University of Agder

Master

# State Estimator using Hybrid Kalman and Particle Filter for Indoor UAV Navigation

Kristoffer Hansen Kruithof, Marius Egeland

SUPERVISOR
Sondre Sanden Tørdal, *UiA*

CO-SUPERVISORS
Kristian Muri Knausgård, *UiA*
Nadia Saad Noori, *Norce*

**University of Agder, 2021**
Faculty of Engineering and Science
Department of Engineering and Sciences

Page intentionally left blank

## Obligatorisk gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

| | | |
|---|---|---|
| 1. | Vi erklærer herved at vår besvarelse er vårt eget arbeid, og at vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen. | **Ja** / Nei |
| 2. | **Vi erklærer videre at denne besvarelsen:**<br>• Ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands.<br>• Ikke refererer til andres arbeid uten at det er oppgitt.<br>• Ikke refererer til eget tidligere arbeid uten at det er oppgitt.<br>• Har alle referansene oppgitt i litteraturlisten.<br>• Ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse. | **Ja** / Nei |
| 3. | Vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen §§ 31. | **Ja** / Nei |
| 4. | Vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert. | **Ja** / Nei |
| 5. | Vi er kjent med at Universitetet i Agder vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens retningslinjer for behandling av saker om fusk. | **Ja** / Nei |
| 6. | Vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider. | **Ja** / Nei |
| 7. | Vi har i flertall blitt enige om at innsatsen innad i gruppen er merkbart forskjellig og ønsker dermed å vurderes individuelt. Ordinært vurderes alle deltakere i prosjektet samlet. | Ja / **Nei** |

## Publiseringsavtale

Fullmakt til elektronisk publisering av oppgaven Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2).
Oppgaver som er unntatt offentlighet eller taushetsbelagt/konfidensiell vil ikke bli publisert.

| | |
|---|---|
| Vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering: | **Ja** / Nei |
| Er oppgaven båndlagt (konfidensiell)? | Ja / **Nei** |
| Er oppgaven unntatt offentlighet? | Ja / **Nei** |

Page intentionally left blank

# Preface

This thesis is submitted as part of the Mechatronics master's programme at the University of Agder (UiA).

The submission of this thesis marks the end of five solar circumnavigations[1] of hard work, learning and general fun at UiA; and thus the beginning of a new epoch in our lives.

We would like to thank our supervisor Sondre Sanden Tørdal(UiA) and co-supervisors Kristian Muri Knausgård(UiA) and Nadia Saad Noori(Norce) for their continued support, advice and council throughout this project.

A general *thank you* is also reached out to staff, faculty and co-students at UiA for being available for discussion, council and general help and advise throughout our five years at the University.

A video of the Hybrid-filter in operation is available at:

`https://youtu.be/pDOOLkh2-aE`.

Further, all code written and used in the project is available on gitlab, at:

`https://gitlab.com/master_monkeys`

[1]About 0.0282 of an arc second around the galactic center

Page intentionally left blank

# Abstract


Unmanned aerial vehicles (UAVs) are being used for outdoors inspection and surveying tasks. When operating in an outdoor environment, the global navigation satellite system (GNSS) is predominantly used for position aiding, and magnetometers are used for heading aiding. In combination with an inertial sensor, these sensors form the backbone for state estimation for drones operating in an outdoor environment.

A desire to utilize UAVs for inspections in indoor environments means that new challenges are faced. One of these challenges is that the traditional GNSS is unavailable for position aiding, and magnetometers can be unreliable in the presence of industrial equipment.

This thesis aims at proposing, developing, and implementing a filtering solution capable of performing indoor autonomous navigation. A Hybrid filter solution is proposed where the GNSS and magnetometer are replaced by a stereo camera for depth perception. The Hybrid-filter is composed of a Kalman filter loosely coupled with a Particle filter. The Kalman filter is the main navigation filter in this framework. The navigation solution is based on integrated inertial measurements and aided by position and heading estimates from the Particle filter. In turn, the particle filter utilizes the velocity and attitude estimates from the Kalman filter, along with the depth data from the stereo camera to estimate the position and heading of the UAV.

A simulation environment is adopted for the project. Further, the Hybrid filter is implemented in *Just-in-Time compiled* Python code and executed on an embedded computer in a hardware-in-the-loop simulation.

The Hybrid-filter developed is capable of navigation in the constructed industrial simulation environments. Several test cases have been performed, and the navigation system is robust in *feature-rich* environments but struggles in *feature-poor* environments. However, improvements have been suggested to aid navigation in *feature-poor* environments.

Page intentionally left blank

# Contents

# Chapter 1

# Introduction

## 1.1 Background & Motivation

Drones are increasingly being used for several outdoor inspection and surveying tasks within the fields of; transportation infrastructure, agri-food applications, electrical transmission and power generation facilities.

However, when it comes to indoor UAV navigation and maneuvering inside factories, warehouses and other industrial sites setup for inspection tasks there are several problems that need to be addressed in relation to navigation. In addition for indoor industrial applications the instrumentation of the UAV becomes a challenge, conventional navigation aids such as GPS, magnetometers and barometers can be unreliable or inaccessible.

Conventional UAVs regulate their position and attitude by continuously monitoring and merging data from an inertial measurement unit, global positioning system, magnetometers and barometer. However in indoor applications GPS, magnetometers and barometers can be assumed to be unreliable. Therefore it is desirable to develop alternative localization methods.

## 1.2 Problem statement

The primary goal of this thesis is to design, implement and test an autonomous navigation system for UAVs performing indoor inspection tasks for industrial environments.

For autonomous indoor navigation, a UAV will need to be equipped with an onboard computer and sensors capable of replacing the traditional GNSS and magnetometer-based navigation solutions. Advancements made in offline mapping give adequate maps for navigation. In addition, digital twins of industrial complexes are becoming popular; therefore, it can be assumed that the operation area is known and mapped. Further, it is desired that the system is modular, with hardware components available off-the-shelf, sized for indoor applications.

- The system should be able to perform all calculations in real time using on-board sensors and computation
- A sensor package is to be selected for the application at hand.
- The proposed system design should be modular.
- A hardware solution is to be prototyped and tested.
- The proposed system is to be simulated in a real world case.

## 1.3 Related work

### Camera based SLAM navigation

There is a large body of research going on in the field of Simultaneous Localization and Mapping(SLAM).

Some solutions use single camera solutions for performing both localization and mapping of an environment simultaneously [8] [39]. The approach works well for slow-moving UAVs' and wheeled or bipedal movers located on the plane. However, for aerial applications, single-camera localization tends to lose track under dynamic movements.

The use of depth color cameras(RGB-D) has been adopted for SLAM and has given good results [20] [11]. Both position and the orientation of the camera frame are tracked with satisfactory accuracy. However, the processing time required for these systems is inhibiting for real-time embedded applications.

The SLAM methods can be utilized for constructing the map of the environment, but for final inspection applications, it is desirable to have a pre-constructed map where inspection paths can be pre-planned.

### ROS localization packages

Some Localization packages already exist as open-source code for use in the Robot Operating system (ROS).

One such package uses the combination of an RGB-D camera and long-baseline sensor for UAV application [30]. The packages rely on visual odometry, and this tends to drift over time. To stabilize this drift, the aforementioned package uses long-baseline sensors to aid the localization solution.

A master thesis written at NTNU compares three different ROS localization packages [29]. Common for the packages is that they all operate on the assumption that the robot operates on a plane or at surface level. This assumption is common for most ROS localization packages, as they are predominantly used for humanoid, differential-drive, or other wheel-driven robots [32].

### Ray tracing, likelihood fields, and 3D likelihood fields

Different measurement models can be used for localization based on point cloud data. A master's thesis written at Chalmers University [10] compares how ray tracing and likelihood field measurement models compare for automotive applications and conclude that both models produce satisfactory results.

3D likelihood fields have been implemented and used for industrial track-based robots operating in a *complex oil and gas industrial environment* with success [28].

