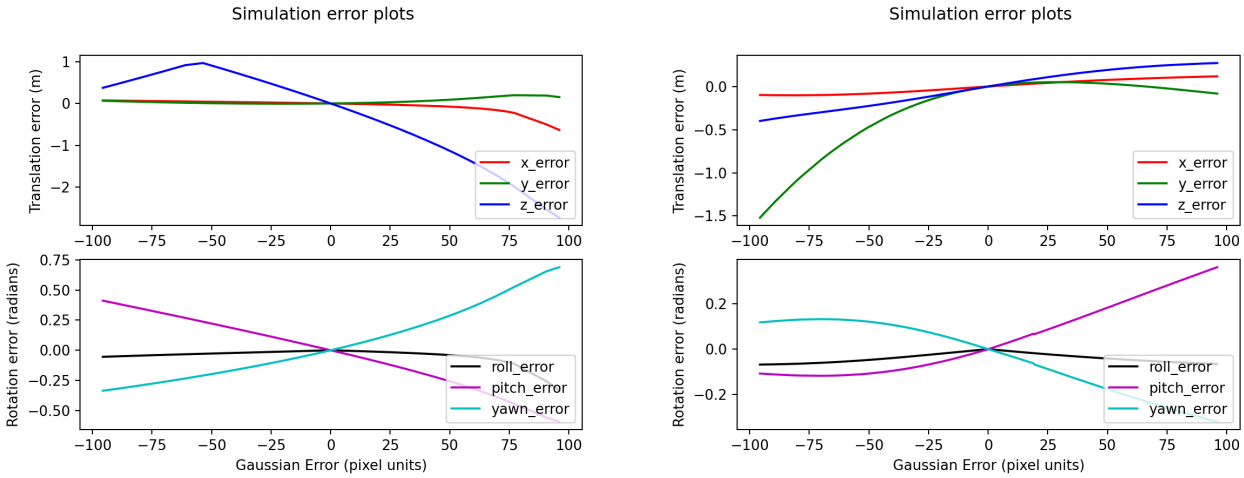
(a) (b)

Figure 4.3: Error on camera localization with (a) range [-400, 400], (b) range [-100, 100]

4.1.5 Intel RealSense Tracking Camera T265 Results

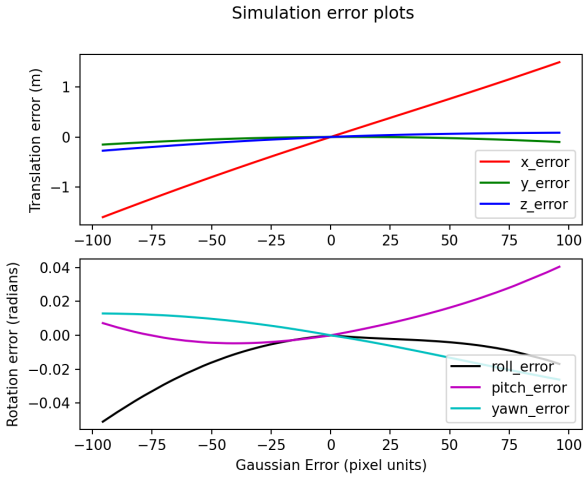
T265 Camera intrinsic parameters

The first camera for which an error analysis is done is the Intel RealSense Tracking Camera T265. For the intrinsic parameters, the following findings were obtained. First, errors on the focal length f_x were performed. Errors were added with a range of -100 and +100 pixel units to see their effects. The results plotted in Figure 4.4(a) show that the error on the f_x is going to affect mainly the estimation on the z coordinate of the camera and the pitch and yaw rotations. The larger the added error on the f_x the lower the z-coordinate will be estimated compared to the ground truth. If the error is negatively increased the z coordinate will be estimated higher until -50 pixel units where the error reaches a maximum error of 1 m too high. If the error becomes even more negative, the error on the estimation of the z-coordinate decreases. For the x and y error there are only small errors on the estimate up to an error of 75 pixel units for which the x coordinate is estimated 50 cm wrong. For the rotations, a clear trend can be found for the pitch and the yaw in Figure 4.4(a). As the implemented error increases, the pitch estimation becomes negatively larger and the yaw estimate becomes positively larger. As the added error becomes more negative, the pitch will be overestimated and the yaw underestimated. Mathematically, the yaw estimate will increase 0.005 radians corresponding 2.86° per added error and the pitch error will decrease by 0.005 radians per added error.

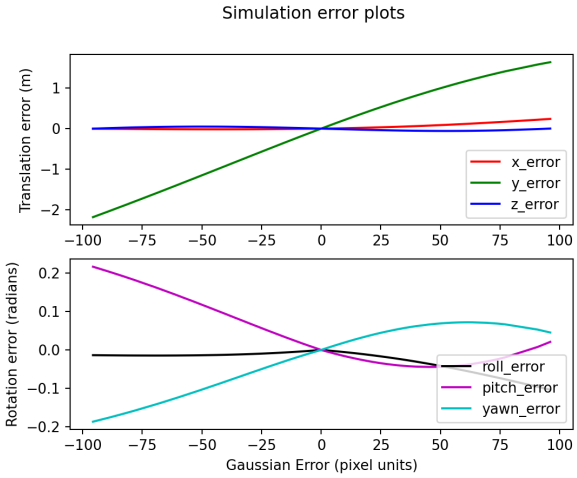


(a)

(b)



(c)



(d)

Figure 4.4: Error on camera localization for chosen range (a) with erroneous f_x, (b) with erroneous f_y, (c) with erroneous C_x, (d) with erroneous C_y

