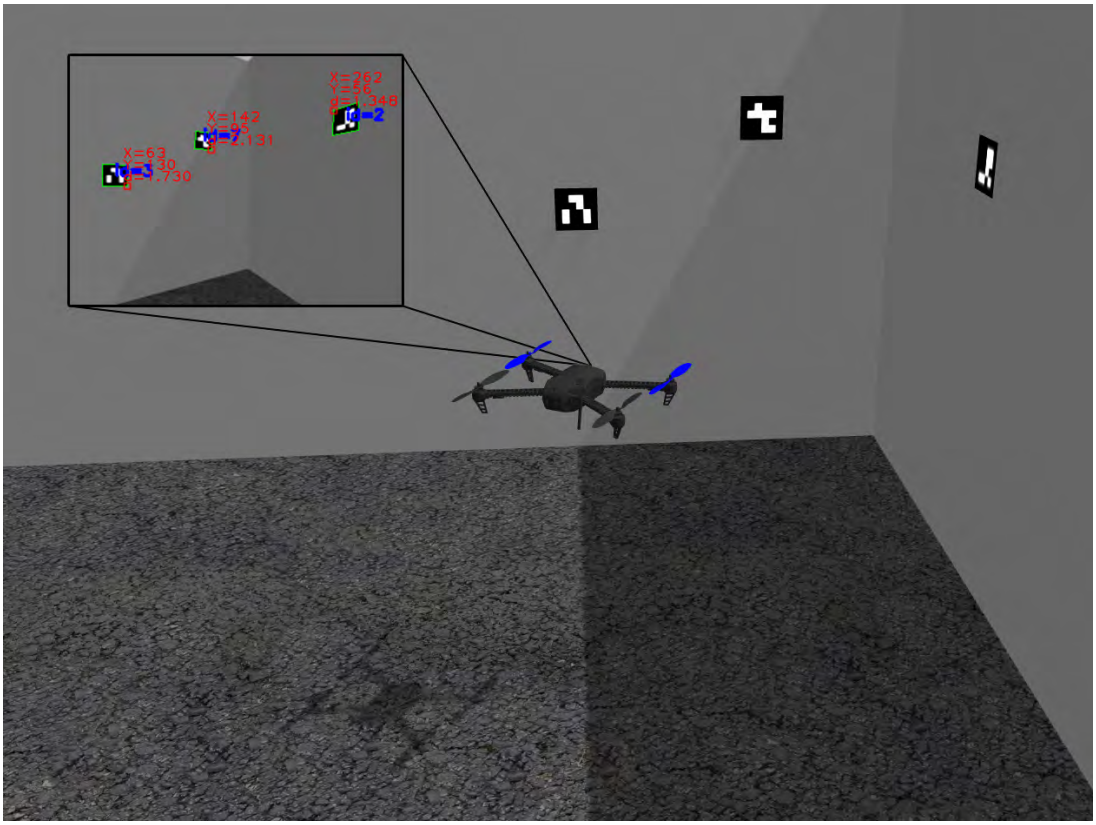
AALBORG UNIVERSITY

STUDENT REPORT

Autonomous indoor navigation for drones using vision-based guidance



Student Report
2. semester
Group 831
Control & Automation
Spring 2017

Control & Automation
Fredrik Bajers Vej 7C
DK-9220 Aalborg Ø

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Autonomous indoor navigation for drones using vision-based guidance

Theme:
Multivariable Control Systems

Project Period:
2. Semester Control and Automation

Project Group: 831

Page Numbers: 172

Date of Completion:
30. May 2017

Supervisor(s):
Henrik Schiøler

Participant(s):

Chris Jeppesen

David Romanos

Joan Calvet Molinas

Malte Rørmose Damgaard

Thomas Kølbæk Jespersen

Abstract:

The UAWorld project tries to introduce UAVs into warehouse and factory environments by the help of GamesOnTrack (GOT), an indoor positioning system. Unfortunately drop-outs and dead zones may occur in an indoor positioning systems. The purpose of this project is to investigate whether a colour and depth camera (RGB-D) can be used on a drone to increase the safety when navigating indoor, even at loss of GOT measurements.

The filtering-based FastSLAM 2.0 algorithm is explored as a real-time SLAM position estimator using a combination of RGB-D measurements and GOT measurements. ArUco markers are placed in the environment as visual landmarks ensuring a static environment. The FastSLAM estimator is used in conjunction with an Extended Kalman Filter (EKF) to estimate a Full-State feedback for a set of position controllers based on a state-space model derived from an identified ARX model of the drone.

The system is developed on the Intel Aero Ready to Fly drone incorporating a PX4 flight controller for attitude stabilization, given attitude and thrust set-points by the position controllers. The Robot Operating System (ROS) is used for both development and test of the FastSLAM estimator, EKF and position controllers.

The results illustrates how the camera-based system is indeed able to estimate the position of a drone, such that the position can either be held still or the drone can continue its' motion, even at loss of GOT measurements.

Preface

This project contains the development of an autonomous navigation solution for drones in GPS-denied indoor environments using an RGB-D camera and is a part of the UAWorld research project. The project is based on multivariable control theory, non-linear state estimation and the Simultaneous Location and Mapping (SLAM) problem.

A general understanding of probability theory and statistics is expected of the reader, however common estimation theory and methods used throughout the report can be found within Appendix E. Any specific concepts not familiar to a general reader from a degree within Control & Automation is described, both with respect to the functionality, choices and decisions.

The report is written with separate chapters, each covering a main part of the entire project.

Chapter 1 gives an introduction to the UAWorld project and the problem of which this project is concerned. The introduction is leading to the problem formulation which is the main question dealt with in the project.

Chapter 2 contains an initial analysis of the necessity of position feedback on drones and furthermore gives a description of the environment and the GOT position sensors installed in the environment. Furthermore the chapter introduces the SLAM problem to the reader.

Chapter 3 contains an analysis of different vision based solutions for Simultaneous Location and Mapping and results in a choice of algorithm.

Chapter 4 presents the Intel Aero Ready to Fly drone platform used in the project.

Chapter 5 gives an overview of the proposed solution and system, describing each element of the system to be developed. The chapter results in a system diagram and implementation overview.

Chapter 6 describes the design, implementation and test of the position controllers on the drone, including a z controller and an x-y controller.

Chapter 7 describes the FastSLAM 2.0 algorithm chosen to solve the SLAM problem and estimate the position of the drone. Motion and measurement models are put up for the RGB-D camera and the GOT positining system and the chapter results in a test of the algorithm using measurements from the drone.

Chapter 8 describes the design, implementation and test of the Extended Kalman Filter used for Full-State estimation for the position controllers.

Chapter 9 concludes on the results of the project and concludes on the problem formulation.

Chapter 10 describes the parts of the project which is left for future development to improve the system.

The group would like to thank the group supervisor Henrik Schiøler for his guidance and suggestions of possible solutions and methods.

Material relevant for the project but not included within this report, e.g. source code, measurement data, guides etc. can be found on the GitHub repository maintained by the project group [1]. The repository mainly contains the source code for the ROS implementation of the system but also MATLAB scripts for the controller development and links to measurement data can be found. To test the code, the guide found in Appendix K describes how to set up the necessary ROS and Gazebo simulation environment, including the PX4 Software in the loop simulator.

Table of Notation

The common notations used throughout the report is shown in Table 1.

c	Scalar (small letter)
$\boldsymbol{s}$	Vector (bold small letter)
$\boldsymbol{M}$	Matrix (bold capital letter)
$\mathbf{0}_{3\times4}$	Zero matrix of 3 rows and 4 colums
$\mathbf{I}_3$	Identity matrix of size 3
s_{GOT}	Text or symbol based subscripts denoting specific types of a variable
z	Backward shift operator within discrete z-transform
$p(X)$	Probability density functions
$p(X, Y)$	Joint probability density functions
$p(X \mid Y)$	Conditional probability density functions
$\mathbb{E}[X]$	Expectation
$X \sim (\mu,\, \sigma^2)$	undefined probability distribution with mean μ and covariance σ^2
$X \sim \mathcal{N}\left(\mu,\, \sigma^2\right)$	Symbol in front of parentheses indicates type of distribution. E.g. $\mathcal{N}$: Normal distribution.
$\mu = \mathbb{E}[X]$	Scalar Mean
$\sigma^2 = \mathrm{Var}(X)$	Variance
$\bar{\boldsymbol{x}} = \mathbb{E}[X, Y]$	Vectorial Mean
$\boldsymbol{\Sigma} = \mathrm{Cov}\,(X, Y)$	Covariance
$\hat{\boldsymbol{x}}$	Estimate
x^{k+1}	Time instance $k+1$ of a time varying variable
$H^{k,T}$	Transposed matrix of a time varying matrix at time instance k
${}^{\mathrm{A}}_{\mathrm{B}}\mathbf{R} \in \mathbb{R}^{3\times3}$	SO(3) rotation matrix from frame B to frame A
${}^{\mathrm{A}}_{\mathrm{B}}\mathbf{T} \in \mathbb{R}^{4\times4}$	SE(3) transformation from frame B to frame A
${}^{\mathrm{A}}\boldsymbol{v}$	Variable described in frame A
$[\cdots]$	Array
$\{\cdots\}$	Set
$\left(\left(\left(\left((\cdots)\right)\right)\right)\right)$	Multi-level parenthesis
$\boldsymbol{s}_{[p]}$	Particle indexing
$\boldsymbol{s}_{[p],i}$	Iterated single particle

Table 1: Table of the nomenclature used in the report.

Contents

1 Introduction

Small multi-rotor unmanned aerial vehicles, loosely referred to as drones in the following, are being recommended for an increasing number of missions and use-cases due to their flexibility, customizability and capability of rapid movements. Unfortunately, drones are highly dependent on position feedback to stabilize their position, as it is inherently unstable due to the uncertainties involved in estimating the exact attitude of the drone, where attitude refers to the roll, pitch and yaw angles.

Most of the drones currently available on the market include a GPS sensor to allow outdoor positioning and trajectory following. For use-cases in GPS-denied environments such as indoor, the drone options are still limited though. Current options based on LiDAR technology allow some commercial drones to navigate autonomously indoor [2], though LiDAR technology is still very expensive and heavy to use on smaller and versatile drones.

GamesOnTrack, a Danish company making indoor positioning systems based on ultrasound and radio communication, can provide a decent indoor position estimate with approximately 10 mm resolution and could thus be a solution in GPS-denied environments. Unfortunately, both the GamesOnTrack system (GOT) and the outdoor GPS system suffer from dead zones in where position measurements are unavailable. The GPS system dead zones are usually due to poor atmospheric conditions, bad antenna placement, multi-path effects (Canyon effect), tree cover or interference, while the dead zones within GOT are usually due to loss of sight or loss of radio contact.

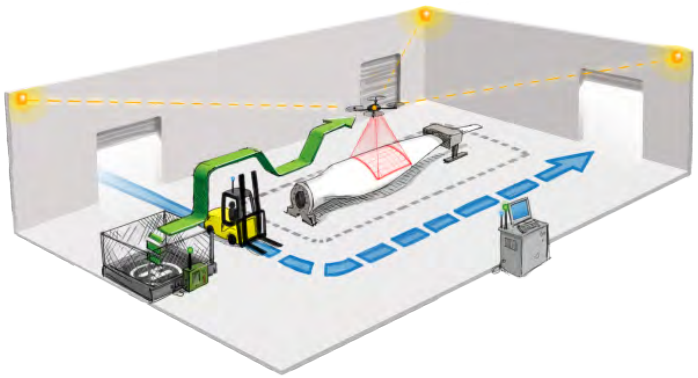


Figure 1.1: UAWorld concept illustration [3]

The UAWorld project [3] [4] tries to introduce UAS into warehouses and factory environments with the help of GOT [5], but due to the uncertainties within the positioning the project has not yet been approved by the Danish authorities to be used in a work environment where humans are present. This significantly reduces the usability of drones in these environments, so to ensure that drones can manoeuvre safely even in the presence of GOT dead zones, an enhancement to the solution proposed by UAWorld has to be found.

One approach to increase the safety of drones within the UAWorld project is to use additional on-board sensors to perceive the environment in such a way that distinct features can be identified and used for position estimation, allowing the drone to manoeuvre safely through a dead zone until position measurements from GOT become available again. Such sensors could e.g. be LiDARs or cameras with real-time video/image processing. As mentioned, LiDAR technology is expensive and heavy, which makes cameras a favourable choice for usage on a drone, since this kind of sensors are usually cheap and lightweight.

This leads to the following problem formulation which is the research topic of this project:

"Is it possible to make the UAWorld drones safer by overcoming the dead zone problem of the GamesOnTrack system by incorporating a color and/or depth camera and real-time image processing on a drone."?

2 Problem analysis

Drones are notoriously unstable and require a constant position feedback to stay put. In this chapter, the problem formulation presented in Chapter 1 is investigated and a description of why it is necessary to provide position feedback to stabilize a drone is given. The purpose of this project is to increase the safety of the UAWorld drones by incorporating a combination of position measurements from a GOT system and measurements from a colour and/or depth camera mounted on the drone. An analysis of the environment in which the UAWorld drones are navigating and an analysis of the existing GOT system are made, such that a proper decision on how to use measurements from a color and/or depth cameras can be taken.
The analysis is concluded with an explanation of the SLAM problem which is inevitable to the UAWorld drones when GOT measurements are unavailable and only measurements from on-board sensors can be used, hereby leading up to the analysis of present SLAM methods in Chapter 3.

2.1 Inevitable position feedback with drones

Attitude stabilized drones, being the focus of this project, are usually stabilized with the help of 3-dimensional accelerometers, measuring the proper acceleration, gyroscopes, measuring the angular velocity, and magnetometers, measuring the direction of a magnetic field, eg. the one generated by the Earth.

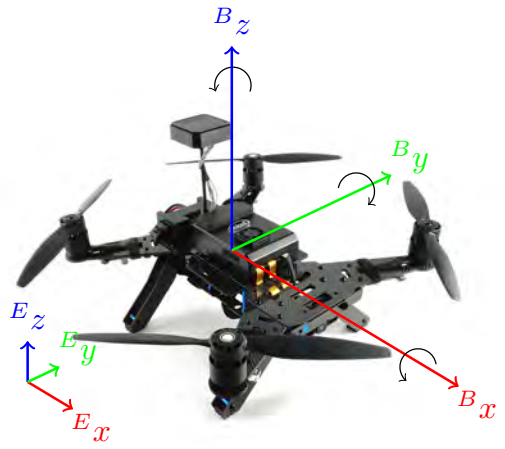


Figure 2.1: Drone coordinate frame with axis and angles

Stabilizing the attitude of a drone requires that the roll (ϕ), pitch (θ) and yaw (ψ) angles, see Section A.1, are controllable and close to hover at all times. At hover the roll and pitch axis align with the inertial frame of the world which allows the thrust vectors generated by the propellers to compensate exactly for the gravity force affecting the drone without causing any movement of the drone. As a result, this will affix the position of the drone, defined as position hold. Even the slightest tilt away from this alignment will cause the drone to accelerate in a given direction.

This is seen by (2.1) which describes a simplified model of the acceleration of the drone within a heading frame, being a frame that is aligned with the heading of the drone but does not tilt, as explained in Section A.2.

$$\begin{bmatrix} {}^{\mathrm{H}}\ddot{x} \\ {}^{\mathrm{H}}\ddot{y} \end{bmatrix} = g \begin{bmatrix} \theta \\ -\phi \end{bmatrix} \tag{2.1}$$

Where:

${}^{\mathrm{H}}\ddot{x}$	is the linear acceleration in the x-coordinate of the heading frame
${}^{\mathrm{H}}\ddot{y}$	is the linear acceleration in the y-coordinate of the heading frame
g	is the gravity of Earth
ϕ	is the rotation around the x-axis of the earth frame
θ	is the rotation around the y-axis of the earth frame

The yaw angle is related to the heading of the drone which can be extracted from the direction of the magnetic field measured by the magnetometer.

Unfortunately neither the roll or pitch angle are directly measurable from any of the sensors normally mounted on a drone, and would thus need to be estimated. The angular velocities from the gyroscopes can be integrated to give a dead-reckoning-based estimate of the roll and pitch angle, if one assumes that the integration is started when the drone is in exact hover. Unfortunately, such estimates drift over time which would cause the drone to accelerate in an arbitrary direction if these estimates were used solely to stabilize the attitude of the drone. It is therefore necessary to correct this estimate with information from other sensors.

Accelerometers are electronic sensors that measure proper acceleration, also known as 'g-force'. At rest or at exact hover the proper acceleration experienced by the drone is an upward pointing acceleration, equal to the opposite of the acceleration caused by gravity. At rest on the floor the accelerometer can be used to estimate the attitude by extracting this opposite gravity vector from the individual directional components. However, when flying it is shown in Appendix B that the accelerometer is only capable of measuring the thrust vector, which will always be pointing in the same direction within the measurement frame of the accelerometer, no matter the roll or pitch of the drone. Except for the thrust vector, only smaller deviations resulting from wind forces, wind gusts and other disturbances are measurable, no matter how much the drone is tilted. One could say that the accelerometer measures the instantaneous of the total aerodynamic forces on the drone.
Hence, the accelerometer is not very useful for estimating the roll and pitch angles, but if the wind resistance is considered it is possible to extract some information about the angles. Unfortunately, such a configuration is very vulnerable to alignment errors between the thrust direction and the accelerometer z-direction. Errors would result in angle offsets which are not correctable with any of the sensors mentioned so far, as actual world coordinate accelerations are not observable. A sensor either capable of measuring the actual attitude or at least a sensor capable of measuring world positions, velocities or accelerations is necessary.

Sensors such as GPS, pressure sensor, wind velocity sensor, camera-based optical flow etc. are all capable of providing some sort of absolute position or relative velocity measurement. Within this project the drones are intended for indoor use, as elaborated in Section 2.3. This does not allow GPS sensors to be used as these are very unreliable indoors. An alternative to GPS has been invented by GamesOnTrack, intended for indoor usage, described in Section 2.2 and Appendix C.

Within the UAWorld project it has been decided to use this sensor as the main indoor positioning source. Other sensors such as pressure sensors for measuring the altitude do not work very well indoor either due to the possibility of sudden pressure changes from room to room or if a door is suddenly opened. Neither do wind velocity sensors as the velocities with which the drone should fly, have to be kept small for safety reasons.

Finally, an optical flow sensor is capable of providing an estimate of the horizontal velocity of the drone by pointing a camera downwards or upwards if flying inside a building. Even though the velocity measurements are relative to the heading and angle of the drone, why a position estimate based on optical flow will drift, such velocity measurements can still be used to correct the drifting behaviour of the dead-reckoning-based attitude estimate. In general the camera-based approaches are investigated as part of this project to find a camera-based solution which can be used in conjunction with the GOT sensor.

2.2 GOT system overview

The GamesOnTrack system is an indoor positioning solution based on a transmitter-receiver configuration using a combination of ultrasound-waves and radio communication.

Figure 2.2: GamesOnTrack system showing two satellites and one beacon [6]

The system consists of a transmitter, denoted as beacon, mounted to the object which needs to be located in a room where several receivers, known as satellites, are installed. Using time-of-flight measurements of ultrasound-waves sent from the transmitter to the receivers, the position of the object can be exactly localized using trilateration, see Appendix C. A common base-station unit connected to a stationary PC over USB performs this trilateration and sends the determined position wirelessly to the drone over a UDP connection.
Unfortunately, the GOT system suffers from problems with dead zones or bad position estimates, just as the regular GPS system does when navigating in closely packed cities with tall buildings. Such unknown, unexpected and assumed unforeseeable errors with the position measurement from the GOT sensor deems it necessary to develop a position estimator that can take other measurements into account, in this case measurements from an RGB and depth camera.

The specific GOT system set-up installed in the Motion Tracking lab on Fredrik Bajers Vej has an accuracy down to approximately 10 mm in all three coordinates of the 3 dimensional space [6] and provides an update rate of approximately 10 Hz. The following covariance matrix of the measurement noise has been estimated by a former group [7].

$$\Sigma_{\text{GOT}} = \begin{bmatrix} 0.225 & -0.038 & 0.022 \\ -0.038 & 0.025 & -0.009 \\ 0.022 & -0.009 & 0.014 \end{bmatrix} \cdot 10^{-4} \text{ m}^2 \tag{2.2}$$

Where:

Σ_{GOT} is the covariance of measurements from the given GOT setup $\left[\text{m}^2\right]$

The GOT system will be seen and modelled as a black-box sensor capable of providing measurements of the drone position with a noise covariance matrix as in (2.2). Furthermore, the specifications mentioned in this section and Appendix C will be used as initial design parameters when designing and simulating the position estimator but will be adjusted and tuned if necessary.

2.3 Environment analysis

The UAWorld project focuses on using multiple drones in indoor factory and warehouse environments thereby exploiting the full open space of indoor factory halls and warehouses [8]. The size of the environment, defining both the flying altitude and the distances which have to be travelled, varies from application to application. A lot of metal is usually present within factory and warehouse environments which will affect the magnetometer readings, therefore it is assumed that the magnetometer cannot be used. The system should therefore be capable of estimating the heading based on other measurements. The indoor environment also limits the usage of GPS as mentioned previously, hence GOT should be used as a substitute. The test hall being part of the UAWorld project is more than 400 m^2 with a height of approximately 8 m and covered by 15 GOT satellites [9]. This hall is taken as a baseline to which design criteria are put, though the position estimation solution should not be limited in any way by the size of the environment as long as it is correctly covered by GOT satellites.

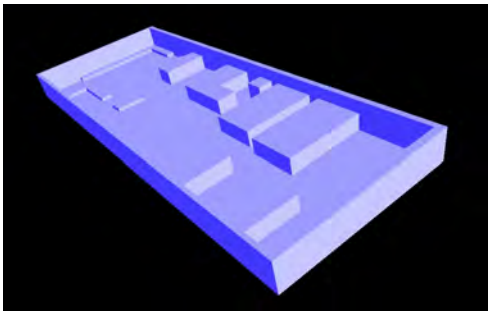


Figure 2.3: 3D modelled map of the baseline environment [10]

The focus of this project is not path planning nor collision-avoidance, and as a consequence no analysis on how to navigate correctly in the environment will be made. It is assumed that a collission-free 3D trajectory has been planned in advance [8] using the coordinate frame of the GOT system. The trajectory is given as sequential waypoints which should be reached by flying in straight lines.

The content of the environment will include several mixed-texture objects and possible dynamic objects. One can therefore not be sure that unique and distinct features will be fixed in space and re-discoverable from every possible location with every possible orientation. As the on-board RGB and depth camera have to be used for the position estimation it is important to consider how the visible environment can provide information about the location of the drone. It is chosen to affix some uniquely identifiable markers on fixed objects of the environment, eg. walls, such that at least one marker is always visible to the camera at any possible location when the drone is close to hover. Uniquely identifiable markers would increase the robustness of the system as the correspondences to the markers would always be known and thus not have to be estimated. This will decrease the likelihood of incorrect markers being used due to incorrect correspondences. Further information such as the actual location of the marker could also be embedded as part of the marker identifier to increase robustness and efficiency of the system. However, using specific markers limits the usability of the system to environments where markers are pre-installed, although the markers would only have to be installed once.

To simplify the marker installation it is chosen that the system should support an arbitrary placement of the markers and that the markers should not contain any specific information about their location. The design of the markers is limited by the uniqueness property which should allow the camera to distinguish between all possible markers and match a detected one with previously discovered markers to find correspondences.

2.4 The SLAM problem

Knowing that the attitude of the drone can not be stabilized without some sort of position feedback, the remaining question is how to get an estimate of the position when the absolute position sensor (GOT) stops working, e.g. due to loss of sight. Any type of sensor has a probability of failing, though the probability of sensors relying on wireless communication failing, is usually larger than local sensors. As the GOT sensor is a wireless sensor providing absolute measurements, the drone would benefit from having a local sensor which can provide measurements for the position estimation while the GOT sensor is not working. Unfortunately, no such sensor exists that can provide absolute position measurements while being an independent local sensor. On-board sensors such as LiDARs Ultrasound sensors, IR sensors or other distance sensors are all local sensors which provide relative measurements of the unknown surrounding environment. If 2D cameras or depth cameras are used to picture an unknown environment without any prior location knowledge of certain detectable features, then cameras are classified as relative sensors as well. Having only such relative measurements of distinct features from an unknown position of the drone inside an unknown environment makes this a non-trivial probabilistic problem known as the Simultaneous Location and Mapping problem.

A short generalized description of the SLAM problem is given below leading up to the analysis of the present state-of-the-art SLAM algorithms described in Chapter 3. The SLAM problem consists of two combined probabilistic problems: the location problem where a map of the environment is known but the location of the drone is unknown and the mapping problem where the location of the drone is known but the environment is unknown.

Let $\boldsymbol{s}$ denote the current relative position and orientation of the drone, later denoted pose, where the position is relative to the GOT reference frame and let $\boldsymbol{M}$ denote the known map of the environment described by a set of distinct landmarks l_1 to l_N. With a model of the drone movement, taking in an input signal $\boldsymbol{u}$, the location problem is a matter of finding the conditional probability density function of the position of the drone given the input, map, and measurement, $\boldsymbol{z}$.

$$p(\boldsymbol{s} \mid \boldsymbol{u}, \boldsymbol{M}, \boldsymbol{z}) \tag{2.3}$$

Where:

$\boldsymbol{s}$	drone pose
$\boldsymbol{u}$	motion model input
$\boldsymbol{M}$	map of the environment including distinct landmarks l_1 to l_N
$\boldsymbol{z}$	relative measurement of features within the environment

The mapping problem on the other hand is a matter of finding the conditional probability distribution function of the map of the environment given the known pose of the drone and a measurement

$$p(\boldsymbol{M} \mid \boldsymbol{s}, \boldsymbol{z}) \tag{2.4}$$

When both the pose of the drone and the map is unknown the problem turns into finding a joint conditional probability distribution function of both the pose and map while only being given inputs and measurement

$$p(\boldsymbol{s}, \boldsymbol{M} \mid \boldsymbol{u}, \boldsymbol{z}) \tag{2.5}$$

For many years a lot of research effort has been put into the development of implementations to calculate or approximate this distribution. Some implementations have been developed for off-line use, that is post-processing while others focus on the on-line and real-time use-cases. These two categories of SLAM implementations are further classified in Chapter 3 where also some of the state-of-the-art algorithms will be mentioned. Stressing the fact that measurements will be provided by an RGB and/or depth camera and that the SLAM algorithm needs to provide position estimates for real-time use, an algorithm will be chosen.

3 Visual Odometry through SLAM

As described in Section 2.4 the SLAM problem consists of determining the relative pose, $\boldsymbol{s}$, of the drone while simultaneously making a map, $\boldsymbol{M}$, of the environment. This can be done in several ways and with numerous types of sensors, though this chapter will focus on the usage of color and depth cameras, in conjunction denoted RGB-D cameras. This chapter results in an overall overview and classification of common state-of-the-art SLAM algorithms which are described in further details in Appendix D. This overview leads up to the decision of an algorithm to use within this project. The considered SLAM algorithm types are [11]:

1. Filtering-based SLAM, e.g. Bayesian SLAM.
2. Keyframe-based SLAM, e.g. Graph SLAM.

When the SLAM problem was originally proposed early solutions relied mostly on IR, laser or ultrasonic sensors and on odometry inputs such as wheel encoders. Due to the rather sparse set of measurements in these sensors compared to 2D images or point clouds generated from depth cameras, the proposed solutions were developed as Bayesian filtering frameworks, see Section E.2. Best known is the work from S. Thrun, D. Fox and W. Burgard, formalised in a later published book known as Probabilistic Robotics. Examples of Baysian filtering SLAM frameworks include EKF-SLAM and FastSLAM, both presented in further details in Section D.1.

Throughout the years from 1990 to 2000 cameras dropped rapidly in price, became more compact and with higher resolutions. As the processing power within computers and microprocessors grew at the same time, an interest was formed with the investigation of recovering relative camera poses and 3D structure from a set of calibrated or un-calibrated camera images. Based in the Computer Vision world this was later denoted as Structure from Motion (SfM), whose focus is to map the 3D world and its structures through post-processing and optimization of all collected images. This can thereby be considered as an early offline version of Visual SLAM which was later denoted as Keyframe-based SLAM being presented in Section D.2.

While the SfM interest mainly came from Computer Vision groups, an interest for Visual Odometry (VO) started to appear in Robot vision groups about the same time, though independent on the work done within SfM. Instead of mainly focusing on the mapping and structuring of the environment, the work within Visual Odometry, later known as Visual SLAM, focused on the problem of estimating the ego-motion of a robot using only the input of a single camera, multiple cameras (stereography) and/or a depth camera. A sub-category of the SLAM algorithms within Visual SLAM is monocular SLAM which uses only one camera. Clearly this shows that Visual SLAM does not denote a specific type of SLAM implementation but rather the overall focus and goal which is to estimate the pose of a robot from camera measurements. This agrees with the overall focus of this project.

3.1 Representation of the SLAM problem

As presented in Section 2.4 the SLAM problem contains the estimation of both the drone pose, s^k, at timestep k and the map, M, described by the landmarks l_1 to l_N, given only measurements of the environment ${z_n}^k$ and motion model inputs, u^k. This problem can be drawn in a constraint graph. Let ${z_n}^k$ denote a measurement of a distinct landmark, l_n, within the environment at timestep k. This implies that the correspondence between a measurement and the actual landmark within the map is known. As presented in Section 2.3 a set of uniquely identifiable markers has to be placed in the environment resulting in known correspondences. These correspondences impose a set of constraints between the hidden pose, the map variables and the image measurements, being the arrows in the graph. Furthermore, a motion model of the drone imposes the dashed constraints from pose to pose at increasing time steps.

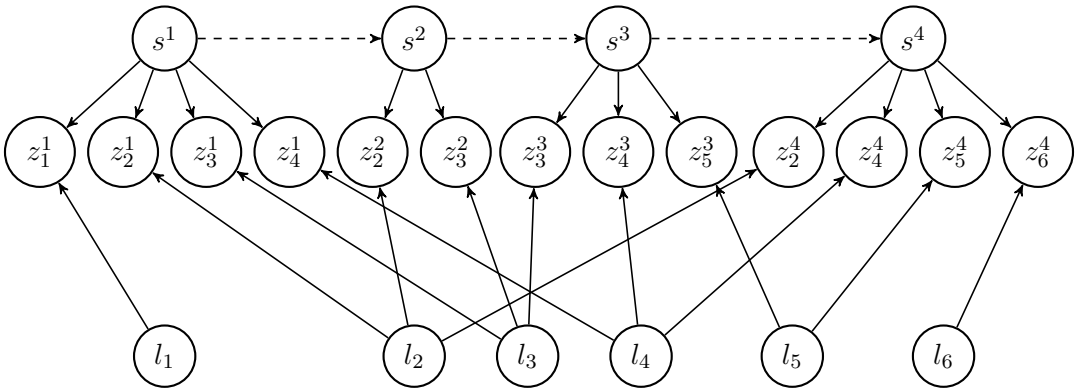


Figure 3.1: Constraint graph [12]

With a sufficient high number of constraints it is possible to estimate the drone's pose and the locations of landmarks. The constraints and variables in such a constraint graph can either be treated as deterministic or probabilistic variables. With probabilistic variables the constraint graph turns into a Bayesian network.
Two categories which both simplify and solve this SLAM problem have been formed: the Filtering-based and Keyframe-based methods. In Appendix D a description of these two categories together with a presentation of a few commonly used state-of-the-art algorithms are given.
The Filtering-based methods take a Bayesian approach to solve the probabilistic SLAM problem, where the constraints in the constraint graph are treated probabilistic variables. This is done by marginalizing out any past poses and measurements. This results in a recursive implementation capable of providing new state estimates in real time for each new measurement, which is especially useful for online applications. As an example EKF-SLAM uses a big Extended Kalman Filter which combines the pose estimate and the estimated location of all landmarks.
The Keyframe-based methods usually impose deterministic constraints in the constraint graph and take an optimization approach to the SLAM problem by keeping track of a subset of previous poses and measurements, using these together with new measurements to find an optimal current pose and an optimal location of the current landmarks within the map. This results in algorithms which are computationally demanding and mostly intended for offline usage, but gives better results with especially large environments where loop closure is crucial. However the computationally demanding algorithms can be split into separate tasks to allow real-time estimates to be generated while running the optimization in the background.

3.2 Visual SLAM Algorithms

A thorough overview of the most commonly used state-of-the-art Visual SLAM Algorithms within the filtering-based and keyframe-based categories is contained within Appendix D. An overview of the algorithms described in Appendix D is shown in Figure 3.2. One of the problems with most of the Visual SLAM methods shown in Figure 3.2, is that these methods are far from being truly usable for mobile robot navigation. Robot navigation requires real time state estimates of the position, hence pure offline solutions are not really usable. On the other hand, the real time solutions, especially the keyframe-based, require a lot of processing power and memory.

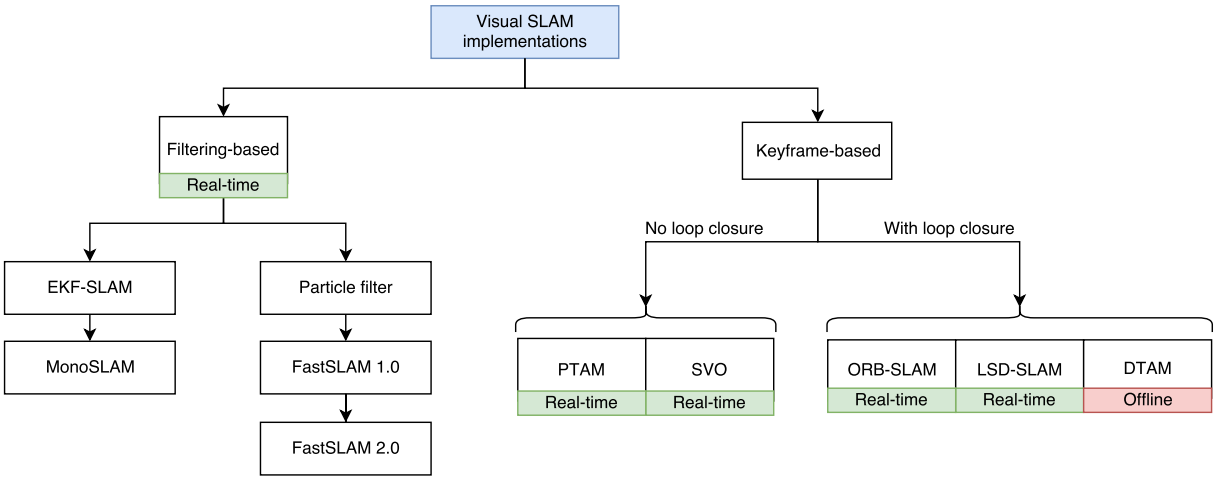


Figure 3.2: State-of-the-art Visual SLAM algorithms

The Visual SLAM Algorithms shown in Figure 3.2 can either be used with monocular, stereo, depth cameras or a combination. Unfortunately, monocular SLAM, using only a single camera, includes a huge limitation in its' unrecovered scale problem as the depth can not be estimated from just a single image and thus has to be inferred from a sequence of images. The leading limitation in monocular SLAM lies in the poor robustness, which is an inherent problem of pure vision-based methods, due to the fact that image tracking is easy to fail for many motion behaviours and environment structures.

Furthermore, the main challenge in large scale mapping remains in long term visual place recognition, also known as loop closure, which is the ability to detect past locations after having discovered new and unknown terrain for a while. In this project the Visual SLAM implementation is combined with external position measurements provided by the GOT system, as long as these are available. If lost, it is anticipated that these external position measurements will only be lost intermittently, hence loop closure will not be of high importance as the position measurements can be used to correct the estimate. Loop closure will in this project therefore not be considered.

To summarize, the interest of this project will be focused on mainly localizing the drone (positioning) without considering loop closures and without focusing on the 3D mapping of the environment. Hence, there will be no focus on Structure from Motion but rather a high focus on Visual Odometry also known as Visual SLAM. The Visual SLAM implementation should take image measurements, GOT measurements and other possible measurements and/or external estimates as input.

3.3 Choice of algorithm

The focus of this project is Visual SLAM using measurements from a color camera and/or a depth camera. As presented in Appendix D the Keyframe-based algorithms are useful for image-based SLAM but unfortunately very computationally intensive. Using measurements from hundreds of pixels simultaneously, the Keyframe-based algorithms optimize the pose and map through Bundle Adjustments. This makes these types of SLAM algorithms robust to sudden movements, changing environments, loop closures etc. In this project however the drone will be flying in a controlled environment where static markers to be detected are put up. The system is also given GOT measurements to help correct the position estimate, and thereby also help against loop closures. Using static markers reduces the number of possible measurements within a single image frame and by applying the GOT position measurements the uncertainty in the position estimate can be reduced significantly. As the movements of the drone is controlled by the controllers to be designed, a model of the system will also be designed. Such a model can provide useful position predictions for a Filtering-based method while a Keyframe-based method will have to be modified to include such model predictions.

Due to the reduced number of measurements, the possibility to include several sensors, and due to the availability of a model of the drone, a Filtering-based method is preferable.

As described in the introduction, either RGB images, depth images or a combination of these, denoted RGB-D, have to be used. Utilizing pure RGB images can result in the unrecovered scale problem, as described in Section 3.2, because depth of detected features has to be estimated initially. This unrecovered scale problem may not be that great of a problem in a solution that incorporates measurements from other sensors than RGB cameras. Especially sensors providing measurements in a global frame like the GOT sensor would probably reduce this problem significantly. Nevertheless, it is deemed appropriate to use a depth camera, since this will remove the uncertainty resulting from the unrecovered scale problem from the system, and thereby make it safer. On the other hand, one cannot be sure that unique features are always detectable within pure depth images, therefore it is chosen to use a combination of both RGB and depth (RGB-D) measurements.
Uniquely identifiable markers will be installed in the factory environment in such a way that at least one marker is always visible to the drone, as described in Section 2.3. Even though the environment is of a finite size the number of markers to install to fulfil this dense requirement will quickly outgrow the efficiency of EKF-SLAM. It is therefore beneficial to use one of the FastSLAM algorithms depending on the correctness of the motion model and the number of states to estimate.

In [13, 14] it is shown that the FastSLAM 2.0 algorithm has superior performance compared to the FastSLAM algorithm. Since the system is supposed to work in real-time, it is decided to use FastSLAM 2.0, in the following referred to as FastSLAM. The FastSLAM 2.0 algorithm includes the improved proposal distribution where the particles are spread less due to the inclusion of the current measurement, see Section D.1.3 and Section E.9. This allows less particles to be used giving the same space density after the prediction step and hence less computational requirements.

3.3.1 RGB-D measurements and feature extraction

An RGB-D camera provides a 2D color image (RGB array) and a 2D depth image (depth array). Using a given pixel coordinate, the intrinsics of the camera and the corresponding depth value, a relative 3D coordinate can be calculated, e.g. through the transformation described in Appendix G. This hereby enables calculation of 3D coordinates, relative to the camera frame, of features present in both the RGB and depth image. Thus, if unique features can be extracted from the 2D images provided by the RGB-D camera they can in turn be converted to relative 3D measurements to be used in the SLAM implementation. A distinct and uniquely identifiable type of markers should be chosen such that the markers are identifiable within at least the RGB image, allowing a corresponding depth value to be found within the depth image. Designing a SLAM implementation using specific markers limits the use-cases to environments where markers are pre-installed, however this is acceptable for the factory environment described in Section 2.3 as the markers would only have to be installed once. Uniquely identifiable markers would as well increase the robustness of the system as the likelihood of incorrect correspondences due to incorrect feature matching, is decreased significantly.

(a) Examples of QR codes

(b) Examples of ArUco markers [15]

Figure 3.3: Illustration of different types of uniquely identifiable markers

Examples of uniquely identifiable markers which can be placed manually in the environment include QR codes, Figure 3.3(a), and ArUco markers, Figure 3.3(b), both being rotation and scale invariant and rediscoverable from different viewing angles. ArUco markers, as presented in Appendix H, were originally developed for augmented reality for tracking and positioning with focus on reliability and rediscoverability whereof QR codes are intended for information storage. As the environment is finite, a huge number of unique markers is not necessary and thus the data length of the identifier to be contained within a marker will be short. Therefore information codes such as QR-codes do not give any benefits.

Based on the analysis of the ArUco markers in Appendix H it is decided to use and install them within the environment according to the requirements specified in Section 2.3.

With the choice of the unique ArUco markers, the RGB image should be used to identify the pixel locations of visible markers. From the same 2D pixel locations depth measurements should be grabbed within the depth image, thereby corresponding to the depth of detected markers. With these sets of 2D coordinates and depths a resulting set of relative 3D measurements of detected ArUco markers follows.

The FastSLAM measurement vector for image measurements relative to the camera frame is thus defined as:

$$\boldsymbol{z}_c = \begin{bmatrix} x_c \\ y_c \\ d \end{bmatrix} \tag{3.1}$$

Where:

$\boldsymbol{z}_c$	is the measurements vector of a feature in the RGB-D image
x_c	is x-coordinate of the feature the camera frame
y_c	is y-coordinate of the feature the camera frame

The FastSLAM implementation thus has to include a measurement model of the camera, including the camera intrinsics, describing how to calculate an estimated measurement vector for a current landmark within the map, given the current pose of the drone. Furthermore, an inverse measurement model has to be defined to describe how to convert a detected marker, given by the markers measurement vector, into a corresponding 3D coordinate relative to the camera frame which can be inserted into the map as a new landmark.

The analysis and descriptions provided within this chapter conclude the problem analysis with the choice of the FastSLAM 2.0 algorithm to be used as the SLAM algorithm for this project. RGB and depth measurements from an RGB-D camera will be used to detect uniquely identifiable ArUco markers placed in the environment such that 3-dimensional measurement vectors, with known correspondences to the detected markers, can be provided to the FastSLAM algorithm.

4 Platform Description

In this project it has been decided to use the Intel Aero Ready to Fly kit. This kit is sold by Intel and is targeted at researchers and developers. The kit contains a fully assembled quadcopter ready to fly with on-board cameras and sensors. The Intel Aero platform is aimed for development within the field of unmanned aerial vehicles. The kit consist of a carbon frame on which the following components are mounted [16]:

- Motors
- Motor controllers
- Power supply
- GPS and magnetometer
- RC serial receiver
- Intel Aero Compute board
- Intel Aero Flight Controller
- Intel RealSense R200 RGB-D camera
- 8 MP RGB camera
- Monocrome VGA camera.

4.1 Intel Aero Flight controller

The Intel Aero flight controller is a board consisting of a microcontroller and a set of peripherals. The microcontroller is specified as an STM32 microcontroller which is running the open source PX4 autopilot code [17]. Within this project the Intel Aero flight controller will be referred to as the PX4.

The STM32 microcontroller is connected to a list of peripherals including a GPS, magnetometer, accelerometer, gyroscope, barometer and motor controllers. These peripherals allow the PX4 autopilot to have control of the drone. The PX4 has two different interfaces from which it can be commanded. The first interface is a 2.4 GHz radio receiver allowing a pilot to command the PX4 with a remote controller. The second interfaces is a HSUART serial connection allowing an onboard companion computer to send commands to it. The communication protocol used on the HSUART connection is the open source MAVLink protocol.

The PX4 autopilot can primarily be operated in two different modes. A position mode, in which it can be commanded to go to specific positions, and a stabilise mode in which the drone can be commanded to maintain specific attitude references. For academic reasons it is decided to use the PX4 in the stabilised mode which requires a position controller to be designed and implemented, providing attitude references to the PX4 autopilot.

When the PX4 is operating in stabilised mode it is using an attitude controller which relies on a quaternion based attitude estimator [18]. This attitude estimator is using inputs from the gyroscope to estimate the attitude of the drone. The attitude estimator is working by integrating the angular velocities measured by the gyroscope to obtain the attitude of the drone. The reason why such integrated gyroscope measurements works as the attitude estimate, is due to an initial calibration of the drone performed before take-off using the accelerometer. The heading of the drone cannot be calibrated in the same way as the pitch and roll angles. Therefore, the PX4 is either relying on a heading estimate from the magnetometer or an external heading measurement received through the MAVLink connection.

The magnetometer is deemed unusable in the analysis of the environment performed in Section 2.3.This means that the PX4 has to be supplied with heading estimates before the attitude controller can be expected to function properly.

4.2 Intel Aero compute board

The Intel Aero compute board is an onboard companion computer built and intended specifically for Aerial vehicles. The board is powered by a quad core Intel processor and contains a variety of peripherals which makes the board suited for unmanned aerial vehicle operations. The operations system running on the Intel processor is the open source Yocto project [19] which is a Linux operating system intended for embedded applications.
Besides the processor, the board is equipped with 4 GB of memory and 32 GB of storage. The important peripherals for this project is the HSUART serial connection to the Intel Aero Flight controller. This connection allows applications running on the Intel Aero compute board to send commands to the PX4 autopilot. The Intel Aero compute board will be referred to as the companion computer.

A second important peripheral being used in this project is the USB 3.0 connection, connecting the compute board with the Intel RealSense R200 RGB-D camera. Another peripheral used is a wireless module which is configured to act as a wireless WiFi hotspot. This allows other computers to connect to the compute board and communicate with applications running on the companion computer through either a TCP or UDP connection. The wireless connection is an important feature in relation to receiving position measurements from the GOT system which is connected to a separate computer as explained in Section 2.2.

The Robot Operating System (ROS), see Appendix I, is installed by default on the companion computer. This gives some advantages when developing and debugging the project. First of all it gives access to already existing robot related libraries. Secondly ROS provides a message passing system which is based on a publish and subscribe policy. This message passing system allows easy inter-task communication and furthermore the opportunity to record and playback a session through so-called rosbags. Thirdly an interface between the ROS environment and the PX4 autopilot, using MAVLink, already exist. This interface is called MAVROS and is implemented as a ROS node within the ROS system. This allows two way communication between any ROS node and the PX4. The MAVROS node works as a bridge between ROS nodes and the PX4, doing all the translation from ROS messages into MAVLink packages. Furthermore the MAVROS node can also be used as an interface to the Gazebo drone simulator described in Appendix J.

4.3 Intel RealSense R200 RGB-D camera

To get measurements of landmarks within the environment, a camera-based solution capable of providing both RGB and depth images is needed. An important peripheral to this project is therefore the RealSense R200 RGB-D camera installed on the the Intel Aero drone.
Capturing depth of an environment can be done in several ways, where most commercial solutions either contain a stereo camera pair, a structured light (projective) solution or Time-Of-Flight technology. The R200 camera consists of one RGB camera with a resolution up to 1920×1080, two infrared (IR) cameras with a resolution up to 640×480, mounted in a stereo configuration, and finally an infrared projector. With enough IR light present in the environment the two IR cameras are enough to capture depth of a scene. However, to make sure that surfaces with a plain texture can be captured as well, an IR laser projector emits a grid of IR lines.

The camera also include a built in ASIC processor doing all necessary preprocessing, rectification, registration and disparity map generation using the IR stereo image pair. The result is a non-distorted depth image where each pixel contain a depth measurement in millimeters. By the help of the ASIC processor, depth images can be provided at full frame-rates of up to 60 FPS. The camera comes pre-calibrated from Intel allowing both intrinsics and extrinsics of the cameras and processed depth image to be extracted. Furthermore ROS libraries and processing nodes allow easy capture of both RGB and depth images, using the image transport protocol being a part of the ROS environment. Unfortunately at the time of development and writing the release of the RealSense camera nodelet for ROS contains an error such that the image registration between the RGB and depth image seem to be misaligned.

A dedicated node is therefore developed within this project to handle the registration and alignment of the RGB and depth image, as described in Section F.5. The result is a 2-dimensional array aligned with the RGB image. Within the 2-dimensional array each element contains a 3-dimensional measurement vector including the corresponding depth pixel location and depth measurement. This allows a feature detector to find certain features within the RGB image and then look up the corresponding 3-dimensional measurement vector within the 2-dimensional array, such that measurements can be provided to the FastSLAM implementation. An example of an RGB and depth image, where the depth is overlaid and visualized as greyscale, with detected ArUco markers and the corresponding measurement vectors, is shown in Figure 4.1.

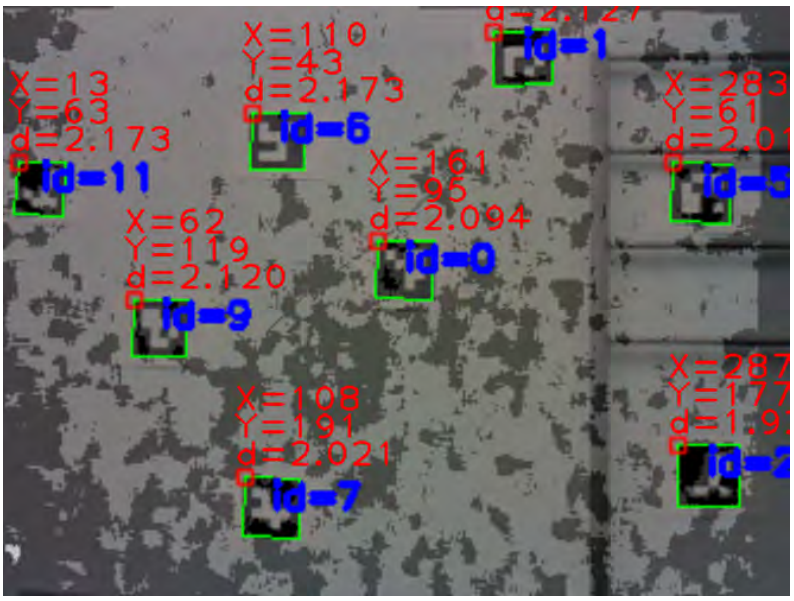


Figure 4.1: Measurement vectors printed next to detected ArUco markers within a combined visualization of the RGB and depth image. Dark areas of the image are due to the wall in the image being outside the range of the depth camera.

Further details of the R200 camera, the functionality of the camera, the factory calibrated parameters and the developed processing node within ROS, can be found in Appendix F.

5 System Overview

The goal of this project is to achieve a robust and error tolerant indoor position estimation for drones, such that the position can be stabilized as described in Section 2.1 even when indoor position measurements becomes unreliable. In this chapter an overview of the proposed solution, designed for the platform described in Chapter 4, is given. Based on the analysis and presentation throughout Chapter 2 to Chapter 4, a list of assumptions and conditions to expect is initially put up. Thereafter some structural decisions are taken, describing how the chosen FastSLAM algorithm fits together with the chosen platform and controllers to implement. The structural decisions are followed by a presentation of the different elements of the system resulting in the system overview diagram shown in Figure 5.2, representing the individual elements of the proposed solution. In the end of the chapter an implementation layout diagram of the system is presented, showing the different hardware and software layers used when implementing and integrating the different parts of the system.

5.1 Assumptions and conditions

The assumptions and conditions under which the developed system should work is presented in the list below. These requirements are all derived from the problem analysis and platform description presented in Chapter 2 to Chapter 4.

- The drone should include a PX4 flight controller capable of stabilizing the attitude. This flight controller needs at least an attitude controller and an attitude estimator using IMU measurements.
- The drone should be equipped with an RGB-D camera and a programmable companion computer running Linux and capable of running ROS.
- The considered environment includes factories and warehouses. These environments will likely contain large amounts of metal affecting the magnetometer readings. Thus measurements from the magnetometer cannot be used.
- The desired trajectory, that the drone should fly, is defined as position references at way-points where in between the drone can fly in straight lines.
- The GOT system should cover the full operating environment only with smaller deadzones. When the system is working the accuracy is expected to be down to 10 mm.
- The GOT measurements are provided at 10 Hz.
- ArUco markers should be installed in the environment such that at least one marker is always visible to the camera from any possible position close to hover.
- When taking off GOT measurements should be available.

5.2 Solution overview

As concluded in Chapter 3 the problem of estimating the position of the drone using only relative measurements is known as the SLAM problem, and the Filtering-based FastSLAM 2.0 algorithm, in the sequel referred to as FastSLAM, is chosen as the real-time estimator to provide the position estimates necessary to stabilize the attitude and position of the drone. The full system is developed on the Intel Aero Ready to Fly drone incorporating a PX4 flight controller however, this flight controller will only be used for stabilizing the attitude of the drone. Attitude set-points for the PX4 are generated by a designed position controller using Full-State estimates generated by an Extended Kalman Filter estimator. As the absolute heading can not be observed by the PX4 flight controller, heading estimates are provided back to the PX4 by the estimator. The estimator will furthermore provide attitude and velocity estimates to FastSLAM to reduce the computational requirement and allowing a simpler motion model to be used. This results in the following structural overview diagram connecting the individual elements to be developed within this project.

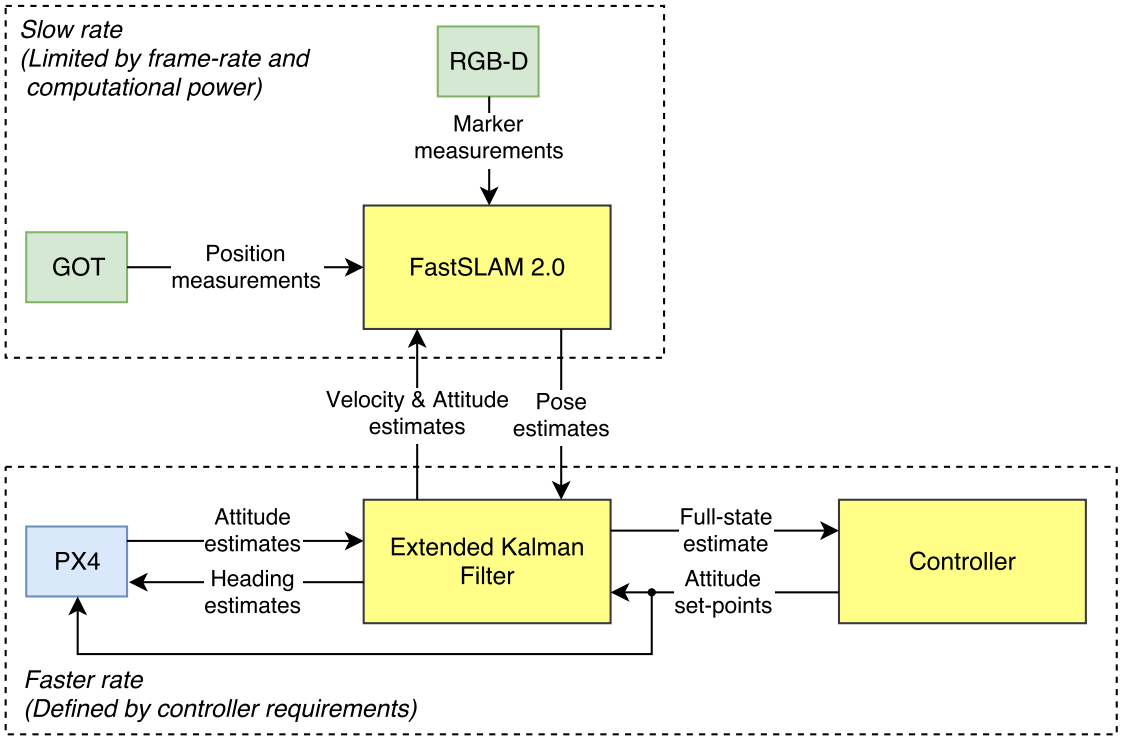


Figure 5.1: Structural overview of proposed solution

The following sections describes the proposed solution composed of the individual blocks, the interconnections, the decisions made and reasoning behind. This presentation of the solution provides an overview of each block and the signal content between each, leading up to the final system overview diagram shown in Figure 5.2.

5.2.1 Controlling the attitude

One of the advantages of choosing the Intel Aero platform is the included PX4 flight controller, which is an open source flight controller widely used in the development of UAVs. The PX4 flight controller hardware comes equipped with accelerometers, gyroscopes and the unusable magnetometers, as described in Section 2.3.

The PX4 flight controller software includes an attitude estimator and an attitude controller. The attitude estimator is used to estimate roll, pitch and yaw angles of the drone based on measurements provided by the IMU, i.e. measurements of drone acceleration, $\ddot{x}$, $\ddot{y}$ and $\ddot{z}$, provided by the accelerometer and angular velocities of the drone, $\dot{\phi}$, $\dot{\theta}$ and $\dot{\psi}$, provided by the gyroscope. However, the absolute yaw angle can not be estimated from any of the internal sensors, since the magnetometer can not be used in the described environment, why yaw angle estimates should be provided as inputs to the internal PX4 attitude estimator such that the attitude controller is able to control the heading as well.

Before taking off the accelerometer provides a good measurement of the attitude which is used to calibrate the internal roll and pitch estimate of the PX4. However when the drone is flying it is assumed that these estimates will slowly drift as described in Section 2.1, which is important to consider when designing the estimator and position controller. The PX4 attitude controller, fed with attitude and thrust references as shown in (5.1), will hereafter generate the necessary signals for the four motor controllers.

$$\boldsymbol{u}_{\text{PX4}} = \begin{bmatrix} \phi_{\text{ref}} \\ \theta_{\text{ref}} \\ \psi_{\text{ref}} \\ T_{\text{ref}} \end{bmatrix} \tag{5.1}$$

Where:

$\boldsymbol{u}_{\text{PX4}}$	is the input reference vector for the PX4 attitude controller
ϕ_{ref}	is the roll reference for the attitude controller in the PX4
θ_{ref}	is the pitch reference for the attitude controller in the PX4
ψ_{ref}	is the yaw reference for the attitude controller in the PX4
T_{ref}	is the trust reference for the attitude controller in the PX4

5.2.2 Controlling the position

One of the goals of this project is for the drone to be able to hover and hold its' position at a defined coordinate in space, i.e. x_{ref}, y_{ref} and z_{ref}, with a desired heading angle, ψ_{ref}. Another goal is to let the drone follow predefined trajectories which can be tracked even at loss of GOT position measurements, where the trajectories are defined with way-points of the reference input vector shown in (5.2).

$$\boldsymbol{s}_{\text{ref}} = \begin{bmatrix} x_{\text{ref}} \\ y_{\text{ref}} \\ z_{\text{ref}} \\ \psi_{\text{ref}} \end{bmatrix} \tag{5.2}$$

Where:

$\boldsymbol{s}_{\text{ref}}$	is the reference vector for the developed position controller
x_{ref}	is the reference of the x-coordinate for the developed position controller
y_{ref}	is the reference of the y-coordinate for the developed position controller
z_{ref}	is the reference of the z-coordinate for the developed position controller
ψ_{ref}	is the yaw reference for the developed position controller

To fulfil these goals a position controller is developed to generate attitude references for the PX4 flight controller, hence a controller giving the ϕ_{ref}, θ_{ref}, ψ_{ref} and thrust set-points to the PX4 as defined in (5.1). For such a controller to be stable a proper position estimator, working even at loss of GOT measurements is needed. As described in Section 2.1 the position estimate will drift if no other sensors are used, why a camera-based solution is developed using RGB-D measurements and the FastSLAM algorithm.

5.2.3 Computational considerations

Due to the fact that the FastSLAM algorithm is highly computational, a series of decisions are taken in order to increase the performance of the algorithm itself and also to increase the accuracy of the state estimate that is fed to the controller.
The model of the drone used in this project is presented in Section 6.1. The state vector, $\boldsymbol{\chi}$, of this model includes the position, i.e. ${}^{\text{H}}x$ and ${}^{\text{H}}y$ in the heading frame and z in Earth frame, linear velocities, i.e. ${}^{\text{H}}\dot{x}$ and ${}^{\text{H}}\dot{y}$ in the heading frame and $\dot{z}$ in Earth frame, and the drone attitude, i.e. ϕ, θ and ψ plus some additional states.
Based on the update rate of FastSLAM that others have obtained when implementing the algorithm for estimating less states [14, 20], it is assumed that FastSLAM will not be able to provide a full state estimate at a rate sufficient for controlling a drone. The update rate of FastSLAM will in this project further be limited by the image processing of the RGB-D images. Besides that, it is anticipated that the update rate of FastSLAM will be sporadic, due to different amount measurements of landmarks being processed at each iteration, which is usually not desired in a control system. Therefore it is chosen to use a Extended Kalman filter (EKF) to estimate the full state vector, $\hat{\boldsymbol{\chi}}$, needed for the controllers, in such a way that a higher and constant update rate is obtained. Thereby it is possible to reduce the state vector estimated by FastSLAM, to only include states that cannot be estimated precisely otherwise, thus saving computational power.

5.2.4 Full-State Extended Kalman Filter

As mentioned a yaw angle estimate has to be fed back to the PX4 to ensure that heading references are tracked by the internal attitude controller. FastSLAM is expected to run at a slower rate than both the internal attitude controller and the position controller due to computational limitations and limitations of the RGB-D frame-rate. It is therefore decided to design a Full-State Extended Kalman Filter (EKF) running at a faster rate than FastSLAM, to estimate all necessary states for the position controller and the necessary yaw angle for the attitude controller within the PX4.

The EKF estimator is based on a model of the drone including the PX4 attitude controller, such that predictions can be made at a sufficiently fast rate. The EKF takes in the drifting attitude estimates from the PX4 flight controller and the pose estimate from FastSLAM, as sensor measurements. The particles within FastSLAM estimates the full probability density function (PDF) of the pose vector used in the algorithm. Thus to be able to use this estimate in the EFK the sample mean and covariance of this PDF is calculated based on the particles of the filter. Thereby approximating the PDF estimated by FastSLAM as Gaussian distributions.
The resulting estimates provided by the EKF includes both the attitude, position and velocity of the drone in the local heading frame. The full state vector, $\hat{\boldsymbol{\chi}}$, is provided for the position controllers, and the yaw angle is supplied back to the attitude estimator of the PX4.

5.2.5 Including attitude estimates into FastSLAM

As the position estimates from FastSLAM are fed back into the EKF to correct the attitude estimate, one must assume that the EKF is capable of correctly estimating the roll and pitch angles of the drone, which in any case is assumed to be close to hover at all times. Consequently there is no need for FastSLAM to estimate the roll and pitch angles as well, because an estimate of roll and pitch is determined by the EKF at a faster rate, which can be provided as inputs to FastSLAM. This allows the pose state vector, $\boldsymbol{s}$, within FastSLAM to be reduced to only four states, as shown in (5.3), being the 3D position of the drone, x, y, z, and the yaw angle, ψ, indicating the heading of the drone. Notice that it is still necessary to include the heading as only RGB-D measurements of the environment can provide information about the absolute yaw angle.

$$\boldsymbol{s} = \begin{bmatrix} {}^{\mathrm{E}}x \\ {}^{\mathrm{E}}y \\ {}^{\mathrm{E}}z \\ \psi \end{bmatrix} \tag{5.3}$$

Where:

$\boldsymbol{s}$	is the state vector, which FastSLAM is used to estimate
${}^{\mathrm{E}}x$	is the x-coordinate of the drone in the earth frame
${}^{\mathrm{E}}y$	is the y-coordinate of the drone in the earth frame
${}^{\mathrm{E}}z$	is the z-coordinate of the drone in the earth frame
ψ	is the rotation of the heading frame around the z-axis of the earth frame

The position of the drone is given in the Earth frame, see Appendix A. If no frame is defined for the variable the Earth frame is used.

5.2.6 Simplified motion model

Estimating the full state vector with an EKF allows a different motion model to be used in the FastSLAM. With the state estimates from the EKF, a simple kinematic motion model can be used in FastSLAM to predict the pose.
Within Section A.3 the dynamics of the drone are modelled as a black-box ARX model, being a motion model from attitude reference inputs to the attitude, velocity and positions. This motion model is used for predictions in the EKF but includes a lot more states than the six states describing the position and orientation of the drone, constituting the pose within FastSLAM. If this motion model is to be used within FastSLAM the state vector would have to be expanded unnecessarily only to require a higher number of particles or resulting in a less robust pose estimate.

