Master's thesis

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

Henriette Sommerseth Mathisen

Positioning a camera underwater

Master's thesis in Cybernetics and Robotics
Supervisor: Annette Stahl
June 2022

Norwegian University of
Science and Technology

Henriette Sommerseth Mathisen

Positioning a camera underwater

Master’s thesis in Cybernetics and Robotics
Supervisor: Annette Stahl
June 2022

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

Norwegian University of
Science and Technology

Abstract

This thesis will provide an overview of how the challenge of underwater positioning and orientation of a camera has been addressed, utilizing a low-cost and simple system. To address the challenges of positioning and orientating a camera underwater, a Raspberry Pi 4, a camera, and an Inertial Measurement Unit (IMU) were employed. The Extended Kalman Filter (EKF) was then used to combine measurement data from the IMU and AprilTag.

Before the thesis attempted to tackle the problem using EKF, IMU, and AprilTag, the thesis attempted to solve the problem with Simultaneous Localization and Mapping (SLAM) and IMU. Installing and implementing SLAM on a Raspberry Pi caused several difficulties.

The IMU was used to determine the camera's position and orientation. The test indicated that the IMU can provide accurate orientation measurements, but that it would drift after some time. The AprilTag was implemented to detect multiple tags and provide the camera's position in relation to the tags. These tests demonstrate that the AprilTag is capable of providing accurate readings when detecting a tag.

Sammendrag

Denne masteroppgaven gir en oversikt over hvordan utfordringene med posisjonering og orientering under vann var undersøkt, ved å benytte et billig og enkelt system. Utfordringene med å posisjonere og orientere et kamera under vann ble undersøkt med å benytte en Raspberry Pi, et kamera og en Inertial Measurement Unit (IMU). Det ble benyttet en Extended Kalman Filter (EKF) hvor posisjonerings dataen fra IMU'en og AprilTag ble slått sammen.

I begynnelsen av masteroppgaven var problemstillingen forsøkt løst med å benytte Simultaneous Localization and Mapping (SLAM) og en IMU. Det oppsto flere probelmer med installasjon og implementasjon av SLAM, ved å benytte en Raspberry Pi.

IMU'en var brukt til å definere posisjon og orientering av kameraet. Tester som ble utført indikerer at IMU'en klare å levere presis orienterings målinger, men der er utfordringer med drifting etter at IMU'en har vært operativ en stund. AprilTag er tagger som brukes for å lese av en binær kode. AprilTag var implementert inn i systemet for å kunne detektere flere tagger og leveren kameraet sin posisjon relativt til plasseringene til taggene. Igjennom tester ble det funnet ut at AprilTag gir veldig presise posisjoneringer når systemet detekterer en tagg.

Preface

This master thesis was written for the Cybernetics and Robotics program at NTNU. The master thesis was assigned by the company Zebop AS. The master thesis will be based on the pre-project and will include a description of how the master thesis was completed. The author of the master thesis has received guidance from both the supervisor and employees of Zebop who have made themselves available.

I would like to thank Zebop AS employees for the project, their good guidance, and the resources supplied to the project. A special thanks to Annette Stahl at NTNU for her guidance and advice during the master thesis.

Table of Contents

Abstract **i**

Sammendrag **ii**

Preface **iii**

List of Figures **vii**

List of Code **ix**

Abbreviations **x**

1 Introduction **1**

1.1 Motivation 1

1.2 Aim of Study 2

1.3 Challenges 2

1.4 Research questions 3

1.5 Contributions 3

1.6 Outline of the thesis 4

2 Related work **5**

3 Background and Methods **8**

3.1 Raspberry Pi High Quality Camera 8

3.2 Inertial measurement unit 10

3.2.1 Adafruit IMU BNO055 12
3.2.2 Calibration of IMU 13
3.3 AprilTag 14
3.3.1 Calibration of camera 17
3.4 Kalman filter 20
3.4.1 Extended Kalman filter 22
3.5 Simultaneous Localization and Mapping 23
3.5.1 ORB-SLAM3 and pySLAM 24

4 Experiments and Results 25
4.1 Simultaneous Localization and Mapping 25
4.1.1 ORB-SLAM3 25
4.1.2 PySLAM 28
4.2 General problems 29
4.2.1 Memory problems 29
4.2.2 Second raspberry Pi 29
4.2.3 IMU problems 29
4.2.4 Ubuntu 30
4.2.5 Gstreamer problems 30
4.2.6 HQ camera and new test model 31
4.3 Inertial measurement unit 32
4.4 AprilTag 36
4.4.1 Calibration 36
4.4.2 Implementation 40
4.5 Extended Kalman filter 44

5 Conclusion and Further work 48
5.1 Further work 49

References 50

Appendix A Installation guide ORB-SLAM3 54

Appendix B Installation guide IMU 64

Appendix C Installation guide Raspberry Pi HQ camera 74

List of Figures

3.1 Raspberry Pi HQ camera 9
3.2 Raspberry Pi HQ camera assembly 9
3.3 Raspberry Pi HQ camera with lens 10
3.4 9-DOF IMU sensor 11
3.5 IMU BNO055 from Adafruit 12
3.6 WebGL example web page are used to calibrate the IMU 13
3.7 AprilTags 15
3.8 AprilTag size 16
3.9 Different camera positions relative to a tag 17
3.10 Distorted picture from the camera 18
3.11 Checkerboard pattern 19
3.12 Kalman filter process diagram 21
3.13 SLAM systems architecture 24

4.1 Changes on CMakeList.txt 26
4.2 EuRoc test of ORB-SLAM3 with camera and IMU 26
4.3 EuRoc test of ORB-SLAM3 with stereo 27
4.4 EuRoc test plot 27
4.5 Modified /boot/firmware/config.txt file 30
4.6 The test model with the raspberry Pi HQ camera installed 32
4.7 The connection of the IMU and camera to the raspberry Pi 32
4.8 IMU sensor readings output 35

4.9 IMU sensor readings after recalibration 35
4.10 Detected checkerboard pattern 38
4.11 Undistorted picture after calibration 40
4.12 AprilTags detected with camera 43
4.13 AprilTags detected pose estimation output 44

Code

1	Python code for importing BNO055 module and connecting the sensor	32
2	Python code for loading the calibration file	33
3	Python code for reading the BNO055 sensor	34
4	Python code for transforming BNO055 readings to acceleration, velocity and angular velocity	34
5	Python code for taking a picture with the camera	36
6	Python code to identify and draw checkerboard corners in all pictures	37
7	Python code for calibrating the camera	38
8	Python code for undistortion	39
9	Python code for importing AprilTag module and configuring detector options	41
10	Python code preprocessing the input video	41
11	Python code for looping through the AprilTag detection results	42
12	Python code for positioning AprilTag	42
13	Python code for covariance noise matrices	45
14	Python code for the prediction step	45
15	Python code for the measurement update step	46
16	Python code for wraping angle to (-pi,pi] range	46

Abbreviations

GPS	Global Positioning System
GNC	Guidance, Navigation, and Control
INS	Inertial Navigation System
AUV	Autonomous Underwater Vehicle
IMU	Inertial Measurement Unit
ORB	Oriented FAST and Rotated BRIEF
SLAM	Simultaneous Localization and Mapping
MAV	Micro Aerial Vehicle
DOF	Degrees of freedom
VO	Visual Odometry
g2o	General Graph Optimization
py	Python
OpenCV	Open Source Computer Vision Library
GPU	Graphics Processing Unit
CPU	Central Processing Unit
ARM	Advanced RISC Machines
AMD	Advanced Micro Devices
MAP	Maximum a posteriori
RGB-D	Red, Green and blue - Depth sensing
GB	Gigabyte
RTSP	Real Time Streaming Protocol
HQ	High Quality

IR Infrared

HD High-definition

MP Megapixel

QR Quick Response

OS Operating System

ID Identification

EKF Extended Kalman Filter

KF Kalman Filter

GPIO General Purpose Input/Output

UART Universal Asynchronous Receiver/Transmitter

PoE Power over Ethernet

Chapter 1

Introduction

This thesis will describe how the challenge of positioning and orientating a camera underwater was addressed using low-cost components. The problem was attempted to be solved by merging IMU readings and AprilTag measurements in an extended Kalman filter utilizing an IMU, a Raspberry Pi 4, and a low-cost Raspberry Pi high quality camera. This will also help to address the issues with the IMU drifting.

This chapter will provide an introduction to the thesis by discussing the motivations and aim of study, as well as the research questions and contributions of the thesis. The challenges that have been encountered during the project will also be briefly explained. The introductory chapter will conclude with a short outline of the thesis.

1.1 Motivation

Norway has a long history as a marine nation. Monitoring and intelligent monitoring are becoming more important in the maritime industries that operate not only in Norway but also globally. Camera systems have been increasingly popular in recent years, and their capabilities are continually expanding. Simultaneously, there is innovation and development of moving cameras, such as those used in autonomous vehicles.

Moving cameras provide a new challenge in terms of quickly orienting itself with the surroundings and understanding the position to carry out tasks. Surface-based positioning, such as mobile networks and Global Positioning System (GPS), do not operate underwater. There are several underwater navigation options on the market today, but they all have several commonalities: they are expensive, resource intensive to scale, require setup by authorized staff, and require additional cables and sensors to be placed out. The latter is difficult since more sensors and cabling on facilities increase the infrastructure's sensitivity to weather and wind.

The idea behind this thesis is that an accelerometer or IMU may be used to detect the cameras position and orientation underwater. This is a compact and inexpensive sensor that is utilized in a variety of industries. One acknowledged issue with IMUs is that they drift with time and hence cannot offer trustworthy data after a period of time. As a result, it is intended to use the camera's picture flow and a known geometry to recalibrate the IMU underwater at intervals sufficient to allow the IMU to be used as an operational tool for orientation and positioning.

1.2 Aim of Study

This thesis will concentrate on the development of a system that will address underwater positioning issues utilizing low-cost components. The technology will not be tested underwater due to timing restrictions. This is due to the fact that constructing a waterproof test model and being able to conduct tests underwater takes time and introduces additional challenges that must be addressed in addition to the challenges that have accrued over water.

1.3 Challenges

Throughout the thesis, various time-consuming challenges were encountered, and as a result, some tasks took longer to accomplish than anticipated. The SLAM-related issues required the most time. Oriented FAST and Rotated BRIEF (ORB)-SLAM3 and Python (py)SLAM were both tested, and both had some installation difficulties as well as some issues getting both systems to work properly. The majority of these issues arose since the system was operated on an Raspberry Pi, which is an Advanced RISC Machines (ARM)64 system, whereas the ORB-SLAM3 and pySLAM were designed for Advanced Micro Devices (AMD)64 systems. The thesis was adjusted so that it would be addressed using an IMU, AprilTag, and an extended Kalman filter instead of utilizing SLAM because all of the issues were time-consuming to solve and make operational.

There have also been several additional challenges that were time-consuming to resolve, such as memory issues, IMU booting problems, and Gstreamer issues. There were experiments conducted utilizing two Raspberry Pi's, one running Ubuntu 21.10 and the other Ubuntu 20.04, during the SLAM problems. This was done to exclude the possibility that the problem was caused by utilizing the wrong Ubuntu version. As a result, it was also necessary to prepare a second IMU for use and testing. The camera was changed from a SONY camera to a Raspberry Pi camera due to issues with Gstreamer. This eliminated issues with getting the camera and Raspberry Pi to communicate properly and allowed the camera to be directly connected to the Raspberry Pi. To be able to utilize the new camera on the test model and connect both the camera and the IMU to the Raspberry Pi at the same time, the test model needed to be modified. All of these problems were resolved, although it took time.

1.4 Research questions

This master's thesis will contribute to answering the following questions:

- *Research questions 1:* What are the challenges of positioning a camera underwater, and how can they be overcome?
- *Research questions 2:* Is it possible to solve the underwater position issues with a low-cost system?
- *Research questions 3:* Can the IMU drifting problems be overcome by combining camera and IMU using sensor fusion?

1.5 Contributions

The contributions for the master's thesis are:

- Contributed to the system integration and calibration of the inertial measurement unit. Has also contributed to testing the inertial measurement unit to ensure it is operational and properly calibrated.
- Contributed to the calibration of the camera and the integration of AprilTag into the system. Has also contributed in system testing to ensure that the AprilTag was operational and the camera was properly calibrated.
- Contributed to the system's implementation of an extended Kalman filter that employs the inertial measurement unit as input and the AprilTag as measurement. Has contributed to testing the extended Kalman filter with the inertial measurement unit as input, but the AprilTag is still not functioning as a filter measurement.

1.6 Outline of the thesis

Chapter 1 - Introduction: This chapter provides an introduction to the thesis, including the motivations, aim of study, challenges, research questions and objectives, and the contributions.

Chapter 2 - Related work: This chapter will provide a overview of previous research and articles related to this topic.

Chapter 3 - Background and Methods: This chapter will go through the thesis's background and methods, including details of the camera utilized, the inertial measurement unit, AprilTag, the Kalman filter, and Simultaneous Localization and Mapping.

Chapter 4 - Experiments and Results: This chapter contains the experiments and results from the thesis's work using SLAM, IMU, AprilTag, and the Extended Kalman Filter. It also gives a summary of some of the difficulties experienced during the thesis.

Chapter 5 - Conclusion and Further work: Finally, this chapter concludes the work conducted for this thesis. It also covers further work that can be performed subsequently.

Chapter 2

Related work

There are multiple studies on various techniques to address underwater positioning, with systems ranging from basic to complicated, low to high precision, and low to high cost. It is challenging to accomplish precise positioning using the conventional positioning approach for an underwater system due to its location and the complex and dynamic environment in which it operates. The paper "Survey of underwater robot positioning navigation" by Yinghao Wu, Xuxiang Ta, Ruichao Xiao, Yaoguang Wei, Dong An, and Daoliang Li, [1], studies and summarizes various standard underwater robot location and navigation approaches. It covers underwater localization technologies such as multisensor information fusion, underwater acoustic localization and navigation techniques, GPS buoy, underwater vision, SLAM, and coordinate localization and navigation of several underwater robots [1].

This study provides an excellent overview of many relatively mature and traditional underwater robot positioning and navigation methods. Although all of the approaches may fill the tasks of positioning and navigation, each has its unique set of features. Because this document evaluates and summarizes the advantages and disadvantages of each system, it is feasible to choose the best strategy for different environments. The most equivalent positioning approach to this study is monocular vision, which studies location based on feature matching. The positioning technique was used in this case to calculate location based on the size of the target in the image. This is a low-cost and basic structure, however it has certain issues, such as being unidentifiable for comparable images. It also considers the usage of multi-sensor fusion, where it implements IMU and the Kalman Filter (KF), however GPS systems are utilized to address the IMU's drifting difficulties. This complicates the system and is influenced more by the environment. In this project, not all of these systems are relevant to examine. The research papers examined in this report are systems that are similar to the system employed in this study, a low-cost, basic positioning system [1].

The research paper "Towards Micro Robot Hydrobatics: Vision-based Guidance, Navigation, and Control for Agile Underwater Vehicles in Confined Environments" by Luyue Huang, Bo He and Tao Zhang, [2], present a compact, affordable, high-performing vision-based self-localization module and its integration into a powerful Guidance, Navigation, and Control (GNC)-framework that enables hydrobatic maneuvering with agile micro underwater robots. Three on-board components were

utilized: a Raspberry Pi 4, an IMU, and a low-cost wide-angle Raspberry Pi camera. Their research was carried out in a freshwater tank that had fiducial markers placed at predetermined locations. They employed EKF technique which they enhance by a dynamic measurement noise model. In two experimental conditions, they assess how well their system performs. They started by examining the localization module's performance to show that it is adequate for precise visual underwater localization and benchmarking tasks. The experiment proved that the module's localization capabilities were quite accurate. The second experiment was centered on the GNC-integration framework's and the localization system's robustness. The outcome was that the suggested GNC-framework could move the robot along the path with repeatability and accuracy [2].

This study article is quite similar to this project. To tackle the challenges of underwater positioning, they employ low-cost components such as a Raspberry Pi camera, an IMU, EKF, and AprilTag. Because of these parallels, the study report can provide useful information on how effectively this system works. The studies were carried out in a fresh water tank with AprilTags scattered across the tank's bottom. Their research demonstrates that their system can provide precise localization and path tracking, which is useful information to have before commencing this project.

The research study "An autonomous navigation algorithm for underwater vehicles based on inertial measurement units and sonar" by Daniel A. Duecker, Nathalie Bauschmann, Tim Hansen, Edwin Kreuzer, and Robert Seifried, [3], introduces the Inertial-SLAM algorithm, a method based on IMU and sonar for underwater vehicle autonomous navigation. In order to calculate the underwater vehicle's velocity, position, attitude, bias errors of the IMU, and features map around the underwater vehicle, the system employs a hybrid Rao-Blackwellised SLAM algorithm. The Inertial-SLAM method combines the SLAM algorithm based on particle filters with the Inertial Navigation System (INS). This approach has a low estimate error and a low time complexity. The IMUs' random drift is not taken into account by the traditional SLAM algorithm; it only considers the observation noise of the IMUs. The INS and SLAM algorithms are combined to create the Inertial-SLAM algorithm for underwater vehicles. IMU errors are included in the state vector to obtain the most accurate estimation of the errors and alter the sensor's output. This considerably decreased the cumulative error. The studies compared the most widely used SLAM algorithm, EKF-SLAM, with the inertial-SLAM method. The Inertial-SLAM is quicker than EKF-SLAM and has a lower time complexity. It can achieve the same level of accuracy with fewer particles than EKF-SLAM. The simulation errors are reduced significantly compared to EKF-SLAM, resulting in more reliable navigation and positioning for the Autonomous Underwater Vehicle (AUV) [3].

This article presents intriguing data because the project's initial plan was to employ a SLAM system in conjunction with an IMU to address the issue of positioning underwater. Through simulations, they discovered that the inertial-SLAM technique for underwater vehicles considerably minimizes IMU drifting error. This implies that employing a more standard SLAM method may provide some difficulties when used with IMU. Additionally, they discovered that the inertial-SLAM algorithm was more

accurate for positioning and navigation underwater. This suggests that employing SLAM systems to address the problem of location underwater can produce promising results, but there are some considerations that must be made during development.

The research paper "Improved Tag-based Indoor Localization of UAVs Using Extended Kalman Filter" by Navid Kayhani, Adam Heins, Wenda Zhao, Mohammad Nahangi, Brenda McCabe and Angela P. Schoellig, [4], discusses the use of AprilTag markers to create an EKF for enhancing the estimate module of an indoor localization system. They were able to merge data from AprilTag and an IMU using the EKF, as well as enhance the estimation technique by considering uncertainty. During the research, they compared the performance of the approach using the EKF to the performance of an older localization method that provided estimations based on simply a single tag or an IMU. Pose estimation was carried out using both approaches in four scenarios. The experimental findings showed that the new method using an EKF improved posture estimation in all cases studied. This means that using an EKF will smooth out the estimated path and reduce errors relative to the ground truth, making it more dependable for usage in the real world [4].

This research paper is relevant to this master thesis because it provides essential information regarding how the usage of an EKF is a superior solution for the problem of locating in non-GPS environments, such as indoors and underwater. The research article also explains how the data from the IMU and AprilTag may be used in an EKF, which is very beneficial for this thesis.

Chapter 3

Background and Methods

This chapter will go through the background and methods used for the master's thesis. It will explain how the challenge of positioning a camera underwater was addressed using a camera, IMU, AprilTag and an EKF.

Initially, the challenges of positioning a camera underwater were attempted to be overcome by combining an IMU with SLAM. However, there were several complications with employing SLAM during the project, so the project was adjusted to try to solve the problem using an IMU, AprilTag, and an EKF instead.

This chapter will begin with an introduction of the camera used for the thesis. Then it will describe what an IMU is, how it works, which IMU was utilized, and how the IMU was calibrated. The chapter will also explain what AprilTag is, how it works, and how the camera was calibrated for improved positioning results using AprilTag. After the AprilTag section, the chapter goes on to describe what a Kalman filter is, how it works, and how the Extended Kalman filter was used to integrate and filter the position data from the IMU and AprilTag. This chapter finishes with a discussion of what SLAM is, why it was not used further in the project, and a brief summary of the two SLAM systems tried, ORB-SLAM3 and pySLAM.

3.1 Raspberry Pi High Quality Camera

The Raspberry Pi High Quality (HQ) camera was used in this thesis. High resolution, sensitivity, and the option of using various lenses with both C- and CS-mount are all features of the HQ camera. The camera has a board with a Sony IMX477 sensor, which has a 12.3Megapixel (MP) resolution, a 7.9mm diagonal picture size, a back-illuminated sensor architecture, adjustable back focus, and a C to CS mounting adaptor. The Raspberry Pi HQ camera is seen in fig. 3.1 without a lens [5], [6].

For industrial and consumer applications, such as security cameras, that demand high quality and/or integration with specialized optics, the Raspberry Pi HQ Camera offers an alternative to the Raspberry Pi Camera Board v2. Using the most recent software update, it is compatible with all Raspberry Pi computer models, starting with the Raspberry Pi 1 Model B [6].

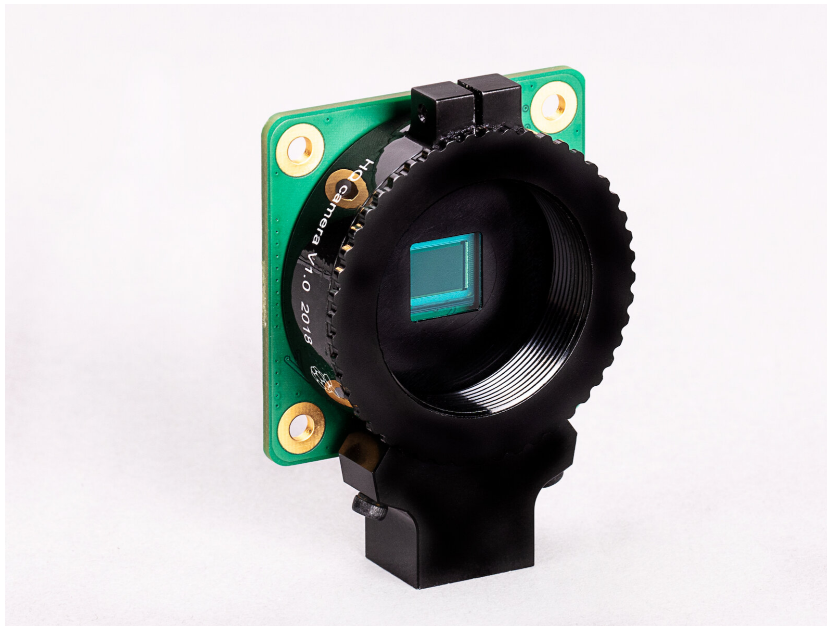


Figure 3.1: Raspberry Pi HQ camera [6]

The HQ camera's assembly is shown in fig. 3.2. When no lens is attached, a dust cap is installed on the camera to protect it from dust. This is because dust might cause the sensor to malfunction. When a lens is used, this cap is removed. The HQ camera is compatible with CS-mount lenses. To make the camera compatible with C-mount lenses, an optional adapter is provided to expand the rear focus by 5mm. When a CS-mounted lens is used, the C to CS adapter is removed, as it is only necessary when a C-mount lens is used. There are two objectives for the back focus adjustment mechanism. The back focus provides focus modification when using a small, inexpensive fixed-focus lens, but it also allows for focal range adjustment when using an adjustable-focus lens. The tripod mount is an additional accessory that may be removed when not required. The ribbon is used to attach the HQ camera to the Raspberry Pi's camera connection [7].