Figure 4.4(b) shows the effects of the errors on the focal length f_y plotted for the T265. The error affects mainly the estimated y coordinate and the pitch and yaw rotations. Especially when a negative error is added, the y coordinate estimation will become much lower, the z coordinate will also experience a negative estimation. For positive errors there is a slower increase for the added errors up to 50 cm. The rotations are estimated incorrectly for positively added errors on f_y. The pitch error will increase to 0.4 radians for an error of 100 pixel units. This corresponds to an error of 23°, so the pitch error increases around 0.23° per error. If the error is less than 0 there will be a maximum error of -0.1 radians on the pitch which corresponds to 5°. For the yaw error it is exactly the opposite of the pitch. For the roll estimate there are small negative estimates as the error increases both positive and negative.
The estimation difference by implementing errors on the principle point c_x is shown in Figure 4.4(c), for the translation this will mainly affect the x coordinate estimate. The estimate increases by 2cm per added pixel unit error on c_x. The errors on c_x only have a small influence on the estimate of all rotations. The largest error is 0.075 radians which corresponds to 4.3°. Per added error there will only be an error of 0.043° which is a small difference. When implementing errors on c_y, as shown in Figure 4.4(d), this will mostly affect the y estimate of the camera. The coordinate will be falsely estimated by 2cm per added error. On rotations, this will have a small effect on the pitch and yaw estimation. Per added error this will cause a misestimation of 0.11 degrees.

T265 Camera distortion coefficients

For the second part of the error analysis, errors are applied to the distortion parameters of the T265 camera. In Figure 4.5(a), the effects of the errors on K_1 are shown. The y estimate acquires the greatest change if errors are implemented. For each error of 0.01 that is added to the distortion factor, the estimate will deviate 3cm from the actual value. The deviation on the x and z estimate is minimal. The pitch and yaw error of 0.01 on K_1, will result in an incorrect estimate of 0.2 radians or 11 degrees. On the roll rotation, it will have nearly no effect.

When looking at the influence of errors on the distortion factor K_2 shown in Figure 4.5(b), an error will affect the y and z estimation of the camera. For the y estimate it is identically the same error as for K_1, a difference of 3 cm per error of 0.01 on the distortion factor. The effect on the estimate of the rotations when changing K_2, shows that it has the same effect as for K_1 but the estimation differences are bigger for the same errors. Figure 4.5(c) shows that with changes in P_1, this is going to produce incorrect estimates for all translations and rotations. Figure 4.5(d) shows that changes on P_2 lead to incorrect estimates for the x and z coordinate and for the roll and pitch rotation.

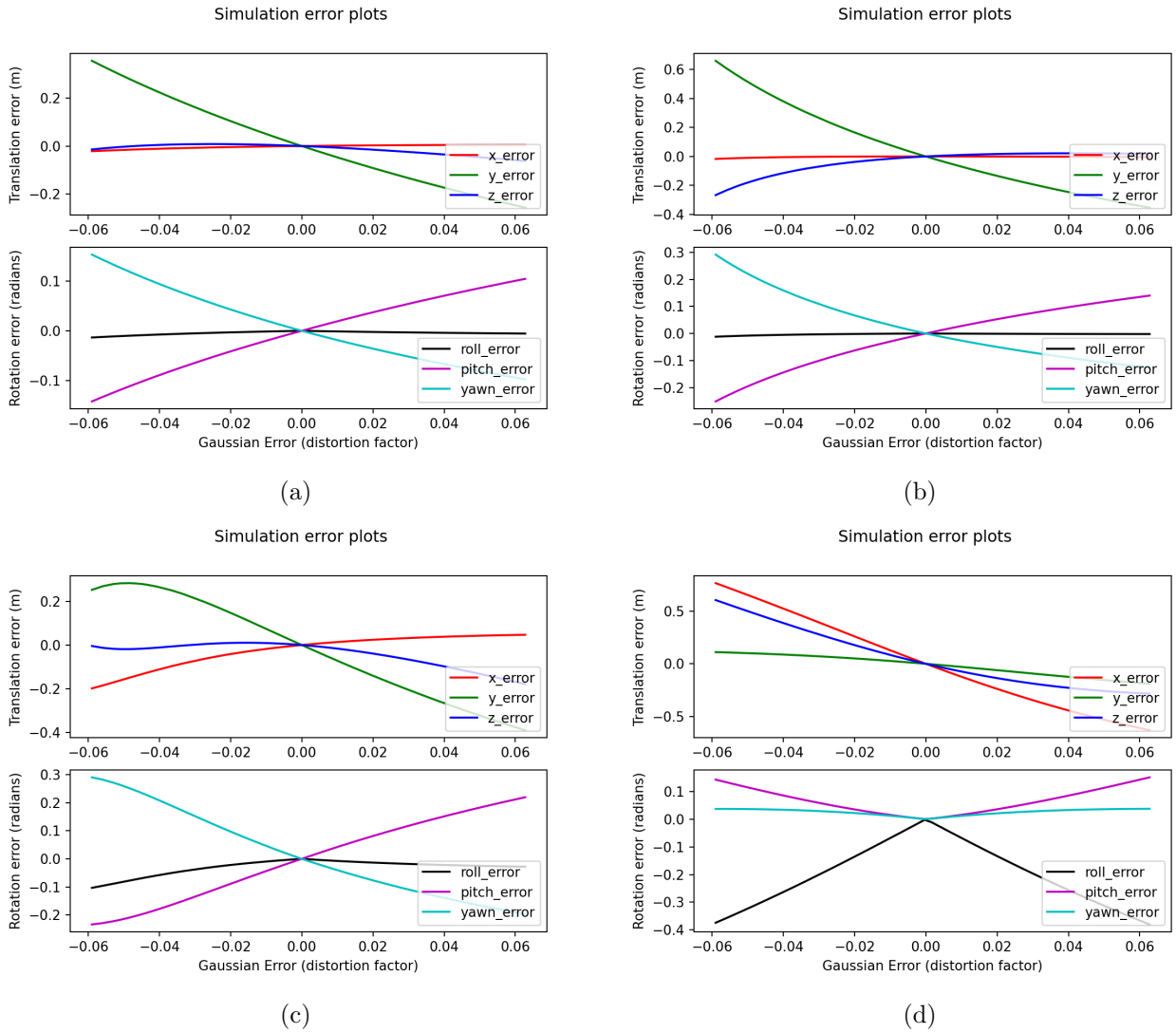


Figure 4.5: Error on camera localization for chosen range of distortion coefficients (a) with erroneous K_1, (b) with erroneous K_2, (c) with erroneous P_1, (d) with erroneous P_2