Furthermore the ARX model is determined for the sample rate of the EKF, which is higher than the expected FastSLAM sample rate, why the motion model would also have to be resampled to be usable. Therefore it is decided to use a simple kinematic motion model for the prediction step within FastSLAM, based on the linear velocity and yaw estimates from the EKF as shown in (5.4).

$$\boldsymbol{u}_{\text{SLAM}} = \begin{bmatrix} {}^{\text{H}}\hat{x} \\ {}^{\text{H}}\hat{y} \\ {}^{\text{H}}\hat{z} \\ \hat{\psi} \end{bmatrix} \tag{5.4}$$

Where:

$\boldsymbol{u}_{\text{SLAM}}$	is the vector signals of that the motions model used in the FastSLAM algorithm
${}^{\text{H}}\hat{x}$	is the x-coordinate of the drone in the heading frame
${}^{\text{H}}\hat{y}$	is the y-coordinate of the drone in the heading frame
${}^{\text{H}}\hat{z}$	is the z-coordinate of the drone in the heading frame
$\hat{\psi}$	is the EKF estimate of the rotation of the heading frame around the z-axis of the earth frame

5.2.7 Including RGB-D measurements

Within the environment a set of static ArUco markers, see Section 3.3.1, are placed such that at least one marker is always visible to the drone and that the markers are visible from different viewing angles. The ArUco markers allow detection and unique identification within the RGB image from the RGB-D camera. For the ease of the implementation, the feature extraction and identification is implemented in OpenCV, being an integrated part of ROS. For further details on the use of ArUco marker and OpenCV in this project see Appendix H. The measurement vector of a detected marker constitutes a converted 2D pixel location of the marker, corresponding to the marker coordinate within the depth image, x_c and y_c, and a measured depth to the landmark, d, as shown in (5.5).

$$\boldsymbol{z}_c = \begin{bmatrix} x_c \\ y_c \\ d \end{bmatrix} \tag{5.5}$$

Where:

$\boldsymbol{z}_c$	is the measurement vector of the RGB-D measurements
x_c	is the x-coordinate of a landmark in the camera frame
y_c	is the y-coordinate of a landmark in the camera frame
d	is the measured depth to of a landmark in the camera frame

Such measurements can be used within the FastSLAM algorithm as relative measurements to detected landmarks present within the map of FastSLAM. Using the pixel locations of detected markers and their corresponding depth measurements the detected markers can be converted into 3D coordinates relative to the camera frame. Furthermore to convert these 3D coordinates relative to the camera frame into world frame, the roll and pitch has to be known. Roll and pitch are therefore included as input to the FastSLAM 2.0 algorithm.

5.2.8 Including GOT measurements

Whenever GOT position measurements are available these should be included into the FastSLAM algorithm in order to increase the estimate accuracy and help with loop closures. This requires that the GOT measurements are included as a main part of the FastSLAM implementation as another type of measurement. The coordinate system of the FastSLAM implementation should align with the GOT coordinate system, as position references (trajectories) are given within this system, see Section 2.3. The GOT position measurements can therefore be seen as relative measurements to a fixed GOT origo, which should be included into FastSLAM as a known landmark at position $(0, 0, 0)$ with zero covariance. The resulting measurement vector constitutes the GOT position measurement of the drone relative to the GOT frame, being x_G, y_G and z_G, as shown in (5.6).

$$\boldsymbol{z}_G = \begin{bmatrix} x_G \\ y_G \\ z_G \end{bmatrix} \tag{5.6}$$

Where:

$\boldsymbol{z}_G$	is the measurement vector of the RGB-D measurements
x_d	is the x-coordinate of the drone in the GOT frame
y_d	is the y-coordinate of the drone in the GOT frame
z_d	is the z-coordinate of the drone in the GOT frame

5.3 System diagram

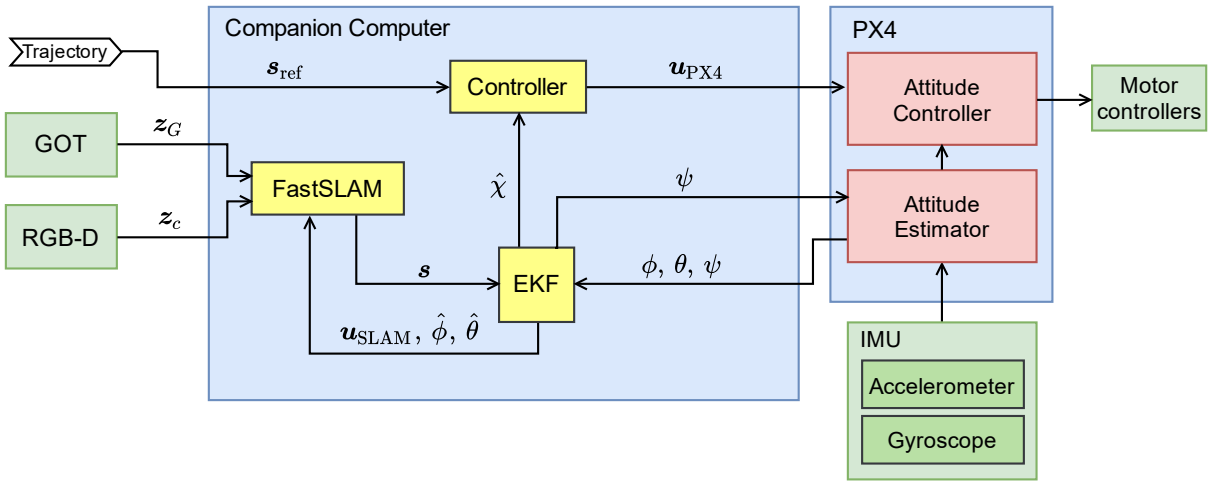


Figure 5.2: System overview diagram

5.4 Implementation considerations

Now that the overview of the system is given it is important to consider how the system can be implemented on the given platform. The individual elements of the proposed solution, being the Controller, EKF and FastSLAM, should be implemented inside the on-board Intel Aero compute-board, which is connected to the PX4 and the rest of the needed peripherals.

Both the PX4 MAVLink connection, the RealSense R200 camera and the Intel Aero compute-board is supported by the ROS environment. It is decided that the proposed solution should be developed and fully contained within ROS. As explained in Appendix I, ROS is an open-source, meta-operating system for robots allowing easier and faster development using many pre-existing libraries, visualization tools and debugging tools. Furthermore the ROS environment provides real-time threading capabilities which is desirable for robot and control applications, as well as all necessary tools and libraries for writing, building and running code across multiple computers in a distributed ROS environment. The individual elements of the system can be developed and implemented as separate tasks, running in parallel. All tasks are able to share messages with each other within the ROS environment by publishing and subscribing to so-called topics.

An overview of the ROS system to be developed is shown in Figure 5.3, including the three main nodes of the system and other necessary nodes to access external peripherals used by the system.

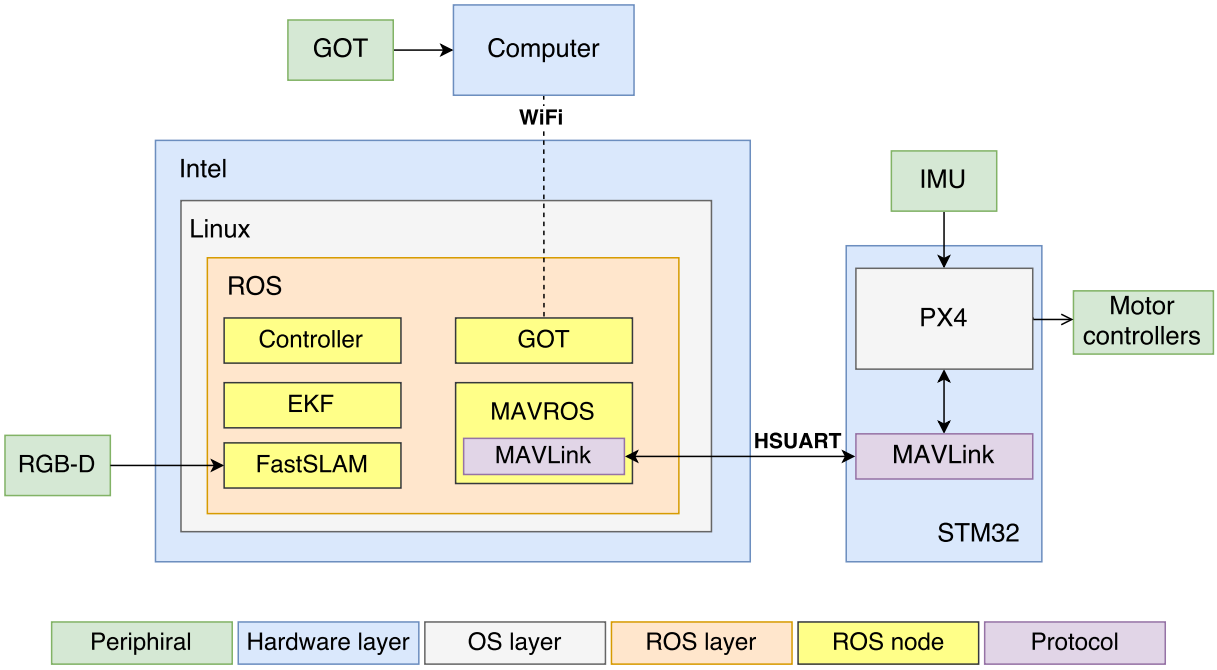


Figure 5.3: Implementation layout within ROS

Another beneficial tool being an integrated part of the ROS environment is the Gazebo simulator including a physics engine and 3D visualization. This simulator can be coupled together with Software in the Loop simulation of the PX4 to give an exact simulation environment of a PX4-based drone equipped with a companion computer running ROS, see Appendix J. This allows simulations and verification of the implemented controllers, estimators and the FastSLAM algorithm, solely using simulated IMU and RGB-D camera measurements. This both saves time and is especially a safe way of doing the initial tests of new implementations to avoid sudden crashes of the drone due to coding errors.
In the following chapters the development of each element of the system, composed by the Controller, FastSLAM and the Extended Kalman Filter, is carried out and described.

6 Controller

In this chapter the control solution for the drone is presented. Firstly, with the purpose of designing the controllers, the behaviour of a drone is described with a model of it.
From the system description in Chapter 5 and illustrated in Figure 5.2, it is known that the controller block should stabilize the position of the drone and reach position references given as way-points, defined in world coordinates x, y and z. With these considerations and using the model found, it is proceeded with the design of the controllers. Finally, the performance of these controllers is tested with the actual drone. To do so, the controllers are first implemented in C++ code as a node in ROS. Their implementation and performance are tested together in a simulation environment which is chosen to be Gazebo, a physics simulator that allows to interface with ROS described in Appendix J.

6.1 Drone Model

In this section, a model to describe the behaviour of a drone is derived. In Appendix A, a first principle based description of a quadrotor is presented. For making the controller design easier, a linearised version of the model found is presented in this section following the derivation from Section A.2. This simplified model is finally expressed in a state-space form for the design of the controllers.

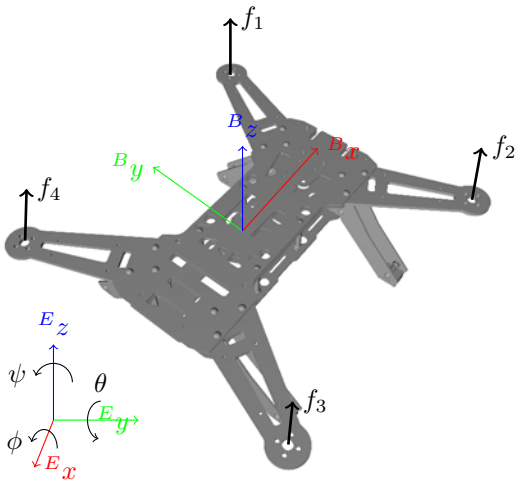


Figure 6.1: Drone description. The different frames and rotations considered for the modelling are shown as earth frame (EF) and body frame (BF). The rotors of the drone are numbered from 1 to 4.

The description of a drone as seen from a static fixed frame defined as the earth frame (EF) is achieved by describing its pose. This is defined as the combination of the position, ${}^E\boldsymbol{\xi}$, and the orientation, $\boldsymbol{\eta}$. The pose is then what describes the variables of the system.

$$ {}^{\mathrm{E}}\boldsymbol{\xi} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \boldsymbol{\eta} = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \tag{6.1} $$

Where:

${}^{\mathrm{E}}\boldsymbol{\xi}$	describes the position of the drone in an earth defined frame
$\boldsymbol{\eta}$	describes the orientation of the drone

The position of the drone is described the drones coordinates in the axes x, y and z, and the orientation is described as the rotation around each of the axes, as shown in Figure 6.1. The rotation ϕ around x is defined as roll, θ around y is defined as pitch and ψ around z as yaw.
The linearised version of the model found describes the drone with the variables in the system decoupled if a frame that only follows the yaw movement of the drone is considered. This frame, defined as the heading frame (HF), together with the assumptions presented in Section A.2, leads to the equations describing the model shown in (6.2).

$$ \begin{aligned} \boldsymbol{I}\ddot{\boldsymbol{\eta}} &= \boldsymbol{\tau} \\ m\ddot{z} &= T - mg \\ \begin{bmatrix} {}^{\mathrm{H}}\ddot{x} \\ {}^{\mathrm{H}}\ddot{y} \end{bmatrix} &= g \begin{bmatrix} \theta \\ -\phi \end{bmatrix} \end{aligned} \tag{6.2} $$

Where:

$\boldsymbol{I}$	is a diagonal matrix defining the moment of inertia of the drone	$\left[\mathrm{kg\,m^2}\right]$
m	is the mass of the drone	[kg]
g	is gravity	$\left[\mathrm{m/s^2}\right]$
$\boldsymbol{\tau}$	is the resultant torque vector	[N m]
T	is the thrust	[N]

The input to the system is defined as the combination of $\boldsymbol{\tau}$ and T and the output as $\boldsymbol{\xi}$ and $\boldsymbol{\eta}$ described in the HF.
However, as explained in Section 4.1, the drone being used has a PX4 Flight controller implemented which requires changes in the description of the model. The controller actuates over the rotor speeds, which are related to $\boldsymbol{\tau}$ and T as explained in Section A.1. However, the inputs given to the flight controller vary depending on the flying mode being used. When flying in stabilised mode, these inputs are given with a remote controller, and consist on roll and pitch references, yaw rate references and a scaled thrust input. According to [21] the control structure in this mode is a combination of P controllers for roll, pitch and yaw angles and PID controllers for the angular rates.

Since the parameters describing these controllers are not known, it is decided to find a description of the drone with an empirical-based model. It is chosen to find a black-box model described with auto-regressive with exogenous input (ARX) type of models by fitting data extracted from flight tests of the drone using the PX4 controller. This fitting assumes no dynamics on the noise or disturbances to the system, allowing to find a unique fitting solution using linear regression. Assuming no noise dynamics, since the model description is linear, using ARX models will allow to find a unique solution for the fitting. The description of the ARX models used is found in Section A.3, and its derivation is based on the linearised model described. The plots showing the fit of the models found are shown in Section A.3 together with the models themselves.

The experiments performed to find the models are done in an environment equipped with a Vicon system that allows the extraction of the position and orientation of the drone, which then can be used to fit a model for the variables of the system with the inputs given to the PX4 in each experiment. For the roll, pitch and $\ddot{z}$ models, the flight is performed in stabilised mode with a remote controller.

To obtain measurements to fit a model from yaw reference to yaw, a different strategy has to be used, since it is not possible to give a yaw reference as input in stabilised mode while flying manually. Therefore measurements for fitting the yaw model are obtained by getting the drone stabilised in a chosen position with the PX4 position hold mode and giving a set of yaw references as input. This allows a model for yaw having yaw references as input. Assuming small angle approximation each of the variables, ϕ, θ, ψ and $\ddot{z}$, is independently related to one of the reference inputs given in the tests performed with the PX4 controller. Thus an independent model can be found between each of the variables and an input. Afterwards, the models for x, y and z can be found by making a discrete integration of their second derivative models twice, following what is described in by (6.2), leaving the models found for x and y only valid in the HF. The variables of the system can then be represented in the form shown in (6.3), where each transfer function, $G(\mathrm{z})$, represents the discrete transfer function found with the ARX model fitting.

$$\begin{aligned}
\phi(\mathrm{z}) &= G_\phi(\mathrm{z})\phi_{\mathrm{ref}}(\mathrm{z}) \\
\theta(\mathrm{z}) &= G_\theta(\mathrm{z})\theta_{\mathrm{ref}}(\mathrm{z}) \\
\psi(\mathrm{z}) &= G_\psi(\mathrm{z})\psi_{\mathrm{ref}}(\mathrm{z}) \\
x(\mathrm{z}) &= \frac{gT_s^2}{(\mathrm{z}-1)^2}\theta(\mathrm{z}) \\
y(\mathrm{z}) &= \frac{-gT_s^2}{(\mathrm{z}-1)^2}\phi(\mathrm{z}) \\
z(\mathrm{z}) &= \frac{T_s^2}{(\mathrm{z}-1)^2}G_{\ddot{z}}(\mathrm{z})T(\mathrm{z})
\end{aligned} \tag{6.3}$$

Where:

$G_\bullet(\mathrm{z})$	is the transfer function corresponding to the output $\bullet$ in the z domain
T_s	is the sampling time of the system

The ARX representation of the models can be rewritten in a state-space form as in (6.4) and the transformation is performed as explained in Section A.4. The sample rate used for extracting the data to fit in the models is 20 Hz.

$$\begin{aligned}
\boldsymbol{\chi}^{k+1} &= \boldsymbol{A}\boldsymbol{\chi}^k + \boldsymbol{B}\boldsymbol{u}^k \\
\boldsymbol{y}^k &= \boldsymbol{C}\boldsymbol{\chi}^k
\end{aligned} \tag{6.4}$$

This new expressions define four systems with ψ, θ, ϕ and $\ddot{z}$ as outputs, each with a corresponding input. This implies that the state vector $\boldsymbol{\chi}$ of each state space system does not need to have a defined meaning related to the model. These state-space representations are combined into a full state-space representation of the whole system by stacking the states, defining the state vector $\boldsymbol{\chi}$ as shown in (6.5).

$$\boldsymbol{\chi} = \begin{bmatrix} {}^{\mathrm{H}}x & {}^{\mathrm{H}}\dot{x} & \boldsymbol{\chi}_\theta & | & {}^{\mathrm{H}}y & {}^{\mathrm{H}}\dot{y} & \boldsymbol{\chi}_\phi & | & \psi & | & z & \dot{z} & \chi_{\ddot{z}} \end{bmatrix}^T \tag{6.5}$$

The combined state-space representation is shown in (6.6), where the matrices and state vectors corresponding to each of the four systems are described with the subscript corresponding to the output they are related to. The derivation is described in further details in Section A.4. As described in (6.3), expressions for x, y and z are achieved with discrete integrations.

$$
\begin{bmatrix} {}^{\mathrm{H}}x \\ {}^{\mathrm{H}}\dot{x} \\ \boldsymbol{\chi}_\theta \\ \hline {}^{\mathrm{H}}y \\ {}^{\mathrm{H}}\dot{y} \\ \boldsymbol{\chi}_\phi \\ \hline \psi \\ \hline z \\ \dot{z} \\ \chi_{\ddot{z}} \end{bmatrix}^{k+1}
=
\left[\begin{array}{ccc|ccc|c|ccc}
1 & T_s & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 0 & 0 \\
0 & 1 & gT_s\boldsymbol{C}_\theta & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 0 & 0 \\
0 & 0 & \boldsymbol{A}_\theta & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & \mathbf{0}_{1\times4} & 1 & T_s & \mathbf{0}_{1\times4} & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{0}_{1\times4} & 0 & 1 & -gT_s\boldsymbol{C}_\phi & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \boldsymbol{A}_\phi & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & A_\psi & 0 & 0 & 0 \\ \hline
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 1 & T_s & 0 \\
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 1 & T_sC_{\ddot{z}} \\
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 0 & A_{\ddot{z}}
\end{array}\right]
\begin{bmatrix} {}^{\mathrm{H}}x \\ {}^{\mathrm{H}}\dot{x} \\ \boldsymbol{\chi}_\theta \\ \hline {}^{\mathrm{H}}y \\ {}^{\mathrm{H}}\dot{y} \\ \boldsymbol{\chi}_\phi \\ \hline \psi \\ \hline z \\ \dot{z} \\ \chi_{\ddot{z}} \end{bmatrix}^{k}
+
\left[\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & \boldsymbol{B}_\theta & 0 & 0 \\ \hline
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\boldsymbol{B}_\phi & 0 & 0 & 0 \\ \hline
0 & 0 & B_\psi & 0 \\ \hline
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & B_{\ddot{z}}
\end{array}\right]
\begin{bmatrix} \phi_{\mathrm{ref}} \\ \theta_{\mathrm{ref}} \\ \psi_{\mathrm{ref}} \\ T \end{bmatrix}^{k}
+
\begin{bmatrix} {}^{\mathrm{H}}x \\ {}^{\mathrm{H}}y \\ z \\ \phi \\ \theta \\ \psi \end{bmatrix}^{k}
=
\left[\begin{array}{ccc|ccc|c|ccc}
1 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{0}_{1\times4} & 1 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 1 & 0 & 0 \\
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \boldsymbol{C}_\phi & 0 & 0 & 0 & 0 \\
0 & 0 & \boldsymbol{C}_\theta & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & C_\psi & 0 & 0 & 0
\end{array}\right]
\begin{bmatrix} {}^{\mathrm{H}}x \\ {}^{\mathrm{H}}\dot{x} \\ \boldsymbol{\chi}_\theta \\ \hline {}^{\mathrm{H}}y \\ {}^{\mathrm{H}}\dot{y} \\ \boldsymbol{\chi}_\phi \\ \hline \psi \\ \hline z \\ \dot{z} \\ \chi_{\ddot{z}} \end{bmatrix}^{k}
\tag{6.6}
$$

Where:

- $\boldsymbol{\chi}_\bullet$ is the state vector of the state-space representation corresponding to the ARX output $\bullet$
- $\boldsymbol{A}_\bullet$ is the system matrix of the state-space representation corresponding to the ARX output $\bullet$
- $\boldsymbol{B}_\bullet$ is the input matrix of the state-space representation corresponding to the ARX output $\bullet$
- $\boldsymbol{C}_\bullet$ is the output matrix of the state-space representation corresponding to the ARX output $\bullet$

From this description of the drone, it is evident that the system can be partitioned. It is decided to split it into three systems: one describing the yaw movement of the drone, another describing the movement in the z direction and a system describing the x and y movement of the drone. since yaw has been modelled from ψ_{ref} to ψ, and because the PX4 has a build in controller for yaw, no control solution is designed for this state.
The following sections presents a control solution for z named z controller and another one for x and y named x-y controller. These controllers will be designed to run at 20 Hz, corresponding to the sample rate from the models.

6.2 Z Controller

In this section the control solution for the z height of the drone is analysed, designed and tested. From Section 6.1 it is known that the acceleration of the drone in z can be controlled through the thrust input. Furthermore, a state space model of the z system has been presented, which is derived from an identified ARX model. This ARX model is identified around a thrust value which makes the drone hover. Therefore, the thrust value has to be added to the control signal determined by the controller as an operating point. This value is expected to deviate while flying mainly due to changes in the state of charge in the battery. The derivations from the hover value are affecting the z system as a disturbance on the input thrust signal.

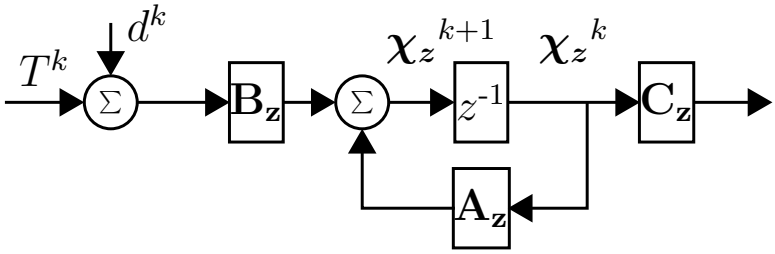


Figure 6.2: Open loop system.

$$\underbrace{\begin{bmatrix} z \\ \dot{z} \\ \chi_{\ddot{z}} \end{bmatrix}^{k+1}}_{\boldsymbol{\chi}_z^{k+1}} = \underbrace{\begin{bmatrix} 1 & T_s & 0 \\ 0 & 1 & T_s C_{\ddot{z}} \\ 0 & 0 & 0 \end{bmatrix}}_{\boldsymbol{A}_z} \underbrace{\begin{bmatrix} z \\ \dot{z} \\ \chi_{\ddot{z}} \end{bmatrix}^{k}}_{\boldsymbol{\chi}_z^{k}} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ B_{\ddot{z}} \end{bmatrix}}_{\boldsymbol{B}_z} (T^k + d^k) \tag{6.7}$$

$$\boldsymbol{\chi}_z^{k+1} = \boldsymbol{A}_z \boldsymbol{\chi}_z^k + \boldsymbol{B}_z T^k + \boldsymbol{B}_z d^k \tag{6.8}$$

Where:

$\boldsymbol{\chi_z}$	is the state vector of the z system
T	is the applied thrust signal
d	is the thrust input disturbance

6.2.1 Feedback design

In order to track the z set-point a reference has to be introduced. From (6.3) and (A.30) it is evident that the open loop transfer function from thrust input T to z is a type 2 system since the discrete transfer function is having two poles in $z = 1$. Hence, no integral action in the controller is needed in order to reach a type 1 system that would yield zero steady-state error to step inputs. In chapter Chapter 8, an extended Kalman filter estimating all the states in the state vector $\boldsymbol{\chi_z}$ is designed. The feedback is designed as if the state vector $\boldsymbol{\chi_z}$ is available without considering the error associated with state estimation. The feedback law is then defined as in (6.9). An illustration of the closed loop system is shown in Figure 6.3.

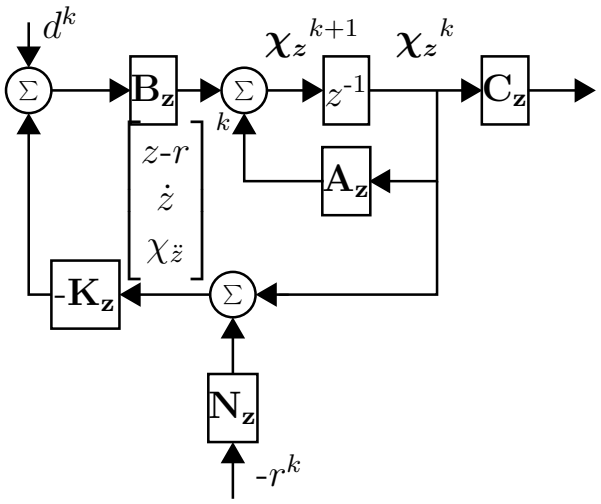


Figure 6.3: Closed loop system.

$$T^k = -\boldsymbol{K}_z\boldsymbol{\chi}_z^k + \boldsymbol{K}_z\boldsymbol{N}_z r^k \tag{6.9}$$

$$\boldsymbol{N}_z = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \tag{6.10}$$

The matrix $\boldsymbol{N}_z$ is designed such that the reference r^k is only affecting the feedback of the z state. With the feedback law defined in (6.9) the closed loop state space description becomes as in (6.11)

$$\boldsymbol{\chi}_z^{k+1} = (\boldsymbol{A}_z - \boldsymbol{B}_z\boldsymbol{K}_z)\boldsymbol{\chi}_z^k + \boldsymbol{B}_z\boldsymbol{K}_z\boldsymbol{N}_z r^k + \boldsymbol{B}_z d^k \tag{6.11}$$

Two widely known ways of designing the feedback matrix $\boldsymbol{K}_z$ are the use of pole placement or Linear Quadratic Regulators (LQR). It is preferred to design it as a LQR since no direct requirements for the pole placement exist. However, a few requirements can be formulated for the size of the states and input which suits the LQR design method.
The quadratic cost function in (6.12) is considered in order to design the gain matrix $\boldsymbol{K}_z$. The matrix $\boldsymbol{K}_z$ is given by (6.13). Where the matrix $\boldsymbol{S}$ is the unique positive definite solution solution to the discrete time Riccati equation. The matrices $\boldsymbol{K}_z$ and $\boldsymbol{S}$ are found with the MATLAB function *dlqr*.

$$J = \sum_{k=0}^{\infty} \left(\boldsymbol{\chi}^{k,T}\boldsymbol{Q}\boldsymbol{\chi}^k + \boldsymbol{u}^{k,T}\boldsymbol{R}\boldsymbol{u}^k \right) \tag{6.12}$$

$$\begin{aligned} \boldsymbol{K}_z &= (\boldsymbol{R} + \boldsymbol{B}_z^T\boldsymbol{S}\boldsymbol{B}_z)^{-1}\boldsymbol{B}_z^T\boldsymbol{S}\boldsymbol{A}_z \\ \boldsymbol{S} &= \boldsymbol{Q} + \boldsymbol{A}_z^T\boldsymbol{S}\boldsymbol{A}_z - \boldsymbol{A}_z^T\boldsymbol{S}\boldsymbol{B}_z(\boldsymbol{R} + \boldsymbol{B}_z^T\boldsymbol{S}\boldsymbol{B}_z)^{-1}\boldsymbol{B}_z^T\boldsymbol{S}\boldsymbol{A}_z \end{aligned} \tag{6.13}$$

Bryson's rule is used as a starting point for designing the weighting matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$ in the cost function in (6.12). The Brysons rule is defined as.

$$Q_{ii} = \frac{1}{\text{maximum acceptable value of } \chi_i^2} \qquad i \in \{1, 2, \ldots, l\} \tag{6.14}$$

$$R_{jj} = \frac{1}{\text{maximum acceptable value of } u_i^2} \qquad j \in \{1, 2, \ldots, m\} \tag{6.15}$$

Where:

l	is the number of states
m	is the number of inputs

In Appendix A, it is found that the thrust value is scaled between 0 and 1. Furthermore, the hover thrust value is found to be 0.587. This means that the controller can output approximately ± 0.5 before hitting saturations. The maximum acceptable input is set to 0.25 since it is not desirable to apply either full thrust or zero thrust. This is not desirable because this would introduce non-linearities in the system that has not been modelled. In Section 2.3 it is found that the drone is supposed to fly in straight lines between way-points. This means that there is no upper limit for the largest expected step size the z controller can experience. Without any upper step size limit tracking errors can become arbitrarily large, resulting in large control signals and possible actuator saturations. With the z controller this would likely result in motors either turning off or running at full speed, thereby leaving no actuation room for the other controllers.

A saturation on the tracking error is then introduced in the feedback path in order to handle large position steps. It is decided to saturate the tracking error to 1 m, which is then used as the maximum acceptable value of the state z. There is no real requirement on the maximum acceptable velocity or acceleration, leading the $\boldsymbol{Q}$ and $\boldsymbol{R}$ matrices to be designed with the above consideration on z. These are defined in (6.16)

$$\boldsymbol{Q} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad R = \frac{1}{0.25^2} \tag{6.16}$$

With the weighting matrices in (6.16) the feedback matrix F becomes as in (6.17)

$$\boldsymbol{K}_z = \begin{bmatrix} 0.2335 & 0.1838 & 0.1367 \end{bmatrix} \tag{6.17}$$

In figure Figure 6.4, a simulated step response of the z controller can be seen, where both the response from the reference and from the disturbance are presented. From this figure it can be seen that the controller has unity gain on steps in the reference. But is can also be seen that the controller is not able to reject stationary disturbances on the thrust input. Furthermore it can be seen that the control signal is within the range of ± 0.25 at all time on a step of 1 m which is the largest step the controller can experience due to the saturation on the tracking error.

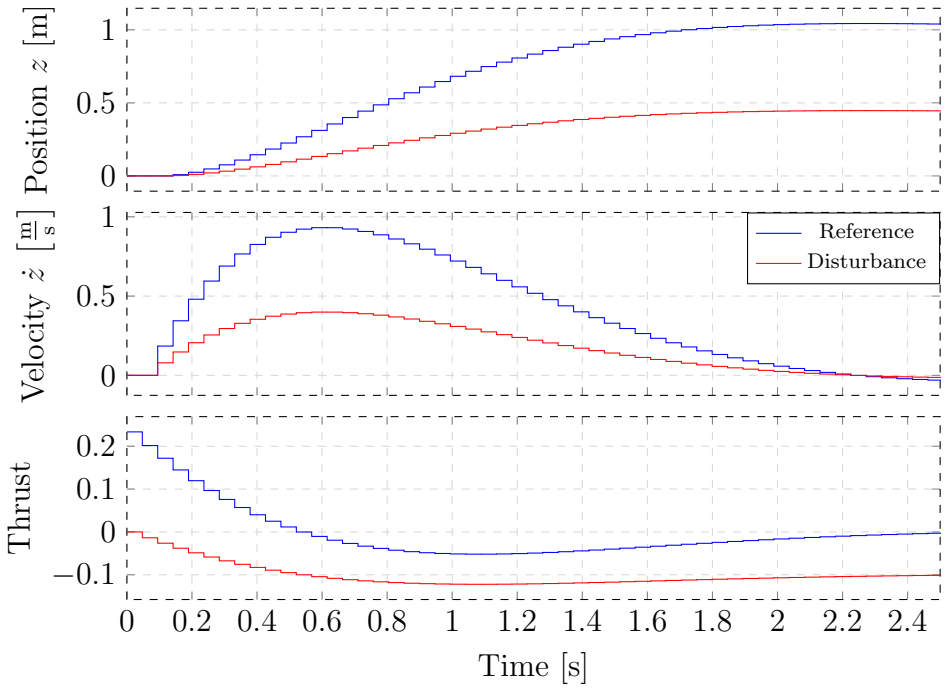


Figure 6.4: Simulation of step response of a reference change of 1 m and a change in disturbance of 0.1 with the designed controller.

The design of the z controller is done with the assumption that the largest step is 1 m but as discussed above there is no upper limit on the applied steps. When a large reference step is applied, the feedback loop is changed as illustrated in Figure 6.5 to model the saturation. The $\boldsymbol{H}$ matrix effectively removes the position feedback and sets tracking error to the constant saturation value. With the changed feedback the closed loop equation becomes the one shown in (6.18).

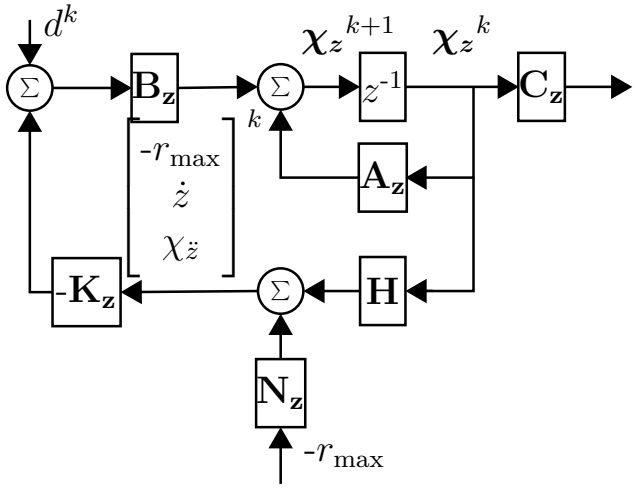


Figure 6.5: Closed loop system when large steps is applied.

$$\boldsymbol{\chi}_z^{k+1} = (\boldsymbol{A}_z - \boldsymbol{B}_z\boldsymbol{K}_z\boldsymbol{H})\boldsymbol{\chi}_z^k + \boldsymbol{B}_z\boldsymbol{K}_z\boldsymbol{N}_z r_{\max} + \boldsymbol{B}_z d^k \tag{6.18}$$

$$\boldsymbol{H} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Where:

$r_{\max}$	is the maximum allowed tracking error	[m]

In Figure 6.6 the effect of the saturation implemented on the tracking error of the z controller is shown when a large step in the reference is applied. In the saturated situation the drone is accelerating to a certain velocity determined by the gain matrix $\boldsymbol{K}_z$ and the designed maximum tracking error $r_{\max}$. From the figure it can be seen that the controller is not generating large control signals even when large steps in references are applied due to the saturation of the tracking error. The effect of the designed saturation is that the drone will fly with a constant saturated velocity towards the set-point, as seen in Figure 6.6. When the error becomes less than the maximum tracking error, $r_{\max}$, in this case set to 1 m, the velocity will slowly decrease and the position will settle at the reference, similar to the behaviour shown in Figure 6.4 with steps smaller than $r_{\max}$.

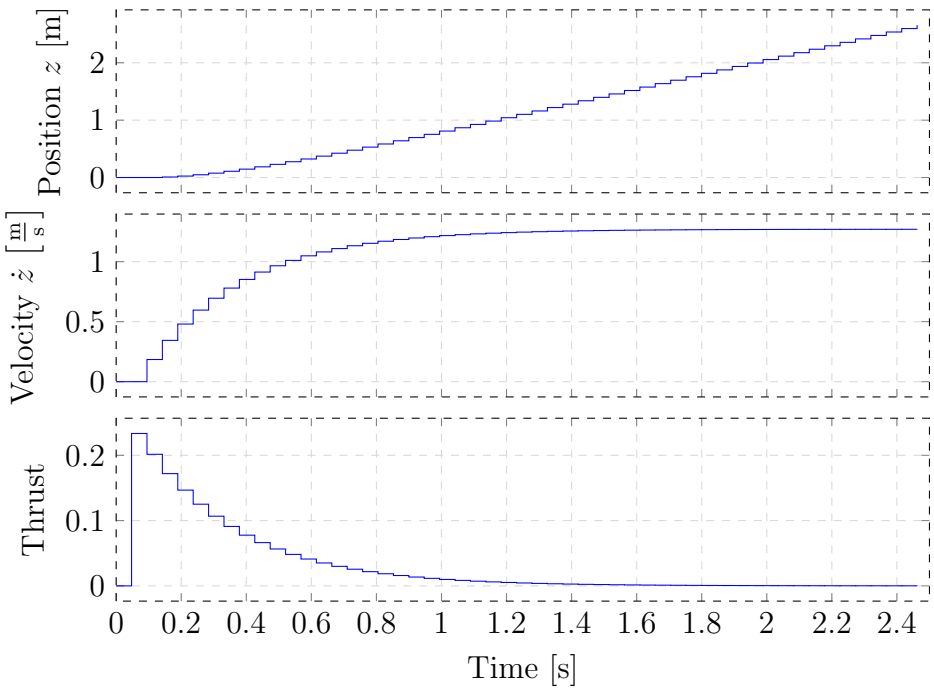


Figure 6.6: Simulation of a step response of a large step in reference, $r_{\max} = 1$ m.

6.2.2 Disturbance rejection

In the previous section the feedback gain $\boldsymbol{K}_z$ is designed. In this section it is investigated how the disturbance from the time varying thrust value can be rejected. It is decided to model this disturbance with an exogenous system like illustrated on Figure 6.7.

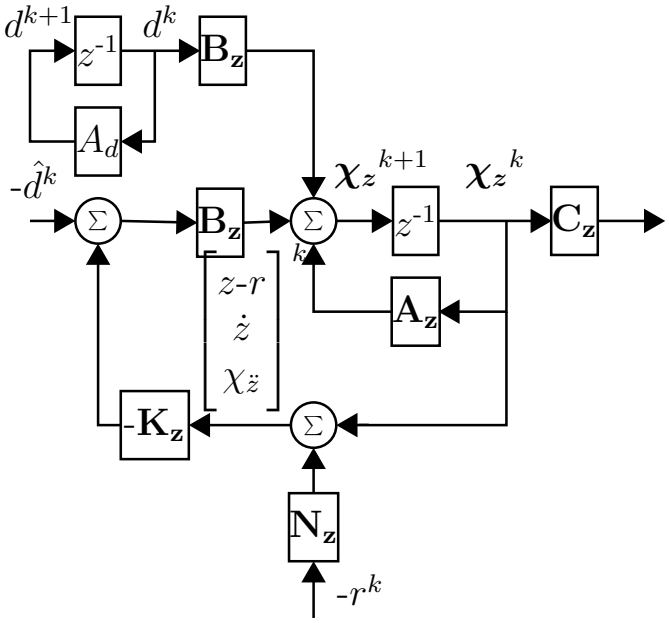


Figure 6.7: Closed loop system with exosystem and disturbance rejection.

These disturbances can be rejected by changing the feedback law defined in (6.9) to (6.19)

$$T^k = -\boldsymbol{K}_z\boldsymbol{\chi}_z^k + \boldsymbol{K}_z\boldsymbol{N}_z r^k - d^k \tag{6.19}$$

The closed loop equations with the feedback law defined in (6.19) become

$$\boldsymbol{\chi}_z^{k+1} = (\boldsymbol{A}_z - \boldsymbol{B}_z\boldsymbol{K}_z)\boldsymbol{\chi}_z^k + \boldsymbol{B}_z d^k - \boldsymbol{B}_z d^k + \boldsymbol{B}_z\boldsymbol{K}_z\boldsymbol{N}_z r^k \tag{6.20}$$

$$\boldsymbol{\chi}_z^{k+1} = (\boldsymbol{A}_z - \boldsymbol{B}_z\boldsymbol{K}_z)\boldsymbol{\chi}_z^k + \boldsymbol{B}_z\boldsymbol{K}_z\boldsymbol{N}_z r^k \tag{6.21}$$

But since the disturbance d^k is unknown and cannot be measured it has to be estimated by the extended Kalman filter. The closed loop equations then become

$$\boldsymbol{\chi}_z^{k+1} = (\boldsymbol{A}_z - \boldsymbol{B}_z\boldsymbol{K}_z)\boldsymbol{\chi}_z^k + \boldsymbol{B}_z d^k - \boldsymbol{B}_z\hat{d}^k + \boldsymbol{B}_z\boldsymbol{K}_z\boldsymbol{N}_z r^k \tag{6.22}$$

$$\boldsymbol{\chi}_z^{k+1} = (\boldsymbol{A}_z - \boldsymbol{B}_z\boldsymbol{K}_z)\boldsymbol{\chi}_z^k + \boldsymbol{B}_z\boldsymbol{K}_z\boldsymbol{N}_z r^k + \boldsymbol{B}_z(d^k - \hat{d}^k) \tag{6.23}$$

If (6.23) is compared with (6.11), it is apparent that the closed loop system with the feedback law defined in (6.19) now is affected by the estimation error $d^k - \hat{d}^k$ instead of the disturbance d^k.

6.2.3 Test

The feedback gain is designed in MATLAB using the linear model described in Section 6.1. Before test the controller has to be implemented as a ROS node in the ROS environment, where the communication with the PX4 controller is performed through the MAVROS node as described in Chapter 4. Before testing the controller in a real flight with the actual drone, the control algorithm is developed and tested in the Gazebo simulation environment described in Appendix J. This simulation environment allows to test the implemented control algorithm on a simulated drone before performing real flights. Once the controller implementation is found to be working as expected, it is tested in the real drone. The result of this test is compared with a simulation of the linear model used for design and the simulated drone within GAZEBO. A flight test is designed for the drone to evaluate the controller, consisting of different set-point references for the drone to track. The flight is performed in a Vicon-equipped environment and the data is extracted from the Vicon system measurements.

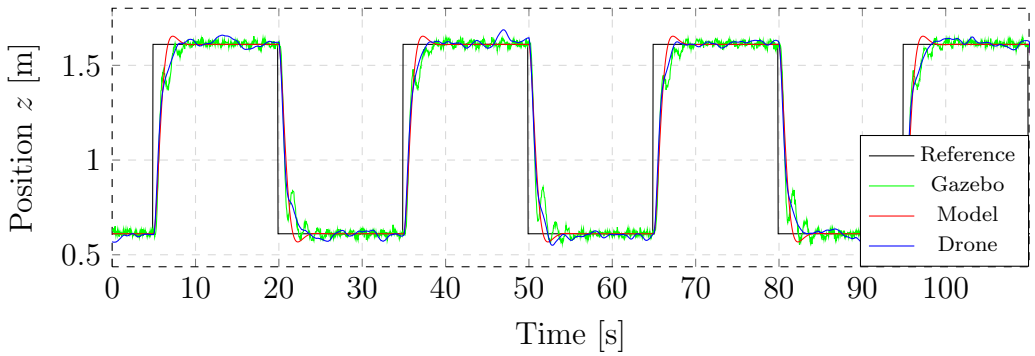


Figure 6.8: Comparison between flight data and simulations. Data in black is the defined setpoint for z, data in red is the linear model simulation, data in blue is the flight extracted data and data in green is the Gazebo simulation data.

The set-points are designed to be 1 m steps in order to check the performance of the controller in the largest input step allowed. It can be seen from Figure 6.8 that the drone tracks the set reference provided. In Figure 6.8 the comparison between the different performances is shown. It can be seen that, although the simulated drone in Gazebo is not the same one used during the real world flight, as described in Appendix J, the controller performance is similar in both cases, reaching the set-points as expected from the linear simulation.

6.3 X-Y Controller

In this section the control solution for the x and y position is presented. Similar to what is described for z controller in Section 6.2, the objective of this x and y controller is to reach coordinate references from an already defined path as presented in Section 2.3. It is known from Section 6.1 that when a frame aligned with the heading of the drone is considered, the system representation for x and y is decoupled. The open-loop system considered for the design of the controller is shown in (6.24), which corresponds to the decouplings related to x and y of the overall system presented in (6.6).

$$\begin{bmatrix} {}^{\mathrm{H}}x \\ {}^{\mathrm{H}}\dot{x} \\ \boldsymbol{\chi}_\theta \\ \hline {}^{\mathrm{H}}y \\ {}^{\mathrm{H}}\dot{y} \\ \boldsymbol{\chi}_\phi \end{bmatrix}^{k+1} = \left[\begin{array}{ccc|ccc} 1 & T_s & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} \\ 0 & 1 & gT_s\boldsymbol{C}_\theta & 0 & 0 & \mathbf{0}_{1\times4} \\ 0 & 0 & \boldsymbol{A}_\theta & 0 & 0 & \mathbf{0}_{1\times4} \\ \hline 0 & 0 & \mathbf{0}_{1\times4} & 1 & T_s & \mathbf{0}_{1\times4} \\ 0 & 0 & \mathbf{0}_{1\times4} & 0 & 1 & -gT_s\boldsymbol{C}_\phi \\ 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \boldsymbol{A}_\phi \end{array}\right] \begin{bmatrix} {}^{\mathrm{H}}x \\ {}^{\mathrm{H}}\dot{x} \\ \boldsymbol{\chi}_\theta \\ \hline {}^{\mathrm{H}}y \\ {}^{\mathrm{H}}\dot{y} \\ \boldsymbol{\chi}_\phi \end{bmatrix}^{k} + \left[\begin{array}{cc} 0 & 0 \\ 0 & 0 \\ 0 & \boldsymbol{B}_\theta \\ \hline 0 & 0 \\ 0 & 0 \\ \boldsymbol{B}_\phi & 0 \end{array}\right] \begin{bmatrix} \phi_{\mathrm{ref}} \\ \theta_{\mathrm{ref}} \end{bmatrix}^{k} \tag{6.24}$$

$$\begin{bmatrix} {}^{\mathrm{H}}x \\ {}^{\mathrm{H}}y \\ \phi \\ \theta \end{bmatrix}^{k} = \left[\begin{array}{ccc|ccc} 1 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} \\ 0 & 0 & \mathbf{0}_{1\times4} & 1 & 0 & \mathbf{0}_{1\times4} \\ 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \boldsymbol{C}_\phi \\ 0 & 0 & \boldsymbol{C}_\theta & 0 & 0 & \mathbf{0}_{1\times4} \end{array}\right] \begin{bmatrix} {}^{\mathrm{H}}x \\ {}^{\mathrm{H}}\dot{x} \\ \boldsymbol{\chi}_\theta \\ \hline {}^{\mathrm{H}}y \\ {}^{\mathrm{H}}\dot{y} \\ \boldsymbol{\chi}_\phi \end{bmatrix}^{k}$$

6.3.1 Feedback design

In order to track x and y set-points, a reference needs to be introduced in the system. From (6.3), (A.26) and (A.25) it can be seen that the transfer functions for these coordinates describes a type 2 system, meaning that no integral action is needed in the control structure in order to track step references with no steady-state error.
However, the set-points for x and y provided as references to the system are defined in a path described in EF, while the model for x and y are defined in HE. This means that in order to design a linear controller that forces the states of the system to track the set-points, the two frames need to match. This is obtained starting by rotating the reference from the EF to the HF. This system description is shown in Figure 6.9.

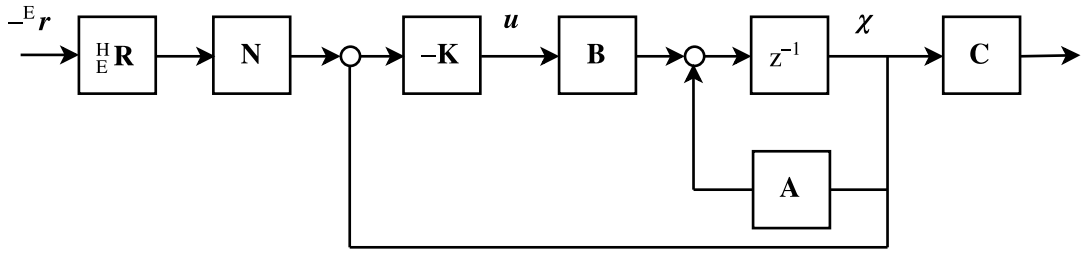


Figure 6.9: State-space description with applied rotation to the reference

Since the applied rotation is not related to the states, the reference to the system can be given in the HF, ${}^{\mathrm{H}}\boldsymbol{r}$, meaning that for the controller structure the rotation does not need to be taken into account. This rotation in the reference is shown in (6.25). The system to consider is then shown in Figure 6.10.

$$ {}^{\mathrm{H}}\boldsymbol{r} = {}^{\mathrm{H}}_{\mathrm{E}}\mathbf{R}\,{}^{\mathrm{E}}\boldsymbol{r} = \begin{bmatrix} \cos\psi & \sin\psi \\ -\sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} {}^{\mathrm{E}}x_{\mathrm{ref}} \\ {}^{\mathrm{E}}y_{\mathrm{ref}} \end{bmatrix} \tag{6.25} $$

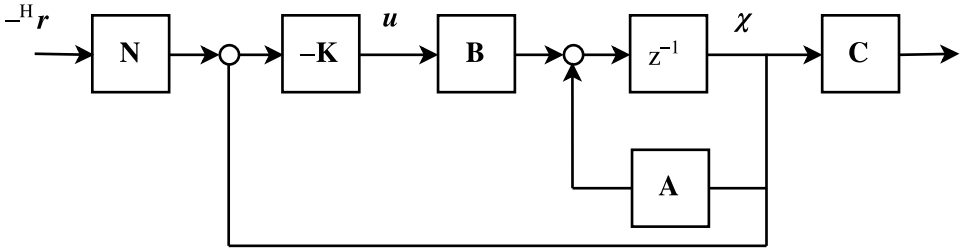


Figure 6.10: State-space description with rotated reference

The chosen control structure for the system consists in designing a feedback gain matrix for the system described in (6.24) which receives references transformed into the HF.
The feedback is applied to an estimation of the states, implying that they need to be estimated in the HF too. In Chapter 8 an extended Kalman filter is designed for the estimation of the full state vector, which is forced then to provide the estimation in the HF.
It is decided to take the LQR approach for the feedback design in the same way that the z controller is calculated in Section 6.2. First, the $\boldsymbol{N}$ matrix for the reference is designed to only affect the x and y states. The $\boldsymbol{N}$ matrix is defined in (6.27) with the reference vector defined in (6.26).

$$ {}^{\mathrm{H}}\boldsymbol{r} = \boldsymbol{r} = \begin{bmatrix} x_r \\ y_r \end{bmatrix} \tag{6.26} $$

$$ \boldsymbol{N} = \left[\begin{array}{ccc|ccc} 1 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} \\ 0 & 0 & \mathbf{0}_{1\times4} & 1 & 0 & \mathbf{0}_{1\times4} \end{array}\right] \tag{6.27} $$

Assuming that the EKF provides the controller with a correct estimated value of the states, the feedback gain is calculated with the use of the Bryson's rule for the weight matrices description. The same problem described in the z controller derivation is encountered here: there are no real limitations on the step sizes of the references. It is therefore decided to use the same saturation strategy on the tracking error as the one described for the z controller.

The controller is designed considering a maximum step size of 1 m. This is achieved by taking the specifications into account in the weight matrix for the states, $\boldsymbol{Q}$. This matrix is designed as a diagonal matrix with values only in the x and y states. Since the inputs to the system are angular references for roll and pitch, the maximum values allowed for these are decided to take into account the small angle approximation used for deriving the simplified model (Section A.2). The maximum value allowed for the roll and pitch references is decided to be 0.2 rad.

$$ \boldsymbol{Q} = \mathrm{diag}\left(1, 0, \mathbf{0}_{1\times4}, 1, 0, \mathbf{0}_{1\times4}\right) \qquad \boldsymbol{R} = \begin{bmatrix} \frac{1}{0.2^2} & 0 \\ 0 & \frac{1}{0.2^2} \end{bmatrix} \tag{6.28} $$

The controller gain can be separated into the gains affecting the states related to x and the ones related to y, and with the weight matrices described in (6.28), the feedback gain found is shown in (6.29).

$$\begin{aligned} \boldsymbol{K} &= \left[\begin{array}{c|c} \boldsymbol{K}_x & \boldsymbol{K}_y \end{array} \right] \\ \boldsymbol{K}_x &= \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0.1910 & 0.2438 & 0.0912 & -0.0358 & -0.0315 & -0.0092 \end{bmatrix} \\ \boldsymbol{K}_y &= \begin{bmatrix} -0.1908 & -0.2412 & 0.0929 & -0.0246 & 0.0004 & 0.0058 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{aligned} \tag{6.29}$$

In order to include a consideration for when the steps in the set-points are bigger than 1 m in the design, the same approach used for the z controller in Section 6.2.1 is used. The position feedback is removed so the error tracked is saturated. By doing this, the controller will behave as designed for the maximum step size instead of increasing the velocity of the drone.

The response of the system to a 1 m reference is shown in Figure 6.11, where the behaviour of x, y, $\dot{x}$, pitch, $\dot{y}$ and roll is presented.

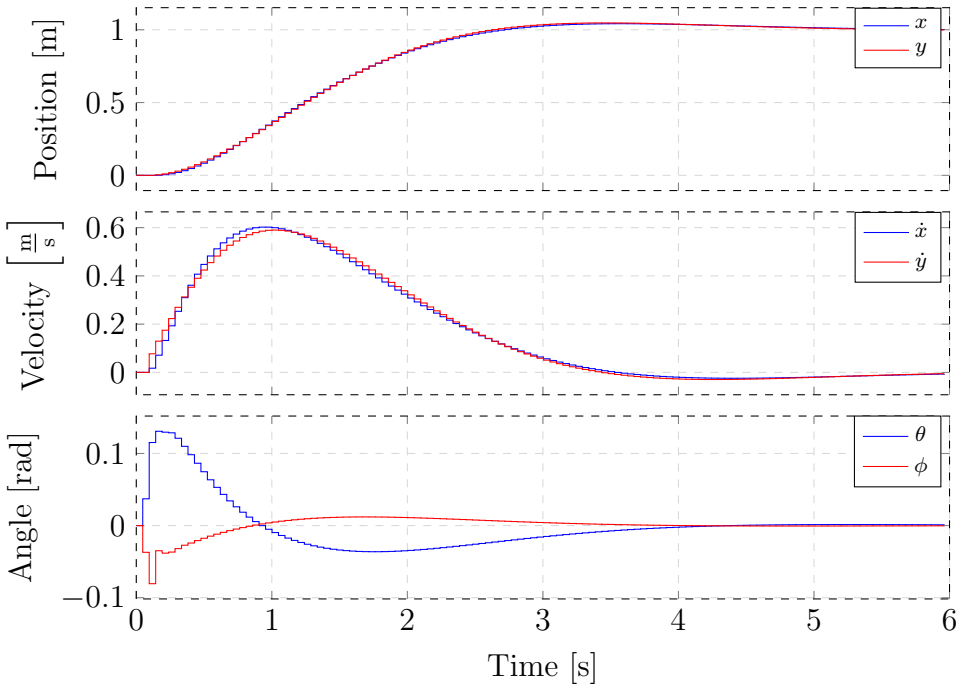


Figure 6.11: Step response of x, $\dot{x}$, pitch, y, $\dot{y}$ and roll to a reference of 1 m to x and y

The response to the system when the saturation on the tracking error is applied is shown in Figure 6.12, where the velocities $\dot{x}$ and $\dot{y}$ saturate since the reference is set to the maximum value. This figure shows a constant velocity behaviour of the system. Once the tracking error is below the maximum value, the system is brought back to the previous situation shown in Figure 6.11.

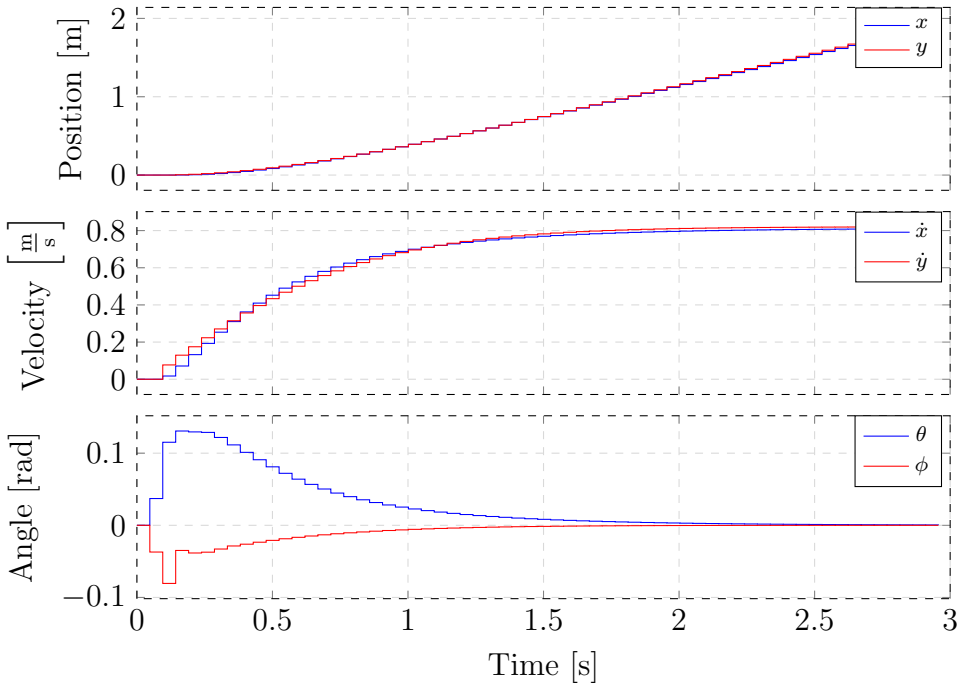


Figure 6.12: Step response of x, $\dot{x}$, pitch, y, $\dot{y}$ and roll to a step on x and y references when the saturation on the error is applied.

6.3.2 Test

Before testing the designed controller in real world the controllers performance is tested in simulation, in the same way as described for the z controller. This is done by using the linear model with which it has been designed and the Gazebo simulation. Once the implementation of it in ROS is found to be correct, a real test for the drone needs to be designed.
During the whole test, the already designed z controller from Section 6.2 is used to keep the drone in a constant altitude.

The experiment is performed with yaw fixed to 0 rad. Once the drone is in a held height, a set of references is given to the it, describing a square in the x-y plane. This includes a set-point in the centre of the square in order to use the controller in both directions at the same time.
Figure 6.13 shows the performance of the drone when these set-points are given. It can be seen that the actual performance differs from the Gazebo one. The simulated controller reaches the set-points whereas the one implemented in the real drone does not actually reach them.

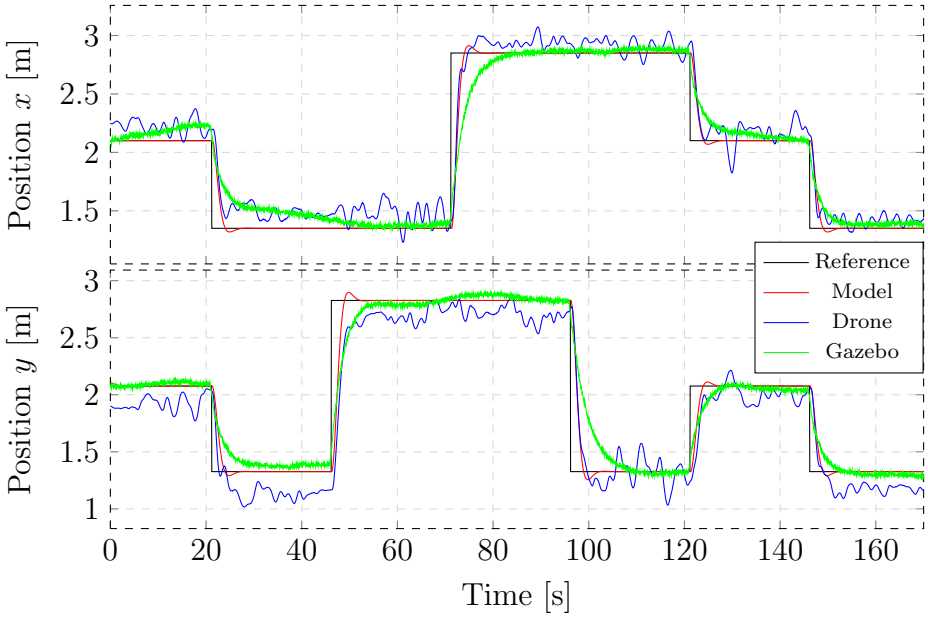


Figure 6.13: Comparison between flight data and simulation. Data in black is the defined setpoint for x and y, data in red is the linear model simulation, data in blue is the flight extracted data and data in green is the Gazebo simulation data.

In Figure 6.14 a 2D plot of the flight is shown. The blue straight lines shown are the references given to the drone and the scattered dots are the measurements taken from the flight and the Gazebo simulation. It can be seen from it that the drone approaches the vertexes of the defined square, although the path taken is not the one defining the square. This is due to different performance of the designed controllers for x and y. The performance of the controllers is deemed to be satisfactory since they are only designed to reach set-points and not to track the trajectory defined by the set-points. Further action has to be taken if it is important to stay closer to the trajectory defined by the set-points.

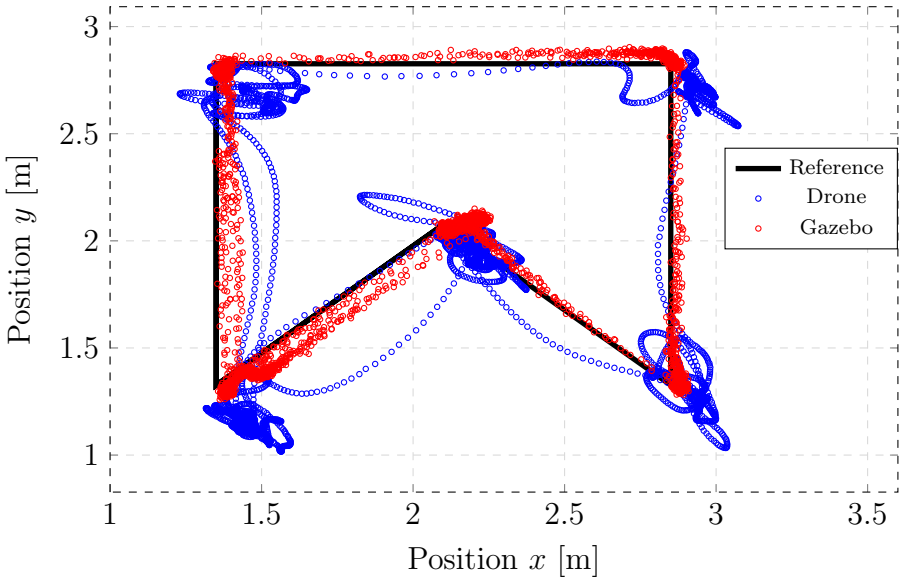


Figure 6.14: 2D plot of the x and y measurements taken from the flight compared to the Gazeo simulation.