# Chapter 2

# Theory

## 2.1 Frames and transforms

### 2.1.1 Orientation representations

There are different ways of representing the same rotational transformation between two coordinate systems. Euler angles are prominently used to visualize the system's orientation due to its intuitive connection to a physical object. The representation has some drawbacks, mainly that the representation is singular and has discontinuities that need addressing. Of the non-singular representations, both DCM representations and quaternion representations are prominently used for orientation representations. Quaternions main advantage is that they are more computationally efficient than the DCM representation; this advantage is becoming less important with the advancements made in computational power available in single board computers and micro-controllers [37].

Table 2.1: Rotation representations, parameters, constraints and ODEs [34]

| | **Euler angles** | **Rotation matrix** | **Quaternion** |
|---|---|---|---|
| Parameters | $3$ | $3 \times 3 = 9$ | $1 + 3 = 4$ |
| Degrees of freedom | $3$ | $3$ | $3$ |
| Constraints | $3 - 3 = 0$ | $9 - 3 = 6$ | $4 - 3 = 1$ |
| ODE | $\dot{\mathbf{\Theta}} = \mathbf{T}(\mathbf{\Theta})\omega$ | $\dot{\mathbf{C}} = \mathbf{C}\ \mathbf{S}(\omega)$ | $\dot{\mathbf{q}} = \frac{1}{2}\mathbf{q} \otimes \omega$ |

It can be seen in table 2.1 that the Euler angle representation has one advantage over the two other non-singular representations, mainly that it is not constrained. That is to say; unlike the DCM and quaternion representation, they can be integrated more freely without the need for normalization, and also for the DCM representation, orthogonality between then vector columns in the matrix must be maintained.

### 2.1.2 Rotation matrices and transformations

The different representations discussed can all be used to create a directional cosine matrix that is used to change the basis of a vector.

#### Rotation between coordinate systems

To transform a vector from one coordinate system to another the following matrix vector operation is used:

$$\mathbf{r}^a = \mathbf{C}_b^a \mathbf{r}^b \tag{2.1}$$

Where, for the vectors ($\mathbf{r}$) the superscript denotes what frame said vector is resolved in, and for the DCM ($\mathbf{C}$) it should be read as: from frame $b$ to frame $a$

#### Translation between coordinate systems

To translate from one coordinate system to another a simple vector sum is used, that is:

$$\mathbf{r}_{ac}^a = \mathbf{r}_{ab}^a + \mathbf{r}_{bc}^a \tag{2.2}$$

Care must be taken to make sure that the vectors are resolved in the same frame, it they are not, then they must first be transformed to the correct frame.

#### Skew symmetric matrix

A skew symmetric matrix is a matrix with the property:

$$\mathbf{A}^T = -\mathbf{A} \tag{2.3}$$

Defining that:

$$\mathbf{S}(\mathbf{a}) = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \tag{2.4}$$

Then the skew symmetric matrix can be used to compute vector cross product as a matrix vector multiplication:

$$\mathbf{a} \times \mathbf{b} = \mathbf{S}(\mathbf{a})\mathbf{b} \tag{2.5}$$

#### Velocities in different coordinate systems

When using vector and matrix operation representing positions and orientations seen from different frames, then care must be taken when doing derivatives.

Starting with equation 2.1:

$$\mathbf{r}^a = \mathbf{C}_b^a \mathbf{r}^b \tag{2.6}$$

Differentiating with respect to time and remembering the ODE for rotation matrices from table 2.1 gives:

$$\dot{\mathbf{r}}^a = \mathbf{C}_b^a \left( \dot{\mathbf{r}}^b + \mathbf{S}(\omega_{ab}^b)\mathbf{r}^b \right) \tag{2.7}$$

Multiplying both sides with the inverse rotation matrix $\mathbf{C}_a^b$ gives:

$$\dot{\mathbf{r}}^b = \dot{\mathbf{r}}^b + \mathbf{S}(\omega_{ab}^b)\mathbf{r}^b \tag{2.8}$$

This should be read as:

$$^a\dot{\mathbf{r}}^b = \ ^b\dot{\mathbf{r}}^b + \mathbf{S}(\omega_{ab}^b)\mathbf{r}^b \tag{2.9}$$

Where the left superscript is read as the coordinate system at which the derivative is taken [38].

### 2.1.3 Euler angle rotation sequence

An Euler angle rotation sequence is a method of rotating from one coordinate frame to another where the rotation is parameterized with three parameters, the so-called Euler angles [38].

A proper Euler rotation sequence is one where only two compound operations are used. The proper Euler rotations sequences are, therefore:

z-x-z, x-y-x, y-z-y, z-y-z, x-z-x, y-x-y

A modified set of operations were introduced by *Peter Guthrie Tait* and *George H. Bryan* that lends themselves more to aeronautics, the so-called Tait-Bryan sequences:

x-y-z, y-z-x, z-x-y, x-z-y, z-y-x, y-x-z

Defining a set of rotation operations based on the airframe's principle rotation axes is convenient, and the so-called "Yaw, pitch, and roll" sequence is detailed below.

Defining the rotation matrices as:

$$\mathbf{C}_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}, \ \mathbf{C}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \ \mathbf{C}_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.10}$$

With the properties:

$$\mathbf{C}_i(agr)^{-1} = \mathbf{C}_i(agr)^T = \mathbf{C}_i(-agr) \tag{2.11}$$

And the rotation matrix from the Body to Ned frame as:

$$\mathbf{C}_b^n = \mathbf{C}_z(\psi)\mathbf{C}_y(\theta)\mathbf{C}_x(\phi) \tag{2.12}$$

This rotation sequence is combined in the following DCM:

$$\mathbf{C}_b^n = \mathbf{C}(\mathbf{\Theta_{nb}}) \tag{2.13}$$

Where the vector $\mathbf{\Theta}_{nb}$ is a vector containing the three compound angles:

$$\mathbf{\Theta}_{nb} = \begin{bmatrix}\phi & \theta & \psi\end{bmatrix}^T \tag{2.14}$$

Then it follows that:

$$\mathbf{C}_n^b = (\mathbf{C}_b^n)^{-1} = \mathbf{C}_x(-\phi)\mathbf{C}_y(-\theta)\mathbf{C}_z(-\psi) \tag{2.15}$$

Figure 2.1 visualize the *Tait-Bryan* rotation sequence *x-y-z*

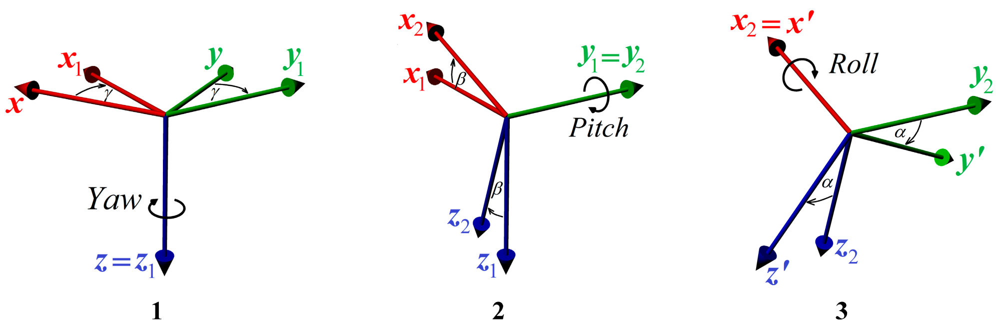



Figure 2.1: Tait-Bryan rotation sequence *x-y-z*, from [17]

#### Angular velocity transformation

To use the body centred angular rates to integrate the Euler angles, some care must be taken. The body rates must be transformed to the right reference frames.

$$\omega_{nb}^b = \begin{bmatrix}\dot{\phi}\\0\\0\end{bmatrix} + \mathbf{C}_x^T(\phi)\begin{bmatrix}0\\\dot{\theta}\\0\end{bmatrix} + \mathbf{C}_x^T(\phi)\mathbf{C}_y^T(\theta)\begin{bmatrix}0\\0\\\dot{\psi}\end{bmatrix} \tag{2.16}$$