4.1.6 Intel Depth Camera D435 results

D435 Camera intrinsic parameters

All of the following error analysis are done on the Intel Depth Camera D435.The influence of the errors on f_x and f_y are plotted in Figure 4.7(a) and Figure 4.7(b). Remarkably, the results are very similar to the previous results of the Intel RealSense Tracking Camera T265. The error analysis on f_x shows that the misestimates occur mainly on the z-, pitch- and gap-estimates. When changing f_y, this results in wrong estimations on the y, pitch- and gap-rotation. For the errors on the principle points there is a different outcome compared with the previous camera. For the errors at the principle point c_x there is a difference in estimation done mainly for the x and z coordinate. In case of changing c_y this is most noticeable at the y and x estimate. On the rotations, the change of the principal point gives only small errors on the estimate, namely per pixel unit error 0.001 radians or 0.06° on c_x and 0.002 radians or 0.12° on c_y.

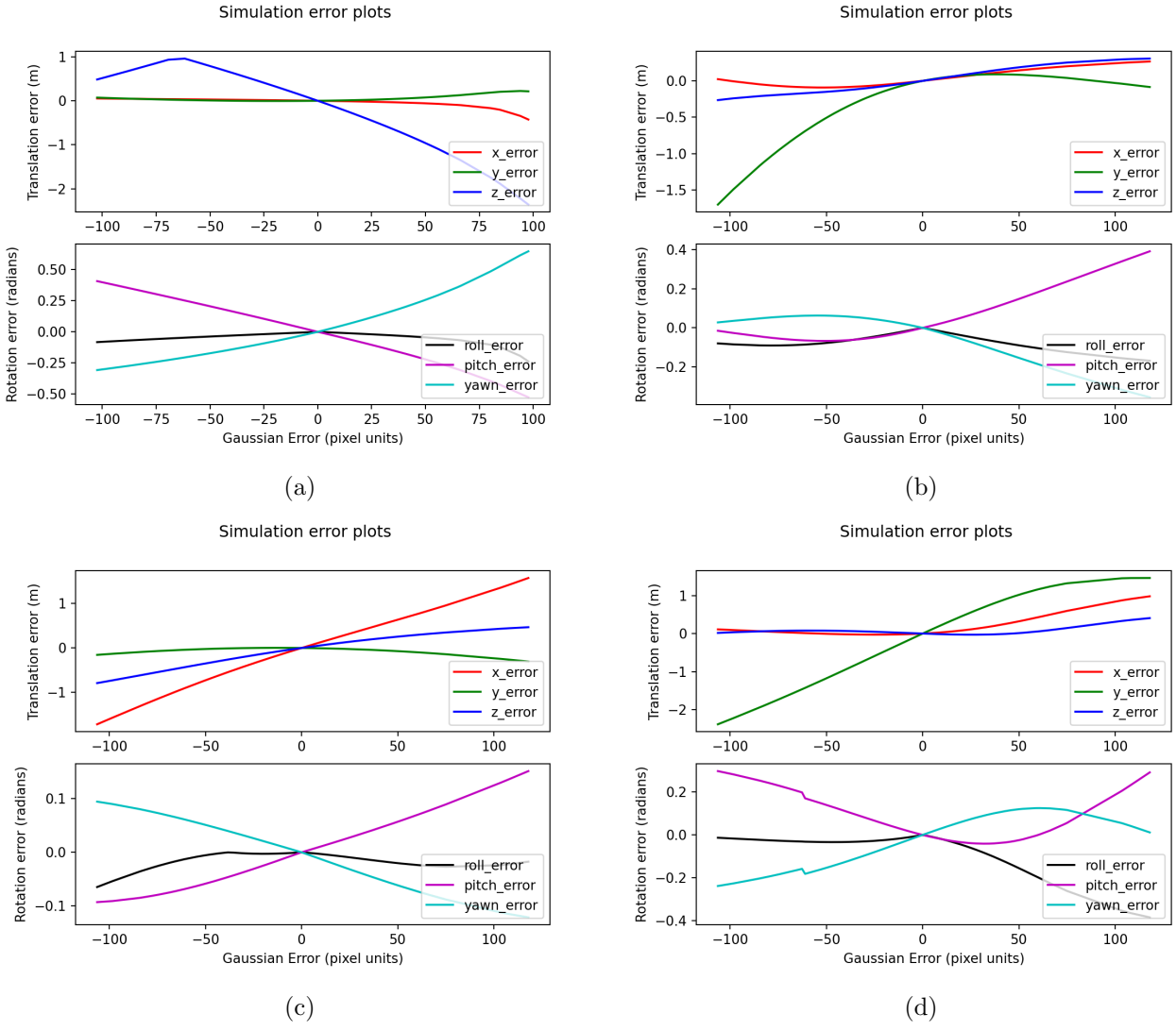


Figure 4.6: Error on camera localization for chosen range (a) with erroneous f_x, (b) with erroneous f_y, (c) with erroneous C_x, (d) with erroneous C_y

D435 camera distortion coefficients

The effects when changing the distortion parameters of the Intel Depth Camera D435, have a different outcome compared to the Intel RealSense Tracking Camera T265. When changing the K_1 coefficient this will produce an error in the estimation of the y-, z-coordinate and for all the rotations of the camera. When adjusting the distortion factor K_2 this is going to impact the y estimate. The x and z estimate will only be affected if K_2 is taken smaller. For all rotations there will be a small estimation error when changing K_2.