7 FastSLAM 2.0 estimator

This Chapter describes how the FastSLAM 2.0 algorithm, in the sequel referred to as FastSLAM, is used to estimate a part of the drone's state vector and the map $\boldsymbol{M}$. In Section 5.2 it is decided that FastSLAM should be used to estimate the pose vector defined in (7.1) and the map defined in (7.2).

$$\boldsymbol{s} = [x, y, z, \psi]^T \tag{7.1}$$

$$\boldsymbol{M} = [\boldsymbol{l}_1, \boldsymbol{l}_2, \ldots, \boldsymbol{l}_N]^T \tag{7.2}$$

Where:

x	is the drone's x coordinate in the Earth frame
y	is the drone's y coordinate in the Earth frame
z	is the drone's z coordinate in the Earth frame
ψ	is the drone's yaw angle in the Earth frame
N	is the number of landmarks at time k
$\boldsymbol{l}_i$	is a landmark

The N landmarks is in this project chosen to be point landmarks described in (7.3)

$$\boldsymbol{l}_i = [x_l, y_l, z_l]^T \tag{7.3}$$

A summary of the FastSLAM algorithm for a general SLAM problem is given in Section 7.1. The motion and measurement models needed for the implementation of the algorithm are presented in Section 7.2 and Section 7.3. An overview of the implementation of the algorithm used in this project is presented in Section 7.5. Finally test results of the implemented algorithm are shown in Section 7.6.

7.1 Algorithmic summary

This section is intended to provide a summary of the FastSLAM algorithm. The full details of the algorithm can be found in [13]. FastSLAM was proposed as a solution to the SLAM problem in [14] in 2003 as an improvement of the original FastSLAM algorithm [22]. The purpose of the algorithm is to estimate the a posteriori probability of the map, $\boldsymbol{M}$, and robot path, $\boldsymbol{s}^{1:t}$. A SLAM problem of the form in (7.4) is assumed.

$$p\left(\boldsymbol{s}^{1:k}, \boldsymbol{M} | \boldsymbol{z}^{1:k}, \boldsymbol{u}^{1:k}\right) \tag{7.4}$$

Where:

$\boldsymbol{s}^{1:k}$	is the robot path from time 1 to k
$\boldsymbol{M}$	is a map of the environment
$\boldsymbol{z}^{1:k}$	is all relative measurements between the robot and landmarks in the map from time 1 to k
$\boldsymbol{u}^{1:k}$	is all inputs to the robot from time 1 to k

The algorithm furthermore assumes that only relative measurements between the drone and each observed landmark are available [13]. Using the definition of conditional probability, see (E.1), the PDF in (7.4) can be rewritten to (7.6).

$$p\left(\boldsymbol{s}^{1:k}, \boldsymbol{M} | \boldsymbol{z}^{1:k}, \boldsymbol{u}^{1:k}\right) = p\left(\boldsymbol{s}^{1:k} | \boldsymbol{z}^{1:k}, \boldsymbol{u}^{1:k}\right) p\left(\boldsymbol{M} | \boldsymbol{s}^{1:k}, \boldsymbol{z}^{1:k}, \boldsymbol{u}^{1:k}\right) \tag{7.5}$$

$$= p\left(\boldsymbol{s}^{1:k} | \boldsymbol{z}^{1:k}, \boldsymbol{u}^{1:k}\right) p\left(\boldsymbol{M} | \boldsymbol{s}^{1:k}, \boldsymbol{z}^{1:k}\right) \tag{7.6}$$

Where the $\boldsymbol{u}^{1:k}$, can be dropped in the PDF for the map since the map conditioned on the path does not depend on the input to the robot. FastSLAM furthermore assumes that landmarks are conditional independent given the path of the robot. Thereby the posterior (7.6) can be factorized as

$$p\left(\boldsymbol{s}^{1:k}, \boldsymbol{M} | \boldsymbol{z}^{1:k}, \boldsymbol{u}^{1:k}\right) = p\left(\boldsymbol{s}^{1:k} | \boldsymbol{z}^{1:k}, \boldsymbol{u}^{1:k}\right) \prod_{i=1}^{N} p\left(\boldsymbol{l}_i | \boldsymbol{s}^{1:k}, \boldsymbol{z}^{1:k}\right) \tag{7.7}$$

as shown in [13]. In words (7.7) simply states that if the true path $\boldsymbol{s}^{1:k}$ is known then information about one landmark will not yield any information about any other landmarks. This is of course only valid due to the assumption about conditional independence between landmarks, and the assumptions about only using relative measurements between the robot and each observed landmark.
The factorisation in (7.7) can be exploited by maintaining low dimensional filters for each factor in (7.7). FastSLAM uses a particle filter, which utilizes an EKF for each particle to obtain an estimate of its proposal distribution, to estimate the target distribution $p\left(\boldsymbol{s}^{1:k} | \boldsymbol{z}^{1:k}, \boldsymbol{u}^{1:k}\right)$. Furthermore FastSLAM uses N extended Kalman filters to estimate $p\left(\boldsymbol{l}_i | \boldsymbol{s}^{1:k}, \boldsymbol{z}^{1:k}\right)$ for each particle in the particle filter since the landmarks, $\boldsymbol{l}_i$, are conditioned on the robot's path. Thus if the particle filter has P particles, then there is a total of $P \cdot N$ EKF's.
In the following, subscript $_{[p]}$ will be used to indicate the index of the p'th particle. Thereby each particle, $\boldsymbol{s}_{[p]}^{1:k}$, in the particle filter is associated with a set of Kalman filters. Each Kalman filter is maintaining an estimated mean, $\bar{\boldsymbol{l}}_{i,[p]}^{k|k}$, and covariance, $\text{Cov}\left(\boldsymbol{l}_i^{k|k}\right)_{[p]}$ for a single landmark in the map based on the estimated path of the specific particle.

The FastSLAM algorithm assumes a motion model and a measurement model on the form

$$\boldsymbol{s}^k = f\left(\boldsymbol{s}^{k-1}, \boldsymbol{u}^{k-1}\right) + \boldsymbol{w}^{k-1} \tag{7.8}$$

$$\boldsymbol{z}_{l_i}^k = h\left(\boldsymbol{s}^k, \boldsymbol{l}_i\right) + \boldsymbol{v}^k \tag{7.9}$$

As a result of this motion model and measurement model and some approximations used to derive the equations of the algorithm, the equations used by the algorithm do not depend on the full path taken by each particle, $\boldsymbol{s}_{[p]}^{1:k}$, but only the current pose, $\boldsymbol{s}_{[p]}^{k}$, and the previous pose, $\boldsymbol{s}_{[p]}^{k-1}$. Therefore the algorithm do not need to store information about the full path. In the following, "a particle" is thus used to refer to a pose, $\boldsymbol{s}_{[p]}^{k}$, instead of the full path, $\boldsymbol{s}_{[p]}^{1:k}$.
The following presents the FastSLAM algorithm, assuming that the reader is familiar with the concepts of Extended Kalman Filter, described in Section E.5, sequential Extended Kalman Filter, described in Section E.6, and particle filters, described in Section E.8.

To explain the steps of the algorithm, additional symbols are introduced:

$\boldsymbol{z}_{i,ex}^{k}$	is the i'th measurement of existing landmarks
$\boldsymbol{z}_{\text{j,new}}^{k}$	is the j'th measurement of a new landmark
$\boldsymbol{s}_{[p]}^{k}$	is the pose estimate of the p'th particle after the i'th iteration of the sequential Kalman filter

The algorithm is presented with a summery of the steps, followed by an example going through these steps. This example is illustrated in Figure 7.1. The algorithm can be summarized as follows:

1. Initialize the filter with a set of particles, $\boldsymbol{s}_{[p]}^{0}$, distributed according to the initial guess of the distribution $p\left(\boldsymbol{s}^{0}\right)$.
2. For $k = 1, ...$ do

   Generate proposal particles approximately distributed according to the proposal distribution

   $$p\left(\boldsymbol{s}^{k}|\boldsymbol{u}^{k}, \boldsymbol{s}^{k-1}, \boldsymbol{z}^{k}\right) \tag{7.10}$$

   by

   (a) Predicting each particle forward in a similar way as a sequential extended Kalman filter based on (7.8), to obtain $\bar{\boldsymbol{s}}_{0,[p]}^{k}$ and $\text{Cov}\left(s_{0,[p]}^{k}\right)$, but with the covariance of the initial pose put to zero. That is $\text{Cov}\left(\boldsymbol{s}_{[p]}^{k-1}\right) = \boldsymbol{0}$.

   (b) For each particle process each measurement of landmarks already initialized in the filter at time k, $\boldsymbol{z}_{i,ex}^{k}$ for $i = 1, ..., I$, through the update step of a sequential Extended Kalman Filter, to obtain an estimate of the mean, $\bar{\boldsymbol{s}}_{I,[p]}^{k}$, and covariance, $\text{Cov}\left(\boldsymbol{s}_{I,[p]}^{k}\right)$, of the pose at time k, $\boldsymbol{s}^{k}$.

   (c) Randomly draw one proposal particle, $\tilde{s}_{[p]}^{k}$, from each of the estimated normal distributions

   $$\tilde{\boldsymbol{s}}_{[p]}^{k} \sim \mathcal{N}\left(\bar{\boldsymbol{s}}_{I,[p]}^{k}, \text{Cov}\left(\boldsymbol{s}_{I,[p]}^{k}\right)\right) \tag{7.11}$$

   The proposal particles $\tilde{\boldsymbol{s}}_{[p]}^{k}$, should now approximately be distributed according to (7.10). Thus proceed and do

   (d) Calculate an importance weight, $q_{[p]}$, for each particle.

   (e) For each particle update the estimate of the landmarks already initialized in the filter, based on $\boldsymbol{z}_{i,ex}^{k}$ for $i = 1, ..., I$, with I being the number of measurements of existing landmarks.

   (f) For each particle add new landmarks to the map by using each measurement of new landmarks at time k, $\boldsymbol{z}_{\text{j,new}}^{k}$ for $j = 1, ..., J$, with J being the number of measurements of new landmarks.

   (g) Perform importance resampling on the particles.

A more thorough description of step 2a/2b, 2d and 2e/2f, with the equations needed for implementation, can be found in Section E.9.1, Section E.9.2 and Section E.9.3 respectively. More details about importance weights and importance sampling in general can be found in Section E.7. The authors of the FastSLAM algorithm do not specify any specific algorithm to perform the resampling step, but state that it should be chosen as a compromise between accuracy and ease of implementation.

The steps of the algorithm are illustrated for a simple one dimensional case with 6 particles and only one known landmark at time k, in Figure 7.1 on the next page. In the example starting at time $k-1$, particle 1, 2 and 3 have the same value, and 4, 5 and 6 the same.

In step 2a the prediction step of the sequential EKF is performed.

In step 2b the correction step of the sequential EKF is performed based on the current estimate of $\boldsymbol{l}_1$ within the particle and a relative measurement of $\boldsymbol{l}_1$.

In step 2c particles are drawn according to normal distributions with the mean and covariances estimated by the sequential EKFs.

In step 2d importance weights are assigned to the particles. Notice how particle 1 and 2 are assigned very small weights.

In step 2e every Extended Kalman Filter estimating the position of $\boldsymbol{l}_1$ for each landmark is updated.

In step 2f a new landmark is added to the map of each particle, based on the measurement of the new landmark and the inverse measurement model.

In step 2g resampling is performed, causing $\tilde{\boldsymbol{s}}^k_{[1]}$ and $\tilde{\boldsymbol{s}}^k_{[2]}$ to be deleted and $\tilde{\boldsymbol{s}}^k_{[4]}$ and $\tilde{\boldsymbol{s}}^k_{[5]}$ to be copied.

As described above, the FastSLAM algorithm needs to keep track of N Kalman filters for each particle within the particle filter. Each of these Kalman filters are described by a mean and covariance, potentially leading to a lot of data that has to be stored for each particle.
Some particles will end up storing the same mean and covariances of the landmarks when no measurements of those landmarks are available at time k, since the particles are copied in the resampling step. This is due to the update step where the mean and covariance is only updated for the landmarks of which measurements are available at time k.
To exploit this property the inventors of the FastSLAM algorithms proposed to use a binary tree to store the means and covariances of the Kalman filters, by which data common to particles could be shared.

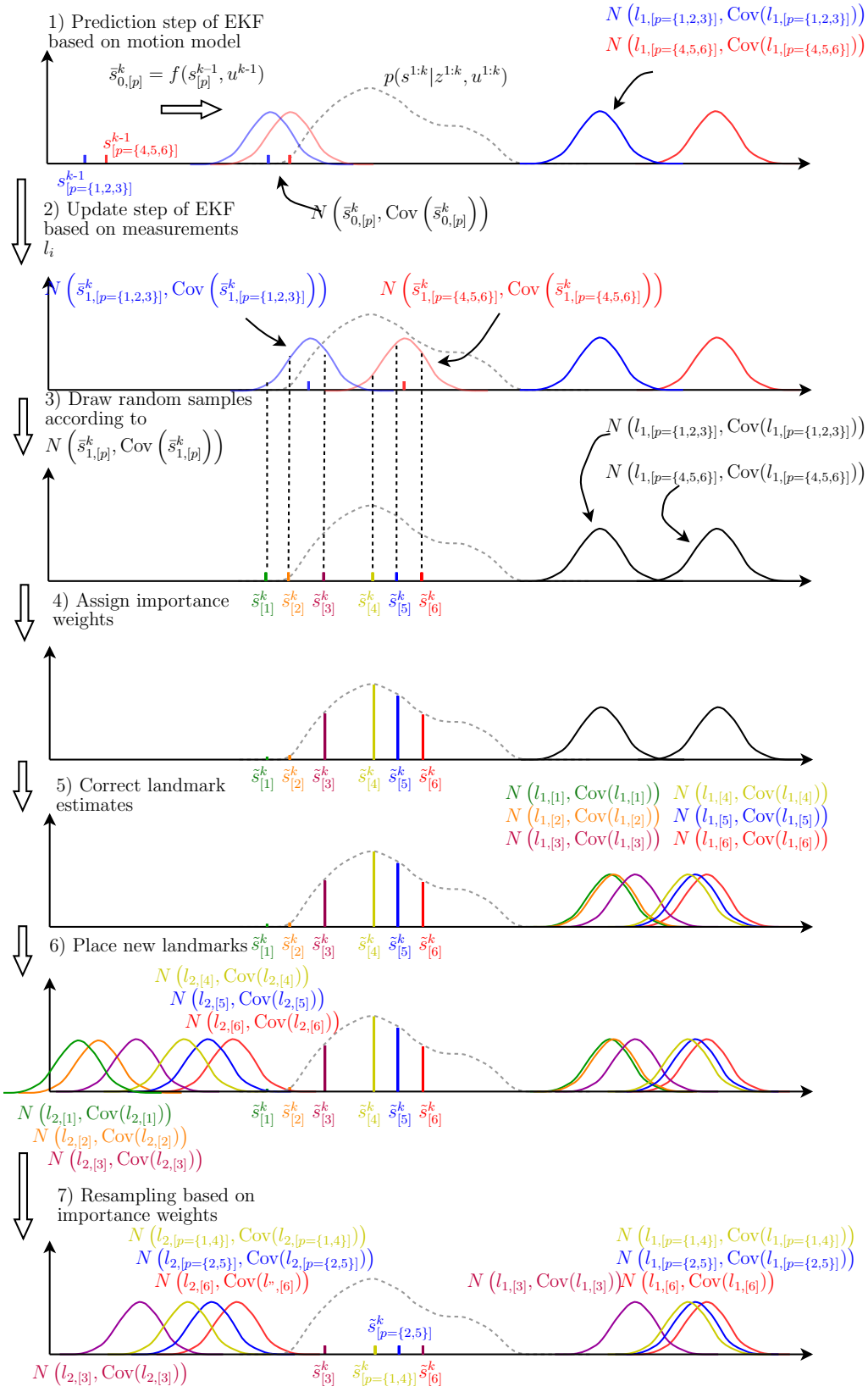


Figure 7.1: Illustration of the steps of the FastSLAM algorithm for a simple one dimensional case, with 6 particles.

7.2 Simplified Motion Model

The FastSLAM implementation needs a motion model to calculate a proposal distribution to be used as part of the prediction step within the particle filter. As described in Section 5.2.6 it is decided that FastSLAM should be using a simplified motion model instead of an ARX based motion model to keep the number of estimated states low. The simplified motion model can be designed as a kinematic model taking the translational velocities from the ARX model state vector, see Section 6.1, estimated by an external Extended Kalman Filter as described in Chapter 8. Let these velocity estimates of the drone relative to the heading frame be denoted ${}^{\mathrm{H}}\boldsymbol{v}$.

$$ {}^{\mathrm{H}}\boldsymbol{v} = \begin{bmatrix} {}^{\mathrm{H}}\hat{x} \\ {}^{\mathrm{H}}\hat{y} \\ {}^{\mathrm{H}}\hat{z} \end{bmatrix} \tag{7.12} $$

For the yaw angle however there is no velocity estimate why it is decided to design the motion model such that a yaw angle difference can be provided as input.

$$ \Delta\psi^k = \psi^k - \psi^{k-1} \tag{7.13} $$

Whenever the FastSLAM algorithm is executed the difference between the current yaw angle estimate from the EKF and the previous yaw angle estimate from last time the algorithm ran, is calculated. This yaw angle difference, $\Delta\psi^k$, is concatenated with the velocity estimates to form the motion model input vector, $\boldsymbol{u}$.

$$ \boldsymbol{u} = \begin{bmatrix} {}^{\mathrm{H}}\boldsymbol{v} \\ \Delta\psi \end{bmatrix} \tag{7.14} $$

The state vector of FastSLAM, see (7.15), is chosen to include only the heading angle of the drone and the full position vector relative to the GOT coordinate system defined in the Earth frame of which the FastSLAM coordinate system should align to.

$$ \boldsymbol{s} = \begin{bmatrix} x \\ y \\ z \\ \psi \end{bmatrix} \tag{7.15} $$

According to (7.8) the motion model should be on the following form:

$$ \boldsymbol{s}^k = f_s\left(\boldsymbol{s}^{k-1}, \boldsymbol{u}^{k-1}\right) + \boldsymbol{w}^{k-1} \tag{7.16} $$

where the Gaussian noise variable $\boldsymbol{w}$ as input described by

$$ \boldsymbol{w}^{k-1} \sim \mathcal{N}\left(\boldsymbol{0}_{4\times 1}, \mathrm{Cov}\left(\boldsymbol{w}^{k-1}\right)\right) \tag{7.17} $$

The motion model will thus describe the distribution:

$$ p(\boldsymbol{s}^k \mid \boldsymbol{u}^{1:k}, \boldsymbol{s}^{1:k-1}) \tag{7.18} $$

As the motion model is a discrete kinematic model one can expect modelling errors residing from discretization to result in noise on the state prediction. Furthermore any noise in the velocity estimates used as inputs, $\boldsymbol{u}$, will also affect the state prediction.

The motion model will take the most recent velocity estimate from the Extended Kalman Filter, which is running at a faster rate, and use this as input for the prediction. Using a more recent velocity estimate than the velocity at the previous time-step, $\boldsymbol{v}^{k-1}$, will also result in small prediction errors. It is assumed though that the prediction error caused by noise in the velocity estimates will be much greater than the modelling errors caused by discretization. Therefore noise on the inputs of the motion model, as shown in (7.19), will be considered instead of the additive noise as assumed in the FastSLAM 2 algorithm, see (7.8).

$$\boldsymbol{s}^k = f_s\left(\boldsymbol{s}^{k-1},\, \left(\boldsymbol{u}^{k-1} + \boldsymbol{w}^{k-1}\right)\right) \tag{7.19}$$

To make the motion model fit with FastSLAM 2, it is decided to make a Taylor series expansion to linearise the motion model around $\boldsymbol{w}^{k-1} = 0$, in a similar fashion as done for the Extended Kalman filter presented in Section E.5. The approximation becomes

$$\begin{aligned} f_s\left(\boldsymbol{s}^{k-1}, \left(\boldsymbol{u}^{k-1} + \boldsymbol{w}^{k-1}\right)\right) &\approx f_s\left(\boldsymbol{s}^{k-1}, \boldsymbol{u}^{k-1}\right) + \frac{f_s\left(\boldsymbol{s}^{k-1}, \boldsymbol{u}^{k-1}\right)}{\partial \boldsymbol{w}^{k-1}}\left[\boldsymbol{w}^{k-1} - \boldsymbol{0}_{4\times 1}\right] \\ &= f_{\boldsymbol{s}}\left(\boldsymbol{s}^{k-1}, \boldsymbol{u}^{k-1}\right) + \boldsymbol{F}_w \boldsymbol{w}^{k-1} \\ &= f_{\boldsymbol{s}}\left(\boldsymbol{s}^{k-1}, \boldsymbol{u}^{k-1}\right) + \tilde{\boldsymbol{w}}^{k-1} \end{aligned} \tag{7.20}$$

Based on Section E.3 and with the input noise modelled as a Gaussian according to (7.17) it follows that:

$$\tilde{\boldsymbol{w}}^{k-1} \sim \left(\boldsymbol{0}_{4\times 1},\, \boldsymbol{F}_w \mathrm{Cov}\left(\boldsymbol{w}^{k-1}\right) \boldsymbol{F}_w^T\right) \tag{7.21}$$

This transforms the input noise into an additive noise that can be used by the FastSLAM 2 algorithm. The covariance of this input noise, $\boldsymbol{w}^{k-1}$, is a combination of the covariance of the velocity estimates, taken from the external Extended Kalman Filter, and the noise of the yaw angle difference. As the yaw angle is not correlated with the velocity estimates because these are in the local heading frame, the variance of the yaw angle difference can just be calculated from the variance of the current and previous yaw angle estimate, being described by Gaussian random variables. The variance thereby become:

$$\mathrm{Var}\left(\Delta\psi^k\right) = \mathrm{Var}\left(\psi^k\right) + \mathrm{Var}\left(\psi^{k-1}\right) \tag{7.22}$$

This allows the actual motion model input covariance to be determined at each timestep and included within FastSLAM.

$$\mathrm{Cov}\left(\boldsymbol{w}^{k-1}\right) = \begin{bmatrix} \mathrm{Cov}\left(\boldsymbol{v}^{k-1}\right) & \boldsymbol{0}_{3\times 1} \\ \boldsymbol{0}_{1\times 3} & \mathrm{Var}\left(\Delta\psi^{k-1}\right) \end{bmatrix} \tag{7.23}$$

As the heading frame only tracks the yaw angle but does not follow the roll and pitch angles, the motion model will only have to consider the velocity inputs and the previous yaw angle. Assuming small yaw angular velocities and assuming that the FastSLAM implementation is run at a sufficiently fast rate such that the yaw angle seems almost constant between samples, then the motion model can be simplified even further by assuming that the drone is flying in straight lines between samples. The motion model thereby only needs to rotate the velocity input vector by the previous yaw angle according to (A.3), and integrate it using the time passed since the previous sample, Δt^k, resulting in a new pose prediction. The simplified motion model is shown in (7.24).

$$f_s\left(\boldsymbol{s}^{k-1}, \left(\boldsymbol{u}^{k-1} + \boldsymbol{w}^{k-1}\right), \Delta t^k\right) = \boldsymbol{s}^{k-1} + \begin{bmatrix} {}^{\mathrm{E}}_{\mathrm{H}}\mathbf{R}_{\psi^{k-1}} \Delta t^k & 0 \\ 0 & 1 \end{bmatrix} \left(\boldsymbol{u}^{k-1} + \boldsymbol{w}^{k-1}\right) \tag{7.24}$$

where

$$ {}^{\mathrm{E}}_{\mathrm{H}}\mathbf{R}_{\psi^{k-1}} = \begin{bmatrix} \cos\psi^{k-1} & -\sin\psi^{k-1} & 0 \\ \sin\psi^{k-1} & \cos\psi^{k-1} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{7.25} $$

This kind of simple kinematic motion model allows the sample rate to vary as the model is not discretized for any specific sample rate but instead includes the integration interval, Δt^k.

7.2.1 Motion model Jacobian

To be able to calculate how noise from the velocity estimates affects the predicted states, a Jacobian of the predicted states with respect to the noise input is derived in (7.26). This Jacobian will be included as part of the Extended Kalman filters correcting the proposal distribution within FastSLAM.

$$ \boldsymbol{F}_w(\boldsymbol{s}^{k-1}, \Delta t^k) = \frac{\partial f_s}{\partial \boldsymbol{w}^{k-1}} = \begin{bmatrix} \Delta t^k \cos\psi^{k-1} & -\Delta t^k \sin\psi^{k-1} & 0 & 0 \\ \Delta t^k \sin\psi^{k-1} & \Delta t^k \cos\psi^{k-1} & 0 & 0 \\ 0 & 0 & \Delta t^k & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.26} $$

7.3 RGB-D Measurement Model

The RealSense R200 RGB-D camera on the Intel Aero drone is capable of providing an RGB image and a depth image, see Appendix F. As described in Section 3.3.1 ArUco markers are put up in the environment and captured by the RGB camera. Using the ArUco feature detector algorithm, described in Appendix H, a list of pixel coordinates of detected markers, $p_{x,i}$ and $p_{y,i}$, and their unique identifiers i, are determined. Using the pixel location of the detected markers, a corresponding depth value to each marker, d, is extracted from the depth image. FastSLAM needs a measurement model of the RGB-D camera and the ArUco marker detection to calculate predicted measurement vectors of current landmarks within the map, and an inverse measurement model to be able to insert landmarks into the map based on measurements.

Let the RGB-D camera measurement vector at time k corresponding to marker i be denoted $\boldsymbol{z}^k_{c,i}$ defined according to (3.1). Within this section a measurement model, shown in (7.27), describing the projection from physical landmark positions to projected pixel coordinates and depth values is developed based on the pin-hole camera model. The model will consider additive noise on the measurement vector as expected by the FastSLAM in (7.9).

$$ \boldsymbol{z}^k_{c,i} = h_c(\boldsymbol{s}^k, \phi^k, \theta^k, \boldsymbol{l}_i) + \boldsymbol{v}^k \tag{7.27} $$

Where:

$\boldsymbol{z}^k_{c,i}$	is the camera measurement vector at time k
$\boldsymbol{s}^k$	is the pose of the drone at time k
ϕ^k	is the roll angle at time k
θ^k	is the pitch angle at time k
$\boldsymbol{l}_i$	is the landmarks position vector
$\boldsymbol{v}^k$	is the measurement noise at time k

A loosened notation will be used throughout the section by letting all model equations be defined for a specific marker, i and at time k. The measurement vector is thus defined as:

$$\boldsymbol{z}_c = \begin{bmatrix} x_c \\ y_c \\ d \end{bmatrix} \tag{7.28}$$

7.3.1 Camera intrinsics

The pin-hole camera model described in Appendix G models the projection behaviour of a lens, describing how a landmark relative to the camera viewpoint is projected onto the image plane. Unfortunately, the actual projection starts to deviate from the pin-hole model when projected objects are close to the edges of the image. This behaviour is known as lens distortion and it is the case with the RGB camera of the R200 camera, as described in Appendix F. The distortion has been modelled and calibrated by Intel using a Plumb Bob model [23] which allows a one-way transformation from undistorted pixel coordinate to distorted coordinate. As only a one-way transformation is possible, the distorted image captured by the camera can only be undistorted through an iterative process. It is therefore recommended to do any "pin-hole model"-based projection and deprojection using only the depth image or aligned images to avoid an iterative distortion-removal algorithm [24].

As described in Appendix F, the depth image is on the other hand based on a rectified disparity map calculated from the stereo images coming from the IR-camera pair on the R200 camera from where distortion has been removed. The depth image contains projected depth values to objects in the environment where the projection happens on to the plane of the disparity map given the intrinsics of the stereo cameras. The pin-hole model can then be used to convert any measured depth value within the depth image back into a world coordinate relative to the location and orientation of the camera.

The four parameters defining the intrinsics used within (G.6) in Appendix G can be extracted for both the RGB and depth camera on the R200 camera and would thus not need to be determined. Both the RGB image and depth image are configured to a resolution of 320×240 pixels and with the help of the rectification process described in Appendix F in Section F.5, the two images are aligned. The resulting aligned images share the same intrinsics, are non-distorted and are physically located on top of each other thereby having no extrinsic between them. This allows the pixel coordinate of a detected marker within the RGB image to be used in the depth image to extract the measured depth to the marker as described in Section 4.3.

7.3.2 Extrinsic transformation

The pin-hole camera model projects the environment relative to the position and orientation of the camera, why such variables need to be included in the measurement model. To project landmarks within the world into the image plane, the landmarks have to be transformed from the world frame (earth frame) into the camera frame. To make the derivation easier to follow, an inverse transformation from camera frame into world frame is derived in this subsection.
The camera frame has its z-axis pointing in the viewpoint direction, its x-axis pointing right and its y-axis pointing down as shown in Figure G.2. The body frame of the drone, to which the camera is rigidly attached, has its x-axis pointing forward as described in Section 6.1. Assuming that the z-axis of the camera is perfectly aligned with the x-axis of the drone and that the camera is aligned with the horizontal axis of the drone, the rotation between the camera frame and the body frame is shown in (7.29).

$$ {}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R} = \mathbf{R}_z(-90^\circ)\mathbf{R}_x(-90^\circ) = \begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix} \tag{7.29} $$

Where:

${}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R}$	is the rotation matrix from the camera frame to the body frame
$\mathbf{R}_z(\alpha)$	is a rotation matrix of an angle α around the x axis
$\mathbf{R}_x(\alpha)$	is a rotation matrix of an angle α around the z axis

The camera itself is not located in the center of the drone why a small extrinsic transformation has to be included as part of the transformation between the camera frame and the body frame. Let the location of the camera within the body frame be denoted ${}^{\mathrm{B}}\boldsymbol{p}_C = \begin{bmatrix} x_{\text{off}}, & y_{\text{off}}, & z_{\text{off}} \end{bmatrix}^T$, defining the translational camera offset. The camera to body frame transformation is then described by:

$$ {}^{\mathrm{B}}_{\mathrm{C}}\mathbf{T} = \begin{bmatrix} {}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R} & {}^{\mathrm{B}}\boldsymbol{p}_C \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & x_{\text{off}} \\ -1 & 0 & 0 & y_{\text{off}} \\ 0 & -1 & 0 & z_{\text{off}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.30} $$

The body frame of the drone is transformed according to the drone pose, ${}^{\mathrm{E}}\boldsymbol{s} = \begin{bmatrix} x & y & z & \psi \end{bmatrix}^T$ and the roll and pitch angles, ${}^{\mathrm{E}}\phi$ and ${}^{\mathrm{E}}\theta$, taken as an input from the Extended Kalman filter estimator. In Section A.1, the rotation matrix, ${}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}$, concatenating the roll, pitch and yaw rotations from body frame to earth frame is derived resulting in the rotation function:

$$ {}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}(\phi,\theta,\psi) = \begin{bmatrix} c_\psi c_\theta & c_\psi s_\theta s_\phi - s_\psi c_\phi & c_\psi s_\theta c_\phi + s_\psi s_\phi \\ s_\psi c_\theta & s_\psi s_\theta s_\phi + c_\psi c_\phi & s_\psi s_\theta c_\phi - c_\psi s_\phi \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi \end{bmatrix} \tag{7.31} $$

Combining this rotation matrix with the translational part of the pose vector results in the transformation matrix from earth frame to body frame:

$$ {}^{\mathrm{E}}_{\mathrm{B}}\mathbf{T} = \begin{bmatrix} {}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R} & {}^{\mathrm{E}}\boldsymbol{s}_{xyz} \\ \mathbf{0} & 1 \end{bmatrix} \tag{7.32} $$

The final transformation from camera frame to earth frame is found by concatenating these two transformations:

$$ {}^{\mathrm{E}}_{\mathrm{C}}\mathbf{T} = {}^{\mathrm{E}}_{\mathrm{B}}\mathbf{T}\,{}^{\mathrm{B}}_{\mathrm{C}}\mathbf{T} = \begin{bmatrix} {}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R} & {}^{\mathrm{E}}\boldsymbol{s}_{xyz} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} {}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R} & {}^{\mathrm{B}}\boldsymbol{p}_C \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} {}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}\,{}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R} & {}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}\,{}^{\mathrm{B}}\boldsymbol{p}_C + {}^{\mathrm{E}}\boldsymbol{s}_{xyz} \\ \mathbf{0} & 1 \end{bmatrix} \tag{7.33} $$

As the last step of the extrinsic derivation, the inverse transformation has to be found, as a transformation from world landmarks into the camera frame is needed such that they can be projected on to the image plane. Using the fact the transformation in (7.33) is an affine transformation and the fact that ${}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}\,{}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R}$ is invertible, since ${}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}\,{}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R}$ is a rotation matrix, the inverse of ${}^{\mathrm{E}}_{\mathrm{C}}\mathbf{T}$ is found by [25]:

$$ {}^{\mathrm{C}}_{\mathrm{E}}\mathbf{T} = {}^{\mathrm{E}}_{\mathrm{C}}\mathbf{T}^{-1} = \begin{bmatrix} \left({}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}\,{}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R}\right)^T & -\left({}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}\,{}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R}\right)^T \left({}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}\,{}^{\mathrm{B}}\boldsymbol{p}_C + {}^{\mathrm{E}}\boldsymbol{s}_{xyz}\right) \\ \mathbf{0} & 1 \end{bmatrix} \tag{7.34} $$

7.3.3 Measurement model

With both a model of the frame transformation (extrinsics) and a model of the projection on to the image plane (intrinsics), a final measurement model can be found by combination. A given landmark within the world, ${}^{\mathrm{E}}\boldsymbol{l}_i$, defined by its location ${}^{\mathrm{E}}x_l$, ${}^{\mathrm{E}}y_l$ and ${}^{\mathrm{E}}z_l$ is transformed into the camera frame by applying the inverse transformation matrix found.

$$\begin{bmatrix} {}^{\mathrm{C}}\boldsymbol{l}_i \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{\mathrm{C}}x_l \\ {}^{\mathrm{C}}y_l \\ {}^{\mathrm{C}}z_l \\ 1 \end{bmatrix} = {}^{\mathrm{C}}_{\mathrm{E}}\mathbf{T} \begin{bmatrix} {}^{\mathrm{E}}\boldsymbol{l}_i \\ 1 \end{bmatrix} \tag{7.35}$$

Writing out (7.35) gives:

$$\begin{aligned} {}^{\mathrm{C}}\boldsymbol{l}_i &= \left({}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}\,{}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R}\right)^T {}^{\mathrm{E}}\boldsymbol{l}_i - \left({}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}\,{}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R}\right)^T \left({}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}\,{}^{\mathrm{B}}\boldsymbol{p}_C + {}^{\mathrm{E}}\boldsymbol{s}_{xyz}\right) \\ &= {}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R}^T {}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}^T \left({}^{\mathrm{E}}\boldsymbol{l}_i - {}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}\,{}^{\mathrm{B}}\boldsymbol{p}_C - {}^{\mathrm{E}}\boldsymbol{s}_{xyz}\right) \\ &= {}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R}^T {}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}^T \left({}^{\mathrm{E}}\boldsymbol{l}_i - {}^{\mathrm{E}}\boldsymbol{s}_{xyz}\right) - {}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R}^T {}^{\mathrm{B}}\boldsymbol{p}_C \\ &= {}^{\mathrm{C}}_{\mathrm{E}}\mathbf{R} \left({}^{\mathrm{E}}\boldsymbol{l}_i - {}^{\mathrm{E}}\boldsymbol{s}_{xyz}\right) - {}^{\mathrm{C}}_{\mathrm{B}}\mathbf{R}\,{}^{\mathrm{B}}\boldsymbol{p}_C \end{aligned} \tag{7.36}$$

Where the rotation ${}^{\mathrm{C}}_{\mathrm{E}}\mathbf{R}$ is defined by the function:

$${}^{\mathrm{C}}_{\mathrm{E}}\mathbf{R}(\phi,\theta,\psi) = \left({}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}\,{}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R}\right)^T = \begin{bmatrix} -c_\psi s_\theta s_\phi + s_\psi c_\phi & -s_\psi s_\theta s_\phi - c_\psi c_\phi & -c_\theta s_\phi \\ -c_\psi s_\theta c_\phi - s_\psi s_\phi & -s_\psi s_\theta c_\phi + c_\psi s_\phi & -c_\theta c_\phi \\ c_\psi c_\theta & s_\psi c_\theta & -s_\theta \end{bmatrix} \tag{7.37}$$

The coordinates of a given landmark within the current camera frame are found by using (7.36) corresponding to:

$$\begin{aligned} {}^{\mathrm{C}}x_l &= y_{\mathrm{off}} + \left(-c_\psi s_\theta s_\phi + s_\psi c_\phi\right)(x_l - x) + \left(-s_\psi s_\theta s_\phi - c_\psi c_\phi\right)(y_l - y) - c_\theta s_\phi (z_l - z) \\ {}^{\mathrm{C}}y_l &= z_{\mathrm{off}} + \left(-c_\psi s_\theta c_\phi - s_\psi s_\phi\right)(x_l - x) + \left(-s_\psi s_\theta c_\phi + c_\psi s_\phi\right)(y_l - y) - c_\theta c_\phi (z_l - z) \\ {}^{\mathrm{C}}z_l &= -x_{\mathrm{off}} + c_\theta c_\psi (x_l - x) + c_\theta s_\psi (y_l - y) - s_\theta (z_l - z) \end{aligned} \tag{7.38}$$

Combining (G.6) and (G.7) from Appendix G for the image plane projection using the intrinsics and the equations for the transformed landmark coordinate within the camera frame, (7.38), the measurement vector components become:

$$\begin{aligned} x_c &= a_x \frac{{}^{\mathrm{C}}x_l}{{}^{\mathrm{C}}z_l} + x_0 \\ y_c &= a_y \frac{{}^{\mathrm{C}}y_l}{{}^{\mathrm{C}}z_l} + y_0 \\ d &= {}^{\mathrm{C}}z_l \end{aligned} \tag{7.39}$$

7.3.4 Inverse measurement model

The measurement model presented previously models how a landmark within the world is projected on to the image plane. To be able to insert new landmarks into the world when detected, an inverse measurement model is needed. The inverse measurement model takes in a measurement, $\boldsymbol{z}_c$, containing a detected pixel coordinate, (x_c, y_c) and depth, d, and performs a deprojection, shown in (7.40).

$$
\begin{aligned}
{}^{\mathrm{C}}x_l &= d\frac{x_c - x_0}{a_x} \\
{}^{\mathrm{C}}y_l &= d\frac{y_c - y_0}{a_y} \\
{}^{\mathrm{C}}z_l &= d
\end{aligned}
\tag{7.40}
$$

When the detected landmark has been deprojected into the camera frame, the landmark is transformed in reverse order into the earth frame as shown in (7.41).

$$
{}^{\mathrm{E}}\boldsymbol{l}_i = {}^{\mathrm{E}}\boldsymbol{s}_{xyz} + {}^{\mathrm{E}}_{\mathrm{B}}\mathbf{R}\left({}^{\mathrm{B}}_{\mathrm{C}}\mathbf{R}\,{}^{\mathrm{C}}\boldsymbol{l}_i + {}^{\mathrm{B}}\boldsymbol{p}_C\right) \tag{7.41}
$$

7.3.5 Measurement model Jacobians

To be able to propagate the noise and covariance matrices in the Extended Kalman filters within FastSLAM a set of Jacobians of the measurement model has to be derived. The first Jacobian to derive is the measurement model Jacobian with respect to landmarks, a Jacobian in where the pose is static hence the rotation matrix function, ${}^{\mathrm{C}}_{\mathrm{E}}\mathbf{R}(\phi,\theta,\psi)$, reduces to just constants. The resulting Jacobian is a function of the current pose, roll and pitch angle to form the rotation matrix constants and the current position of the landmark.

$$
\boldsymbol{H}_{c,l_i}(\boldsymbol{s}^k,\phi^k,\theta^k,\boldsymbol{l}_i) = \frac{\partial h_c}{\partial \boldsymbol{l}_i} = \begin{bmatrix} \frac{\partial x_c}{\partial x_l} & \frac{\partial x_c}{\partial y_l} & \frac{\partial x_c}{\partial z_l} \\ \frac{\partial y_c}{\partial x_l} & \frac{\partial y_c}{\partial y_l} & \frac{\partial y_c}{\partial z_l} \\ \frac{\partial z_c}{\partial x_l} & \frac{\partial y_c}{\partial y_l} & \frac{\partial z_c}{\partial z_l} \end{bmatrix} \tag{7.42}
$$

The second Jacobian needed is the one of the measurement model with respect to the state vector, describing how changes in the state affect the measurement vector.

$$
\boldsymbol{H}_{c,s}(\boldsymbol{s}^k,\phi^k,\theta^k,\boldsymbol{l}_i) = \frac{\partial h_c}{\partial \boldsymbol{s}^k} = \begin{bmatrix} \frac{\partial x_c}{\partial x^k} & \frac{\partial x_c}{\partial y^k} & \frac{\partial x_c}{\partial z^k} & \frac{\partial x_c}{\partial \psi^k} \\ \frac{\partial y_c}{\partial x^k} & \frac{\partial y_c}{\partial y^k} & \frac{\partial y_c}{\partial z^k} & \frac{\partial y_c}{\partial \psi^k} \\ \frac{\partial z_c}{\partial x^k} & \frac{\partial z_c}{\partial y^k} & \frac{\partial z_c}{\partial z^k} & \frac{\partial z_c}{\partial \psi^k} \end{bmatrix} \tag{7.43}
$$

This Jacobian, derived by the help of MATLAB, includes a lot of sine and cosine terms as derivatives of the rotation matrix function, ${}^{\mathrm{C}}_{\mathrm{E}}\mathbf{R}(\phi,\theta,\psi)$, are included. However, one important aspect to notice is how none of the depth-row elements, as shown in (7.44), depends on the roll angle, as the depth axis corresponds to the x-axis, around which roll is performed.

$$
\begin{bmatrix} \frac{\partial z_c}{\partial x^k} & \frac{\partial z_c}{\partial y^k} & \frac{\partial z_c}{\partial z^k} & \frac{\partial z_c}{\partial \psi} \end{bmatrix} = \begin{bmatrix} -c_\theta c_\psi & -c_\theta s_\psi & s_\theta & \Big(c_\theta s_\psi (x - x_l) - c_\theta c_\psi (y - y_l)\Big) \end{bmatrix} \tag{7.44}
$$

Finally, the last Jacobian to derive of the measurement model is related to the additive noise which is expected to affect the measurements. Assuming that the ArUco detector is capable of exactly locating the corner of a detected marker within the image, the noise introduced by the detector will only consist of possible quantization errors being ± 1 pixel on both x- and y-axis. For the depth measurement any possible deviations in the disparity map and internal calculation of the depth value will result in measurement errors. It is assumed that this noise on the depth measurement can be modelled as additive noise whose variance should be tuned when testing. Thereby, all noise elements affect the measurement vector as additive noise, why the Jacobian of the measurement model with respect to the additive noise variable, $\boldsymbol{v}$, is just an identity matrix.

$$
\boldsymbol{H}_{c,v} = \frac{\partial h_c}{\partial \boldsymbol{v}^k} = \begin{bmatrix} \frac{\partial x_c}{\partial v^k_{x_c}} & \frac{\partial x_c}{\partial v^k_{y_c}} & \frac{\partial x_c}{\partial v^k_{d}} \\ \frac{\partial y_c}{\partial v^k_{x_c}} & \frac{\partial y_c}{\partial v^k_{y_c}} & \frac{\partial y_c}{\partial v^k_{d}} \\ \frac{\partial z_c}{\partial v^k_{x_c}} & \frac{\partial z_c}{\partial v^k_{y_c}} & \frac{\partial z_c}{\partial v^k_{d}} \end{bmatrix} = \mathbf{I}_3 \tag{7.45}
$$

7.4 GOT Measurement model

It was decided in Section 5.2 that the FastSLAM algorithm should include GOT measurements as another type of measurement. Including the GOT measurements should ensure that the coordinate system of the estimated position aligns with the GOT coordinate system such that the position controller can track way-points given in the GOT frame. One way of including the GOT measurements into FastSLAM is by using it as relative distance measurement to a virtual landmark placed in the origo, $(0, 0, 0)$, of the GOT coordinate system. Being another relative distance measurement it can be included into the FastSLAM algorithm by developing a measurement model on the form shown in (7.46) where additive noise is considered.

$$\boldsymbol{z}_G^k = h_G(\boldsymbol{s}^k, \boldsymbol{l}_{\text{GOT}}) + \boldsymbol{v}^k \tag{7.46}$$

Where:

$\boldsymbol{z}_G^k$	is the Games on Track measurement vector at time k
$\boldsymbol{s}^k$	is the pose of the drone at time k
$\boldsymbol{l}_{\text{GOT}}$	is the position vector of the virtual landmark in earth frame
$\boldsymbol{v}^k$	is the measurement noise at time k

The measurement vector contains the three coordinate components described in (7.47).

$$\boldsymbol{z}_G = \begin{bmatrix} x_{\text{GOT}} \\ y_{\text{GOT}} \\ z_{\text{GOT}} \end{bmatrix} \tag{7.47}$$

It is assumed that the GOT system is installed in such a way that the whole environment is covered similarly. As GOT measurements are only position measurements they are assumed not to be affected by the orientation of the drone. It is expected that additive noise, $\boldsymbol{v}^k$, will model the stochastic part of the measurement reasonably well with the determined covariance matrix from (2.2).

$$\boldsymbol{v}^k \sim \mathcal{N}\left(\boldsymbol{0}_{3\times 1}, \Sigma_{\text{GOT}}\right) \tag{7.48}$$

7.4.1 Measurement model

A simple relative measurement model describing how the GOT system gives measurements of the drone position relative to the GOT origo landmark, is put up in (7.49). The model is designed such that the length of the measurement vector will increase if the drone is moved further away from the GOT origo.

$$h_G(\boldsymbol{s}^k, \boldsymbol{l}_{\text{GOT}}) = \boldsymbol{s}_{xyz}^k - \boldsymbol{l}_{\text{GOT}} \tag{7.49}$$

The measurement will increase in the direction of the movement which agrees with the expected measurements from GOT. The corresponding Jacobian describing this change is shown in (7.50)

$$\boldsymbol{H}_{G,s} = \frac{\partial h_G}{\partial \boldsymbol{s}^k} = \begin{bmatrix} \frac{\partial x_G}{\partial x^k} & \frac{\partial x_G}{\partial y^k} & \frac{\partial x_G}{\partial z^k} & \frac{\partial x_G}{\partial \psi^k} \\ \frac{\partial y_G}{\partial x^k} & \frac{\partial y_G}{\partial y^k} & \frac{\partial y_G}{\partial z^k} & \frac{\partial y_G}{\partial \psi^k} \\ \frac{\partial z_G}{\partial x^k} & \frac{\partial z_G}{\partial y^k} & \frac{\partial z_G}{\partial z^k} & \frac{\partial z_G}{\partial \psi^k} \end{bmatrix} = \begin{bmatrix} \mathbf{I}_3 & \mathbf{0}_{3\times 1} \end{bmatrix} \tag{7.50}$$

The Jacobian of the measurement vector with respect to the landmark location is shown in (7.51). If the GOT origo is moved towards the current position of the drone the length of the measurement vector will decrease, hence the negative Jacobian.

$$\boldsymbol{H}_{G,l_{\text{GOT}}} = \frac{\partial h_G}{\partial \boldsymbol{l}_{\text{GOT}}} = \begin{bmatrix} \frac{\partial x_G}{\partial x_l} & \frac{\partial x_G}{\partial y_l} & \frac{\partial x_G}{\partial z_l} \\ \frac{\partial y_G}{\partial x_l} & \frac{\partial y_G}{\partial y_l} & \frac{\partial y_G}{\partial z_l} \\ \frac{\partial z_G}{\partial x_l} & \frac{\partial y_G}{\partial y_l} & \frac{\partial z_G}{\partial z_l} \end{bmatrix} = -\mathbf{I}_3 \tag{7.51}$$

where x_l, y_l and z_l is coordinate of the GOT landmark corresponding to origo. However, due to the static landmark location corresponding to origo, this Jacobian will not have any effect.
Due to the additive noise the Jacobian with respect to the noise variable, $\boldsymbol{v}$, becomes an identity matrix as well.

$$\boldsymbol{H}_{G,v} = \frac{\partial h_G}{\partial \boldsymbol{v}^k} = \begin{bmatrix} \frac{\partial x_G}{\partial v^k_{x_G}} & \frac{\partial x_G}{\partial v^k_{y_G}} & \frac{\partial x_G}{\partial v^k_{z_G}} \\ \frac{\partial y_G}{\partial v^k_{x_G}} & \frac{\partial y_G}{\partial v^k_{y_G}} & \frac{\partial y_G}{\partial v^k_{z_G}} \\ \frac{\partial z_G}{\partial v^k_{x_G}} & \frac{\partial z_G}{\partial v^k_{y_G}} & \frac{\partial z_G}{\partial v^k_{z_G}} \end{bmatrix} = \mathbf{I}_3 \tag{7.52}$$

7.4.2 Landmark initialization

Initially, before starting the FastSLAM algorithm, a landmark placed in $(0, 0, 0)$ is inserted into the map of landmarks within all particles. Therefore an inverse measurement model will not be needed as no other GOT landmarks would have to be inserted into the map after initialization.As the origo of the GOT coordinate system and the origo of the FastSLAM coordinate system should align completely, the location of this virtual landmark is fully known and deterministic so the covariance of the inserted landmark is set to $\mathbf{0}_{3\times3}$ to reflect this.

Whenever a GOT measurement is available, the likelihood of getting such a measurement is used when calculating the importance weights of each particle. Using the pose of each particle and the deterministic location of the landmark, due to the zero covariance, the likelihood of getting this measurement is solely evaluated based on the measurement model and measurement covariance. This will result in particles whose pose are closer to the GOT measurement getting higher weights than particles being far away from the received measurement. In return this should align the FastSLAM coordinate system with GOT.

7.5 Implementation

With the given motion and measurement models the FastSLAM algorithm needs to be implemented in C++ such that these models can be incorporated. The following implementation section is intended to give the reader an overview of the specific implementation of the FastSLAM algorithm made for this project. It has been decided to use an object oriented programming language for the implementation of the algorithm, due to the nature of the algorithm, where particles and their associated maps are constantly copied and reused. C++ is one of the object oriented programming languages supported by ROS as presented in Appendix I, why this language is chosen.

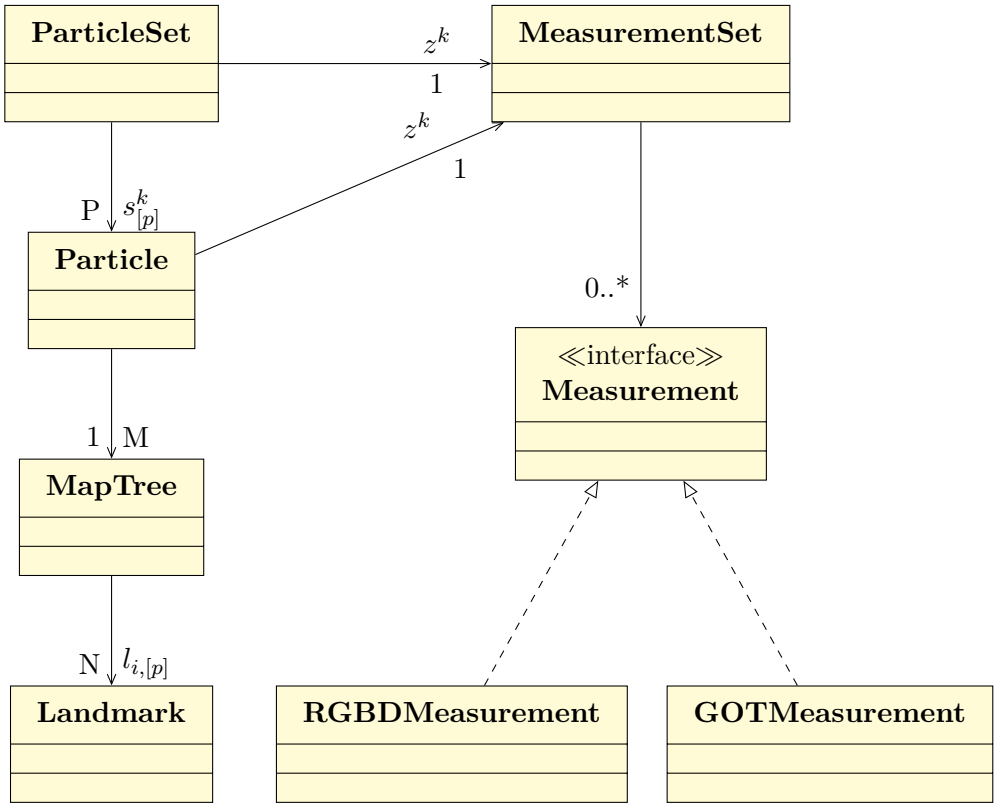


Figure 7.2: UML class diagram showing the connections between the main classes of the FastSLAM implementation made for this project.

Figure 7.2 shows a class diagram of the main classes chosen for the implementation and the relations between them. An instance of the **ParticleSet** class is associated to P instances of the **Particle** class. The **Particle** class implements step 2a to 2f of the algorithm described in Section 7.1. The **ParticleSet** class implements step 1 and 2g, and furthermore includes functions to tell each of the instances of the **Particle** class to perform step 2a to 2f. To be able to perform these steps, both the **ParticleSet** and **Particle** class are associated to the **MeasurementSet** class that keeps track of the available measurements at time k, $\boldsymbol{z}^k$.

To be able to use different types of measurements in the algorithm, the **Measurement** interface has been defined such that it ensures that each type of measurement have the necessary attributes and operations to be used in the **ParticleSet** and **Particle** class. To store the information about the map, **M**, each particle is associated to an instance of the **MapTree** class, that implements the binary tree functionalities described in Section 7.1. Therefore each instance of the **MapTree** class is also associated with N instances of the **Landmark** class.

To implement the mathematical steps of FastSLAM, which involves vector and matrix arithmetic, the Eigen3 library is used. No specific algorithm for resampling is suggested for the FastSLAM algorithm as mentioned in Section 7.1. Therefore it is decided to use the Resampling Wheel algorithm described in [26] and [27] by Sebastion Thrun, one of the co-authors of FastSLAM. This algorithm was chosen due to the ease of implementation.

When implementing the FastSLAM algorithm, converting the math presented in Section 7.1 and Section E.9, into C++ code, a few practical problems have to be considered. One of them, step 2c, involves drawing random samples from an N-dimensional Gaussian distribution with an arbitrary mean vector and covariance matrix, also known as a multivariate Gaussian.

To do so the procedure described in [28], was implemented. To draw a sample, $\boldsymbol{x}$, of the distribution $\mathcal{N}(\mu, \mathbf{C})$ the procedure is as follows

1. Perform a Cholesky decomposition of the covariance matrix, $\mathbf{C}$ to yield and $N \times N$ non-singular matrix $\mathbf{G}$, where $\mathbf{C} = \mathbf{G}\mathbf{G}^T$.
2. Draw a realization $\mathbf{u}$ of the PDF, $\mathcal{N}(\mathbf{0}, \mathbf{I})$.
3. The desired sample is then calculated as $\mathbf{x} = \mathbf{G}\mathbf{u} + \mu$

To perform the second step the code found in [29] is used. The first step can furthermore be performed by using the Eigen3 library, but requires a symmetric matrix. Unfortunately the steps involved in calculating the covariance matrix do sometimes not yield a symmetric matrix, probably due to numeric precision. This results in a failing Cholesky decomposition due to the non-symmetric matrix and the way Eigen3 handles the decomposition. To solve this issue the approach of symmetrizing matrices proposed in [30] is used. This symmetrization is simply done as follows

$$\mathbf{C}^* = \frac{1}{2}\left(\mathbf{C} + \mathbf{C}^{\mathbf{T}}\right) \tag{7.53}$$

In this project it is chosen to initialize the FastSLAM algorithm by placing the initial particles based on the first GOT position measurement and with a yaw angle of 0 degrees, assuming that the drone is initially aligned with the GOT frame (zero heading). Furthermore, all particles are initialized with a fictive landmark, representing the GOT origo. This landmark is initialied with a zero mean, $x = y = z = 0$, and a covariance matrix of zero, thereby forcing the uncertainty of the origo to be zero, preventing any movement of the landmark.

7.6 Test of the FastSLAM 2 algorithm

In this section the implemented FastSLAM algorithm is tested and verified. The performed test is based on considerations about the scenarios which can be experienced during a flight in a factory environment. The scenario consists of the drone flying in a planed path through areas with no GOT measurements.
The test is performed in a Motion Tracking lab equipped with a Vicon camera system. Measurements from the Vicon system are used as reference for the performance evaluation of FastSLAM. Furthermore the Vicon measurements are used as a replacement for the GOT system. The substitution of the GOT system is reasonable since the Vicon measurements can be described with the same measurement model as the GOT measurements, described in (7.49).
The described motion model within FastSLAM relies on velocity and yaw angle estimates from the Extended Kalman filter. To verify the functionality of FastSLAM seperately, and because the Extended Kalman Filter, presented in Chapter 8, has not been integrated with FastSLAM, it is decided to obtain these velocity inputs from the Vicon system. The estimate of yaw can be directly substituted with Vicon yaw measurements while the velocities are obtained through differentiation of the Vicon position measurements.

The measured heading and the velocity estimates obtained from differentiating the Vicon position measurements, corresponds to quite accurate estimates without much noise. It is not expected to get this good estimates from the EKF and it is therefore decided to make these estimates worse. The velocity estimates is worsened by low-pass filtering them to introduce a phase lag, and further worsened by adding a bias and Gaussian noise. The yaw measurement is worsened by only adding Gaussian noise to the signal.

The performance test of the FastSLAM algorithm is compared to a dead reckoning based estimate. The dead reckoning estimate is solely based on the kinematic motion model using the worsened velocity estimates and yaw angle.

A test flight is recorded in a ROS bag which allows playback of the data into the ROS environment. The recorded data contains the RGB-D camera stream and the measurements from the Vicon system. This allows several tests to be performed with the same flight, e.g. changing the time-window in where GOT measurements are available.

7.6.1 Test results

The test is designed to simulate a scenario in which the drone is flying a planned trajectory without GOT measurements. To imitate a planned trajectory the drone is remote controlled. The drone is flown in such a way that ArUco markers are visible all the time. Three different tests are performed to compare the performance of the algorithm with a varying amount of available GOT measurements. In the three executions of the algorithm the GOT measurements are always available initially and then disabled after some time.

The first test leaves GOT measurements available for the first 5 s. The second test leaves the GOT measurements available for the first 75 s. The third test leaves the GOT measurements available for the whole flight of 145 s.

The result of the flight can be seen on Figure 7.4 where the individual executions are visualized. The Vicon measurement of the four states being estimated, hence being the comparable reference, is plotted as well. Besides this measured trajectory, the dead reckoning based estimate is also shown.

From the test results it can be seen that the implementation is working and is able to estimate the position and heading of the drone, even when GOT measurements are no longer available. It is especially worth noticing how the dead reckoning based estimate quickly drifts away from the measured trajectory. On Figure 7.4 the red curve is overlaying the black curve the entire flight, indicating that only a small estimation error is present when GOT measurements are available. But from Figure 7.4 it can also be seen that larger estimation errors appears when GOT measurements are lost. The estimation error for the different tests is presented in Figure 7.3 and Figure 7.5. The error is calculated as the Euclidean distance between the estimated position and the reference position measured with Vicon.

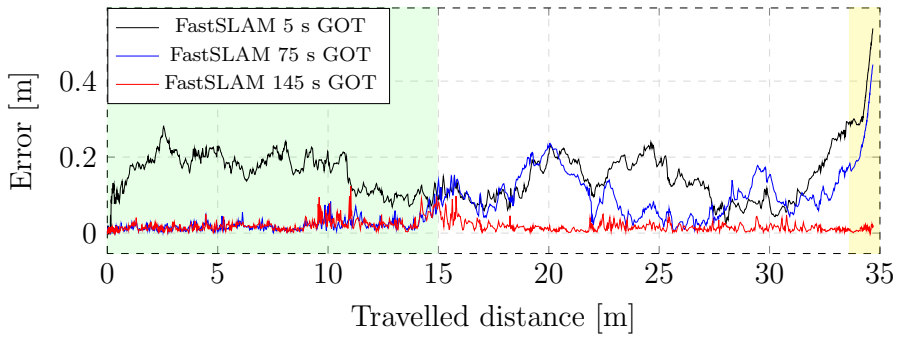


Figure 7.3: Euclidean estimation error of FastSLAM relative to the travelled distance. Yellow area indicates a duration of the flight with no ArUco markers detected, hence a region which does not represent the test scenario. The green area indicates the first 75 s of the flight.

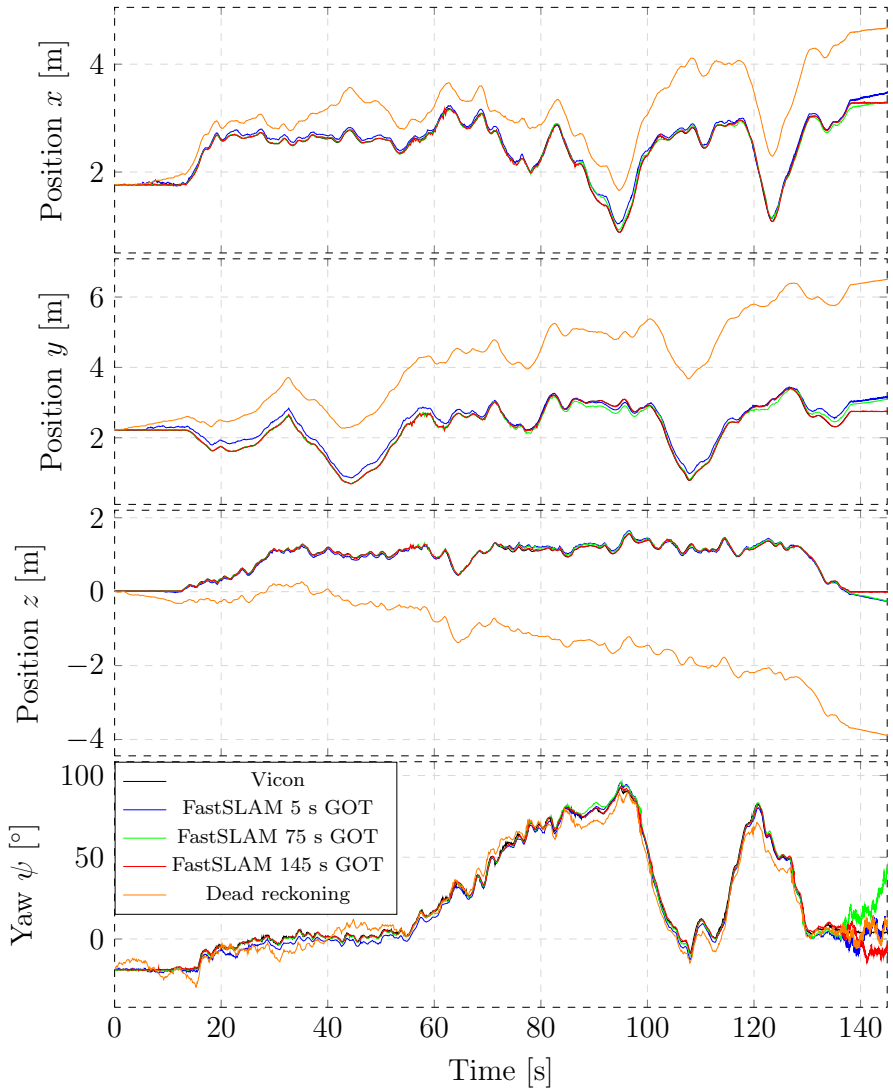


Figure 7.4: Timeseries plot showing the performance of the FastSLAM estimator. On the plot the performence of the estimator is shown for various durations of available GOT measurements.