Factorising the vectors containing the Euler angles gives:

$$\omega_{nb}^b = \mathbf{T}^{-1}(\mathbf{\Theta}_{nb})\dot{\mathbf{\Theta}}_{nb} \tag{2.17}$$

Solving for $\dot{\mathbf{\Theta}}_{nb}$ gives:

$$\dot{\mathbf{\Theta}}_{nb} = \mathbf{T}(\mathbf{\Theta}_{nb})\omega_{nb}^b \tag{2.18}$$

Where:

$$\mathbf{T}^{-1}(\mathbf{\Theta}_{nb}) = \begin{bmatrix}1 & 0 & -sin\theta\\0 & cos\phi & cos\theta sin\phi\\0 & -sin\phi & cos\theta cos\phi\end{bmatrix}, \quad \mathbf{T}(\mathbf{\Theta}_{nb}) = \begin{bmatrix}1 & sin\phi tan\theta & cos\phi tan\theta\\0 & cos\phi & -sin\phi\\0 & \frac{sin\phi}{cos\theta} & \frac{cos\phi}{cos\theta}\end{bmatrix} \tag{2.19}$$

In equation 2.19 the Euler angles singularity occurs when the angle $\theta$ approaches ±90 degrees[1], the fractions will then approach infinity.

[1]Often referred to as gimbal-lock for historical reasons

## 2.2 Statistics

### 2.2.1 Probability distributions

#### Normal Distribution

A Normal distribution, also called Gaussian distribution, is a continuous distribution for a real-valued random number defined by two parameters, the mean $\mu$ and the standard deviation $\sigma$. The general formulation for a Gaussian probability density function is:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) \tag{2.20}$$

A variable following a Gaussian distribution is described by:

$$a \sim \mathcal{N}(b, c) \tag{2.21}$$

This notation means that the variable $a$ is drawn from a Gaussian with mean $b$ and variance deviation $c$.

A Gaussian distribution is symmetric about the mean.

#### Zero mean white Gaussian

Zero mean white Gaussian noise is random numbers drawn from a normal distribution with $\mu = 0$. A fundamental property of white Gaussian noise is the statistical independence of values no matter how close they are to each other in time.

Given two vector valued zero mean white Gaussian's:

$$\mathbf{a}(t) = \mathcal{N}(0, \mathbf{A}) \tag{2.22}$$

$$\mathbf{b}(t) = \mathcal{N}(0, \mathbf{B}) \tag{2.23}$$

Assuming they are uncorrelated, they will then have the following expected values:

$$E[\mathbf{a}_k \mathbf{a}_j^T] = \begin{cases} \mathbf{A}_k & , k = j \\ 0 & , k \neq j \end{cases} \tag{2.24}$$

$$E[\mathbf{b}_k \mathbf{b}_j^T] = \begin{cases} \mathbf{B}_k & , k = j \\ 0 & , k \neq j \end{cases} \tag{2.25}$$

$$E[\mathbf{a}_k \mathbf{b}_j^T] = 0 \tag{2.26}$$

#### Arbitrary distributions

Probability density functions can take any arbitrary shape depending on the underlying process. A multimodal distribution, like the one shown in figure 2.2, is a good example of how measures like the mean and standard deviation can be deceptive. The mean value calculated from the distribution is shown as the vertical black dashed line, displays that the mean is not a typical value from the distribution.

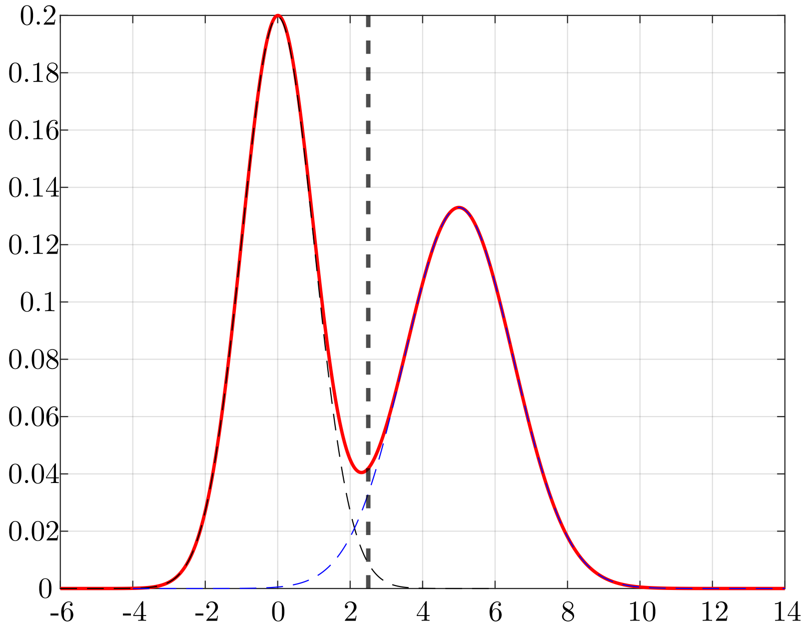



Figure 2.2: A bimodal distribution composed of two gaussian distributions

#### Histogram

A histogram is an approximate representation of the underlying distribution in a data set, splitting the data into "bins" with the area of each column denoting the weight of that bin. A histogram is often shown with each sample weighting 1, resulting in the height of the column being equal to the frequency of observations contained in the bin.

For representing a probability distribution, it is often preferred to have a normalized histogram, meaning that the total area of the columns equals one, giving an approximation of the underlying probability density function. Figure 2.3 illustrates two histograms created from data drawn from Gaussian distributions.

### 2.2.2 Variance

Variance is the expected square deviation from the mean value of a dataset. Calculating the variance from a set of normalized weighted samples is done with the following equation [19]:

$$\sigma^2 = \frac{V_1}{V_1^2 - V_2} \sum_{i=0}^{n} (x_i - x_\mu)^2 * w_i \tag{2.27}$$

Where $x_\mu$ is the mean of the data set, $V_1$ and $V_2$ is the sum of weights and sum of square weights, respectively.

$$V_1 = \sum_{i=0}^{n} w_i \qquad\qquad V_2 = \sum_{i=0}^{n} w_i^2$$

When all samples are given the weight 1, this simplifies to the more common

$$\sigma^2 = \frac{1}{n-1} \sum_{i=0}^{n} (x_i - x_\mu)^2$$

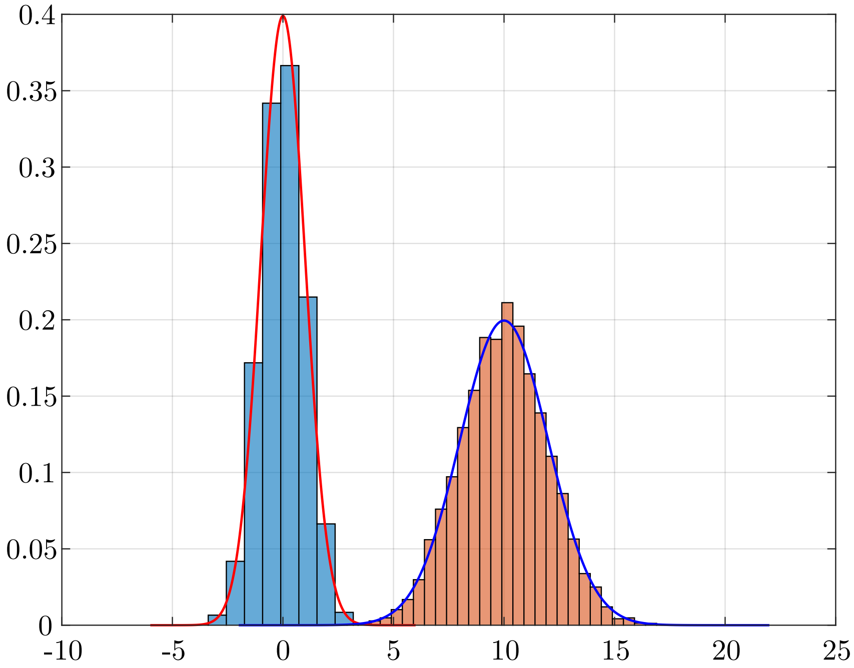



Figure 2.3: Data from two gaussian distributions, collected into normalized histograms with different bin sizes

Mean Square Error (MSE) is calculated in the same manner, except that the value $x_\mu$ can be selected to be any arbitrary value.