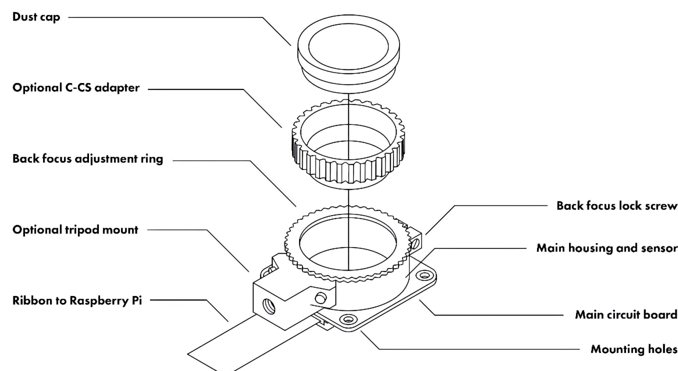


Figure 3.2: Raspberry Pi HQ camera assembly [6]

The lens that was used in this thesis can be seen in fig. 3.3. This 6mm wide-angle Infrared (IR) lens is designed to work with the Raspberry Pi HQ Camera. Since this lens is CS mounted, the C-CS adaptor had to be removed before usage. This lens has a back focal length of 7,53mm, a 1/2" picture format, and a 6mm focal length. Its field of view is 63°, and its resolution is 3MP High-definition (HD) [8].

Figure 3.3: Raspberry Pi HQ camera with lens [8]

This camera can easily be installed in a waterproof model for usage in water. The benefit of employing this camera for underwater use is that it can be directly connected to the Raspberry Pi, eliminating potential sources of errors. This camera is also affordable, and if autofocus is required later, it is possible to upgrade to the high-resolution autofocus camera for the Raspberry Pi [9]. The main reasons for selecting this camera for the thesis were that it was easy to use and was already accessible for usage at the firm Zebop, which provided the master's thesis.

3.2 Inertial measurement unit

The inertial measurement unit, also known as IMU, is a sensor that measures motion in a given timeframe. The IMU is used to measure and provide specific force, angular rate, and body orientation by combining three sensors: a gyroscope, an accelerometer, and a magnetometer. The angular rate is measured by the gyroscope, the specific force and acceleration are measured by the accelerometer, and the magnetic direction is measured by the magnetometer [10], [11], [12].

An IMU may offer information ranging from 2-Degrees of freedom (DOF) to 9-DOF, which defines how many different ways an item can move in three dimensions. The

most popular form of IMU is a 6-DOF IMU with one gyroscope and one accelerometer. It is also becoming increasingly typical to employ a 9-DOF IMU with all three sensors: a gyroscope, an accelerometer, and a magnetometer [11], [13].

Using sensor fusion, the IMU can combine all sensor data from the three sensors and provide information about the orientation of the device. The accelerometer provides information about the orientation along the x, y, and z axes, while the gyroscope provides information about the orientation along the yaw, roll, and pitch angles. The magnetometer provides information about the object's orientation in relation to the Earth's frame [11], [13].

Figure 3.4 depicts the operation of a 9-DOF sensor, where the black X, Y, and Z axes represent the accelerometer, the red Ω_x, Ω_y, and Ω_z axes represent the gyroscope, and the blue x, y, and z axes represent the magnetometer.

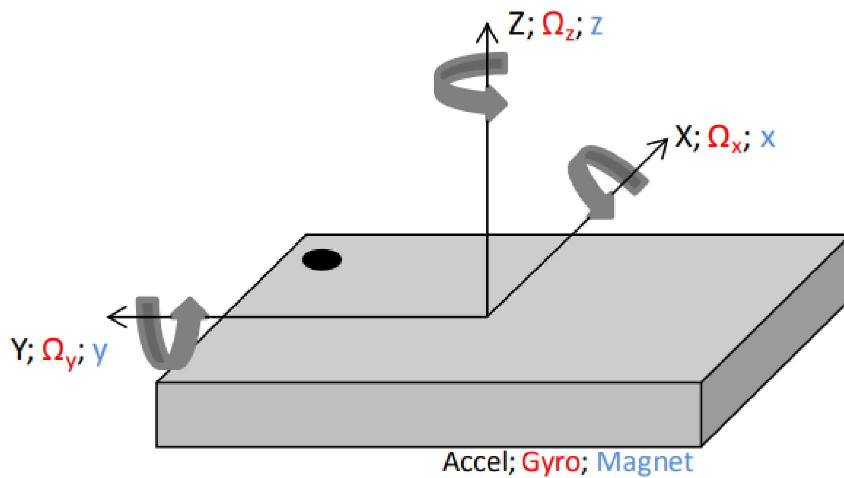


Figure 3.4: 9-DOF IMU sensor [14]

The main problem with an IMU is that it is sensitive to "drift," or accumulation of inaccuracy over time. The IMU repeatedly rounds off tiny fractions in its estimations since it is always sensing changes relative to itself, rather than comparing against a fixed or known external component, which adds up over time. These minor inaccuracies might build-up to big errors if left untreated [10], [15].

An IMU was utilized for underwater positioning because it is inexpensive, compact, and energy-efficient, making it simple to attach to a camera. Since GPS does not work underwater, it is feasible to determine the position and orientation of the camera using outputs like acceleration, velocity, angular rate, and magnetic direction. The accuracy and performance of the IMU are, however, affected by a variety of parameters, such as temperature, calibration, sensor noise, and bias. As a result, it should not be utilized alone.

3.2.1 Adafruit IMU BNO055

Adafruit BNO055 IMU was utilized in this thesis. This IMU is a 9-DOF sensor with sensor fusion algorithms that combine the orientation data from the accelerometer, gyroscope, and magnetometer. It is equipped with a small microprocessor that collects and combines all sensor data to convert it to true orientation without the use of a Kalman filter [16], [17].

Three degrees of acceleration, magnetic orientation, and angular velocity are among the nine forms of motion or orientation data that the IMU can acquire. It converts accelerometer, gyroscope, and magnetometer sensor data into accurate "3D space orientation." Both in the Euler vector and the quaternion, the IMU may output absolute orientation. The Euler vector offers orientation data based on a three-axis 360-degree sphere, but the quaternions' four-point output allows for more exact data management. It may also produce an angular velocity vector as a three-axis rotation speed and a three-axis acceleration vector. The magnetic field strength vector and the linear acceleration vector can both be generated by the IMU as three-axis magnetic field sensing data and three-axis linear acceleration data. Finally, the IMU may generate a gravity vector as a three-axis of gravitational acceleration, as well as the ambient temperature in degrees Celsius [16], [17]. The Adafruit BNO055 IMU is shown in fig. 3.5.

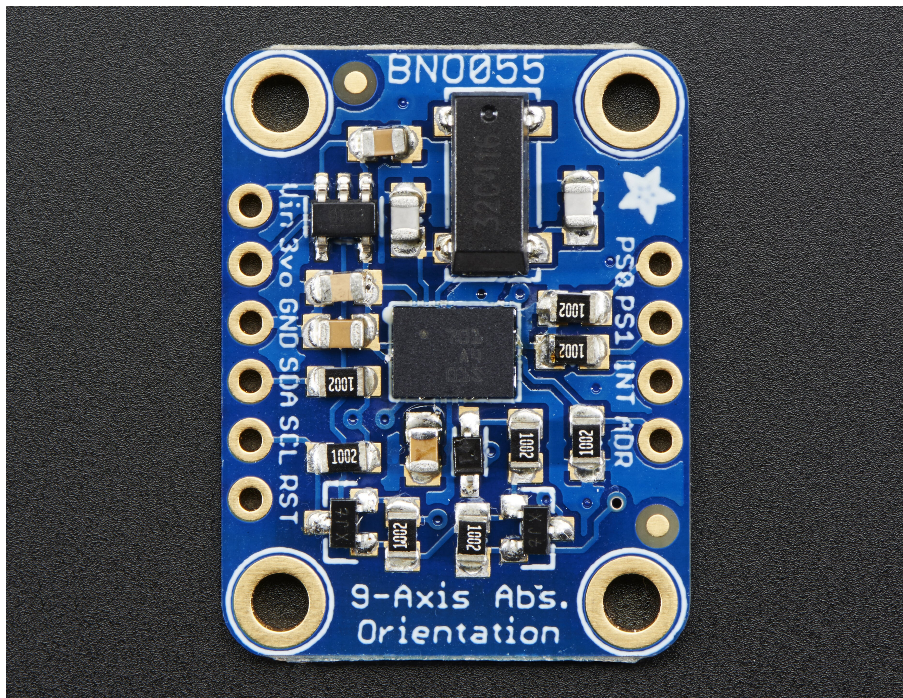


Figure 3.5: IMU BNO055 from Adafruit [17]

3.2.2 Calibration of IMU

Calibrating the IMU before use is critical to guarantee adequate performance. The BNO055 comes with a WebGL example of how to calibrate the IMU. This example demonstrates how to transmit orientation measurements to a web page and rotate a 3D model using them. The gyroscope, accelerometer, magnetometer, and system were calibrated using this example. The 3D model in the example will not provide proper orientation data without calibration, which is why it is critical to calibrate the IMU before using it. The IMU is reset every time the power is switched off. This means that every time the IMU is turned on or reset, it is necessary to calibrate it. The BNO055 IMU does much of the calibration on its own; all that is required to finish the calibration is for the sensor to be moved in particular directions [18].

Figure 3.6 depicts how the web page appeared when the WebGL example was launched. The current calibration status of the BNO055 sensor is displayed in the bottom center column of the web page. When calibrating the sensor, each component, including the system (or fusion algorithm), gyroscope, accelerometer, and magnetometer, must be calibrated independently. Each component has a calibration level ranging from 0 to 3, with 0 being uncalibrated and 3 denoting fully calibrated. All four components should have a calibration level of at least 3 to acquire the best orientation data. However, if only a few of the components are calibrated to level 2 or 3, the results should be acceptable [18]. Because all four components, gyroscope, accelerometer, magnetometer, and the system have calibration value 3 in fig. 3.6, they are properly calibrated.

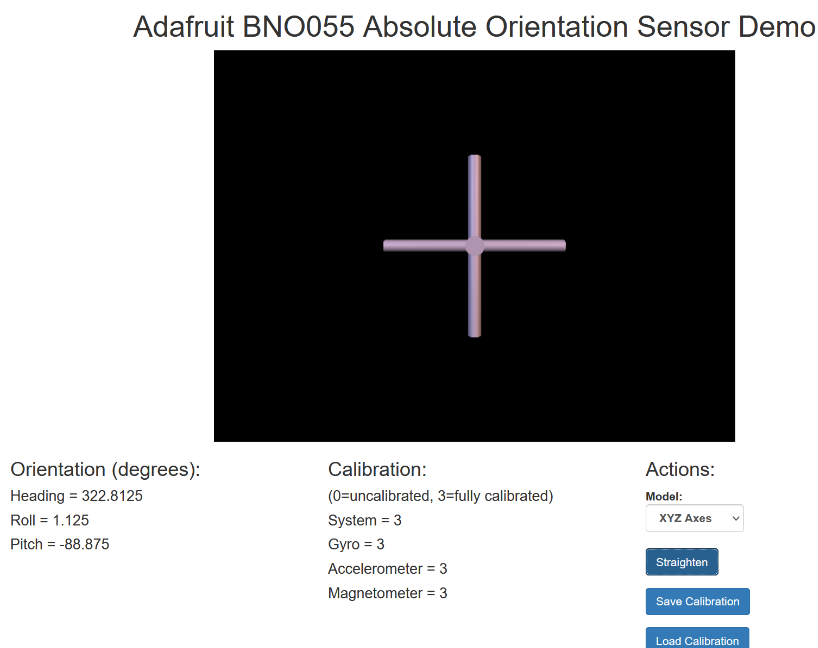


Figure 3.6: WebGL example web page are used to calibrate the IMU

The gyroscope was the simplest component to calibrate. All that was required was to place the IMU on a level surface for a few seconds. Calibrating the accelerometer required a bit more effort. The accelerometer was calibrated by carefully moving the IMU in six different directions and holding it for a few seconds in each direction. To calibrate the magnetometer, it was vital to ensure that the IMU was not too close to any metal objects, as this might affect the calibration or slow down the calibration process. To calibrate the magnetometer, the IMU had to be moved in an infinite pattern until the magnetometer was properly calibrated. The system was the last component to be calibrated. The calibration of the system began to calibrate while the other components were being calibrated. When one component was properly calibrated, the system began its calibration process. When all of the other components had been thoroughly calibrated, the system calibration was nearly complete. All that remained was to leave the IMU alone for a few seconds so that it could finish calibrating the system.

The IMU's current orientation was presented in the bottom left column of the web page, as shown in fig. 3.6. These numbers represent orientation in terms of heading, roll, and pitch in degrees. These are the angles of the IMU axes X, Y, and Z in the local plane. The rotation around the Z-axis is known as heading, the rotation around the Y-axis is known as pitch, and the rotation around the X-axis is known as roll. The orientation values vary as the IMU rotates and the 3D model rotates with it.

The last column, at the bottom right corner of the web page, was where the calibration could be saved and loaded. It was possible to use the calibration later using this feature, eliminating the need to repeat all of the stages to calibrate each component. The "Save Calibration" button saved all of the calibration settings to a file called "calibration.json" on the Raspberry Pi. It was possible to use the functions "get_calibration" and "set_calibration" to utilize the finalized calibration in a code. Section 4.3 code 2 shows how these two functions were implemented in the code.

The IMU has to be recalibrated when the device is used underwater since variables like depth, temperature, and other underwater environmental factors might create errors and inaccurate readings. To get more information and a step-by-step tutorial on how the IMU was calibrated, see attachment B and the pre-project [19].

3.3 AprilTag

«AprilTag is a visual fiducial system, useful for a wide variety of tasks including augmented reality, robotics, and camera calibration»[20].

The AprilTag detecting program computes the precise 3D location, orientation, and Identification (ID) of the tags with respect to the camera. It is intended to be both portable to embedded devices and simple to integrate into other programs. Even with Central Processing Unit (CPU)s made for cell phones, real-time performance

is possible. Since AprilTags are a form of two-dimensional bar code, they share certain conceptual similarities with Quick Response (QR) codes. They are made to encode much smaller data payloads, ranging from 4 to 12 bits, enabling more reliable detection over a wider area. Additionally, because of its high localization precision design, it is feasible to determine the exact 3D position of the AprilTag in relation to the camera [20].

Figure 3.7 depicts the six different AprilTag families. The collection of tags that the AprilTag detector will look for in an input image is defined by an AprilTag family. The default AprilTag family is "Tag36h11," which is the tag used in this thesis [21]. The tags can be any size, with a black outline square on a white background and an embedded black bare-code inside the square. Even in poor visibility conditions, the tags offer a method of identification and 3D placement. The tags function similarly to barcodes by storing a limited amount of data, such as tag ID, and allowing for quick and precise calculation of the tag's 6D (x, y, z, roll, pitch, and yaw) position [22], [23].

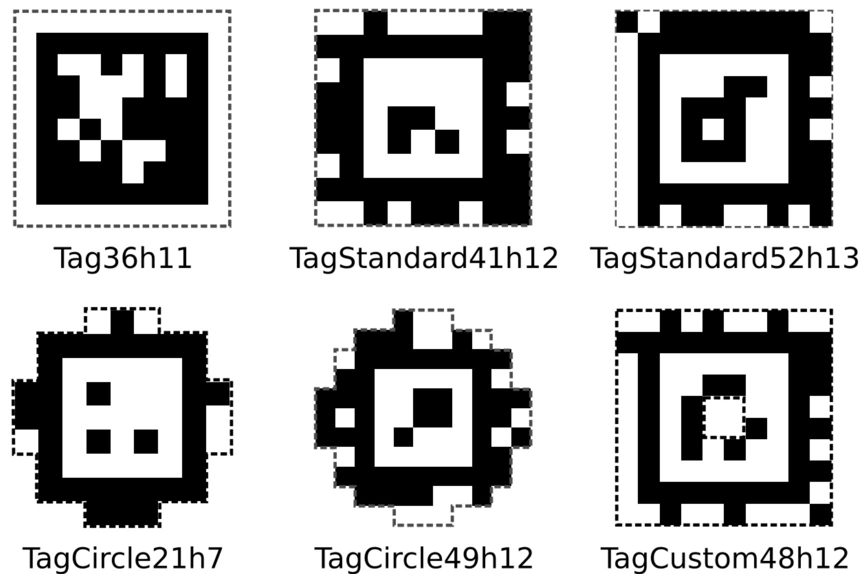


Figure 3.7: AprilTags [20]

To compute tag pose estimate, the tag size must be known. The tag size was measured from where the white and black borders meet, as shown in fig. 3.8, and not from the exterior of the tag. The length of the edge between the white border and the black border, or alternatively, the distance between the corners that can be detected, was used to define the tag size. The tag's dimensions are measured in meters [24].

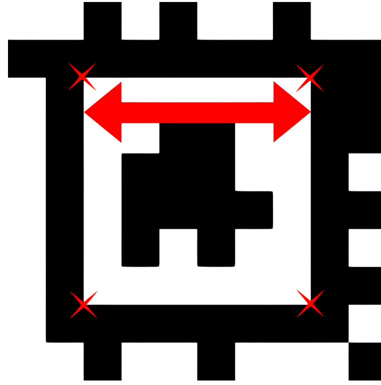


Figure 3.8: AprilTag size [24]

A set of poses from AprilTag were generated each time the camera switched positions. Each pose describes the camera's position along the moving camera trajectory at a certain moment in time. The homogeneous transformation in eq. (3.1) was used to provide a continuous trajectory between two positions [22].

$$d_i = \begin{bmatrix} p_{i_x} & p_{i_y} & p_{i_z} \end{bmatrix}^T \tag{3.1a}$$

$$R_i = \begin{bmatrix} c\phi c\theta & c\theta s\psi s\phi - c\psi s\theta & c\psi c\theta s\phi + s\psi s\theta \\ c\phi s\theta & c\psi c\theta + s\psi s\phi s\theta & -c\theta s\psi + c\psi s\phi s\theta \\ -s\phi & c\phi s\psi & c\psi c\phi \end{bmatrix} \tag{3.1b}$$

$$T_i^\tau = \begin{bmatrix} R_i & d_i \\ 0 & 1 \end{bmatrix}_{4\times 4} \tag{3.1c}$$

The camera frame of reference was used as the input angles in eq. (3.1). θ is the rotation about the z-axis that represents the camera's roll motion, ϕ is the rotation about the y-axis that represents the camera's yaw motion, and ψ is the rotation about the x-axis that represents the camera's pitch motion. Equation (3.2) represents the transformation T_i^{i+1} from point p_i to point p_{i+1} and can be used to calculate the trajectory in a single frame of reference [22].

$$T_i^{i+1} = T_\tau^{i+1} \times (T_\tau^i)^{-1}, \text{where} (T)^{-1} = \begin{bmatrix} R^T & -R^T d \\ 0 & 1 \end{bmatrix} \tag{3.2}$$

Figure 3.9 illustrates the camera moving and changing positions at different time stamps. Every position was at a unique location in relation to the tag. The continuous trajectory was created by detecting AprilTags at each position. The camera's center serves as the origin of the coordinate system. The z-axis extends from the lens of the camera outward. In the image that was captured by the camera, the y-axis was down and the x-axis was to the right. The coordinate frame for the tag was oriented with the z-axis into the tag, the x-axis to the right, and the y-axis down [24].

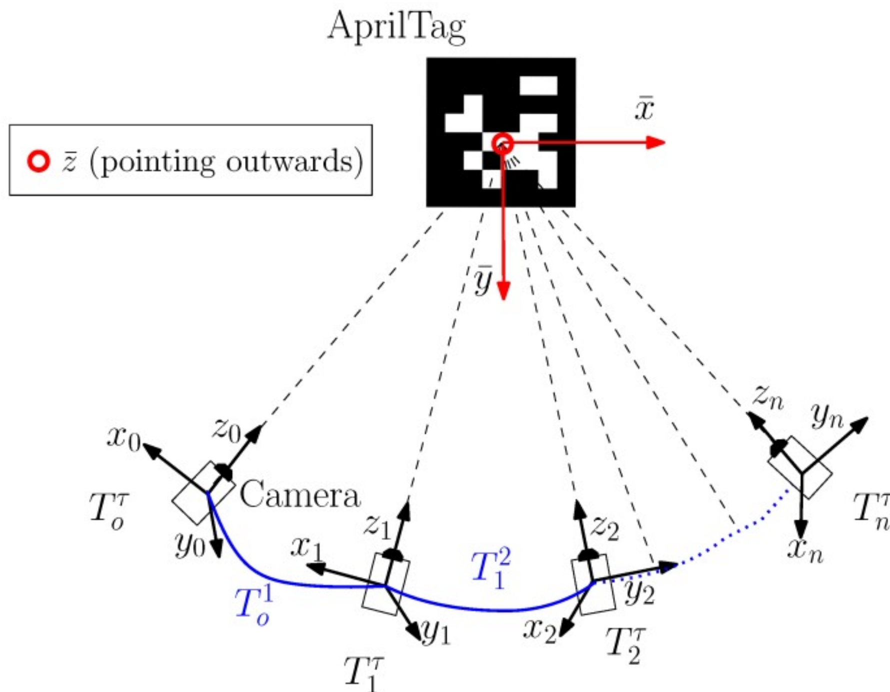


Figure 3.9: Different camera positions relative to a tag [22]

AprilTag was chosen as a good solution for underwater positioning because it can provide rapid and accurate pose estimation of tags even in low visibility environments. It is possible to locate the camera's position based on the locations of numerous tags that have been placed in a specific underwater area.

3.3.1 Calibration of camera

It was necessary to calibrate the camera in order to utilize AprilTag for pose estimation. This was because the images are distorted due to the use of low-cost camera. Figure 3.10 displays a checkerboard pattern captured using the Raspberry Pi HQ camera used in this thesis. This demonstrates that the image was distorted since the checkerboard's straight lines appear to be curved, which does not match the checkerboard pattern that was photographed. The image was getting increasingly distorted as the camera was moved away from the center of the image. Radial distortion is the term used to describe this type of distortion [25], [26].