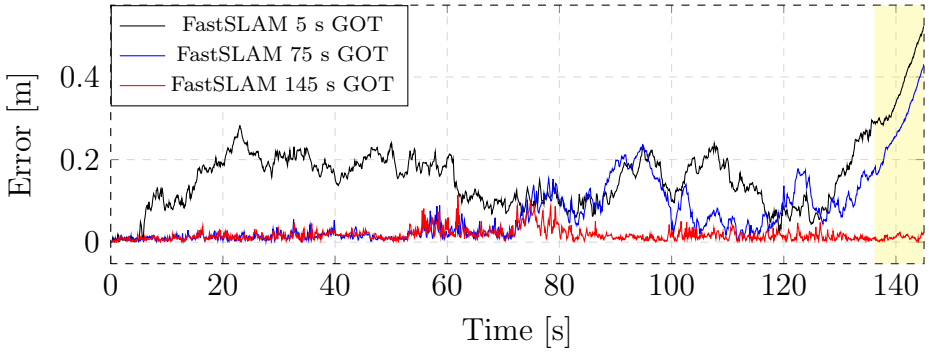


Figure 7.5: Euclidean estimation error of FastSLAM relative to time. Yellow area on the plot indicates a duration of the flight with no ArUco markers detected, hence a region which does not represent the test scenario.

The plots in Figure 7.3 and Figure 7.5 are obtained by initially performing a linear interpolation of the trajectories estimated by FastSLAM, to generate estimates matching up with the time indices of the Vicon data. At each time index an error is calculated as the Euclidean distance between the estimated position and the Vicon measurement. Figure 7.5 is obtained by plotting this estimation error as a function of time. The distance travelled is calculated as the sum of Euclidean distances between consecutive Vicon measurements. Figure 7.3 is obtained by plotting the estimation error as a function of this travelled distance.

From Figure 7.3 and Figure 7.5 it is concluded that the estimation error does not seem to depend on the travelled distance or time, as long as the drone can detect at least one marker. This is apparent by inspecting the relation between the estimation error and the travelled distance or time. For all of the three executions it is evident that the estimation error is not increasing with the travelled distance or time, however, it is evident that the estimation error is larger when no GOT measurements are available. This is seen by comparing the black and blue with the red curve. Notice that the green area in Figure 7.3 indicates the first 75 s of the flight where GOT measurements are included.

A requirement for the FastSLAM algorithm to work is that at least one marker is detected all the time. This is not fulfilled in the end of the test, hence a region which does not fulfil the requirements for the test scenario. The consequence of not detecting any markers is seen in the yellow area of Figure 7.3 and Figure 7.5. The estimation error in this area is increasing with the travelled distance, similar the dead reckoning based estimate shown in Figure 7.4. Finally the increase in estimation error when no GOT measurements are available, can be explained by estimation errors within the landmark positions and RGB-D measurement model errors.

The 3D plot shown in Figure 7.6 visualizes both the measured and estimated trajectory while also illustrating the actual marker locations and the estimated marker locations extracted from the map of FastSLAM. The estimates comes from the test where GOT measurements are only included the first 5 s.

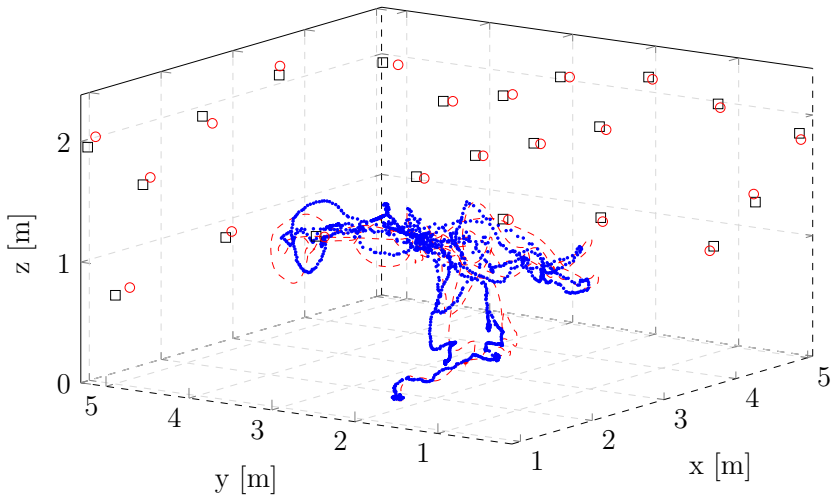


Figure 7.6: 3D visualization of the test flight with the manually controlled drone. The red reference trajectory is measurements taken with the Vicon system. The blue scatter trajectory is the estimated position from FastSLAM. Red circles indicate the actual placement of ArUco markers while the black squares is the estimated position of the markers.

The measured position of landmarks is indicated with red circles while the black markers indicates the resulting estimated position extracted from the FastSLAM map at the end of the test. A small error between every estimated marker position and the actual position is apparent. This error is partly due to measurement errors resulting from the physical measurement of the marker locations and actual estimation errors within the FastSLAM algorithm. The actual estimation error in the estimated position of the markers becomes important when GOT measurements are not available as incorrectly placed markers will easily result in estimation errors within the estimated position and heading of the drone. This is evident from Figure 7.3 when comparing the black and blue graph with the red graph.

7.6.2 Test discussion

The performed test demonstrates a working FastSLAM solution which is able to estimate the position and heading of the drone even when GOT measurements are not available. From the test it is concluded that small but bounded estimation errors are present when the GOT measurements are not available. However, the Motion Tracking lab used for testing the algorithm is putting up some physical constrains on the test which can be performed due to its limited size. The limited size does not allow a scenario where the drone is flying for a longer distance without GOT measurements in a not yet discovered area. In such a scenario the error might not be bounded but instead increase with the travelled distance.
Furthermore the performance of the FastSLAM algorithm has not been tested in conjunction with the EKF, why the motion model with actual velocity estimates from the EKF, has not been verified.

8 Full-state EKF estimator

In this chapter the Extended Kalman filter used for estimating the entire state vector of the system is described. In Section E.5, the probabilistic theory behind the discrete Extended Kalman filter is shown, reaching the description of how the state estimation problem can be solved if both a motion model and a measurement model are assumed. The motion model should be on the form described in (8.1). And the measurement model on the form described in (8.2). Within this chapter, an introduction to the objective of the Extended Kalman Filter is first given, followed by a description of the assumed measurement and motion models. Afterwards follows a description of the design and how the noise is assumed to affect these models. Finally the estimator is implemented and test results are presented.

$$\boldsymbol{\chi}^k = f\left(\boldsymbol{\chi}^{k-1}, \boldsymbol{u}^{k-1}, \boldsymbol{w}^{k-1}\right) \tag{8.1}$$

$$\boldsymbol{z}^k = h\left(\boldsymbol{\chi}^k, \boldsymbol{v}^k\right) \tag{8.2}$$

$$\boldsymbol{w}^k \sim \left(0, \mathrm{Cov}\left(\boldsymbol{w}^k\right)\right) \tag{8.3}$$

$$\boldsymbol{v}_i^k \sim \left(0, \mathrm{Cov}\left(\boldsymbol{v}_i^k\right)\right) \tag{8.4}$$

The Extended Kalman filter designed in this chapter has multiple purposes. Firstly, it is known from Chapter 6 that the controllers need a Full-State estimate as feedback, since they are designed as full state feedback controllers. The full state vector to estimate is shown in (8.5).

$$\hat{\boldsymbol{\chi}} = \begin{bmatrix} {}^{\mathrm{H}}x & {}^{\mathrm{H}}\dot{x} & \boldsymbol{\chi}_\theta & | & {}^{\mathrm{H}}y & {}^{\mathrm{H}}\dot{y} & \boldsymbol{\chi}_\phi & | & \psi & | & z & \dot{z} & \chi_{\ddot{z}} \end{bmatrix}^T \tag{8.5}$$

Besides providing the controllers with estimates, it is known that the FastSLAM algorithm uses a motion model with velocities as input, which should be supplied by the EKF. Lastly, the PX4 attitude estimator has to be supplied with heading estimates since the on-board attitude estimator does not have any sensor to measure the yaw angle except integrating the gyroscope, which is likely to drift.

8.1 Measurement model

From Chapter 5, it is known that the EKF is supplied with measurements from the FastSLAM algorithm and the PX4 attitude estimator. From the FastSLAM algorithm it is supplied with an estimated position in world frame and an estimated yaw angle. Before the position from FastSLAM can be used, it has to be rotated from the EF to the HF in which the EKF is estimating the position. The estimates from the FastSLAM are related to the state vector with the measurement model in (8.6). From the PX4 attitude estimator it is supplied with an attitude estimate, which is described by the state vector in (8.7).
In the two measurement models it is assumed that the noise affecting the measurements is additive, represented by the noise vectors $\boldsymbol{v}_{\mathrm{fs}}$ and $\boldsymbol{v}_{\mathrm{PX4}}$. The measurements of roll and pitch coming from the PX4 attitude estimator are sensitive to initial calibration and prone to drift due to imperfect numerical integration of angular velocities measured by the on-board gyroscope.

This is why the measurement model for the $\boldsymbol{z}^k_{\text{PX4}}$ presented in (8.7) presents two additional states to be estimated. These two states are a bias on the measurements of the angles.

$$\boldsymbol{z}^k_{\text{fs}} = \begin{bmatrix} {}^H x \\ {}^H y \\ z \\ \psi \end{bmatrix}^k = h_{\text{fs}}({}^H x, {}^H y, z, \psi, \boldsymbol{v}_{\text{fs}}) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 & 0 \\ \sin(\psi) & \cos(\psi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^H x \\ {}^H y \\ z \\ \psi \end{bmatrix}^k + \boldsymbol{v}^k_{\text{fs}} \tag{8.6}$$

$$\boldsymbol{z}^k_{\text{PX4}} = \begin{bmatrix} \theta \\ \phi \\ \psi \end{bmatrix}^k = h_{\text{PX4}}(\chi_\theta, \chi_\phi, \psi, b_\theta, b_\phi, \boldsymbol{v}_{\text{PX4}}) = \begin{bmatrix} C_\theta & 0 & 0 & 1 & 0 \\ 0 & C_\phi & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \chi_\theta \\ \chi_\phi \\ \psi \\ b_\theta \\ b_\phi \end{bmatrix}^k + \boldsymbol{v}^k_{\text{PX4}} \tag{8.7}$$

Where:

$\boldsymbol{z}^k_{\text{fs}}$	is the measurement vector from the FastSLAM algorithm
$\boldsymbol{z}^k_{\text{PX4}}$	is the measurement vector from the PX4 attitude estimator
b_θ	is the bias in the pitch measurement
b_ϕ	is the bias in the roll measurement
$\boldsymbol{v}^k_{\text{fs}}$	is the noise vector affecting the estimates from the FastSLAM algorithm
$\boldsymbol{v}^k_{\text{PX4}}$	is the noise vector affecting the estimates from the PX4 attitude estimator

8.2 Motion model

The motion model used in the extended Kalman filter is the state space model derived in Section 6.1. This is a linear state-space model consisting of 16 states on the form described in (8.1), describing both the position and attitude of the drone.

In Section 6.2 it is presented how the z controller is vulnerable to disturbances entering the system at the thrust input signal. These disturbances are due to model uncertainties related to the thrust value which makes the drone hover. This value acts as an operation point for the z controller, and if it is not known, the controller will have a steady state error when trying to reach a set-point. In Section 6.2.2 it is presented how the input disturbance is modelled with an exosystem, illustrated in Figure 6.7.

The thrust value that makes the drone hover is expected to vary slowly compared with the sample rate of the extended Kalman filter, and the disturbance is therefore modelled as it is not changing. This model is presented in (8.8).

$$d^{k+1} = A_d d^k = d^k \tag{8.8}$$

Where:

d^k	is the hover thrust value

In order to include the estimation of the disturbance d^k in the extended Kalman filter, the state-space model describing the z movement has to be extended. This is done by including the model of the disturbance into the z model. The augmented state-space model for z is given in (8.9)

$$\begin{bmatrix} z \\ \dot{z} \\ \chi_{\ddot{z}} \\ d \end{bmatrix}^{k+1} = \begin{bmatrix} 1 & T_s & 0 & 0 \\ 0 & 1 & T_s C_{\ddot{z}} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & A_d \end{bmatrix} \begin{bmatrix} z \\ \dot{z} \\ \chi_{\ddot{z}} \\ d \end{bmatrix}^{k} + \begin{bmatrix} B_{\ddot{z}} \\ 0 \end{bmatrix} T^k \tag{8.9}$$

In Section 8.1 it is presented that a bias estimate has to be obtained for the pitch and roll measurements obtained from the PX4 attitude estimator, requiring then a motion model for them. This is done by modelling the biases as exogenous systems affecting the measurements. These biases are modelled as being constant since, as explained for the disturbance in the thrust input, they are expected to vary slow compared to the sample rate of the EKF.

In the design of the Extended Kalman filter it is assumed that the state noise in the motion model is additive noise. This assumption is simplifying the motion model in (8.1) to a motion model of the form in (8.10)

$$\boldsymbol{\chi}^k = f\left(\boldsymbol{\chi}^{k-1}, \boldsymbol{u}^{k-1}\right) + \boldsymbol{w}^{k-1} \tag{8.10}$$

8.3 Observability and sample rates

The rate in which the measurements are available is not expected to be the same. The ones coming from the PX4 attitude estimator are acquired with a rate of 20 Hz, which is the same as the rate the controllers run at. However, the one at which the measurements from the FastSLAM algorithm can be acquired is expected to be less than 20 Hz. The used measurement model at each iteration of the kalman filter is therefore changing between two different models: one where both the measurement vector $\boldsymbol{z}_{\text{fs}}^k$ and $\boldsymbol{z}_{\text{PX4}}^k$ are available, and another where only the measurement vector $\boldsymbol{z}_{\text{PX4}}^k$ is available. The observability of the system is analysed in order to investigate the effect of the changing measurement model.
The measurement model for $\boldsymbol{z}_{\text{fs}}^k$ is linearised in order to do the observability analysis since the Observability rank criterion is only valid for linear systems. This is deemed reasonable since the non-linearity in the measurement model is due to the position x and y being measured in a rotated frame which is not expected to change the observability of the system. The Observability rank criterion states that the Observability matrix defined in (8.11) should have full rank in order for the system to be observable.

$$\boldsymbol{\mathcal{O}} = \begin{bmatrix} \boldsymbol{C} \\ \boldsymbol{CA} \\ \boldsymbol{CA}^2 \\ \vdots \\ \boldsymbol{CA}^{n-1} \end{bmatrix} \tag{8.11}$$

Where:

n is the number of states in the system

The state-space model found in the modelling section is composed of 16 states, but as mentioned in Section 8.1 and Section 8.2 three additional states have been added. This means that the observability matrix has to have rank 19 in order for the system to be observable.

The observability matrix and its rank are found with MATLAB. The rank of the observability matrix is found to be 19 when both the measurement vector $\boldsymbol{z}_{\text{fs}}^k$ and $\boldsymbol{z}_{\text{PX4}}^k$ are available, which indicates that the system is observable. When only the measurement vector $\boldsymbol{z}_{\text{PX4}}^k$ is available the rank of the observability matrix is found to be 11, indicating that the system is not observable. The 11 states being observable are the four states from the pitch ARX model, the four states from the roll ARX model, the state from the yaw ARX model and the pitch and roll bias included in the state-space description. The remaining states not being observable are the three states from the ARX z model, the x and y position of the drone, the x and y velocities of the drone and finally the input disturbance affecting the z ARX model. The system not being observable in some iterations of the Kalman filter results in dead reckoning in the unobservable states in between measurements from the FastSLAM algorithm.

8.4 Design

Section E.5 describes that a requirement for using the discrete extended Kalman filter is to know the covariances of both the state noise vector $\boldsymbol{w}^k$ and the measurement noise vector $\boldsymbol{v}^k$. The state noise vector $\boldsymbol{w}^k$ is a 19 dimensional vector while the measurement noise vector is either a three or seven dimensional vector depending on if a FastSLAM pose estimate is available.
The FastSLAM algorithm is maintaining a time-varying covariance matrix for the current estimated pose vector which is used in the extended Kalman filter together with the estimated pose. A constant covariance matrix is used in tests where the Vicon motion tracking system is used instead of FastSLAM. The entries in the covariance matrix of the measurements vectors $\boldsymbol{z}_{\text{PX4}}^k$ and $\boldsymbol{z}_{\text{fs}}^k$ can either be estimated from measurements sampled while the drone is in rest or considered as a collection of tuning parameters.
For the design of this Kalman filter it is decided to treat the entries as tuning parameters since it is considered to be the easiest way of designing a filter with a satisfactory performance and it is likely that additional tuning has to be performed with the estimated covariance matrices. Furthermore, it is assumed that the covariance between the different measurements in both the measurement vector $\boldsymbol{z}_{\text{PX4}}^k$ and $\boldsymbol{z}_{\text{fs}}^k$ is zero. This assumption is reducing the covariance matrices to being diagonal matrices and hereby also reducing the number of parameters to tune.

Regarding the state noise vector $\boldsymbol{w}^k$ it is more involved to estimate the entries of the corresponding covariance matrix. In the design of the Kalman filter it is first of all assumed that the state noise vector is generated by a stationary process such that the covariance matrix becomes independent of time. Moreover, it is assumed that the entries in the state noise vector are uncorrelated. These two assumptions make the covariance matrix of the state noise a constant diagonal matrix. The entries in the diagonal are then used as tuning parameters the same way as for the measurement noise covariance.

8.5 Test results

In this section the test results are shown after the Kalman filter has been designed and tuned. The flight used for the test results is performed with the drone being remotely controlled by a pilot. The Kalman filter has been tested in two different scenarios, one with the measurements from the FastSLAM algorithm assumed to be available with the same rate as the update rate as the Kalman filter, and another one with the rate of the FastSLAM measurements reduced to a fifth of the update rate of the Kalman filter. The rate of the Kalman filter is 20 Hz. This means that the two scenarios tested are with FastSLAM measurements available with a rate of 20 Hz and 4 Hz.

In order to test the performance of the EKF independently of the FastSLAM algorithm performance, the FastSLAM measurements are replaced with measurement from a Vicon motion tracking system for the test.

In Figure 8.1 the plots show the performance of the Kalman filter in the three position coordinates. It can be seen that the Kalman filter is tracking the Vicon measurement properly when they are available with a rate of 20 Hz, whereas the estimate gets more noisy when the Vicon measurements are only available with 4 Hz. This noisy behaviour can be explained by the position states not being observable when the FastSLAM measurements are not available, which leads to a dead reckoning behaviour in between FastSLAM measurements. A noticeable remark is the ability of the Kalman filter to track the z position when the drone has landed. The floor level is calibrated to be at 0 m. From the plot it can be seen that the drone has landed in the last few seconds and as a consequence the operation point for the z motion model has changed. The Kalman filter is able to track this change in operation point, as shown with the red graph on the z plot.

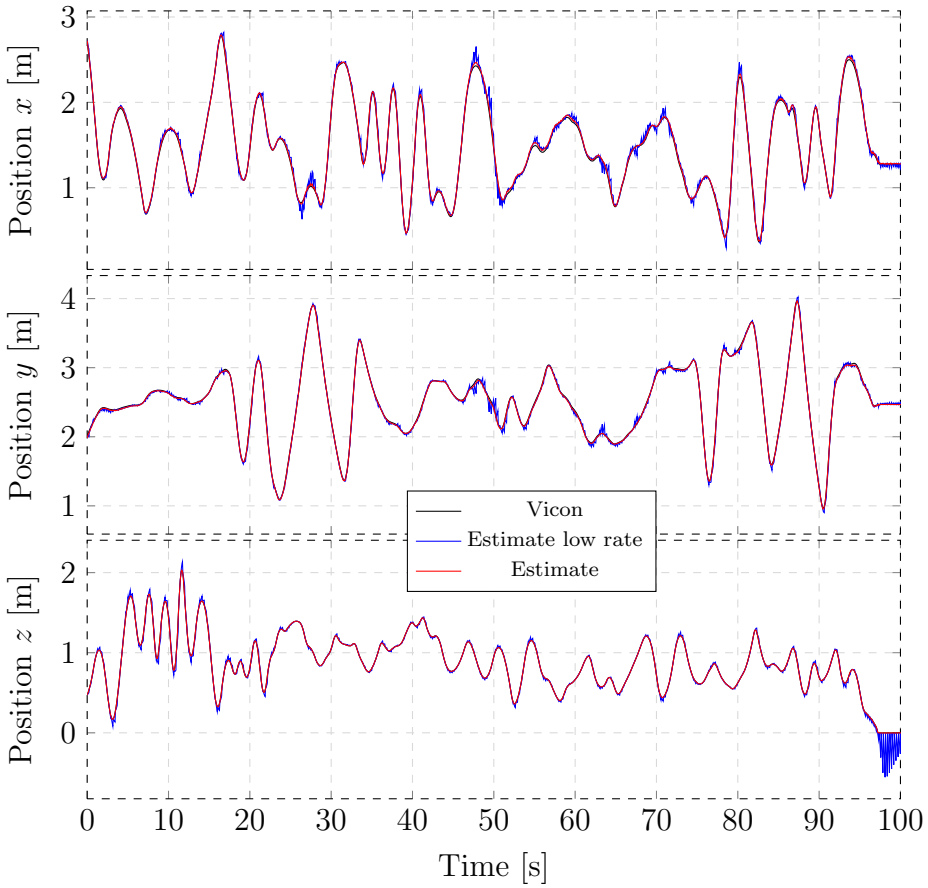


Figure 8.1: Estimation of the position of the drone. The plots is both showing the performance of the filter when the FastSLAM measurements is available with a rate of 20 Hz and 4 Hz

In Figure 8.2 similar test results are shown for the velocity states. It can be seen that the Kalman filter is worse at estimating velocities compared to positions, which can be due to the Kalman filter not having measurements of velocities but only attitude and position. In the same way as with the z position estimate, the z velocity estimate is affected by the thrust value which makes the drone hover. This is evident in the final part where the drone has landed and the thrust value is not yet estimated correctly. In this part the velocity estimate is wrong but approaching the correct velocity as the estimated thrust value is approaching the correct value. Hence it can be concluded that the estimation of the thrust value is working.

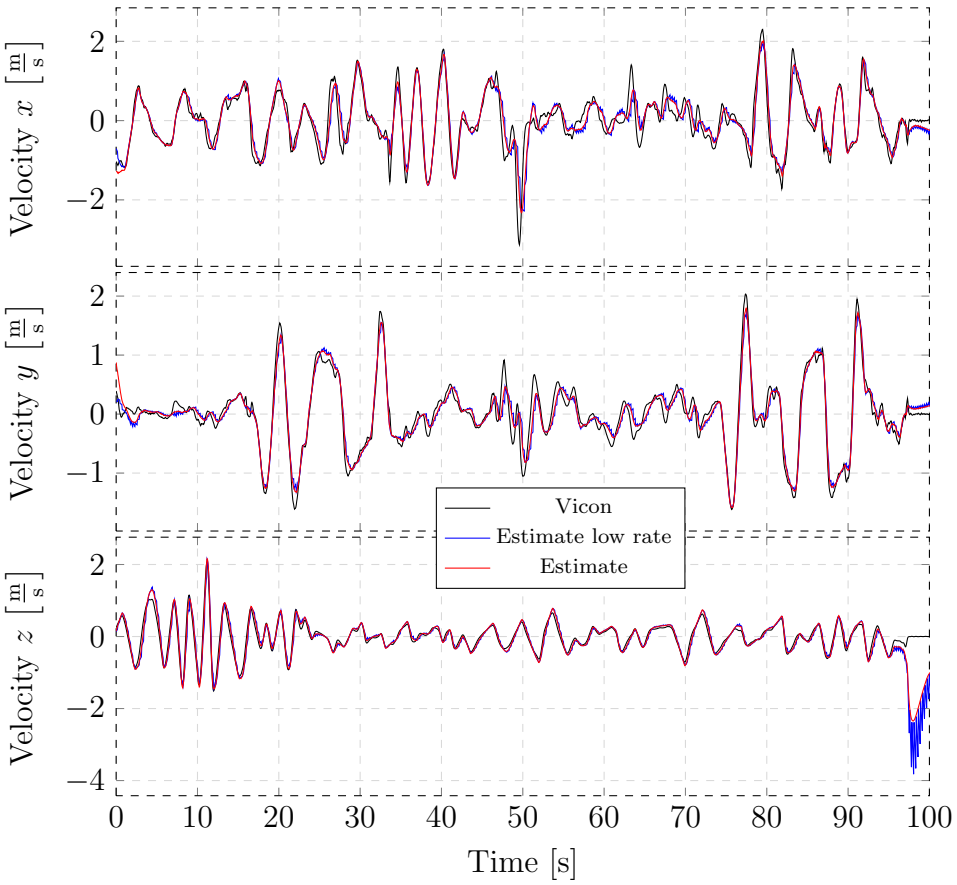


Figure 8.2: Estimation of the velocity of the drone. The plots are both showing the performance of the filter when the FastSLAM measurements are available with a rate of 20 Hz and 4 Hz

In Figure 8.3 the attitude estimation is shown for the pitch and roll angles. On the two plots the measurements from the Vicon system and PX4 attitude estimator are plotted, together with the estimate of the angles and the estimate of the biases affecting the PX4 measurements. From the plots it is evident that there is an offset between the PX4 measurements and the Vicon measurements. This offset is estimated by the extended Kalman filter which is shown as the green graphs on the plots. It can be concluded that the estimate of the biases is working as intended since the estimated pitch and roll angles are tending towards the Vicon measurements. Here it should be noted that the Kalman filter is not using the Vicon pitch and roll measurements but only position measurement to estimate the biases, which allows the desired outcome of the angles being close to the Vicon measurements.

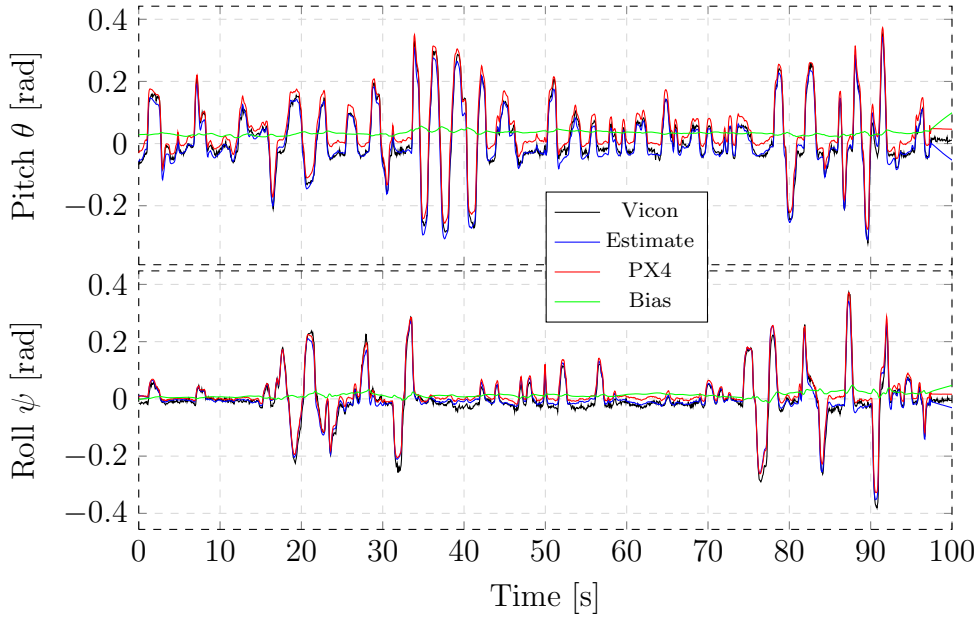


Figure 8.3: Estimation of the pitch and roll angles.

In Figure 8.4 the Yaw estimation is showed. From this plot is shown that the Kalman filter is able to track the yaw angle.

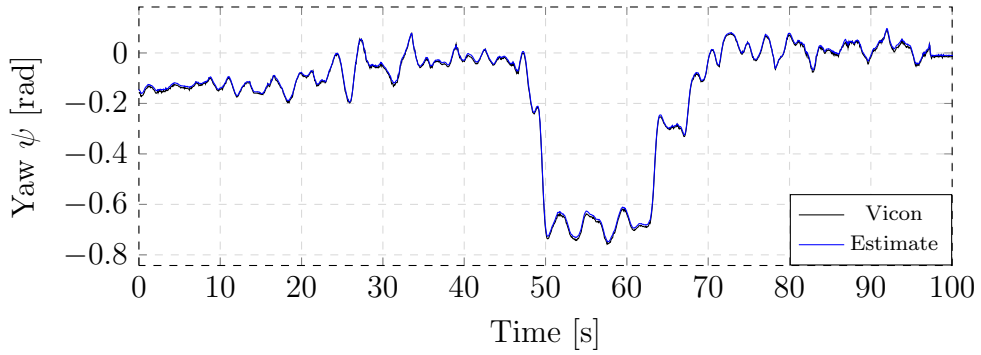


Figure 8.4: Estimation of the yaw angle.

9 Conclusions

The purpose of this project is to investigate whether a combination of a colour and depth camera can be used on a drone to increase the safety of indoor navigation in factory environments equipped with a GamesOnTrack (GOT) positioning system drop-outs and dead zones may occur. A SLAM-based absolute position estimator and controller solution for drones equipped with a PX4 flight controller, a small Linux companion computer running ROS and an RGB-D camera are developed.

Throughout a pre-analysis it is concluded how drones are notoriously unstable and needs absolute position measurements to have a stable and non-drifting position. Without any absolute position measurements a relative sensor such as a camera can be used to detect distinct landmarks within the environment, enabling simultaneous localization and mapping. It is decided to solve the SLAM problem using the Filtering-based algorithm, FastSLAM 2.0, due to its capability of providing real-time estimates, its computational efficiency, and the good stochastic framework wherein other sensors such as GOT can easily be included. To assure known correspondences to the landmarks within the environment and to increase robustness of the system it is chosen to put up static ArUco markers with known identifiers.

The proposed solution is composed of an existing drone equipped with a PX4 flight controller coupled with a set of position controllers taking in position and attitude estimates from an interconnected combination of an Extended Kalman Filter and the FastSLAM estimator taking in GOT measurements whenever available. Every element of the solution is developed within the ROS environment, and tested with the Intel Aero Ready to Fly drone equipped with a PX4 flight controller, an Intel Aero compute board running Linux and an Intel RealSense R200 RGB-D camera. The PX4 flight controller ensures a stabilized attitude and tracks the attitude and thrust references given by the position controllers.

Based on an identified ARX model, a state-space model of the system is derived and used to design two decoupled position controllers running at 20 Hz: a z-controller adjusting the thrust reference based on z-position estimates and an xy-controller adjusting the roll and pitch references based on x and y position estimates. The yaw angle is held constant by the PX4 controller. Trajectories defined as way-points within the GOT coordinate system are sent to the positions controllers as position step references. Saturations on the position errors are included to limit the velocity while allowing any step size.

The controllers get a Full-State estimate from an Extended Kalman Filter designed to take in attitude measurements from the PX4, together with position and yaw estimates from the FastSLAM estimator. The EKF furthermore estimates a roll and pitch bias to accommodate for misalignments and a thrust bias value to accommodate for battery drain.

The FastSLAM algorithm consisting of a particle filter and several Extended Kalman Filters is implemented with a simplified motion model that takes in velocity estimates from the Full-State Extended Kalman Filter. These estimates are used to predict the position of the drone. Furthermore, RGB-D measurements of ArUco markers within the environment are used to correct the particle distribution and landmark locations. Whenever available, GOT measurements are used within FastSLAM to correct the position and ensure that the coordinate system of FastSLAM converges towards the GOT coordinate system. When new landmarks are detected they are inserted into the map allowing the drone to explore new areas while still estimating the position.

Each element of the system is tested individually within a Motion Tracking lab equipped with a Vicon camera system, to verify the functionality and performance. The Full-State EKF is tested with both simulated and real measurements and compared with the measured path. The EKF is capable of estimating the necessary Full-State vector even at a reduced position measurement rate of 4 Hz. However, a small lag is apparent in the velocity estimates which seems to be slightly incorrect. By the help from the bias estimation within the Full-State EKF the z-position reference is tracked without any apparent overshoot even at decreasing battery level. The xy-controller does not perform as nicely but references are tracked and the position can be held, although with small oscillations. FastSLAM is able to provide converging position estimates both with and without GOT position measurements. The estimator is capable of inserting detected landmarks correctly into its internal map, which is verified to match with the actual location of the landmarks within just a few centimetres. When GOT measurements are lost, the FastSLAM estimator is capable of continuously generating a position estimate solely based on the RGB-D measurements of detected landmarks, with an estimate error of up to approximately 20 cm.

It has not been possible to perform tests with longer trajectories or within larger environments, and as a consequence it has not been verified how well FastSLAM performs in e.g. a long hall. Furthermore a combined test has not been carried out and the individual elements have not been tested together, and thus no conclusions can be made on the robustness of the overall systems.

Based on the results it has not been possible to verify whether or not the proposed solution is able to increase the safety of indoor drones as a combined test still has to be carried out to verify the robustness. On the other hand the individual tests results presented within this project illustrate how a FastSLAM-based RGB-D camera solution with static markers in the environment and with a GOT positioning system is indeed able to estimate the position of a drone, such that the position can either be held still or the drone can continue its motion, even at loss of GOT measurements. With only few new landmarks discovered while GOT measurements are lost, the position estimate does not seem to drift. A combination of the FastSLAM 2.0 algorithm and an Full-State Extended Kalman filter providing motion model inputs to FastSLAM seems like a computationally efficient choice.

It is deemed likely that a camera based solution can help to increase safety of drones flying in factory halls where local position measurements can be lost.

10 Discussion

As stated in Chapter 9 the full system has not been tested as a whole, and only the sub modules have been verified to work. The full suggested system introduces multiple loops, which could possible lead to instability. Therefore a test, a simulation, or a thorough analysis of the full combined system has to be performed before any conclusion about the performance of the full system can be made.

As stated in Chapter 6 the position controllers developed in this project where not designed to follow a given trajectory, but only reach certain way-points. This results in controllers that do not necessarily make the drone fly in straight lines between the way-points. This is possible due to the controllers not controlling the ratio between the velocities of the x, y and z coordinates. This limits the usability of the system since certain safety precautions have to be taken when designing the way-points, to make sure that the controllers do not make the drone fly into obstacles. Since the focus of this project is to ensure safety when drones fly in an factory environment, where humans could potentially be present, it is deemed as a necessity to be able to define strict bounds on the drones deviation from the given path, such that this can be taken into account in the planning of the way-points. No analysis has been made to investigate if such strict bounds exist for the developed controllers. Thus, some further analysis of the controllers has to be made before these can be used in a critical environment. On the other hand, if this analysis of the developed controllers shows that the bound is too high for the system to be really useful, or if no bound exists, then new controllers have to be developed. A suggested improvement of the control system is a system which penalise deviation from the planned trajectory and not just tracking references.

In the current implementation of the FastSLAM algorithm the roll and pitch values needed in the RGB-D measurement model coming from the EKF are used as deterministic values. Thus the FastSLAM algorithm does not take the probabilistic properties of these random variables into account, although the EKF is estimating the covariance of these random variables. This could possible degrade the performance of the FastSLAM algorithm. Therefore, it should be considered if the probabilistic properties of these variables should be included in the FastSLAM algorithm.

Bibliography

[1] Chris Jeppesen, David Romanos, Joan Calvet, Malte Damgaard, and Thomas Jespersen. Github repository with relevant materials. https://github.com/davidromanos/intel_aero_rtf_gr871/tree/master.

[2] A. et al. Bachrach. "range - robust autonomous navigation in gps-denied environments.". *Robotics and Automation (ICRA), 2010 IEEE International Conference*.

[3] Uaworld project. http://www.uaworld.dk/.

[4] UAWorld. Indoor autonomous flight in industrial applications: Sw and solution components - status. 2017.

[5] GamesOnTrack. Uaworld autonomous flight based on gt-drone indoor positioning. https://www.youtube.com/watch?v=cFlSfX234HM.

[6] Games on track, . http://www.gamesontrack.com/pages/webside.asp?articleGuid=164202.

[7] Mikael Juhl Kristensen Nikolaj Holm Brian Gasberg Thomsen, Jens Nielsen and Nikolaj Horsevad Sørensen. Sensor fault tolerant quadrotor flight in gps denied environments, 2015.

[8] Yohanes Khosiawan, Izabela Nielsen, Ngoc Anh Dung Do, and Bernardo Nugroho Yahya. *Concept of Indoor 3D-Route UAV Scheduling System*, pages 29–40. Springer International Publishing, Cham, 2016. ISBN 978-3-319-28555-9. doi: 10.1007/978-3-319-28555-9_3.

[9] Games on track satellites, . http://www.gamesontrack.com/pages/visnyhed.asp?newsguid=133168.

[10] Anders la Cour-Harbo John Josef Leth Henrik Schiøler, Luminita Totu. Easy 3d mapping for indoor navigation of micro uavs. 2016.

[11] Visual slam history review. http://fzheng.me/2016/05/30/slam-review/.

[12] Andrew J. Davison Javier Civera and José María Martínez Montiel. *Structure from Motion Using the Extended Kalman Filter*. 2012.

[13] Michael Montemerlo. Fastslam: A factored solution to the simultaneous localization and mapping problem with unknown data association, 2003. https://www.cs.cmu.edu/~mmde/mmdeijcai2003.pdf.

[14] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. 2003. https://www.cs.cmu.edu/~mmde/mmdeijcai2003.pdf.

[15] Aruco opencv library documentation. http://docs.opencv.org/3.1.0/d5/dae/tutorial_aruco_detection.html.

[16] Intel®. Intel® aero ready to fly drone. https://software.intel.com/en-us/aero/drone-dev-kit.

[17] Px4 pro autopilot software, 2017.

[18] Px4 attitude estimator source code. https://github.com/ArduPilot/PX4Firmware/tree/master/src/modules/attitude_estimator_q.

[19] Yocto project. https://www.yoctoproject.org/.

[20] Mohamed Abouzahir, Abdelhafid Elouardi, Samir Bouaziz, Rachid Latif, and Abdelouahed Tajer. Fastslam 2.0 running on a low-cost embedded architecture. *2014 13th International Conference on Control Automation Robotics & Vision (ICARCV)*, pages 1421–1426, 2014. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7064524&tag=1.

[21] Px4 attitude control firmware. https://github.com/PX4/Firmware/tree/4453e4201b7a245cff52beeb38a293161aea4c48/src/modules/mc_att_control.

[22] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam: A factored solution to the simultaneous localization and mapping problem. 2002. http://www.cs.cmu.edu/~mmde/mmdeaaai2002.pdf.

[23] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. 2008.

[24] Librealsense intrinsics description, . https://github.com/IntelRealSense/librealsense/blob/master/doc/projection.md/#intrinsic-camera-parameters.

[25] John J. Craig. *Introduction to Robotics: Mechanics and Control*. 2013.

[26] Udacity. Resampling wheel - artificial intelligence for robotics, 2012. https://www.youtube.com/watch?v=wNQVo6uOgYA.
[27] Udacity. Resampling wheel solution - artificial intelligence for robotics, 2012. https://www.youtube.com/watch?v=aHLslaWO-AQ.
[28] Steven M. Kay. *Intuituve Probability and Random Processes Using Matlab*. Springer, 2006. ISBN 9780387241579.
[29] Bradley M. Bell. Normal random matrix source code. http://moby.ihme.washington.edu/bradbell/mat2cpp/randn.cpp.xml.
[30] Dan Simon. *Optimal State Estimation - Kalman, H_∞, and Nonlineare Approaches*. Wiley, 2006. ISBN 9780471708582.
[31] Teppo Luukkonen. Modelling and control of quadcopter. 2011.
[32] Rogelio Lozano Claude Pégard Luis Rodolfo García Carrillo, Alejandro Enrique Dzul López. *Quad Rotorcraft Control*. Springer, 2013. ISBN 978-1-4471-4399-4.
[33] Jung Soon Jang S.L. Waslander, G.M. Hoffman. Multi-agent quadrotor testbed control design: Integral sliding mode vs. reinforcement learning. *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1–6, 2005.
[34] Trilateration with gps satellites. http://gisgeography.com/trilateration-triangulation-gps/.
[35] Yohanes Khosiawan and Izabella Nielsen. A system of uav application in indoor environment. 2016.
[36] Games on track accuracy, . http://www.gamesontrack.com/pages/webside.asp?articleGuid=164202.
[37] Zeyneb Kurt-Yavuz and Sırma Yavuz. A comparison of ekf, ukf, fastslam2.0, and ukf-based fastslam algorithms. 2012.
[38] Nicholas D. Molton Andrew J. Davison, Ian D. Reid and Olivier Stasse. Monoslam: Real-time single camera slam. 2007.
[39] Kohta Ishikawa. Particle filter example in python, 2011.
[40] Thomas Schön. (rao-blackwellized) particle filters and localization.
[41] Wikus Brink. Stereo vision for simultaneous localization and mapping, 2012.
[42] Robin Lindholm and Carl-Johan Pålsson. Simultaneous localization and mapping for vehicle localization using lidar sensors, 2015.
[43] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. 2007.
[44] Steven J. Lovegrove Richard A. Newcombe and Andrew J. Davison. Dtam: Dense tracking and mapping in real-time. 2011.
[45] J. M. M. Montiel Raul Mur-Artal and Juan D. Tardos. Orb-slam: a versatile and accurate monocular slam system. 2015.
[46] Fredrik Hjelmare and Jonas Rangsjö. Simultaneous localization and mapping using a kinect™ in a sparse feature indoor environment, 2012.
[47] Andrew J. Davison Hauke Strasdat, J.M.M. Montiel. Visual slam: Why filter? 2012.
[48] Intel realsense r200 rgb-d usb 3.0 camera. https://software.intel.com/en-us/realsense/r200camera.
[49] Anders Grunnet-Jepsen Leonid Keselman, John Iselin Woodfill and Achintya Bhowmik. Intel realsense stereoscopic depth cameras. 2017.
[50] Intel realsense r200 camera layout, . https://software.intel.com/en-us/articles/realsense-r200-camera.
[51] Jacob Dueholm. 3d vision.
[52] Intel®. Intel® realsense™ camera r200, 2016.
[53] Librealsense supported camera formats. https://github.com/IntelRealSense/librealsense/blob/master/doc/supported_video_formats.pdf.
[54] Librealsense specifications for the r200 camera, . https://github.com/IntelRealSense/librealsense/blob/master/doc/projection.md/#r200.
[55] Librealsense distortion model parameters, . https://github.com/IntelRealSense/librealsense/blob/master/doc/projection.md/#distortion-models.
[56] Point cloud example, . https://github.com/IntelRealSense/librealsense/blob/master/examples/c-tutorial-3-pointcloud.c.
[57] Intel realsense r200 rgb-d usb 3.0 camera developer guide, . https://software.intel.com/sites/products/realsense/camera/developer_guide.html.

[58] Librealsense library, . http://wiki.ros.org/librealsense.
[59] Image transport ros protocol. http://wiki.ros.org/image_transport.
[60] Ros wiki: Rosbag. http://wiki.ros.org/rosbag.
[61] Ros wiki: rqt image view. http://wiki.ros.org/rqt_image_view.
[62] R200 depth issue, . https://github.com/intel-ros/realsense/issues/233.
[63] Peter Corke. *Robotics, Vision and Control - Fundamental Algorithms in MATLAB*. 2011.
[64] Jacob Dueholm. Registration and camera calibration. 2017.
[65] Mark Fiala. Desingning highly reliable fiducial markers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(7):1317 – 1324, July 2009.
[66] F. J. Madrid-Cuevas S. Garrido-Jurado, R. Muñoz-Salinas and M. J. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recogn. 47, 6 (June 2014)*, 2014.
[67] Quick response (qr) code. http://www.qrcode.com/en/.
[68] Leonid Naimark and Eric Foxlin. Circular data matrix fiducial system and robust image processing for a wearable vision-inertial self-tracker. 2002.
[69] M. Billinghurst H. Kato. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. *Proceedings of the 2nd IEEE and ACM InternationalWorkshop on Augmented Reality*, pages 85–, 1999.
[70] D. Schmalstieg D. Wagner. Artoolkitplus for pose tracking on mobile devices. *Computer Vision Winter Workshop*, pages 139 – 146, 2007.
[71] Richard w. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 1950.
[72] Ros wiki. https://www.wiki.ros.org.
[73] Thomas Gubler. Simulating px4 flight control with ros/gazebo.
[74] Simulation description format. www.sdformat.org.
[75] Meet kinect for windows. https://developer.microsoft.com/en-us/windows/kinect.

A Modelling

A.1 First principle model

This section is intended to show an approach to the modelling of a quadrotor. To derive the dynamic equations of the system, the Euler-Lagrange approach will be used.
Generally, there are two different configurations used for defining the position and orientation to model quadcopters. One approach is to place opposite rotors along the same axis, also described as plus configuration. The other approach which is the one used in the following modelling is to place the axes in the same direction as the sides of the drone, described as X configuration. The two approaches are shown in Figure A.1.

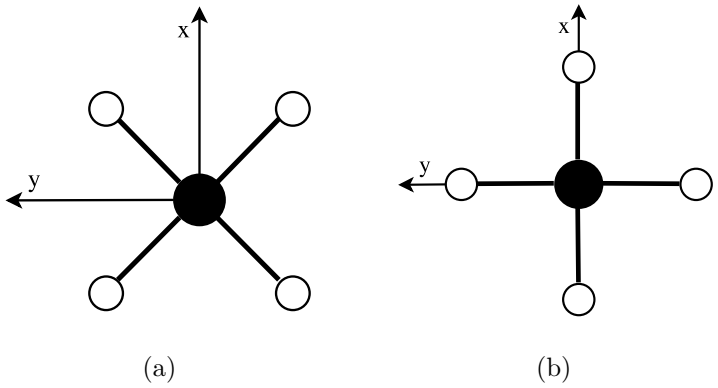


Figure A.1: X (Figure A.1(a)) and plus (Figure A.1(b)) configurations of the drone.

Two different coordinate frames are considered: one embedded in the center of mass of the drone defined as the body frame (BF), and another one fixed on earth defined as earth frame (EF). The axes of the BF are aligned with the drone's principal moments of inertia, and their direction is defined with the x axis pointing in the forward direction of the drone, and the z axis is pointing upwards. The description of the drone is shown in Figure A.2.

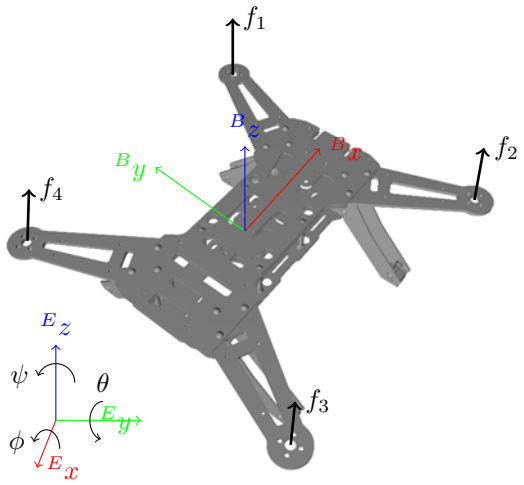


Figure A.2: Drone description. The different frames and rotations considered for the modelling are shown as earth frame (EF) and body frame (BF). The rotors of the drone are numbered from 1 to 4.

First of all, the variables of the system is defined. The position $\boldsymbol{\xi}$ and orientation $\boldsymbol{\eta}$ of the quadrotor are defined in the EF as

$$^{\mathrm{E}}\boldsymbol{\xi} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \boldsymbol{\eta} = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \tag{A.1}$$

Where the different axis x, y, and z are defined as shown in Figure A.2 and ϕ, θ and ψ are the angles defining the rotations around each of the EF defined axes respectively; defined as roll, pitch and yaw.
The angular velocities in the BF are defined as follows, with each component of the vector corresponding to the rotation around the axis in the BF.

$$\boldsymbol{\nu} = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{A.2}$$

The rotation matrix changing the coordinates from the BF to the EF has to be defined. It can be generated by multiplying the rotation matrices around each of the axis with a defined order. These are defined as follows.

$$\mathbf{R}_\phi = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \quad \mathbf{R}_\theta = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \quad \mathbf{R}_\psi = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{A.3}$$

By using the matrices defined in (A.3), the rotation matrix $\mathbf{R}_\psi\mathbf{R}_\theta\mathbf{R}_\phi = \mathbf{R}$ defined with the sequence of rotations ϕ, θ, ψ is computed. Using fixed angles, this sequence is achieved by rotating the EF by roll around the earth x-axis, followed by a pitch rotation around the earth y-axis and finally a rotation around the earth z-axis of yaw.

$$^{\mathrm{E}}_{\mathrm{B}}\mathbf{R} = \begin{bmatrix} c_\psi c_\theta & c_\psi s_\theta s_\phi - s_\psi c_\phi & c_\psi s_\theta c_\phi + s_\psi s_\phi \\ s_\psi c_\theta & s_\psi s_\theta s_\phi + c_\psi c_\phi & s_\psi s_\theta c_\phi - c_\psi s_\phi \\ -s_\theta & c_\theta s_\phi & c_\theta c_\phi \end{bmatrix} \tag{A.4}$$

Where c_α denotes $\cos\alpha$ and s_α denotes $\sin\alpha$.
The transformation matrix from the orientation velocities in the EF to the angular velocities in the BF has to be found. This is done by considering the rotation around each axis with a specific order while considering small changes in each angle. It is chosen to consider the same sequence used to find the rotation matrix. Therefore, the matrices used for transforming from BF to EF have to be inverted; which due to the orthogonality in the rotation matrices is the same as transposing.

$$\boldsymbol{\nu} = {\mathbf{R}_\phi}^T {\mathbf{R}_\theta}^T \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} + {\mathbf{R}_\phi}^T \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} \tag{A.5}$$

This way, $\mathbf{W}(\boldsymbol{\eta})$ defined as $\boldsymbol{\nu} = \mathbf{W}(\boldsymbol{\eta})\dot{\boldsymbol{\eta}}$ is found.

$$\mathbf{W}(\boldsymbol{\eta}) = \begin{bmatrix} 1 & 0 & -s_\theta \\ 0 & c_\phi & c_\theta s_\phi \\ 0 & -s_\phi & c_\theta c_\phi \end{bmatrix} \tag{A.6}$$

It is assumed that the quadrotor has a symmetric structure as shown in Figure A.2. This leaves the inertia of the drone defined as

$$\boldsymbol{I} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{A.7}$$

To compute the Euler-Lagrange equations, the external forces and torques of the system need to be defined. The definition of these is based on the derivation of a quadcopter model shown in [31] and [32]. Each rotor with rotational speed ω_i creates a force in the direction of the rotor axis, creating a resultant thrust defined in the BF as

$${}^{\mathrm{B}}\boldsymbol{T} = \begin{bmatrix} 0 \\ 0 \\ T \end{bmatrix}$$

$$T = \sum_{i=1}^{4} f_i = k \sum_{i=1}^{4} {\omega_i}^2 \tag{A.8}$$

Where:

T	is the sum of each thrust force from the rotors	N
ω_i	is the rotational speed of the i_{th} rotor	rad/s
k	is the lift constant	kg

At the same time, the rotors create a torque around different axis direction. The resultant torque on each axis in the BF can be described as

$${}^{\mathrm{B}}\boldsymbol{\tau} = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} \tag{A.9}$$

generating each of the movements roll, pitch and yaw.
The torque creating the roll movement, τ_ϕ, can be generated by increasing the speed of two rotors placed in the same side of the y axis and decreasing the other two.

$$\tau_\phi = lk(-{\omega_1}^2 - {\omega_2}^2 + {\omega_3}^2 + {\omega_4}^2) \tag{A.10}$$

The torque creating the pitch movement, τ_θ, can be generated by increasing the speed of two rotors placed in the same side of the x axis and decreasing the other two.

$$\tau_\theta = lk(-{\omega_1}^2 - {\omega_3}^2 + {\omega_2}^2 + {\omega_4}^2) \tag{A.11}$$

The torque creating the yaw movement, τ_ψ, can be generated by increasing the speed of two opposite rotors and decreasing the speed in the other two. This torque is a result of the combination between the moment of inertia and the drag of the each motor $\tau_{M_i} = b\omega_i^2 + I_M\dot{\omega}_i$. If the effect of the rotor acceleration is neglected, the resulting yaw torque is defined as follows.

$$\tau_\psi = b(-{\omega_1}^2 - {\omega_4}^2 + {\omega_2}^2 + {\omega_3}^2) \tag{A.12}$$

The parameter l describes the distance between the rotors and the center of mass, and b is the drag constant.

Now, the Euler-Lagrange equations can be used. First of all, the lagrangian is defined as the kinetic energy of the system minus the potential energy.

$$\mathcal{L} = E_{\text{kinetic}} - E_{\text{potential}} = E_{\text{trans}} + E_{\text{rot}} - E_{\text{potential}} \tag{A.13}$$

With E_{trans} being the translation energy of the quadrotor and E_{rot} being its rotation energy, the Lagrange expression becomes the following.

$$\mathcal{L} = \frac{1}{2}m\dot{\boldsymbol{\xi}}^T\dot{\boldsymbol{\xi}} + \frac{1}{2}\boldsymbol{\nu}^T\boldsymbol{I}\boldsymbol{\nu} - mg\begin{bmatrix}0\\0\\z\end{bmatrix} \tag{A.14}$$

The Euler-Lagrange expression,

$$\begin{bmatrix}\boldsymbol{f}\\\boldsymbol{\tau}\end{bmatrix} = \frac{d}{dt}\left(\frac{\partial\mathcal{L}}{\partial\dot{\boldsymbol{q}}}\right) - \frac{\partial\mathcal{L}}{\partial\boldsymbol{q}} \tag{A.15}$$

with $[\boldsymbol{f}\ \boldsymbol{\tau}]^T$ being the external forces and torques of the system and defining $\boldsymbol{q}$ as $[\boldsymbol{\xi}\ \ \boldsymbol{\eta}]^T$, can be separated into the linear and angular components. This leaves two systems of differential equations.

$$\boldsymbol{f} = {}^{\text{E}}_{\text{B}}\mathbf{R}\,{}^{\text{B}}\boldsymbol{T} = m\ddot{\boldsymbol{\xi}} + mg\begin{bmatrix}0\\0\\1\end{bmatrix} \tag{A.16}$$

$$\boldsymbol{\tau} = {}^{\text{B}}\boldsymbol{\tau} = \frac{1}{2}\frac{d}{dt}\left(\frac{\partial}{\partial\dot{\boldsymbol{\eta}}}\boldsymbol{\nu}^T\boldsymbol{I}\boldsymbol{\nu}\right) - \frac{1}{2}\frac{\partial}{\partial\boldsymbol{\eta}}\boldsymbol{\nu}^T\boldsymbol{I}\boldsymbol{\nu} = \frac{1}{2}\frac{d}{dt}\left(\frac{\partial}{\partial\dot{\boldsymbol{\eta}}}\left(\dot{\boldsymbol{\eta}}^T\mathbf{W}^T(\boldsymbol{\eta})\boldsymbol{I}\mathbf{W}(\boldsymbol{\eta})\dot{\boldsymbol{\eta}}\right)\right) - \frac{1}{2}\frac{\partial}{\partial\boldsymbol{\eta}}\left(\dot{\boldsymbol{\eta}}^T\mathbf{W}^T(\boldsymbol{\eta})\boldsymbol{I}\mathbf{W}(\boldsymbol{\eta})\dot{\boldsymbol{\eta}}\right) \tag{A.17}$$

By calling the transformation of the inertia matrix $\mathbf{W}^T(\boldsymbol{\eta})\boldsymbol{I}\mathbf{W}(\boldsymbol{\eta}) = \boldsymbol{J}(\boldsymbol{\eta})$, (A.17) can be rewritten into the following expression.

$$\boldsymbol{\tau} = \boldsymbol{J}(\boldsymbol{\eta})\ddot{\boldsymbol{\eta}} + \frac{d}{dt}\left(\boldsymbol{J}(\boldsymbol{\eta})\right)\dot{\boldsymbol{\eta}} - \frac{1}{2}\frac{\partial}{\partial\boldsymbol{\eta}}\left(\dot{\boldsymbol{\eta}}^T\boldsymbol{J}(\boldsymbol{\eta})\dot{\boldsymbol{\eta}}\right) = \boldsymbol{J}(\boldsymbol{\eta})\ddot{\boldsymbol{\eta}} + \boldsymbol{C}(\boldsymbol{\eta},\dot{\boldsymbol{\eta}})\dot{\boldsymbol{\eta}} \tag{A.18}$$

The term $\boldsymbol{C}(\boldsymbol{\eta},\dot{\boldsymbol{\eta}})$ is a compact rewriting of the resulting terms in the equation, which results in a matrix depending on $\boldsymbol{\eta}$ and $\dot{\boldsymbol{\eta}}$.
This two systems of differential equations describe the dynamic equations of the quadrotor considered.

A.2 Model simplification

It is deemed necessary to find a simplified linear version of the model in order to ease the control of the quadcopter. To define a simplified version of the model found, some assumptions need to be taken.
The first assumption is that the roll and pitch angles are small. When close to hovering this is a good approximation. Therefore, using the small angle approximation [33], $\boldsymbol{\nu} = \dot{\boldsymbol{\eta}}$, which leads to the relation between the torques and the angular accelerations.

$$\boldsymbol{I}\ddot{\boldsymbol{\eta}} = \boldsymbol{\tau} \tag{A.19}$$

If a frame that only rotates with the yaw movement is considered, denoted as the heading frame (HF), the dynamic equations for the x and y axes in the HF become the following.

$$m \begin{bmatrix} {}^{\text{H}}\ddot{x} \\ {}^{\text{H}}\ddot{y} \end{bmatrix} = \begin{bmatrix} S_\theta C_\phi & 0 \\ 0 & -S_\phi \end{bmatrix} \begin{bmatrix} T \\ T \end{bmatrix} \approx \begin{bmatrix} \theta & 0 \\ 0 & -\phi \end{bmatrix} \begin{bmatrix} T \\ T \end{bmatrix} \tag{A.20}$$

Where T is the thrust generated by the rotors and the small angle approximation $\cos\alpha \approx 1$ and $\sin\alpha \approx \alpha$ when $\alpha \to 0$ is used. If the movement in the z axis is considered independently, it can be seen that the dynamic equation becomes the following

$$m\ddot{z} = T - mg \tag{A.21}$$

where, in case of hovering, the thrust must be equal to the weight of the drone.
Since the drone is supposed to fly close to hovering in all cases, this equality allows to consider x and y only dependant on roll and pitch respectively, leaving the following relation.

$$\begin{bmatrix} {}^{\text{H}}\ddot{x} \\ {}^{\text{H}}\ddot{y} \end{bmatrix} = g \begin{bmatrix} \theta \\ -\phi \end{bmatrix} \tag{A.22}$$

Where roll and pitch in the HF define the rotations around x and y in the same frame.

A.3 Black-box model / System identification

The drone used for the development of the project is chosen to be the Intel® Aero Ready to Fly Drone which has already a PX4 flight controller implemented as explained in Section 4.1, it is decided to proceed with the modelling of the drone considering it as a black box. In attitude mode, this flight controller is based on proportional controllers for roll and pitch angular references and PID controllers for roll, pitch and yaw angular rates. The code of the controller implemented can be found in [21]. To proceed with the system identification, the simplified model from Section A.2 is taken into account combined with the knowledge of the flight controller.
The models used for the system identification are Auto-Regressive with Exogenous input (ARX) models. This type of models consider no dynamics on the error of the system. This allows the fit to have a unique solution that can e found by linear regression. However, this consideration will cause the possible dynamics of disturbances to be included in the system model. The ARX models are defined as shown in (A.23).

$$y^k + a_1 y^{k-1} + ... + a_{n_a} y^{k-n_a} = b_1 u^{k-n_k} + ... + b_{n_b} u^{k-n_b-n_k+1} + e^k \tag{A.23}$$

Where:

y^k	denotes the output at sample time k
n_a	is the number of poles
n_b	is the number of zeroes plus 1
n_k	is the number of input samples that occur before affecting the output (dead-time)
y^{k-i}	are the previous outputs affecting the current one, with $i \in \{1, \ldots, n_a\}$
u^{k-j}	are the previous and delayed inputs, with $j \in \{n_k, \ldots, n_k + n_b - 1\}$
e^k	is a white noise disturbance

This can be rewritten into the matrix form shown in (A.24).

$$A(q)y^k = B(q)u^{k-n_k} + e^k \tag{A.24}$$

Where:

$A(q)$	$= 1 + a_1 q^{-1} + ... + a_{n_a} q^{-na}$
$B(q)$	$= b_1 + b_2 q^{-1} + ... + b_{n_b} q^{-n_b+1}$
q	is the forward shift operator

To proceed with the identification, measurements of the drone's position and orientation are taken from a Vicon system and together with the extracted reference inputs of the drone's remote controller (RC) they are used to fit ARX models to the desired outputs.

A.3.1 Roll and pitch ARX models

Since the controllers implemented for roll and pitch have the same structure and the modelling of the drone shows that they behave the same way, the models for roll and pitch are decided to be of the same order. The block diagram for roll and pitch given angle references from the RC is described in Figure A.3.

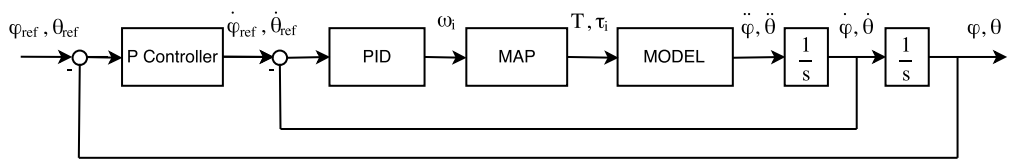


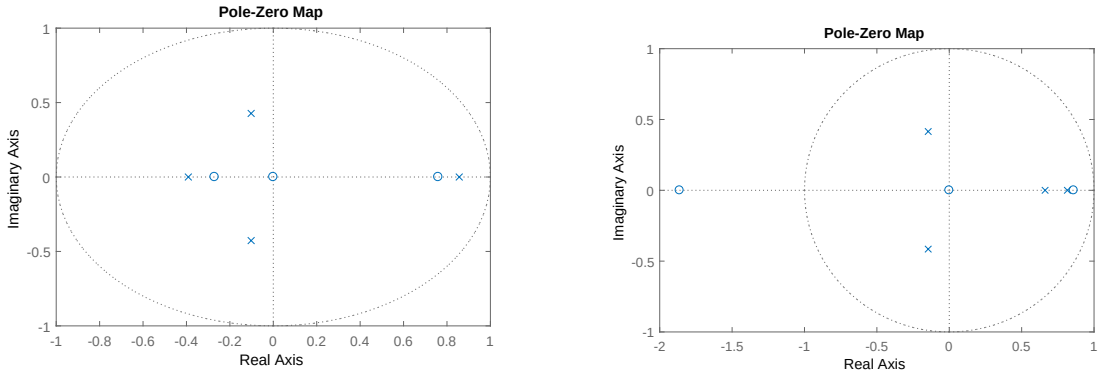
Figure A.3: Roll and pitch block diagram. The MAP block describes the transformation from rotor speeds to thrust and torque inputs.