### 2.2.3 Propagation of uncertainty

Propagation of uncertainty is needed when both the result of an equation and the uncertainty of that result is of concern. Then the underlying uncertainty of the variables making up the equations has to be propagated trough the same function[21].

Given a multi-variable differentiable function:

$$c = f(a, b) \tag{2.28}$$

Then the variance of the variable $c$ is calculated using the variance equation:

$$\sigma_c^2 = \left|\frac{\partial f}{\partial a}\right|^2 \sigma_a^2 + \left|\frac{\partial f}{\partial b}\right|^2 \sigma_b^2 \tag{2.29}$$

This equation assumes that $a$ and $b$ are independent variables[2].

For a multi variable vector function the variance equation takes a slightly different form.

Given the vector function:

$$\mathbf{c} = \mathbf{f}(\mathbf{a}, \mathbf{b}) \tag{2.30}$$

[2]if this assumption does not hold then an additional term must be added: $\sigma_c^2 = \left|\frac{\partial f}{\partial a}\right|^2 \sigma_a^2 + \left|\frac{\partial f}{\partial b}\right|^2 \sigma_b^2 + \left|\frac{\partial f}{\partial a}\frac{\partial f}{\partial b}\right| \sigma_a \sigma_b$

Under the condition that the uncertainties in the now vector variable **a** and **b** are uncorrelated, then it follows that:

$$\mathbf{\Sigma}_c = \mathbf{W}_a \mathbf{\Sigma}_a \mathbf{W}_a^T + \mathbf{W}_b \mathbf{\Sigma}_b \mathbf{W}_b^T \tag{2.31}$$

Where $\mathbf{\Sigma}_i$ is a matrix with the variances of the vector variable **i** on its diagonal[3]:

$$\mathbf{\Sigma}_i = \begin{bmatrix} \sigma_{i,1}^2 & & \\ & \ddots & \\ & & \sigma_{i,n}^2 \end{bmatrix} \tag{2.32}$$

The matrix $\mathbf{W}_i$ is the partial derivative of the vector function with respect to variable **i**

$$\mathbf{W}_i = \frac{\partial \mathbf{f}}{\partial \mathbf{i}} \tag{2.33}$$

### 2.2.4 Importance sampling

Importance sampling is a method to estimate properties of a particular distribution (*target*) by drawing samples from another distribution (*proposal*) [3], and is often used in statistical analysis when one particular distribution is either *unknown* or unpractical to sample from.

The procedure is to draw samples from the proposal distribution, and weighting the samples with an *importance weight*. Assuming the ability to evaluate the target distribution at $x$, the target can be estimated as weighted samples from the proposal distribution following:

$$p(x) \approx \frac{p(x^i)}{q(x^i)} q(x^i) \tag{2.34}$$

Where the fraction $\frac{p(x)}{q(x)}$ is the importance weight, as the amount of samples goes toward infinity this approximation will go towards the true target distribution.

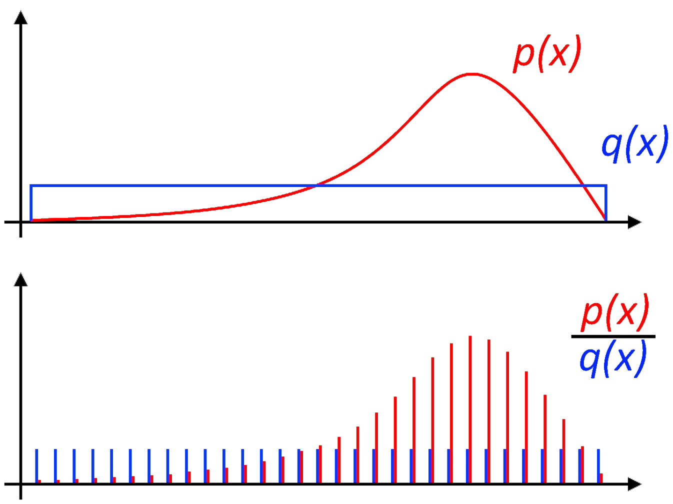



Figure 2.4: Importance sampling of a distribution $p(x)$ over $q(x)$

[3]If the variances in the vector variable is correlated then the matrix $\mathbf{\Sigma}$ becomes the covariance matrix for the vector variable

### 2.2.5 Combination of "independent observations"

In probability theory the combination of independent observations, meaning observations where one outcome does not influence the next[4], is done multiplicatively. For a set of $n$ independent observations of probability $p$, the total probability $p_{tot}$ is calculated using the following equation:

$$p_{tot} = \prod_{i=1}^{n} p^i \tag{2.35}$$

### 2.2.6 Markov property

In statistics and probability theory, processes that are only dependent on the current state, or so-called "memoryless" processes are said to possess the Markov Property [1]. This means that the previous states have no effect on future states; or in other words, given the present, the future does not depend on the past.

#### Hidden Markov Model

A Hidden Markov Model (*HMM*), is a statistical model where the system is assumed to be a Markov Process. The system states $x(t)$ are unobservable, whereas another process $y(t)$, dependant on $x(t)$ is observable. The goal is to get an estimate of $x(t)$ by observing $y(t)$.

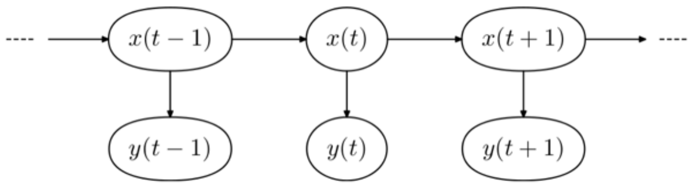



Figure 2.5: A Hidden Markov Model $x$, with observations $y$

[4]An example of independent observations can be consecutive rolls of a dice

## 2.3 State space modelling

State-space modeling is a modeling methodology used to represent a time-varying system by a set of states that vary in time. The next system state is dependent on the previous states and the current input to the system. The output of the system is a combination of the state's current states.

The modeling approach under certain conditions [5] lends itself nicely to algebraic manipulation, analysis, and matrix-vector representation.

State-space modeling can be done either in continuous time or transformed to a discrete difference equation.

### 2.3.1 Linearization

For many systems a linear set of equation can not be directly obtained, if this is the case the system of differential equations must be linearized to fit into the state space regime.

Given an nonlinear set of differential equations:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \tag{2.36}$$

Then a linear set of equations can be obtained by taking the partial derivatives of $\mathbf{f}(\cdot)$ with respect to both the state vector $\mathbf{x}$ and control input $\mathbf{u}$:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \tag{2.37}$$

Where:

$$\mathbf{A} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}, \ \mathbf{B} = \frac{\partial \mathbf{f}}{\partial \mathbf{u}} \tag{2.38}$$

### 2.3.2 Continuous time model

A continuous state-space system is represented in the following form.

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) + \mathbf{w}(t) \tag{2.39}$$

$$\mathbf{y} = \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t) + \mathbf{v}(t) \tag{2.40}$$

Here the matrix $\mathbf{A}$ describes the evolution of the states based on the previous states, the matrix $\mathbf{B}$ describes how the states evolve based on the external input $\mathbf{u}$. The variable $\mathbf{y}$ represents the system output and is composed of the system states through the output matrix $\mathbf{C}$ and the feed-through matrix $\mathbf{D}$ which describes the effect of the input on the output of the system.

The process noise $\mathbf{w}(t)$ and measurement noise $\mathbf{v}(t)$ are both uncorrelated zero mean white Gaussian noise as described in section 2.2.1

Where the algebraic variables used are defined in table 2.2.