When the tangential distortion P_1 is adjusted, it will affect all translations. The error on the estimate does not have a linear slope. In contrast, when adjustments are made on tangential distortion P_2 there is a clear linear slope for the error on the estimate of the translations. The estimate on the x coordinate experiences the largest error, an difference of 12.5cm per 0.01 additional distortion factor. An erroneous estimate is made for all rotations if P_1 is adjusted. A difference of 0.06 on the distortion factor gives a misestimate of 0.2 radians, which is equal to 11°. When changing P_2, this has a minor effect on the yaw estimate, but on the pitch a larger error of 0.2 radians for a change of 0.06 of the distortion factor and twice this value on the roll estimate.

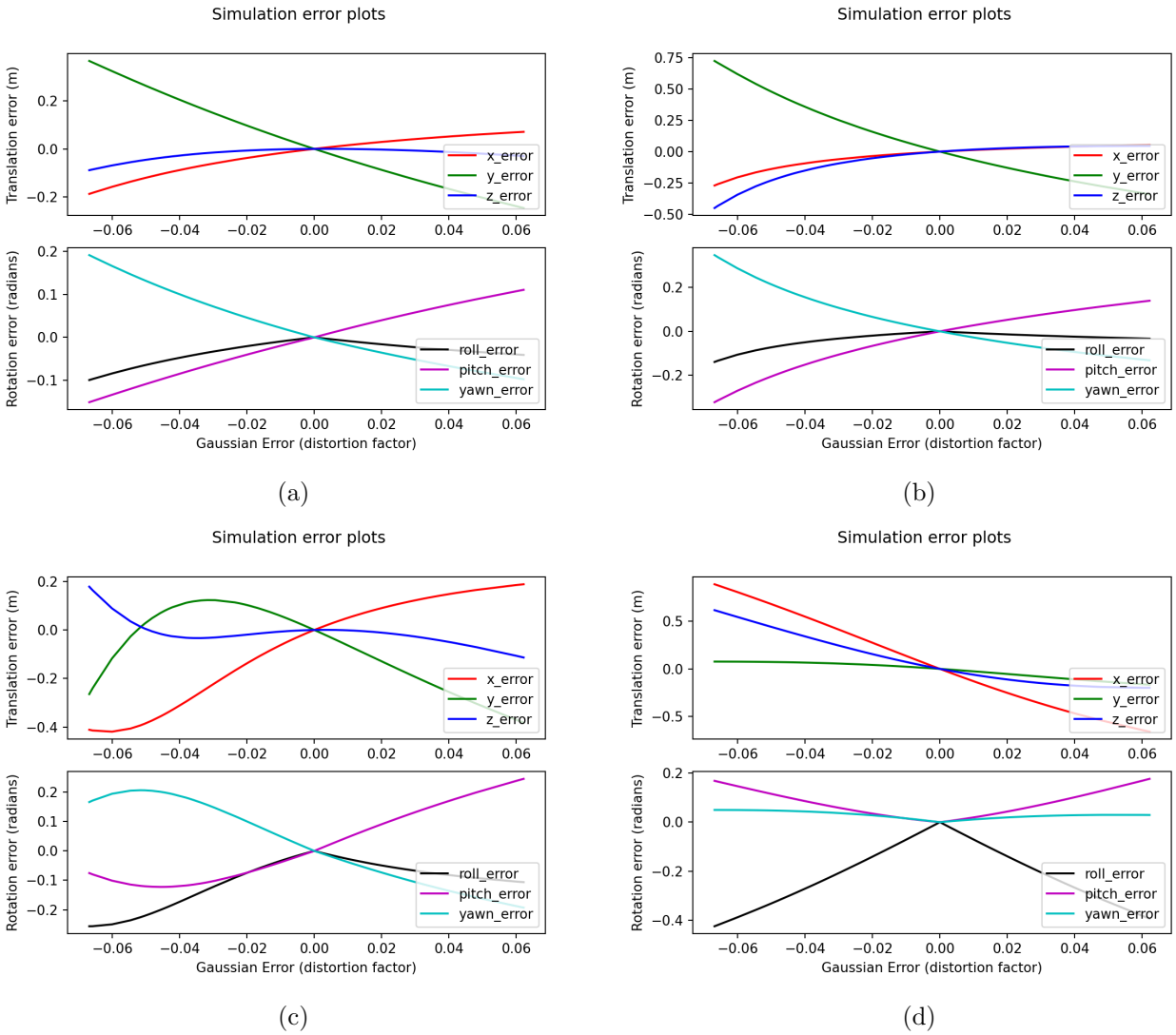


Figure 4.7: Error on camera localization for chosen range of distortion coefficients (a) with erroneous K_1, (b) with erroneous K_2, (c) with erroneous P_1, (d) with erroneous P_2

4.2 Gazebo simulation results

The Gazebo simulation is used to analyze the influence of the marker size, with respect to the distance to the camera, on the localization accuracy. The accuracy including various calibration errors is also implemented by providing erroneous camera info to the ArUco-detector. The camera will be represented by a virtual camera, and will use the specifications listed in Table 4.1 for the Intel D435. The virtual camera in combination with the other modules listed in Section 3.2 can perform a localization and the error on the true pose will be plotted and discussed.