By inspecting the orders of the system shown in Figure A.3, the first try of fitting the data is performed with $n_a = 3$ and $n_b = 2$ which does not result in a good fitting. Based on the model, it is decided to fit models for roll and pitch with the same order. When the orders of the system are increased to be $n_a = 4$ and $n_b = 3$, the model shows a better fit to the measurements. This gives the corresponding description of the system describing roll and pitch, plotted in Figure A.5.

$$\phi(z) = G_\phi(z)\phi_{ref}(z) = \frac{0.8608z^{-1} - 0.4211z^{-2} - 0.177z^{-3}}{1 - 0.261z^{-1} - 0.2374z^{-2} - 0.1581z^{-3} - 0.06463z^{-4}}\phi_{ref}(z) \quad (A.25)$$

$$\theta(z) = G_\theta(z)\theta_{ref}(z) = \frac{0.1938z^{-1} + 0.1944z^{-2} - 0.3103z^{-3}}{1 - 1.187z^{-1} + 0.3037z^{-2} - 0.1289z^{-3} + 0.1045z^{-4}}\theta_{ref}(z) \quad (A.26)$$

The poles of both models are found to be inside the unit circle, and thus stable. The poles and zeros of the models found for roll and pitch are shown in Figure A.4.



(a) (b)

Figure A.4: Roll Figure A.4(a) and pitch Figure A.4(b) poles and zeros plot. The pole locations of the models indicate a stable system.

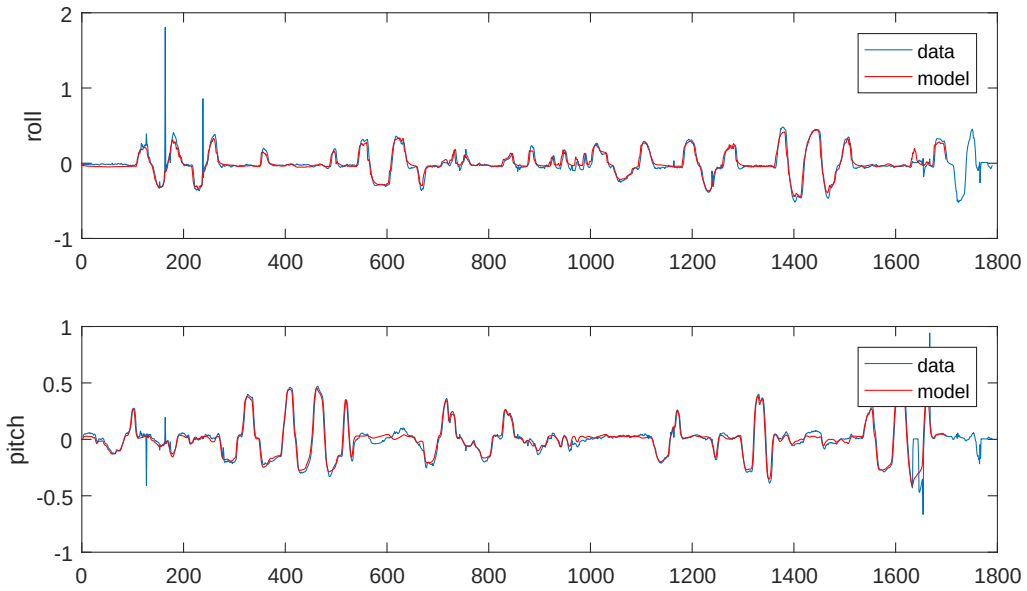


Figure A.5: Roll and pitch fitted models. Data in blue is measurements used for fitting and data in red is the resulting fitting

Data extracted from a different experiment is compared to the output of the models found with the corresponding inputs in order to validate them, shown in Figure A.6.

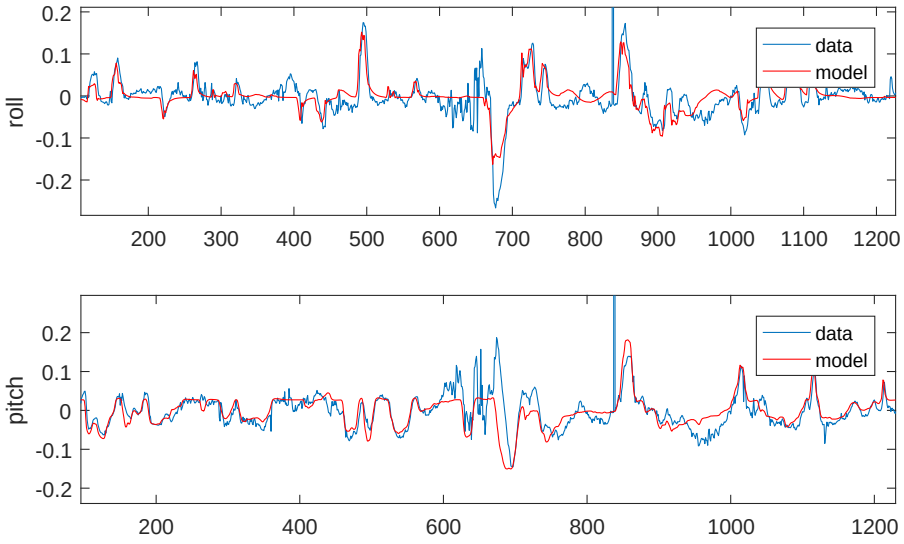


Figure A.6: Roll and pitch models validation. Data in blue is measurements used from a different experiment and data in red is the result of using the inputs of the experiment with the found models

Taking the drone model described in (A.22) into account, it can be seen how $\ddot{x}$ and $\ddot{y}$ are directly related to roll and pitch, meaning that in order to find a model for x and y only two integrators are needed from the angles.

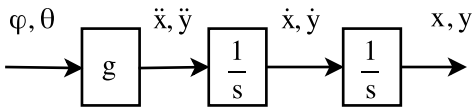


Figure A.7: x and y block diagram derived from the simplified model.

For the validation of x and y, the measurements from the Vicon set-up are differentiated twice, giving the accelerations in each direction. These values found are compared to the roll and pitch models found previously multiplied times gravity as described in the model. The result is shown in Figure A.8.

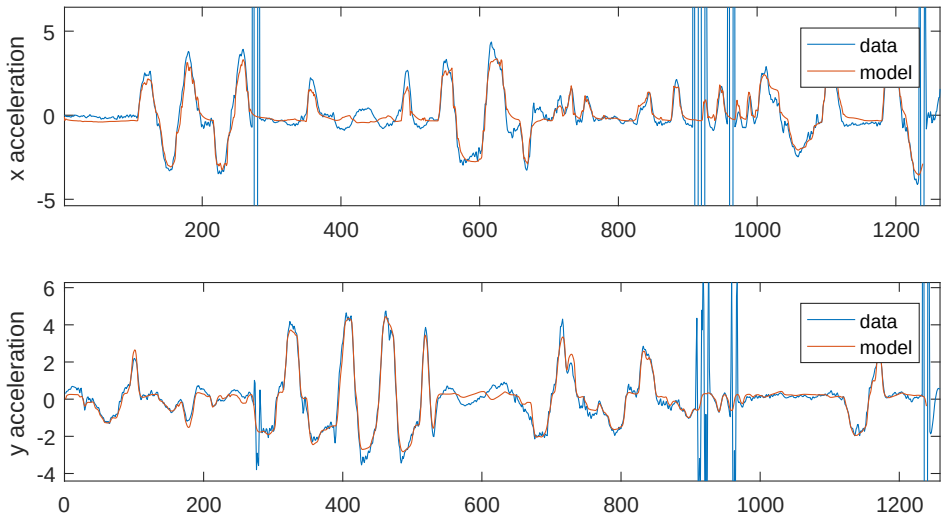


Figure A.8: x and y acceleration plots. Data in blue is the second derivative of the measurements from Vicon not used for fitting roll and pitch and data in red is the result of multiplying the models found simulated with the inputs from the experiment times gravity Figure A.8

A.3.2 Yaw ARX model

The input for yaw in the RC is an angular rate reference, which can not be used as an input when designing a position controller. Therefore, it is decided not to fit a model for yaw having the RC as input. Since the drone has a position controller implemented, a test for finding a model for yaw with a usable input can be designed by holding the drone in a specific position and generating an input signal of yaw reference. The model found for yaw is then from yaw reference to yaw, which describes the closed loop of a yaw controller.

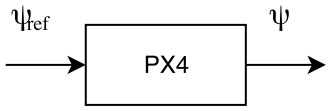


Figure A.9: Yaw block diagram.

In the same way as in the models for roll and pitch, a portion of the data is fitted to a model and this one is afterwards verified with a different set of measurements.
The orders considered for the ARX model from yaw references to yaw velocity are $na = 1$ and $nb = 1$ since the dynamics are shown to be close to a first order system.

$$\dot{\psi}(z) = G_{\psi}(z)\psi_{ref}(z) = \frac{0.06215z^{-1}}{1 - 0.9383z^{-1}}\psi_{ref}(z) \tag{A.27}$$

In Figure A.10 the data extracted from the experiment is compared to a linear simulation of the model found with the yaw reference inputs given during the flight.

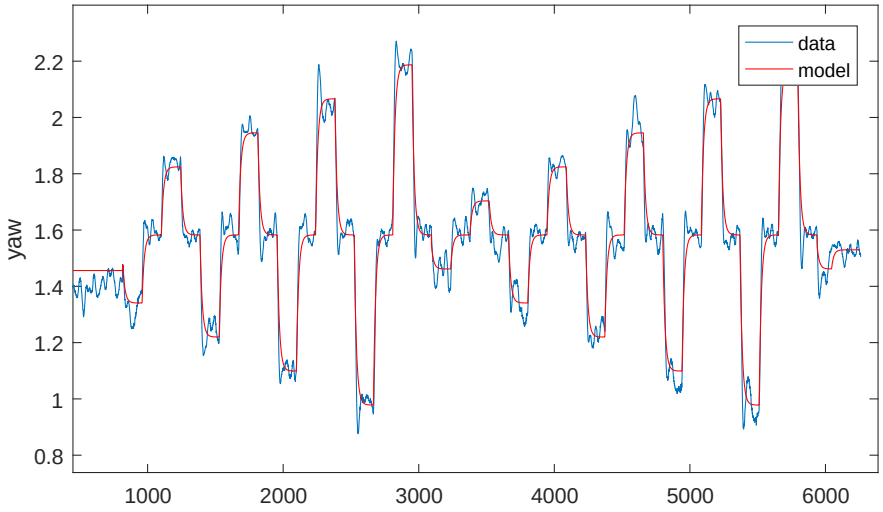


Figure A.10: Yaw plot. Data in blue are the measurements from Vicon used for fitting the model and data in red is the result of simulating the model found with the inputs of the experiment

A.3.3 Z ARX model

The input reference from the RC for thrust is scaled from 0 to 1. This value is related to $\ddot{z}$ as explained in the modelling. Therefore, with a model from input to $\ddot{z}$, two integrators are added to reach z.
To be able to fit the thrust input reference to the z acceleration linearly, it has to be taken into account that the drone needs to have a certain thrust value (operating point) to overcome the effect of gravity and thus take off and hover.
Consequently, acceleration values taken when the drone is in zero height won't relate linearly to the input. This way, only data from when the drone is flying is used for the fitting. First, the values of $\ddot{z}$ are found y differentiating twice the measurements of z. Afterwards, following the model found in Section A.2, gravity is added to these values. Then, a model for $\ddot{z} + g$ is found with the thrust value as input.

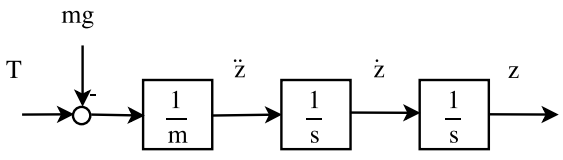


Figure A.11: z block diagram derived from the simplified model.

It has to be taken into account that the input thrust to the drone is scaled from 0 to 1 and therefore, the gain from thrust to acceleration won't match with the values expected from the model.

$$\ddot{z}(\mathrm{z}) = 16.71\mathrm{z}^{-1}T_{\mathrm{ref}}(\mathrm{z}) - g \tag{A.28}$$

Now, for control purposes, it is deemed to have an affine model,i.e. with zero input matching with zero output. To do so, the thrust input value B matching with gravity has to be found as explained in (A.29).

$$\begin{aligned} \ddot{z}(\mathrm{z}) &= 16.71\mathrm{z}^{-1}(T^*_{\mathrm{ref}}(\mathrm{z}) + B) - g \\ 0 &= 16.71(0 + B) - g \\ g &= 16.71B \end{aligned} \tag{A.29}$$

This leads to the model shown in (A.30), with $T^*_{\mathrm{ref}}(\mathrm{z}) = T_{\mathrm{ref}}(\mathrm{z}) - B$.

$$\ddot{z}(\mathrm{z}) = G_{\ddot{z}}(\mathrm{z})T^*_{\mathrm{ref}}(\mathrm{z}) = 16.71\mathrm{z}^{-1}T^*_{\mathrm{ref}}(\mathrm{z}) \tag{A.30}$$

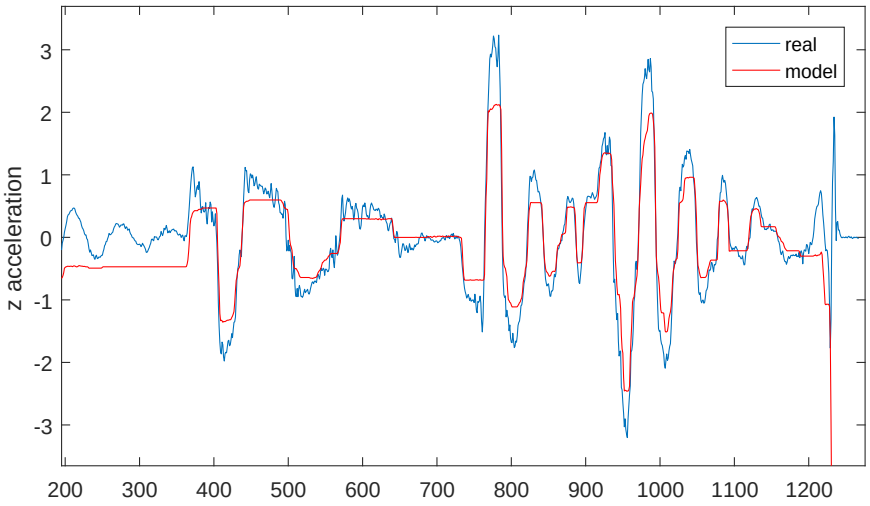


Figure A.12: z acceleration plot. Data in blue is the second derivative of the measurements from Vicon used for fitting the model and data in red is the result of simulating the model found with the inputs of the experiment. The model at the start of the experiment predicts an acceleration of the value of gravity not show in the data due to the drone being on the floor.

To validate the model found, data from a different experiment is used with the model found and the corresponding inputs. To compare the data with the model, the values of $\ddot{z}$ of the new experiment are compared to the output of the model subtracting gravity to it.

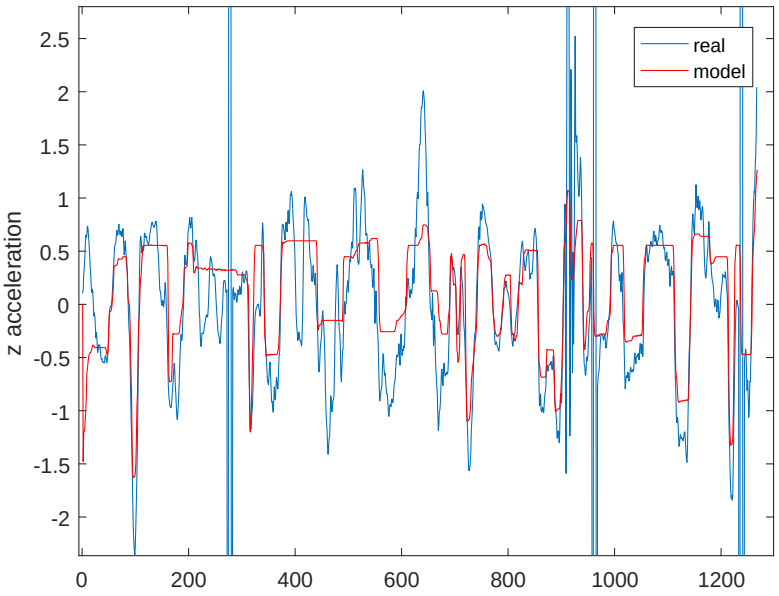


Figure A.13: z acceleration plot. Data in blue is the second derivative of the measurements from Vicon not used for fitting the model and data in red is the result of simulating the model found with the inputs of the experiment. The model at the start of the experiment predicts an acceleration of the value of gravity not show in the data due to the drone being on the floor.

A.4 State-space description

It is decided to use state-space controllers designed according to the LQR/G paradigm for the position controllers. Therefore, the models found in Section A.3 have to be transformed into a state-space representation. Since the transfer functions of the variables are known, the polynomials describing the poles and zeros of the each system are also known. This knowledge can be used for expressing a state-space description in controllable canonical form.
Having a transfer function described as in (A.31), a state-space representation in controllable canonical form can be found as described in (A.32).

$$Y(\mathrm{z}) = \frac{b_1 \mathrm{z}^{-1} + \cdots + b_{n_b} \mathrm{z}^{-n_b}}{1 + a_1 \mathrm{z}^{-1} + \cdots + a_{n_a} \mathrm{z}^{-n_a}} U(\mathrm{z}) \tag{A.31}$$

$$\begin{aligned} \boldsymbol{\chi}^{k+1} &= \begin{bmatrix} -a_1 & -a_2 & \cdots & -a_{n_a-1} & -a_{n_a} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \boldsymbol{\chi}^k + \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} u^k \\ y^k &= \begin{bmatrix} b_1 & b_2 & \cdots & b_{n_b} & 0_{1\times(n_a-n_b)} \end{bmatrix} \boldsymbol{\chi}^k \end{aligned} \tag{A.32}$$

A state-space representation is found for the ARX models of roll, pitch, yaw and $\ddot{z}$, and they are consequently expanded with the desired integrators. The state-space representation of roll is shown in (A.33), pitch is shown in (A.34), yaw in (A.35), and $\ddot{z}$ is shown in (A.36). These are then used to find a state-space description of the full system shown in (6.6).

$$\begin{aligned}\boldsymbol{\chi_\phi}^{k+1} = \begin{bmatrix}\chi_{\phi,1}\\ \chi_{\phi,2}\\ \chi_{\phi,3}\\ \chi_{\phi,4}\end{bmatrix}^{k+1} &= \boldsymbol{A}_\phi \boldsymbol{\chi_\phi}^k + \boldsymbol{B}_\phi \phi_{\text{ref}}^k = \begin{bmatrix}0.261 & 0.2374 & 0.1581 & 0.06463\\ 1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\end{bmatrix}\begin{bmatrix}\chi_{\phi,1}\\ \chi_{\phi,2}\\ \chi_{\phi,3}\\ \chi_{\phi,4}\end{bmatrix}^k + \begin{bmatrix}1\\0\\0\\0\end{bmatrix}\phi_{\text{ref}}^k\\ \phi^k = \boldsymbol{C}_\phi \boldsymbol{\chi}_\phi^k &= \begin{bmatrix}0.8608 & -0.4211 & -0.177 & 0\end{bmatrix}\boldsymbol{\chi}_\phi^k\end{aligned} \tag{A.33}$$

$$\begin{aligned}\boldsymbol{\chi_\theta}^{k+1} = \begin{bmatrix}\chi_{\theta,1}\\ \chi_{\theta,2}\\ \chi_{\theta,3}\\ \chi_{\theta,4}\end{bmatrix}^{k+1} &= \boldsymbol{A}_\theta \boldsymbol{\chi_\theta}^k + \boldsymbol{B}_\theta \theta_{\text{ref}}^k = \begin{bmatrix}1.187 & -0.3037 & 0.1289 & -0.1045\\ 1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\end{bmatrix}\begin{bmatrix}\chi_{\theta,1}\\ \chi_{\theta,2}\\ \chi_{\theta,3}\\ \chi_{\theta,4}\end{bmatrix}^k + \begin{bmatrix}1\\0\\0\\0\end{bmatrix}\theta_{\text{ref}}^k\\ \theta^k = \boldsymbol{C}_\phi \boldsymbol{\chi}_\theta^k &= \begin{bmatrix}0.1938 & 0.1944 & -0.3103 & 0\end{bmatrix}\boldsymbol{\chi}_\theta^k\end{aligned} \tag{A.34}$$

$$\psi^{k+1} = A_\psi \psi^k + B_\psi \psi_{\text{ref}}^k = 0.944\psi^k + 0.05639\psi_{\text{ref}}^k \tag{A.35}$$

$$\begin{aligned}\chi_{\ddot{z}}^{k+1} &= A_{\ddot{z}}\chi_{\ddot{z}}^k + B_{\ddot{z}}T^k = T^k\\ \ddot{z}^k &= C_{\ddot{z}}\chi_{\ddot{z}}^k = 16.71\chi_{\ddot{z}}^k\end{aligned} \tag{A.36}$$

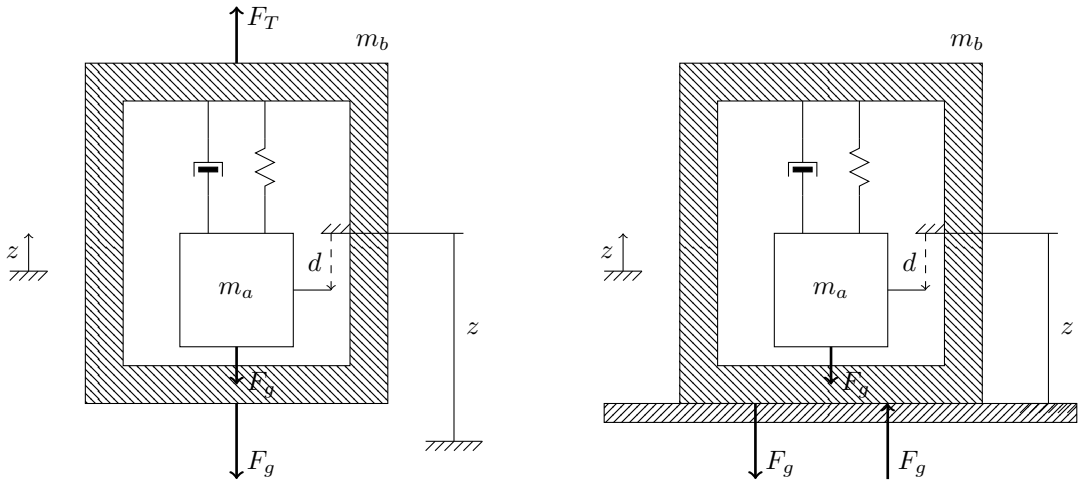
$$
\begin{bmatrix} {}^{\mathrm{H}}x \\ {}^{\mathrm{H}}\dot{x} \\ \boldsymbol{\chi}_\theta \\ \hline {}^{\mathrm{H}}y \\ {}^{\mathrm{H}}\dot{y} \\ \boldsymbol{\chi}_\phi \\ \hline \psi \\ \hline z \\ \dot{z} \\ \chi_{\ddot{z}} \end{bmatrix}^{k+1}
=
\left[\begin{array}{ccc|cc|c|c|ccc}
1 & T_s & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 0 & 0 \\
0 & 1 & gT_s\boldsymbol{C}_\theta & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 0 & 0 \\
0 & 0 & \boldsymbol{A}_\theta & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & \mathbf{0}_{1\times4} & 1 & T_s & \mathbf{0}_{1\times4} & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{0}_{1\times4} & 0 & 1 & -gT_s\boldsymbol{C}_\phi & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \boldsymbol{A}_\phi & 0 & 0 & 0 & 0 \\ \hline
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & A_\psi & 0 & 0 & 0 \\ \hline
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 1 & T_s & 0 \\
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 1 & T_sC_{\ddot{z}} \\
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 0 & A_{\ddot{z}}
\end{array}\right]
\begin{bmatrix} {}^{\mathrm{H}}x \\ {}^{\mathrm{H}}\dot{x} \\ \boldsymbol{\chi}_\theta \\ \hline {}^{\mathrm{H}}y \\ {}^{\mathrm{H}}\dot{y} \\ \boldsymbol{\chi}_\phi \\ \hline \psi \\ \hline z \\ \dot{z} \\ \chi_{\ddot{z}} \end{bmatrix}^{k}
+
$$

$$
+
\left[\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & \boldsymbol{B}_\theta & 0 & 0 \\ \hline
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\boldsymbol{B}_\phi & 0 & 0 & 0 \\ \hline
0 & 0 & B_\psi & 0 \\ \hline
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & B_{\ddot{z}}
\end{array}\right]
\begin{bmatrix} \phi_{\mathrm{ref}} \\ \theta_{\mathrm{ref}} \\ \psi_{\mathrm{ref}} \\ T \end{bmatrix}^{k}
\tag{A.37}
$$

$$
\begin{bmatrix} {}^{\mathrm{H}}x \\ {}^{\mathrm{H}}y \\ z \\ \phi \\ \theta \\ \psi \end{bmatrix}^{k}
=
\left[\begin{array}{ccc|cc|c|ccc}
1 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{0}_{1\times4} & 1 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 1 & 0 & 0 \\
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \boldsymbol{C}_\phi & 0 & 0 & 0 & 0 \\
0 & 0 & \boldsymbol{C}_\theta & 0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & 0 & 0 \\
0 & 0 & \mathbf{0}_{1\times4} & 0 & 0 & \mathbf{0}_{1\times4} & C_\psi & 0 & 0 & 0
\end{array}\right]
\begin{bmatrix} {}^{\mathrm{H}}x \\ {}^{\mathrm{H}}\dot{x} \\ \boldsymbol{\chi}_\theta \\ \hline {}^{\mathrm{H}}y \\ {}^{\mathrm{H}}\dot{y} \\ \boldsymbol{\chi}_\phi \\ \hline \psi \\ \hline z \\ \dot{z} \\ \chi_{\ddot{z}} \end{bmatrix}^{k}
$$

B Accelerometer model

In this chapter a simplified model of the 3-dimensional accelerometer, being part of the IMU within the drone, is developed. As explained in Section 2.1, accelerometers are electronic sensors that measure proper acceleration, also known as 'g-force'. At rest or at exact hover the proper acceleration experienced by the drone is an upward pointing acceleration, see see Figure B.1(a), equal to the opposite of the acceleration caused by gravity. At rest this can be used to give an estimate of the attitude by extracting this opposite gravity vector from the individual directional components, see Figure B.1(b). Though as shown in the following chapter the measurable accelerations when flying corresponds only to the thrust vector which always points upwards.



(a) Accelerometer on drone in hover with upward thrust vector

(b) Accelerometer at rest with surface providing upward force

Figure B.1: Accelerometer model

A simplified model of a 3-dimensional accelerometer contains a small mass, m_a, and three orthogonal spring-dampener connections to the housing of the accelerometer. At rest (on a surface) the accelerometer mass will be exposed to the gravity of earth, see Figure B.1(b). Similarly, the housing will be exposed to the same acceleration but will be provided with an equalling upward force from the surface, resulting in zero acceleration of the housing. As the mass itself is exposed to an acceleration while the housing is not, this will be measurable as an increase in the length of the spring, d, allowing the accelerometer to measure the gravity component as an upward acceleration. The dampener is included to model the asymptotically stable acceleration measurement, whereof only including the spring would result in oscillations. The mathematical model of a single axis is shown in (B.1).

$$\begin{aligned} z_a &= z - d \\ m_a \ddot{z}_a &= c\dot{d} + kd \\ m_a\ddot{d} + c\dot{d} + kd &= m_a\ddot{z} \end{aligned} \tag{B.1}$$

When the accelerometer is mounted on a drone in hover, the force, previously supplied by the surface, will instead be provided by the thrust vector from the propellers. As described above the accelerometer will exactly in hover measure the gravity component as an upward acceleration. However, if the drone is tilted and thereby not exactly in hover, the tilted thrust vector will make both the drone and hence the accelerometer housing accelerate, while the mass within the accelerometer would only be affected by gravity. See Figure B.2.

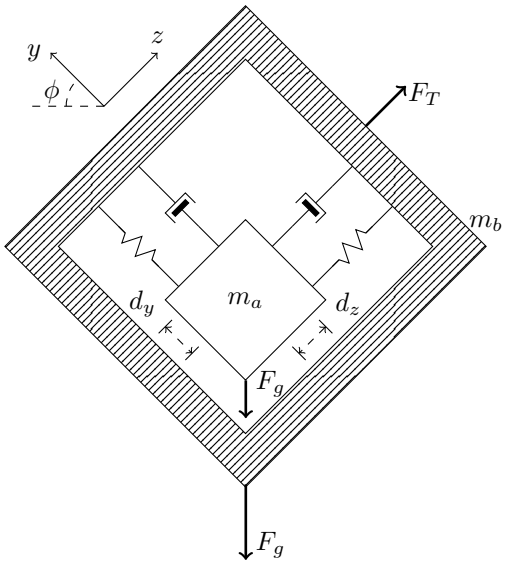


Figure B.2: 2D accelerometer mounted on tilted drone

It is possible to put up a model for the 2D accelerometer to identify how a tilted roll angle, ϕ, affects the accelerometer measurements. If the center of the drone is located at (y, z) and the center of the accelerometer mass is located at (y_a, z_a), assuming that the accelerometer frame is aligned with the drone frame. Then let d_y and d_y denote the displacement of the accelerometer mass.

$$\begin{aligned} z_a &= z - d_z \\ y_a &= y - d_y \end{aligned} \tag{B.2}$$

A dynamic model of the accelerometer mass is now derived:

$$\begin{aligned} m_a \ddot{z_a} &= -\cos(\phi) m_a g + c\dot{d_z} + k d_z \\ m_a \ddot{y_a} &= -\sin(\phi) m_a g + c\dot{d_y} + k d_y \end{aligned} \tag{B.3}$$

Inserting the relationship with the mass displacement:

$$\begin{aligned} m_a \ddot{d_z} + c\dot{d_z} + k d_z &= m_a \ddot{z} + \cos(\phi) m_a g \\ m_a \ddot{d_y} + c\dot{d_y} + k d_y &= m_a \ddot{y} + \sin(\phi) m_a g \end{aligned} \tag{B.4}$$

Another dynamic model is derived for the housing of the accelerometer:

$$\begin{aligned} m_b \ddot{z} &= -\cos(\phi) m_b g + F_T \\ m_b \ddot{y} &= -\sin(\phi) m_b g \end{aligned} \tag{B.5}$$

Combining (B.4) and (B.5) yields a closed model of the displacements given the thrust vector input and the roll angle:

$$\begin{aligned} m_a \ddot{d_z} + c\dot{d_z} + k d_z &= \cos(\phi) m_a g + m_a \left(\frac{F_T}{m_b} - \cos(\phi) g \right) \\ m_a \ddot{d_y} + c\dot{d_y} + k d_y &= \sin(\phi) m_a g - m_a \sin(\phi) g \end{aligned} \tag{B.6}$$

The accelerometer readings correspond to the spring displacements, d_y and d_z. When the accelerometer mass has settled, and hence the displacement derivatives are zero, it is apparent that the accelerometer is only be able to measure the thrust vector component.

$$\begin{aligned} kd_z &= \frac{m_a}{m_b} F_T \\ kd_y &= 0 \end{aligned} \tag{B.7}$$

All information about the tilted roll angle and the gravity vector is lost. The roll and pitch angles are therefore not easily estimated using the accelerometer. Although, if a simple velocity based wind resistance model is included, components of the accelerometer end up acting like a velocity sensor.

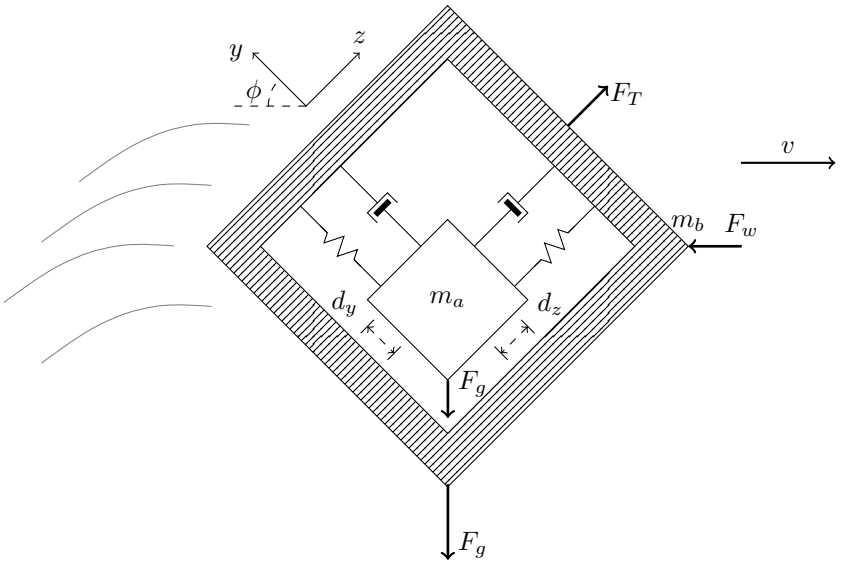


Figure B.3: Wind resistance model added to a tilted drone

If the drone is tilted as shown in Figure B.2 it will accelerate to the right, resulting in an increasing velocity to the right, v. This would create a wind resistance force in the opposite direction, $F_w = c_w v$, making the steady state x- and y-components of the accelerometer measurements proportional to the wind velocity.

$$\begin{aligned} m_b \ddot{y} &= -\sin(\phi) m_b g + \cos(\phi) F_w \\ kd_y &= \cos(\phi) \frac{m_a}{m_b} c_w v \end{aligned} \tag{B.8}$$

C GOT indoor positioning system

The GamesOnTrack system is an indoor positioning solution based on a transmitter-receiver configuration using a combination of ultrasound-waves and radio communication.

Figure C.1: GamesOnTrack system showing two satellites and one beacon [6]

The system consists of a transmitter, denoted as beacon, mounted to the object which needs to be located in a room where several receivers, known as satellites, are installed. At given timestamps the beacon sends an identifiable ultrasound pulse which is captured by the individual satellites. Simultaneously to the transmission of the ultrasound pulse the transmitter sends a message to all satellites over an RF channel with the timestamp of the just sent ultrasound pulse. This allows all receivers to independently determine the time-of-flight of the ultrasound pulse to their specific location. Every time-of-flight measurement is converted into a distance using the speed of sound. Through a calibration procedure the positions of all satellites, denoted the satellite configuration, are determined initially. With the distance measurements from the beacon to the individual satellites and with a known satellite configuration it is possible to perform a trilateration to determine the position of the beacon, see Figure C.2.

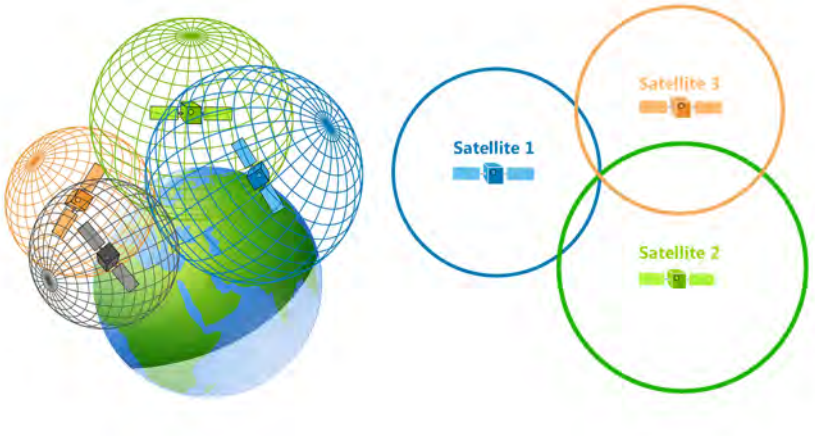


Figure C.2: Example of trilateration with GPS satellites [34]

At least four distance measurements from four satellites are necessary to uniquely determine the position of the beacon. With only distance measurements to three satellites, the trilateration will result in two possible positions. If the satellite configuration is installed in such a way that one of these positions is always invalid, it will be sufficient to use just three measurements at all times. This behaviour comes pre-programmed in a USB-connected GOT controller which takes care of both capturing the time-of-flight measurements and determining the position from trilateration. The GOT system can therefore be seen as a black-box sensor capable of providing measurements of the drone position.

Unfortunately the GOT system suffers from problems with deadzones or bad position estimates, just as the regular GPS system does when navigating in closely packed cities with tall buildings. As the GOT system is using ultrasound-waves the system is less vulnerable to multi-path effects due to the less energy in the ultrasound-waves, but the ultrasound-waves can still bounce uncontrollably from the objects in the environment causing unexpected behaviour of the determined position. As a consequence of using ultrasound-waves the system becomes more vulnerable to loss of sight and thus requires the beacon to always be in line-of-sight to at least three satellites. These unknown, unexpected and assumed unforeseeable errors with the position measurement from the GOT sensor deems it necessary to develop a position estimator that can take other measurements into account, as it is the case with this project where measurements from an RGB and depth camera is used.

Assuming that a small single emitter beacon is used, being the ones provided for the project, each satellite has a range of approximately 7-8 m [35]. As long as the transmitter is visible and within range of at least 3 satellites at all times the GOT system is capable of providing position measurements with an accuracy down to approximately 10 mm [36]. The update rate of the position measurements depends on the number of beacons. With a single beacon the GOT system is capable of providing position measurements at a rate of approximately 10 Hz.
This rate is limited due to the GOT design where an ultrasound package has a length of 50 ms together with the maximum possible distance of 8 m yielding approximately another 25 ms. For each extra beacon added the rate is halved as the ultrasound transmission channel has to be split equally. Within this project only one beacon is used why the update rate is assumed to be 10 Hz.

For the specific setup in the Motion Tracking lab on Fredrik Bajers Vej the following covariance matrix of the measurement noise has been estimated by a former group [7].

$$\Sigma_{GOT} = \begin{bmatrix} 0.225 & -0.038 & 0.022 \\ -0.038 & 0.025 & -0.009 \\ 0.022 & -0.009 & 0.014 \end{bmatrix} \cdot 10^{-4} \text{ m}^2 \tag{C.1}$$

Where:

Σ_{GOT}	covariance of measurements from GOT with given setup	$\left[\text{m}^2\right]$

This covariance together with the previously mentioned specifications of the system is used as the initial design parameter when designing and simulating the position estimator but will be adjusted as a tuning parameter if necessary.

D SLAM algorithms

In Chapter 3 the SLAM problem is introduced and the constraint graph visualizing the problem is shown in Section 3.1. Another way of modelling the SLAM problem when including the probabilistic constraints, hence another way of modelling the Bayesian network, is as a Growing Markov random field, see Figure D.1. The Bayesian network graph in Figure 3.1 can be simplified by including the constraints imposed by the measurements, as direct link between the hidden variables. This is known as a Markov random field abbreviated MRF. In the case of sequential stream of measurements this MRF would continuously grow resulting in the MRF shown in Figure D.1.

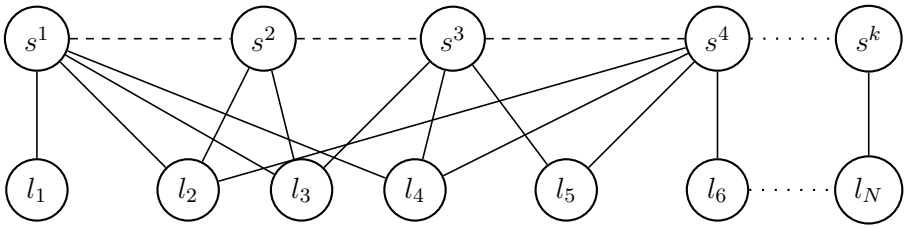


Figure D.1: Growing Markov random field [12]

Two common approaches to solve such an MRF and find the pose and map variables at a given timestep, given that all measurements are known, is either through recursive filtering or by performing an offline global optimization over the whole network. Related to the SLAM problem and especially within the field of 3D position estimation, such offline global optimizations are known as Bundle Adjustments.

In this appendix algorithms within these two approaches, solving the Markov random field in Figure D.1, are considered. The approaches are categorized as:

1. Filtering-based, eg. Bayesian SLAM
2. Keyframe-based, eg. Graph SLAM

The appendix describes the difference between these two categories and presents a few of the commonly used state-of-the-art implementations contained within each category.

D.1 Filtering-based SLAM

To use SLAM within control applications, eg. robotics, the pose estimates would need to come in a realtime sequential stream such that each new measurement also yields a new estimate. One way to do so is through filter-based methods. Filtering-based SLAM marginalizes out the past poses and keeps instead a joint probabilistic estimate of the current pose and map. Whenever a new measurement arrives, a prediction of the drone pose is added to the network (MRF) whereafter the previous drone pose is marginalized out and the whole network is updated based on the arrived measurement. This is known as Recursive Bayes filtering, see Section E.2.

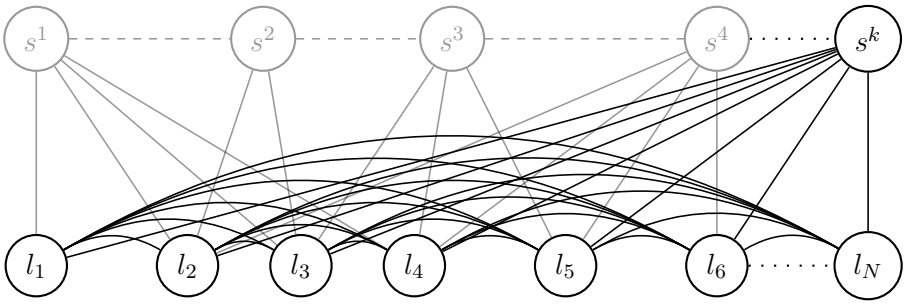


Figure D.2: Bayesian Network [12]

Applying the marginalization to the global MRF example shown in Figure D.1 results in the Bayesian network shown in Figure D.2. The past drone poses which are marginalized into the distribution of the current pose are marked as grey circles. Due to the marginalization probabilistic links are introduced between every pair of landmarks. Even though the filtering-based SLAM contains less nodes than the MRF, on which global optimization was possible, one could say that the filtering-based SLAM does not contain less information as the extra links contain the marginalized information.

Existing state-of-the-art filtering-based SLAM implementations include:

- EKF-SLAM
- Particle filter SLAM - FastSLAM 1.0 and 2.0
- MonoSLAM
- Information filter SLAM
- UKF-SLAM

A short description of EKF-SLAM, MonoSLAM and FastSLAM, being some of the most commonly used state-of-the-art implementations within Filtering-based SLAM, is given in the following sections. For further descriptions and comparisons between EKF, UKF, FastSLAM and an optimized FastSLAM the reader is referred to [37].

D.1.1 EKF-SLAM

One way of implementing Bayesian SLAM is by using an Extended Kalman Filter, see Section E.5, which is just one type of implementation of the Bayes filter, see Section E.2. In the case of using an Extended Kalman Filter for SLAM, also known as EKF-SLAM, the state vector of the estimator will include the full drone pose and position of all mapped landmarks. At each timestep the Kalman filter prediction and correction step is performed, taking in all available measurements of detected landmarks. As the number of mapped landmarks increases, the state vector and hence also the covariance matrix inside the Kalman filter grows. By comparison this would correspond to the constraints in the Bayesian Network growing unbounded as new measurements arrive as described previously. This makes EKF-SLAM more computationally heavy as this growing number of constraints has to be handled.

So at the cost of computational power and memory consumption EKF-SLAM maintain a joint probability density estimate of the current pose and map in a more efficient manner than keyframe-based SLAM, which keeps no such uncertainty estimate.

Most of the time the higher accuracy provided by the commonly used sensors within Visual SLAM, eg. cameras or depth sensors, makes it practically unnecessary to calculate such uncertainty, though this uncertainty is helpful if the algorithm is fused with other sensors.

D.1.2 MonoSLAM

As mentioned with both EKF-SLAM and FastSLAM both algorithms were originally developed and intended for simple measurements such as distance sensors and odometry inputs. In the early 2000s though A. Davison brought vision into the world of filtering-based SLAM by introducing his MonoSLAM approach performing EKF-SLAM with a single camera [38]. The EKF-SLAM approach was extended to include the 3D position of landmarks and the pose extended to six dimensions, hence extending the dimension of the otherwise sparse map well known to filtering-based SLAM approaches. This opens up for the possibility of using 3D measurements of the landmarks within the environment to correct the estimate. Although a single camera does not provide 3D measurements and instead contains the problem of unrecovered scale, Davison proposed a probabilistic initialization of the 3D position of landmarks before inserting them into the map, later known as inverse depth parametrization. When detecting a new landmark a probability distribution of its' inverse depth estimate is iteratively updated through frame-to-frame matching. This allows the depth and thereby the 3D position of a landmark to be determined before it is inserted into the map.

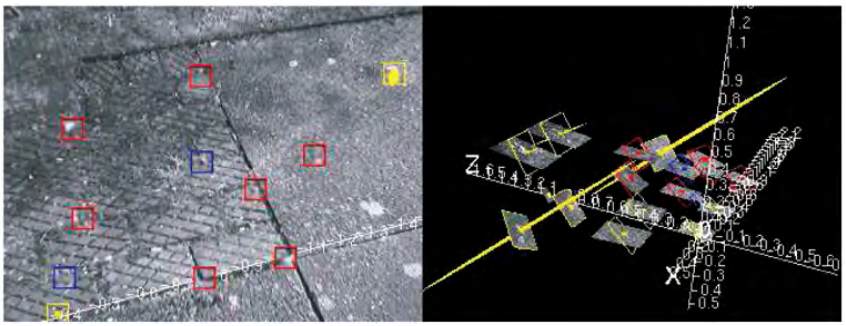


Figure D.3: Example of filtering-based SLAM, Notice the large depth uncertainty of the uninitialized landmark (yellow line) [38]

For camera measurements to result in actual uniquely identifiable landmark measurements to be used as input to EKF-SLAM, Davison proposed a method of finding salient image patches within the image and extract these as landmarks. Whenever a new salient feature is found, an image patch of 11×11 pixels is stored temporarily together with the inverse depth parametrization. This allows the image patch to be rediscovered and thus the fully inverse depth to be estimated over a short amount of time. After the depth has been estimated the landmark, now defined by its' image patch, identifier and 3D location, is inserted as a landmark into the map of EKF-SLAM, hence inserted as a part of the state vector.

D.1.3 Particle filter SLAM

One of the assumptions when using EKF-SLAM is that the pose estimate can be approximated by a Gaussian distribution and hence assumed unimodal but with the benefit of using very simple parametrizations of the continuous distribution. On the other hand a Bayes filter working on discrete distributions is able to describe any kind of distribution but at the cost of discretization.

When navigating around in an environment, observing only a few landmarks, one can easily end up in multi-modal situations where multiple viewing angles of the same subset of landmarks will yield the exact same measurement. This is both a consequence of the measurement sensor, in this case an RGB-D camera which projects the 3D world onto 2D image planes, and a consequence of seeing few landmarks. EKF-SLAM will in this case not be able to approximate such multi-modal distribution and will likely diverge if such a situation is experienced.

One solution to this problem is to use a discrete approximation of the estimated pose distribution which can be contained and handled with a Particle Filter, see Section E.8. In general the functionality of a Particle Filter is similar to the discrete implementation of the Bayes filter known as the Histogram Filter. Where the Histogram Filter contains individual containers for the probability each possible but discretized pose estimates, the Particle Filter takes the Monte-Carlo approach by spawning a certain amount of particles which each is described by a pose estimate. In areas of the discrete probability distribution with high probability a lot of particles will be gathered whereof low-probability areas will only have a few or no particles.

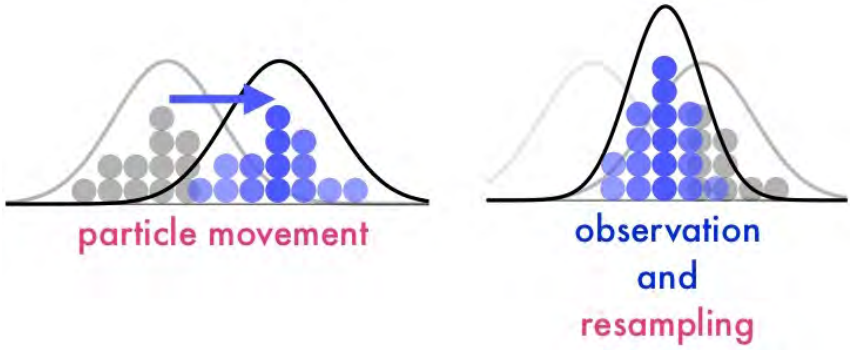


Figure D.4: Particle filter example approximating a Gaussian distribution [39]

When performing corrections on the particle filter, incorporating measurements, a weight is calculated for each particle based on the likelihood of the single particle experiencing the given measurement. The collection of weights resembles the desired posterior distribution and thus defines how the particle distribution should look after resampling. In short, the resampling consists of duplicating particles with high weight and removing particles with low weight.
The Particle Filter can be used to solve the probabilistic SLAM problem but the large and growing state vector will make such an implementation computationally infeasible. Such an implementation will also yield worse results than EKF-SLAM as almost infinite numbers of particles would be necessary to reasonably well describe the estimated pose distribution due to the large state space.

FastSLAM
The computational problem of filtering-based SLAM has been approached by several researchers but the state-of-the-art method simplifying the problem is known as FastSLAM proposed in 2002. The FastSLAM algorithms utilise the definition of conditional probability to split the SLAM problem into two sub-problems, one estimating the pose of the drone and another estimating the location of all landmarks within the map, see Section E.9.

$$p(\boldsymbol{s}, \boldsymbol{M} \mid \boldsymbol{u}, \boldsymbol{z}) = p(\boldsymbol{s} \mid \boldsymbol{u}, \boldsymbol{z}) \prod_{i=1}^{N} p(\boldsymbol{l}_i \mid \boldsymbol{s}, \boldsymbol{z}) \tag{D.1}$$

Assuming conditional independence between all landmarks within the map, the posterior can be factorized into the probability of the pose given the map and the product of the individual probabilities of each landmark within the map, as shown in (D.1). If one then assumes that the individual landmark estimates can be described by Gaussian distributions the individual landmark probabilities can be contained within many individual but small Extended Kalman Filters. This allows the estimation of the pose to be governed by a particle filter and the estimation of the map of landmarks to be governed by individual and independent Extended Kalman Filters. The independency stems from the fact that all landmarks are only dependent of each other if the location of the pose is not known, as shown with the constraints in Figure D.1. As described previously each particle within the particle filter represents a given possible pose, why the pose is known to the landmarks within a certain particle such that each landmark become independent and can be estimated and handled individually. This approach also known as the Rao-Blackwellized particle filter being a combination of the particle filter and the Kalman filter, where each particle has one or more Kalman filters associated to them [40].
FastSLAM has become the most commonly used filtering-based SLAM algorithm especially used with odometry sensors, such as wheel encoders and distance sensors such as laser or ultrasound as perception input. Filtering-based SLAM is in general limited in the number of landmarks that can be included and the number of degrees of freedom to estimate. This is the reason that filtering-based methods are usually used within 2D robotic applications.

FastSLAM 2.0
As the FastSLAM algorithm is still a discrete approximation of the pose estimate distribution a very accurate motion model or a certain minimum amount of particles is needed for the filter to be able to track the correct pose. At state vectors of low dimensionality this works out nicely with just a few hundred particles but if the dimension of the state vector becomes larger than 3 states one would likely need more than thousand particles to be able to represent the pose distribution equally well. Hence a need of further optimization of the algorithm was needed, which has been the research topic for several of the FastSLAM researchers after the initial presentation.

In 2003 an improved version of FastSLAM was proposed called FastSLAM 2.0. The main difference which makes FastSLAM 2.0 more efficient than its' predecessor is how measurements are included in the proposal distributions defining the distribution used when prediction the pose of a given particle, see Section E.9. By including the current measurement the uncertain motion model will thus be narrowed down and the prediction of the particles will be concentrated to poses where it is more likely to experience the given measurement, see (7.10). This allows less particles to be used, hence lowering the computational requirement, while the estimate is still tracking the pose well. A good overview of previous work with FastSLAM and 3D measurements can be found within [41] however using a stereo-camera set-up.

D.2 Keyframe-based SLAM

Another way to get realtime sequential position estimates is with Keyframe-based SLAM which adapts the unbounded global optimization problem, the offline Bundle Adjustment, to sequential processing of a video stream by separating the process into two parallel threads. One thread runs at a fast rate (full camera frame rate) which estimates the pose from the known features already existing in the map, calculating an initial pairwise transformation estimate. A second thread runs at a much slower rate performing the global optimization over all the tracked features and a selected set of stored keyframes (pose and map), thereby minimizing the reprojection error and refines the initial pairwise estimation into a globally consistent one.

Some Keyframe-based approaches stores only the past few frames, while others store frames at different locations eg. when detecting and inserting new landmarks into the map which is usually done when the amount of tracked landmarks are sparse, see Figure D.5. In this sense the Keyframe-based SLAM differ in many ways from the previously dominant filtering-based approaches such as EKF-SLAM or FastSLAM mentioned in Section D.1.

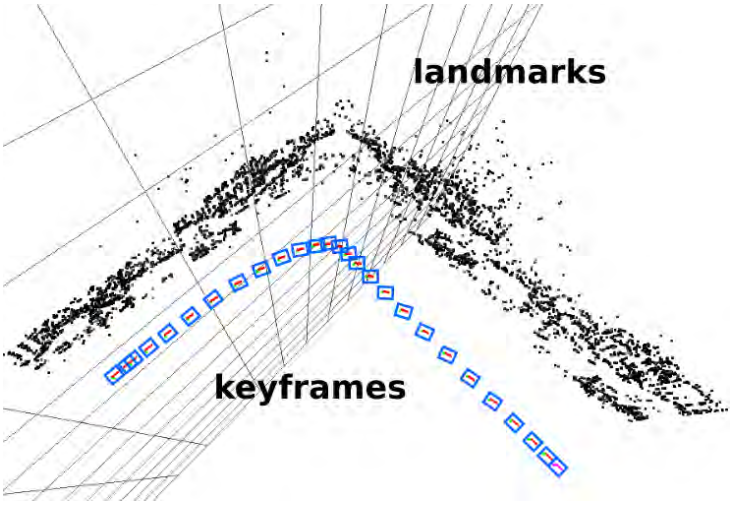


Figure D.5: Keyframes and landmarks stored for Global Bundle Adjustment [11]

Instead of marginalizing out previous poses and summarizing all information into one probability distribution, as with the filtering-based SLAM, Keyframe-based approaches retain a selected subset of previous observations, called keyframes, explicitly representing past knowledge shown as the solid black circles in Figure D.6. These keyframes are used in the optimization (Bundle Adjustment) to estimate the pose, trajectory and landmark map, whereof the grey circles marks the keyframes which are not stored for optimization. The constraints to these keyframes, hence possibly valuable information, are thus lost, eg. the link between l_3 and l_4 through s_3 in Figure D.6.

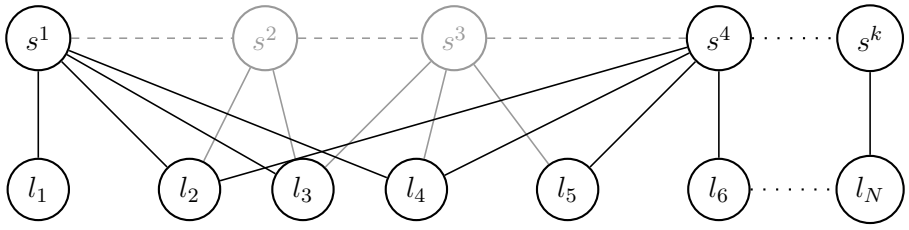


Figure D.6: Keyframe-based SLAM includes a graph of keyframes and constraints [12]

The pairwise transformation estimation is usually performed through an iterative process, eg. Iterative Closest Point, in where the Euclidean distance between two consecutive keyframes, containing the detected landmarks within that keyframe, are minimized to get the transformation between the two adjacent frames. Other methods to perform the pairwise estimation includes MICP, GICP or 3D-RANSAC which helps to remove possible outliers by fitting the measurements to a model instead of just minimizing the Euclidean distance [42].It is apparent that the keyframes and the resulting constraints can be described by a graph as shown in Figure D.6. This allows graph-based optimization techniques to be used for the global optimization. Due to the graph-based structure, Keyframe-based SLAM algorithms are also classified as Graph SLAM.

Existing state-of-the-art Keyframe-based SLAM implementations include:

- DTAM
- PTAM
- RGB-D SLAM
- ORB-SLAM
- SVO
- LSD-SLAM

A short description of the most commonly used, being PTAM, ORB-SLAM and RGB-D SLAM is provided below.

D.2.1 DTAM and PTAM

Simultaneous to the research done within MonoSLAM an interest for realtime tracking and mapping using monocular cameras was formed within computer vision groups. PTAM, an abbreviation of Parallel Tracking and Mapping, was presented in 2007 [43] intended for Augmented Reality (AR) applications using handheld cameras where rapid unmodelled motion is likely to occur. Similar to MonoSLAM the objective is to estimate the 6-dimensional pose containing the current camera location and orientation. The research within PTAM identifies the main issues of the filtering-based methods, eg. EKF-SLAM and MonoSLAM, as the dependency on a well-fitting motion model to predict a prior pose to assure correct feature matching. When applying these methods to hand-held cameras instead of robots, in where the motion can be both rapid and irregular, these algorithms were likely to diverge or do incorrect mapping.

The work of PTAM splits the problem into two parallel threads, one doing the realtime tracking and another doing the mapping. The tracking thread receives the incoming images from the hand-held camera from which coarsest-scale features are extracted. A simple motion model is used to generate a prior pose such that the landmarks within the built map can be projected onto the current image plane and used together with the extracted features to update the pose estimate. The pose is updated by iteratively minimising a robust objective function of the reprojection error. This results in fast tracking of the pose, although depending on an already built map.

When the system is started the map itself will be empty and has to be initialized. The map is densely initialised from a stereo keyframe pair resulting from just a small translational movement of the camera and landmarks are inserted into the map based on a 5-Point algorithm. The other thread doing the mapping hereafter evaluates when to insert new keyframes and landmarks into the map, depending on the actual movement. Many sequential images contain redundant information, particularly when the camera is not moving. MonoSLAM and other incremental systems would waste their time re- filtering the same measurement image after image, though PTAM will only focus on a smaller number of useful and cleverly selected keyframes. Landmarks however will be continously added based on an epipolar search originating from the extracted features. Whenever new keyframes are added to the map a Global Bundle Adjustment is performed to optimize and adjust the pose of all keyframes and location of landmarks in such a way that the constraints are tightened. Even though the optimization might easily take tens of seconds to converge when 150 keyframes or more are included, the mapping thread can run at a much lower rate than the tracking thread without ruining the tracking.

PTAM use the FAST corner detector to extract coarsest-scale features to use to limit the number of landmarks within the map, hence reducing the computational requirements. In 2011 PTAM was extended to DTAM focusing on Dense Tracking and Mapping [44]. Reliying on high-performance GPU systems DTAM removes the feature extraction part of PTAM and instead includes all possible measurements, being all pixels, in both the tracking and mapping thread. The result, as seen in Figure D.7, is a much more dense point-cloud of landmarks. A point-cloud from which it is clearly possible to identify the structure of objects in comparison to PTAM where the individual features are more of less arbitrary scattered.

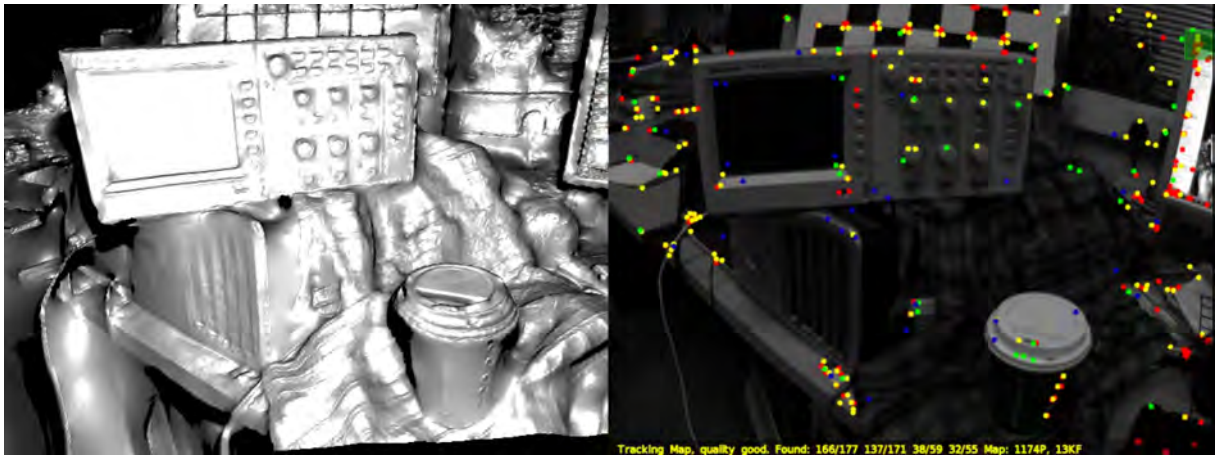


Figure D.7: Comparison between DTAM (left) and PTAM (right) [44]

Performing the Bundle Adjustments on the dense maps of DTAM requires tremendous amounts of compational power why DTAM is usually only used when mapping is the primary objective. Although PTAM was designed specifically for AR applications and only works well in smaller environments where global map management is not needed, the method of parallelizing tracking and mapping and the way of doing keyframe-based map management is used by most of todays state-of-the-art feature-based visual SLAM systems, eg. ORB-SLAM and SVO [11].

D.2.2 ORB-SLAM

Unfortunately some caveats were later discovered with PTAM where tracking would be likely to fail at occlusions or severe motion clutter or in environments where only few features can be found. Lastly PTAM does not handle loop closures in any careful way, why the tracking can easily diverge if an incorrect loop closure is made.

ORB-SLAM is a more traditional feature based system but quite similar to PTAM. ORB-SLAM was presented in 2015 [45] as a new and efficient way of doing Visual SLAM using monocular or stereo cameras with a highly improved performance in practice compared to PTAM. Its main improvement compared to PTAM includes but are not limited to:

1. Implements 3 parallel threads for tracking, mapping, and loop closure. This allows ORB-SLAM to achieve consistent localization and mapping. In comparison PTAM does not have loop closure.

2. Automatic map initialization by calculating the initial homography and fundamental ego-motion of the camera using RANSAC on a set of different models. In comparison PTAM requires manual operation to finish initialization.

3. Use ORB feature detector and descriptor instead of image patches used in PTAM. The ORB detector improves the robustness of image tracking and feature matching under scale and orientation changes, where the image patches within PTAM is likely to fail.

4. Multi-scale mapping, including several graphs on which Bundle Adjustments (BA) are performed. A local graph for pose BA, a co-visibility graph for local BA and an essential graph for global BA after loop closure detection.

The benefits of using ORB-SLAM over PTAM is the increased performance and robustness. Whereas PTAM uses FAST corners for feature extraction ORB-SLAM uses an extension to this being oriented multi-scale FAST corners defined as Oriented Brief. These features are faster to extract and both rotation and scale invariant, hence increasing the robustness as features are easier to rediscover from different viewing angles. Thereby using this improved image feature detector and descriptor allows ORB-SLAM implementations to achieve real-time performance on regular desktop CPUs which other feature detectors such as SIFT or SURF, see Section 3.3.1, cannot provide [11].

D.2.3 RGB-D SLAM

Finally if measurements are provided by an RGB-D camera, as it is the case within this project, providing both an RGB image and a disparity (depth) image, the sparse RGB-D SLAM implementation from 2012 [46] is an interesting way to use the keyframe-based SLAM approach presented above with RGB-D measurements. The implementation is very similar to PTAM or ORB-SLAM except that the RGB image is mainly used for finding salient features which are then projected into the depth image to pick salient point from the generated point-clouds. This removes the need for a specific initialization of the map as the depth measurements are always given. A flowchart diagram of the RGB-D SLAM algorithm in shown in Figure D.8.