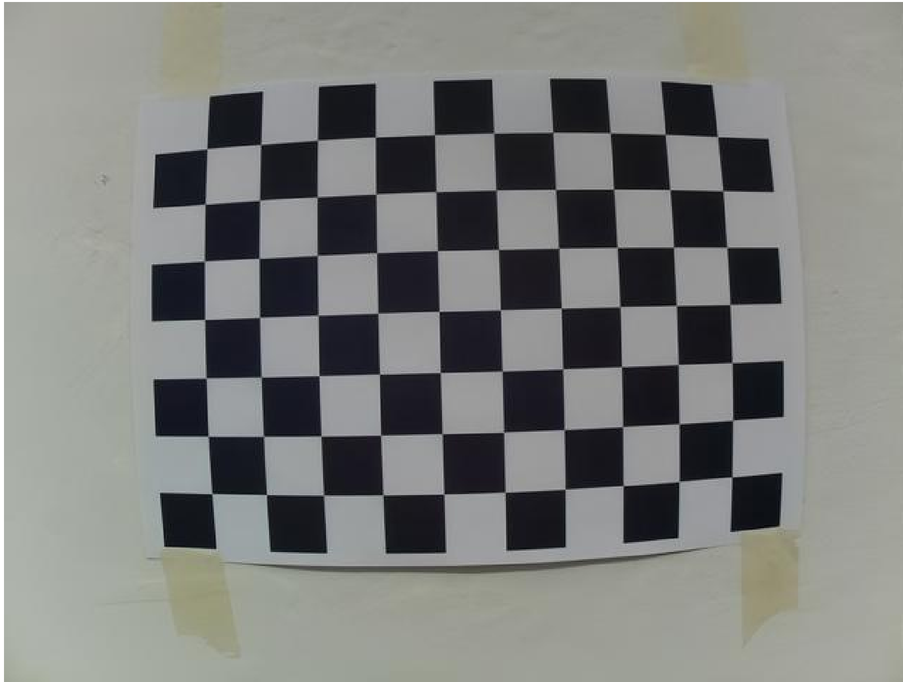


Figure 3.10: Distorted picture from the camera

Radial distortion was calculated using eq. (3.3). The old pixel point coordinates in the input picture are (x, y), while the corrected output image's location is $(x_{distorted}, y_{distorted})$ [25], [26].

$$x_{\text{distorted}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{3.3a}$$

$$y_{\text{distorted}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{3.3b}$$

Tangential distortion was another essential distortion to address. When the picture plane was not parallel to the image capturing lens, tangential distortion occurs. As a result, certain regions in the photos may appear closer than they are. Equation (3.4) was used to compute the tangential distortion [25], [26].

$$x_{\text{distorted}} = x + [2p_1 xy + p_2(r^2 + 2x^2)] \tag{3.4a}$$

$$y_{\text{distorted}} = y + [p_1(r^2 + 2y^2) + 2p_2 xy] \tag{3.4b}$$

The distortion coefficient is represented by eq. (3.5). It is represented by five distortion parameters in a one-row, five-column matrix [25], [26].

$$\text{Distortion coefficients} = (k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3) \tag{3.5}$$

To calibrate the camera, the 3x3 camera matrix shown in eq. (3.6) had to be found. The camera matrix consists of the focal length (f_x, f_y) and optical centers (c_x, c_y) given as pixel coordinates. Once computed, it can be saved for use in the future because it only depends on the camera.

$$\text{Camera matrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

The parameters in eq. (3.5) and eq. (3.6) were determined during camera calibration. The camera was calibrated using a checkerboard pattern. The checkerboard pattern that was utilized to calibrate the camera is seen in fig. 3.11. The design was printed on A4 paper and has squares measuring 25mm by 11x8. To prevent errors, it was crucial to ensure that the printed pattern was scaled correctly. The squares in the checkerboard pattern have to be large enough and distinct from one another in order to achieve the optimum calibrating results. The checkerboard pattern was provided by [27].

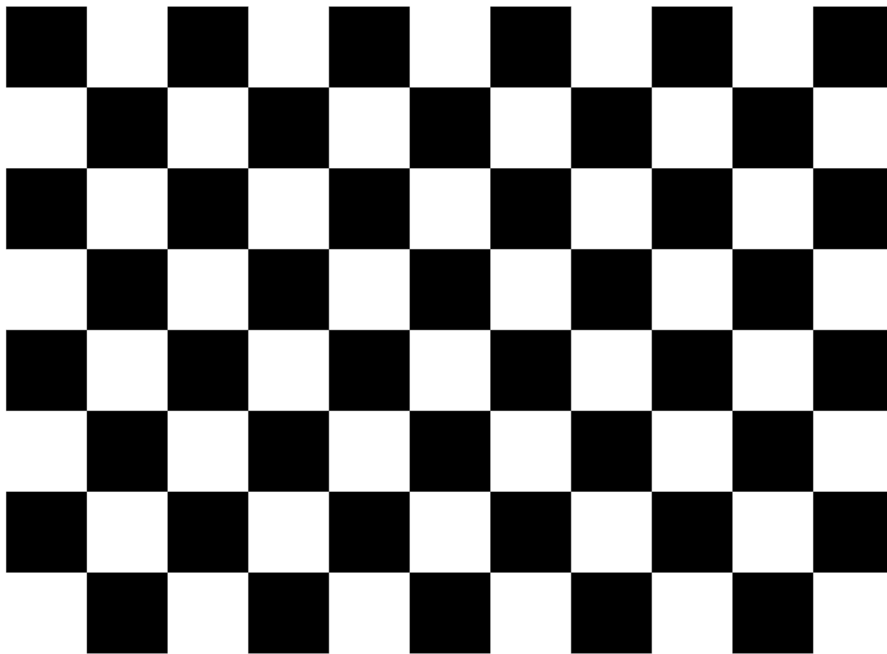


Figure 3.11: Checkerboard pattern

3.4 Kalman filter

A filter makes it possible to continually estimate a system's state using noisy measurements derived from different sensors, such as location, velocity, and orientation. A motion model is frequently used to explain the dynamics of the system, which defines how the states change over time. The filter's purpose is to combine the motion model data and sensor readings as effectively as possible to determine the new state estimate. The challenge in effectively fusing data from several sensors and a motion model, a process known as sensor fusion, is determining how much of the different sensors and the motion model to believe [28], [29].

The Kalman filter (KF) is an estimating algorithm that generates hidden variable estimates based on faulty and unreliable measurements. Additionally, the KF gives a prediction of the future state of the system based on prior estimates. The discrete-data linear filtering problem was addressed in a 1960 study by Rudolf E. Kalman, after whom the filter is named [30]. The KF is utilized in many different applications today, including computer graphics, location and navigation systems, control systems, and target tracking [31].

The processes that are being measured must be able to be modeled by a linear system in order to utilize a KF to eliminate noise from a signal. At each time increment, the new state is obtained by applying a linear operator to the previous state, as described in eq. (3.7a). It is also introducing noise to deal with unmodeled changes in the state. Equation (3.7b) describes the new measurement. In eq. (3.7), x_k represents the state vector, u_k represents the control input vector, and z_k represents the measurement vector at time k. The transition matrix A connects the previous time step $k-1$ to the current time step k, the control input matrix B connects the control input u to the state x, and the transformation matrix H connects the state vector to the measurement domain. The final two variables are random Gaussian noise, with w_k representing process noise and v_k representing measurement noise [29], [32].

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1} \tag{3.7a}$$

$$z_k = Hx_k + v_k \tag{3.7b}$$

The KF uses a type of feedback control to estimate the process. The KF algorithm is depicted in a process diagram in fig. 3.12. The state x_0, including position and velocity, and the covariance matrix P_0, are initialized in the first step, which is called the initial step. The subsequent phase turns these initial states into previous states, x_{k-1} and P_{k-1}. The prediction model, $\hat{x}_k^-$ and P_k^-, which is the following phase, is developed based on previous values and model. The Kalman gain, K_k, is calculated using this prediction model. The final step, known as the updated step, combines the measured, z_k, and predicted, $\hat{x}_k^-$, locations to provide the updated location $\hat{x}_k$. Depending on the level of uncertainty with each measurement, the KF will choose either the predicted location or the measured location. The procedure is repeated

when the system receives new sensor data and the updated state, $\hat{x}_k$ and P_k, is utilized as feedback to the prediction model. The KF returns the updated state x_k and covariance matrices P_k [33].

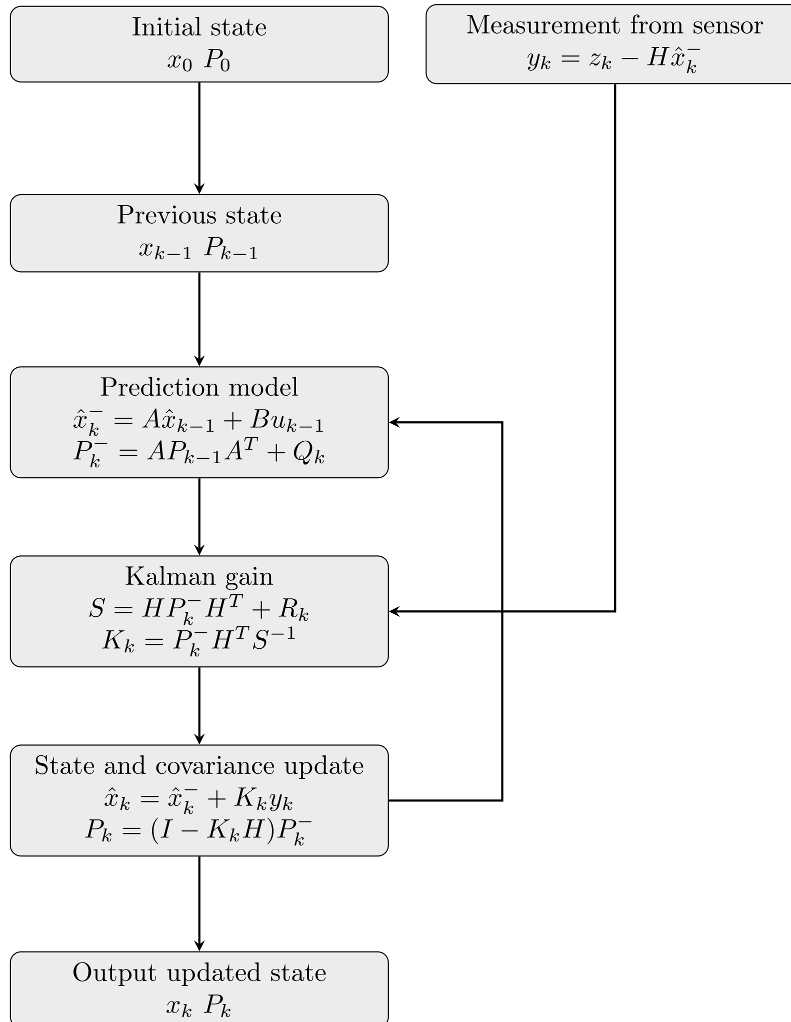


Figure 3.12: Kalman filter process diagram

The prediction step of the KF is described in eq. (3.8). To obtain the a posteriori estimations, the prediction step makes temporal predictions into the future while estimating the current state and error covariance [29], [32]. The predicted state estimate is $\hat{x}_k^-$, the system inputs is u_{k-1}, the predicted estimate error covariance matrix is P_k^-, and the process noise covariance matrix is Q_k.

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1} \tag{3.8a}$$

$$P_k^- = AP_{k-1}A^T + Q_k \tag{3.8b}$$

The measurement update process for the KF is described in eq. (3.9). To get a better a posteriori estimate, the update step adds new measurements to the a priori estimations. As a weighted average of the a priori state estimate $\hat{x}_k^-$ and the measurement residual, the posteriori state estimate $\hat{x}_k$ in equation eq. (3.9d) is produced. The weights' function is to help select which measurement estimate should be trusted more. Values with better predicted uncertainty are more trustworthy since the Kalman gain is calculated based on their uncertainties. The Kalman gain cancels out the impact of the prediction if the new measurements have better estimates, but it does the opposite if the prediction has superior estimates [29], [32]. The updated state estimate is $\hat{x}_k$, the measurement is z_k, the Kalman gain at time k is K_k, the state-to-measurement matrix is H, and the measurement covariance matrix is R_k.

$$y_k = z_k - H\hat{x}_k^- \tag{3.9a}$$
$$S = HP_k^- H^T + R_k \tag{3.9b}$$
$$K_k = P_k^- H^T S^{-1} \tag{3.9c}$$
$$\hat{x}_k = \hat{x}_k^- + K_k y_k \tag{3.9d}$$
$$P_k = (I - K_k H)P_k^- \tag{3.9e}$$

Every time step, this process is repeated, with the updated estimate and its covariance determining the prediction utilized in the subsequent iteration. As a result, the Kalman filter calculates a new state using only the most recent best prediction rather than the history of a system's state [29], [32].

3.4.1 Extended Kalman filter

Since the KF can only be used for linear systems, it is essential to overcome this limitation and enable non-linear system state estimation. An Extended Kalman filter (EKF) is a kind of KF that linearizes about the current mean and covariance. Equation (3.10a) represents the new state model and eq. (3.10b) represents the new measurement, both of which contain non-linear functions, f and h, and random Gaussian noise w_k and v_k [32].

$$x_k = f(x_{k-1}, u_k) + w_k \tag{3.10a}$$
$$z_k = h(x_k) + v_k \tag{3.10b}$$

The covariance cannot be calculated with the same approach as for the KF since the system is non-linear. By calculating the Jacobians in eq. (3.11), the EKF linearizes with respect to the mean of the current estimate and covariance [32].

$$A_k = \left.\frac{\partial f}{\partial x}\right|_{\hat{x}_{k-1}^-, \hat{u}_k} \tag{3.11a}$$

$$H_k = \left.\frac{\partial h}{\partial x}\right|_{\hat{x}_{k-1}^-} \tag{3.11b}$$

In the same way as the KF, the prediction step of the EKF in eq. (3.12) predicts the state and covariance estimates from the previous time step to the current time step [32].

$$\hat{x}_k^- = f(x_{k-1}, u_{k-1}, w_{k-1}) \tag{3.12a}$$

$$P_k^- = A_k P_{k-1} A_k^T + Q_k \tag{3.12b}$$

Similar to the KF, the state and covariance estimates are updated with the measurement z_k using the EKF measurement update equations in eq. (3.13) [32].

$$y_k = z_k - H_k \hat{x}_k^- \tag{3.13a}$$

$$S_k = H_k P_k^- H_k^T + R_k \tag{3.13b}$$

$$K_k = P_k^- H_k^T S_k^{-1} \tag{3.13c}$$

$$\hat{x}_k = \hat{x}_k^- + K_k y_k \tag{3.13d}$$

$$P_k = (I - K_k H_k) P_k^- \tag{3.13e}$$

By using an EKF in the underwater positioning system, it is possible to fuse the positioning data for the IMU and the AprilTag. All measurement data are taken into account during the estimating step of the EKF method. The prediction step creates a predicted position using data from the IMU and the previous position. In the correction stage, when tag data is combined to form an overall estimate, the prediction is updated. This estimate is then used as the prior position in the next time step prediction. By doing this, information from the IMU and all of the visible tags will be combined, rather than changing between the IMU and the tag-based localization in various circumstances.

3.5 Simultaneous Localization and Mapping

Simultaneous Localization and Mapping, also known as SLAM, refers to the technique of mapping an unfamiliar environment while maintaining track of the device's location inside it. Engineers utilize map data for purposes like trajectory tracking and obstacle detection. SLAM systems make data collecting easier and may be utilized in both outdoor and indoor scenarios [34], [35].

The SLAM technology combines data from the system's onboard sensors and employs computer vision algorithms to analyse it in order to identify features in the immediate area. As a result, SLAM is able to create a rough map and determine the system's location. When the camera moves, SLAM uses the original location estimate as a starting point and updates it with additional information from the system's on-board sensors. The cycle is completed when the new position estimate is known, at which point the map is updated in turn. SLAM monitors the route as the camera moves through the assets by continually repeating these steps. It also creates a thorough map concurrently [36].

A SLAM system's architecture consists of the front end and the back end as its two main components. While the back end does estimate and infer on data from the front end, the front end collects pertinent data from raw sensor measurements and makes data associations between the measurements and the map. Other types of sensors, such as IMU, may also contribute readings to the back end. Figure 3.13 provides a summary of this architecture [37], [38].

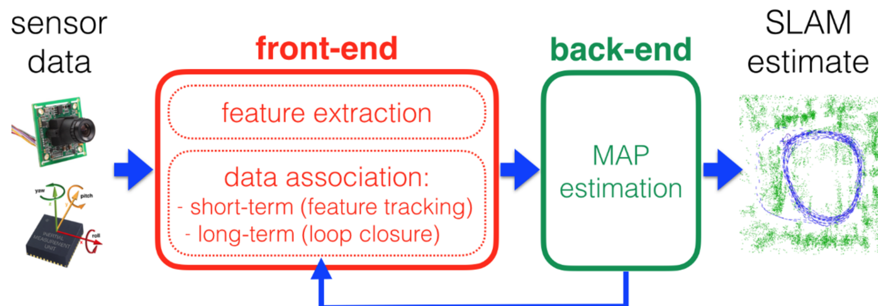


Figure 3.13: SLAM systems architecture [37]

In the beginning of the master thesis, the idea to solve the problem of positioning under water was to use SLAM. During testing there where tried out two different SLAM systems, ORB-SLAM3 and pySLAM, but because allot of problems occurred during installation and testing of these systems, and all the troubleshooting was time consuming, the project was to take a slightly different angle to solve the task.

3.5.1 ORB-SLAM3 and pySLAM

ORB-SLAM3 is the first real-time SLAM framework capable of performing Visual, Visual-Inertial, and Multi-Maximum a posteriori (MAP) SLAM using monocular, stereo, and Red, Green and blue - Depth sensing (RGB-D) cameras with pinhole and fisheye lens models. ORB-SLAM3 outperforms the top systems in all sensor combinations in terms of robustness and accuracy [39].

PySLAM includes a python version of a monocular Visual Odometry (VO) pipeline. It supports a wide range of traditional and modern local characteristics, as well as providing a user-friendly interface. It also includes a collection of other typical and essential VO and SLAM technologies. [40]

Chapter 4

Experiments and Results

This chapter will review all of the project's experiments and their results. It will begin by discussing all of the issues that have arisen in relation to SLAM. It will then discuss general problems that have arisen throughout the project. The implementation of the IMU will then be explained, along with some test results that were obtained. The next section will explain how AprilTag was calibrated and implemented, and it will then present some test findings. Finally, the chapter will finish with an explanation of how the EKF was implemented and a discussion of the findings.

4.1 Simultaneous Localization and Mapping

This section will go through all of the issues that were experienced when using ORB-SLAM3 and pySLAM. It will provide a brief description of how the problems developed and how they were handled.

4.1.1 ORB-SLAM3

ORB-SLAM3 was the first SLAM software to be tested. There where several problems that occurred during the installation and testing of ORB-SLAM3. During the pre-project, [19], Open Source Computer Vision Library (OpenCV) 4.5.2 and Pangolin were installed successfully, however ORB-SLAM3 had certain issues that needed to be rectified. The issue that caused the ORB-SLAM3 installation to fail was that it attempted to build using C++11, however this system has an earlier version of C++. The C++ version was upgraded because this system runs Ubuntu 21.10, which was a newer version than the one used by the ORB-SLAM3 team [39]. In this system, C++14 was utilized instead of C++11. Changing the "CMakeList.txt" file to use C++14 instead of C++11 fixed the problem. Once this error was rectified, ORB-SLAM3 installation was finished successfully. Figure 4.1 shows where C++11 was replaced with C++14.

```
15 # Check C++14 or C++0x support
16 include(CheckCXXCompilerFlag)
17 CHECK_CXX_COMPILER_FLAG("-std=c++14" COMPILER_SUPPORTS_CXX11)
18 CHECK_CXX_COMPILER_FLAG("-std=c++0x" COMPILER_SUPPORTS_CXX0X)
19 if(COMPILER_SUPPORTS_CXX11)
20    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++14")
21    add_definitions(-DCOMPILEDWITHC11)
22    message(STATUS "Using flag -std=c++14.")
23 elseif(COMPILER_SUPPORTS_CXX0X)
24    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x")
25    add_definitions(-DCOMPILEDWITHC0X)
26    message(STATUS "Using flag -std=c++0x.")
27 else()
28    message(FATAL_ERROR "The compiler ${CMAKE_CXX_COMPILER} has no C++14 support. Please use a
   different C++ compiler.")
29 endif()
```

Figure 4.1: Changes on CMakeList.txt

An installation guide for ORB-SLAM3 on Ubuntu 21.10 has been created. This installation tutorial covers memory swapping, OpenCV installation, Pangolin installation, and ORB-SLAM3 installation, as well as instructions on how to upgrade from C++11 to C++14. It also contains a guidance on how to conduct testing of the system. Appendix A contains this installation guide.

Following the installation of ORB-SLAM3, certain tests were performed on the system to ensure that it was functioning correctly and to determine how accurate the mapping estimation was in comparison to the ground truth. The dataset EuRoc, which was utilized by the ORB-SLAM3 team, was employed in these tests. EuRoc is a collection of visual-inertail datasets gathered from a Micro Aerial Vehicle (MAV). Stereo pictures, synchronized IMU measurements, and accurate motion and structural ground-truth are all included in the datasets [41]. Figure 4.2 depicts how it appeared when the system was tested with EuRoc using camera and an IMU.

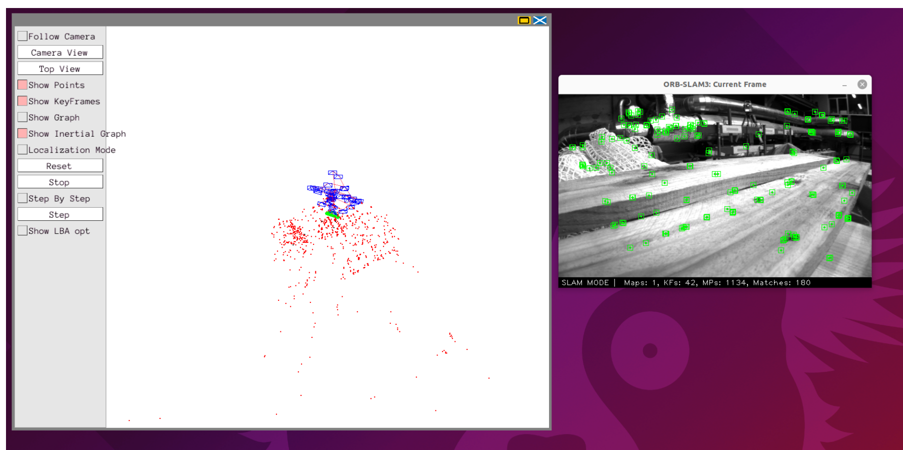


Figure 4.2: EuRoc test of ORB-SLAM3 with camera and IMU

The ORB-SLAM3's functionality may be verified after the test. The following test was performed to assess how well the ORB-SLAM3 estimator compares to a file named ground truth. This test was performed on EuRoc using stereo. The output of ORB-SLAM3 is seen in fig. 4.3. This test was completed entirely, and the ORB-SLAM3 estimation was plotted against the ground truth. There were some issues during the plotting since the plotting file is written in Python2.7 and the system is

written in Python3. To make the plotting work, certain changes were done to the files home/username/Dev/ORB_SLAM3/evaluation/evaluate_ate_scale.py and home/username/Dev/ORB_SLAM3/evaluation/associate.py. The modifications made from python2.7 to python3 were as follows ".sort()" was changed to "sorted()", ".keys()" was changed to "list()", and some parentheses were added to the print functions.