4.2.1 Simulation setup

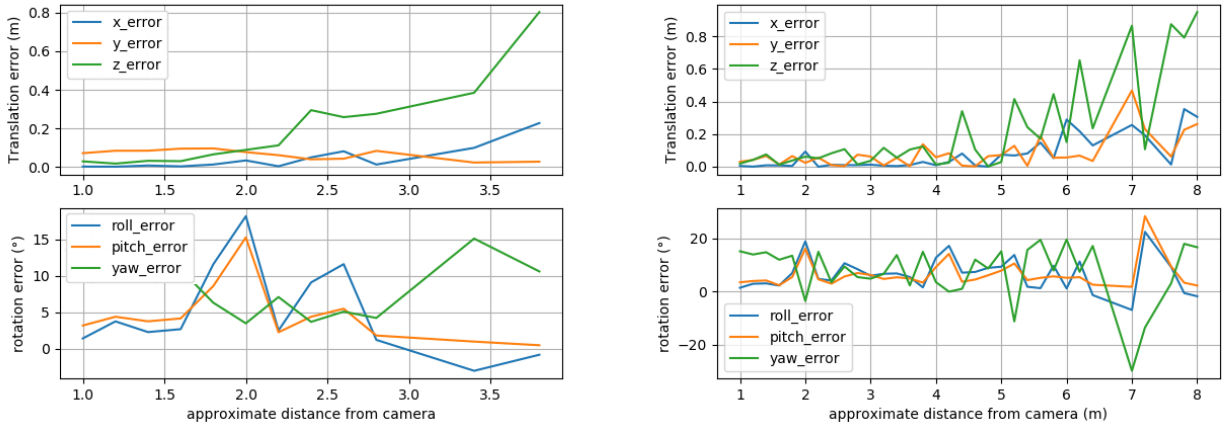
To produce following results a sequence of tests is performed using four markers at various positions in the camera's FOV. Outliers are possible because not all markers are always detected, occasionally there may be a large deviation on a specific marker. This introduces a significant deviation on the average error which is plotted in the resulting graphs. The first section of results implements the various marker sizes at realistically chosen distances. The second section introduces a faulty camera calibration with static markers, and the error on the pose estimation.

4.2.2 Intel RealSense Tracking Camera T265 Results

The camera will be simulated to test its accuracy to provide the images for a pose estimation. This will be conducted at various ranges, and this is a camera with a FOV of 87° so it can perform well at short range, as well as medium range depending on the marker size. The second section can use the theoretical analysis as a foundation for the expected localization errors at various implemented calibration errors. Using previous results, a range of errors can be deduced and implemented in the more realistic Gazebo model.

Distance and size simulation

The first analysis for the Gazebo environment is visualized in Figure 4.8, this shows the error on the pose of the camera for (a) markers with a size of 9 cm in a range of 1-4 m, (b) markers with a size of 18 cm in a detection range of 1-8 m. During this simulation four markers were placed in the simulation x- and y-coordinates at a distance fitting in the range.

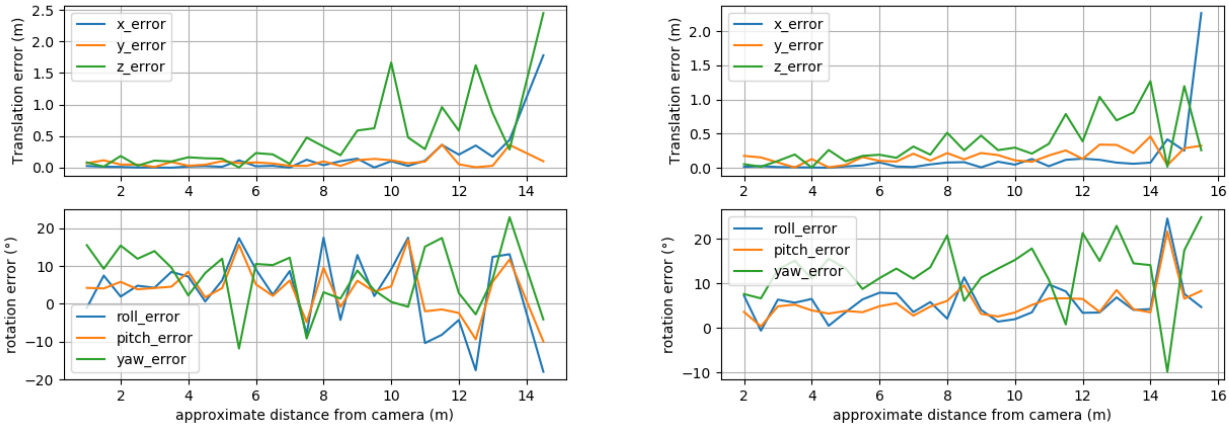


(a) (b)

Figure 4.8: Error on localization (a) Marker size 0.09 m (b) Marker size 0.18 m

Figure 4.8 visualizes the steady decrease of accuracy when increasing the distance between camera and markers. As the markers become smaller in the camera image, it becomes harder to determine the exact orientation. The size of the detected markers becomes harder to determine as there are less pixels that cover the side of the marker, so an estimate of the distance to the marker is less accurate. For 4.8 (a) the localization accuracy falls off heavily after the 2.25 metre mark, for (b) this happens at around 5 metres, this is a logical consequence as the marker in (b) is double the size. In (b) for markers at a distance of maximum 4 metres, a stable trend can be observed. Hereafter, markers are sometimes perceived incorrectly which leads to irregular spikes in the localization accuracy. The rotation on the other hand is very inconsistent after the first 1.5 metres, with an average deviation of 0°-20° per rotation angle, this is due to the low accuracy of the rotation estimate for the ArUco-markers.