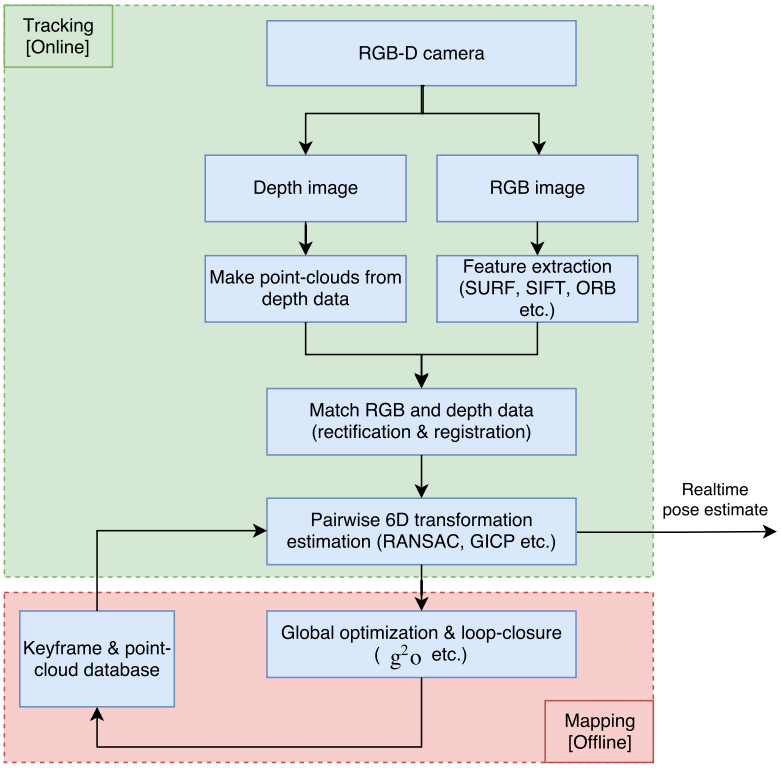


Figure D.8: RGB-D SLAM algorithm flowchart, [46]

Whenever the RGB-D SLAM receives a set of images, RGB and depth, a set of salient features is extracted from the RGB image using an efficient and invariant feature detector, see Section 3.3.1, similar to how it is done within PTAM and ORB-SLAM. These features then decide which points to pick from the generated point-cloud based on the depth image, and the RGB image is thus only used to extract salient features from the point-cloud.

The extracted set of points are then used in a similar fashion as extracted features within PTAM and ORB-SLAM. Within the tracking thread the points are used to calculate a pairwise 6-dimensional transformation between the previous keyframe and the currently extracted points. This results in an realtime pose estimate. The extracted points are further applied to the global optimization and loop-closure algorithms running at a slower rate within the mapping thread. This allows stored keyframes consisting of pose estimates and recorded point-clouds to be optimized, tightening their constraints which improves the map and keeps it consistent.

D.3 Comparison between algorithms

Based on the descriptions of the different algorithms within this appendix and a deeper analysis of their capabilities and requirements, a recommendation chart shown in Figure D.9 is drawn. This diagram recommends certain SLAM implementations, though without limiting, depending on different properties and design considerations of the system. Especially the computational cost is considered but also the complexity of the implementation [47].

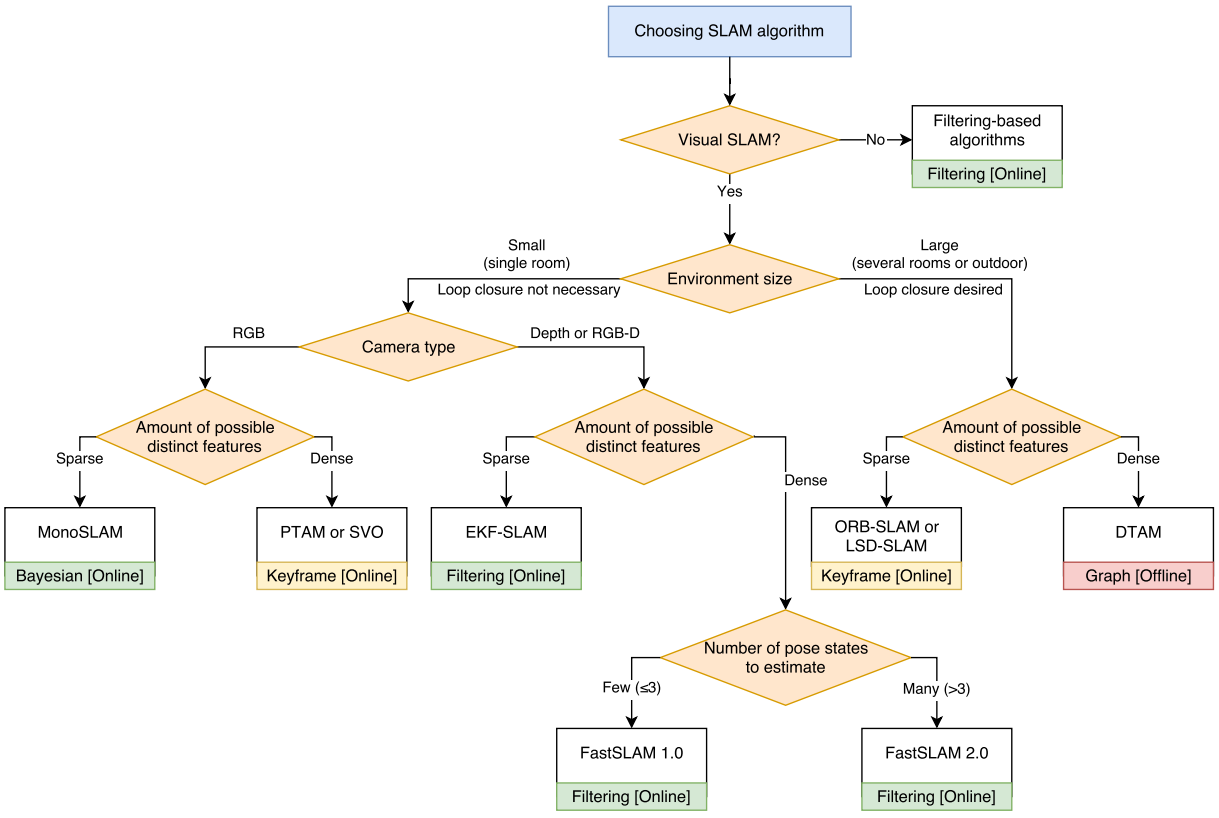


Figure D.9: Recommended SLAM algorithm depending on use-case

E Estimation Theory and Methods

This appendix presents different methods for estimating a state vector based on knowledge about a model of a system, a model of how measurements are obtained, and actual measurements. Initially a common list of probability theorems and terms is introduced to the reader followed by a presentation of the recursive Bayesian Estimator in Section E.2. In Section E.3 and Section E.4 prerequisites for the derivation of the Discrete Extended Kalman Filter are presented. The Discrete Extended Kalman Filter, being a special case of the Bayesian Estimator, is presented in Section E.5. In Section E.8 the particle filter is presented which can best be described as a numerical implementation of the Bayesian Estimator.

E.1 Common probability theorems and terms

Throughout both this appendix and the report, terms such as 'a priori' and 'a posteriori' estimates are used including a pipe notation indicating the time of included empirical evidence within an estimate

- **a priori**: is used about estimates at time k that are not based on empirical evidence obtained at time k. E.g. an a priori estimate of a state vector at time k could be obtained by using a motion model and will be denoted as $\bar{\boldsymbol{\chi}}^{k|k-1}$.
- **a posteriori**: is used about estimates at time k that are based on empirical evidence obtained at time k. E.g. when observed measurements are incorporated into an a priori estimate resulting in an estimate denoted as $\bar{\boldsymbol{\chi}}^{k|k}$.

Besides that, the symbol p is used to denote a probability density function, which is abbreviated as PDF. Furthermore is the following common probability theorems are used in the derivations in the following sections, and the reader should be familiar with those.
Bayes' formula:

$$p(x,y) = p(x \mid y)\, p(y) = p(y \mid x)\, p(x) \tag{E.1}$$

$$p(x \mid y) = \frac{p(x,y)}{p(y)} = \frac{p(y \mid x)\, p(x)}{p(y)} \tag{E.2}$$

$$= \eta p(y \mid x)\, p(x) \tag{E.3}$$

$$\propto p(y \mid x)\, p(x) \tag{E.4}$$

Bayes' formula can also easily be extended to multiple variables being either jointly distributed or conditioned:

$$p(x,y,z) = p(x,y \mid z)\, p(z) \tag{E.5}$$

$$p(x,y,z) = p(x \mid y,z)\, p(y,z) \tag{E.6}$$

$$\tag{E.7}$$

If a joint probability is conditioned on a random variable, applying Bayes' formula will result in all terms being conditioned on this variable:

$$p(x,y \mid z) = p(x \mid y,z)\, p(y \mid z) \tag{E.8}$$

$$\tag{E.9}$$

Marginalization of joint distributions:

$$p(x) = \int p(x, y)\, dy \tag{E.10}$$

E.2 Recursive Bayesian State Estimator

A recursive Bayesian state estimator tries to recursively approximate the conditional PDF

$$p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k}) \tag{E.11}$$

Where:

$\boldsymbol{\chi}^k$	is the state vector of the system at time k
$\boldsymbol{z}^{1:k}$	is the set of all measurement available up to time k

In this section a system and measurement model given on the form in (E.12) and (E.13), respectively, are assumed [30].

$$\boldsymbol{\chi}^k = f^{k-1}\left(\boldsymbol{\chi}^{k-1}, \boldsymbol{w}^{k-1}\right) \tag{E.12}$$

$$\boldsymbol{z}^k = h^k\left(\boldsymbol{\chi}^k, \boldsymbol{v}^k\right) \tag{E.13}$$

Where:

$f^{k-1}(\bullet)$	is the time-varying system model
$h^k(\bullet)$	is the time-varying measurement model
$\boldsymbol{\chi}$	is the state vector
$\boldsymbol{w}$	is a noise variable called the process noise
$\boldsymbol{v}$	is a noise variable called the measurement noise

The process noise, $\boldsymbol{w}$, and measurement noise, $\boldsymbol{v}$, are assumed to be independent and white. Furthermore, it is assumed that $p(\boldsymbol{\chi}^0 \,|\, \boldsymbol{z}^0) = p\left(\boldsymbol{\chi}^0\right)$ is known. With these assumptions a recursive estimator of the PDF in (E.11) can be derived [30]. This recursive estimator can be summarised by the formulas given in (E.14) and (E.15).

$$p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1}) = \int p(\boldsymbol{\chi}^k \mid \boldsymbol{\chi}^{k-1}) p(\boldsymbol{\chi}^{k-1} \mid \boldsymbol{z}^{1:k-1})\, d\boldsymbol{\chi}^{k-1} \tag{E.14}$$

$$p(\boldsymbol{z}^k) = p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k}) = \frac{p(\boldsymbol{z}^k \mid \boldsymbol{\chi}^k)\, p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1})}{p(\boldsymbol{z}^k \mid \boldsymbol{z}^{1:k-1})} = \frac{p(\boldsymbol{z}^k \mid \boldsymbol{\chi}^k)\, p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1})}{\int p(\boldsymbol{z}^k \mid \boldsymbol{\chi}^k)\, p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1})\, d\boldsymbol{\chi}^k} \tag{E.15}$$

These equations are very general and could in principal be applied to a broad range of problems. Unfortunately, analytical solutions to these equations are often hard to derive, and thus approximations are often used instead.

E.2.1 Properties

The good thing about this kind of estimator is that all the statistical properties about the state vector at time k, $\boldsymbol{\chi}^k$, are available, since the full PDF is approximated. Thus, the estimate of the state vector, $\hat{\boldsymbol{\chi}}^k$, can be chosen freely using the PDF. To emphasize that this is indeed a good feature of an estimator, consider the PDF shown in Figure E.1.

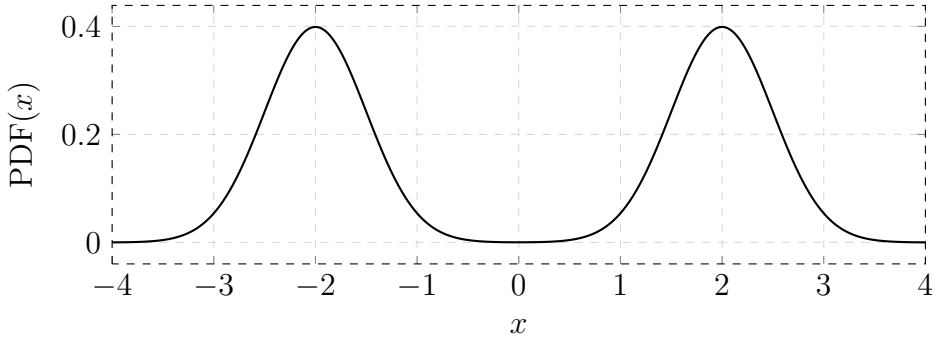


Figure E.1: Example of a multimodal PDF

Using the mean, $\bar{\boldsymbol{\chi}}^k = 0$, as an estimate of the variable with the PDF given in Figure E.1 would probably not be beneficial, since $p\left(\boldsymbol{\chi}^k = 0\right) = 0$. Thus, knowing the entire PDF, other features could be chosen as an estimate for the state vector. For example those values of the state vector, $\boldsymbol{\chi}^k$, that locally maximizes the PDF could be chosen as estimates. In the example above this would result in an estimate being both $\hat{\chi} = -2$ and $\hat{\chi} = 2$.

E.3 Propagation of States and Covariances Through Linear Discrete Systems

In the following a model on the form given in (E.16) is assumed.

$$\boldsymbol{\chi}^k = \boldsymbol{F}\boldsymbol{\chi}^{k-1} + \boldsymbol{G}\boldsymbol{u}^{k-1} + \boldsymbol{w}^{k-1} \tag{E.16}$$

Where:

$\boldsymbol{F}$	is a model dynamics matrix
$\boldsymbol{G}$	is an input matrix
$\boldsymbol{\chi}$	is the state vector
$\boldsymbol{u}$	is a known input to the system
$\boldsymbol{w}$	is a noise variable called the process noise

In this section it is assumed that w has zero mean and known covariance, i.e.

$$\boldsymbol{w}^k \sim \left(0, \mathrm{Cov}\left(\boldsymbol{w}^k\right)\right) \tag{E.17}$$

The mean, $\bar{\boldsymbol{\chi}}^k$, of $\boldsymbol{\chi}^k$, is calculated as the expected value of (E.16):

$$\begin{aligned}
\bar{\boldsymbol{\chi}}^k &= \mathbb{E}\left[\boldsymbol{\chi}^k\right] && \text{(E.18)}\\
&= \mathbb{E}\left[\boldsymbol{F}\boldsymbol{\chi}^{k-1} + \boldsymbol{G}\boldsymbol{u}^{k-1} + \boldsymbol{w}^{k-1}\right] && \text{(E.19)}\\
&= \mathbb{E}\left[\boldsymbol{F}\boldsymbol{\chi}^{k-1}\right] + \mathbb{E}\left[\boldsymbol{G}\boldsymbol{u}^{k-1}\right] + \mathbb{E}\left[\boldsymbol{w}^{k-1}\right] && \text{(E.20)}\\
&= \boldsymbol{F}\bar{\boldsymbol{\chi}}^{k-1} + \boldsymbol{G}\boldsymbol{u}^{k-1} && \text{(E.21)}
\end{aligned}$$

And the covariance can be calculated as:

$$\text{Cov}\left(\boldsymbol{\chi}^k\right) = \mathbb{E}\left[\left(\boldsymbol{\chi}^k - \bar{\boldsymbol{\chi}}^k\right)\left(\boldsymbol{\chi}^k - \bar{\boldsymbol{\chi}}^k\right)^T\right] \tag{E.22}$$

$$= \mathbb{E}\left[\left(\boldsymbol{F}\boldsymbol{\chi}^{k-1} + \boldsymbol{G}\boldsymbol{u}^{k-1} + \boldsymbol{w}^{k-1} - \left(\boldsymbol{F}\bar{\boldsymbol{\chi}}^{k-1} + \boldsymbol{G}\boldsymbol{u}^{k-1}\right)\right)\right. \tag{E.23}$$

$$\left.\left(\boldsymbol{F}\boldsymbol{\chi}^{k-1} + \boldsymbol{G}\boldsymbol{u}^{k-1} + \boldsymbol{w}^{k-1} - \left(\boldsymbol{F}\bar{\boldsymbol{\chi}}^{k-1} + \boldsymbol{G}\boldsymbol{u}^{k-1}\right)\right)^T\right] \tag{E.24}$$

$$= \mathbb{E}\left[\left(\boldsymbol{F}\left(\boldsymbol{\chi}^{k-1} - \bar{\boldsymbol{\chi}}^{k-1}\right) + \boldsymbol{w}^{k-1}\right)\left(\boldsymbol{F}\left(\boldsymbol{\chi}^{k-1} - \bar{\boldsymbol{\chi}}^{k-1}\right) + \boldsymbol{w}^{k-1}\right)^T\right] \tag{E.25}$$

$$= \mathbb{E}\left[\boldsymbol{F}\left(\boldsymbol{\chi}^{k-1} - \bar{\boldsymbol{\chi}}^{k-1}\right)\left(\boldsymbol{\chi}^{k-1} - \bar{\boldsymbol{\chi}}^{k-1}\right)^T\boldsymbol{F}^T + \boldsymbol{w}^{k-1}\left(\boldsymbol{w}^{k-1}\right)^T + \right. \tag{E.26}$$

$$\left.\boldsymbol{F}\left(\boldsymbol{\chi}^{k-1} - \bar{\boldsymbol{\chi}}^{k-1}\right)\left(\boldsymbol{w}^{k-1}\right)^T + \boldsymbol{w}^{k-1}\left(\boldsymbol{\chi}^{k-1} - \bar{\boldsymbol{\chi}}^{k-1}\right)^T\boldsymbol{F}^T\right] \tag{E.27}$$

From (E.16) and (E.21) it is evident that $\boldsymbol{\chi}^{k-1}$ and $\boldsymbol{w}^{k-1}$ are uncorrelated, and that $\bar{\boldsymbol{\chi}}^{k-1}$ and $\boldsymbol{w}^{k-1}$ are uncorrelated. Therefore $\left(\boldsymbol{\chi}^{k-1} - \bar{\boldsymbol{\chi}}^{k-1}\right)$ and $\boldsymbol{w}^{k-1}$ must also be uncorrelated. Thus (E.27) reduces to

$$\text{Cov}\left(\boldsymbol{\chi}^k\right) = \mathbb{E}\left[\boldsymbol{F}\left(\boldsymbol{\chi}^{k-1} - \bar{\boldsymbol{\chi}}^{k-1}\right)\left(\boldsymbol{\chi}^{k-1} - \bar{\boldsymbol{\chi}}^{k-1}\right)^T\boldsymbol{F}^T + \boldsymbol{w}^{k-1}\left(\boldsymbol{w}^{k-1}\right)^T\right] \tag{E.28}$$

$$= \boldsymbol{F}\text{Cov}\left(\boldsymbol{\chi}^{k-1}\right)\boldsymbol{F}^T + \text{Cov}\left(\boldsymbol{w}^{k-1}\right) \tag{E.29}$$

E.4 Optimal Affine Recursive Least Squares Estimation

This section describes a affine recursive least square estimator on the form given in (E.30) that minimizes the sum of variances of the estimation errors [30]. This type of estimator is intended for estimating a random variable, $\boldsymbol{\chi}$, based on several noisy measurements of that variable, $\boldsymbol{z}_i$, taken at the same time instant.

$$\hat{\boldsymbol{\chi}}_i = \hat{\boldsymbol{\chi}}_{i-1} + \boldsymbol{K}_i\left(\boldsymbol{z}_i - \hat{\boldsymbol{z}}_i\right) \tag{E.30}$$

Where:

$\hat{\boldsymbol{\chi}}_i$	is the estimate after incorporating the i'th measurement
$\boldsymbol{z}_i$	is the i'th independent measurement
$\hat{\boldsymbol{z}}_i$	is an estimate of the i'th measurement calculated based on the measurement model and the current best estimate, $\hat{\boldsymbol{\chi}}_{i-1}$

In this section a measurement model on the form in (E.31) is assumed.

$$\boldsymbol{z}_i = \boldsymbol{H}_i\boldsymbol{\chi} + \boldsymbol{y}_i + \boldsymbol{v}_i \tag{E.31}$$

Where:

$\boldsymbol{H}_i$	is a known matrix of the measurement model
$\boldsymbol{y}_i$	is a known vector of the measurement model
$\boldsymbol{v}_i$	is a noise variable, adding uncertainty to the i'th measurement, that is assumed to be unknown

Since the value of $\boldsymbol{v}$ and the true value of $\boldsymbol{\chi}$ are unknown, the best estimate of the measurement is thus $\hat{\boldsymbol{z}}_i = \boldsymbol{H}_i\hat{\boldsymbol{\chi}}_{i-1} + \boldsymbol{y}_i$. This estimator has an estimation error mean of

$$\begin{aligned}
\mathbb{E}\left[\boldsymbol{\epsilon}_i\right] &= \mathbb{E}\left[\boldsymbol{\chi} - \hat{\boldsymbol{\chi}}_i\right] && \text{(E.32)}\\
&= \mathbb{E}\left[\boldsymbol{\chi} - \hat{\boldsymbol{\chi}}_{i-1} + \boldsymbol{K}_i\left(\boldsymbol{z}_i - \hat{\boldsymbol{z}}_i\right)\right] && \text{(E.33)}\\
&= \mathbb{E}\left[\boldsymbol{\epsilon}_{i-1} - \boldsymbol{K}_i\left(\boldsymbol{H}_i\boldsymbol{\chi} + \boldsymbol{y}_i + \boldsymbol{v}_i - \left(\boldsymbol{H}_i\hat{\boldsymbol{\chi}}_{i-1} + \boldsymbol{y}_i\right)\right)\right] && \text{(E.34)}\\
&= \mathbb{E}\left[\boldsymbol{\epsilon}_{i-1} - \boldsymbol{K}_i\boldsymbol{H}_i\left(\boldsymbol{\chi} - \hat{\boldsymbol{\chi}}_{i-1}\right) - \boldsymbol{K}_i\boldsymbol{v}_i\right] && \text{(E.35)}\\
&= \mathbb{E}\left[\boldsymbol{\epsilon}_{i-1} - \boldsymbol{K}_i\boldsymbol{H}_i\boldsymbol{\epsilon}_{i-1} - \boldsymbol{K}_i\boldsymbol{v}_i\right] && \text{(E.36)}\\
&= \left(\boldsymbol{I} - \boldsymbol{K}_i\boldsymbol{H}_i\right)\mathbb{E}\left[\boldsymbol{\epsilon}_{i-1}\right] - \boldsymbol{K}_i\mathbb{E}\left[\boldsymbol{v}_i\right] && \text{(E.37)}
\end{aligned}$$

Thus if $\mathbb{E}\left[\boldsymbol{v}_i\right] = 0$ and $\mathbb{E}\left[\boldsymbol{\epsilon}_{i-1}\right] = 0$ then $\mathbb{E}\left[\boldsymbol{\epsilon}_i\right] = 0$, which is achieved if $\boldsymbol{v}_i$ is zero mean and $\hat{\boldsymbol{\chi}}_0 = \mathbb{E}\left[\boldsymbol{\chi}\right]$. If this is satisfied then this is an unbiased estimator no matter what value is chosen for $\boldsymbol{K}_i$. Noting that $\mathbb{E}\left[\boldsymbol{\epsilon}_i\right] = 0$ the sum of the variances of the estimation errors at time k, can be written as

$$\begin{aligned}
\mathbb{E}\left[\left(\boldsymbol{\chi}_1 - \hat{\boldsymbol{\chi}_{i,1}}\right)^2\right] + ... + \mathbb{E}\left[\left(\boldsymbol{\chi}_n - \hat{\boldsymbol{\chi}_{i,n}}\right)^2\right] &= \mathbb{E}\left[\boldsymbol{\epsilon}_{i,1}^2 + ... + \boldsymbol{\epsilon}_{i,n}^2\right] && \text{(E.38)}\\
&= \mathbb{E}\left[\boldsymbol{\epsilon}_i^T\boldsymbol{\epsilon}_i\right] && \text{(E.39)}\\
&= \mathbb{E}\left[\mathrm{Tr}\left(\boldsymbol{\epsilon}_i\boldsymbol{\epsilon}_i^T\right)\right] && \text{(E.40)}\\
&= \mathrm{Tr}\left(\mathbb{E}\left[\boldsymbol{\epsilon}_i\boldsymbol{\epsilon}_i^T\right]\right) && \text{(E.41)}\\
&= \mathrm{Tr}\left(\mathrm{Cov}\left(\boldsymbol{\epsilon}_i\right)\right) && \text{(E.42)}
\end{aligned}$$

and the covariance of the estimation error at time k can be calculated by:

$$\begin{aligned}
\mathrm{Cov}\left(\boldsymbol{\epsilon}_i\right) &= \mathbb{E}\left[\boldsymbol{\epsilon}_i \boldsymbol{\epsilon}_i^T\right] && \text{(E.43)}\\
&= \mathbb{E}\left[\left(\boldsymbol{\chi}_i - \hat{\boldsymbol{\chi}}_i\right)\left(\boldsymbol{\chi}_i - \hat{\boldsymbol{\chi}}_i\right)^T\right] && \text{(E.44)}\\
&= \mathbb{E}\left[\left(\left(\boldsymbol{I} - \boldsymbol{K}_i \boldsymbol{H}_i\right)\boldsymbol{\epsilon}_{i-1} - \boldsymbol{K}_i \boldsymbol{v}_i\right)\left(\left(\boldsymbol{I} - \boldsymbol{K}_i \boldsymbol{H}_i\right)\boldsymbol{\epsilon}_{i-1} - \boldsymbol{K}_i \boldsymbol{v}_i\right)^T\right] && \text{(E.45)}\\
&= \mathbb{E}\left[\left(\left(\boldsymbol{I} - \boldsymbol{K}_i H\right)\boldsymbol{\epsilon}_{i-1} - \boldsymbol{K}_i \boldsymbol{v}_i\right)\left(\left(\left(\boldsymbol{I} - \boldsymbol{K}_i H\right)\boldsymbol{\epsilon}_{i-1}\right)^T - \left(\boldsymbol{K}_i \boldsymbol{v}_i\right)^T\right)\right] && \text{(E.46)}\\
&= \mathbb{E}\left[\left(\left(\boldsymbol{I} - \boldsymbol{K}_i \boldsymbol{H}_i\right)\boldsymbol{\epsilon}_{i-1}\right)\left(\left(\boldsymbol{I} - \boldsymbol{K}_i \boldsymbol{H}_i\right)\boldsymbol{\epsilon}_{i-1}\right)^T - \left(\left(\boldsymbol{I} - \boldsymbol{K}_i \boldsymbol{H}_i\right)\boldsymbol{\epsilon}_{i-1}\right)\left(\boldsymbol{K}_i \boldsymbol{v}_i\right)^T - \right. && \text{(E.47)}\\
&\quad \left. \boldsymbol{K}_i \boldsymbol{v}_i \left(\left(\boldsymbol{I} - \boldsymbol{K}_i H\right)\boldsymbol{\epsilon}_{i-1}\right)^T + \boldsymbol{K}_i \boldsymbol{v}_i \left(\boldsymbol{K}_i \boldsymbol{v}_i\right)^T\right] \\
&= \mathbb{E}\left[\left(\boldsymbol{I} - \boldsymbol{K}_i H\right)\boldsymbol{\epsilon}_{i-1}\boldsymbol{\epsilon}_{i-1}^T\left(\boldsymbol{I} - \boldsymbol{K}_i H\right)^T - \left(\boldsymbol{I} - \boldsymbol{K}_i H\right)\boldsymbol{\epsilon}_{i-1}\boldsymbol{v}_i^T \boldsymbol{K}_i^T - \right. && \text{(E.48)}\\
&\quad \left. \boldsymbol{K}_i \boldsymbol{v}_i \boldsymbol{\epsilon}_{i-1}^T \left(\boldsymbol{I} - \boldsymbol{K}_i H\right)^T + \boldsymbol{K}_i \boldsymbol{v}_i \boldsymbol{v}_i^T \boldsymbol{K}_i^T\right] \\
&= \left(\boldsymbol{I} - \boldsymbol{K}_i H\right)\mathbb{E}\left[\boldsymbol{\epsilon}_{i-1}\boldsymbol{\epsilon}_{i-1}^T\right]\left(\boldsymbol{I} - \boldsymbol{K}_i H\right)^T - \left(\boldsymbol{I} - \boldsymbol{K}_i H\right)\mathbb{E}\left[\boldsymbol{\epsilon}_{i-1}\boldsymbol{v}_i^T\right]\boldsymbol{K}_i^T - && \text{(E.49)}\\
&\quad \boldsymbol{K}_i \mathbb{E}\left[\boldsymbol{v}_i \boldsymbol{\epsilon}_{i-1}^T\right]\left(\boldsymbol{I} - \boldsymbol{K}_i H\right)^T + \boldsymbol{K}_i \mathbb{E}\left[\boldsymbol{v}_i \boldsymbol{v}_i^T\right]\boldsymbol{K}_i^T \\
&= \left(\boldsymbol{I} - \boldsymbol{K}_i H\right)\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\left(\boldsymbol{I} - \boldsymbol{K}_i H\right)^T - \left(\boldsymbol{I} - \boldsymbol{K}_i H\right)\mathbb{E}\left[\boldsymbol{\epsilon}_{i-1}\boldsymbol{v}_i^T\right]\boldsymbol{K}_i^T - && \text{(E.50)}\\
&\quad \boldsymbol{K}_i \mathbb{E}\left[\boldsymbol{v}_i \boldsymbol{\epsilon}_{i-1}^T\right]\left(\boldsymbol{I} - \boldsymbol{K}_i H\right)^T + \boldsymbol{K}_i \mathrm{Cov}\left(\boldsymbol{v}_i\right)\boldsymbol{K}_i^T
\end{aligned}$$

From (E.37) it should be evident that $\boldsymbol{\epsilon}_{i-1}$ is independent of $\boldsymbol{v}_i$, and still assuming $\boldsymbol{v}_i$ to have zero mean, that is $\mathbb{E}\left[\boldsymbol{v}_i\right] = 0$, the following is true

$$\mathbb{E}\left[\boldsymbol{\epsilon}_{i-1}\boldsymbol{v}_i^T\right] = \mathbb{E}\left[\boldsymbol{\epsilon}_{i-1}\right]\mathbb{E}\left[\boldsymbol{v}_i^T\right] = 0 \quad \text{(E.51)}$$

$$\mathbb{E}\left[\boldsymbol{v}_i \boldsymbol{\epsilon}_{i-1}^T\right] = \mathbb{E}\left[\boldsymbol{v}_i\right]\mathbb{E}\left[\boldsymbol{\epsilon}_{i-1}^T\right] = 0 \quad \text{(E.52)}$$

Therefore the covariance equation is simplified to:

$$\mathrm{Cov}\left(\boldsymbol{\epsilon}_i\right) = \left(\boldsymbol{I} - \boldsymbol{K}_i \boldsymbol{H}_i\right)\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\left(\boldsymbol{I} - \boldsymbol{K}_i \boldsymbol{H}_i\right)^T + \boldsymbol{K}_i \mathrm{Cov}\left(\boldsymbol{v}_i\right)\boldsymbol{K}_i^T \quad \text{(E.53)}$$

which is an recursive formula for the estimation error covariance, $\mathrm{Cov}\left(\boldsymbol{\epsilon}_i\right)$. In [30] a set of useful algebraic rules are derived. If B is a symmetric matrix the following derivative is easily found:

$$\frac{d\,\mathrm{Tr}\left(ABA^T\right)}{dA} = 2AB \quad \text{(E.54)}$$

And another useful rule when manipulating with the trace of matrices:

$$\mathrm{Tr}\left(A + B\right) = \mathrm{Tr}\left(A\right) + \mathrm{Tr}\left(B\right) \quad \text{(E.55)}$$

By noting that covariance matrices are symmetric the derivative of $\mathrm{Tr}\left(\mathrm{Cov}\left(\boldsymbol{\epsilon}_i\right)\right)$ becomes

$$\frac{d\,\mathrm{Tr}\left(\mathrm{Cov}\left(\boldsymbol{\epsilon}_i\right)\right)}{d\boldsymbol{K}_i} = \frac{d\,\mathrm{Tr}\left(\left(\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i\right)\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\left(\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i\right)^T+\boldsymbol{K}_i\mathrm{Cov}\left(\boldsymbol{v}_i\right)\boldsymbol{K}_i^T\right)}{d\boldsymbol{K}_i} \tag{E.56}$$

$$= \frac{d\left(\mathrm{Tr}\left(\left(\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i\right)\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\left(\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i\right)^T\right)+\mathrm{Tr}\left(\boldsymbol{K}_i\mathrm{Cov}\left(\boldsymbol{v}_i\right)\boldsymbol{K}_i^T\right)\right)}{d\boldsymbol{K}_i} \tag{E.57}$$

$$= \frac{d\,\mathrm{Tr}\left(\left(\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i\right)\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\left(\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i\right)^T\right)}{d\boldsymbol{K}_i}+\frac{d\,\mathrm{Tr}\left(\boldsymbol{K}_i\mathrm{Cov}\left(\boldsymbol{v}_i\right)\boldsymbol{K}_i^T\right)}{d\boldsymbol{K}_i} \tag{E.58}$$

$$= \frac{d\,\mathrm{Tr}\left(\left(\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i\right)\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\left(\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i\right)^T\right)}{d\boldsymbol{K}_i}+2\boldsymbol{K}_i\mathrm{Cov}\left(\boldsymbol{v}_i\right) \tag{E.59}$$

Defining $\boldsymbol{A}=\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i$ and using the chain-rule

$$\frac{d\,\mathrm{Tr}\left(\mathrm{Cov}\left(\boldsymbol{\epsilon}_i\right)\right)}{d\boldsymbol{K}_i} = \frac{d\,\mathrm{Tr}\left(\boldsymbol{A}\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{A}^T\right)}{d\boldsymbol{A}}\frac{d\boldsymbol{A}}{d\boldsymbol{K}_i}+2\boldsymbol{K}_i\mathrm{Cov}\left(\boldsymbol{v}_i\right) \tag{E.60}$$

$$= 2\boldsymbol{A}\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\frac{d\boldsymbol{A}}{d\boldsymbol{K}_i}+2\boldsymbol{K}_i\mathrm{Cov}\left(\boldsymbol{v}_i\right) \tag{E.61}$$

$$= 2\left(\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i\right)\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\frac{d\left(\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i\right)}{d\boldsymbol{K}_i}+2\boldsymbol{K}_i\mathrm{Cov}\left(\boldsymbol{v}_i\right) \tag{E.62}$$

The derivative of $\boldsymbol{A}$ is rewritten

$$\frac{d\left(\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i\right)}{d\boldsymbol{K}_i}=-\frac{d\left(\boldsymbol{K}_i\boldsymbol{H}_i\right)}{d\boldsymbol{K}_i}=-\frac{d\left(\boldsymbol{K}_i\boldsymbol{H}_i\right)^T}{d\boldsymbol{K}_i^T}=-\frac{d\left(\boldsymbol{H}_i^T\boldsymbol{K}_i^T\right)}{d\boldsymbol{K}_i^T}=-\boldsymbol{H}_i^T \tag{E.63}$$

Such that the derivative of $\mathrm{Tr}\left(\mathrm{Cov}\left(\boldsymbol{\epsilon}_i\right)\right)$ becomes

$$\frac{d\boldsymbol{A}}{d\boldsymbol{K}_i}=\frac{d\,\mathrm{Tr}\left(\mathrm{Cov}\left(\boldsymbol{\epsilon}_i\right)\right)}{d\boldsymbol{K}_i}=-2\left(\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i\right)\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}_i^T+2\boldsymbol{K}_i\mathrm{Cov}\left(\boldsymbol{v}_i\right) \tag{E.64}$$

The optimal value of $\boldsymbol{K}_i$ can now be found by equating (E.64) to zero:

$$0=-2\left(\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i\right)\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}_i^T+2\boldsymbol{K}_i\mathrm{Cov}\left(\boldsymbol{v}_i\right) \tag{E.65}$$

$$\boldsymbol{K}_i\mathrm{Cov}\left(\boldsymbol{v}_i\right)=\left(\boldsymbol{I}-\boldsymbol{K}_i\boldsymbol{H}_i\right)\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}_i^T \tag{E.66}$$

$$\boldsymbol{K}_i\mathrm{Cov}\left(\boldsymbol{v}_i\right)=\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}_i^T-\boldsymbol{K}_i\boldsymbol{H}_i\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}^k k,T \tag{E.67}$$

$$\boldsymbol{K}_i\mathrm{Cov}\left(\boldsymbol{v}_i\right)+\boldsymbol{K}_i\boldsymbol{H}_i\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}^k k,T=\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}_i^T \tag{E.68}$$

$$\boldsymbol{K}_i\left(\mathrm{Cov}\left(\boldsymbol{v}_i\right)+\boldsymbol{H}_i\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}_i^T\right)=\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}_i^T \tag{E.69}$$

$$\boldsymbol{K}_i=\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}_i^T\left(\mathrm{Cov}\left(\boldsymbol{v}_i\right)+\boldsymbol{H}_i\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}_i^T\right)^{-1} \tag{E.70}$$

The value of $\boldsymbol{K}_i$ that minimizes the sum of the variances of the estimation errors at time k given by (E.70) is often called the Kalman gain. Together (E.30), (E.53) and (E.70) constitute a recursive affine least square error estimator.

There are multiple equivalent formulas for the Kalman gain, $\boldsymbol{K}_i$, and the estimation error covariance, $\mathrm{Cov}\left(\boldsymbol{\epsilon}_i\right)$ [30]. These are summarized from (E.71) to (E.75).

$$\boldsymbol{K}_i = \mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}_i^T\left(\mathrm{Cov}\left(\boldsymbol{v}_i\right) + \boldsymbol{H}_i\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}_i^T\right)^{-1} \tag{E.71}$$

$$= \mathrm{Cov}\left(\boldsymbol{\epsilon}_i\right)\boldsymbol{H}_i^T\mathrm{Cov}\left(\boldsymbol{v}_i\right)^{-1} \tag{E.72}$$

$$\mathrm{Cov}\left(\boldsymbol{\epsilon}_i\right) = \left(\boldsymbol{I} - \boldsymbol{K}_i\boldsymbol{H}_i\right)\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\left(\boldsymbol{I} - \boldsymbol{K}_i\boldsymbol{H}_i\right)^T + \boldsymbol{K}_i\mathrm{Cov}\left(\boldsymbol{v}_i\right)\boldsymbol{K}_i^T \tag{E.73}$$

$$= \left(\boldsymbol{I} - \boldsymbol{K}_i\boldsymbol{H}_i\right)\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right) \tag{E.74}$$

$$= \left(\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)^{-1} + \boldsymbol{H}_i^T\mathrm{Cov}\left(\boldsymbol{v}_i\right)^{-1}\boldsymbol{H}_i\right)^{-1} \tag{E.75}$$

Even though (E.74) is simple and thus probably appealing to use, great care should be taken, since numerical computing can cause this expression for $\mathrm{Cov}\left(\boldsymbol{\epsilon}_i\right)$ to become non-positive definite. Instead (E.75) can be used which assures positive definiteness.

E.4.1 Summary

If an initial estimate of the vector to estimate $\hat{\boldsymbol{\chi}}_0 = \mathbb{E}\left[\boldsymbol{\chi}\right]$, the covariance of that estimate error, $\mathrm{Cov}\left(\boldsymbol{\epsilon}_0\right) = \mathbb{E}\left[\left(\boldsymbol{\chi} - \hat{\boldsymbol{\chi}}_0\right)\left(\boldsymbol{\chi} - \hat{\boldsymbol{\chi}}_0\right)^T\right] = \mathrm{Cov}\left(\boldsymbol{\chi}\right)$, and a system with a measurement model on the form

$$\boldsymbol{z}_i = \boldsymbol{H}_i\boldsymbol{\chi} + \boldsymbol{y}_i + \boldsymbol{v}_i \tag{E.76}$$

where the noise variable, $\boldsymbol{v}_i$, is distributed according to

$$\boldsymbol{v}_i \sim \left(0, \mathrm{Cov}\left(\boldsymbol{v}_i\right)\right) \tag{E.77}$$

are given, then an optimal linear estimate of the state vector, which minimizes the sum of the state estimate error variance, can be calculated recursively from a series of independent measurements, $\boldsymbol{z}_i$ for $i = 1, ..., I$, by the formulas given in (E.78) to (E.81).

$$\hat{\boldsymbol{z}}_i = \boldsymbol{H}_i\hat{\boldsymbol{\chi}}_{i-1} + \boldsymbol{y}_i \tag{E.78}$$

$$\boldsymbol{K}_i = \mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}_i^T\left(\mathrm{Cov}\left(\boldsymbol{v}_i\right) + \boldsymbol{H}_i\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right)\boldsymbol{H}_i^T\right)^{-1} \tag{E.79}$$

$$\hat{\boldsymbol{\chi}}_i = \hat{\boldsymbol{\chi}}_{i-1} + \boldsymbol{K}_i\left(\boldsymbol{z}_i - \hat{\boldsymbol{z}}_i\right) \tag{E.80}$$

$$\mathrm{Cov}\left(\boldsymbol{\epsilon}_i\right) = \left(\boldsymbol{I} - \boldsymbol{K}_i\boldsymbol{H}_i\right)\mathrm{Cov}\left(\boldsymbol{\epsilon}_{i-1}\right) \tag{E.81}$$

E.5 Discrete Extended Kalman Filter

The Extended Kalman Filter (EKF) is a filter that recursively estimates Gaussian representation of the state vector of a system based on measurements and a model of the system.

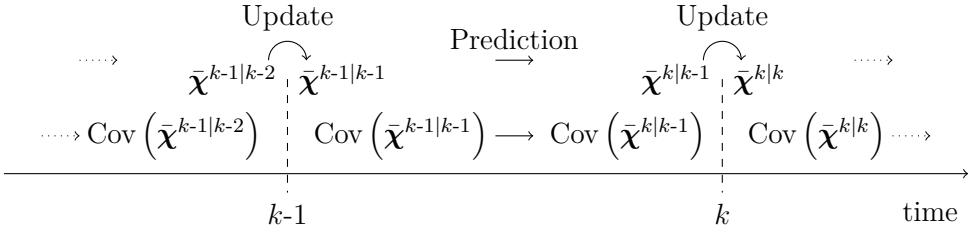


Figure E.2: Figure illustrating the steps in the Kalman filter estimation.

Figure E.2 shows how a recursive estimation is performed. Given an a posteriori estimate of the mean, $\bar{\boldsymbol{\chi}}^{k-1|k-1}$, and of the covariance matrix, $\text{Cov}\left(\boldsymbol{\chi}^{k-1|k-1}\right)$, of the state vector at time $k-1$, a prediction step is performed based on the system model. This results in an a priori estimate of the mean, $\bar{\boldsymbol{\chi}}^{k|k-1}$, and of the covariance matrix, $\text{Cov}\left(\boldsymbol{\chi}^{k|k-1}\right)$, of the state vector at time k. After this prediction step, an update step is performed taking the measurements of the state vector into account to get an a posteriori estimate of the mean, $\bar{\boldsymbol{\chi}}^{k|k}$, and of the covariance matrix, $\text{Cov}\left(\boldsymbol{\chi}^{k|k}\right)$, of the state vector at time k.

The Extended Kalman filter is, as its name implies, an extension to the linear Kalman filter. The linear Kalman filter can be derived directly from the Recursive Bayesian estimator presented in Section E.2 as an optimal estimator for a linear system with a linear measurement model including additive, independent and Gaussian noise [30]. However deriving the linear Kalman filter from results presented in Section E.3 and Section E.4 would in fact show that the linear Kalman filter is the optimal linear filter, if the system and measurement noise is additive, zero-mean, uncorrelated and white [30]. Thus the Extended Kalman filter, that works by linearising the system and measurement model, is a natural extension to the linear Kalman filter.
Usually the equations for the Extended Kalman filter are presented for a system with additive noise on both the system model and the measurement model, but to keep the description more general, this section considers a system model and a measurement model on the form given in (E.82) and (E.83) respectively [30].

$$\boldsymbol{\chi}^k = f\left(\boldsymbol{\chi}^{k-1}, \boldsymbol{u}^{k-1}, \boldsymbol{w}^{k-1}\right) \tag{E.82}$$

$$\boldsymbol{z}^k = h\left(\boldsymbol{\chi}^k, \boldsymbol{v}^k\right) \tag{E.83}$$

Where:

$f(\bullet)$	is the system model
$h(\bullet)$	is the measurement model
$\boldsymbol{\chi}$	is the state vector
$\boldsymbol{u}$	is a known input to the system
$\boldsymbol{w}$	is a noise variable called the process noise
$\boldsymbol{v}$	is a noise variable called the measurement noise

$\boldsymbol{w}^k$ and $\boldsymbol{v}^k$ are assumed to be uncorrelated zero-mean white noise variables with known covariance, $\text{Cov}\left(\boldsymbol{w}^k\right)$ and $\text{Cov}\left(\boldsymbol{v}^k\right)$, meaning that they can be distributed according to any distribution that satisfies the following properties

$$\boldsymbol{w}^k \sim \left(0, \text{Cov}\left(\boldsymbol{w}^k\right)\right) \tag{E.84}$$

$$\boldsymbol{v}^k \sim \left(0, \text{Cov}\left(\boldsymbol{v}^k\right)\right) \tag{E.85}$$

$$\mathbb{E}\left[\boldsymbol{w}^k \left(\boldsymbol{w}^j\right)^T\right] = \text{Cov}\left(\boldsymbol{w}^k\right) \delta\left(k-j\right) \tag{E.86}$$

$$\mathbb{E}\left[\boldsymbol{v}^k \left(\boldsymbol{v}^j\right)^T\right] = \text{Cov}\left(\boldsymbol{v}^k\right) \delta\left(k-j\right) \tag{E.87}$$

$$\mathbb{E}\left[\boldsymbol{v}^k \left(\boldsymbol{w}^j\right)^T\right] = 0 \tag{E.88}$$

Where:

$\delta(\bullet)$ is the the Kronecker delta function; that is $\delta(k-j)=1$ for $k=j$ and $\delta(k-j)=0$ for $k\neq j$

Performing a Taylor series expansion on $f(\bullet)$ around $\boldsymbol{\chi}^{k-1}=\bar{\boldsymbol{\chi}}^{k-1|k-1}$ and $\boldsymbol{w}^{k-1}=0$, (E.93) is obtained.

$$\boldsymbol{\chi}^{k}=f\left(\boldsymbol{\chi}^{k-1},\boldsymbol{u}^{k-1},\boldsymbol{w}^{k-1}\right) \tag{E.89}$$

$$\approx f\left(\bar{\boldsymbol{\chi}}^{k-1|k-1},\boldsymbol{u}^{k-1},0\right)+\left.\frac{\partial f}{\partial\boldsymbol{\chi}}\right|_{\bar{\boldsymbol{\chi}}_{k-1|k-1},u_{k-1}}\left(\boldsymbol{\chi}^{k-1}-\bar{\boldsymbol{\chi}}^{k-1|k-1}\right)+\left.\frac{\partial f}{\partial\boldsymbol{w}}\right|_{\bar{\boldsymbol{\chi}}_{k-1|k-1},u_{k-1}}\left(\boldsymbol{w}^{k-1}-0\right) \tag{E.90}$$

$$=f\left(\bar{\boldsymbol{\chi}}^{k-1|k-1},\boldsymbol{u}^{k-1},0\right)+\boldsymbol{F}^{k}\left(\boldsymbol{\chi}^{k-1}-\bar{\boldsymbol{\chi}}^{k-1|k-1}\right)+\boldsymbol{L}^{k}\boldsymbol{w}^{k-1} \tag{E.91}$$

$$=\boldsymbol{F}^{k}\boldsymbol{\chi}^{k-1}+\left(f\left(\bar{\boldsymbol{\chi}}^{k-1|k-1},\boldsymbol{u}^{k-1},0\right)-\boldsymbol{F}^{k}\bar{\boldsymbol{\chi}}^{k-1|k-1}\right)+\boldsymbol{L}^{k}\boldsymbol{w}^{k-1} \tag{E.92}$$

$$=\boldsymbol{F}^{k}\boldsymbol{\chi}^{k-1}+\tilde{\boldsymbol{u}}^{k-1}+\tilde{\boldsymbol{w}}^{k-1} \tag{E.93}$$

All the variables in

$$\tilde{\boldsymbol{u}}^{k-1}=f\left(\bar{\boldsymbol{\chi}}^{k-1|k-1},\boldsymbol{u}^{k-1},0\right)-\boldsymbol{F}^{k}\bar{\boldsymbol{\chi}}^{k-1|k-1} \tag{E.94}$$

are known, and thus can be calculated directly. Furthermore, the expectation and covariance of $\tilde{\boldsymbol{w}}^{k}$ are given by (E.95) and (E.96).

$$\mathbb{E}\left[\tilde{\boldsymbol{w}}^{k}\right]=\mathbb{E}\left[\boldsymbol{L}^{k}\boldsymbol{w}^{k-1}\right]=\boldsymbol{L}^{k}E\left[\boldsymbol{w}^{k-1}\right]=0 \tag{E.95}$$

$$\mathrm{Cov}\left(\tilde{\boldsymbol{w}}^{k-1}\right)=\mathbb{E}\left[\tilde{\boldsymbol{w}}^{k-1}\tilde{\boldsymbol{w}}^{k-1,T}\right]=\mathbb{E}\left[\boldsymbol{L}^{k}\boldsymbol{w}^{k-1}\left(\boldsymbol{L}^{k}\boldsymbol{w}^{k-1}\right)^{T}\right]=\mathbb{E}\left[\boldsymbol{L}^{k}\boldsymbol{w}^{k-1}\boldsymbol{w}^{k-1,T}\boldsymbol{L}^{k,T}\right]=\boldsymbol{L}^{k}\mathrm{Cov}\left(\boldsymbol{w}^{k-1}\right)\boldsymbol{L}^{k,T} \tag{E.96}$$

And thus

$$\tilde{\boldsymbol{w}}^{k-1}\sim\left(0,\,\boldsymbol{L}^{k}\mathrm{Cov}\left(\boldsymbol{w}^{k-1}\right)\boldsymbol{L}^{k,T}\right) \tag{E.97}$$

The system described in (E.93) is linear which allows a prediction step to be performed by using the equations in Section E.3 with the a posteriori estimate of the mean, $\bar{\boldsymbol{\chi}}^{k-1|k-1}$, and covariance matrix, $C\left(\boldsymbol{\chi}^{k-1|k-1}\right)$, of the state vector at time $k-1$. The equations of the prediction step become

$$\bar{\boldsymbol{\chi}}^{k|k-1}=\boldsymbol{F}^{k}\bar{\boldsymbol{\chi}}^{k-1|k-1}+\tilde{\boldsymbol{u}}^{k-1} \tag{E.98}$$

$$=\boldsymbol{F}^{k}\bar{\boldsymbol{\chi}}^{k-1|k-1}+f\left(\hat{\boldsymbol{\chi}}^{k-1|k-1},\boldsymbol{u}^{k-1},0\right)-\boldsymbol{F}^{k}\bar{\boldsymbol{\chi}}^{k-1|k-1} \tag{E.99}$$

$$=f\left(\bar{\boldsymbol{\chi}}^{k-1|k-1},\boldsymbol{u}^{k-1},0\right) \tag{E.100}$$

$$\mathrm{Cov}\left(\boldsymbol{\chi}^{k|k-1}\right)=\boldsymbol{F}^{k}\mathrm{Cov}\left(\boldsymbol{\chi}^{k-1|k-1}\right)\boldsymbol{F}^{k,T}+\mathrm{Cov}\left(\tilde{\boldsymbol{w}}^{k-1}\right) \tag{E.101}$$

$$=\boldsymbol{F}^{k}\mathrm{Cov}\left(\boldsymbol{\chi}^{k-1|k-1}\right)\boldsymbol{F}^{k,T}+\boldsymbol{L}^{k}\mathrm{Cov}\left(\boldsymbol{w}^{k-1}\right)\boldsymbol{L}^{k,T} \tag{E.102}$$

To obtain the equations for the update step, a Taylor series expansion is performed on $h(\bullet)$ around $\boldsymbol{\chi}^{k}=\bar{\boldsymbol{\chi}}^{k|k-1}$ and $\boldsymbol{v}^{k}=0$ as shown in (E.103) through (E.107).

$$\boldsymbol{z}^k = h\left(\boldsymbol{\chi}^k, \boldsymbol{v}^k\right) \tag{E.103}$$

$$\approx h\left(\bar{\boldsymbol{\chi}}^{k|k-1}, 0\right) + \left.\frac{\partial h}{\partial \boldsymbol{\chi}}\right|_{\bar{\boldsymbol{\chi}}^{k|k-1}} \left(\boldsymbol{\chi}^k - \bar{\boldsymbol{\chi}}^{k|k-1}\right) + \left.\frac{\partial h}{\partial \boldsymbol{v}}\right|_{\bar{\boldsymbol{\chi}}^{k|k-1}} \left(\boldsymbol{v}^k - 0\right) \tag{E.104}$$

$$= h\left(\bar{\boldsymbol{\chi}}^{k|k-1}, 0\right) + \boldsymbol{H}^k \left(\boldsymbol{\chi}^k - \bar{\boldsymbol{\chi}}^{k|k-1}\right) + \boldsymbol{M}^k \boldsymbol{v}^k \tag{E.105}$$

$$= \boldsymbol{H}^k \boldsymbol{\chi}^k + \left(h\left(\bar{\boldsymbol{\chi}}^{k|k-1}, 0\right) - \boldsymbol{H}^k \bar{\boldsymbol{\chi}}^{k|k-1}\right) + \boldsymbol{M}^k \boldsymbol{v}^k \tag{E.106}$$

$$= \boldsymbol{H}^k \boldsymbol{\chi}^k + \boldsymbol{y}^k + \tilde{\boldsymbol{v}}^k \tag{E.107}$$

All the variables in

$$\boldsymbol{y}^k = h\left(\bar{\boldsymbol{\chi}}^{k|k-1}, 0\right) - \boldsymbol{H}^k \bar{\boldsymbol{\chi}}^{k|k-1} \tag{E.108}$$

are known and can thus be calculated directly. Furthermore, as done for $\tilde{\boldsymbol{w}}^k$ in (E.95) and (E.96), it can be shown that

$$\tilde{\boldsymbol{v}}^k \sim \left(0, \boldsymbol{M}^k \mathrm{Cov}\left(\boldsymbol{v}^k\right) \boldsymbol{M}^{k,T}\right) \tag{E.109}$$

Section E.4 describes an optimal linear recursive least square estimator that minimizes the sum of variances of the estimation error. Since (E.107) is a linear approximation of the measurement model, the estimator described in Section E.4 can be used to incorporate measurements of the states into the update step of the Extended Kalman filter. Doing so, the measurement estimate becomes

$$\hat{\boldsymbol{z}}^k = \boldsymbol{H}^k \bar{\boldsymbol{\chi}}^{k|k-1} + \boldsymbol{y}^k \tag{E.110}$$

$$= \boldsymbol{H}^k \bar{\boldsymbol{\chi}}^{k|k-1} + h\left(\bar{\boldsymbol{\chi}}^{k|k-1}, 0\right) - \boldsymbol{H}^k \bar{\boldsymbol{\chi}}^{k|k-1} \tag{E.111}$$

$$= h\left(\bar{\boldsymbol{\chi}}^{k|k-1}, 0\right) \tag{E.112}$$

From Section E.4.1 it is known that the Kalman gain can be calculated by:

$$\boldsymbol{K}^k = \mathrm{Cov}\left(\boldsymbol{\epsilon}^{k-1}\right) \boldsymbol{H}^{k,T} \left(\mathrm{Cov}\left(\tilde{\boldsymbol{v}}^k\right) + \boldsymbol{H}^k \mathrm{Cov}\left(\boldsymbol{\epsilon}^{k-1}\right) \boldsymbol{H}^{k,T}\right)^{-1} \tag{E.113}$$

Well knowing that the covariance of the state estimate error, $\mathrm{Cov}\left(\boldsymbol{\epsilon}_0\right)$, is the same as the covariance of the state estimate, $\mathrm{Cov}\left(\boldsymbol{\chi}\right)$, the covariance of the a priori is used when calculating the Kalman gain

$$\boldsymbol{K}^k = \mathrm{Cov}\left(\boldsymbol{\chi}^{k|k-1}\right) \boldsymbol{H}^{k,T} \left(\boldsymbol{M}^k \mathrm{Cov}\left(\boldsymbol{v}^k\right) \boldsymbol{M}^{k,T} + \boldsymbol{H}^k \mathrm{Cov}\left(\boldsymbol{\chi}^{k|k-1}\right) \boldsymbol{H}^{k,T}\right)^{-1} \tag{E.114}$$

and the update equations hereby become

$$\bar{\boldsymbol{\chi}}^{k|k} = \bar{\boldsymbol{\chi}}^{k|k-1} + \boldsymbol{K}^k \left(\boldsymbol{z}^k - \hat{\boldsymbol{z}}^k\right) \tag{E.115}$$

$$= \bar{\boldsymbol{\chi}}^{k|k-1} + \boldsymbol{K}^k \left(\boldsymbol{z}^k - h\left(\bar{\boldsymbol{\chi}}^{k|k-1}, 0\right)\right) \tag{E.116}$$

$$\mathrm{Cov}\left(\boldsymbol{\chi}^{k|k}\right) = \left(\boldsymbol{I} - \boldsymbol{K}^k \boldsymbol{H}^k k\right) \mathrm{Cov}\left(\boldsymbol{\epsilon}_0\right) \tag{E.117}$$

$$= \left(\boldsymbol{I} - \boldsymbol{K}^k \boldsymbol{H}^k k\right) \mathrm{Cov}\left(\boldsymbol{\chi}^{k|k-1}\right) \tag{E.118}$$

E.5.1 Summary

For a given system, assume a system model and measurement model on the form given in (E.119) and (E.120) respectively.

$$\boldsymbol{\chi}^k = f\left(\boldsymbol{\chi}^{k-1}, \boldsymbol{u}^{k-1}, \boldsymbol{w}^{k-1}\right) \tag{E.119}$$

$$\boldsymbol{z}^k = h\left(\boldsymbol{\chi}^k, \boldsymbol{v}^k\right) \tag{E.120}$$

Furthermore assume that

$$\boldsymbol{w}^k \sim \left(0, \mathrm{Cov}\left(\boldsymbol{w}^k\right)\right) \tag{E.121}$$

$$\boldsymbol{v}^k \sim \left(0, \mathrm{Cov}\left(\boldsymbol{v}^k\right)\right) \tag{E.122}$$

$$\mathbb{E}\left[\boldsymbol{w}^k \left(\boldsymbol{w}^j\right)^T\right] = \mathrm{Cov}\left(\boldsymbol{w}^k\right) \delta\left(k-j\right) \tag{E.123}$$

$$\mathbb{E}\left[\boldsymbol{v}^k \left(\boldsymbol{v}^j\right)^T\right] = \mathrm{Cov}\left(\boldsymbol{v}^k\right) \delta\left(k-j\right) \tag{E.124}$$

$$\mathbb{E}\left[\boldsymbol{v}^k \left(\boldsymbol{w}^j\right)^T\right] = 0 \tag{E.125}$$

Given these assumptions the state vector of the system at time k, $\boldsymbol{\chi}^k$, can be estimated recursively with an Extended Kalman Filter with the formulas given in (E.126) to (E.134).

Prediction Step:

$$\bar{\boldsymbol{\chi}}^{k|k-1} = f\left(\bar{\boldsymbol{\chi}}^{k-1|k-1}, \boldsymbol{u}^{k-1}, 0\right) \tag{E.126}$$

$$\mathrm{Cov}\left(\boldsymbol{\chi}^{k|k-1}\right) = \boldsymbol{F}^k \mathrm{Cov}\left(\boldsymbol{\chi}^{k-1|k-1}\right) \boldsymbol{F}^{k,T} + \boldsymbol{L}^k \mathrm{Cov}\left(\boldsymbol{w}^{k-1}\right) \boldsymbol{L}^{k,T} \tag{E.127}$$

where

$$\boldsymbol{F}^k = \left.\frac{\partial f}{\partial \boldsymbol{\chi}}\right|_{\bar{\boldsymbol{\chi}}_{k-1|k-1}, u_{k-1}} \tag{E.128}$$

$$\boldsymbol{L}^k = \left.\frac{\partial f}{\partial \boldsymbol{w}}\right|_{\bar{\boldsymbol{\chi}}_{k-1|k-1}, u_{k-1}} \tag{E.129}$$

Update Step:

$$\boldsymbol{K}^k = \mathrm{Cov}\left(\boldsymbol{\chi}^{k|k-1}\right) \boldsymbol{H}^{k,T} \left(\boldsymbol{M}^k \mathrm{Cov}\left(\boldsymbol{v}^k\right) \boldsymbol{M}^{k,T} + \boldsymbol{H}^k \mathrm{Cov}\left(\boldsymbol{\chi}^{k|k-1}\right) \boldsymbol{H}^{k,T}\right)^{-1} \tag{E.130}$$

$$\bar{\boldsymbol{\chi}}^{k|k} = \bar{\boldsymbol{\chi}}^{k|k-1} + \boldsymbol{K}^k \left(\boldsymbol{z}^k - h\left(\bar{\boldsymbol{\chi}}^{k|k-1}, 0\right)\right) \tag{E.131}$$

$$\mathrm{Cov}\left(\boldsymbol{\chi}^{k|k}\right) = \left(\boldsymbol{I} - \boldsymbol{K}^k \boldsymbol{H}^k k\right) \mathrm{Cov}\left(\boldsymbol{\chi}^{k|k-1}\right) \tag{E.132}$$

where

$$\boldsymbol{H}^k = \left.\frac{\partial h}{\partial \boldsymbol{\chi}}\right|_{\bar{\boldsymbol{\chi}}^{k|k-1}} \tag{E.133}$$

$$\boldsymbol{M}^k = \left.\frac{\partial h}{\partial \boldsymbol{v}}\right|_{\bar{\boldsymbol{\chi}}^{k|k-1}} \tag{E.134}$$

E.6 Sequential Extended Kalman Filter

In the normal EKF, if multiple measurements are available at a single time step, $\boldsymbol{z}_1^k, ..., \boldsymbol{z}_r^k$, they have to be incorporated by stacking them in one large measurement vector, $\boldsymbol{z}^k = \left[\boldsymbol{z}_1^k, ..., \boldsymbol{z}_r^k\right]^T$. If there are r measurements at time k, then the calculation of the kalman gain $\boldsymbol{K}^k$, given by (E.130), requires the inversion of an $r \times r$ matrix. If many measurements are available this will degrade the performance of the EKF due to computational limitations. However, if the measurements are independent, that is

$$\text{Cov}\left(\boldsymbol{z}^k\right) = \text{diag}\left(\text{Cov}\left(\boldsymbol{z}_1\right), ..., \text{Cov}\left(\boldsymbol{z}_r\right)\right) \tag{E.135}$$

then every measurement model for the available measurements will be on the form

$$\boldsymbol{z}_i^k = h_i\left(\boldsymbol{\chi}^k, \boldsymbol{v}_i^k\right) \tag{E.136}$$

Where:

h_i	is the measurement model related to the i'th measurement
$\boldsymbol{v}_i$	is the noise variable related to the i'th measurement

which can be linearised as in (E.107), yielding

$$\boldsymbol{z}_i^k \approx \boldsymbol{H}_i^k \boldsymbol{\chi}^k + \boldsymbol{y}_i^k + \tilde{\boldsymbol{v}}_i^k \tag{E.137}$$

where

$$\boldsymbol{y}_i^k = h_i\left(\bar{\boldsymbol{\chi}}^{k|k-1}, 0\right) - \boldsymbol{H}_i^k \bar{\boldsymbol{\chi}}^{k|k-1} \tag{E.138}$$

$$\tilde{\boldsymbol{v}}_i^k \sim \left(0, \boldsymbol{M}_i^k \text{Cov}\left(\boldsymbol{v}_i^k\right) \boldsymbol{M}_i^{k,T}\right) \tag{E.139}$$

$$\boldsymbol{H}_i^k = \left.\frac{\partial h_i}{\partial \boldsymbol{\chi}}\right|_{\bar{\boldsymbol{\chi}}_i^{k|k}} \tag{E.140}$$

$$\boldsymbol{M}_i^k = \left.\frac{\partial h_i}{\partial \boldsymbol{v}_i}\right|_{\bar{\boldsymbol{\chi}}_i^{k|k}} \tag{E.141}$$

with

$$\bar{\boldsymbol{\chi}}_0^{k|k} = \bar{\boldsymbol{\chi}}^{k|k-1} \tag{E.142}$$

These measurement models fulfil the requirements for the optimal affine recursive least squares estimator described in Section E.4. Thus, the measurements can be processed sequentially in the update step of the Extended Kalman filter by using the optimal affine recursive least squares estimator described in Section E.4.