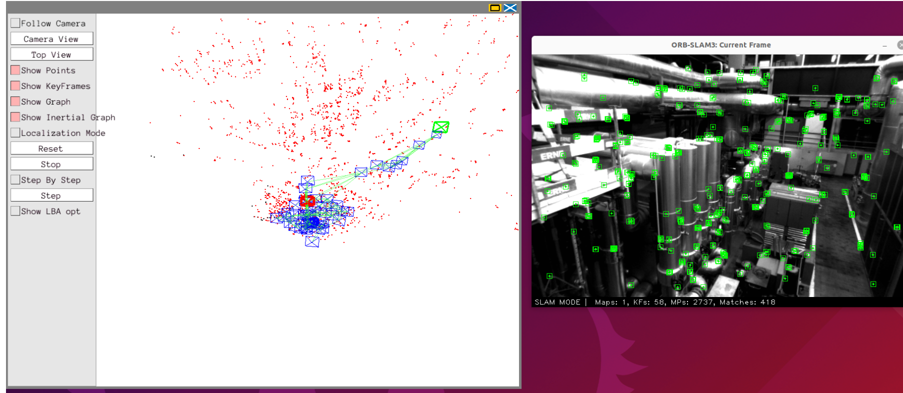


Figure 4.3: EuRoc test of ORB-SLAM3 with stereo

The validation was plotted after the files were updated. The plot is depicted in fig. 4.4, with a zoomed-in section of the plot. This zoomed-in section demonstrates how well ORB-SLAM3 estimate works. Under the difference in red, the estimation in blue and the ground truth in black are shown. This signifies that the difference between the estimate and the ground truth is quite minimal.

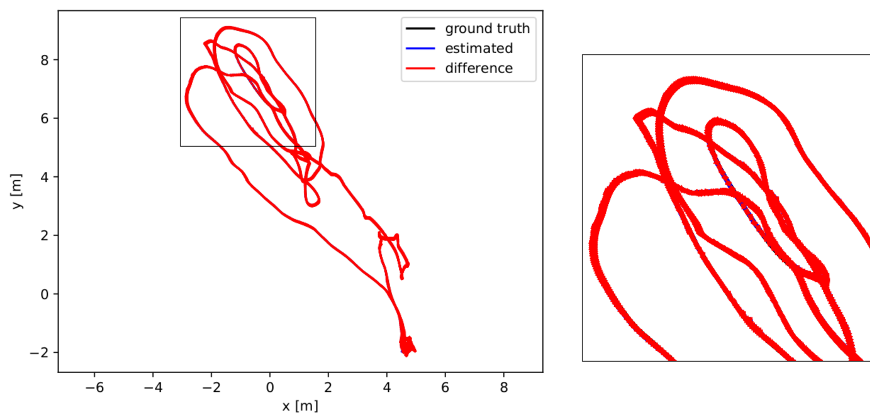


Figure 4.4: Plot of EuRoc test of ORB-SLAM3 estimation against the ground truth, including a zoomed in section of the plot

The issues with ORB-SLAM3 occurred while attempting to utilize a camera to deploy ORB-SLAM3. There have been various attempts to make it operate, as well as some attempts to calibrate the camera, but the primary issue was that ORB-SLAM3 was utilizing software that has not been updated for usage with Ubuntu

21.10. There were also a lot of difficulties with that there was little information on how to address all of the problems, therefore it took a long time to resolve each problem. There was also an attempt to run ORB-SLAM3 on a virtual machine, however this did not provide a solution to the problems. Because all of this troubleshooting was so time consuming and complicated, it was determined that the focus of this thesis would shift and that one other SLAM system would be used to complete the thesis assignment.

A calibration guide for ORB-SLAM3 was created, however because the calibration was never finished, the calibration guide was likewise incomplete and will not be included in this report.

4.1.2 PySLAM

PySLAM was the second SLAM software tested. The issues with pySLAM began during the installation process. The biggest issue was meeting the criteria that pySLAM requires in order to be installed. Python 3.6.9, numpy 1.18.2, OpenCV 4.5.1, Pytorch >= 1.4.0, and Tensorflow-Graphics Processing Unit (GPU) 1.14.0 are required to install pySLAM. In order to execute the full version of pySLAM pangolin and General Graph Optimization (g2o)py, must also be installed. Meeting all of these requirements was challenging since pySLAM was developed and tested on Ubuntu 18.04, but the systems utilized in this thesis were Ubuntu 20.04 and Ubuntu 21.10 [40].

Throughout the installation process, there were some challenges with installing the necessary versions of tensorflow-GPU, OpenCV, and g2opy for pySLAM. Attempts were made to install newer versions of these softwares, however this simply resulted in pySLAM installation failing. There were several attempts to get the pySLAM installation to work, but none were successful. Due to these problems, an attempt was made to install pySLAM using conda. Conda installation had several complications, but they were fixed by identifying the proper version of conda that supported aarch64 and Python 3.8.

Despite all of this debugging, there were still issues with pySLAM installation. After much study, the issues appeared to be that this thesis was utilizing a Raspberry Pi 4 which is an ARM64 system and hence requires installation files that support aarch64, but pySLAM is built on AMD64 systems and thus supports x86_64 files. There were attempts to resolve this issue by forcing the system to utilize aarch64 files rather than x86_64 files, however this operation proved too time consuming and complicated. As a result, it was decided that the thesis would not employ SLAM to solve the thesis challenge.

An installation guide for pySLAM was developed; however, because the installation was never completed, the installation guide is also incomplete and will not be included in this report.

4.2 General problems

Additional to SLAM difficulties, some other problems that have occurred and have been time consuming to solve will be described in this chapter. All of these problems have been explained as to why they arose and how they were resolved.

4.2.1 Memory problems

During the installation and testing of ORB-SLAM3 and pySLAM, there were some issues with the system having insufficient memory. The cause for this issue is that multiple attempts were made to make these softwares operate, and as a result, several minor installs were performed, which combined occupied a large amount of memory on the system. Some attempts to free up space on the memory were made, but they had little effect, therefore the problem was solved by reinstalling the memory card. There was also a change to a bigger memory card, from 32Gigabyte (GB) to 64GB, to lessen the likelihood of experiencing the same issue in the future. This indicated that everything on the system had been deleted, and it would be necessary to reinstall everything. This includes the IMU, camera, and ORB-SLAM3 installation. In addition, several codes were lost, including the code to get the IMU operational, as well as the calibration of the IMU. The code to get the camera to function with Real Time Streaming Protocol (RTSP)-stream and HQ Raspberry Pi camera with OpenCV, as well as a code that contained both the camera and the IMU, were also lost.

4.2.2 Second raspberry Pi

A second Raspberry Pi 4 with Ubuntu 20.04 was also used in the experiment. The purpose for this was so that tests could be run simultaneously on both Ubuntu 21.10 and Ubuntu 20.04 systems to ensure that this was not the issue. As a result, it was also essential to prepare a second IMU for use and testing.

4.2.3 IMU problems

The booting problem that occurred when the IMU was attached was mentioned in the pre-project, [19], and since an IMU was necessary to be utilized with Ubuntu 20.04, it was important to find a solution. Several ways were tried to address this problem, but the one that worked was a comparison of the files /boot/config.txt, /boot/firmware/config.txt, and /boot/firmware/cmdline.txt on Ubuntu 20.04 and Ubuntu 21.10. The solution was to make the files from Ubuntu 20.04 and the files from Ubuntu 21.10 identical, and certain commands were added to both systems files.

The following modifications were made to the file /boot/config.txt: "force_turbo=1", "dtoverlay=disable-bt", and "dtoverlay=miniuart-bt" were inserted at the bottom of the file. The only modification in the file /boot/firmware/cmdline.txt was the removal of "console=serial0,115200". Several changes were made to the final file /boot/firmware/config.txt, as seen in fig. 4.5. All places where the kernel uses "uboot" for the correct Raspberry Pi are replaced by a single kernel that uses "vmlinuz," and the line "initramfs initrd.img followkernel" was added. Another modification was the removal of the line "device_tree_address=0x03000000". The last changes are "force_turbo=1", "dtoverlay=disable-bt", and "dtoverlay=miniuart-bt" inserted at the end of the file.

The IMU installation instructions from the pre-project, [19], have been updated and are included as attachment B in this thesis.

```
  GNU nano 4.8                /boot/firmware/config.txt
[pi4]
#kernel=uboot_rpi_4.bin

[pi2]
#kernel=uboot_rpi_2.bin

[pi3]
#kernel=uboot_rpi_3.bin

[pi0]
#kernel=uboot_rpi_3.bin

[all]
#device_tree_address=0x03000000

[pi4]
max_framebuffers=2
arm_boost=1

[all]
kernel=vmlinuz
initramfs initrd.img followkernel

# Enable the audio output, I2C and SPI interfaces on the GPIO header. As these
# parameters related to the base device-tree they must appear *before* any
# other dtoverlay= specification
dtparam=audio=on
dtparam=i2c_arm=on
dtparam=spi=on

# Comment out the following line if the edges of the desktop appear outside
# the edges of your display
disable_overscan=1
```

```
# If you have issues with audio, you may try uncommenting the following line
# which forces the HDMI output into HDMI mode instead of DVI (which doesn't
# support audio output)
#hdmi_drive=2

# Config settings specific to arm64
arm_64bit=1
dtoverlay=dwc2

[cm4]
# Enable the USB2 outputs on the IO board (assuming your CM4 is plugged into
# such a board)
dtoverlay=dwc2,dr_mode=host

[all]

# The following settings are "defaults" expected to be overridden by the
# included configuration. The only reason they are included is, again, to
# support old firmwares which don't understand the "include" command.

enable_uart=1
cmdline=cmdline.txt

include syscfg.txt
include usercfg.txt
start_x=1
gpu_mem=128

force_turbo=1

# Disable Bluetooth
dtoverlay=disable-bt
dtoverlay=miniuart-bt
```

Figure 4.5: The file /boot/firmware/config.txt, which contains all of the modifications

4.2.4 Ubuntu

Now that everything is compatible with both Ubuntu 21.10 and Ubuntu 20.04, it was decided to continue with Ubuntu 20.04 for the remainder of the thesis. This was due to the fact that Ubuntu 21.10 does not have long-term support and hence was not as widely used and tested as Ubuntu 20.04.

4.2.5 Gstreamer problems

Another issue that arose was that a basic installation of OpenCV was performed, and so a version of OpenCV that included gstreamer was not installed. This was a challenge since the camera intended for use in this thesis was a Sony camera, see the pre-project [19]. To make this camera operate, RTSP was required, and so gstreamer was required to make the camera function properly. Gstreamer includes

a variety of plugins that may be used to enhance the utilization of a video stream. These plugins are required for RTSP to function properly. The only method to make OpenCV utilize gstreamer was to remove the old OpenCV and then reinstall it with gstreamer. With this not being a straightforward or guaranteed solution, it was decided to try uninstalling and reinstalling OpenCV on the raspberry pi used for testing before carrying out the procedure on the original raspberry pi, which already has everything else installed and functioning as it should.

Initially an attempt to remove OpenCV was made, however, this proved to be far more complex than initially expected. This was due to the fact that a large number of additional directories and packages were automatically installed when OpenCV was installed, and these directories and packages were scattered across the system. All of this was not immediately removed when uninstalling OpenCV, and it was difficult to discover. After uninstalling OpenCV, an effort was made to remove and uninstall the remaining directories and packages. However, after reinstalling OpenCV, it appears that something from the previous OpenCV was still present. When OpenCV was reinstalled, everything throughout the installation indicated that the installation was completed smoothly and without errors. However, when OpenCV was imported using Python, the system was unable to locate it. The only way to solve this issue was to reinstall the system with a fresh Ubuntu 20.04 and then install OpenCV with gstreamer from the beginning. This seemed to make gstreamer function with OpenCV, but there were still a number of issues, including the fact that not all of the required plugins could be installed on an ARM64 system. Some alternative approaches were explored to remedy this problem, but because they were time consuming, it was determined the best path forward would be to replace the camera.

4.2.6 HQ camera and new test model

The Raspberry Pi's function to directly connect to the Raspberry Pi HQ camera, made it easy to use. The HQ camera must be enabled with raspi-config in order to function. Because Ubuntu 20.04 was used instead of the Raspberry Pi Operating System (OS), it was necessary to install raspi-config. This technique, as well as everything else done to make the Raspberry Pi HQ camera function, was documented in an installation guide, which is provided as attachment C.

When it was decided to switch to a new camera, certain adjustments to the test model were required, as shown in the pre-project [19]. Figure 4.6 depicts the upgraded test model, which now includes a new camera and a spirit level. To connect the camera to the Raspberry Pi, the Adafruit Perma-Proto Pi hat, that had been used to attach the IMU to the Raspberry Pi, had to be removed. This was due to the Pi hat blocking the input for the Raspberry Pi's camera; see attachment C for details on how the camera is attached to the Raspberry Pi. The IMU had to be attached directly to the Raspberry Pi because the Pi hat had been removed. To resolve this, jumper wires were soldered to the IMU wires and then connected to the proper Raspberry Pi pins. Figure 4.7 shows how the IMU and HQ Pi camera was connected to the raspberry pi without the Adafruit Perma-Proto Pi hat.

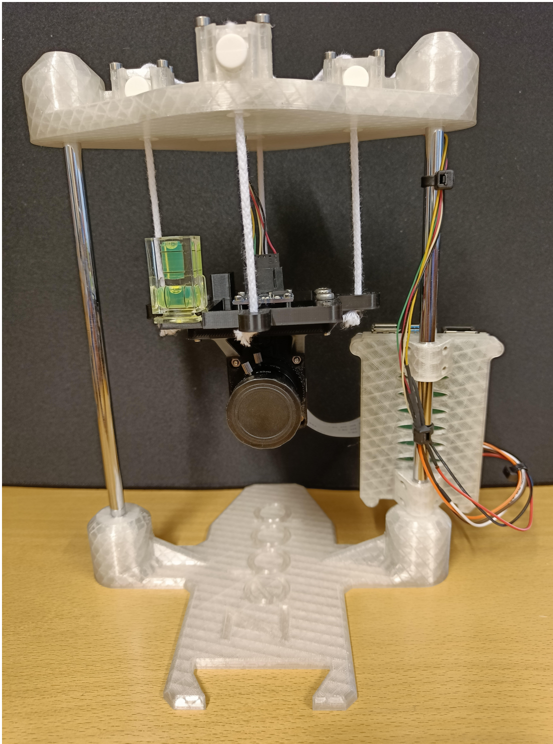


Figure 4.6: The test model with the raspberry Pi HQ camera installed

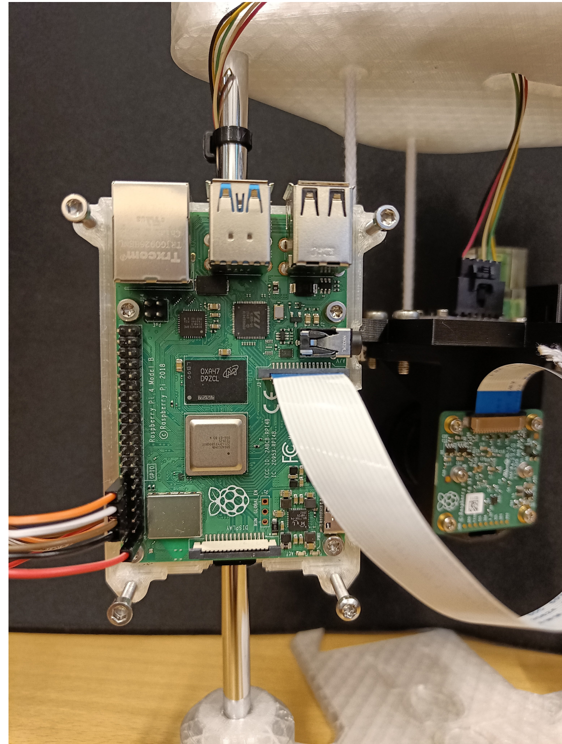


Figure 4.7: The connection of the IMU and camera to the raspberry Pi

4.3 Inertial measurement unit

The IMU code was originally tested separately before being included in the main code for this thesis. This section will provide and explain the essential parts of the IMU code that were tested and later incorporated into the main code. The first section of code that was written was the one that imports the BNO055 module from the Adafruit library. Code 1 demonstrates how the BNO055 module was imported as well as the configuration of the BNO055 sensor connection. It was important to set the serial port for the BNO055 sensor to be "/dev/ttyAMA0" in order for it to operate in the Raspberry Pi's serial Universal Asynchronous Receiver/Transmitter (UART) mode. Additionally, it was essential to set the reset function to be configured to utilize the relevant pin. This was the pin to which the IMU's reset function was connected to the Raspberry Pi, and in this thesis, the reset function is connected to the Raspberry Pi's General Purpose Input/Output (GPIO)18.

```
1   # Import BNO055
2   from Adafruit_BNO055 import BNO055
3
4   # Create and configure the BNO055 sensor connection.
5   bno = BNO055.BNO055(serial_port='/dev/ttyAMA0', rst=18)
```

Code 1: Python code for importing BNO055 module and connecting the sensor

Code 2 demonstrates how the calibration file from section 3.2.2 was used. After opening the calibration file, the functions bno.set_calibration and bno.get_calibration were utilized. The set_calibration function sets the IMU calibration settings, while the get_calibration function reads the gyroscope, accelerometer, magnetometer, and system calibration values. This function returns a value between zero and three for each sensor, with zero indicating that the sensor is uncalibrated and three indicating that the sensor is fully calibrated.

```
1    # Calibration file
2    CALIBRATION_FILE = 'calibration.json'
3
4    # Load calibration from disk.
5    with open(CALIBRATION_FILE, 'r') as cal_file:
6        data = json.load(cal_file)
7
8    # Set the IMU calibration
9    bno.set_calibration(data)
10
11   while True:
12       # Read the calibration status,
13       # 0=uncalibrated and 3=fully calibrated.
14       sys, gyro, accel, mag = bno.get_calibration_status()
```

Code 2: Python code for loading the calibration file

Code 3 shows the following section of the code. This section is used to read the various sensors, the code presented here displays more sensor values than were utilized. Only the sensor data from the linear accelerometer and gyroscope was utilized in the main code since these were the sensor values required by the EKF. This code demonstrates that the IMU is capable of providing sensor readings such as temperature in $^{\circ}C$, accelerometer in m/s^2, linear accelerometer in m/s^2, current gravity accelerometer in m/s^2, gyroscope (angular velocity) in rad/s, magnetometer in μT, quaternions, and Euler angles in degrees. Linear acceleration is the acceleration caused by movement rather than gravity.

```
1    # Grab new BNO055 sensor readings.
2    # Read the temperature in °C
3    temp = bno.read_temp()
4
5    # Read the accelerometer for x, y and z
6    # in m/s^2
7    ax, ay, az = bno.read_accelerometer()
8
9    # Read the linear acceleration for x, y and z
10   # in m/s^2
11   ax, ay, az = bno.read_linear_acceleration()
12
13   # Read the current gravity acceleration for x, y and z
14   # in m/s^2
15   gax, gay, gaz = bno.read_gravity()
```

```
16
17      # Read the gyroscope (angular velocity) for
18      # x, y and z in rad/s
19      gx, gy, gz = bno.read_gyroscope()
20
21      # Read the magnetometer for x, y and z in
22      # micro-Teslas
23      mx, my, mz = bno.read_magnetometer()
24
25      # Read the Quaternions for x, y, z and w
26      qx, qy, qz, qw = bno.read_quaternion()
27
28      # Read the Euler angles for heading, roll, pitch
29      # in degrees.
30      heading, roll, pitch = bno.read_euler()
```

Code 3: Python code for reading the BNO055 sensor

The final section of the IMU code demonstrates how the linear acceleration vector and gyroscope vector were converted to acceleration and angular velocity. The system's velocity was computed by adding the initial velocity to the acceleration multiplied by the time interval. The velocity will have a small inaccuracy since the acceleration must be time-integrated to obtain velocity. If the resultant velocity estimate is time-integrated again to obtain a position estimate, the inaccuracy grows quadratically with time. This means that utilizing acceleration to predict position will provide some challenges. This is why using an IMU alone to determine the camera's position underwater is not a suitable option.

```
1       # Acceleration vector
2       array_a = array([ax,ay,az])
3       # Gyroscope vector
4       array_g = array([gx,gy,gz])
5
6       # Acceleration [m/s^2]
7       a = norm(array_a,2)
8       # Velocity[m/s]
9       v = u + a*delta_t
10      # Angular velocity [rad/s]
11      om = norm(array_g,2)
```

Code 4: Python code for transforming BNO055 readings to acceleration, velocity and angular velocity

Figure 4.8 depicts the sensor output during testing. Despite the fact that the calibration value was three on all three sensors, the calibration was not always as precise. When the IMU was stationary and level, the acceleration in the z-axis was $9.52m/s^2$, rather than $9.81m/s^2$ as expected owing to gravitation. This shows that the calibration was not as successful as expected, and it is thus critical that all calibrations are conducted correctly. After a few seconds of executing the code, the magnetometer calibration was reset to zero. This was to be expected given the magnetometer calibration's high dynamic range. As a result, each time the IMU code is executed, the

camera with the IMU attached must be moved in an infinity pattern, as described in section 3.2.2. The system calibration began at zero, as shown in fig. 4.8, since the magnetometer calibration was not calibrated. After the magnetometer calibration, the system calibration was complete.

```
System status: 5
Self test result (0x0F is normal): 0x0F
Software version:   785
Bootloader version: 21
Accelerometer ID:   0xFB
Magnetometer ID:    0x32
Gyroscope ID:       0x0F

Reading BNO055 data, press Ctrl-C to quit...
Heading=0.00 Roll=0.00 Pitch=0.00
Acceleration: x=0.28 y=-0.04 z=9.52
Linear acceleration: x=0.00 y=0.00 z=0.00
Gyroscope: x=-0.00 y=-0.00 z=0.01
Magnetometer: x=23.56 y=88.06 z=-66.25
Temperature=26.0
Sys_cal=0 Gyro_cal=3 Accel_cal=3 Mag_cal=3
```

Figure 4.8: IMU sensor readings output

The system calibration was properly calibrated once the magnetometer was recalibrated. The accelerometer calibration also decreased to zero after the magnetometer was recalibrated and the code had been running for some time. This indicates that the accelerometer calibration was not completed correctly, despite the fact that the calibration value was set to three. The previously reported z value remained at $9.52m/s^2$ after the calibration value was decreased. The z value climbed to $9.79m/^2$ once the accelerometer was recalibrated, and $9.81m/s^2$ after the camera was level in all directions. Because the z-axis is pointing down from the IMU center, a slight angle will affect the z-axis value. It is also necessary to take into consideration the fact that gravitational pull is not constant and decreases as it gets farther from the center of the Earth. In fact, it is not even a constant at the surface, varying from $9.8m/s^2$ at the poles to $9.78m/^2$ at the equator. Figure 4.9 depicts the sensor output after recalibration of the magnetometer and accelerometer.

```
Heading=56.75 Roll=1.44 Pitch=-3.19
Acceleration: x=0.26 y=0.50 z=9.81
Linear acceleration: x=0.01 y=-0.05 z=0.02
Gyroscope: x=-0.00 y=0.00 z=0.00
Magnetometer: x=-31.25 y=15.75 z=-63.06
Temperature=27.0
Sys_cal=3 Gyro_cal=3 Accel_cal=3 Mag_cal=3
```

Figure 4.9: IMU sensor readings after recalibration

With the proper calibration, the IMU provided reliable and accurate positioning and orientation measurements for a period of time. As predicted, the measurements began to drift after the IMU had been operational for a while. When utilized alone, this drifting makes the IMU's position and orientation measurements unreliable. When the system is utilized underwater, the IMU must be recalibrated since environmental factors such as temperature, pressure, depth, and so on will make the calibration unreliable. Furthermore, the findings of the trials suggest that adequate calibration is required to obtain trustworthy readings. As a result, the first time the IMU is exposed to water, it must be calibrated, and that calibration can be saved for future use.