The next analysis is conducted (a) for markers with a size of 36 cm in a range of 1-15 metres and (b) markers of 45 cm in a range of 2-15 metres, this is visualized in Figure 4.9.



(a) (b)

Figure 4.9: Error on localization (a) Marker size 0.36 m (b) Marker size 0.45 m

As illustrated in Figure 4.9 an accurate camera-location can be estimated for a maximum distance of around 8 metres and 10-11 metres in (a) and (b) respectively. For the rotation an unstable pattern is uncovered similar to the pattern in Figure 4.8.

For the chosen camera, an accurate pose can be determined at various distances, depending on the size of the markers. This is a logical conclusion, as the marker is visible in more pixels when using a bigger size. In reality, the accuracy could deteriorate faster or slower at the estimated distance to the camera, this can be tested using a real camera. But the simulation gives a realistic estimate of the detection distance.

Calibration error implementation

The camera calibration error is implemented on the various factors of the camera matrix, as well as the distortion factors K1, K2, P1 and P2. This analysis is performed using a single marker in the camera image. An overview of the results for the erroneous camera matrix is visualized in Figure 4.10.

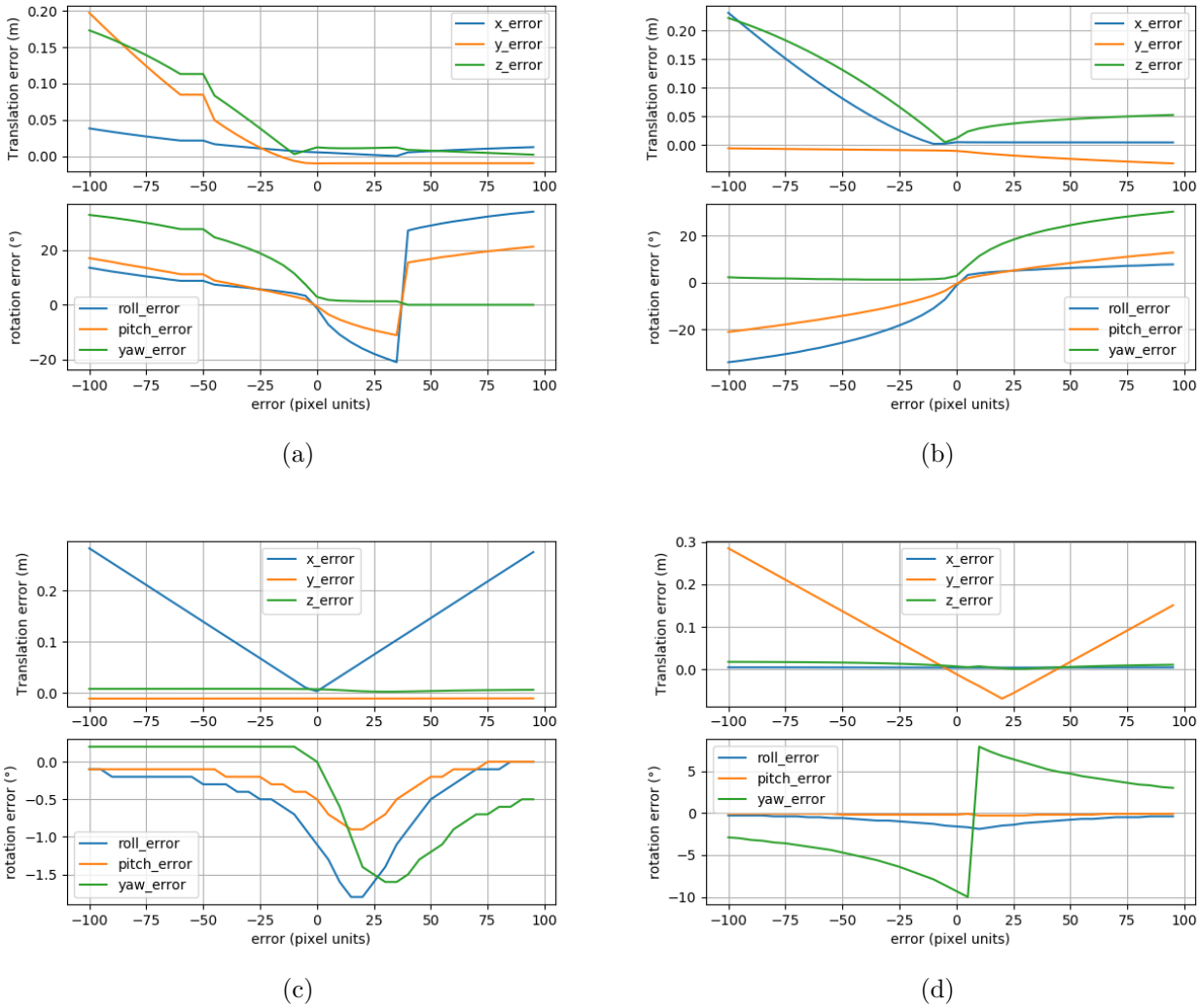
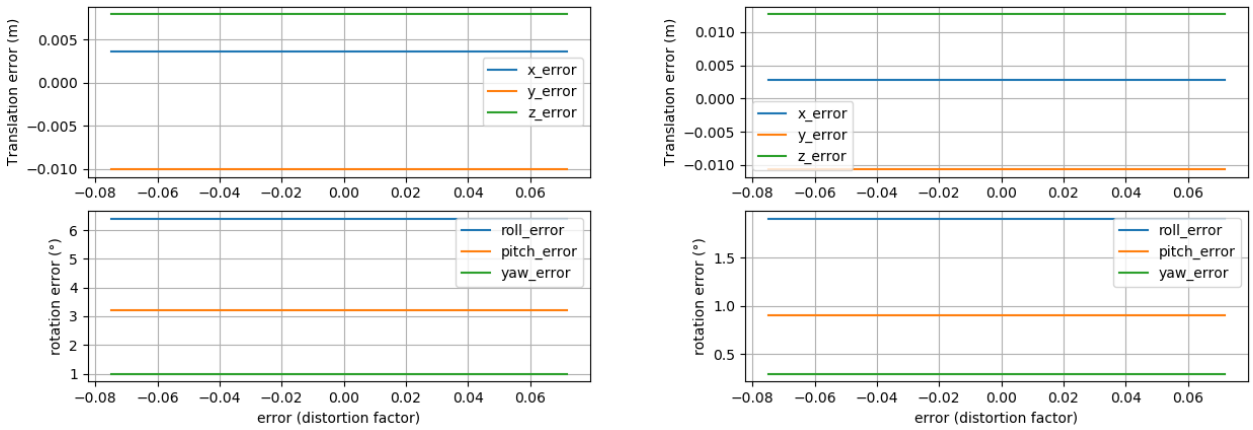