E.6.1 Summary

For a given system, assume a system model and measurement model on the form given in (E.143) and (E.144) respectively.

$$\boldsymbol{\chi}^k = f\left(\boldsymbol{\chi}^{k-1}, \boldsymbol{u}^{k-1}, \boldsymbol{w}^{k-1}\right) \tag{E.143}$$

$$\boldsymbol{z}_i^k = h_i\left(\boldsymbol{\chi}^k, \boldsymbol{v}_i^k\right) \tag{E.144}$$

Where $\boldsymbol{z}_i^k$, for $i = 1, ..., r$, are independent measurements. Furthermore assume that

$$\boldsymbol{w}^k \sim \left(0, \text{Cov}\left(\boldsymbol{w}^k\right)\right) \tag{E.145}$$

$$\boldsymbol{v}_i^k \sim \left(0, \text{Cov}\left(\boldsymbol{v}_v i^k\right)\right) \tag{E.146}$$

$$\mathbb{E}\left[\boldsymbol{w}^k \left(\boldsymbol{w}^j\right)^T\right] = \text{Cov}\left(\boldsymbol{w}^k\right) \delta\left(k - j\right) \tag{E.147}$$

$$\mathbb{E}\left[\boldsymbol{v}_i^k \left(\boldsymbol{v}_v j^k\right)^T\right] = \text{Cov}\left(\boldsymbol{v}^k\right) \delta\left(k - j\right) \quad \text{for} \quad j = i \tag{E.148}$$

$$\mathbb{E}\left[\boldsymbol{v}_i^k \left(\boldsymbol{v}_v j^k\right)^T\right] = 0 \quad \text{for} \quad j \neq i \tag{E.149}$$

$$\mathbb{E}\left[\boldsymbol{v}_i^k \left(\boldsymbol{w}^j\right)^T\right] = 0 \tag{E.150}$$

Given these assumptions, the state vector of the system at time k, $\boldsymbol{\chi}^k$, can be estimated recursively with the Extended Kalman filter with the formulas given from (E.151) to (E.161).

Prediction Step:

$$\bar{\boldsymbol{\chi}}^{k|k-1} = f\left(\bar{\boldsymbol{\chi}}^{k-1|k-1}, \boldsymbol{u}^{k-1}, 0\right) \tag{E.151}$$

$$\text{Cov}\left(\boldsymbol{\chi}^{k|k-1}\right) = \boldsymbol{F}^k \text{Cov}\left(\boldsymbol{\chi}^{k-1|k-1}\right) \boldsymbol{F}^{k,T} + \boldsymbol{L}^k \text{Cov}\left(\boldsymbol{w}^{k-1}\right) \boldsymbol{L}^{k,T} \tag{E.152}$$

where

$$\boldsymbol{F}^k = \left.\frac{\partial f}{\partial \boldsymbol{\chi}}\right|_{\bar{\boldsymbol{\chi}}_{k-1|k-1}, \boldsymbol{u}_{k-1}} \tag{E.153}$$

$$\boldsymbol{L}^k = \left.\frac{\partial f}{\partial \boldsymbol{w}}\right|_{\bar{\boldsymbol{\chi}}_{k-1|k-1}, \boldsymbol{u}_{k-1}} \tag{E.154}$$

Update Step: Start by defining $\bar{\boldsymbol{\chi}}_0^{k|k} = \bar{\boldsymbol{\chi}}^{k|k-1}$ and $\text{Cov}\left(\boldsymbol{\chi}_i^{k|k}\right) = \text{Cov}\left(\boldsymbol{\chi}^{k|k-1}\right)$. Then, for $i = 1, ..., r$

$$\boldsymbol{K}_i^k = \text{Cov}\left(\boldsymbol{\chi}_i^{k|k}\right) \boldsymbol{H}_i^{k,T} \left(\boldsymbol{M}_i^k \text{Cov}\left(\boldsymbol{v}_i^k\right) \boldsymbol{M}_i^{k,T} + \boldsymbol{H}_i^k \text{Cov}\left(\boldsymbol{\chi}_i^{k|k}\right) \boldsymbol{H}_i^{k,T}\right)^{-1} \tag{E.155}$$

$$\bar{\boldsymbol{\chi}}_i^{k|k} = \bar{\boldsymbol{\chi}}_i^{k|k} + \boldsymbol{K}_i^k \left(\boldsymbol{z}_i^k - h_i\left(\bar{\boldsymbol{\chi}}_i^{k|k}, 0\right)\right) \tag{E.156}$$

$$\text{Cov}\left(\boldsymbol{\chi}_i^{k|k}\right) = \left(\boldsymbol{I} - \boldsymbol{K}_i^k \boldsymbol{H}_i^k\right) \text{Cov}\left(\boldsymbol{\chi}_i^{k|k}\right) \tag{E.157}$$

where

$$\boldsymbol{H}_i^k = \left.\frac{\partial h_i}{\partial \boldsymbol{\chi}}\right|_{\bar{\boldsymbol{\chi}}_i^{k|k}} \tag{E.158}$$

$$\boldsymbol{M}_i^k = \left.\frac{\partial h_i}{\partial \boldsymbol{v}_i}\right|_{\bar{\boldsymbol{\chi}}_i^{k|k}} \tag{E.159}$$

Finally, the a posteriori at time k, being the estimates of the mean and covariance, are given from

$$\bar{\boldsymbol{\chi}}^{k|k} = \bar{\boldsymbol{\chi}}_r^{k|k} \tag{E.160}$$

$$\text{Cov}\left(\boldsymbol{\chi}^{k|k}\right) = \text{Cov}\left(\boldsymbol{\chi}_r^{k|k}\right) \tag{E.161}$$

E.7 Importance Sampling

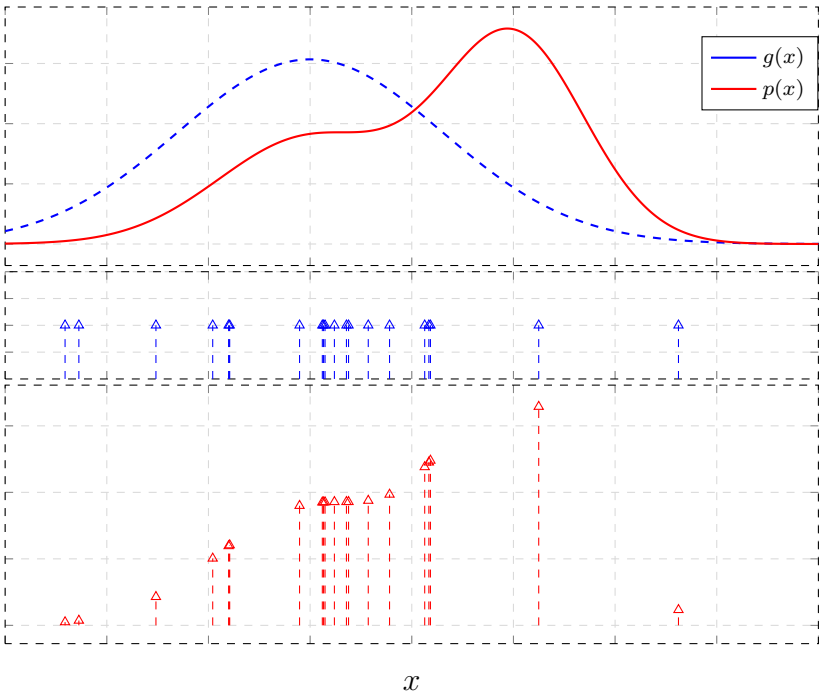


Figure E.3: Illustration of importance Sampling. Here $p\,(x)$ is approximated by drawing samples from $g\,(x)$ shown with blue arrows. And then weighted to approximate $p\,(x)$ shown with red arrows.

Importance Sampling is a general technique make a discrete approximation of a distribution, $p(x)$, called the target distribution, based on samples generated from another distribution, $g(x)$, called the proposal distribution. The expectation of any function, $f(x)$, of a continues random variable, X, with a PDF, $p\,(x)$, can be written as [28]

$$\mathbb{E}_p\left[f\,(X)\right] = \int_{-\infty}^{\infty} f\,(x)\,p\,(x)\;dx \tag{E.162}$$

Where:

$p\,(x)$ is the PDF of the random variable X.

The subscript on the expectation, eg. $\mathbb{E}_p$, is used to emphasize which PDF the random variable of which the expectation is being found, is distributed according to. Define the indicator function, $I_A\,(x)$, on the set A as

$$I_A\,(x) = \begin{cases} 1 & x \in A \\ 0 & \text{otherwise} \end{cases} \tag{E.163}$$

It now follows that

$$\mathbb{E}_p\left[I_A\left(X\right)\right] = \int_{-\infty}^{\infty} I_A\left(x\right) p\left(x\right)\, dx \tag{E.164}$$
$$= \int_A 1 \cdot p\left(x\right)\, dx \tag{E.165}$$
$$= \int_A p\left(x\right)\, dx \tag{E.166}$$
$$= P\left(A\right) \tag{E.167}$$

The expectation of the indicator function, $I_A\left(x\right)$, of a random variable, X, on the set A is equal to the probability of the set, also called the event, A. With these definitions it can now be shown how to approximate the target distribution, $p(x)$, by drawing samples from the proposal distribution, $g(x)$. Start by assuming that

$$p\left(x\right) > 0 \qquad \Rightarrow \qquad g\left(x\right) > 0 \tag{E.168}$$

Meaning that the proposal distribution should not be zero in a region of the state space, where the target distribution is non-zero.
Now for any set A the following is true

$$\mathbb{E}_p\left[I_A\left(X\right)\right] = \int_{-\infty}^{\infty} I_A\left(x\right) p\left(x\right)\, dx \tag{E.169}$$
$$= \int_{-\infty}^{\infty} \frac{g\left(x\right)}{g\left(x\right)} I_A\left(x\right) p\left(x\right)\, dx \tag{E.170}$$
$$= \int_{-\infty}^{\infty} \frac{p\left(x\right)}{g\left(x\right)} I_A\left(x\right) g\left(x\right)\, dx \tag{E.171}$$
$$= \int_{-\infty}^{\infty} q\left(x\right) I_A\left(x\right) g\left(x\right)\, dx \tag{E.172}$$
$$= E_g\left[q\left(X\right) I_A\left(X\right)\right] \tag{E.173}$$

Where:
$q\left(x\right)$ is called the importance weight function.

From this it follows that

$$P\left(A\right) = \int_A p\left(x\right)\, dx = \int_A q\left(x\right) g\left(x\right)\, dx = \mathbb{E}_g\left[q\left(X\right) I_A\left(X\right)\right] \tag{E.174}$$

Now assume that N samples drawn IID according to the PDF $g\left(x\right)$ are available, i.e. $x_1, ..., x_N \sim g$. Then an estimate of $\mathbb{E}_g\left[q\left(X\right) I_A\left(X\right)\right]$ can be calculated with the sample mean, and thus

$$\int_A p\left(x\right)\, dx = \mathbb{E}_g\left[q\left(X\right) I_A\left(X\right)\right] \approx \frac{1}{N} \sum_{i=1}^{N} q\left(x_i\right) I_A\left(x_i\right) \tag{E.175}$$

From the law of large numbers it furthermore follows that the sample mean converges to the true mean [28]. Thus,

$$\lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} q\left(x_i\right) I_A\left(x_i\right) = \mathbb{E}_g\left[q\left(X\right) I_A\left(X\right)\right] = \int_A p\left(x\right)\, dx = P\left(A\right) \tag{E.176}$$

Notice that in these derivations no assumptions are made about the set A and thus, the above holds for any set A. Therefore these derivations show that given N samples, $x_1, ..., x_N$, drawn

IID according to an arbitrary proposal distribution, $g(x)$, fulfilling (E.168), it is possible to approximate the target distribution, $p(x)$, by weighting the samples, $x_1, ..., x_N$, based on the importance weight function

$$q(x_i) = \frac{p(x_i)}{g(x_i)} \tag{E.177}$$

This requires that $q(x_i)$ can be evaluated for all possible x_i.

E.7.1 Self-normalized Importance Sampling

Sometimes it is only possible to compute an unnormalized version of $q(x)$. That is

$$q(x) = \eta \cdot \tilde{q}(x) = \frac{\eta_1 \cdot \tilde{p}(x)}{\eta_2 \cdot \tilde{g}(x)} \tag{E.178}$$

Where:

$\tilde{q}$	is the unnormalized version of $q(x)$
$\tilde{p}$	is the unnormalized version of $p(x)$
$\tilde{g}$	is the unnormalized version of $g(x)$
η_i	is an unknown normalizing constants where $\eta_i > 0$.

In words this means that q, p and g are known up to some unknown constants, η, η_1 and η_2. Now notice that since $q(x)$ and $g(x)$ are PDF's, the following is true

$$\int_{-\infty}^{\infty} q(x)\, g(x)\, dx = 1 \tag{E.179}$$

Still assuming the condition in (E.168) it is therefore possible to rewrite (E.173) to

$$\begin{aligned}
\mathbb{E}_g\left[q(X)\, I_A(X)\right] &= \int_{-\infty}^{\infty} q(x)\, I_A(x)\, g(x)\, dx && \text{(E.180)}\\
&= \frac{\int_{-\infty}^{\infty} q(x)\, I_A(x)\, g(x)\, dx}{\int_{-\infty}^{\infty} q(x)\, g(x)\, dx} && \text{(E.181)}\\
&= \frac{\int_{-\infty}^{\infty} \eta \cdot \tilde{q}(x)\, I_A(x)\, g(x)\, dx}{\int_{-\infty}^{\infty} \eta \cdot \tilde{q}(x)\, g(x)\, dx} && \text{(E.182)}\\
&= \frac{\eta \cdot \int_{-\infty}^{\infty} \tilde{q}(x)\, I_A(x)\, g(x)\, dx}{\eta \cdot \int_{-\infty}^{\infty} \tilde{q}(x)\, g(x)\, dx} && \text{(E.183)}\\
&= \frac{\int_{-\infty}^{\infty} \tilde{q}(x)\, I_A(x)\, g(x)\, dx}{\int_{-\infty}^{\infty} \tilde{q}(x)\, g(x)\, dx} && \text{(E.184)}\\
&= \frac{\mathbb{E}_g\left[\tilde{q}(X)\, I_A(X)\right]}{\mathbb{E}_g\left[\tilde{q}(X)\right]} && \text{(E.185)}
\end{aligned}$$

Therefore it must be concluded that

$$P(A) = \int_A p(x)\, dx = \mathbb{E}_g\left[q(X)\, I_A(X)\right] = \frac{\mathbb{E}_g\left[\tilde{q}(X)\, I_A(X)\right]}{\mathbb{E}_g\left[\tilde{q}(X)\right]} \tag{E.186}$$

Assume again that N samples drawn IID according to the PDF $g(x)$, i.e. $x_1, ..., x_N \sim g$, are available. Then an estimate of $\mathbb{E}_g\left[q(X)\, I_A(X)\right]$ can be calculated with the sample mean, and thus

$$\int_A p(x)\, dx = \frac{\mathbb{E}_g\left[\tilde{q}(X) I_A(X)\right]}{\mathbb{E}_g\left[\tilde{q}(X)\right]} \approx \frac{\frac{1}{N}\sum_{i=1}^{N} \tilde{q}(x_i) I_A(x_i)}{\frac{1}{N}\sum_{i=1}^{N} \tilde{q}(x_i)} \tag{E.187}$$

Again from the law of large numbers it follows

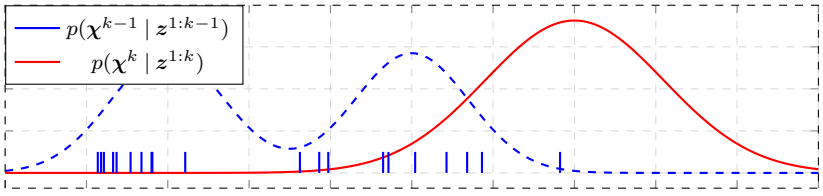
$$\lim_{N\to\infty} \frac{\frac{1}{N}\sum_{i=1}^{N} \tilde{q}(x_i) I_A(x_i)}{\frac{1}{N}\sum_{i=1}^{N} \tilde{q}(x_i)} = \frac{\lim_{N\to\infty} \frac{1}{N}\sum_{i=1}^{N} q(x_i) I_A(x_i)}{\lim_{N\to\infty} \frac{1}{N}\sum_{i=1}^{N} q(x_i)} = \frac{\mathbb{E}_g\left[\tilde{q}(X) I_A(X)\right]}{\mathbb{E}_g\left[\tilde{q}(X)\right]} = \int_A p(x)\, dx \tag{E.188}$$

Which proves that the estimate given by (E.187) converges to the true $p(x)$ for any given set A whenever a sufficient number of samples are used. Therefore these derivations show that given N samples, $x_1, ..., x_N$, drawn IID according to an arbitrary proposal distribution, $g(x)$, fulfilling (E.168), it is possible to approximate the target distribution, $p(x)$, by weighting the samples, $x_1, ..., x_N$, based on the importance weight function

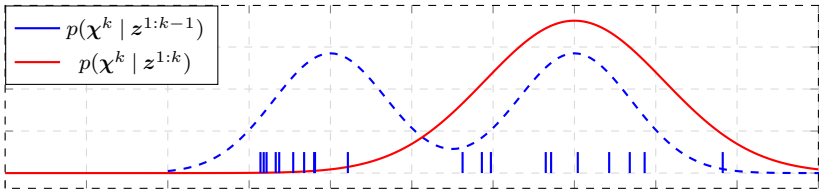
$$\frac{\tilde{q}(x_i)}{\sum_{i=1}^{N} \tilde{q}(x_i)} \tag{E.189}$$

E.8 Particle Filter

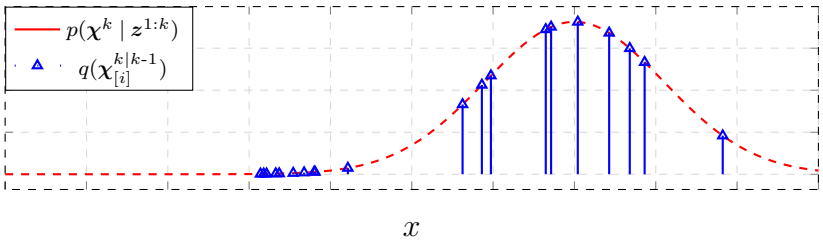
As mentioned in Section E.2 the Bayesian Estimator has the property that it keeps track of the full PDF of the state vector, but unfortunately analytical solutions are only available for special cases. As a solution to this problem the particle filter was invented as a numerical implementation of the Bayesian Estimator using a Monte Carlo methodology [30].



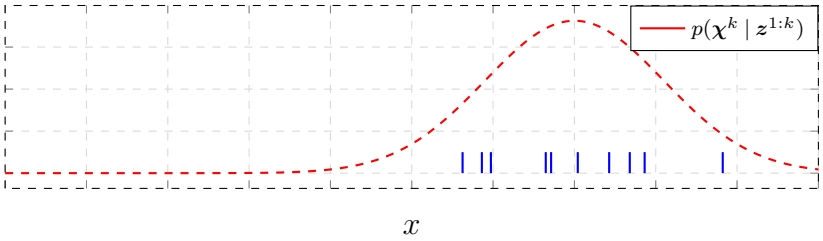
(a) A set of a posteriori particles for time k-1 distributed according to $p(\boldsymbol{\chi}^{k-1} \mid \boldsymbol{z}^{1:k-1})$.



(b) The posteriori particles for time k-1 propagated through the motion model, to obtain a priori particles at time k distributed according to $p(\boldsymbol{\chi}^{k} \mid \boldsymbol{z}^{1:k-1})$.



(c) A priori particles weighted according to the importance weight $q\left(\boldsymbol{\chi}^{k|k-1}\right)$.



(d) Resampling of the a priori particles to obtain a posteriori particles for time k, distributed according to $p(\boldsymbol{\chi}^{k} \mid \boldsymbol{z}^{1:k})$.

Figure E.4: Illustration of the steps within a particle filter.

The basic idea behind the particle filter is to draw a number of samples, called particles, from a known a priori distribution $p(\boldsymbol{\chi}^{k} \mid \boldsymbol{z}^{1:k-1})$. Using importance sampling, as described in Section E.7, the generated particles are weighted, such that they approximate the a posteriori distribution $p(\boldsymbol{\chi}^{k} \mid \boldsymbol{z}^{1:k})$.

A re-sampling is then performed on the weighted particles to generate a new particle set distributed according to the a posteriori distribution $p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k})$.
In this section a system model and measurement model given on the form in (E.190) and (E.191), respectively, are assumed [30].

$$\boldsymbol{\chi}^k = f^{k-1}\left(\boldsymbol{\chi}^{k-1}, \boldsymbol{w}^{k-1}\right) \tag{E.190}$$

$$\boldsymbol{z}^k = h^k\left(\boldsymbol{\chi}^k, \boldsymbol{v}^k\right) \tag{E.191}$$

Where:

$\boldsymbol{\chi}$	is the state vector
$\boldsymbol{z}$	is a measurement
$f^k(\bullet)$	is the time-varying system model
$h^k(\bullet)$	is the time-varying measurement model
$\boldsymbol{w}$	is the process noise
$\boldsymbol{v}$	is the measurement noise

Furthermore it is assumed that the process noise, $\boldsymbol{w}$, and measurement noise, $\boldsymbol{v}$, are independent, white and have known PDF's. Moreover, assume that the PDF $p(\boldsymbol{\chi}^0)$ is known.

To obtain the a priori particles, $\boldsymbol{\chi}_{[i]}^{k|k-1} \sim p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1})$ the old a posteriori particles, $\boldsymbol{\chi}_{[i]}^{k-1|k-1}$, are propagated using the motion model with random generated noise. That is

$$\boldsymbol{\chi}_{[i]}^{k|k-1} = f^{k-1}\left(\boldsymbol{\chi}_{[i]}^{k-1|k-1}, \boldsymbol{w}_{[i]}^{k-1}\right) \tag{E.192}$$

where $\boldsymbol{w}_{[i]}^{k-1}$ are drawn as samples from the known PDF of $\boldsymbol{w}^{k-1}$.
In the framework of importance sampling, the a priori distribution $p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1})$ should be seen as the proposal distribution, and the a posteriori distribution $p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k})$ should be seen as the target distribution. Thus based on (E.177) the importance weight function for a priori particles, $\boldsymbol{\chi}^{k|k-1}$, become

$$q\left(\boldsymbol{\chi}^k\right) = \frac{p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k})}{p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1})} \tag{E.193}$$

$$= \frac{p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^k, \boldsymbol{z}^{1:k-1})}{p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1})} \tag{E.194}$$

$$= \eta \cdot \frac{p(\boldsymbol{z}^k \mid \boldsymbol{\chi}^k, \boldsymbol{z}^{1:k-1}) p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1})}{p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1})} \tag{E.195}$$

$$= \eta \cdot p(\boldsymbol{z}^k \mid \boldsymbol{\chi}^k, \boldsymbol{z}^{1:k-1}) \tag{E.196}$$

$$= \eta \cdot p(\boldsymbol{z}^k \mid \boldsymbol{\chi}^k) \tag{E.197}$$

$$\propto p(\boldsymbol{z}^k \mid \boldsymbol{\chi}^k) \tag{E.198}$$

$$= \tilde{q}\left(\boldsymbol{\chi}^k\right) \tag{E.199}$$

(E.197) comes from the Markovian assumptions given in (E.190) and (E.191). Since the normalizer, η, coming from Bayes' rule, is not known, the self-normalized importance sampling, described in Section E.7.1, has to be used. Since the denominator of the fraction in (E.189) is not known

before all $\tilde{q}(x_i)$ have been calculated, the process of calculating the weights is divided into two steps. First $\tilde{q}(x_i)$ is calculated for all particles, and then they are normalized based on

$$\frac{\tilde{q}(x_i)}{\sum_{i=1}^{N} \tilde{q}(x_i)} \tag{E.200}$$

By doing so, the weighted particles become an approximation of the a posteriori distribution, $p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k})$. Resampling is then performed to generate a new random set of particles that are distributed according to this proposal distribution. Different algorithms to perform this resampling step have been developed [30], and the specific algorithm to use is usually decided based on a trade-off between computational efficiency and implementation complexity. After the resampling, any statistical measure of the a posteriori distribution, $p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k})$, can be calculated based on the newly generated particles. However, as the number of particles, N, defines the quantization of the a posteriori distribution, choosing the right number of particles is a trade-off between efficiency and estimation accuracy, since the accuracy of the estimates depend on the number of particles.

E.8.1 Calculating the importance weight

To calculate the importance weight of a priori particles, (E.201) has to be evaluated.

$$\tilde{q}\left(\boldsymbol{\chi}^k\right) = p(\boldsymbol{z}^k \mid \boldsymbol{\chi}^k) \tag{E.201}$$

$p(\boldsymbol{z}^k \mid \boldsymbol{\chi}^k)$ can be calculated by substituting a measurement, $\boldsymbol{z}^{k,*}$ into the non-linear measurement model, using the current state of the particle, $\boldsymbol{\chi}_{[i]}^{k|k-1}$, and then solve for the measurement noise variable, $\boldsymbol{v}^k$. Let $\boldsymbol{v}^{k,*}$ denote the solution. With the solution the PDF, $p\left(\boldsymbol{v}^k\right)$, assumed to be known, is evaluated at the current noise, $\boldsymbol{v}^{k,*}$. That is

$$\boldsymbol{z}^{k,*} = h^k\left(\boldsymbol{\chi}_{[i]}^{k|k-1}, \boldsymbol{v}^k\right) \qquad \Rightarrow \qquad \boldsymbol{v}^{k,*} = h^{k\,-1}\left(\boldsymbol{\chi}_{[i]}^{k|k-1}, \boldsymbol{z}^{k,*}\right) \tag{E.202}$$

$$q_i = p\left(\boldsymbol{v}^k = \boldsymbol{v}^{k,*}\right) \tag{E.203}$$

E.8.2 Particle filter algorithm summary

The particle filter algorithm can be summarized as follows

1. Initialize the filter by drawing N random state vectors based on the knowledge of $p(\boldsymbol{\chi}^0)$. That is
$$\boldsymbol{\chi}_{[i]}^{0|0} \sim p(\boldsymbol{\chi}^0) \qquad \text{for} \quad i = 1, ..., N \tag{E.204}$$
Each of these random generated state vectors are called particles. This is illustrated in Figure E.4(a).

2. For k = 1,2,... perform the following steps

   (a) Propagate the particles to obtain a priori particles, $\boldsymbol{\chi}_{[i]}^{k|k-1}$, by using the system model:
   $$\boldsymbol{\chi}_{[i]}^{k|k-1} = f^{k-1}\left(\boldsymbol{\chi}_{[i]}^{k-1|k-1}, \boldsymbol{w}_{[i]}^{k-1}\right) \tag{E.205}$$
   Where $\boldsymbol{w}_{[i]}^{k-1}$ is random samples drawn from the known PDF of $\boldsymbol{w}^{k-1}$. Thereby the particles will be distributed according to the proposal distribution $p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1})$. This is illustrated in Figure E.4(b).

(b) To approximate the target distribution $p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k})$, based on the a priori particles, perform self-normalized importance sampling For each $i = 1, ..., N$ calculate the unnormalized importance weight, $\tilde{q}_{[i]}\left(\boldsymbol{\chi}_{[i]}^{k|k-1}\right)$ Now normalize the weights, $\tilde{q}_{[i]}$ according to

$$q_{[i]} = \frac{\tilde{q}_{[i]}}{\sum_{j=1}^{N} \tilde{q}_{[j]}} \tag{E.206}$$

This is illustrated in Figure E.4(c).

(c) Draw a new set of a posteriori particles, $\boldsymbol{\chi}_{[i]}^{k|k}$, relative to the calculated normalized weights $q_{[i]}$.

3. The a posteriori particles, $\boldsymbol{\chi}_{[i]}^{k|k}$, are distributed according to $p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k})$, as illustrated in Figure E.4(d). Thus any desired statistical measure of $p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k})$ can be computed based on the a posteriori particles.

It should be noted, that different approaches to improve the particle filter exists and thus some of the steps mentioned above can vary. One approach is described in Section E.8.4 while others can be found in [30].

E.8.3 Sample impoverishment

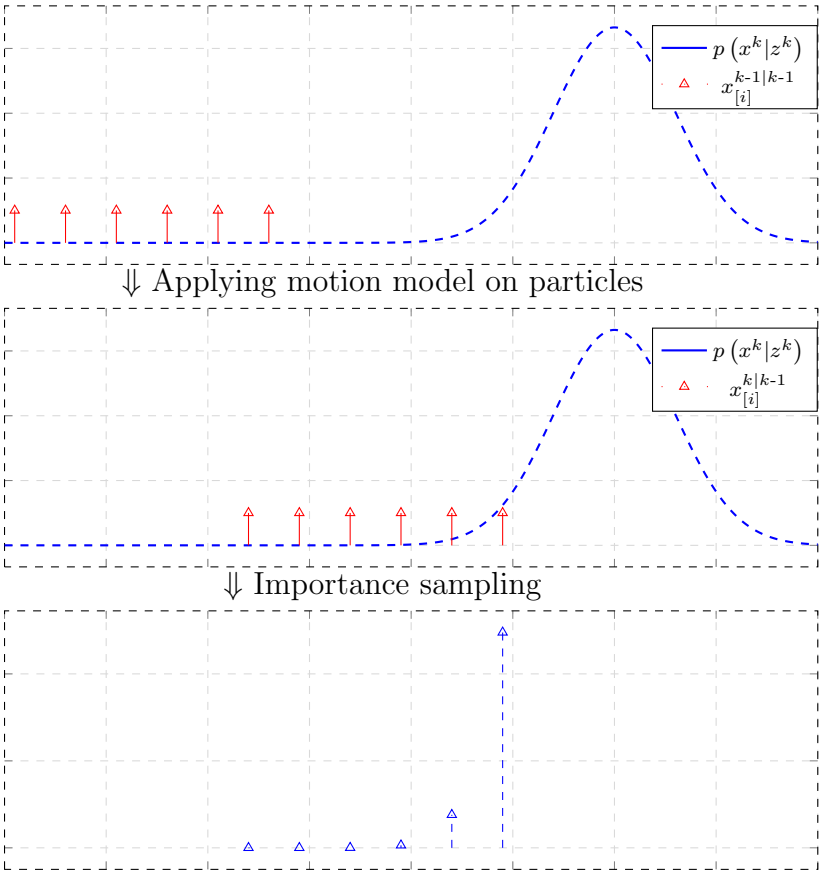


Figure E.5: Illustration of sample impoverishment due to bad motion model and a low number of particles.

As stated in Section E.7, the requirement given by (E.168) has to be fulfilled for the importance sampling to work. For the particle filter described in this section, this means that

$$p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k}) > 0 \qquad \Rightarrow \qquad p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1}) > 0 \tag{E.207}$$

Since the a priori particles, $\boldsymbol{\chi}_{[i]}^{k|k-1}$, are only discrete representations of the proposal distribution, $p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1})$, (E.207) may not be fulfilled even though the continuous proposal distribution fulfils the requirement.
As illustrated in Figure E.5, this is likely to happen if a bad motion model is used to obtain the a priori particles, $\boldsymbol{\chi}_{[i]}^{k|k-1} \sim p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1})$. In that case only a few or a single particle will be given a significant importance weight, resulting in only a few or even just a single a priori particle being selected for the a posteriori particles in the resampling step. This is known as sample impoverishment.

If the continuous proposal and target distribution fulfil (E.207), then this can partly be solved by increasing the number of particles. But this quickly lowers the computational efficiency of the filter and often only delays the sample impoverishment [30]. However different solutions to prevent sample impoverishment is suggested in [30]. One of these solutions is described in details in Section E.8.4.

E.8.4 Particle Filter combined with other filters

A suggested improvement of the particle filter to avoid sample impoverishment, see Section E.8.3, is to combine it with other filters [30]. One approach is to refine the a priori particles, $\boldsymbol{\chi}_{[i]}^{k|k-1}$, before calculating importance weights and performing the resampling. By refining the distribution with another filter taking measurements into account, the a priori particles will thus turn into a posteriori particles, $\boldsymbol{\chi}_{[i]}^{k|k}$, on which the importance weights are calculated. Therefore the likelihood that these particles will overlap with the target distribution is higher, yielding a better estimate of the target distribution.

As an example shown below the algorithm for a particle filter can be combined with the sequential Extended Kalman Filter described in Section E.6.

1. Initialize all particles in the filter by drawing N random state vectors based on the knowledge of $p(\boldsymbol{\chi}^0)$. That is

$$\boldsymbol{\chi}_{[i]}^{0|0} \sim p(\boldsymbol{\chi}^0) \qquad \text{for} \quad i = 1, ..., N \tag{E.208}$$

   Furthermore, assign each of these particles a covariance $\text{Cov}\left(\boldsymbol{\chi}_{[i]}^{0|0}\right) = \text{Cov}\left(\boldsymbol{\chi}^{0|0}\right)$.

2. For k = 1,2,... perform the following steps

   (a) Propagate the particles to obtain a priori particles, $\boldsymbol{\chi}_{[i]}^{k|k-1}$, by using the system model as described for the sequential Extended Kalman filter in (E.151) to (E.152). However, the noise in the motion model should not be put to zero. Then, (E.151) becomes

$$\boldsymbol{\chi}_{[i]}^{k|k-1} = f^{k-1}\left(\boldsymbol{\chi}_{[i]}^{k-1|k-1}, \boldsymbol{w}_{[i]}^{k-1}\right) \tag{E.209}$$

   Where $\boldsymbol{w}_{[i]}^{k-1}$ is random samples drawn from the known PDF of $\boldsymbol{w}^{k-1}$. Thereby the particles will be distributed according to the proposal distribution $p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k-1})$.

(b) Perform the update step of the sequential Extended Kalman filter on each of the a priori particles, $\boldsymbol{\chi}_{[i]}^{k|k-1}$, to obtain a posteriori particles, $\tilde{\boldsymbol{\chi}}_{[i]}^{k|k}$. The update step is performed seperately for each particle according to (E.161) to (E.155).

(c) Since the sequential Extended Kalman filter makes use of a linear approximation of the system model and measurement model, the obtained a posteriori particles, $\tilde{\boldsymbol{\chi}}_{[i]}^{k|k}$, will not be distributed exactly according to the target distribution $p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k})$. To approximate the correct target distribution based on the a posteriori particles, $\tilde{\boldsymbol{\chi}}_{[i]}^{k|k}$, the self-normalized importance sampling is used. For each $i = 1, ..., N$ calculate the unnormalized importance weight, $\tilde{q_{[i]}}\left(\tilde{\boldsymbol{\chi}}_{[i]}^{k|k}\right)$

(d) Now normalize the weights, $\tilde{q}_{[i]}$ according to

$$q_{[i]} = \frac{\tilde{q}_{[i]}}{\sum_{j=1}^{N} \tilde{q}_{[j]}} \tag{E.210}$$

(e) Draw a new set of a posteriori particles, $\boldsymbol{\chi}_{[i]}^{k|k}$, relative to the calculated normalized weights $q_{[i]}$.

3. The obtained a posteriori particles, $\boldsymbol{\chi}_{[i]}^{k|k}$, are distributed according to $p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k})$. Thus any desired statistical measure of $p(\boldsymbol{\chi}^k \mid \boldsymbol{z}^{1:k})$ can be computed based on the a posteriori particles, $\boldsymbol{\chi}_{[i]}^{k|k}$.

E.9 FastSLAM 2 details

This appendices describes some further details and the equations of the FastSLAM 2.0 algorithm left out of the summery of the algorithm in Section 7.1.

E.9.1 Drawing Samples from the Proposal Distribution

In the following subscript $_{[p]}$ denotes particle index, and $p = 1, ..., N$, where N is the number of particles in the particle filter.
FastSLAM 2.0 draws proposal particles, $\tilde{\boldsymbol{s}}_{[p]}^{1:k}$, from an approximation of the proposal distribution given in (7.10). These proposal particles is obtained as follows. For each particle at time k-1, $\boldsymbol{s}_{[p]}^{1:k-1}$, a mean, $\bar{\boldsymbol{s}}_{[p]}^{k}$ and covariance, $\text{Cov}\left(\boldsymbol{s}_{[p]}^{k}\right)$, of a Gaussian distribution is estimated with the sequential Extended Kalman Filter described in Section E.6.1.

For the update step of the sequential Extended Kalman Filter only measurements, $\boldsymbol{z}_{ex}^{k}$, of landmarks already initialized in the filter at time k, are used. For the sake of completeness all equations for this sequential Extended Kalman Filter, proposed by the authors of FastSLAM in [13], is shown from (E.211) to (E.214). These equations are calculated recursively for $i = 1, ..., I$ where I is the number of landmark measurements of already initialized landmarks.

$$\text{Cov}\left(\boldsymbol{s}_{0,[p]}^{k}\right) = \text{Cov}\left(\boldsymbol{w}^{k}\right) \tag{E.211}$$

$$\bar{\boldsymbol{s}}_{0,[p]}^{k} = f\left(\boldsymbol{s}_{[p]}^{k-1}, \boldsymbol{u}^{k}\right) \tag{E.212}$$

$$\text{Cov}\left(\boldsymbol{s}^{k}\right)_{i,[p]} = \left(\boldsymbol{H}_{s,i}^{T}\left(\boldsymbol{Z}_{i}^{k}\right)^{-1}\boldsymbol{H}_{s,i} + \left(\text{Cov}\left(\boldsymbol{s}_{i-1,[p]}^{k}\right)\right)^{-1}\right)^{-1} \tag{E.213}$$

$$\bar{\boldsymbol{s}}_{i,[p]}^{k} = \bar{\boldsymbol{s}}_{i-1,[p]}^{k} + \text{Cov}\left(\boldsymbol{s}_{i,[p]}^{k}\right)\boldsymbol{H}_{s,i}^{T}\left(\boldsymbol{Z}_{i}^{k}\right)^{-1}\left(\boldsymbol{z}_{ex}^{k} - \hat{\boldsymbol{z}}_{ex,i}^{k}\right) \tag{E.214}$$

where

$$\boldsymbol{H}_{\boldsymbol{l},i} = \left.\frac{\partial h_M\left(\boldsymbol{s}^{k}, \boldsymbol{l}_i\right)}{\partial \boldsymbol{l}_i}\right|_{\boldsymbol{s}^{k}=\bar{\boldsymbol{s}}^{k}, \boldsymbol{l}=\boldsymbol{l}_{[p],i}^{k-1}}$$

$$\boldsymbol{H}_{\boldsymbol{s},i} = \left.\frac{\partial h_M\left(s^{k}, \boldsymbol{l}_i\right)}{\partial \boldsymbol{s}^{k}}\right|_{s^{k}=\bar{\boldsymbol{s}}^{k}, \boldsymbol{l}=\boldsymbol{l}_{[p],i}^{k-1}}$$

$$\boldsymbol{Z}_{i}^{k} = \text{Cov}\left(\boldsymbol{v}^{k}\right) + \boldsymbol{H}_{l,i}\text{Cov}\left(\boldsymbol{l}_{[p],i}^{k-1}\right)\boldsymbol{H}_{l,i}^{T}$$

$$\hat{\boldsymbol{z}}_{ex,i}^{k} = h\left(\bar{\boldsymbol{s}}^{k}, \boldsymbol{l}_{[p],i}^{k-1}\right)$$

and $\boldsymbol{l}_i$ is used to denote the landmark that matches the i'th measurement, $\boldsymbol{z}_{ex,i}$, of landmarks already initialized in the filter.

After having estimated the a posteriori distribution one random sample is drawn, $\tilde{\boldsymbol{s}}_{[p]}^{k}$, from the normal distribution with mean, $\bar{\boldsymbol{s}}_{[p]}^{k|k}$, and covariance, $C\left(\boldsymbol{s}_{[p]}^{k|k}\right)$. That is

$$\tilde{\boldsymbol{s}}_{[p]}^{k} \sim \mathcal{N}\left(\bar{\boldsymbol{s}}_{I,[p]}^{k}, C\left(\boldsymbol{s}_{I,[p]}^{k}\right)\right) \tag{E.215}$$

This sample, $\tilde{\boldsymbol{s}}_{[p]}^{k}$, is then added to the particle, $\boldsymbol{s}_{[p]}^{1:k-1}$, to obtain proposal particles, $\tilde{\boldsymbol{s}}_{[p]}^{1:k}$.

E.9.2 Calculate Importance Weights

As stated in Section E.8, the importance weights of the particles in a particle filter should be calculated as

$$q_{[p]}^{k} = \frac{\text{target distribution}}{\text{proposal distribution}} \tag{E.216}$$

The proposal particles, $\tilde{\boldsymbol{s}}_{[p]}^{1:k}$ will not be distributed exactly according to (7.10). Nevertheless, the importance weight function used by FastSLAM 2.0 is derived as in (E.216) assuming that the proposal particles, $\tilde{\boldsymbol{s}}_{[p]}^{1:k}$, are actually distributed according to (7.10).

In [14] it is shown that there exist no closed form of $q^k_{[p]}$, for the proposal and target distributions used within FastSLAM, from which the importance weight can easily be determined. Therefore FastSLAM 2.0 approximates the importance weight as being a product of Gaussian distributions, with means given by (E.217) and covariances given by (E.218) [13]. Each of these Gaussian distributions represents the probability of each measurement of landmarks already initialized in the filter.

$$\hat{\boldsymbol{z}^k_{ex,i}} = h\left(\tilde{\boldsymbol{s}}^k_{[p]}, \bar{l}^{k-1}_{i,[p]}\right) \tag{E.217}$$

$$\mathrm{Cov}\left(q^k_{i,[p]}\right) = \boldsymbol{H}_{s,i}\mathrm{Cov}\left(\boldsymbol{w}^k\right)\boldsymbol{H}^T_{s,i} + \boldsymbol{H}_{l,i}\mathrm{Cov}\left(\boldsymbol{l}^{k-1}_{i,[p]}\right)\boldsymbol{H}^T_{l,i} + \mathrm{Cov}\left(\boldsymbol{v}^k\right) \tag{E.218}$$

where

$$\boldsymbol{H}_{s,i} = \left.\frac{\partial h\left(\boldsymbol{s}^k, \boldsymbol{l}_i\right)}{\partial \boldsymbol{s}^k}\right|_{\boldsymbol{s}^k=\tilde{\boldsymbol{s}}^k_{[p]}, \boldsymbol{l}_i=\bar{\boldsymbol{l}}^{k-1}_{i,[p]}} \tag{E.219}$$

$$\boldsymbol{H}_{l,i} = \left.\frac{\partial h\left(\boldsymbol{s}^k, \boldsymbol{l}_i\right)}{\partial \boldsymbol{l}_i}\right|_{\boldsymbol{s}^k=\tilde{\boldsymbol{s}}^k_{[p]}, \boldsymbol{l}_i=\bar{\boldsymbol{l}}^{k-1}_{i,[p]}} \tag{E.220}$$

The probability of a given measurement within a particle is calculated by evaluating PDF using the measurement and the properties of the distribution defined by $\hat{\boldsymbol{z}^k_{ex,i}}$ and $\mathrm{Cov}\left(q^k_{i,[p]}\right)$.

$$\tilde{q}^k_{i,[p]} \approx \left|2\pi C\left(q^k_{i,[p]}\right)\right|^{-\frac{1}{2}} \mathrm{e}^{-\frac{1}{2}\left(\boldsymbol{z}^k_{ex,i} - \hat{\boldsymbol{z}^k_{ex,i}}\right)^T \mathrm{Cov}\left(q^k_{i,[p]}\right)^{-1}\left(\boldsymbol{z}^k_{ex,i} - \hat{\boldsymbol{z}^k_{ex,i}}\right)} \tag{E.221}$$

The unnormalized importance weight of each particle is calculated as the product of probability of each measurement.

E.9.3 Update Landmarks

After estimating the proposal distribution, the map maintained by each particle is updated based on the recent measurements, z^k. This update step of the map can be divided in two stages. The first stage takes measurements of landmarks already initialized in the filter, $z_{ex,i}$, and updates the landmarks based on these. The second stage add new landmarks to the map by using measurements of landmarks not already initialized in the filter, $z^k_{\mathrm{new},i}$. Due to the assumption of conditional independence, the estimate of landmarks which are already initialized in the map, but of which no measurements are available, will stay unchanged.

As stated the estimate of landmarks is performed by an EKF for each of the landmarks. Since the landmarks are assumed to be static, the prediction step of these EKF's can be discarded, meaning that the equations for the update step of the landmarks effectively become the equations of the optimal affine recursive least squares estimator described in Section E.4. For completeness sake the equations of these Kalman Filters estimating the landmark location of the i'th landmark, $\boldsymbol{l}_i$, is shown in (E.222) and (E.223) [13].

$$\bar{\boldsymbol{l}}_{i,[p]}^{k} = \bar{\boldsymbol{l}}_{i,[p]}^{k-1} + \boldsymbol{K}^{k}\left(\boldsymbol{z}_{ex,i} - \hat{\boldsymbol{z}_{ex,i}^{k}}\right) \tag{E.222}$$

$$\text{Cov}\left(\boldsymbol{l}_{i,[p]}^{k}\right) = \left(\boldsymbol{I} - \boldsymbol{K}^{k}\boldsymbol{H}_{l}\right)\text{Cov}\left(\boldsymbol{l}_{i,[p]}^{k-1}\right) \tag{E.223}$$

where

$$\boldsymbol{H}_{l} = \left.\frac{\partial h\left(s^{k}, l_{i}\right)}{\partial \boldsymbol{l}_{i}}\right|_{\boldsymbol{s}^{k}=\tilde{\boldsymbol{s}}_{[p]}^{k}, \boldsymbol{l}_{i}=\bar{\boldsymbol{l}}_{i,[p]}^{k-1}} \tag{E.224}$$

$$\boldsymbol{K}^{k} = \text{Cov}\left(\boldsymbol{l}_{i,[p]}^{k-1}\right)\boldsymbol{H}_{l}^{T}\left(\text{Cov}\left(\boldsymbol{v}^{k}\right) + \boldsymbol{H}_{l}\text{Cov}\left(\boldsymbol{l}_{i,[p]}^{k-1}\right)\boldsymbol{H}_{l}^{T}\right)^{-1} \tag{E.225}$$

$$\hat{\boldsymbol{z}_{ex,i}^{k}} = h\left(\tilde{\boldsymbol{s}}_{[p]}^{k}, \bar{\boldsymbol{l}}_{i,[p]}^{k-1}\right) \tag{E.226}$$

The initialization of a new landmark depends upon if the measurement model for the landmarks (7.9) is invertible or not. If the measurement model is not invertible, then a single measurement is not sufficient to initialize a landmark and special techniques have to be utilized. E.g. for monocular SLAM algorithms using only one RGB camera, the relative depth of a landmark cannot be estimated based on just one image and thus the landmark cannot be placed in an absolute frame.
One way of dealing with this problem is to initialize new landmarks into an EKF with some other parameters, of which one of them is the depth. Hence letting a Kalman Filter estimate the depth before the landmark can be inserted into the map. When the depth estimate has converged, it can be used to place the landmark in an absolute frame [20].

However if the measurement model is invertible then landmarks can be initialized directly and the posterior distribution for the i'th landmark can be calculated as [13]

$$\bar{\boldsymbol{l}}_{i,[p]}^{k} = h_{M}^{-1}\left(\tilde{\boldsymbol{s}}_{[p]}^{k}, \boldsymbol{z}_{\text{new},i}^{k}\right) \tag{E.227}$$

$$\text{Cov}\left(\boldsymbol{l}_{i,[p]}^{k}\right) = \left(\boldsymbol{H}_{l}^{T}\text{Cov}\left(\boldsymbol{v}^{k}\right)^{-1}\boldsymbol{H}_{l}\right)^{-1} \tag{E.228}$$

where

$$\boldsymbol{H}_{l} = \left.\frac{\partial h\left(\boldsymbol{s}^{k}, \boldsymbol{l}_{i}\right)}{\partial \boldsymbol{l}_{i}}\right|_{\boldsymbol{s}^{k}=\tilde{\boldsymbol{s}}_{[p]}^{k}, \boldsymbol{l}_{i}=\bar{\boldsymbol{l}}_{i,[p]}^{k}}$$

F Intel RealSense R200 camera

The RealSense R200 RGB-D USB 3.0 camera [48] provided as part of the Intel Aero drone is a combined RGB and depth camera. This appendix describes the function, technical specifications and how to use the R200 camera. The content within this appendix is based on a combination of details, specifications and datasheets from Intel, a paper from Intel giving a comprehensive overview of the camera [49]and own experience from using the R200 camera on the Intel Aero drone.

Capturing depth of an environment can be done in several ways, where most commercial solutions either contain a stereo camera pair, a structured light (projective) solution or Time-Of-Flight technology. The RealSense R200 camera combines multiple solutions by being equipped with an infrared (IR) projector and two infrared cameras capturing a stereo image of the environment. If enough IR light is present in the environment, the two IR cameras would be enough to capture the depth of the scene, but to make sure that surfaces with a plain texture can be captured as well, an IR laser projector emits a grid of IR lines which can be captured by the cameras.

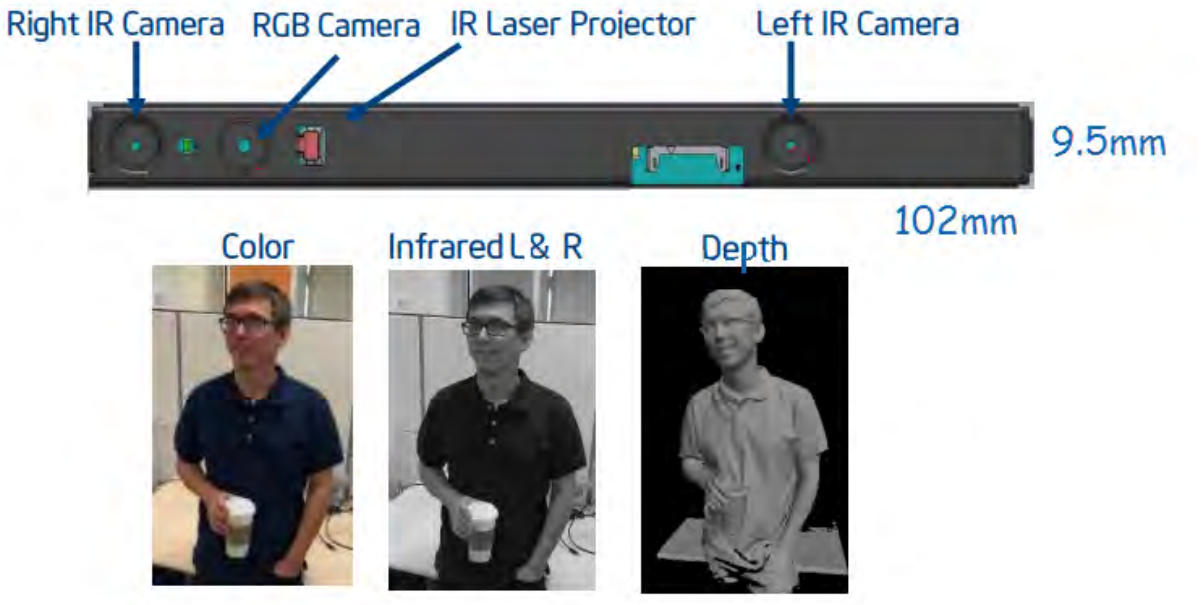
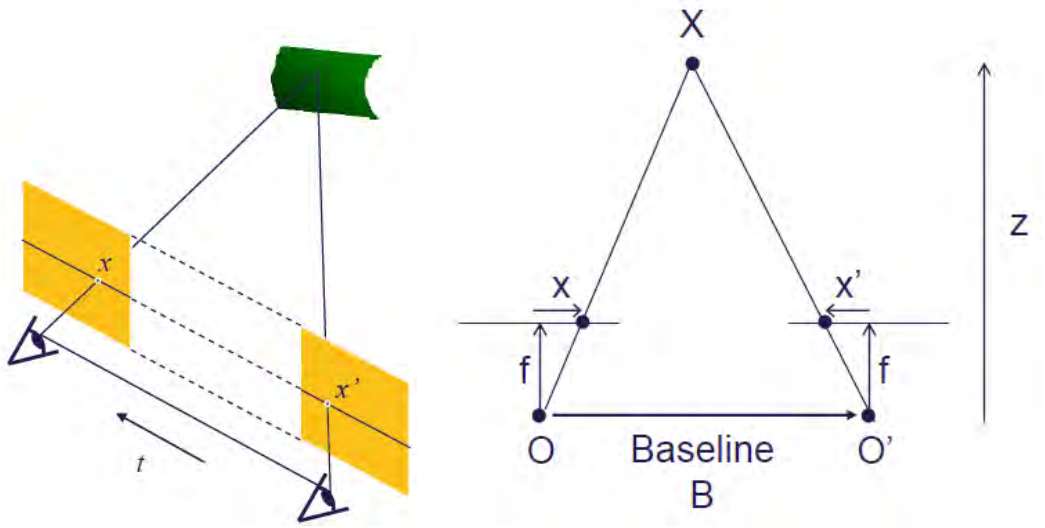


Figure F.1: R200 camera layout [50]

With two rectified stereo images, that is two images where the epipolar lines are horizontal and only the horizontal translation is different, see Figure F.2(a), it is possible to calculate a disparity map.
Similar triangles can then be used to easily calculate the depth from the disparity, $(x - x')$, by using the intrinsics of the stereo cameras, as part of the pin-hole camera model, see Appendix G, and the extrinsics, being just the baseline B for rectified images. This is shown in Figure F.2(b).



(a) Rectified images [51]

(b) Disparity to depth conversion using similar triangles [51]

Figure F.2: Rectified stereo images used to calculate depth

Performing a scan-based matching between similar points present in both images allows the R200 camera to generate a depth image where each pixel value corresponds to the depth value to a projected point. Hence, the depth values are projected back into a common frame, being the left IR camera in the R200 case, resulting in a 2D depth image. Depth images are usually pictured as greyscale images even though the images rather represent a 2D array of depth values usually in millimetres.

A summary of the above mentioned functionality being a part of the R200 camera is shown in the functional overview diagram provided by Intel, Figure F.3.

Figure F.3: Functional overview of R200 camera [52]

The important benefit to notice from this functional overview of the R200 camera is how the disparity processing is contained within an onboard ASIC that does the processing and depth image generation at full frame-rate, see the tables in Section F.1.

F.1 Technical details

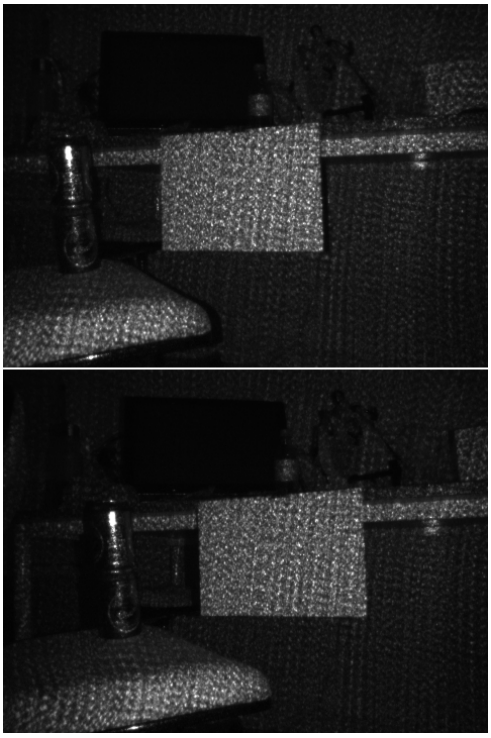
The RealSense R200 camera comes equipped with a lens giving the RGB camera a vertical field of view of $43° \pm 2°$ and a horizontal field of view of $70° \pm 2°$ and the IR cameras a slightly narrower field of view with $59° \pm 2°$ of horizontal field of view but $46° \pm 2°$ of vertical field of view. According to the datasheet of the RealSense R200 camera [52], the individual cameras of the R200 camera and the generated depth image have a defined set of possible resolutions and frame-rates as shown in Table F.1. These specifications define an obvious limit for possible resolution and maximum frame-rate for the FastSLAM implementation.

Camera	**Supported resolutions**	**Supported frame-rates [FPS]**
RGB	320×240	15, 30, 60
	640×480	30, 60
	1920×1080	15, 30
IR	332×252	30, 60, 90
	492×372	30, 60, 90
	640×480	30, 60, 90
Depth	320×240	30, 60, 90
	332×252	30, 60, 90
	480×360	30, 60, 90
	492×372	30, 60, 90
	628×468	30, 60, 90

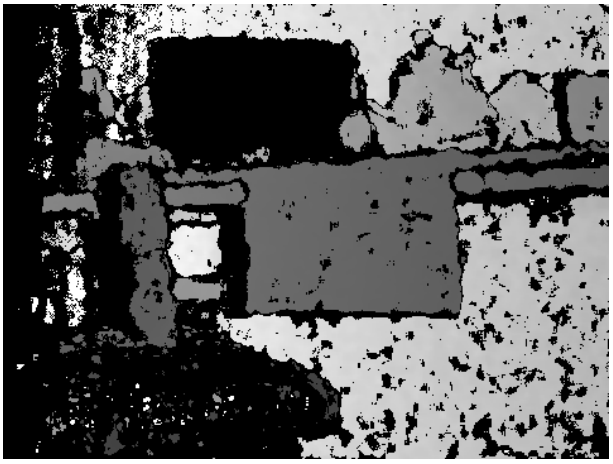
Table F.1: Supported resolutions and frame-rates [53]

Figure F.4: Example of non-rectified, raw and hence distorted RGB image at 640×480 resolution

An example of a raw RGB image taken at a resolution 640×480, that is an image not being rectified and containing distortion as well, is shown in Figure F.4. The corresponding pair of IR stereo images taken at a resolution of 492×372 pixels is shown in Figure F.5(a), stacked on top of each other to be able to see the difference (disparity). Finally, the resulting depth image captured at a resolution of 480×360 pixels is shown in Figure F.5(b).



(a) IR stereo images with left image at the top and right image at the bottom



(b) Depth image at 480×360 resolution

Figure F.5: Rectified stereo images used to calculate depth

The choices of IR resolution and resulting depth resolution are linked as the depth image is generated based on the two IR images. For this matter it is also apparent how the generated depth resolution is slightly smaller than the corresponding IR resolution. This is a result of the depth image generation where an internal morphological operation is performed on the disparity map to generate a smoothed and less noisy depth image. However, within the depth image one should also expect the usable region to be smaller than the actual image resolution due to the fact that the overlapping part of the stereo images is smaller than the image size itself. This is also noticeable in the depth image in Figure F.5(b) where a black bar, indicating no depth, is present in the left side of the image. This unusable area is known as the dead-zone between the stereo images, see Figure F.6.

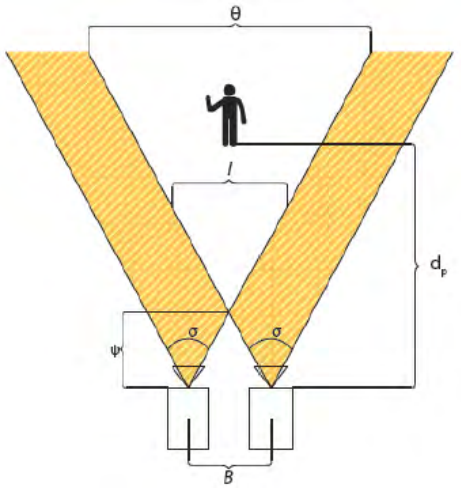


Figure F.6: The yellow area marks the dead-zones between the stereo cameras [51]

Furthermore, the R200 camera has some other limitations for the depth range, limited by the disparity processing and the visibility of the projected IR grid. In the datasheet for the R200 camera a limited depth range of 0.4 m to 2.8 m is specified, but within an article by Intel an inside range of approximately 0.5 m to 3.5 m and an outside range up to 10 m is mentioned [50]. The depth range has been tested within the Motion Tracking lab at Aalborg University and has been confirmed to work at least up to 3 m even with the Vicon cameras running at the same time, emitting IR light.

A resolution of 320×240 pixels and a frame-rate of 30 FPS is chosen for both the RGB and depth image to be used. As the speed and performance requirements have not been investigated as part of this project, the increased frame-rate of 60 FPS has not been considered.

F.2 Factory calibrated camera parameters

The Intel RealSense R200 camera comes with a set of factory calibrated parameters including camera intrinsics, extrinsics and distortion parameters. These parameters are all saved internally inside the camera and can be extracted through the Intel RealSense SDK or librealsense library (C++) communicating with the camera over USB. The specifications for the R200 camera taken directly from the librealsense library [54]are shown below:

Left and right infrared images are rectified

- The two infrared streams have identical intrinsics
- The two infrared streams have no distortion
- There is no rotation between left and right infrared images (identity matrix)
- There is only a horizontal translation between left and right infrared images
- Therefore, the y component of pixel coordinates can be used interchangeably between these two streams

Depth images are pixel aligned with the first infrared stream except for an optional 6 pixel offset

- Native depth images are six pixels smaller on all four sides, but are otherwise pixel aligned with infrared
- librealsense will pad the depth image or crop the infrared image if matching resolutions are requested
- If matching resolutions are requested, depth and infrared images will use the exact same intrinsics
- If not, pixel coordinates can be mapped by adding or subtracting six pixels from both components

R200 color images use Modified Brown-Conrady Distortion, but can be rectified in software

- Request frames from the rectified color stream to received images with no distortion
- There is no rotation between depth/infrared and rectified color (identity matrix)
- There can be translation in all three axes between depth/infrared and rectified color
- Therefore, the x and y component of pixel coordinates can be mapped independently between depth/infrared and rectified color

F.2.1 Extrinsics

As seen by the R200 camera layout shown in Figure F.3 the RGB camera and stereo cameras are obviously not located on top of each other, why a physical difference in the location of the camera frames will be apparent described by the extrinsics of the images. These extrinsics can be extracted from the calibrated camera parameters and are listed in Table F.2. However, there is no rotation between the cameras why the rotation is included in the table, indicating that the orientation is aligned or at least assumed to be aligned for all cameras [54].

Camera	**Extrinsic translation, $[\mathbf{x\ y\ z}]^{\mathbf{T}}$**
RGB	$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$ m
Left IR	$\begin{bmatrix} -0.0007056731 \\ 0.05833369 \\ 0.0003919501 \end{bmatrix}$ m
Depth frame	$\begin{bmatrix} -0.0007056731 \\ 0.05833369 \\ 0.0003919501 \end{bmatrix}$ m
Right IR	$\begin{bmatrix} -0.0007056731 \\ -0.01150674 \\ 0.0003919501 \end{bmatrix}$ m

Table F.2: Factory calibrated R200 camera extrinsics for the camera used

The extrinsics are all given in a body frame relative to the RGB camera and aligned with the body frame of the drone, where the x-axis points forward and y-axis to the left. Hence, the optical frame of the cameras, where the z-axis is pointing in the direction of the viewpoint, is different from the frame wherein the extrinsics are defined.