4.4 AprilTag

The AprilTag code, like the IMU, was tested separately before being incorporated into the main code for this thesis. The implemented AprilTag code is based on examples from [21], [24] and [42]. This section will explain how the camera was calibrated, how AprilTag was implemented, and how the calibration parameters were used to determine the pose estimation of the tags. It will go through the important parts of the codes and explain what they do.

4.4.1 Calibration

The checkerboard, shown in section 3.3, had to be photographed using the camera that would be calibrated. Prior to taking images of the checkerboard, it was important to ensure that the checkerboard was flat on a surface and had a plain, preferably white, background. The camera was moved around while the photo was being taken to capture the checkerboard pattern from various distances and viewpoints. It was required that at least 10 photos of the checkerboard be taken [25], [26]. Code 5 demonstrates how the image was taken every 100th frame, allowing the operator to reposition the camera between shots. The code would capture 30 images before stopping, resulting in three times the number of pictures required. One of the 30 photos that were captured throughout this procedure is fig. 3.10.

```
1     # Extract picture every 100th frame
2     if i%100 == 0:
3         cv2.imwrite(os.path.join(path , 'image_%d.jpg')% num, frame)
4     i+=1
5     # Number of pictures taken
6     if num == 30:
7         break
```

Code 5: Python code for taking a picture with the camera

The calibration process involved the code iterating over each image of the checkerboard one at a time in search of a checkerboard pattern. Code 6 is the Python code that identifies and draws the checkerboard corners in all of the pictures.

The function **cv2.findChessboardCorners(gray, (10,7), None)** searches for inner checkerboard corners and determines whether or not the input image contains a checkerboard pattern. The number of inner corners in each checkerboard row and column, as well as a grayscale image source, are inputs for the function. Because the checkerboard has 11x8 squares, it has 10x7 internal corners, intersections where the black squares meet. It was important to use the proper checkerboard pattern size as input because if the size was incorrect, the function would fail to detect any checkerboards in the input picture. If all of the corners are located and are arranged in a specified order, going from left to right in each row, the function returns a non-zero number. However if the function is unable to detect or rearrange all of the corners, it returns zero [26], [43].

The corner coordinates that findChessboardCorners returns as output are simply approximations. The **cv2.cornerSubPix(gray, corners, (11,11), (-1,-1), criteria)** function was used to enhance this. The feature enhances the corner positions to offer better calibration results. It requires the checkerboard picture in grayscale as well as the corners identified by the findChessboardCorners function as inputs. To gather all of the equations into a single container, the given input results were added to a **imgpoints.append(corners)** vector [26], [44].

Finally, the detected corners were displayed on the input picture using the **cv2.drawChessboardCorners(img, (10,7), corners2, ret)** function. The target picture as well as the number of inner corners for each checkerboard row and column are inputs for this function. Additionally, it receives as inputs the parameters from findChessboardCorners indicating whether or not the entire board was located, as well as the array of the detected corners from cornerSubPix [26], [43].

```
1     # termination criteria
2     criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30,
      ↪  0.001)
3     # Find the chess board corners
4     ret, corners = cv2.findChessboardCorners(gray, (10,7), None)
5     # If found, add object points, image points (after refining them)
6     if ret == True:
7         objpoints.append(objp)
8         corners2=cv2.cornerSubPix(gray,corners, (11,11), (-1,-1),
          ↪  criteria)
9         imgpoints.append(corners)
10        # Draw and display the corners
11        cv2.drawChessboardCorners(img, (10,7), corners2, ret)
```

Code 6: Python code to identify and draw checkerboard corners in all pictures

Figure 4.10 illustrates what the output of the drawChessboardCorners function looks like. Here, all of the inner corners have been identified and are displayed with corner markers. Additionally, it demonstrates how each row's corners are drawn from left to right, row by row. This example's inner corner dimensions are 8x6, not the same as the checkerboard used in this project, which has an inner corner dimension of 10x7.

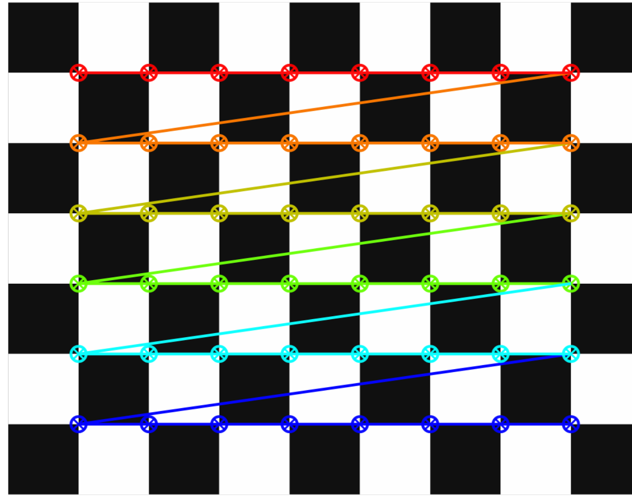


Figure 4.10: Detected checkerboard pattern, 8x6 corners [27]

The **cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None, None)** function was used to calibrate the camera once all the determined parameters from identifying the inner corners of the checkboard pattern were completed. This function takes the picture size, a vector of 2D points in the image plane, and a vector of 3D points in the real-world space as inputs. It returns the camera matrix together with rotation and translation vectors, distortion coefficients, and other data. Code 7 demonstrates how the calibrateCamera function was implemented in the code. The parameters for the focal length (f_x, f_y) and optical centers (c_x, c_y) were obtained from the camera matrix [25], [43].

```
1    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints,
     ↪  imgpoints, gray.shape[::-1], None, None)
2
3    # Printing the results
4    fx = mtx[0,0]
5    fy = mtx[1,1]
6    cx = mtx[0,2]
7    cy = mtx[1,2]
8
9    camera_params = (fx, fy, cx, cy)
```

Code 7: Python code for calibrating the camera

The output parameters from calibratedCamera were used to undistort fig. 3.10 in order to verify that the camera calibration functioned as intended. Code 8 demonstrates how the code's undistortion was implemented. Based on the free scaling parameter, the method **cv2.getOptimalNewCameraMatrix(mtx, dist, (w,h), 1, (w1,h1))** computes and returns the optimal new camera matrix. It takes as inputs the camera matrix, distortion coefficients, the size of the original picture, and the size of the new image.

This created a curved path, so it was possible to obtain the inputs for the **cv2.remap(img, mapx, mapy, cv2.INTER_LINEAR)** function by using the **cv2.initUndistortRectifyMap(mtx, dist, None, newcameramtx, (w1,h1), 5)** function to find a map from the distorted picture to the undistorted image. The initUndistortRectifyMap function takes as inputs the old camera matrix, distortion coefficients, the generated new camera matrix, and the new picture size.

```
1     h,  w = img.shape[:2]
2     newcameramtx, roi=cv2.getOptimalNewCameraMatrix(mtx, dist, (w,h),
      ↪  1, (w1,h1))
3
4     # undistort
5     mapx, mapy = cv2.initUndistortRectifyMap(mtx, dist, None,
      ↪  newcameramtx, (w1,h1), 5)
6     dst = cv2.remap(img, mapx, mapy, cv2.INTER_LINEAR)
7
8     # crop the image
9     x, y, w, h = roi
10    dst = dst[y:y+h, x:x+w]
11    cv2.imwrite('calibresult.png', dst)
```

Code 8: Python code for undistortion

After the remapping, it was feasible to generate a fresh, calibrated image that was not distorted. Figure 4.11 shows how the image appears after calibration. Now that all of the lines are straight and match the actual checkerboard pattern, the checkerboard pattern is no longer distorted.

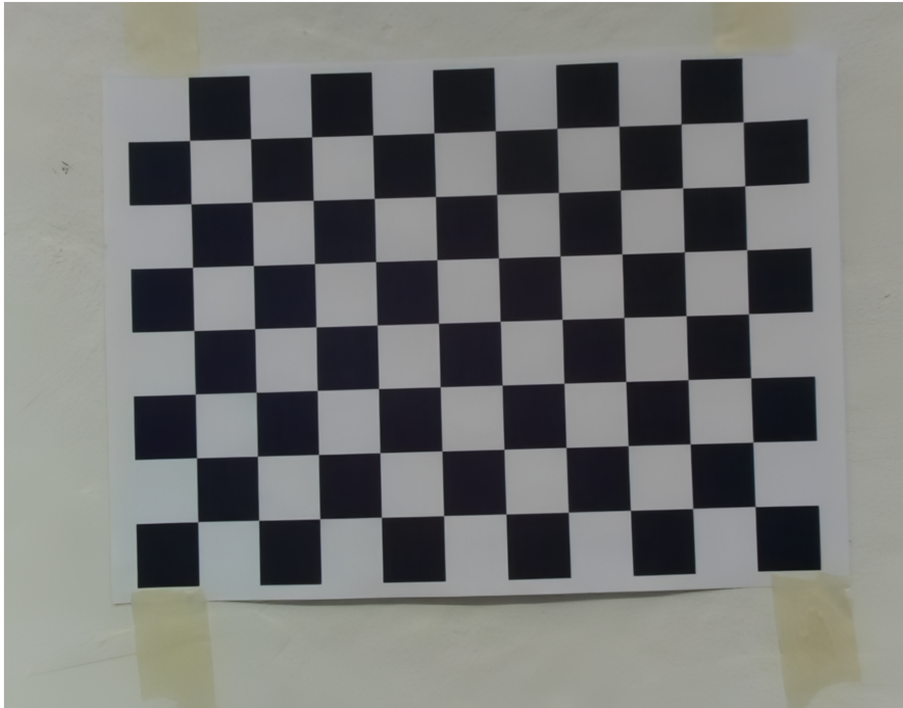


Figure 4.11: Undistorted picture after calibration

Using this data, the camera calibration was considered successful, and the camera parameters that were saved for future use are presented in eq. (4.1). When operated underwater, the system must be recalibrated. This is because variables such as low visibility and light conditions might impact the system, causing the calibration performed over water to no longer work properly.

$$f_x = 620.480 \tag{4.1a}$$
$$f_y = 619.309 \tag{4.1b}$$
$$c_x = 310.751 \tag{4.1c}$$
$$c_y = 235.317 \tag{4.1d}$$

4.4.2 Implementation

The AprilTag code was written to detect tags and give tag positions after the camera calibration. Code 9, the first part of the AprilTag code, shows the importation of the AprilTag library and the configuration of the AprilTag detector parameters. To detect AprilTags in an image, settings such as the AprilTag family must be specified. The AprilTag family used in this project is Tag36h11, as explained in section 3.3. The AprilTag detector command utilizes these options as inputs to detect AprilTags in the input image. The OpenCV library was also loaded in order to utilize the camera. Being directly connected to the Raspberry Pi through the camera input allowed the camera video stream to be opened using the OpenCV function VideoCapture(0).

```
1     # Import the necessary packages
2     import apriltag
3     import cv2
4
5     # Open Raspberry Pi camera
6     cam=cv2.VideoCapture(0)
7
8     # Define the AprilTag detector options and then detect the
      ↪  AprilTags
9     # in the input image
10    print("[INFO] detecting AprilTags...")
11    options = apriltag.DetectorOptions(families="tag36h11")
12    detector = apriltag.Detector(options)
```

Code 9: Python code for importing AprilTag module and configuring detector options

After importing the AprilTag library and configuring the detector, the input image needed to be preprocessed to meet the AprilTag requirements. Code 10 describes how the input image was preprocessed. The preprocessing involved converting the image to grayscale using the frame-by-frame readings from the input image. The image was also flipped since the camera was upside down after being installed on the new test model. In the OpenCV flip function, 0 indicates that the image has been flipped in the y-axes, 1 indicates that the image has been flipped in the x-axes, and -1 indicates that both axes have been flipped. Finally, the total number of detected AprilTags is saved in the variable "result".

```
1     # Capture frame-by-frame
2     _,frame=cam.read()
3
4     # Flip the video, 0 filp y, 1 flip x and -1 flip both
5     frame = cv2.flip(frame,0)
6
7     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
8     results = detector.detect(gray)
```

Code 10: Python code preprocessing the input video

With the detected AprilTags, it was easy to loop through the results and create bounding boxes, center coordinates, and tag families on the image. Code 11 demonstrates how the detected AprilTags are processed. The AprilTag algorithm loops over the detected results to get the (x, y)-coordinates for all AprilTag corners. The bounding box of the AprilTag detection was drawn using the known corners, and the center (x, y)-coordinates of each AprilTag detection were obtained and drawn as circles onto the image. The tag family where drawn on the image for each identified tag.

```
1       # Loop over the AprilTag detection results
2       for r in results:
3           # Extract the bounding box (x, y)-coordinates for the
4           # AprilTag
5           (ptA, ptB, ptC, ptD) = r.corners
6           ptA = (int(ptA[0]), int(ptA[1]))
7           ptB = (int(ptB[0]), int(ptB[1]))
8           ptC = (int(ptC[0]), int(ptC[1]))
9           ptD = (int(ptD[0]), int(ptD[1]))
10
11          # Draw the bounding box of the AprilTag detection
12          cv2.line(frame, ptA, ptB, (0, 255, 0), 2)
13          cv2.line(frame, ptB, ptC, (0, 255, 0), 2)
14          cv2.line(frame, ptC, ptD, (0, 255, 0), 2)
15          cv2.line(frame, ptD, ptA, (0, 255, 0), 2)
16
17          # Draw the center (x, y)-coordinates of the AprilTag
18          (cX, cY) = (int(r.center[0]), int(r.center[1]))
19          cv2.circle(frame, (cX, cY), 5, (0, 0, 255), -1)
20
21          # Draw the tag family on the image
22          tagFamily = r.tag_family.decode("utf-8")
23          cv2.putText(frame, tagFamily, (ptA[0], ptA[1] - 15),
24              cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
```

Code 11: Python code for looping through the AprilTag detection results

The final part of the AprilTag code determines the pose estimation of each detected AprilTag and is given in code 12. The detected AprilTags, camera parameters, and tag size are used as inputs for AprilTag pose detection. The camera parameters are the parameters (f_x, f_y, c_x, c_Y) obtained during the calibration mentioned in section 4.4.1. The tag size is the length of the tag's inside border in meters, as stated in section 3.3. In addition to the pose, the pose detection function returns the initial and final error values.

```
1           # Camera parameters from the calibration
2           camera_params = (fx, fy, cx, cy)
3
4           # The tag size in meters
5           tag_size = 0.124 # Must be change to the right tag size
6
7           # Position matrix for the tags, and initial and final error
             ↪   values
8           pose, e0, e1 = detector.detection_pose(r,
9                                                  camera_params,
10                                                 tag_size)
```

Code 12: Python code for positioning AprilTag

Figure 4.12 shows how the camera captures the AprilTags that was scattered around when the code was executed. It is clear from this picture that AprilTag is capable of detecting all tags, regardless of their placement or orientation. The tag that is the furthest away from the camera is placed 2.5m from the center of the camera. The AprilTag where capable of detecting tags from greater distances than 5m, but because of space constraints, it was not possible to test this more. The tags in this test were relatively large, but the maximum distance will vary depending on the size of the tags. The detecting distance will also be reduced when utilized underwater due to factors such as low visibility and light conditions.

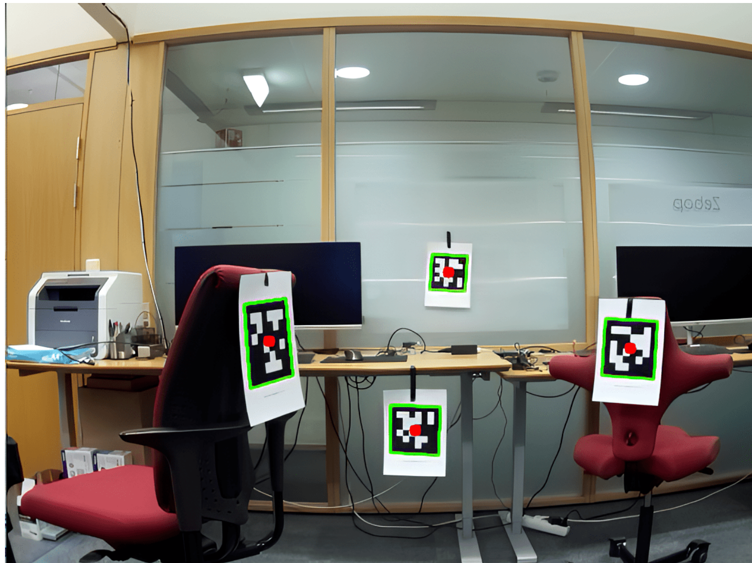


Figure 4.12: AprilTags detected with camera

Figure 4.13 depicts the output from the previous test. With tag-IDs ranging from 0 to 3, it is apparent that the code is detecting all four tags. The code also displays the tag family each tag belongs to, in this case all of the tags belong to tag36h11. Each tag receives a pose matrix, a rigid 4x4 transformation matrix. The translation vectors are the top values in the last column of the pose matrix. These translation vectors are measured in meters with respect to the camera frame, with the origin being in the center of the camera. Here, x is pointing to the right, y is pointing down, and z is pointing out the lens. By examining the z-values of the pose matrix for tag ID 1, it is possible to interpret that the calibration was successful because the z values are 2.5m, which corresponds to the distance measured between the tag and the camera center.

Finally, the function outputs the initial and final error, which corresponds to the reprojection error associated with a specific tag. The final error should be less than the initial error. It can be observed that tags placed at an angle relative to the camera have a larger initial error than tags placed more straightly. This is because the detected corners of tags positioned at an angle relative to the camera are more susceptible to noise.

```
[INFO] total AprilTags detected:  4
[INFO] tag family: tag36h11
[INFO] tag ID: 0
Pose [[-0.99595787  0.00614564 -0.08961113  0.01694792]
 [-0.01230894 -0.99758281  0.06838882 -0.35638015]
 [-0.08897423  0.0692154   0.99362609  1.86106466]
 [ 0.          0.          0.          1.        ]]
InitError 3.9695735330879756
FinalError 0.05668290793201446
[INFO] tag family: tag36h11
[INFO] tag ID: 1
Pose [[-0.99492592  0.04531813  0.08982586 -0.09155218]
 [-0.04968293 -0.99766017 -0.04696586  0.05664822]
 [ 0.08748727 -0.05119036  0.9948495   2.5155838 ]
 [ 0.          0.          0.          1.        ]]
InitError 0.7424073813668859
FinalError 0.013837545127215963
[INFO] tag family: tag36h11
[INFO] tag ID: 2
Pose [[-0.32708828 -0.07916447  0.94167205  0.23278842]
 [ 0.0427287  -0.9967047  -0.06894921 -0.07941035]
 [ 0.94402728  0.01768395  0.32939302  1.12506266]
 [ 0.          0.          0.          1.        ]]
InitError 29.323938538145285
FinalError 0.14552971788408173
[INFO] tag family: tag36h11
[INFO] tag ID: 3
Pose [[-0.70855472  0.09876747 -0.69870967 -0.44592013]
 [ 0.0124854  -0.98824668 -0.15235687 -0.12779278]
 [-0.70554541 -0.11667685  0.6989937   1.57184556]
 [ 0.          0.          0.          1.        ]]
InitError 5.077502416640412
FinalError 0.14690710589598646
```

Figure 4.13: AprilTags detected pose estimation output

This test examines the reliability of the AprilTag posture estimates. As a result, using the AprilTag as the ground truth in the EKF should produce adequate results. With a reliable date on positioning and orientation relative to the tags, the AprilTag can be a useful resource for the underwater positioning system. However, because problems such as low visibility and lightning conditions may make utilizing AprilTag underwater difficult, employing AprilTag alone may not be a good solution.

4.5 Extended Kalman filter

The EKF code was obtained from [45]. This code has been modified to meet the requirements of this project. This section will describe how the EKF's relevant aspects were implemented. The EKF code was written as a function and then called in the main code. The function takes AprilTag, IMU, and the initial positions as inputs and produces the updated state x_k and covariance matrices P_k. Code 13 demonstrates how the covariance noise matrices were calculated. Q_k denotes the input covariance noise matrix derived from the IMU, whereas R_k denotes the measurement covariance noise matrix obtained from AprilTag. For the EKF to achieve optimum performance, the diagonal values of these two matrices must be tuned.

```
1     # Covariance
2     v_var = 0.01 # Translation velocity variance
3     om_var = 0.1 # Rotational velocity variance
4     r_var = 0.01 # Range measurements variance
5     b_var = 10   # Bearing measurement variance
6 
7     # Nois covariance matrices
8     Qk = np.diag([v_var, om_var]) # Input nois from IMU
9     Rk = np.diag([r_var, b_var])  # Measurement nois from AprilTag
```

Code 13: Python code for covariance noise matrices

The prediction step is demonstrated in code 14 and is based on the EKF prediction model that is provided insection 3.4. To linearize the nonlinear system, the Jacobian has to be determined. The state and covariance estimates from the previous time step to the current time step are predicted using a for loop that loops from the initial prediction. When the loop is first executed, the initial values are utilized instead of updating the stat with the most recent odometry measurements. Using the IMU measurements as inputs, the predicted state estimate and estimate error covariance matrix were calculated using the derived motion model function and Jacobian with respect to the last state.

```
1     for k in range(1, len(t)):
2 
3         # Update state with odometry readings
4         if (flag_initial == 1):
5             #copy initial state values by taking first row
6             x_k = np.array(x_est[0, :]).reshape(3, 1)
7 
8             #copy initial covariance by taking first row
9             P_k = P_est[0]
10 
11        # Create the motion model function
12        A1 = np.array([[np.cos(wraptopi(theta)), 0],
         ↪  [np.sin(wraptopi(theta)), 0], [0, 1]], dtype='float')
13        A2 = np.array([[v[k-1]], [om[k-1]]])
14 
15        # Motion model Jacobian with respect to last state
16        F_km = np.zeros([3, 3])
17        F_km = np.array([[1, 0, -1 * delta_t * v[k-1] *
         ↪  np.sin(wraptopi(theta))], [0, 1, delta_t * v[k-1] *
         ↪  np.cos(wraptopi(theta))], [0, 0, 1]], dtype='float')
18 
19        # Predicted state estimate
20        x_k = x_k + delta_t * np.matmul(A1, A2)
21        x_k[2] = wraptopi(x_k[2])
22 
23        # Predicted estimate error covariance matrix
24        P_k = F_km*P_k*np.transpose(F_km) + Qk
```

Code 14: Python code for the prediction step

The measurement update step was written as a function that was later incorporated into the same loop as the prediction step. This function is seen in code 15, where it receives the tag AprilTag location data and the prediction model as inputs and returns the updated state and covariance estimates. The measurement update function computes the Jacobian transformation matrix H_k based on the predicted state estimations. The predicted estimating error covariance P_k^-, measurement covariance noise matrix R_k, and the transformation matrix H_k were used to calculate the Kalman gain K_k. The function could update the state estimate and the covariance matrix when the predicted state was adjusted.

```
1     def measurement_update(lk, rk, bk, P_k, x_k):
2         # Jacobian transformation matrix
3         H_k = np.array([[-d_x/range_exp, -d_y/range_exp, d *
          ↪  (d_x*np.sin(th) - d_y*np.cos(th))/range_exp],
4                         [d_y/frac, -d_x/frac, -1 - d * (np.sin(th)*d_y
                          ↪  + np.cos(th)*d_x)/frac]])
5         H_k = H_k.reshape(2, 3)
6
7         # Kalman Gain. Here is a 3x2 array
8         S_k = H_k*P_k*np.transpose(H_k) + Rk
9         K_k= P_k*np.transpose(H_k)*inv(S_k)
10
11        # Correct predicted state
12        phi = np.arctan2(d_y, d_x) - th
13        y_k = np.array([[range_exp], [wraptopi(phi)]])
14        y_k = y_k.reshape(2,1)
15        y_measured = np.array([[rk], [wraptopi(bk)]])
16
17        # Updated state estimate
18        x_k = x_k + K_k*(y_measured - y_k)
19        x_k[2] = wraptopi(x_k[2])
20
21        # Updated covariance matrix
22        P_k = (np.identity(3) - K_k*H_k)*P_k
```

Code 15: Python code for the measurement update step

A function called wraptopi was designed and is demonstrated in code 16. This function is used to wrap the angle in radians to the interval $[-\pi, \pi]$ such that π maps to π and $-\pi$ maps to $-\pi$. Positive π multiples of π are mapped to π, and negative π multiples of π are mapped to $-\pi$.

```
1     # Wraps angle to (-pi,pi] range
2     def wraptopi(x):
3         if x > np.pi:
4             x = x - (np.floor(x / (2 * np.pi)) + 1) * 2 * np.pi
5         elif x < -np.pi:
6             x = x + (np.floor(x / (-2 * np.pi)) + 1) * 2 * np.pi
7         return x
```

Code 16: Python code for wraping angle to (-pi,pi] range

The EKF integrating the AprilTag and IMU data has not been tested. This was due to the fact that so much time was spent in the beginning of the project attempting to get SLAM operating that the project was running out of time. What remains to be done to make EKF operate is to accurately apply the AprilTag data and tune the EKF. When this is done, it is easy to do tests by scattering AprilTags around an area while moving the camera around such that the camera may occasionally detect the tags while the IMU provides the location in between the tags. After some testing, it is feasible to create three-dimensional maps that illustrate the route that was taken. When doing so, the position from the IMU, AprilTag, and EKF may be plotted to see how well the EKF is doing. Since the AprilTag measurements by themselves produced satisfactory results, it is possible that the EKF may be a suitable option for positioning. The EKF has been tested on the IMU to check that everything functions as intended. This test produced encouraging findings. The IMU's drifting challenges are likely to be improved by utilizing the EKF, because incorporating the AprilTag data as ground truth allows the IMU position to be updated, reducing drifting.