Figure 4.10: Error on localization (a) with erroneous f_x, (b) with erroneous f_y, (c) with erroneous C_x, (d) with erroneous C_y

From Figure 4.10(a), it can be deduced that an error on f_x when provided to the ArUco-marker detector yields a significant error on the camera pose estimate for values lower than the actual f_x-value. The error on the translation becomes less significant when f_x is replaced by a bigger than actual value, while the rotation estimate exhibits an increasing unstable pattern. For (b) a similar pattern can be recognized for the translation, but it displayed an increase in translation error for bigger than actual values of f_y. The rotation on the other hand follows a smooth trajectory from a negative error to a positive error on the roll, pitch and yaw angles. Part (c) displays an error on the optical center on the x-axis. For the translation error only the x-value is affected. The absolute value of the translation values is plotted, so normally the x-value would be a continuous line. The rotation angles already show a small static deviation, and follow an unstable pattern

during the analysis. However the deviation is very small and can be neglected. The last subfigure (d) visualizes an error on the optical center for the y-axis. Here a similar pattern pattern as the previous image can be detected, but now for the y-value. The yaw rotation shows a mirrored pattern around the center-point, with a significant angular deviation. The roll value is affected slightly, while the pitch angle stays consistent during the entire simulation.

Following the erroneous camera matrix, the various distortion coefficients are analyzed, and an overview is displayed in Figure 4.11.



(a) (b)

Figure 4.11: Error on localization (a) with erroneous K1, (b) with erroneous P1

In Figure 4.11 can be seen that the localization accuracy is not affected by the camera distortion factors and error on these factors, which seems very unlikely. The other explanation is that the AruCo-marker detector does not account for these factors in a correct manner. Only K1 and P1 were visualized as K2 and P2 yielded similar results.

4.3 Comparison between environments

The proposed simulators can focus on different aspects of the Localization and Mapping equation. The theoretical simulator in Python can directly implement various camera models with various parameters that can be implemented to perform a wide variety of tests. It still has its limitations in real world factors like natural noise, lighting and depth of field effect. The Gazebo simulation in contrary uses a virtual environment to use a virtual camera in combination with the marker detector, this does include lighting and depth of field effect. The trade-off of this implementation is the combination and cooperation of the different modules. But the resulting analysis of both simulators yielded interesting results. An estimation of the camera localization error can be determined, when including various factors in a specified sensitivity analysis. a remarkable result is that the Gazebo simulator gives a smaller deviation on the estimation of the camera pose than the theoretical analysis. An important factor is the manner of determining the camera pose estimate. The theoretical analysis uses the solvePnP method, while the Gazebo simulator uses a marker detector to acquire a pose of a marker and then transforms this using the predefined map. The solvePnP method leans closer to the algorithms used in reality, as structure from motion algorithms also define points in the 3D space which can then be solved to find the camera location.

Chapter 5

Conclusion

In the framework of this masters thesis in cooperation with Intermodalics, a sensitivity analysis is performed on the various camera parameters and models have an impact on the localization algorithm. To perform the analysis, two simulators were proposed that are able to create a virtual parametric space where camera's can be simulated and observed. This enables full control over various aspects for the camera localization accuracy. The objective of these simulators is to simplify and control the environment and cameras during the analysis. In real world conditions, not all parameters can be controlled, and external factors can compromise the accuracy of the gained results.
Both simulators use a different approach in order to analyse the sensitivity of a camera. The Python simulator uses a theoretical approach, with a camera localization estimation based on the PnP-problem. This approach is more realistic, in the sense of using points for the localization. In a real environment, without markers, key-features need to be detected and these will also be represented by points. The simulator is able to account for deviations in the camera matrix and deformation coefficient when estimating the camera pose. This allowed the necessary experiments to be performed. The second simulator tries to implement an approach based on the virtual environment Gazebo, with the localization based on ArUco-markers of a fixed size. The markers can be detected and their pose can be estimated in the camera coordinate frame. The detected pose can then be transformed to calculate the camera pose. This simulator accounts for camera matrix deviations. By using a marker detector, markers of various sizes at various distances can be implemented. An estimation of the detection distance and accompanying pose accuracy can be analyzed as well.

Both simulators provide the desired results. When comparing the results of both simulators, a similar trend is observed with a variation in the size of the error on the localization. An increasing difference in the camera parameter for the pose estimation corresponds to an increasing error on the translations of the camera coordinates. The rotation can deviate in the comparison, as a small error in the marker detection can lead to a significant variation in the pose estimate. The analysis on the detection distance of the second simulator yielded interesting results. The analysis can be used to do an estimate of the detection range when trying to implement a Visual SLAM system using markers. The sensitivity analysis, shows that every deviation on the parameters of the camera matrix or distortion coefficients, has an influence on specific components of the localization accuracy. There is overlap between the effects of various parameters, this was evaluated in the results.