[5] The system must be time-invariant and finite-dimensional

Table 2.2: State space variables

| Symbol | Description | Dim of element |
|---|---|---|
| $\mathbf{x}$ | state vector | $n\ x\ 1$ |
| $\mathbf{u}$ | input vector | $p\ x\ 1$ |
| $\mathbf{y}$ | output vector | $q\ x\ 1$ |
| $\mathbf{A}$ | state transition matrix | $n\ x\ n$ |
| $\mathbf{B}$ | input matrix | $n\ x\ p$ |
| $\mathbf{C}$ | output matrix | $q\ x\ n$ |
| $\mathbf{D}$ | feed-through matrix | $q\ x\ p$ |
| $\mathbf{Q}$ | process noise covariance | $n\ x\ n$ |
| $\mathbf{R}$ | observation noise covariance | $q\ x\ q$ |

### 2.3.3 Discrete time model

For many purposes, a continuous time state space representation is not optimal. This is often the case when the system is to be implemented in code. The state space system then takes a slightly different form using forward Euler integration and zero order hold for the control variable $\mathbf{u}_k$[6][5]:

$$\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1} + \mathbf{B_d}\mathbf{u}_k + \mathbf{w}_k \tag{2.41}$$

$$\mathbf{y}_k = \mathbf{H}\mathbf{x}_k + \mathbf{D_d}\mathbf{u}_k + \mathbf{v}_k \tag{2.42}$$

Where again the quantities $\mathbf{w}_k$ and $\mathbf{v}_k$ are both zero mean white Gaussian noise.

### 2.3.4 Exact discretization

To arrive at an exact discretization of the continuous-time system, the following transformation can be applied:

$$\mathbf{F} = e^{\mathbf{A}\cdot dt} \tag{2.43}$$

$$\mathbf{B_d} = \mathbf{A}^{-1}(\mathbf{F} - \mathbf{I})\mathbf{B} \tag{2.44}$$

$$\mathbf{H} = \mathbf{C} \tag{2.45}$$

$$\mathbf{D_d} = \mathbf{D} \tag{2.46}$$

$$\mathbf{Q_d} = \int_{\tau=0}^{T} e^{\mathbf{A}\tau}\mathbf{Q}e^{\mathbf{A}^T\tau}d\tau \tag{2.47}$$

$$\mathbf{R_d} = \mathbf{R}\cdot\frac{1}{dt} \tag{2.48}$$

[6] Alternative methods will be discussed in later subsections

### 2.3.5 Approximate discretizations

There are several ways to arrive at an approximate discretization of a continuous-time system. Typical for most of the methods is that they treat all but the state transition matrix $\mathbf{A}$ the same.

There are several reasons for using an approximate solution, the most prominent of which is to reduce the computational expense of calculating the matrix exponential function and the involved integral required for an exact discretization.

**Forward Euler:**

Probably the most used approximation is the forward Euler method, here the two first terms of the matrix exponential is used:

$$\mathbf{F} = \mathbf{I} + \mathbf{A}dt \approx e^{\mathbf{A}dt} \tag{2.49}$$

**Backwards Euler:**

Another used approximation is the backwards Euler method:

$$\mathbf{F} = (\mathbf{I} - \mathbf{A}dt)^{-1} \approx e^{\mathbf{A}dt} \tag{2.50}$$

**Tustin transformation**

*Tustin transformation* is a discretization method that retains the stability properties of the original state transition matrix.

$$\mathbf{F} = \left(\mathbf{I} + \mathbf{A} \cdot \frac{dt}{2}\right) \left(\mathbf{I} - \mathbf{A} \cdot \frac{dt}{2}\right)^{-1} \approx e^{\mathbf{A}dt} \tag{2.51}$$

**Remaining variables**

The output matrix $\mathbf{H}$, feed-forward matrix $\mathbf{D_d}$ and observation noise covariance $\mathbf{R}$ remains the same as for the exact discretization.

$$\mathbf{B_d} = \mathbf{B}dt \tag{2.52}$$

$$\mathbf{Q_d} = \mathbf{FQF}^T \tag{2.53}$$

### 2.3.6 Alternative integration method

**Multi-step method**

In a multi step method one uses the information at previous time steps to gain a better solution of the differential equation. This is opposed to the forward Euler method where only the previous solution is used to move the solution forwards in time. Other methods exists for moving a differential equation forwards in time that are more accurate then the forward Euler method but without using information at previous time steps, like the Runge-Kutta method.

#### Two-step Adams-Bashforth

Adams-Bashforth methods uses the solution of one previous step to improve the estimate:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{dt \cdot (3 \cdot \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k) - \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_{k-1}))}{2} \tag{2.54}$$

Linearizing the above equation gives:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + (\mathbf{F} - \mathbf{I}) \left( \frac{3 \cdot \mathbf{x}_k - \mathbf{x}_{k-1}}{2} \right) + \mathbf{B}_d \left( \frac{3 \cdot \mathbf{u}_k - \mathbf{u}_{k-1}}{2} \right) \tag{2.55}$$

## 2.4 Maps

A map $\mathcal{M}$ is a representation of an environment, and is composed of a list of objects and their properties.

$$\mathcal{M} = \{m_1, m_2, ..., m_N\}$$

Where each index $m_n$ specifies a property in the map, and $N$ is the total number of objects in the environment.

The two most common representations for robotics are *feature-based* and *location-based* maps [36], where the indexes have different meanings. In feature-based maps, each feature gets its respective index, with each $m_n$ containing the properties and Cartesian location of a feature in the map. In location-based maps, each index corresponds to one specific location in the map. For 2D planar maps, it is common to index each map element $m_{x,y}$ to emphasize that the property is specific to a given coordinate $(x, y)$.

Where feature-based maps contain only information about each feature, location-based maps contain information about all locations in the environment.

#### Grid maps

Grid maps are location-based of the environment discretized into cells of equal size and can be both 2D or 3D. An inherent weakness of grid-based maps is their tendency to use large amounts of memory when mapping larger environments. An example of a grid map is the binary occupancy grid map.

A binary occupancy grid map is a 2D location-based map where each location in the map is given a binary value to denote whether a cell $(x, y)$ is occupied or not. A good source on the Occupancy Grid mapping algorithm is [36].

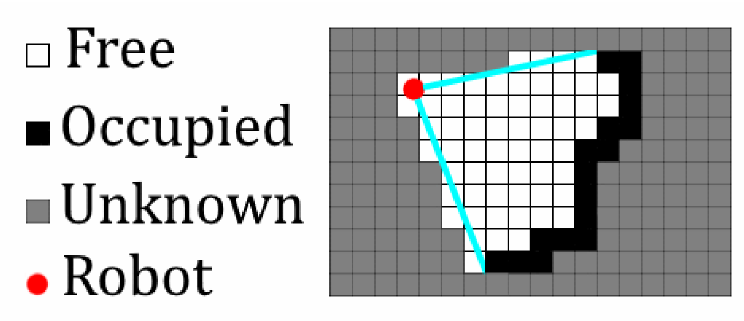



Figure 2.6: A depiction of a binary occupancy map being mapped by a robot

## 2.5 Sensors

### 2.5.1 Inertial measurement unit

An *Inertial Measurement Unit* (IMU) is a device that measures and reports specific force measured by an accelerometer, angular rate measured by a gyroscope, and sometimes earth's magnetic field measured by a magnetometer[7].

#### Accelerometer

An accelerometer measures the proper acceleration of a body, that is, the acceleration of the body relative to its own instantaneous rest frame. Proper acceleration differs from coordinate acceleration in that the coordinate acceleration is relative to a fixed frame [33][34].

In addition, the accelerometer measurement is typically laded with measurement bias and zero-mean white Gaussian noise.

$$\mathbf{a}_m = \mathbf{C}_n^b(\mathbf{a}_{nb}^n + \mathbf{g}^n) + \mathbf{a}_\beta + \mathbf{a}_n \tag{2.56}$$

Here $\mathbf{a}_m$ is the measured acceleration reported by the accelerometer, $\mathbf{a}_{nb}^n$ is the accelerometers coordinate acceleration, $\mathbf{g}^n$ is earths gravitational acceleration[8]. The variable $\mathbf{a}_\beta$ is the accelerometer's bias and has its own dynamical properties. $\mathbf{a}_n$ is the accelerometer's noise, often characterized as zero-mean white Gaussian noise.