There are going to be a lot of new issues when the system is utilized underwater since there are a lot more challenges to be overcome in that environment than there are in the indoor and outdoor testing environments. Some of the anticipated challenges have already been solved, while others will be addressed during the system's future underwater calibrations.

Chapter 5

Conclusion and Further work

Positioning underwater has many complicated and costly solutions because of the challenges with GPS devices not working underwater, as well as difficulties such as low visibility, lightning conditions, pressure and temperature. Solving the challenges of underwater positioning using a low-cost and simple system was attempted to be solved using a Raspberry Pi 4, a Raspberry Pi HQ camera, and a BNO055 IMU. Additionally, this thesis combined data of positioning measurements from an IMU and AprilTag with an EKF. The exact location of the camera in relation to the tags may be determined with the use of AprilTag. The problem was attempted to be solved using SLAM at the beginning of the thesis, but utilizing a Raspberry Pi 4 to implement this approach proved to be challenging. If less time had been spent at the beginning of the thesis on the SLAM challenges, the thesis would have had more time to address the issue using an IMU, AprilTag, and EKF. The likelihood that the underwater positioning issue can be resolved with the use of an EKF has increased with the IMU and AprilTag being operational and showing encouraging results. The EKF would have been done if there had been more time, allowing for the completion of more experiments and, therefore, more comprehensive results.

The challenges of positioning a camera underwater, such as the lack of GPS signals, limited visibility, lightning conditions, pressure, and temperature, can be overcome by employing an EKF, IMU, and AprilTag. This is because all of the outcomes to date have been encouraging. This means that the underwater positioning problem can be solved with low-cost technology. With the AprilTag measurements being accurate and reliable, it is possible to use the AprilTag measurements as ground truth in the EKF, which makes it possible to overcome the IMU drifting problem using sensor fusion.

5.1 Further work

This chapter discusses the effort required to get the system up and running, as well as what is required to get it working underwater. It will begin by going over the tasks that must be completed in order for the EKF to work and for the system to be tested above water. The chapter's conclusion will cover how to get the system ready for usage underwater, as well as what is anticipated to be required to get the system to work properly and provide accurate positioning data.

The AprilTag positioning must be accurately provided to make the EKF operational. Additionally, the EKF must be tuned for it to provide the most accurate pose estimate with the minimum level of noise. When the EKF is ready and tuned, the system may be tested by placing multiple tags around a certain area and moving the camera in between each tag. The IMU will then offer pose estimates between the tags, and AprilTags will provide correct pose estimates when it detects tags. Making plots of EKF, IMU, and AprilTag positioning measurements allows you to assess how well those three measures are in comparison to one another.

An underwater model that can house the Raspberry Pi HQ camera, the IMU, and the Raspberry Pi 4 must be constructed to operate the system underwater. It will be necessary for the system to have a cable that can supply power, although Power over Ethernet (PoE) can solve this. Use a Raspberry Pi PoE+ HAT that can supply up to 25W of power to a Raspberry Pi 4 at maximum load to make the Raspberry Pi function with PoE [46]. When utilizing PoE to power the device underwater, RTSP may be used to run the code and receive a visual image of what the camera is viewing. To be able to implement this solution, it is required to determine whether the Raspberry Pi can support it and whether all system testing can be completed via RTSP.

When the system is underwater, there are several difficulties that might arise, and it is thus advisable to be prepared for this. The system needs to be calibrated once again underwater because all previous calibrations performed above water were incorrect, which would reduce the system's reliability. Another problem to consider is that underwater visibility will be limited. As a result, tests must be conducted to determine the maximum distance at which the system can detect tags. The system may also be affected by lighting conditions, temperature, and pressure.

When the system is operating well, it may be utilized in a variety of situations. For example, it can be combined with a system that detects damage to fish farming equipment. Or it may be used to operate underwater robots or drones, such as those used to repair damage to oil platforms. In this case, the system can be utilized to determine both the position of the damage and the position of the robot or drone performing the repairs.

Bibliography

[1] Y. Wu, X. Ta, R. Xiao, Y. Wei, D. An, and D. Li, "Survey of underwater robot positioning navigation," *Applied Ocean Research*, 2019.

[2] D. A. Duecker, N. Bauschmann, T. Hansen, E. Kreuzer, and R. Seifried, "Towards micro robot hydrobatics: Vision-based guidance, navigation, and control for agile underwater vehicles in confined environments," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[3] L. Huang, B. He, and T. Zhang, "An autonomous navigation algorithm for underwater vehicles based on inertial measurement units and sonar," in *2010 2nd International Asia Conference on Informatics in Control, Automation and Robotics (CAR 2010)*, 2010.

[4] N. Kayhani, A. Heins, W. Zhao, M. Nahangi, B. McCabe, and A. P. Schoellig, "Improved Tag-based Indoor Localization of UAVs Using Extended Kalman Filter," *36th International Symposium on Automation and Robotics in Construction (ISARC 2019)*, 2019.

[5] Raspberry Pi, "Raspberry Pi High Quality Camera," [Online] https://www.raspberrypi.com/products/raspberry-pi-high-quality-camera/ [Searched 17/06/2022].

[6] RaspberryPi.dk, "Raspberry Pi High Quality Camera," [Online] https://raspberrypi.dk/produkt/raspberry-pi-hq-camera/ [Searched 16/06/2022].

[7] *Raspberry Pi High Quality Camera Getting started*, Raspberry Pi, April 2020.

[8] RaspberryPi.dk, "6mm Wide Angle Lens for Raspberry Pi HQ Camera," [Online] https://raspberrypi.dk/en/product/6mm-wide-angle-lens-raspberry-pi-hq-camera/ [Searched 16/06/2022].

[9] ArduCam, "High-resolution autofocus camera for Raspberry Pi," [Online] https://www.arducam.com/16mp-autofocus-camera-for-raspberry-pi/ [Searched 17/06/2022].

[10] B. Or, "What is imu?" [Online] https://towardsdatascience.com/what-is-imu-9565e55b44c [Searched 14/04/2022].

[11] Elprocus, "Imu sensor working and its applications," [Online] https://www.elprocus.com/imu-sensor-working-applications/ [Searched 14/04/2022].

[12] Vectornav, “What is an inertial measurement unit?” [Online] tinyurl.com/yc866d78 [Searched 14/04/2022].

[13] C. Pao, “What is an imu sensor?” [Online] https://www.ceva-dsp.com/ourblog/what-is-an-imu-sensor/ [Searched 14/04/2022].

[14] *BNO055 Intelligent 9-axis absolute orientation sensor*, Bosch Sensortec, June 2016, rev. 1.4.

[15] Arrow, “What is IMU? Inertial Measurement Unit Working & Applications,” [Online] https://www.arrow.com/en/research-and-events/articles/imu-principles-and-applications [Searched 10/01/2022].

[16] Adafruit, “Adafruit 9-DOF Absolute Orientation IMU Fusion Breakout - BNO055,” [Online] https://www.adafruit.com/product/2472?gclid=EAIaIQobChMIx_Lv1t7Y8gIVCHAYCh1U1gF1EAAYBCAAEgK72fD_BwE [Searched 21/01/2022].

[17] ——, “Types of SLAM and application examples,” [Online] https://learn.adafruit.com/adafruit-bno055-absolute-orientation-sensor [Searched 21/01/2022].

[18] T. DiCola, “BNO055 Absolute Orientation Sensor with Raspberry Pi & BeagleBone Black,” [Online] https://learn.adafruit.com/bno055-absolute-orientation-sensor-with-raspberry-pi-and-beaglebone-black/overview [Searched 31/05/2022].

[19] H. S. Mathisen, “Pre-project, Positioning a camera underwater,” 2021, Pre-project to Master’s thesis. Norwegian University of Science and Technology.

[20] E. Olson, “AprilTag,” [Online] https://april.eecs.umich.edu/software/apriltag [Searched 12/06/2022].

[21] A. Rosebrock, “AprilTag with Python,” [Online] https://pyimagesearch.com/2020/11/02/apriltag-with-python/ [Searched 20/06/2022].

[22] S. M. Abbas, S. Aslam, K. Berns, and A. Muhammad, “Analysis and Improvements in AprilTag Based State Estimation,” [Online] https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6960891/ [Searched 12/06/2022].

[23] Robotics Knowledgebase, “AprilTag,” [Online] https://roboticsknowledgebase.com/wiki/sensing/apriltags/ [Searched 15/06/2022].

[24] AprilRobotics, “AprilTag 3,” [Online] https://github.com/AprilRobotics/apriltag [Searched 20/06/2022].

[25] OpenCV, “Camera Calibration,” [Online] https://docs.opencv.org/3.3.1/dc/dbb/tutorial_py_calibration.html [Searched 12/06/2022].

[26] ——, “Camera calibration With OpenCV,” [Online] https://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html [Searched 13/06/2022].

[27] M. H. Jones, “Calibration Checkerboard Collection,” [Online] https://markhedleyjones.com/projects/calibration-checkerboard-collection [Searched 12/06/2022].

[28] H. Fåhraeus, “Fusion of IMU and monocular-SLAM in a loosely coupled EKF,” Master’s thesis, Linköping University, 2017.

[29] A. Zsíros, “Using Kalman filters for pose estimation of mobile devices in simulated underwater environments,” Bachelor’s thesis, Masaryk University, 2019.

[30] R. E. Kálmán, “A New Approach to Linear Filtering and Prediction Problems,” *Journal of basic Engineering*, 1960.

[31] A. Becker, “About the kalman filter,” [Online] https://www.kalmanfilter.net/default.aspx [Searched 16/06/2022].

[32] G. Welch and G. Bishop, “An Introduction to the Kalman Filter,” *University of North Carolina at Chapel Hill*, 2006.

[33] S. Zanj, “Extended kalman filter,” [Online] https://medium.com/@siddheshzanj/extended-kalman-filter-94fe07fd5c79 [Searched 18/06/2022].

[34] MathWorks, “What is SLAM?” [Online] https://se.mathworks.com/discovery/slam.html [Searched 08/05/2022].

[35] Geo SLAM, “What is SLAM?” [Online] https://geoslam.com/what-is-slam/ [Searched 08/05/2022].

[36] NavVis, “The definitive guide to SLAM & mobile mapping,” [Online] https://www.navvis.com/technology/slam [Searched 19/06/2022].

[37] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, “Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age,” *IEEE Transactions on Robotics*, 2016.

[38] T. V. Haavardsholm, “A handbook in visual slam,” 2021.

[39] C. Campos, R. Elvira, J. J. Gómez, J. M. M. Montiel, and J. D. Tardós, “ORB-SLAM3: An accurate open-source library for visual, visual-inertial and multi-map SLAM,” *arXiv preprint arXiv:2007.11898*, 2020.

[40] L. Freda, “pySLAM v2,” [Online] https://github.com/luigifreda/pyslam [Searched 27/04/2022].

[41] M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari, M. W. Achtelik, and R. Siegwart, “The euroc micro aerial vehicle datasets,” *The International Journal of Robotics Research*, 2016.

[42] swatbotics, “AprilTag,” [Online] https://github.com/swatbotics/apriltag [Searched 20/06/2022].

[43] OpenCV, "Camera Calibration and 3D Reconstruction," [Online] https://docs.opencv.org/3.3.1/d9/d0c/group__calib3d.html [Searched 14/06/2022].

[44] ——, "Feature Detection," [Online] https://docs.opencv.org/3.3.1/dd/d1a/group__imgproc__feature.html [Searched 14/06/2022].

[45] NekSfyris, "Ekf-lidar-odometry," [Online] https://github.com/NekSfyris/EKF-Lidar-Odometry [Searched 18/06/2022].

[46] E. Upton, "Announcing the raspberry pi poe+ hat," [Online] https://www.raspberrypi.com/news/announcing-the-raspberry-pi-poe-hat/ [Searched 24/06/2022].

Appendix A

Installation guide ORB-SLAM3

Installation of ORB-SLAM3 on the Raspberry Pi

This is an ORB-SLAM3 installation tutorial. The installation is carried out on a Raspberry Pi running Ubuntu 21.10. This installation guide also contains instructions on how to test the system after it has been installed.

Configuration

Before you can begin installing ORB-SLAM3, you must first complete certain system preparations. The first step is to ensure that the Raspberry Pi is up to date.

```
sudo apt update
sudo apt upgrade
```

Memory swapping

You should begin by providing Ubuntu more breathing room than a Raspberry Pi OS. The swap file prevents the Raspberry Pi from crashing.

1. Run the following command to ensure that the Raspberry Pi does not already have swap space:
   ```
   free -h
   ```
2. If you don't have enough swap space, create a 4GB swap file by running:
   ```
   sudo fallocate -l 4G /var/swapfile
   sudo chmod 600 /var/swapfile
   sudo mkswap /var/swapfile
   sudo swapon /var/swapfile
   sudo bash -c 'echo "/var/swapfile swap swap defaults,_netdev,x-initrd.mount  0 0" >>
   /etc/fstab'
   ```
3. Reboot by running the following command:
   ```
   sudo reboot
   ```
4. Now, check to see if the Raspberry Pi has swap space.
   ```
   free -h
   ```

   NOTE: If you see something like this, you need fix the swapfile.
   ```
   $ sudo mkswap /var/swapfile
   Setting up swapspace version 1, size = 1020 KiB
   no label, UUID=eeb4e9e0-386f-4fbd-919a-130e2c17079e
   ```

Installation of OpenCV

You may now begin installing OpenCV. This is a large installation that will take several hours to finish. See: https://qengineering.eu/install-ubuntu-20.04-on-raspberry-pi-4.html

1. The first step is to install all liberay requirements.
   ```
   sudo apt-get install build-essential cmake gcc g++ git unzip pkg-config
   sudo apt-get install libjpeg-dev libpng-dev libtiff-dev
   sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev
   sudo apt-get install libgtk2.0-dev libcanberra-gtk*
   sudo apt-get install libxvidcore-dev libx264-dev
   sudo apt-get install python3-dev python3-numpy python3-pip
   sudo apt-get install libtbb2 libtbb-dev libdc1394-22-dev
   ```

```
sudo apt-get install libv4l-dev v4l-utils
sudo apt-get install libopenblas-dev libatlas-base-dev libblas-dev
sudo apt-get install liblapack-dev gfortran libhdf5-dev
sudo apt-get install libprotobuf-dev libgoogle-glog-dev libgflags-dev
sudo apt-get install protobuf-compiler
```

2. Now, check your Memory; you'll need at least 6.5 GB! If you don't have enough Memory, increase your swap space as previously described.

```
free -m
```

3. Download the most recent version of OpenCV.

```
cd ~
wget -O opencv.zip https://github.com/opencv/opencv/archive/4.5.2.zip
wget -O opencv_contrib.zip https://github.com/opencv/opencv_contrib/archive/4.5.2.zip
```

4. Unzip the zip files

```
unzip opencv.zip
unzip opencv_contrib.zip
```

5. To make life simpler later on, run the following commands:

```
mv opencv-4.5.2 opencv
mv opencv_contrib-4.5.2 opencv_contrib
```

6. To build the OpenCV files, start with creating a directory:

```
cd ~/opencv
mkdir build
cd build
```

7. Now run build make

```
cmake -D CMAKE_BUILD_TYPE=RELEASE \ -D CMAKE_INSTALL_PREFIX=/usr/local \ -D
OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib/modules \ -D ENABLE_NEON=ON \ -D
BUILD_TIFF=ON \ -D WITH_FFMPEG=ON \ -D WITH_GSTREAMER=ON \ -D WITH_TBB=ON \ -D
BUILD_TBB=ON \ -D BUILD_TESTS=OFF \  -D WITH_EIGEN=OFF \ -D WITH_V4L=ON \ -D
WITH_LIBV4L=ON \ -D WITH_VTK=OFF \ -D OPENCV_ENABLE_NONFREE=ON \ -D
INSTALL_C_EXAMPLES=OFF \ -D INSTALL_PYTHON_EXAMPLES=OFF \ -D
BUILD_NEW_PYTHON_SUPPORT=ON \ -D BUILD_opencv_python3=TRUE \ -D
OPENCV_GENERATE_PKGCONFIG=ON \ -D BUILD_EXAMPLES=OFF ..
```

8. You are ready to build OpenCV. This process will take several hours.

```
make -j4
```

NOTE: If the building process fails before reaching 100% completion, repeat the cmake command (step 7) and execute the'make -j4' command again.

9. Install the libraries

```
sudo make install
sudo ldconfig
```

10. Do some cleaning (free 300 KB)

```
make clean
sudo apt-get update
```

11. To verify your Python 3 installation and the version of OpenCV that is installed, use the following command:
```
python3
>>> import cv2
>>> cv2.__version__
>>> exit()
```

Install Pangolin

You must additionally install Pangolin for visualization and user interface in order for the ORB-SLAM3 to function.

See: https://github.com/Mauhing/ORB_SLAM3/blob/master/README.md

1. To begin, install the required packages, you need to install “libglew-dev” for installation of Pangolin, “libboost-all-dev” for the DBoW, and “libssl-dev” for g2o. Run the following commands:
```
sudo apt-get install libglew-dev libboost-all-dev libssl-dev
sudo apt install libeigen3-dev
```
2. It is now time to install Pangolin; use the following commands to begin the installation:
```
cd ~/Dev
git clone https://github.com/stevenlovegrove/Pangolin.git
cd Pangolin
mkdir build
cd build
cmake .. -D CMAKE_BUILD_TYPE=Release
make -j 3
sudo make install
```

Installation of ORB-SLAM 3

Now that all of the criteria have been met, it is time to install ORB-SLAM3.

1. The first step is to clone the GitHub software for ORB-SLAM3, which may be done by running:
```
cd ~/Dev
git clone https://github.com/UZ-SLAMLab/ORB_SLAM3.git
cd ORB_SLAM3
```
2. Before you begin the installation, there may be certain issues that need to be addressed. If you proceed through the next stages and everything is in order, you can go ahead to step 4. To begin making modifications, open the header file "gedit./include/LoopClosing.h" with the command:
```
gedit ./include/LoopClosing.h
```
3. In order for this to compile, travel to line 51 and make the following change:
```
Eigen::aligned_allocator<std::pair<const KeyFrame*, g2o::Sim3> > > KeyFrameAndPose;
```
to
```
Eigen::aligned_allocator<std::pair<KeyFrame *const, g2o::Sim3> > > KeyFrameAndPose;
```
4. You may now compile ORB-SLAM3 and its dependencies, such as DBoW2 and g2o. To install, simply type:
```
./build.sh
```
NOTE: If you experience any issues, try running this shell script two or three more times.

NOTE: If you receive the following error:

```
In file included from /usr/local/include/pangolin/utils/signal_slot.h:3,
                 from /usr/local/include/pangolin/windowing/window.h:35,
                 from /usr/local/include/pangolin/display/display.h:34,
                 from /usr/local/include/pangolin/pangolin.h:38,
                 from /home/zebop/Dev/ORB_SLAM3/include/Map.h:27,
                 from /home/zebop/Dev/ORB_SLAM3/include/MapPoint.h:25,
                 from /home/zebop/Dev/ORB_SLAM3/include/KeyFrame.h:23,
                 from /home/zebop/Dev/ORB_SLAM3/include/LoopClosing.h:23,
                 from /home/zebop/Dev/ORB_SLAM3/src/LoopClosing.cc:20:
/usr/local/include/sigslot/signal.hpp:109:79: error: ‘decay_t’ is not a member of ‘std’; did you mean ‘decay’?
  109 | constexpr bool is_weak_ptr_compatible_v = detail::is_weak_ptr_compatible<std::decay_t<P>>::value;
      |                                                                               ^~~~~~~
      |                                                                               decay
/usr/local/include/sigslot/signal.hpp:109:79: error: ‘decay_t’ is not a member of ‘std’; did you mean ‘decay’?
  109 | constexpr bool is_weak_ptr_compatible_v = detail::is_weak_ptr_compatible<std::decay_t<P>>::value;
      |                                                                               ^~~~~~~
      |                                                                               decay
/usr/local/include/sigslot/signal.hpp:109:87: error: template argument 1 is invalid
  109 | constexpr bool is_weak_ptr_compatible_v = detail::is_weak_ptr_compatible<std::decay_t<P>>::value;
      |                                                                                       ^
```

You must make some adjustments to the "CMakeLlists.txt" file since it is attempting to run using "C++11," which is an outdated version. To make the installation work, you must update it so that it runs by using "C++14" instead of "C++11" [3].