Bibliography

[1] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (F. Loizides and B. Schmidt, eds.), pp. 87 – 90, IOS Press, 2016.

[2] "Opencv: camera calibration and 3d reconstruction_2022," 2022.

[3] C. Express, "Fiducial markers (aruco) · clover," 2022.

[4] "Opencv: perspective-n-point (pnp) pose computation_2022," 2022.

[5] X. X. L. 2018, ""a review of solutions for perspective-n-point problem in camera pose estimation"," *Journal of Physicss: Conference Series*, vol. 1087, Sep. 2018.

[6] X. Gao, X. Hou, J. Tang, and H.-F. Cheng, "Complete solution classification for the perspective-three-point problem," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 25, pp. 930–943, 2003.

[7] V. Lepetit, F. Moreno-Noguer, and P. Fua, "Epnp: An accurate o(n) solution to the pnp problem," *International Journal of Computer Vision*, vol. 81, p. 155–166, Jul 2008.

[8] G. Terzakis and M. Lourakis, "A consistently fast and globally optimal solution to the perspective-n-point problem," in *Computer Vision – ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part I*, (Berlin, Heidelberg), p. 478–494, Springer-Verlag, 2020.

[9] M. A. Fischler and R. C. Bolles, "Random sample consensus," *Communications of the ACM*, vol. 24, pp. 381–395, jun 1981.

[10] W. Contributors, "Rodrigues' rotation formula," Apr 2022.

[11] Scratchapixel, "3d viewing: the pinhole camera model," Dec 2014.

[12] A. engineering, "the pinhole camera," Feb 2014.

[13] Legend, "Image sampling and quantization process.," 2013.

[14] cambridge university, "understanding digital camera sensors," 2020.

[15] Opencv.org, "Image thresholding methods."

[16] J. Yousefi, "Image binarization using otsu thresholding algorithm," *Ontario, Canada: University of Guelph*, 2011.

[17] T. Pattnaik and P. Kanungo, "Gmm based adaptive thresholding for uneven lighting image binarization," *Journal of Signal Processing Systems*, vol. 93, p. 1253–1270, Nov 2021.

[18] H. Kato and M. Billinghurst, "Marker tracking and hmd calibration for a video-based augmented reality conferencing system," in *Proceedings 2nd IEEE and ACM International Workshop on Augmented Reality (IWAR'99)*, pp. 85–94, 1999.

[19] M. Kalaitzakis, B. Cain, S. Carroll, A. Ambrosi, C. Whitehead, and N. Vitzilaios, "Fiducial markers for pose estimation," *Journal of Intelligent & Robotic Systems*, vol. 101, Mar 2021.

[20] E. Olson, "Apriltag: A robust and flexible visual fiducial system," *2011 IEEE International Conference on Robotics and Automation*, May 2011.

[21] A. Mutka, D. Miklic, I. Draganjac, and S. Bogdan, "A low cost vision based localization system using fiducial markers," *IFAC Proceedings Volumes*, vol. 41, no. 2, p. 9528–9533, 2008.

[22] F. Bergamasco, A. Albarelli, E. Rodolà, and A. Torsello, "Rune-tag: A high accuracy fiducial marker with strong occlusion resilience," in *CVPR 2011*, pp. 113–120, 2011.

[23] Z. Tian, C. J. Carver, Q. Shao, M. Roznere, A. Q. Li, and X. Zhou, "Polartag: Invisible data with light polarization," in *Proceedings of the 21st International Workshop on Mobile Computing Systems and Applications*, HotMobile '20, (New York, NY, USA), p. 74–79, Association for Computing Machinery, 2020.

[24] D. H. Lee, S. S. Lee, H. H. Kang, and C. K. Ahn, "Camera position estimation for uavs using solvepnp with kalman filter," in *2018 1st IEEE International Conference on Hot Information-Centric Networking (HotICN)*, pp. 250–251, 2018.

[25] A. Kriegler and W. Wöber, "Vision-based docking of a mobile robot." EasyChair Preprint no. 5092, EasyChair, 2021.

[26] R. A. Boby, "Hand-eye calibration using a single image and robotic picking up using images lacking in contrast," *2020 International Conference Nonlinearity, Information and Robotics (NIR)*, Dec 2020.

[27] F. I. Pereira, G. Ilha, J. Luft, M. Negreiros, and A. Susin, "Monocular visual odometry with cyclic estimation," *2017 30th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, Oct 2017.

[28] R. I. Hartley and P. Sturm, "Triangulation," *Computer Vision and Image Understanding*, vol. 68, no. 2, pp. 146–157, 1997.

[29] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *Journal of the Society for Industrial and Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963.

[30] A. Kendall, M. Grimes, and R. Cipolla, "Posenet: A convolutional network for real-time 6-dof camera relocalization," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[31] D. Zhang, Y. Nomura, and S. Fujii, "Error analysis and optimization of camera calibration," in *Proceedings IROS '91:IEEE/RSJ International Workshop on Intelligent Robots and Systems '91*, pp. 292–296 vol.1, 1991.

[32] "projectpoints - projects 3d points to an image plane.," 2022.

[33] OpenCV, "Detection of aruco markers," 2014.

[34] I. R. Depth and T. Cameras, "Intel tracking camera t265_2022," Mar 2022.

[35] I. R. Depth and T. Cameras, "Intel depth camera d435_2022," May 2022.

[36] IntelRealSense, "T265 distortion matrix · issue 3497 · intelrealsense/librealsense," Mar 2019.

[37] IntelRealSense "D435" intrinsic parameters. - ros answers: open source qa forum2020,″ 2020.