F.2.2 Intrinsics and distortion

As mentioned, all the cameras are equipped with a lens, why the images will be prone to distortion near the edge of the images. Luckily the onboard ASIC also does some preprocessing to remove the distortion from the IR images before rectification and depth image generation. Therefore, the resulting depth image is non-distorted and aligned with the intrinsics of the left IR image. For the RGB image however the image is distorted from which a set of calibrated distortion parameters is included. These parameters model the distortion using the Modified Brown-Conrady Distortion model [55] which includes five distortion parameters defined by the $\boldsymbol{D}$ matrix. The distortion parameters include three radial distortion coefficients, $k_{1:3}$, and two tangential distortion coefficients, $t_{1:2}$.

$$\boldsymbol{D} = \begin{bmatrix} k_1 \\ k_2 \\ t_1 \\ t_2 \\ k_3 \end{bmatrix} \tag{F.1}$$

The intrinsics of the individual cameras are all defined according to the pin-hole camera model as described in Appendix G. The intrinsics are defined as the projection matrix, $\boldsymbol{P}$, see also (G.6).

$$\boldsymbol{P} = \begin{bmatrix} a_x & 0 & x_0 & 0 \\ 0 & a_y & y_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{F.2}$$

Both the intrinsics and distortion parameters of the individual cameras depend on the configured resolution. As an example, the intrinsics for the default resolution, where RGB is 640×480 and depth is 480×360, is shown in Table F.3 as well as the used resolution with both RGB and depth being 320×240.

Camera	Resolution	Intrinsics matrix, P	Distortion parameters, D
RGB	640×480	$\begin{bmatrix} 617.8589 & 0 & 321.3710 & 0 \\ 0 & 623.4426 & 253.7631 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$	$\begin{bmatrix} -0.07299995 \\ 0.04706055 \\ 0.001392152 \\ 0.0004241989 \\ 0.0 \end{bmatrix}$
Depth	480×360	$\begin{bmatrix} 457.4863 & 0 & 241.1611 & 0 \\ 0 & 457.4863 & 179.5000 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix}$
RGB	320×240	$\begin{bmatrix} 308.9294 & 0 & 160.6855 & 0 \\ 0 & 311.7213 & 126.8816 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$	$\begin{bmatrix} -0.07299995 \\ 0.04706055 \\ 0.001392152 \\ 0.0004241989 \\ 0.0 \end{bmatrix}$
Depth	320×240	$\begin{bmatrix} 309.2598 & 0 & 160.1654 & 0 \\ 0 & 309.2598 & 119.4571 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{bmatrix}$

Table F.3: Factory calibrated R200 camera intrinsics for the camera used

F.2.3 Distortion

The Modified Brown-Conrady Distortion model [23], is a combined radial and tangential distortion model which has been used to model the distortion on the RGB image. Unfortunately, these parameters can only be used one-way distortion for applying distortion to an undistorted image. This means that only points in the world can be projected onto the image plane using the intrinsics, whereafter the distortion model can be applied to yield a distorted image similar to the captured RGB image.

Applying the Modified Brown-Conrady Distortion model to an undistorted pixel location, (x_u, y_u), to get a corresponding distorted pixel location, (x_d, y_d), is shown in (F.3) to (F.9).

$$u' = \frac{x_u - x_0}{a_x} \tag{F.3}$$

$$v' = \frac{y_u - y_0}{a_y} \tag{F.4}$$

$$r^2 = u'^2 + v'^2 \tag{F.5}$$

$$x' = \left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) u' + 2t_1 u' v' + t_2 \left(r^2 + 2u'^2\right) \tag{F.6}$$

$$y' = \left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) v' + 2t_2 u' v' + t_1 \left(r^2 + 2v'^2\right) \tag{F.7}$$

$$x_d = x' a_x + x_0 \tag{F.8}$$

$$y_d = y' a_y + y_0 \tag{F.9}$$

Distortion removal with this type of model is only possible through an iterative process where the pixels from the distorted image are deprojected assuming an undistorted image and then projected back to correct the pixel location. This procedure is then repeated several times until the pixel locations have converged.

F.3 Generating point clouds

To show how the distortion of the RGB image can be handled and combined with the depth image through an iterative approach, Intel has provided a C and C++ example of how to generate a coloured point cloud. A coloured point cloud is a collection of several coloured points placed in a 3-dimensional space, commonly containing thousands of points. The example generates such a point cloud by grabbing pixel colors from the distorted image and placing them in a 3D dimensional world relative to the camera frame using the depth values and intrinsics [56].
The algorithmic steps involved in generating the point cloud, which will be used as inspiration for the manual registration process described in Section F.5.1, are shown in the list below. The steps in the list are performed in a loop iterating over all pixels within the depth image.

1. Grab the depth value, d, of the current depth pixel, (x_d, y_d).
2. Use the intrinsics of the depth image, $\boldsymbol{P}_{\text{depth}}$, to deproject the depth pixel into a 3-dimensional point, ${}^{\text{D}}\boldsymbol{p}$, in the depth camera frame.
3. Use the extrinsics between the depth camera frame and the RGB camera frame to translate the point into the RGB camera frame, ${}^{\text{C}}\boldsymbol{p}$.
4. Project this point, ${}^{\text{C}}\boldsymbol{p}$, onto the image plane of the RGB image by using the intrinsics of the RGB camera, $\boldsymbol{P}_{\text{RGB}}$ resulting. in pixel location (x_c, y_c).
5. Apply the distortion model to this pixel location, (x_c, y_c), to determine the distorted pixel location from which the pixel color can be grabbed.
6. Round the calculated distorted pixel location to get a quantified location and verify that this location is within the size of the RGB image.
7. Grab the pixel color from the distorted pixel location and apply to the 3-dimensional point within the RGB frame, $\boldsymbol{P}_{\text{RGB}}$.
8. Go to the next pixel in the depth image and repeat.

An example of a resulting point cloud is shown in Figure F.7.

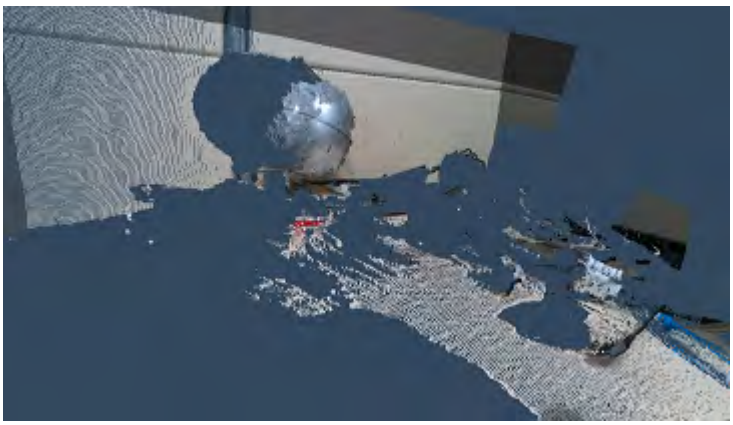


Figure F.7: Point cloud generated by librealsense library example provided by Intel [57]

F.4 Interfacing with ROS

Intel provides a C++ library and SDK to communicate, configure and grab images from the RealSense cameras. This library also includes several examples on how to use the built in functions, eg. aligning images, generating point clouds or doing other image processing on the RGB and depth images. The Intel Aero drone comes equipped with the Intel Aero compute board running a Yocto-based distribution of Linux including ROS. ROS includes many pre-developed features, libraries, algorithms, visualization and debugging tools and Intel has also developed a version of the librealsense library supported by ROS [58]. To link the library to the standardized ROS environment, already including a dedicated method to transfer images and configure cameras, Intel has developed a RealSense camera nodelet. This nodelet is a minimalistic ROS node whose only task is to read ROS camera configuration messages and pass these on to the librealsense library and furthermore redirect the incoming image streams from the librealsense library into the correct and dedicated image transport protocol within ROS [59].

On the Intel Aero drone the RealSense camera nodelet is started by first launching a local ROScore on the drone and thereafter launching the nodelet by calling:

```
1 roslaunch realsense_camera r200_nodelet_rgbd.launch
```

Code snippet F.1: Launching the RealSense nodelet on the drone with default configuration

This will use the default configuration resulting in an RGB image of 640×480 pixels and a depth image of 480×360. To change this resolution the width and height parameters should be adjusted within the 'r200_nodelet_rgbd.launch' launch file located on the drone within the folder: '/opt/ros/indigo/share/realsense_camera/launch'.

Having everything related to the R200 camera contained within ROS allows any node developed with the ROS environment to access the image streams coming from the RGB and depth camera in an easily accessible manner using the standardized image transfer protocol. It has been decided that both the controllers and the FastSLAM algorithm should be implemented as part of the ROS environment as individual ROS nodes. Implementing the whole system as a contained package within ROS thereby allows all the existing tools within ROS to be used to visualize the image streams and debug the developed processing, controller and FastSLAM nodes. Furthermore, it allows all tests to be recorded in a so-called ROSbag [60] for post-visualization or post-analysis.

All tests can thereby be performed separately and the analysis or processing, eg. the FastSLAM algorithm node, can be run and tuned after recording.

The distributed structure of ROS also allows the image streams to be streamed wirelessly and visualized on a connected computer. The 'rqt' toolset is a GUI based plugin set for ROS including several different visualization tools, including the 'rqt_image_view' [61] which can be used to visualize both the RGB and depth image streams. However 'rviz' can also be used for visualization, especially point clouds.

F.5 Image rectification & registration

Except for providing the non-distorted rectified depth image and distorted RGB image the RealSense camera nodelet also provides a registered depth image and RGB image. A registered set of images means that all the images have been aligned to the same frame and intrinsics, thereby sharing both extrinsics and intrinsics allowing them to be overlaid. The RealSense camera nodelet is hereby doing the iterative distortion removal as explained previously but in an optimized way as the library has been contained within a ROS nodelet, intended for performance optimization.

Figure F.8: Overlaid registered RGB and depth images generated by RealSense camera nodelet are not matching

Unfortunately the provided registered images, both being non-distorted and rectified and sharing the same intrinsics, do not seem to be registered correctly as seen by Figure F.8. Tests have shown that the registered depth image generated by the RealSense camera nodelet does not seem to be registered correctly on to the registered RGB image or vice versa [62].
To handle this issue, the registration of the images has been handled manually by a developed ROS node taking in the depth image and distorted RGB image. At a bare minimum the FastSLAM algorithm needs a map or link between RGB pixel points to corresponding depth points, such that a depth value can be found for detected ArUco marker pixel and that the depth image intrinsics can be used to deproject that pixel into the camera frame.

F.5.1 RGB and depth image registration for FastSLAM

Inspired by the example from Intel generating coloured point clouds, as explained in Section F.3, a 2-dimensional array of 3-dimensional measurement vectors is generated and aligned with the RGB image. The steps are similar to the steps from Section F.3 except that instead of grabbing the pixel color to store it as part of the 3-dimensional point, the 2-dimensional pixel location of the depth pixel is concatenated with the depth value to a 3-dimensional vector, which is stored in a 2-dimensional array at a location equal to the distorted RGB pixel location. If only the 3rd component of the measurement vector is used, this also allows a distorted greyscale depth image to be overlaid the RGB image for easy visualization of the depth measurements.

An example with the RGB and depth image from Figure F.4 and Figure F.5(b) is shown in Figure F.9.

Figure F.9: Overlaid registered RGB and depth images generated with developed rectification process

After processing all depth pixels, the result is a 2-dimensional array whose content corresponds to 3-dimensional measurement vectors for every given pixel location where a depth value has been available. This 2-dimensional array is aligned with the 2-dimensional RGB image array which allows the ArUco detector to find markers within the distorted RGB image and thereafter look up the corresponding 3-dimensional measurement vector to be provided to FastSLAM.
An example of an RGB and depth image with detected ArUco markers, visualized similarly to Figure F.9 but where the measurement vector has been printed for each detected marker, is shown in Figure F.10.

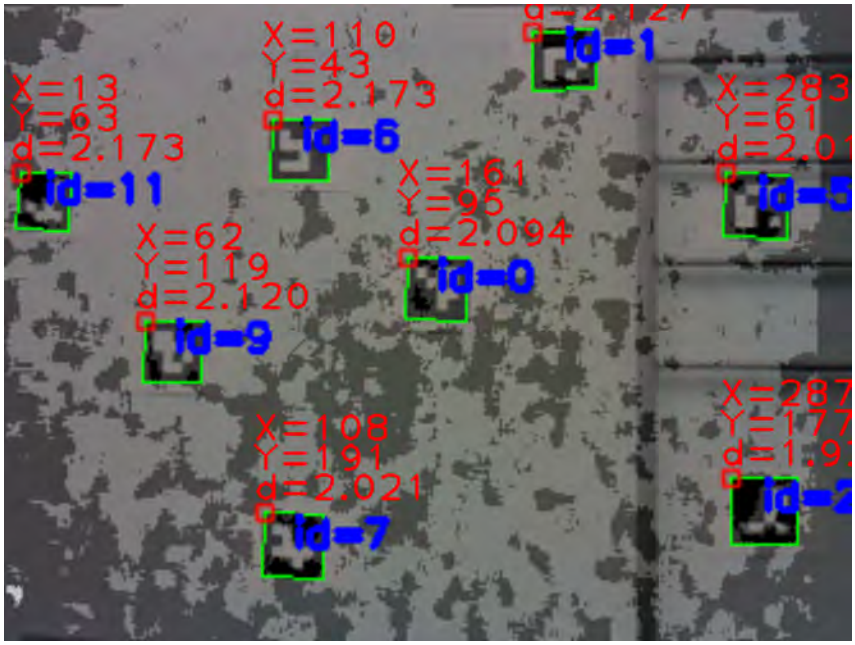


Figure F.10: Measurement vectors printed next to detected ArUco markers within a combined visualization of the RGB and depth image

To summarize, the pixel location of every detected ArUco marker within the distorted RGB image is used to lookup within a generated 2-dimensional array of 3-dimensional measurement vectors, to get the corresponding measurement vector for each detected marker, as long as a depth value exists at the location.

G Pin-hole camera model

A camera captures the environment through an optical lens, where a single point of an object in the environment will enter the lens at different places but all refract into the same point on the image plane a certain distance, f, behind the lens. The distance which corresponds to all rays originating from the same object meeting at the same point is defined as the focal length.

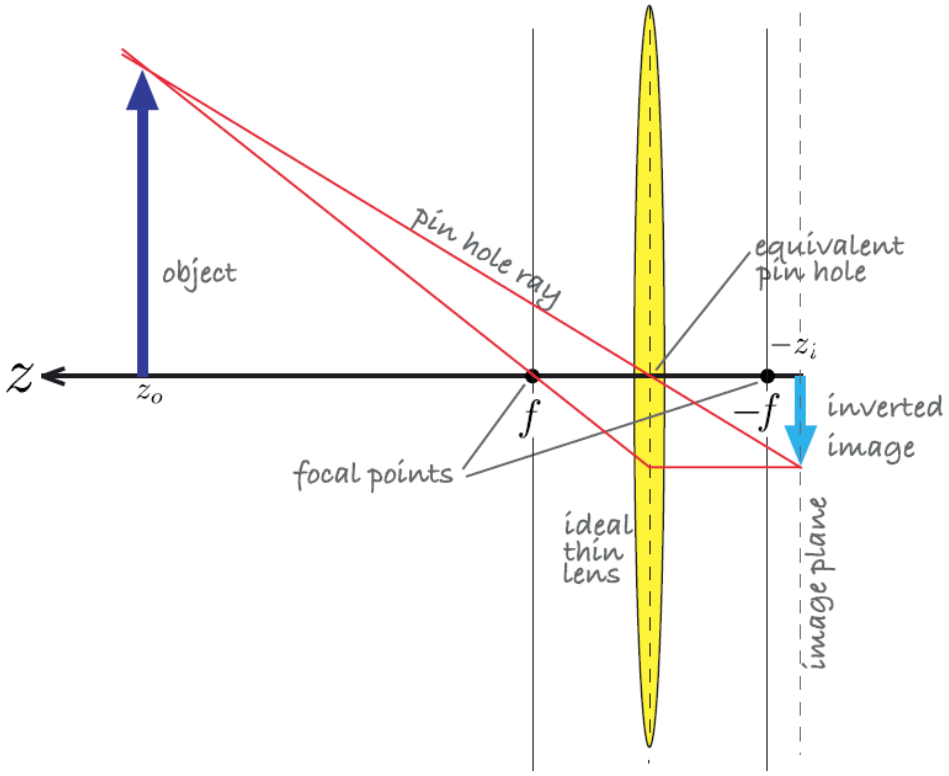


Figure G.1: Pin-hole model of a camera with a lens [63]

To simplify the modelling such a camera with an optical lens is usually modelled with the simple pin-hole camera model. With the pin-hole model only one specific light ray from each point in the environment will pass through the hole and result in a focused point on the image plane. As shown in Figure G.1 the light rays are captured by the image plane placed behind the lens, although flipped both horizontally and vertically. Mathematically this image plane can be moved in front of the lens such that all light rays passing through the camera origin would be projected onto this plane, creating the image.

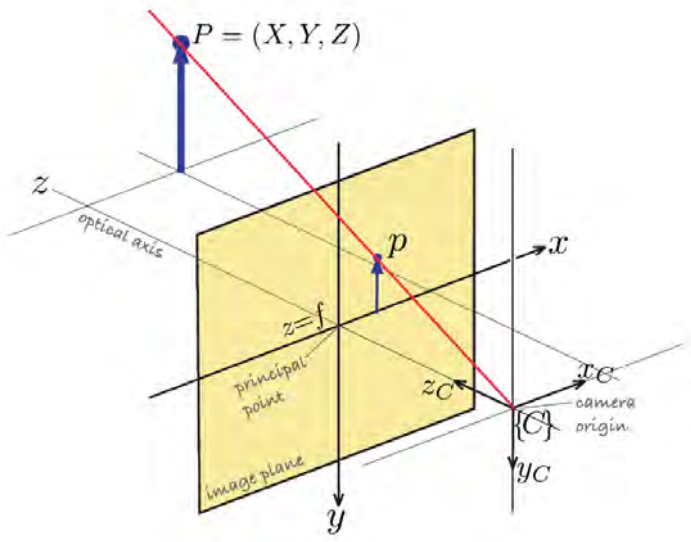


Figure G.2: Image plane projection using pin-hole model [63]

Notice how the coordinate frame of the image plane is defined as right-handed with the z-axis pointing in the direction of the viewpoint while the x-axis is positive to the right and y-axis is positive downwards. When projecting a point in the environment, $P = (X, Y, Z)$, on to the image plane, the world point coordinates must be defined within this coordinate frame.

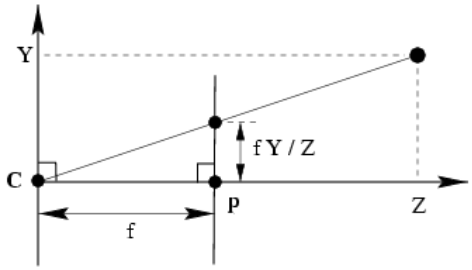


Figure G.3: Similar triangles defines the pin-hole model projection - Consider to change the axis direction such that it matches with the picture above [64]

Using similar triangles, as in Figure G.3, it can be shown that the world point is projected on to the image plane according to:

$$\begin{aligned} x &= f\frac{X}{Z} \\ y &= f\frac{Y}{Z} \end{aligned} \tag{G.1}$$

A projection matrix can be defined to describe this projection by using homogeneous coordinates to both represent the world point and the image plane coordinate. Homogeneous coordinates contains a scale component which makes them scale invariant. A world point is defined as

$$\boldsymbol{P} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{G.2}$$

and an image plane coordinate is defined as

$$\boldsymbol{p} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{G.3}$$

The projection matrix transforming from a world point into image plane coordinate is then given by:

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{G.4}$$

Notice that the image plane coordinates are given in metres. The image plane coordinates are transformed into pixel coordinates by applying a scale factor, k_x and k_y, defined by the manufacturer of the image sensor used to capture the image. The scale factor are usually multiplied with the focal length to form the horizontal and vertical scaling, defining how many pixels a 1 metre object at 1 metre distance will take up in the image.

$$\begin{aligned} a_x &= fk_x \\ a_y &= fk_y \end{aligned} \tag{G.5}$$

The center of the image plane is defined by the point where the optical axis intersects with the image plane as shown in Figure G.2. However pixel coordinates within an image are indexed from zero and up, why the first pixel of an image with coordinate $(0, 0)$ is formed in the upper left corner of the image plane. Thus an image plane offset, x_0 and y_0, has to be applied. For cameras where the lens is well-aligned on top of the image sensor the offset would correspond to half of the image resolution, but for cameras with slight lens offsets the image plane offset would have to be calibrated.

The full pin-hole camera model projection matrix is finally defined as:

$$\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} = \begin{bmatrix} a_x & 0 & x_0 & 0 \\ 0 & a_y & y_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{G.6}$$

Where:

u'	x-axis pixel coordinate
v'	y-axis pixel coordinate
w'	homogeneous depth

Finally the actual pixel value corresponding to the projected world point is found by converting the homogeneous pixel coordinate into the actual pixel coordinate:

$$\begin{aligned} x_c &= \frac{u'}{w'} \\ y_c &= \frac{v'}{w'} \end{aligned} \tag{G.7}$$

Both the focal length, coordinate scaling factors and the image center are usually parameters provided by the manufacturer or parameters which can be calibrated and determined manually. Altogether these parameters define the intrinsics of a camera.

H ArUco markers

From the SLAM implementation it arises that some kind of feature extraction is needed such that it is possible to extract distinct and unique identifiable features from the environment. From Section 3.3.1 it is known that the FastSLAM algorithm is fed with three dimensional landmarks. To extract 3D landmarks, an RGB-D camera is used. As introduced in Section 2.3 the kind of environment in which this project is based on, might be non-stationary. In such environment the feature extraction can benefit from using fiducial markers placed in the stationary regions of the given environment. This appendix is meant to introduce the different alternatives of fiducial markers, to elaborate the decision taken in Section 3.3.1 of using ArUco Markers and finally develop on how the feature extraction is implemented using these.

H.1 Fiducial markers

A fiducial marker is an element present in the field of view of an imaging system which allows to measure something in the real world trough the image obtained. Fiducial markers are used in many different fields, they can be used in metrology as a scaling tool, they can be used to improve Printed Circuit Boards (PCB) manufacturing or in AR among other applications.

Each application has its own kind of fiducial marker. The fiducial marker used for feature extraction must fulfil, at least, the requirements listed below:

- The marker has to be detectable and uniquely identifiable using an RGB-D camera.
- The marker needs to be scale invariant such that it can be identified at different distances.
- The marker needs to be rotation invariant such that can be identified at any rotation of it.

Because it is not the purpose of this project to develop specific markers for the feature extraction process it is reasonable to investigate the available markers used in AR and choose the one that better fits to the project.

[65] and [66] give an overview of the different available fiducial markers used in AR. If planar markers are considered, it makes sense to first take into account 2D barcodes because of its broad usage. A widely used type of 2D barcode is the quick response (QR) code [67]. Within QR codes also exist other subtypes for different applications. The standard QR code has a squared, shape, its minimum size is 21x21 modules which is able to store up to 152 bits, an example is shown in Figure H.1. Furthermore, QR codes are readable from any direction if the position patterns are used. These are placed in three of the corners of the QR codes. However, QR codes are intended for encoding of information not for localization. This yields problems when read from large distance or from views with perspective.

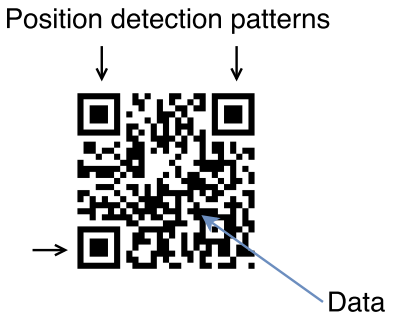


Figure H.1: A simple representation of the structure of a QR marker. The position detection allows to determine the rotation of the marker when detected and the rest is encoding data.

Fiducial markers are commonly used in augmented reality (AR) applications, where the requirements just mentioned also have to be fulfilled. One approach to be considered is to use circular markers as the ones presented in [68], this kind of markers usually only provide one point. Alternatively, one can use square-based fiducial markers, their main characteristic is that they can provide four correspondence points, one for each corner. This feature can be useful if the four correspondence points are taken as different landmarks. A popular system using this approach is the so-called ARToolkit [69] which is composed of a black border and an inner image which is then stored in a database. Later approaches based on the same principle are ARTag [65], ARToolkit Plus[70] and ArUco among others.

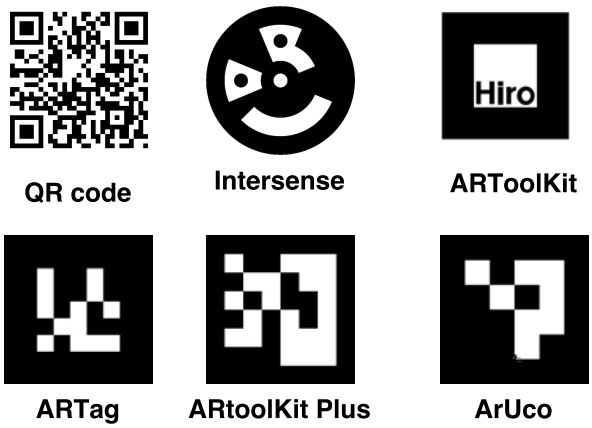


Figure H.2: Different types of fiducial markers that can potentially be used for the feature extraction process

It seems reasonable to choose square-based markers, QR codes are discarded due to its difficulties to be read from distance and from views with perspective. Among the rest of fiducial markers is the ArUco markers preferred since an OpenCV library is available. The OpenCV library allows fast integration with ROS and thus with the FastSLAM implementation.

H.2 Feature extraction with ArUco markers

The ArUco library presented in [66] propose a squared-based fiducial marker system with binary codes. Further than providing a method for detecting markers it also provides a method for generating configurable marker dictionaries with configurable size and number of markers. In fact, the dictionary is used to identify the markers in the image view.

ArUco markers are composed of a black frame and a binary grid of $n \times n$ bits within this frame. n is used to define the size of the marker, different size options are shown in Figure H.3. Each row in the binary grid is a binary word, thus, a marker with size $n = 4$ have four words four bits long.

The method to generate a dictionary is intended to maximize the Hamming distance between markers forming the dictionary. The Hamming distance between two markers is defined as the sum of Hamming distances between each pair of words. The Hamming distance [71] is, given two strings of the same length, the number of positions where the corresponding symbol is different, e.g. consider in this case two binary words, 1001 and 1100, its Hamming distance is 2 because the elements in position 0 and 2 are different from one word to the other. To introduce rotation invariance, this process is repeated for each 90 degrees rotation of the marker.

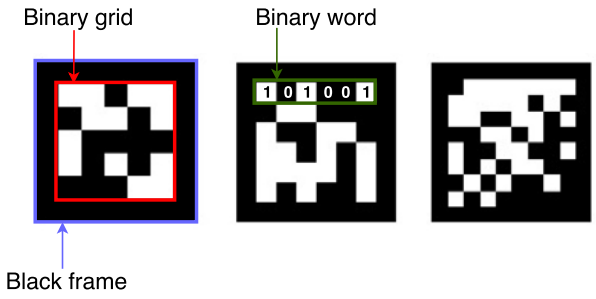


Figure H.3: Three different sizes of ArUco markers, from left to right sizes are 5×5, 6×6 and 8×8. It is also indicated the black frame and the binary grid that compose this type of markers.

Within the implementation in OpenCV three options are available to generate markers from a dictionary. The first and simple option is to use predefined dictionaries with configurable marker size from $n = 4$ to $n = 7$ and configurable dictionary size from 50 to 1000 markers. Another option is to automatically generate a custom dictionary with customizable marker size and customizable dictionary size. Last option is to the manually create a dictionary using the class Dictionary. For the ease of the implementation it is decided to chose a predefined dictionary. It is recommended to choose small dictionaries and big markers. Thus, the dictionary of 50 markers is chosen as it is the smallest predefined dictionary available, with markers of size $n = 4$ for the ease of the marker's recognition from long distances. The definition of the used dictionary in the OPenCV is shown in line 3 of the Code snippet H.1.

Once the dictionary is set it is possible to proceed with the marker detection process which is split in two phases:

1. Image process; phase where, from the image given, candidates to be markers are recognized.

2. Identification of valid markers; phase where the detected markers are identified if part of the generated dictionary or discarded otherwise.

Both phases are encapsulated within one command in the OpenCV, a call example of this command can be found at line 5 of the code snippet shown in the Code snippet H.1. For that implementation one have to specify the dictionary containing the used markers. It is also possible to adjust the detector parameters which allow to tune the detection process, in this case the default parameters have been used, no further tuning has been done as the results have been satisfactory.

```
1 #include <opencv2/aruco.hpp>
2
3 cv::aruco::Dictionary markerDictionary =
     cv::aruco::getPredefinedDictionary(cv::aruco::DICT_4X4_50);
4
5 cv::aruco::detectMarkers(RGB_Image, markerDictionary, markerCorners,
     markerIds);
6
7 cv::aruco::drawDetectedMarkers(blended, markerCorners, markerIds);
```

Code snippet H.1: Implementation commands performing the ArUco marker detection process. A predefined dictionary with size 50 and marker size 4 has been used. The marker detection is performed with the default parameters and finally the markers are drawn for visualisation.

The image process defined by the ArUco marker system is developed in [66] and its OpenCV implementation details are described in [15]. This process is divided in different steps:

1. **Image segmentation**: it is done using a local adaptive thresholding approach, an example is shown in Figure H.4(b).

2. **Controur extraction**: it is desired to detect candidates to markers, for that, a contour extraction is done, its result is shown in Figure H.4(c) where it can be seen that most of the contours detected are irrelevant, for that, first a size filter is applied, in the OpenCV implementation a minimum and maximum perimeter of the marker can be specified for that filtering, the default values are 0.03 as the minimum and 4.0 as the maximum.

3. **Polygonal approximation**: It is performed such that the polygons that are not estimated as 4-vertex polygons are discarded, the result then is shown in Figure H.4(d).

4. **Code Extraction**: once the perimeters have been detected, see Figure H.4(e). The obtained image without perspective is then thresholded and the resulting binary image is divided into a regular grid where each element is set to zero or one depending on the pixel values within each element, this can be seen in Figure H.4(e). Realize that in Figure H.4(d) some error contours are still detected, now that the binary grid of each element is known, most of these error contours can be discarded if they do not present a grid which border is all zeros.

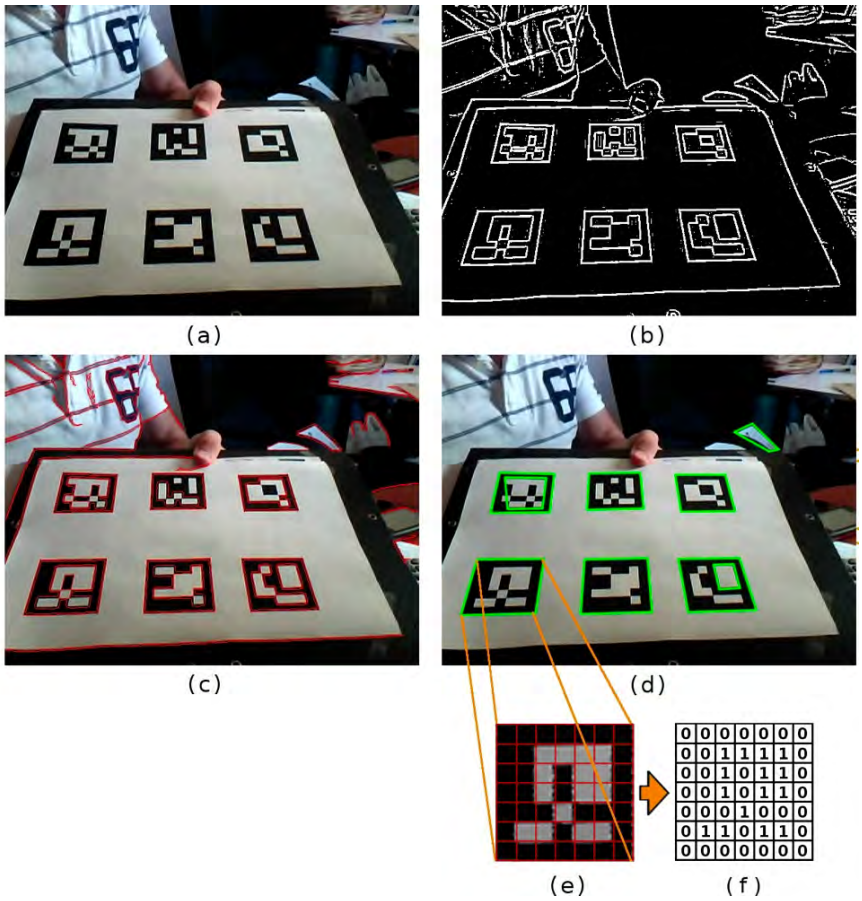


Figure H.4: From [66] it describes the different steps in the image processing where: (a) shows the original input image, (b) shows the image segmentation after an adaptive thresholding, (c) shows the result of the contour extraction, (d) shows the result of the 4-vertex polygonal approximation, (e) shows the result of the projection removal divided into a regular grid and (d) shows the corresponding marker code extraction.

At this point the image process phase is finished and the marker identification proceeds. Whenever a candidate marker code is obtained, four different identifiers are computed, one for each rotation. Just that one of these is found in the set dictionary it is directly considered as a valid marker.

When the marker detection process is finished, it outputs the pixel coordinates of the four corners of the found marker. For visualization purposes it is possible to draw the contour of the found markers in the input image using the OpenCV implementation, line 7 in the Code snippet H.1 shows an example of the command call used, the result of this visualisation is shown in Figure H.5.

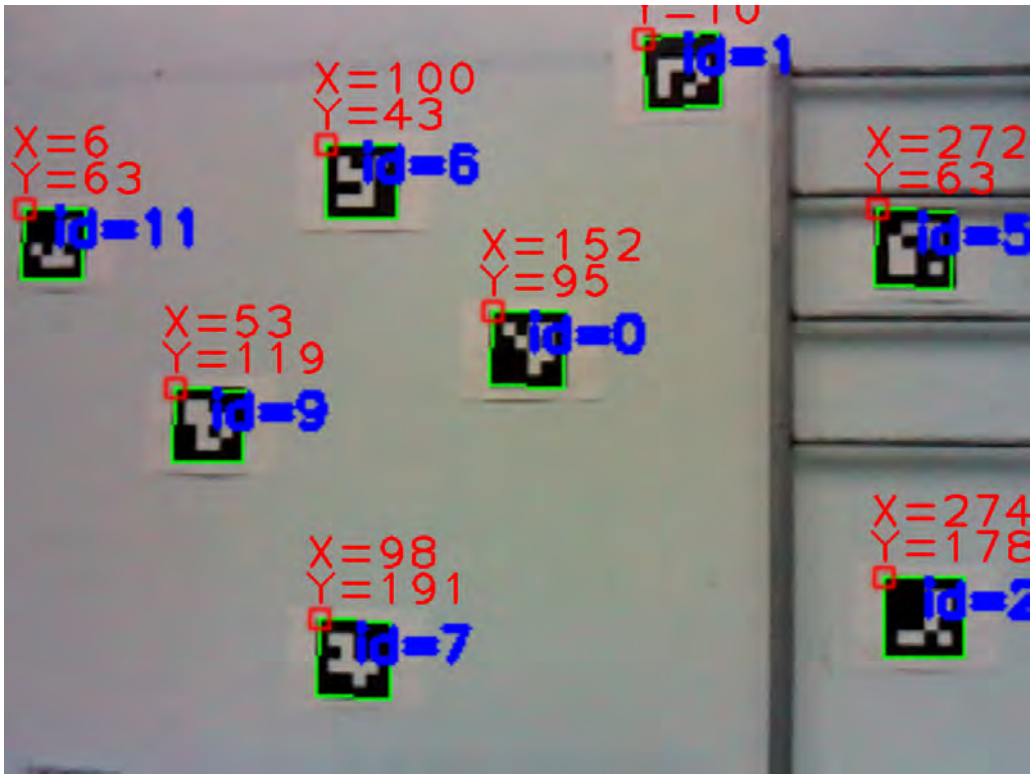


Figure H.5: Implementation of the ArUco markers detection is done in ROS. ArUco markers are printed and placed in the walls. The image shows the detection and identification of some of the markers placed in the laboratory. The implementation streams the marker identifier and its pixel position for visualisation purposes.

ArUco markers are printed in paper of size DIN-A4 and placed on the walls of the laboratory. The size of each printed marker is $17.5 \times 17.5cm$. Result of this implementation is shown in Figure H.5.

I ROS

Robotics Operating System (ROS) is considered as a flexible framework that aims to simplify the writing of software for robots. It is defined in [72] as a meta-operating system, called so because it provides the services expected from an operating system such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management, but still need another operating system to work.

The implementation of this project in the given platform has been done using ROS. This appendix is intended to give an overview of ROS and how it has been used in this particular implementation.

I.1 Introduction to ROS

ROS computational graph is a peer-to-peer network of different processes that are running simultaneously. It is basically composed of a Master and different nodes, the master works as a lookup for the rest of the computational graph while the nodes are the different processes performing computation which can interact by communicating with each other by passing messages. A message is nothing else than a data structure, this data structure varies over different kind and can include different standard primitives such as integers, floating point and boolean among others, the structure can be defined for a particular application. ROS includes different styles of communication, it includes synchronous communication which is done over services and also includes asynchronous communication of data, which is done through topics. These are some of the basic computational graph concepts of ROS and will be further explained within this section.

Master

The ROS master is meant to provide naming and registration services for the nodes in the ROS system. The ROS master keeps track of the different nodes existing in the computational graph and the topics that they subscribe/publish as well as services. The ROS master is used by the nodes as a lookup so they can locate the different nodes in the computational graph. In other words, it enables the communication between nodes. To be able to run a ROS system, a ROS master is always needed.

Launching a ROS master can be done by running the command-line tool `roscore` in the bash shell, which will also load other essential components for the ROS system.

Nodes

Nodes are the processes performing computation within the computational graph. Nodes communicate with each other through topics and services. A robot control system can imply several nodes, for example, in this project one node is the one running the controllers, another node is running the EKF, another one is running the camera and so on. Figure I.1, Figure I.2 and Figure I.3 shows a representation of the computational graph used in this project, including the active nodes.

ROS nodes are written using a ROS client library, such as roscpp and rospy, a C++ and a python implementations of ROS respectively. Notice that it is possible to use roscpp to write one node and rospy to write another node within the same computational graph.

ROS provides some command-line tools under the call of `rosnode`, it can provide information about the nodes, for example, `rosnode list` generates a list of active nodes and `rosnode info` provides information of a specified node.

Messages

Nodes communicate with each other by exchanging messages. A message is a data structure comprised of standard primitives such as integer, boolean, floating point among others. These data structures can be specified using .msg files.
ROS provides some command-line tools under the call of `rosmsg`, some of the available commands are `rosmsg show` which will display the fields of a particular ROS message type or `rosmsg list` which displays a list of all messages.

Topics

Topics are named buses and are used in ROS systems for asynchronous communication between ROS nodes. A node can send out a message by publishing it to a given topic. When a node is publishing in a topic it is registered by the master, thus other nodes can see that this topic is being published by this node and read the messages in the topic by subscribing to it. There is no limit of nodes publishing or subscribing to a topic, thus there may be multiple concurrent publishers and subscribers for a single topic. Moreover, a single node can publish and/or subscribe to multiple topics.

ROS provides some command-line tools under the call of `rostopic`, some of the available commands are `rostopic list` which will display a list of the active topics and `rostopic type` which displays the topic type of a topic. To display the messages published in a particular topic the command `rostopic echo` can be used. It is also possible to publish into a topic from the command-line using `rostopic pub`.

Services

Services are used for synchronous communication between nodes. Compared to the publish/subscribe model, services follow a request/reply model. A service is defined by two messages, one doing the request and the other one the reply. Services are specified with .srv files, which are compiled into source code by a ROS client library.

ROS provides some command-line tools under the call of `rosservice`, some of the available commands are `rosservice list` which displays a list of active services, `rosservice type` which displays the type of a particular service or `rosservice call` which one can use to call a particular service with specified arguments.

I.2 Debugging tools

To ease the writing of code for robot control systems, different debugging tools are available within the ROS environments to help the user. Some of these tools that have been used in this

project are rqt, rviz and rosbag. This section will give a short overview of these tools and explain how they have been used in this project.

rqt

rqt is a software framework of ROS that implements various GUI tools in the form of plugins. Among several of the available tools within rqt, mainly 4 of them have been used in the project.

One of these tools is `ros_graph`, a tool used to visualise the computational graph. Elements such as nodes and topics are shown. The two computational graphs generated by `ros_graph` is seen in Figure I.1 and Figure I.3. The ellipsoids represent the nodes and the rectangular boxes represent topics. If an arrow is going out of a node it means that the node is publishing into one topic, otherwise, if an arrow points into the node, it means that this node is subscribing to a topic.

It is seen in Figure I.1 that the ROS system running the controllers involve three nodes, one node is providing the interface with the VICON system and it is publishing the drone pose generated from the VICON system. Another node is running the controllers and the third one is the Mavros node which interfaces to the PX4 controller.

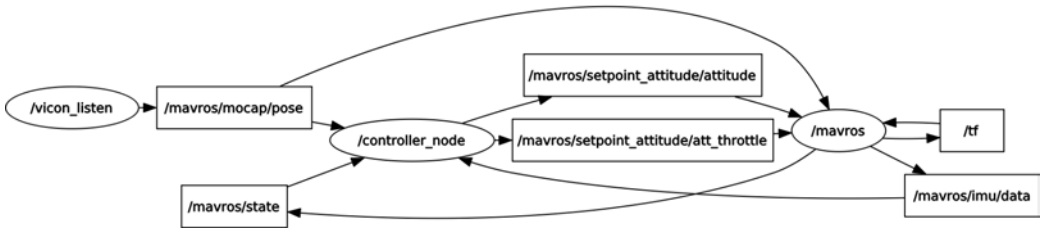


Figure I.1: Computational graph generated with `ros_graph` of the ROS system including the controllers

Two different scenarios are given in the ROS system running FastSLAM, one scenario when GOT measurements are provided, in this project emulated by VICON measurments, and a scenario when GOT measurements are not provided. Each scenario has its own computational graph which are respectively shown in Figure I.2 and Figure I.3.

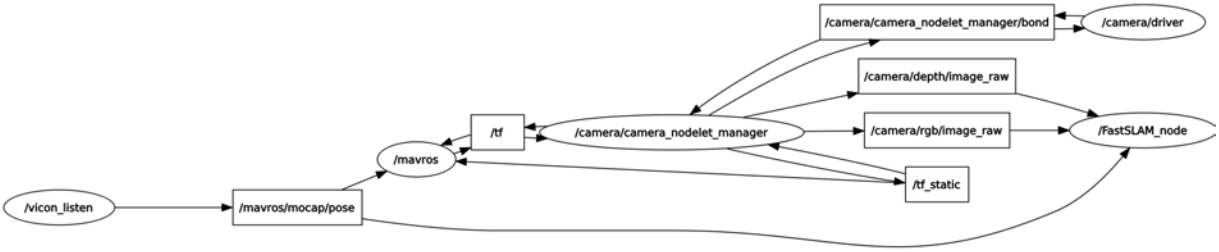


Figure I.2: Computational graph representation generated with `ros_graph` of the ROS system running FastSLAM with VICON measurements

The computational graph of the scenario where VICON measurements are available, shown in Figure I.2, is composed of five nodes. As it is for the ROS system running the controllers, there is one node for the VICON system and one for Mavros. Furthermore one node is running the

FastSLAM algorithm, which is also doing the image processing. That is why the FastSLAM node is subscribing to the RGB and depth topics published by the camera node. Notice that the fifth node is just running a camera driver.

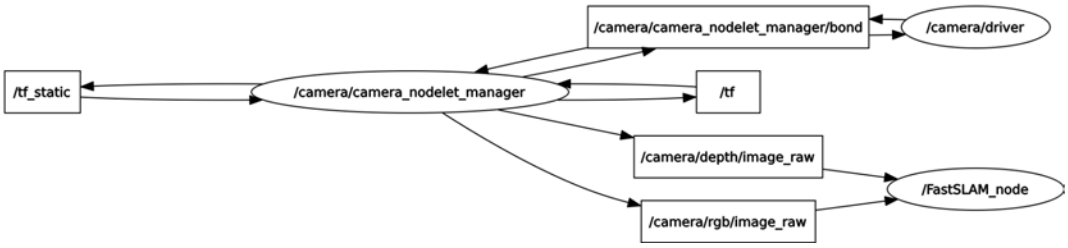


Figure I.3: Computational graph representation generated with `ros_graph` of the ROS system running FastSLAM without VICON measurements

The computational graph of the scenario when VICON measurements are not available, shown in Figure I.2, is composed of three nodes. Similarly to the ROS system running FastSLAM with GOT measurements, two nodes are running the camera while FastSLAM is running in one other node. It is evident that the node running VICON is not active and because there is no direct link between FastSLAM and the PX4 flight controller, Mavros is not active neither.
Moreover rqt has been used for image view visualisation, i.e. a tool to visualise the published image views such as the RGB image or the depth published by the camera node. rqt has also been used to setting up the camera parameters and for rosbag visualisation.

rviz

rviz is a 3D visualization environment which allows to see what the robot is seeing whether using cameras, laser scanners or joint encoders. It can also be used for point cloud visualisation from the RGB-D camera. It helps the user debugging its implementation and see if the RGB-D camera is well set-up.

rosbags

rosbags are files in ROS, intended to store ROS message data. This data can then be processed, analysed and visualized later. It provides command-line tools such as `rosbag record`, which subscribes to topics and writes to a bag file with the contents of all messages that have been published in the topic. Another command-line tool is `rosbag play`, which enables to reproduce bag files by reading the content of one or more bag files such that the messages stored in the bag file are published in the respective topics.

rosbags have been used in the project, mainly to debug the FastSLAM implementation. To do that flights with the drone have been recorded with rosbags, including VICON measurements and video recorded by the RGB-D camera among other topics.

J Simulation with Gazebo & SITL

Simulation has been used for testing along the development of the project, i.e. test of controllers. For that purpose it has been decided to use the Software-in-the-Loop simulator (SITL) provided by the PX4 autopilot software. This allows to simulate the PX4 flight controller together with the developed software. To test the ROS implementation, SITL is available as a plugin for Gazebo. Thereby making it possible to take into account the interface between the ROS implementation and the PX4 flight controller through MAVROS and Mavlink. The connection between MAVROS and Mavlink is done in this case through UDP. Moreover, the use of Gazebo makes it possible to include sensors such as an IMU or an RGB-D sensor. A representation of how the implementation looks when using sim

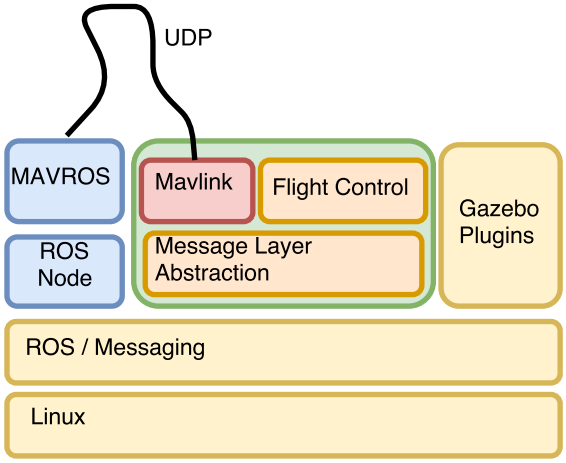


Figure J.1: Implementation overview using SITL and Gazebo interfaced with ROS [73]

J.1 Gazebo

Gazebo is a physics simulator known for its usage in robotics simulation due to its relation with ROS. This section is intended to give a brief introduction to the different elements forming the simulator and how it has been defined in the present project.

As stated in Chapter 5, the version of ROS used in this project is Indigo due to the installation already existing in the Intel® RTF drone. The Gazebo version used, is version seven since the PX4 SITL plugin is made for Gazebo version six or seven. Version two of Gazebo is installed, along this ROS indigo installation. Hence, the Gazebo version two has to be remove, and Gazebo version seven (or six) has to be installed from source instead.
A Gazebo simulation is composed of multiple components. Three of these components is described in this section. These components are:

- World files
- Model files
- Plugins

J.1.1 World files

World description files are used to define the simulation environment. These can include descriptions of the lighting, the ground and other static elements, like buildings obstacles and sensors. Furthermore, a world file describes how these model files are placed in simulation and how they interact with each other. World files are formatted using Simulation Description Format (SDF) [74] and typically have a *.world* extension.
Two different worlds are considered in this project, both of them inspired from the empty.world provided by the PX4 SITL. The first world used is directly the empty.world file which is composed of an asphalt ground and sun light. The ground contains a description of its appearance and collision properties. The collision properties of the ground are important since they define a plane at which the drone can rest without free falling. This world has been used to test implementation of the Z controller, the EKF and the X-Y controller. The visualisation outcome using the Gazebo graphic user interface (GUI) is shown in Figure J.2.

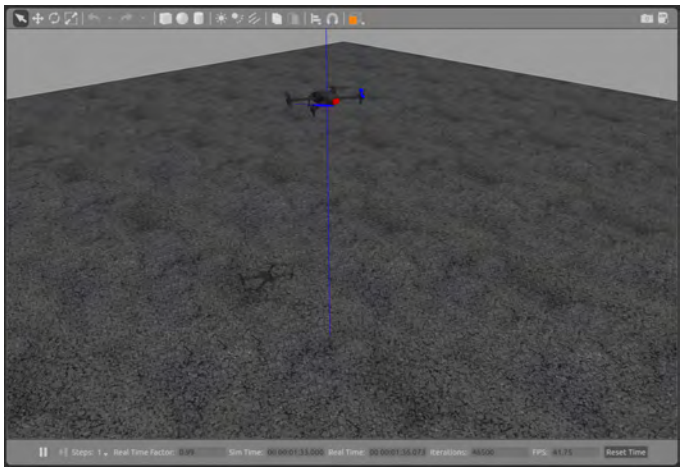
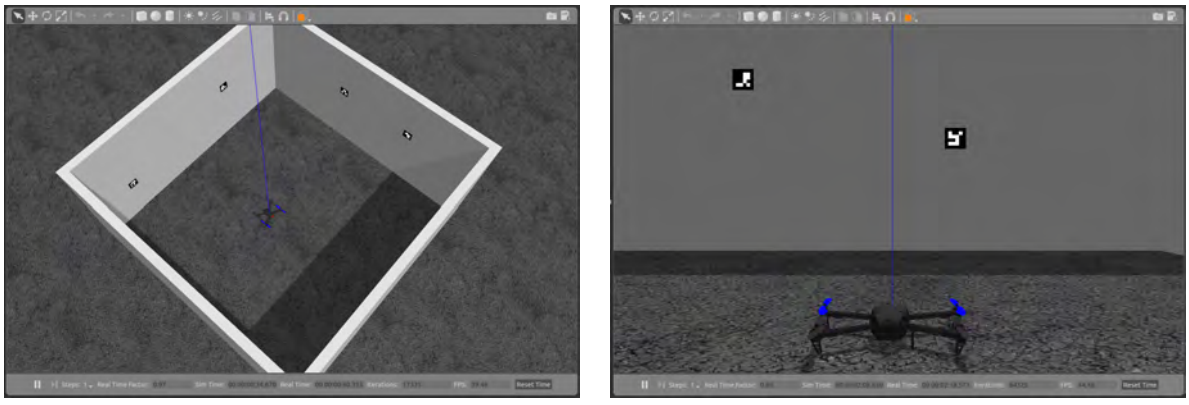


Figure J.2: *empty.world* file visualisation in Gazebo GUI including sun light and ground

The second world file used is the same empty world but including a representation of a square room with ArUco markers placed on the walls. This world is intended to test implementation of the FastSLAM algorithm and the full implementation. The visualisation outcome in Gazebo GUI is shown in Figure J.3.



(a) (b)

Figure J.3: Two different views of the world file visualisation in Gazebo GUI where it is included a square room with ArUco markers.

J.1.2 Model files

Model files are used to describe the individual elements used in the simulation.

Gazebo uses SDF files for the description of model files. However, they can also be written in Universal Robot Description Format (URDF) but Gazebo will first convert those to SDF before it is used by Gazebo, this can be done because URDF is a useful and standardized format for describing robot models in ROS.

Model files describes any kind of element that has to be included in simulation. One good property of those is that they are stackable, an example would be the drone shown in Figure J.2 and Figure J.3 which includes four motors, here only one model file describing one motor is needed and it can be included four times in the model file of the drone. This property allows to reuse model files, e.g. this model file describing a motor can be used in other drones. In fact, a world file is nothing else than a collection of several model files, which can be imported or directly defined in the world file. For example, in the present implementation, the drone, the ground, the sun and the square room are described in different model files and then included into the world file. Each ArUco marker is described directly in the world file instead.

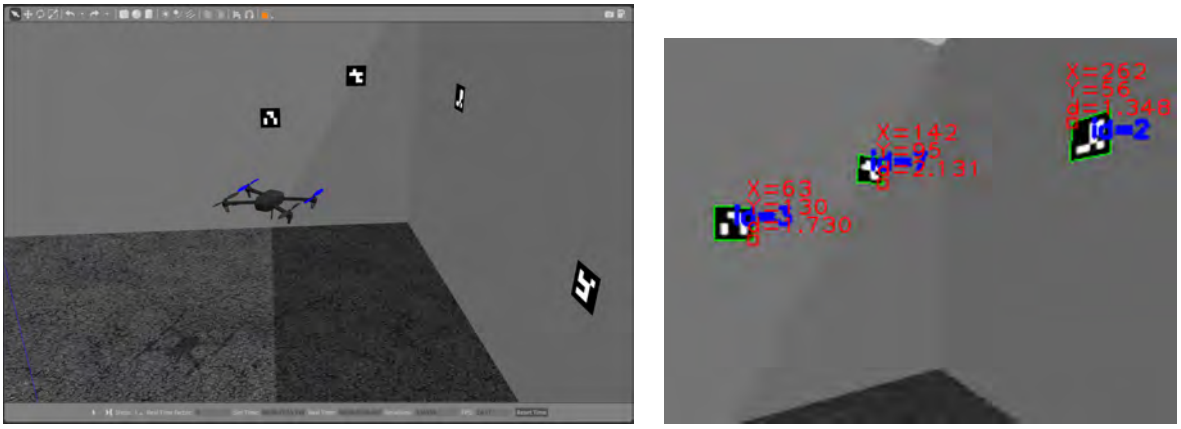
The SITL plugin is providing a model of the Iris drone, a similar drone to the Intel® RTF drone. Due to time constraints it is decided to use the Iris drone in simulation rather than making a model of the Intel® RTF drone. Although this Iris drone model does not include nor an RGB camera nor a depth sensor, it is possible to add an RGB-D camera with the proper plugin to emulate the Intel® Realsense camera mounted in the Intel® RTF drone.

J.1.3 Plugins

Plugins are a simple mechanism to interface with the Gazebo simulation. Some plugins are already given with the Gazebo installation, but it is also possible to develop new plugins to interface with Gazebo in a particular way. The different plugins can be classified as:

- World
- Model
- Sensor
- System
- Visual
- GUI

The 3DR® Iris drone Model includes plugins such as the motor plugin, the IMU plugin and a plugin for mavlink among others. On top of those, it has been necessary to include a plugin for the RGB-D camera. Among the plugins included within the Gazebo installation, there is a plugin that emulates a Kinect camera [75], an active RGB-D sensor similar to the Intel® Realsense camera. For the ease of the implementation it is decided to use this Kinect plugin instead of making a new one for the Intel® Realsense camera. This may differ slightly but is not expected to affect the controllers nor the FastSLAM algorithm implementation. It can be seen in Figure J.4 how the camera view is seen in simulation, together with the implementation of the ArUco markers identification.



(a) (b)

Figure J.4: Image view in simulation using a plugin for a Kinect camera, an RGB-D camera similar to the Intel® Realsense camera. Figure J.4(a) shows the Gazebo GUI visualisation and Figure J.4(b) shows the image view show the ArUco marker identification and it displays the respective identifier with its measurement vector.

J.2 Software-in-the-Loop

Software-in-the-Loop is a common choice for testing implementation of software in simulation such that the hardware is not compromised. The hardware involved in this project is a drone, which is a rather expensive piece of hardware and implementation test could easily imply crashing it. Hence, it is deemed necessary to use SITL. For that, a SITL provided by PX4 is used, this makes it possible to use the PX4 flight controller together with simulated sensor data generated by the flight simulator, in this case Gazebo.

K Getting started guide

This appendix is intended as a guide which explains the installation steps involved in getting the drone up and running in simulation. The guide is tested to work on Ubuntu 14.04 LTS.

K.1 Prerequisites

A few programs need to be installed before the guides can be followed. First program is git which is installed with the following command in Code snippet K.1.

```
1 sudo apt-get install git
```

Code snippet K.1: Command to install git from a terminal.

Furthermore a few build tools need to be installed in order to compile the PX4 autopilot software in the loop simulation. This is obtained with the following commands in Code snippet K.2

```
1 sudo add-apt-repository ppa:george-edison55/cmake-3.x -y
2 sudo apt-get update
3 sudo apt-get install python-argparse git-core wget zip python-empy
    qtcreator cmake build-essential genromfs -y
4 sudo apt-get install ant protobuf-compiler libeigen3-dev libopencv-dev
    clang-3.5 lldb-3.5 python-jinja2 -y
```

Code snippet K.2: Commands to install prerequisite the build tools necessary to compile PX4 SITL.

K.2 ROS Indigo

ROS Indigo can be installed from a Debian package. This is achieved by using the commands in Code snippet K.3

```
1 sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release
    -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
2 sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net --recv-key
    421C365BD9FF1F717815A3895523BAEEB01FA116
3 sudo apt-get update
4 sudo apt-get install ros-indigo-desktop-full
5 sudo rosdep init
6 rosdep update
```

Code snippet K.3: Commands to install ROS Indigo from a Debian package

A few environment variables have to be set in the shell environment before the ROS installation can be used. These environment variables can be sourced with the following command in Code snippet K.4

```
1 source /opt/ros/indigo/setup.bash
```

Code snippet K.4: Command to set the environment variables in the shell.

The environment variables have to be sourced in every terminal session. Run the following command in Code snippet K.5 to source them automatically in every new terminal session.

```
1 echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
```

Code snippet K.5: Command to set the environment variables for every new shell.

Rosinstall has to be installed also. It is a tool for downloading many source trees for ROS packages with one command. Rosinstall will be used throughout the guide. Rosinstall is installed with the following command in Code snippet K.6.

```
1 sudo apt-get install python-rosinstall
```

Code snippet K.6: Command to install rosisnstall.

K.3 Catkin tools

Catkin tools is a collection of commands which is useful for working with and compiling source code within ROS workspaces. These tools are used throughout the guide and they are installed from a debian package with the following commands in Code snippet K.7.

```
1 sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu `lsb_release -sc`
     main" > /etc/apt/sources.list.d/ros-latest.list'
2 wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
3 sudo apt-get install python-catkin-tools
```

Code snippet K.7: Commands to install catkin tools.

K.4 Gazebo 7

Gazebo 2 is installed along the installation of ROS Indigo. But the software in the loop simulation of the PX4 autopilot is made for Gazebo 7 and not Gazebo 2. Gazebo 7 should therefore be installed also with the following commands in Code snippet K.8.
Gazebo 7 can be installed from a Debian package. With the following commands.

```
1 sudo sh -c 'echo "deb
     http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb_release
     -cs` main" > /etc/apt/sources.list.d/gazebo-stable.list'
2 wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
3 sudo apt-get update
4 sudo apt-get install ros-indigo-gazebo7-ros-pkgs
     ros-indigo-gazebo7-ros-control gazebo7 libgazebo7-dev
```

Code snippet K.8: Commands to install Gazebo 7 from a Debian package.

K.5 MAVROS

MAVROS is a ROS node. In order to get the latest version of MAVROS it has to be compiled from source code and installed within a ROS workspace. To install MAVROS a ROS workspace should be created. In this guide the workspace will be installed in the home folder. The ROS workspace is created with the following commands in Code snippet K.9.

```
cd ~/
mkdir catkin_ws
cd catkin_ws
mkdir src
cd src
catkin_init_workspace
```

Code snippet K.9: Commands to set-up a ROS workspace

In order to complete the installation of MAVROS a few python tools need to be installed. This is done with the commands in Code snippet K.10

```
sudo apt-get install python-wstool python-rosinstall-generator
   python-catkin-tools python-pip
sudo pip install future
```

Code snippet K.10: Commands to install necessary python tools for the MAVROS installation

Finally, proceed with the installation of MAVROS within the ROS workspace, following the commands in Code snippet K.11

```
wstool init ~/catkin_ws/src
rosinstall_generator --upstream mavros --rosdistro kinetic | tee -a
   /tmp/mavros.rosinstall
rosinstall_generator --rosdistro kinetic mavlink | tee
   /tmp/mavros.rosinstall
cd ~/catkin_ws
wstool merge -t src /tmp/mavros.rosinstallcd
wstool update -t src
rosdep install --from-paths src --ignore-src --rosdistro indigo -y
```

Code snippet K.11: Commands to install MAVROS within a ROS workspace.

K.6 Software in the loop simulation

The software in the loop simulation of PX4 autopilot should be installed from source within the ROS workspace created during the MAVROS installation. This is obtained with the following commands in Code snippet K.12.

```
cd ~/catkin_ws/src
git clone https://github.com/PX4/Firmware.git
cd Firmware
git submodule update --init --recursive
make posix_sitl_default gazebo
```

Code snippet K.12: Commands to install the Software in the loop simulation from source.

K.7 Installation of the project code

The source code developed by the project group should be installed within the same ROS workspace as well. This is done with following steps in Code snippet K.13

```
cd ~/catkin_ws/src
git clone https://github.com/davidromanos/intel_aero_rtf_gr871.git
cd ~/catkin_ws
catkin build
```

Code snippet K.13: Commands to clone the repository containing the project code and and build it within the ROS workspace.

K.8 Execution of the programs

To start the software in the loop simulation in Gazebo, open a terminal and source the environment variables and source the necessary set-up files with the following commands in Code snippet K.14.

```
cd ~/catkin_ws
source devel/setup.bash
source src/Firmware/Tools/setup_gazebo.bash $(pwd)/src/Firmware
    $(pwd)/src/Firmware/build_posix_sitl_default
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$(pwd)/src/Firmware
export
    ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$(pwd)/src/Firmware/Tools/sitl_gazebo
```

Code snippet K.14: Commands to set-up the proper ROS environment before launching the simulation.

Once the ROS environment is set, in the same terminal the software in the loop simulation is launched in Gazebo together with MAVROS with the following command in Code snippet K.15

```
roslaunch px4 mavros_posix_sitl.launch
```

Code snippet K.15: Command to launch the software in the loop simulation in Gazebo together with MAVROS.

In the simulated environment is GOT or Vicon motion capture system implemented with a node called true position. This node has to be started before the controller node is started. The true position node is started in a new terminal session with the following commands in Code snippet K.16.

```
cd ~/catkin_ws
source devel/setup.bash
rosrun intel_aero_rtf_gr871 true_position
```

Code snippet K.16: Command to initialize the node running the motion capture system.

The controller node running the Extended Kalman filter and the controllers is started in a new terminal sessions with the following commands in Code snippet K.17.

```
cd ~/catkin_ws
source devel/setup.bash
rosrun intel_aero_rtf_gr871 controller
```

Code snippet K.17: Commands to initialize the node running the Extended Kalman filter and controllers