1. To convert "-std=++11" to "-std=c++14", go to the file "/home/username/Dev/ORB SLAM3/CMakeLists.txt", scroll down until you find something like the before picture, then make the changes so that it looks like the after picture (all the changes that were made are highlighted):

   Before:

```
15 # Check C++11 or C++0x support
16 include(CheckCXXCompilerFlag)
17 CHECK_CXX_COMPILER_FLAG("-std=c++11" COMPILER_SUPPORTS_CXX11)
18 CHECK_CXX_COMPILER_FLAG("-std=c++0x" COMPILER_SUPPORTS_CXX0X)
19 if(COMPILER_SUPPORTS_CXX11)
20    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
21    add_definitions(-DCOMPILEDWITHC11)
22    message(STATUS "Using flag -std=c++11.")
23 elseif(COMPILER_SUPPORTS_CXX0X)
24    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x")
25    add_definitions(-DCOMPILEDWITHC0X)
26    message(STATUS "Using flag -std=c++0x.")
27 else()
28    message(FATAL_ERROR "The compiler ${CMAKE_CXX_COMPILER} has no C++11 support. Please use a
   different C++ compiler.")
29 endif()
```

   After:

```
15 # Check C++14 or C++0x support
16 include(CheckCXXCompilerFlag)
17 CHECK_CXX_COMPILER_FLAG("-std=c++14" COMPILER_SUPPORTS_CXX11)
18 CHECK_CXX_COMPILER_FLAG("-std=c++0x" COMPILER_SUPPORTS_CXX0X)
19 if(COMPILER_SUPPORTS_CXX11)
20    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++14")
21    add_definitions(-DCOMPILEDWITHC11)
22    message(STATUS "Using flag -std=c++14.")
23 elseif(COMPILER_SUPPORTS_CXX0X)
24    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x")
25    add_definitions(-DCOMPILEDWITHC0X)
26    message(STATUS "Using flag -std=c++0x.")
27 else()
28    message(FATAL_ERROR "The compiler ${CMAKE_CXX_COMPILER} has no C++14 support. Please use a
   different C++ compiler.")
29 endif()
```

2. Then re-run with the commands:

```
cd ~/Dev/ORB_SLAM3
./build.sh
```

Test the system

You can now start testing to ensure that the installation was successful.

Download test datasets

You must first download some test datasets before you can test the system. Execute the following commands:

```
cd ~
mkdir -p Datasets/EuRoc
cd Datasets/EuRoc/
wget -c http://robotics.ethz.ch/~asl-datasets/ijrr_euroc_mav_dataset/machine_hall/MH_01_easy/MH_01_easy.zip
mkdir MH01
unzip MH_01_easy.zip -d MH01/
```

If you're looking for more datasets in EuRoc, check out:
[https://projects.asl.ethz.ch/datasets/doku.php?id=kmavvisualinertialdatasets]

Run simulation

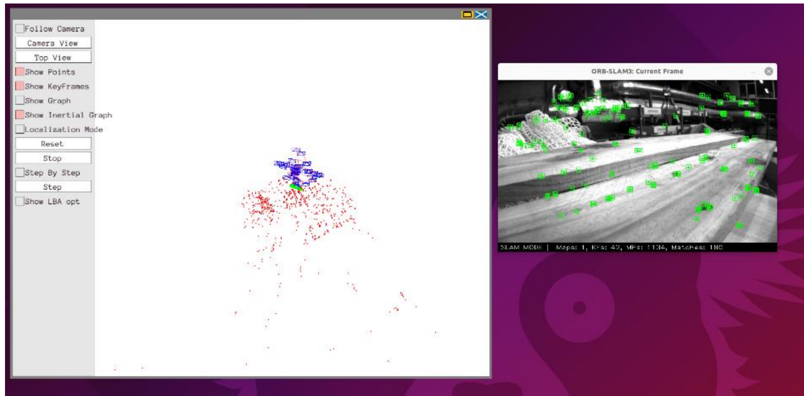
After downloading the test datasets, you are ready to begin a simulation. You have three different examples to choose from. You can run one of the examples to ensure that everything is working properly.

NOTE: When executing the examples, copy and execute all three lines at the same time

1. Begin by navigating to the correct directory.
   ```
   cd ~/Dev/ORB_SLAM3
   ```

   Choose one of them from the list below to run.
2. Mono
   ```
   ./Examples/Monocular/mono_euroc ./Vocabulary/ORBvoc.txt
   ./Examples/Monocular/EuRoC.yaml ~/Datasets/EuRoc/MH01
   ./Examples/Monocular/EuRoC_TimeStamps/MH01.txt dataset-MH01_mono
   ```
3. Mono + Inertial
   ```
   ./Examples/Monocular-Inertial/mono_inertial_euroc ./Vocabulary/ORBvoc.txt ./Examples/Monocular-Inertial/EuRoC.yaml ~/Datasets/EuRoc/MH01 ./Examples/Monocular-Inertial/EuRoC_TimeStamps/MH01.txt dataset-MH01_monoi
   ```
4. Stereo
   ```
   ./Examples/Stereo/stereo_euroc ./Vocabulary/ORBvoc.txt ./Examples/Stereo/EuRoC.yaml ~/Datasets/EuRoc/MH01 ./Examples/Stereo/EuRoC_TimeStamps/MH01.txt dataset-MH01_stereo
   ```
5. Stereo + Inertial
   ```
   ./Examples/Stereo-Inertial/stereo_inertial_euroc ./Vocabulary/ORBvoc.txt ./Examples/Stereo-Inertial/EuRoC.yaml ~/Datasets/EuRoc/MH01 ./Examples/Stereo-Inertial/EuRoC_TimeStamps/MH01.txt dataset-MH01_stereoi
   ```

6. You should receive something similar to the image below:



<u>Validation Estimate vs Ground Truth</u>

Now, compare the estimation to the ground truth to see how well the ORB-SLAM3 works.

1. Python3 requires numpy and matplotlib, thus the first thing you should do is install pip3 on python3 by running:
```
sudo apt install python3-pip
```
2. To check the version of pip3, use the following command:
```
pip3 --version
```
You should receive something like the following output:
```
pip 20.3.4 form /usr/lib/python3/dist-packages/pip (python3.9)
```
3. Now, execute the following commands to install numpy and matplotlib:
```
pip3 install numpy matplotlib
```
4. The plot script will need to be modified to operate with Python3 rather than Python2.7. The first step is to find the files that require modification. Files "home/username/Dev/ORB_SLAM3/evaluation/evaluate_ate_scale.py" and "home/username/Dev/ORB_SLAM3/evaluation/associate.py" must be modified.
5. Several adjustments must be made to the file "evaluate_ate_scale.py." Simply follow the before photo to go to the right location, then make the modifications shown in the after pictures.
6. On line 114, replace "stamp.sort()" to "sorted(stamps)," as seen below:

<u>Before:</u>
```
114     stamps.sort()
```
<u>After:</u>
```
114     #stamps.sort()
115     sorted(stamps)
```

7. Change all "x.keys()" and "x.sort()" to "list(x)" and "sorted(x)" in lines 168-173, as shown below:

<u>Before:</u>

```
168     first_stamps = first_list.keys()
169     first_stamps.sort()
170     first_xyz_full = numpy.matrix([[float(value) for value in first_list[b][0:3]] for b in first_stamps]).transpose()
171
172     second_stamps = second_list.keys()
173     second_stamps.sort()
```

<u>After:</u>

```
168     #first_stamps = first_list.keys()
169     first_stamps = list(first_list)
170     #first_stamps.sort()
171     sorted(first_stamps)
172     first_xyz_full = numpy.matrix([[float(value) for value in first_list[b][0:3]] for b in first_stamps]).transpose()
173
174     #second_stamps = second_list.keys()
175     second_stamps = list(second_list)
176     #second_stamps.sort()
177     sorted(second_stamps)
```

8. The final step in this file is to include parenthesis in all "print" functions, as seen below:

<u>Before:</u>

```
176     if args.verbose:
177         print "compared_pose_pairs %d pairs"%(len(trans_error))
178
179         print "absolute_translational_error.rmse %f m"%numpy.sqrt(numpy.dot(trans_error,trans_error) / len(trans_error))
180         print "absolute_translational_error.mean %f m"%numpy.mean(trans_error)
181         print "absolute_translational_error.median %f m"%numpy.median(trans_error)
182         print "absolute_translational_error.std %f m"%numpy.std(trans_error)
183         print "absolute_translational_error.min %f m"%numpy.min(trans_error)
184         print "absolute_translational_error.max %f m"%numpy.max(trans_error)
185         print "max idx: %i" %numpy.argmax(trans_error)
186     else:
187         # print "%f, %f " % (numpy.sqrt(numpy.dot(trans_error,trans_error) / len(trans_error)),  scale)
188         # print "%f,%f" % (numpy.sqrt(numpy.dot(trans_error,trans_error) / len(trans_error)),  scale)
189         print "%f,%f,%f" % (numpy.sqrt(numpy.dot(trans_error,trans_error) / len(trans_error)), scale, numpy.sqrt(numpy.dot(trans_errorGT,trans_errorGT) / len(trans_errorGT)))
190         # print "%f" % len(trans_error)
191     if args.verbose2:
192         print "compared_pose_pairs %d pairs"%(len(trans_error))
193         print "absolute_translational_error.rmse %f m"%numpy.sqrt(numpy.dot(trans_error,trans_error) / len(trans_error))
194         print "absolute_translational_errorGT.rmse %f m"%numpy.sqrt(numpy.dot(trans_errorGT,trans_errorGT) / len(trans_errorGT))
```

<u>After:</u>

```
181     if args.verbose:
182         print ("compared_pose_pairs %d pairs"%(len(trans_error)))
183
184         print ("absolute_translational_error.rmse %f m"%numpy.sqrt(numpy.dot(trans_error,trans_error) / len(trans_error)))
185         print ("absolute_translational_error.mean %f m"%numpy.mean(trans_error))
186         print ("absolute_translational_error.median %f m"%numpy.median(trans_error))
187         print ("absolute_translational_error.std %f m"%numpy.std(trans_error))
188         print ("absolute_translational_error.min %f m"%numpy.min(trans_error))
189         print ("absolute_translational_error.max %f m"%numpy.max(trans_error))
190         print ("max idx: %i" %numpy.argmax(trans_error))
191     else:
192         # print ("%f, %f " % (numpy.sqrt(numpy.dot(trans_error,trans_error) / len(trans_error)),  scale))
193         # print ("%f,%f" % (numpy.sqrt(numpy.dot(trans_error,trans_error) / len(trans_error)),  scale))
194         print ("%f,%f,%f" % (numpy.sqrt(numpy.dot(trans_error,trans_error) / len(trans_error)), scale, numpy.sqrt(numpy.dot(trans_errorGT,trans_errorGT) / len(trans_errorGT))))
195         # print ("%f" % len(trans_error))
196     if args.verbose2:
197         print ("compared_pose_pairs %d pairs"%(len(trans_error)))
198         print ("absolute_translational_error.rmse %f m"%numpy.sqrt(numpy.dot(trans_error,trans_error) / len(trans_error)))
199         print ("absolute_translational_errorGT.rmse %f m"%numpy.sqrt(numpy.dot(trans_errorGT,trans_errorGT) / len(trans_errorGT)))
```

9. You may now head over to the "associate.py" file and make the necessary adjustments. On this file, only one update is required. All of the "x_lists.keys()" must be replaced with "list(x_list)", as illustrated below:

<u>Before:</u>

```
88      first_keys = first_list.keys()
89      second_keys = second_list.keys()
```

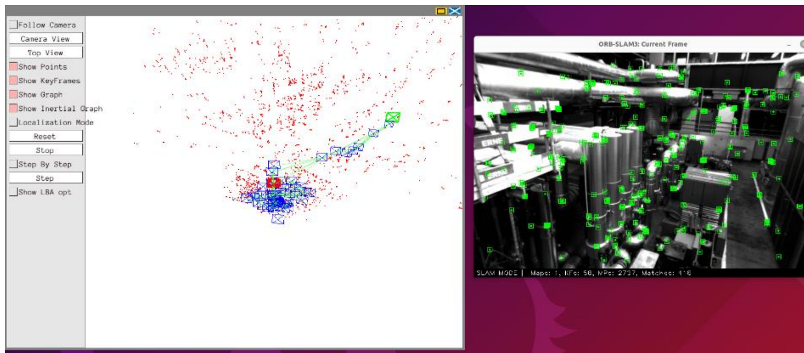
<u>After:</u>

```
88      #first_keys = first_list.keys()
89      first_keys = list(first_list)
90      #second_keys = second_list.keys()
91      second_keys = list(second_list)
```

10. Now that you're ready to execute the examples, use the following command:
    NOTE: For "./example..." to "...MH01 stereo," you must execute everything at the same time.

```
cd ~/Dev/ORB_SLAM3
./Examples/Stereo/stereo_euroc ./Vocabulary/ORBvoc.txt ./Examples/Stereo/EuRoC.yaml
~/Datasets/EuRoc/MH01 ./Examples/Stereo/EuRoC_TimeStamps/MH01.txt dataset-
MH01_stereo
```

    You should now see something like the image below:



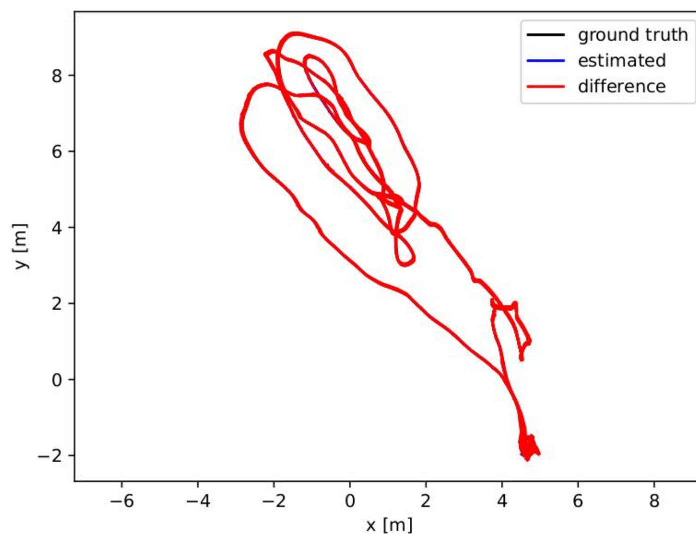
11. Plot the estimate against the ground truth

```
cd ~/Dev/ORB_SLAM3
python3 evaluation/evaluate_ate_scale.py
evaluation/Ground_truth/EuRoC_left_cam/MH01_GT.txt f_dataset-MH01_stereo.txt --plot
MH01_stereo.pdf
```

12. Run the following instructions to open the document "MHO1 stereo.pdf":

```
ls
evince MH01_stereo.pdf
```

    You will obtain a plot like the one shown below:



You can see here that the ORB-SLAM3 is following the ground truth really well.

Sources

[1] Q-enginerring, “Install Ubuntu 20.04 + OpenCV + TensorFlow (Lite) on Raspberry Pi 4,” [Online] https://qengineering.eu/install-ubuntu-20.04-on-raspberry-pi-4.html [Hentet 04/02/2022]

[2] M. Yip, “Installation guide by Mauhing Yip,” [Online] https://github.com/Mauhing/ORB_SLAM3/blob/master/README.md [Hentet 04/02/2022]

[3] egdw, “ORB_SLAM3_Ubuntu20.04,” [Online] https://github.com/egdw/ORB_SLAM3_Ubuntu20.04 [Hentet 04/02/2022]

Appendix B

Installation guide IMU

Installation of the IMU on the Raspberry Pi

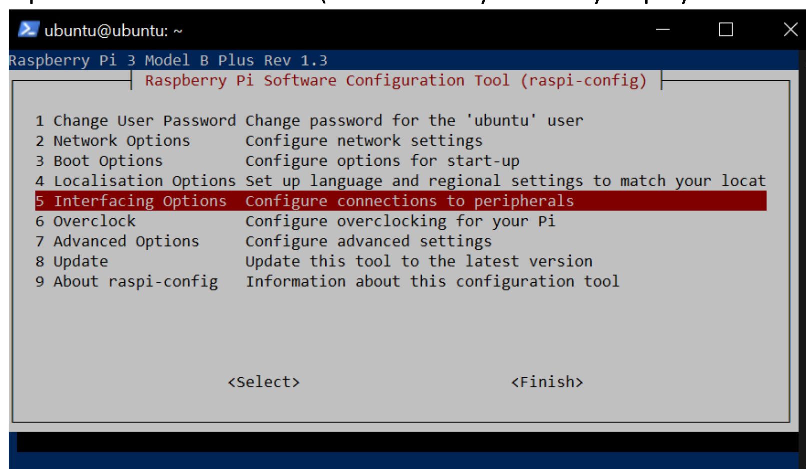
This installation guide is made for the Adafruit BNO055 IMU. Before you start with the installation, make sure to do the wiring between the IMU and the Raspberry Pi correctly. See the link below to get information on the wiring and how to set up the Raspberry Pi to connect to the IMU:
https://learn.adafruit.com/bno055-absolute-orientation-sensor-with-raspberry-pi-and-beaglebone-black/overview

Configuration

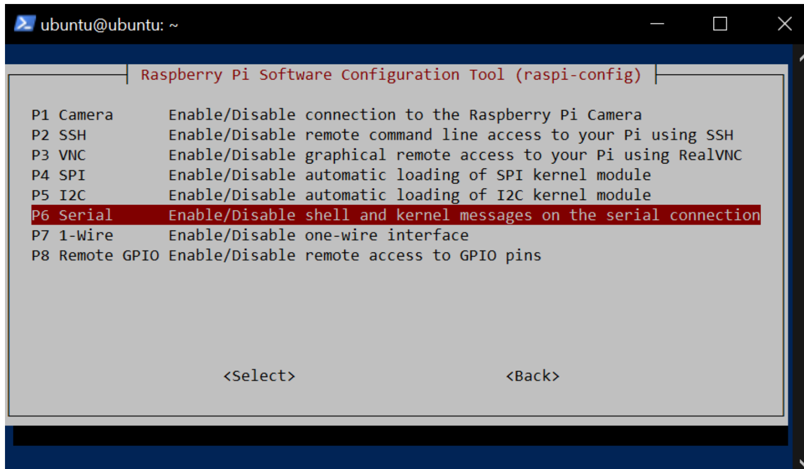
https://waldorf.waveform.org.uk/2021/you-boot-no-u-boot-first.html (Ubuntu 20.04)

On the Raspberry Pi, various preparations must be made before the IMU can be installed and used. Because the IMU is linked to the Raspberry Pi through serial UART mode, you must disable the kernel's use of the Pi's serial port. The Pi kernel will normally put a login terminal on the serial port when it starts up, but if you attach a device like the IMU to the serial port, the login terminal will confuse it. However, using the raspi-config tool, you can deactivate the kernel's usage of the serial port.

1. First you need to install the “raspi-config” tool on your Ubuntu. To install “raspi-config” run:
   ```
   wget https://archive.raspberrypi.org/debian/pool/main/r/raspi-config/raspi-config_20200601_all.deb -P /tmp
   sudo apt-get install libnewt0.52 whiptail parted triggerhappy lua5.1 alsa-utils -y
   sudo apt-get install -fy
   sudo dpkg -i /tmp/raspi-config_20200601_all.deb
   ```
2. To make the "raspi-config" work, create a symbolic link between the files "/boot/firmware/cmdline.txt" and "/boot/cmdline.txt," by execute the command:
   ```
   sudo ln -s /boot/firmware/cmdline.txt /boot/cmdline.txt
   ```
3. Now that you're ready to utilize the "raspi-config" tool to deactivate the kernel serial port, run:
   ```
   sudo raspi-config
   ```
4. You should now see a window similar to the one shown below. Scroll down to "5 Interfacing Options" and hit "ENTER." (NOTE: Your system may display this differently)

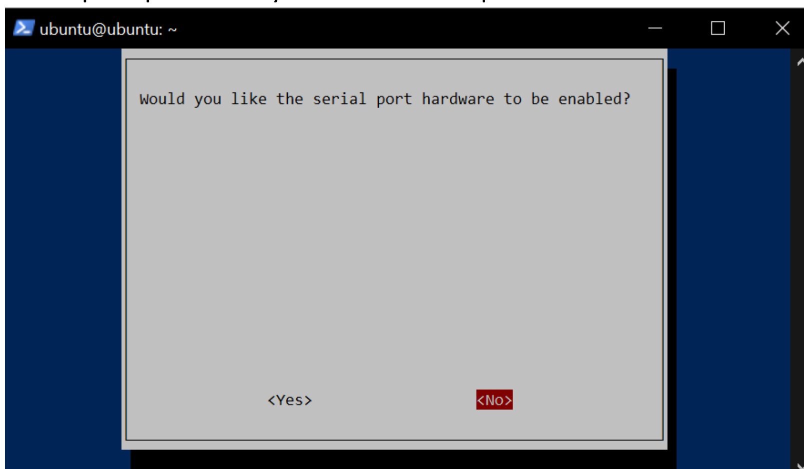


5. Scroll down to “P6 Serial” and press “ENTER”



6. When prompted “Would you like a login shell to be accessible over serial?” select “No”

7. When prompt “Would you like the serial port hardware to be enabled?” select “Yes”



8. Now finish the raspi-config by going down to “Finish” and press “ENTER”
9. Reboot the system by running:

```
sudo reboot
```

10. Because the IMU installation is a Python-module that interfaces with the GPIO header, this module is only for Python 2.7. To enable Python3, you must first install the RPi.GPIO module for Python3. To install the RPi.GPIO module for Python3, use the following command:
```
sudo apt-get install python3-rpi.gpio
```

Installation

Now that you've completed all the necessary preparations, the system should be ready to install the IMU.

1. The first step is to connect the IMU to the Raspberry Pi if you haven't previously.
2. When everything is ready, use the following commands to install the essential dependencies:
```
sudo apt-get update
sudo apt-get install -y build-essential python3-dev python3-smbus python3-pip git
```
3. Then execute the instructions below to get the most recent version of the BNO055 Python module code from GitHub:
```
cd ~
git clone https://github.com/adafruit/Adafruit_Python_BNO055.git
cd Adafruit_Python_BNO055
sudo python3 setup.py install
```
NOTE: If the installation fails with an error message, double-check that the requirements listed above were installed before proceeding. Also, ensure that your board is connected to the internet, since the installation will download and install several Python modules that are necessary.
4. You will need to do some changes to the config.txt file, run:
```
sudo nano /boot/config.txt
```
5. Add the following to the bottom of the file:
```
force_turbo=1
# Disable Bluetooth
dtoverlay=disable-bt
dtoverlay=miniuart-bt
```
6. And you need to do some changes to the cmdline.txt file, run
```
sudo nano /boot/firmware/cmdline.txt
```
7. Find the following text and remove it:
```
console=serial0,115200
```
8. You will also need to do some changes to the config.txt file in firmware, run:
```
sudo nano /boot/firmware/config.txt
```
9. Make the changes so it looks like:

```
  GNU nano 4.8
[pi4]
#kernel=uboot_rpi_4.bin

[pi2]
#kernel=uboot_rpi_2.bin

[pi3]
#kernel=uboot_rpi_3.bin

[pi0]
#kernel=uboot_rpi_3.bin

[all]
#device_tree_address=0x03000000

[pi4]
max_framebuffers=2
arm_boost=1

[all]
kernel=vmlinuz
initramfs initrd.img followkernel

# Enable the audio output, I2C and SPI interfaces on the GPIO header. As these
# parameters related to the base device-tree they must appear *before* any
# other dtoverlay= specification
dtparam=audio=on
dtparam=i2c_arm=on
dtparam=spi=on

# Comment out the following line if the edges of the desktop appear outside
# the edges of your display
disable_overscan=1

# If you have issues with audio, you may try uncommenting the following line
# which forces the HDMI output into HDMI mode instead of DVI (which doesn't
# support audio output)
#hdmi_drive=2

# Config settings specific to arm64
arm_64bit=1
dtoverlay=dwc2

[cm4]
# Enable the USB2 outputs on the IO board (assuming your CM4 is plugged into
# such a board)
dtoverlay=dwc2,dr_mode=host
[all]

# The following settings are "defaults" expected to be overridden by the
# included configuration. The only reason they are included is, again, to
# support old firmwares which don't understand the "include" command.

enable_uart=1
cmdline=cmdline.txt

include syscfg.txt
include usercfg.txt
start_x=1
gpu_mem=128

force_turbo=1
# Disable Bluetooth
dtoverlay=disable-bt
dtoverlay=miniuart-bt
```

10. Now reboot the system:

```
sudo reboot
```

You're ready to use the module after it's been installed.