By inspection of equation 2.56 it can be seen that an accelerometer in free-fall in earths gravitational field will only measure sensor bias and sensor noise. This is because the coordinate acceleration $\mathbf{a}_{nb}^n$ will then be exactly equal in magnitude to earth's gravitational acceleration $\mathbf{g}^n$.

$$\mathbf{a}_m = \mathbf{C}_n^b(\mathbf{a}_{nb}^n - \mathbf{g}^n) + \mathbf{a}_\beta + \mathbf{a}_n,\ \mathbf{a}_{nb}^n - \mathbf{g}^n = \mathbf{0}\ \rightarrow\ \mathbf{a}_m = \mathbf{a}_\beta + \mathbf{a}_n \tag{2.57}$$

Following the same reasoning it can also be seen that an accelerometer at rest, ie. $\mathbf{a}_{nb}^n = \mathbf{0}$ will measure the same sensor bias and sensor noise, but also measure and report a measure of earth gravitational acceleration.

$$\mathbf{a}_m = \mathbf{C}_n^b(\mathbf{a}_{nb}^n - \mathbf{g}^n) + \mathbf{a}_\beta + \mathbf{a}_n,\ \mathbf{a}_{nb}^n = \mathbf{0}\ \rightarrow\ \mathbf{a}_m = -\mathbf{C}_n^b\mathbf{g}^n + \mathbf{a}_\beta + \mathbf{a}_n \tag{2.58}$$

The accelerometer sensor bias dynamic can be described by a random walk process defined by a zero-mean white Gaussian noise process[41]:

$$\dot{\mathbf{a}}_\beta = \mathbf{a}_{n,\beta} \tag{2.59}$$

$$\mathbf{a}_{n,\beta} = \mathcal{N}(0, \sigma_{a,\beta}) \tag{2.60}$$

#### Gyroscope

A gyroscope is a device to measure and report a body's angular velocities. The measurement of the angular rates is measured in the body frame. The measure is laded with a bias and

[7]A magnetometer is not an inertial sensor but is included due to the commonness of such a sensor in IMU IC packages.

[8]Earths gravitational acceleration is defined to be pointing straight up from a level surface

zero mean white Gaussian noise[9].

$$\omega_m = \omega_{nb}^b + \omega_\beta + \omega_n \tag{2.61}$$

Where $\omega_m$ is the rate reported by the gyroscope, $\omega_{nb}^b$ is the true angular rate of the body, $\omega_\beta$ is the sensor's bias, and $\omega_n$ is a zero mean white Gaussian noise used to model the sensor noise.

The sensor bias dynamic can be modeled by a white Gaussian random walk process:

$$\dot{\omega}_\beta = \omega_{n,\beta} \tag{2.62}$$

$$\omega_{n,\beta} = \mathcal{N}(0, \sigma_{\omega,\beta}) \tag{2.63}$$

### 2.5.2 Range finders

A *range finder* is a device that measures the distance from the range finder to an object. There is a multitude of different range-finding sensors basing themselves on different measurement principles.

In sensors like single measurement LiDARs and ultrasonic range sensors, the time of flight principle is used. This is where a known signal is sent out and the time of its return to the sensor is used to determine the distance from the sensor to the object measured.

#### Ray tracing sensor model

The ray tracing sensor model is an approximate model of the physical workings of a range finder. $p(z_t^k|\mathbf{x}_t, \mathcal{M})$ models a ray moving from the range finder to a position in the map, and is composed of four different densities, each corresponding to a certain kind of error [36].

(a) The gaussian distribution $p_{hit}$ modelling the actual hit, here denoted $z_t^{k*}$ with measurement noise.

(b) The exponential distribution $p_{close}$ giving the probability of close-measurements, often given by unmodelled disturbances in the environment like people walking in front of the sensor.

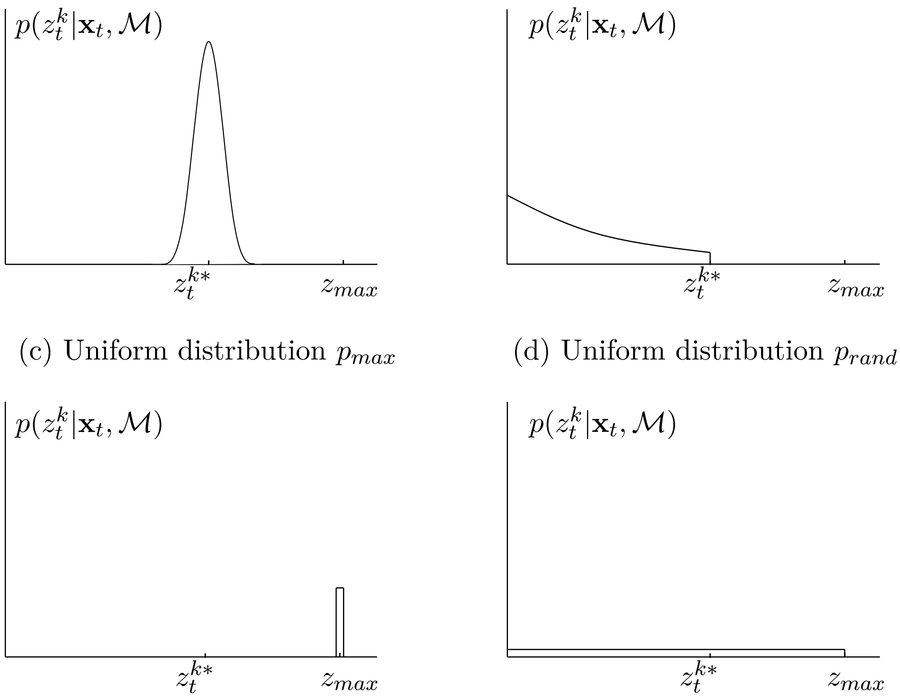
(c) The uniform distribution $p_{max}$ giving a distinct likelihood of max-range measurements.

(d) The uniform distribution $p_{rand}$ adding some probability to all possible readings up to $z_{max}$.

The probability from each distribution is then mixed using four mixing parameters in the following manner:

$$p(z_t^k|\mathbf{x}_t, \mathcal{M}) = \begin{pmatrix} z_{hit} \\ z_{short} \\ z_{max} \\ z_{rand} \end{pmatrix}^T \cdot \begin{pmatrix} p_{hit} \\ p_{short} \\ p_{max} \\ p_{rand} \end{pmatrix} \tag{2.64}$$

[9]Earths rotation is not modeled, but could be added in the following way $\omega_m = \omega_{nb}^b + \omega_\beta + \omega_n + \omega_{ei}^b$, where $\omega_{ei}^b$ is earths rotation relative to the *fixed stars* resolved in the body frame



(a) Gaussian Distribution $p_{hit}$

(b) Exponential distribution $p_{close}$

(c) Uniform distribution $p_{max}$

(d) Uniform distribution $p_{rand}$

Figure 2.7: The ray tracing sensor model is a combination of four distributions

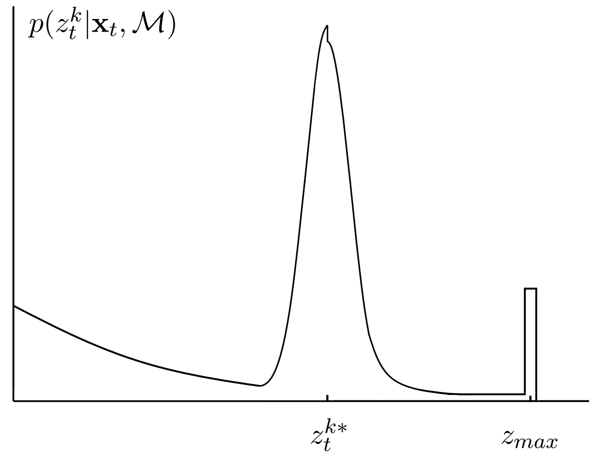


Figure 2.8: The full probability distribution for a single ray in the ray tracing sensor model

Yielding the probability of one ray from the scan, where the sum of the mixing parameters $z_x$ equal 1.