Test the system

To make sure everything is working how it is supposed to do, you can run a simple test name "simpletest.py". In this example gives you information about the heading, roll and pitch of the IMU, it will also print the calibration value of the system, gyroscope, accelerometer, and magnetometer.

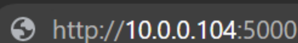
1. To run the example you need to navigate to the library "examples" by running
```
cd ~/Adafruit_Python_BNO055/examples
```
2. Depending on how the BNO055 sensor is attached to your board, you may need to adjust the code to initialize it before running the example. Run the following command to open the file in the nano text editor:
```
sudo nano simpletest.py
```
3. In the file, navigate with the down key, to the part that looks like the code shown below:
```
# Create and configure the BNO sensor connection.  Make sure only ONE of the
# below 'bno = ...' lines is uncommented:
# Raspberry Pi configuration with serial UART and RST connected to GPIO 18:
bno = BNO055.BNO055(serial_port='/dev/ttyAMA0', rst=18)
# BeagleBone Black configuration with default I2C connection (SCL=P9_19, SDA=P9_20),
# and RST connected to pin P9_12:
#bno = BNO055.BNO055(rst='P9_12')
```
4. Depending on how it's connected to your board, you'll want to leave only one of the bno =... lines uncommented, as mentioned in the comments. Leave this line uncommented for a Raspberry Pi that uses the serial port and GPIO 18 as the reset pin:
```
bno = BNO055.BNO055(serial_port='/dev/ttyAMA0', rst=18)
```
Adjust the serial port and rst parameters of the initializer if you're using a different serial port or GPIO for the reset line.
5. After you've finished editing the file, use Ctrl-S to save it, and then Ctrl-X to leave nano.
6. Now you are ready to run the file. Execute the following command as a root user with sudo:
```
sudo python3 simpletest.py
```
7. After a few seconds, if everything is operating well, you should see something like this:
```
System status: 5
Self test result (0x0F is normal): 0x0F
Software version:   776
Bootloader version: 21
Accelerometer ID:   0xFB
Magnetometer ID:    0x32
Gyroscope ID:       0x0F

Reading BNO055 data, press Ctrl-C to quit...
Heading=0.00 Roll=0.00 Pitch=0.00       Sys_cal=0 Gyro_cal=0 Accel_cal=0 Mag_cal=0
Heading=0.00 Roll=-0.69 Pitch=0.81      Sys_cal=0 Gyro_cal=3 Accel_cal=0 Mag_cal=0
Heading=0.00 Roll=-0.69 Pitch=0.81      Sys_cal=0 Gyro_cal=3 Accel_cal=0 Mag_cal=0
Heading=0.00 Roll=-0.69 Pitch=0.81      Sys_cal=0 Gyro_cal=3 Accel_cal=0 Mag_cal=0
Heading=0.00 Roll=-0.69 Pitch=0.81      Sys_cal=0 Gyro_cal=3 Accel_cal=0 Mag_cal=0
Heading=0.00 Roll=-0.69 Pitch=0.81      Sys_cal=0 Gyro_cal=3 Accel_cal=0 Mag_cal=0
```

A system status of 5 indicates that the fusion algorithm is running, and a self-test result of 0x0F shows that all sensors are operational. The orientation data from the sensor is produced every second as Euler angles, which represent the sensor's heading, roll, and pitch in degrees. Each second, the calibration level of each sensor is printed in addition to the orientation. It's critical to calibrate the BNO055 sensor if you want accurate orientation measurements. The system (fusion algorithm) is visible, and each sensor has its own calibration level. A level of 0 indicates that the device has not been calibrated, while a level of 3 indicates that it has been fully calibrated (with 1 and 2 being levels of partial calibration). The IMU will then be calibrated as the following step.

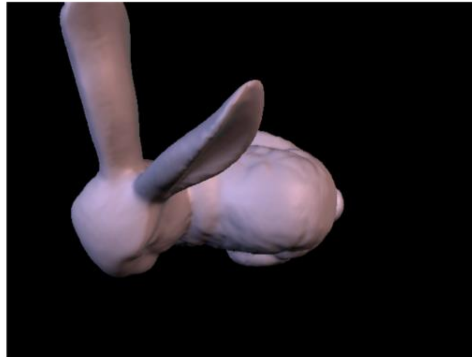
Calibration of the IMU

As you could see running the example above, the IMU needs to be calibrated to function properly. So, this part will go to a calibration by running the "webgl_demo". This example shows how to send orientation readings to a webpage and use it to rotate a 3D model.

1. Before you can run the web example, you need to install the flask Python web frame by running:
   ```
   sudo pip install flask
   ```
2. The second thing you need to do before running the example is make sure you have a web browser that supports WebGL. It is recommended to use Chrome.
3. Now you are ready to run the example. The first thing to do is to navigate to the "webgl_demo" example folder, run:
   ```
   cd ~/Adafruit_Python_BNO055/examples/webgl_demo
   ```
4. Just like the simpletest.py example, you'll need to update the server.py file and change the bno setup lines. Depending on how the BNO055 is coupled to your hardware, only one bno =... line should be left uncommented. Run:
   ```
   sudo nano server.py
   ```
5. Navigate down and make sure the only bno = … that is uncommented is the:
   ```
   "bno = BNO055.BNO055(serial_port='/dev/ttyAMA0', rst=18)
   ```
6. After you've finished editing the file, use Ctrl-S to save it, and then Ctrl-X to leave nano.
7. Now you are ready to run the file. Execute the following command as a root user with sudo:
   ```
   sudo python3 server.py
   ```
8. When the server is up and running, you should see something like this:
   ```
   * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
   * Restarting with stat
   ```
9. Now it's time to head over to your computer and launch a web browser. Go to your board's IP address on port 5000, as seen in below:

10. If everything went correctly, you should see something similar to the image below:



If you move the BNO055 sensor, the 3D model should move with it. When the demo initially runs, however, the sensor will be uncalibrated and will most likely not provide accurate orientation data. So now you must begin the calibration process.

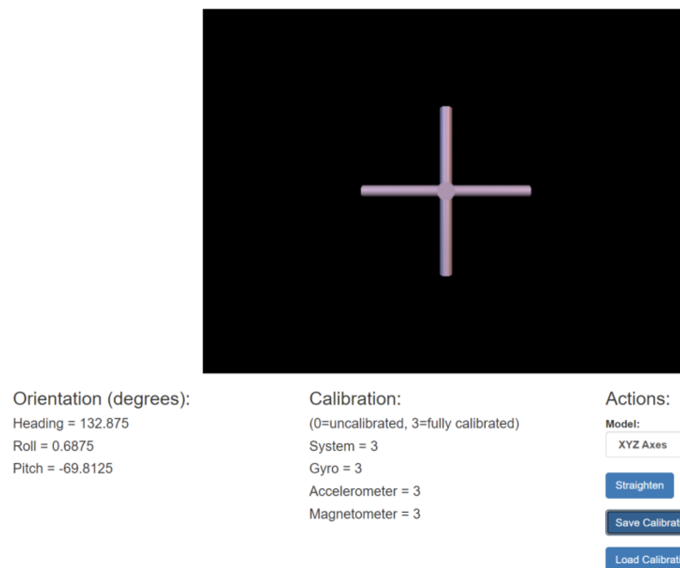
If you want to utilize the BNO055 sensor, make sure it's calibrated every time it's turned on or reset. The BNO055 takes care of most of the calibration for you, but you will need to move the sensor in specific ways to finish it.

The current calibration state of the BNO055 sensor is displayed in the bottom center column of the web page. The system (or fusion algorithm), gyroscope, accelerometer, and magnetometer are the four components of the sensor that are separately calibrated. Each component has a calibration level ranging from 0 to 3, with 0 indicating that it is uncalibrated and 3 indicating that it is completely calibrated. To receive the optimum orientation data, all four components should have a calibration level of at least 3. However, if a few of the sensors and the system are calibrated to level 2 or 3, you should still obtain good results.

1. Calibrate the Gyroscope first. This is the most straightforward to calibrate, and it will almost certainly be fully calibrated by the time you access the web page. Place the sensor on a table and let it immobile for a few seconds to calibrate the gyroscope.
2. The magnetometer should then be calibrated. It's a little more difficult to calibrate this one. You must constantly move the BNO055 sensor through a figure 8 of infinity pattern until the magnetometer calibrates. After roughly a dozen moves in the figure 8 pattern, the sensor will usually calibrate. Any significant metal items near the sensor may cause the calibration to change or slow down.
3. Begin calibrating the accelerometer now. The accelerometer may be calibrated in two different ways. The first method is to hold the sensor for a few seconds in around six different locations. Consider a cube with six faces. Slowly move the sensor between each face and hold it there for a few seconds. If the accelerometer is calibrating, you'll notice its level climb from 0 to 1 and then up to 3 after switching to more faces after roughly 3-4 faces.

The accelerometer may also be calibrated by rotating the IMU along an axis and holding it for a few seconds at each 45-degree angle. When you observe the calibration level move from 0 to 1 after holding at a couple different 45-degree angles, you'll know it's working.

4. The system, or fusion algorithm, is the last item to calibrate. Once all the sensors have started to calibrate, this will calibrate. As each sensor completes its calibration, the system calibration will most likely rise. Allow the sensor to finish calibrating the system after all the sensors have been calibrated.
5. Your IMU should now be calibrated, and each calibration level should be at a level 3, as seen below:



6. You may save time in the future by clicking the “Save Calibration” option on the right, which will save the calibration data to a calibration.json file. Press “Load Calibration” to load the file and its calibration when you restart the server in the future. After loading calibration, you may need to re-calibrate the magnetometer, although the accelerometer and system calibration are usually significantly faster in a loaded configuration.

NOTE: For optimal performance, the sensor must be calibrated every time it is switched on or reset (for example, when the server is restarted). After calibrating, call the “get_calibration()” method in your own scripts that utilize the BNO055 library and store the resulting list of data (it will return 22 integers), then reload it later using the library's “set_calibration()” function.

7. To align the axes of the IMU and the 3D model, use the "Straighten" button.
8. You may also modify the 3D model by selecting one from the "Model" drop-down menu on the right.
9. Return to the terminal where the server was launched and hit Ctrl-C to stop it. You may also need to execute the following command to terminate any Python processes that are still running (the browser can occasionally keep a zombie flask process alive):

```
sudo pkill -9 python
```

You have now completed the installation of the IMU BNO055. Congratulation!!

Sources

[1] T. DiCola, "BNO055 Absolute Orientation Sensor with Raspberry Pi & BeagleBone Black," [Online] https://learn.adafruit.com/bno055-absolute-orientation-sensor-with-raspberry-pi-and-beaglebone-black/overview [Hentet 17/01/2022]

[2] JOEL, "Using the RPi.GPIO module with Python 3," [Online] https://www.caretech.io/2018/01/20/using-the-rpi-gpio-module-with-python-3/ [Hentet 17/01/2022]

[3] linuxtut, "Install raspi-config on Ubuntu 20.04 (LTS)," [Online] https://linuxtut.com/en/a252676a3ce6bd1410da/ [Hentet 17/01/2022]

Appendix C

Installation guide Raspberry Pi HQ camera

Raspberry Pi HQ camera on Ubuntu 20.04

If you want to utilize a Raspberry Pi HQ camera on Ubuntu 20.04, you must do certain steps since programs like raspi-config, raspi-clone, and raspi-imager do not work. The first step is to install raspi-config. After that, you must use raspi-config to activate the camera. Once this is completed, you may test the camera to ensure that everything works as it should.

Install raspi-config

Before you can use your pi camera, you must first install raspi-config in order to activate it. However, before you begin the installation, you must connect the camera to the correct input on the Raspberry Pi.

1. Connect the Raspberry Pi HQ camera to the Raspberry Pi when it is powered off, as demonstrated in the image below:

   NOTE: Do not connect the camera to the display port as seen below:

2. Now install the “raspi-config” tool on your Ubuntu. To install “raspi-config” run:

```
wget https://archive.raspberrypi.org/debian/pool/main/r/raspi-config/raspi-config_20200601_all.deb -P /tmp
```

```
sudo apt-get install libnewt0.52 whiptail parted triggerhappy lua5.1 alsa-utils -y
sudo apt-get install -fy
sudo dpkg -i /tmp/raspi-config_20200601_all.deb
```

3. To use "raspi-config," build a symbolic link between the files "/boot/firmware/cmdline.txt" and "/boot/cmdline.txt" by running the command:

```
sudo ln -s /boot/firmware/cmdline.txt /boot/cmdline.txt
```

4. Now, execute the following command to get your device number: See the image below for a sample of the output you will see when you execute the command.

```
df -h
```

```
zebop@zebop-desktop:~$ df -h
Filesystem       Size  Used Avail Use% Mounted on
tmpfs            776M  3.6M  773M   1% /run
/dev/mmcblk0p2    30G   21G  7.1G  75% /
tmpfs            3.8G     0  3.8G   0% /dev/shm
tmpfs            5.0M  4.0K  5.0M   1% /run/lock
/dev/mmcblk0p1   253M  111M  142M  44% /boot/firmware
tmpfs            776M  1.3M  775M   1% /run/user/1000
```

5. Now mount the /boot directory by running:

```
sudo mount /dev/mmcblk0p1 /boot
```

NOTE: Yours may be different

6. Run df -h again to see if the adjustments were made

```
df -h
```

```
zebop@zebop-desktop:~$ df -h
Filesystem       Size  Used Avail Use% Mounted on
tmpfs            776M  3.6M  773M   1% /run
/dev/mmcblk0p2    30G   21G  7.2G  75% /
tmpfs            3.8G     0  3.8G   0% /dev/shm
tmpfs            5.0M  4.0K  5.0M   1% /run/lock
/dev/mmcblk0p1   253M  111M  142M  44% /boot
tmpfs            776M  1.3M  775M   1% /run/user/1000
```

Run raspi-config

Now that you've successfully installed raspi-config, you'll need to run it to activate the camera.

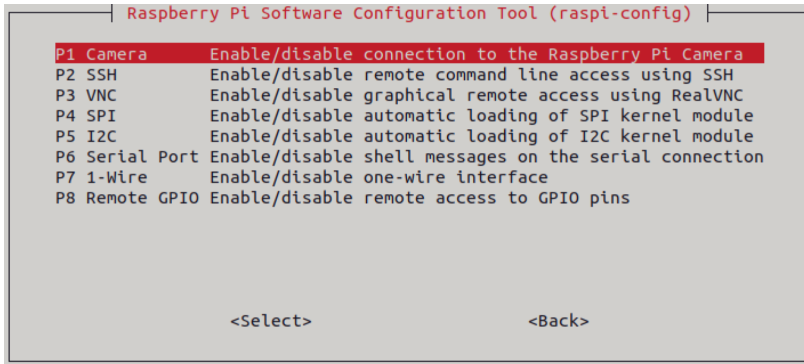
1. Run the following command to activate the camera using the "raspi-config" tool:

```
sudo raspi-config
```

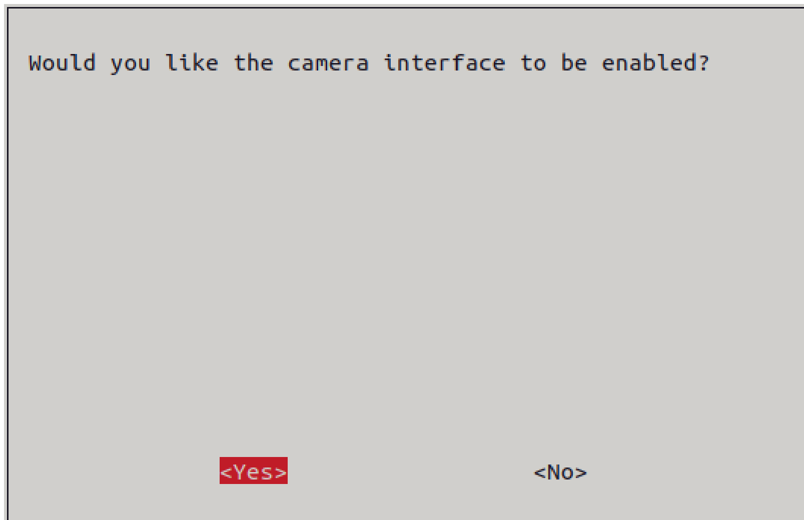
2. Go to “3 Interface Option” (sometimes it is “5 Interface Option”)

```
Raspberry Pi Software Configuration Tool (raspi-config)

1 System Options       Configure system settings
2 Display Options      Configure display settings
3 Interface Options    Configure connections to peripherals
4 Performance Options  Configure performance settings
5 Localisation Options Configure language and regional settings
6 Advanced Options     Configure advanced settings
8 Update               Update this tool to the latest version
9 About raspi-config   Information about this configuration tool

          <Select>                 <Finish>
```

3. Select “P1 Camera” to enable

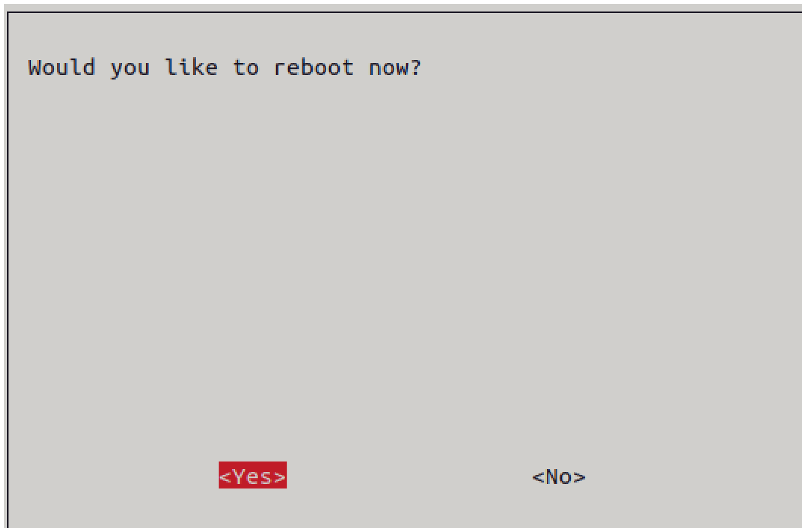


4. Then select "Yes"



5. Select "OK"

6. Now finish the raspi-config by going down to "Finish" and press "ENTER"
7. Reboot when prompt



Test the camera

All that remains is to ensure that the camera is operational. Begin by locating the camera, and then conduct some tests to ensure that everything is functioning properly.

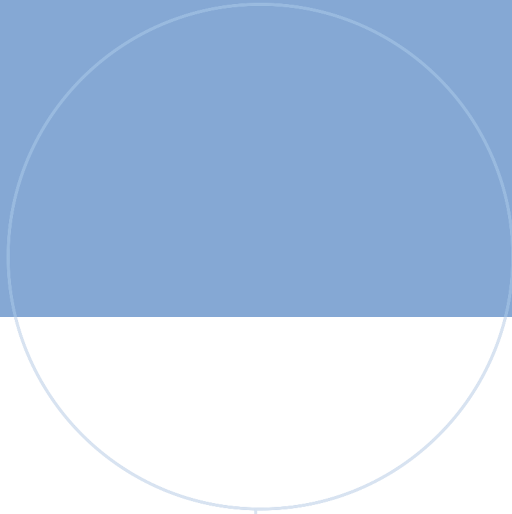
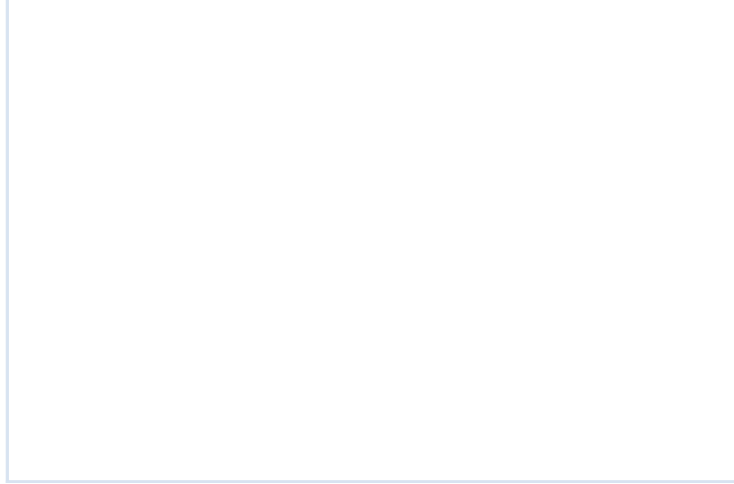
1. Install v4l2-ctl to locate the camera by running:
   ```
   sudo apt-get install v4l-utils
   ```
2. Run the following command to locate the camera: The image below displays an example of how the output may look.
   ```
   v4l2-ctl --list-devices
   ```
   ```
   zebop@zebop-desktop:~$ v4l2-ctl --list-devices
   bcm2835-codec-decode (platform:bcm2835-codec):
           /dev/video10
           /dev/video11
           /dev/video12

   bcm2835-isp (platform:bcm2835-isp):
           /dev/video13
           /dev/video14
           /dev/video15
           /dev/video16

   mmal service 16.1 (platform:bcm2835-v4l2-0):
           /dev/video0
   ```
3. Now, use the following command to install ffmpeg:
   ```
   sudo apt install ffmpeg
   ```
4. Finally, you may execute the following command to test the camera:
   ```
   ffplay /dev/video0
   ```
   Your camera should now be operational.

Source

[1] Raspberry Pi Trading Ltd., “Raspberry Pi High Quality Camera Getting started,” [Online] https://static.raspberrypi.org/files/product-guides/Raspberry_Pi_High_Quality_Camera_Getting_Started.pdf [04.02.2022]

[2] IChuck, “Enable Pi Camera with Raspberry Pi 4 Ubuntu 20.10 #樹梅派相機設置,” [Online] https://chuckmails.medium.com/enable-pi-camera-with-raspberry-pi4-ubuntu-20-10-327208312f6e [Hentet 04/0/2022]

[3] linuxtut, “Install raspi-config on Ubuntu 20.04 (LTS),” [Online] https://linuxtut.com/en/a252676a3ce6bd1410da/ [Hentet 17/01/2022]

NTNU

Norwegian University of
Science and Technology