The full scan $\mathbf{z}_t$ will have a probability equal to the product of the probability of each ray, assuming they are considered independent measurements, and can be combined using equation 2.35.

Depending on the environment, the ray cast sensor model might produce probabilities that are not smooth over the state $\mathbf{x}_t$, as small changes in $\mathbf{x}_t$ might yield considerable differences in $p(z_t^k|\mathbf{x}_t, \mathcal{M})$. Meaning that a hypothesis close to the actual state could be given a low weight due to tiny errors in the state.

Consider a robot in 2D-space, with position $(x, y)$ and orientation $(\theta)$. If the robot is looking through an open door, a small variation of the orientation $\theta$ might make the ray hit the door-frame instead of going through the door, creating a large shift in $p_{hit}$.

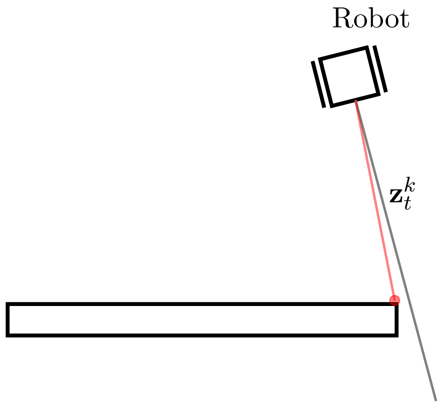



Figure 2.9: A small difference in the robot state can produce very different readings in the ray-cast sensor model.

#### Likelihood field sensor model

The likelihood field sensor model is an end-point model of the sensor. Each reading is projected into the map and the distance from each end-point to the closest object in the map is used to calculate the probability of hit [36].

The end-point model deviates from the physical aspects of the range-finder as projected end-points do not take into account that there might be a wall between the robot and the point. This means that the sensor model can effectively see through walls, rendering close-readings unobservable for the model. Max range readings also need to be removed from the algorithm, as they do not have any meaning in the model. In the physical world, there is no object in the current sensing direction between the robot and the maximum range of the sensor. But placed in the map, max range readings can be regarded as a hit or miss depending on the current robot position and thus produce inaccurate data.

For a 2D-case $(x, y, \theta)$, the points can be projected into the map in the following manner:

$$\begin{pmatrix} x^k_{map} \\ y^k_{map} \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x_{k,sens} \\ y_{k,sens} \end{pmatrix} + z^k_t \begin{pmatrix} \cos(\theta + \theta_{k,sens}) \\ \sin(\theta + \theta_{k,sens}) \end{pmatrix} \tag{2.65}$$

With $(x, y, \theta)$ being the position and orientation of the robot, $\theta_{k,sens}$ being the angle of the ray from the robot sensor axis and $z^k_t$ being the range of the reading. A graphic representation of this projection can be seen in figure 2.10

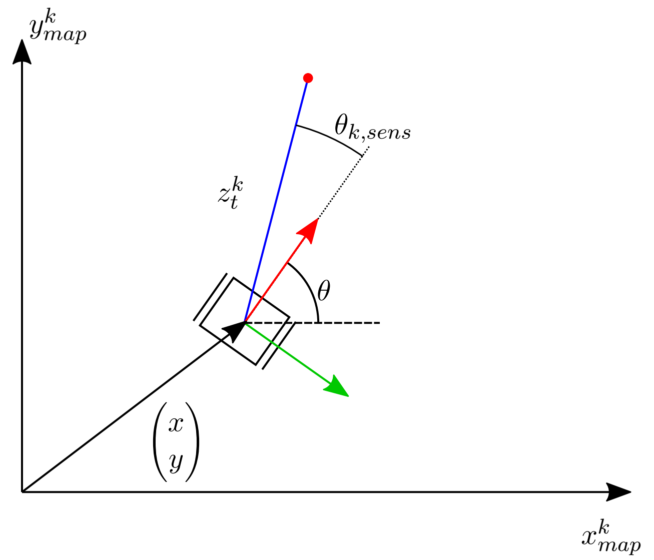



Figure 2.10: A point from a scan projected into map frame

The most time-consuming part of the likelihood field sensor model is finding the distance ($d_{min}$) to the closest object in the map. When this is done the probability of hit can be calculated as a zero-mean Gaussian, with a chosen standard deviation for the map $\sigma_{map}$:

$$p_{hit} = \mathcal{N}_{(0,\sigma_{map})}(d^k_{min}) \tag{2.66}$$

Where $\sigma_{map}$ is a combination of the uncertainty in the map and the sensor. Using the previously mentioned mixing-parameters $z_{hit}$, $z_{rand}$ and $z_{max}$, omitting $z_{close}$, the probability for one endpoint can be found:

$$p(z^k_t|\mathbf{x}_t, \mathcal{M}) = z_{hit} * p_{hit} + \frac{z_{rand}}{z_{max}} \tag{2.67}$$

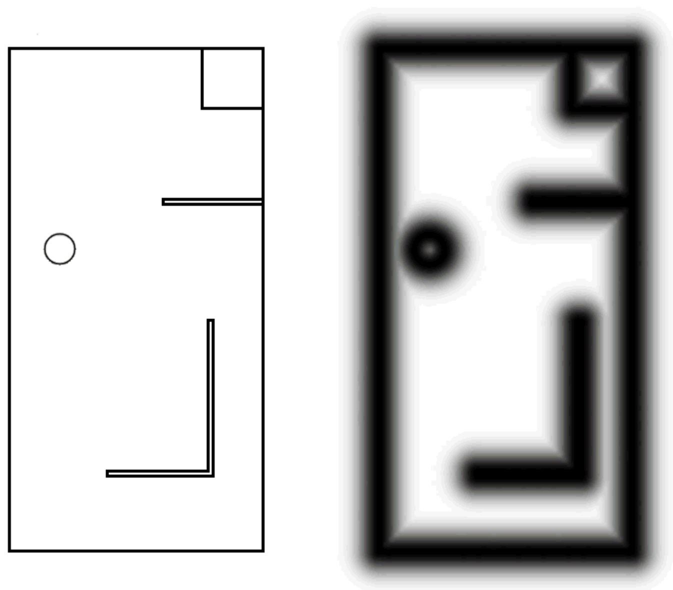


Figure 2.11: A Simple map (left) and it's corresponding likelihood-field (right), with darker colour being more likely locations for hits

The likelihood field is smooth over $\mathbf{x}_t$ compared to the ray-tracing method. Small changes in the state will only move the end-point a tiny bit in the map, yielding small changes in $p_{hit}$.

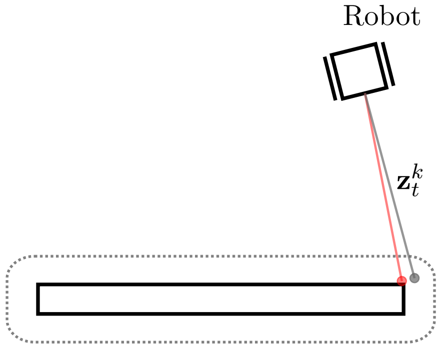



Figure 2.12: A small difference in the robot state produce very similar readings in the likelihood field.

### 2.5.3 Depth Cameras

Depth perception from cameras falls under the field of photogrammetry. At present, depth perception from imagery can be divided into two main categories; active methods and passive methods[27].

#### Active methods

Common for active methods is that they all rely on the emission of some light; herein lies the method's main disadvantage. The light can become insignificant compared to bright sunlight or other bright light sources. Sometimes the environment the camera operates in can be a factor in deciding against the use of active sensors.

#### Structured light methods

In the structured light method, a known pattern is projected onto the camera's field of view. The light can either be visible or in the infrared light spectrum. Depth data is retrieved based on the principle of triangulation, and the computations to retrieve the depth information are relatively trivial and easy to compute, at the cost of a comparatively expensive sensor.

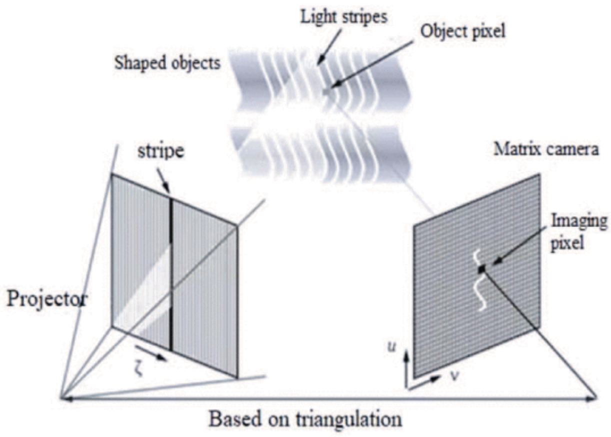



Figure 2.13: Depth Camera structured light method [27]

Figure 2.13 illustrates the principle of the structured light method.

#### Time of flight

In the time of flight method, a light source is pulsed or modulated. Thus, the light is the reflection of the scene. The received light waveform is then measured, and the phase lag in the waveform can be used to infer depth in the scene since the original modulation frequency and the wavelength of the used light is known.

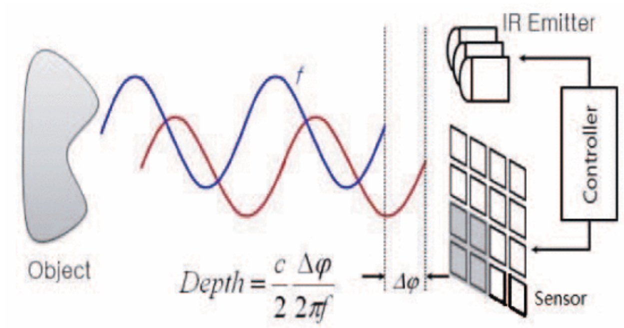



Figure 2.14: Depth Camera time of flight method[27]

Figure 2.14 illustrates how a *Time of flight* depth camera operates.

#### Passive methods

Passive methods differ from the active methods in that they do not relay an active light emission from the camera system. Therefore they can generally be operated in any environment where light is present. However, one disadvantage of passive systems is that they will struggle in featureless environments. This can, in some cases, pose a challenge in retrieving depth information and, in the worst cases, mean that the method will fail outright.

**Stereo vision**

Stereo vision is a method, whereas the name entails two cameras separated by a known distance. Correlated features in the two images can then be used to compute the distance from the camera system to the point in the image. This operation is reasonably computationally cheap. However, the problem of feature extraction and correlation is more problematic and remains computationally expensive.

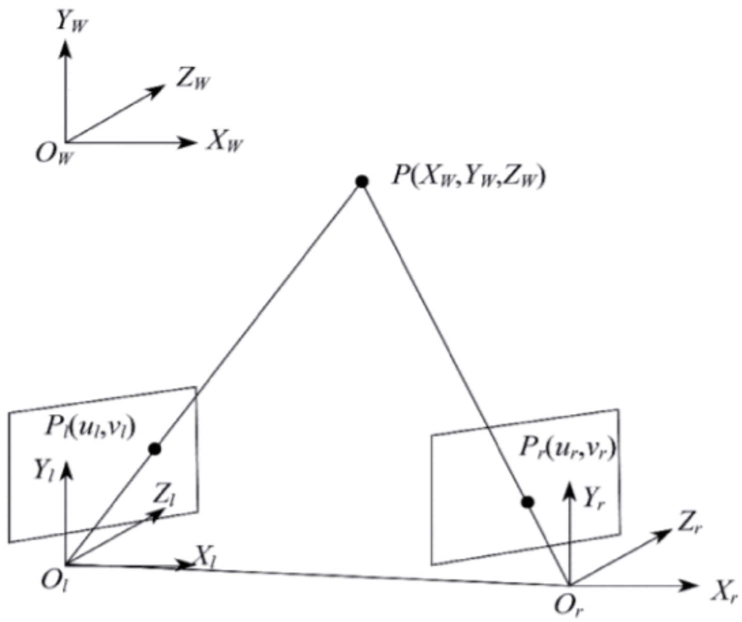



Figure 2.15: Depth camera stereo vision method[27]

Figure 2.15 illustrates how a correlated point in each of the camera frames is used to retrieve the depth information.

## 2.6 Kalman filter

The Kalman filter, sometimes called a linear quadratic estimator, is an algorithm for estimating the state of a system given a series of measurements and observations over time. Unfortunately, the measurements and observations are typically laden with statistical noise and other inaccuracies (like biases). The filter aims to produce an estimate that is better than what a single measurement could provide and make *hidden* system-states observable. This is advantageous for later implementation for control purposes and to determine underlying errors in sensor measurements that are a combination of several system states.

The algorithm can be separated into two main steps, prediction and update/measurement.

The prediction step advances the state of the system based on the previous state and the system input. In this step, the uncertainty in the states is propagated through the state transition matrix, along with the additional uncertainty added by the actuation of the system.

The measurement step incorporates observations made by the sensors and calculates a new optimal estimation based on this new information; the state uncertainty covariance is updated to reflect this new information.

### 2.6.1 Standard filter

Given a linear discrete state-space system, as described in section 2.3.3:

$$\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1} + \mathbf{B_d}\mathbf{u}_k + \mathbf{w}_k \tag{2.68}$$

$$\mathbf{y}_k = \mathbf{H}\mathbf{x}_k + \mathbf{v}_k \tag{2.69}$$

The estimated values of the system-state and state-covariance are computed using the following two equations:

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{F}\hat{\mathbf{x}}_{k-1|k-1} + \mathbf{B_d}\mathbf{u}_k \tag{2.70}$$

$$\mathbf{P}_{k|k-1} = \mathbf{F}_k\mathbf{P}_{k-1|k-1}\mathbf{F}_k^T + \mathbf{Q}_k \tag{2.71}$$

Given a new measurement $\mathbf{z}_k$, the newfound information is used to update the filter state estimates. The measurement innovation is calculated as the difference between the observed and predicted measurement:

$$\tilde{\mathbf{y}}_k = \mathbf{z}_k - \mathbf{H}_k\hat{\mathbf{x}}_{k|k-1} \tag{2.72}$$

With measurement innovation covariance given by:

$$\mathbf{S}_k = \mathbf{H}_k\mathbf{P}_{k|k-1}\mathbf{H}_k^T + \mathbf{R}_k \tag{2.73}$$

The new optimal Klaman gain $\mathbf{K}_k$ is computed based on the measurement covariance $\mathbf{S}_k$:

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}_k\mathbf{S}_k^{-1} \tag{2.74}$$

The Kalman gain $\mathbf{K}_k$ and the measurement innovation $\tilde{\mathbf{y}}_k$ are then used to update the state- and state-covariance estimates.

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k \tilde{\mathbf{y}}_k \tag{2.75}$$

$$\mathbf{P}_{k|k} = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k)\mathbf{P}_{k|k-1} \tag{2.76}$$

The last step in the algorithm after updating the state- and state-covariance is to change indexes on the variables so that:

$$\hat{\mathbf{x}}_{k-1|k-1} = \hat{\mathbf{x}}_{k|k} \tag{2.77}$$

$$\mathbf{P}_{k-1|k-1} = \mathbf{P}_{k|k} \tag{2.78}$$

### 2.6.2 Nonlinear variety

Given a nonlinear discreet state transition function and measurement equation on the form:

$$\mathbf{x}_k = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{w}_k) \tag{2.79}$$

$$\mathbf{y}_k = h(\mathbf{x}_{k-1}, \mathbf{v}_k) \tag{2.80}$$

The nonlinear function now also contains the noise vectors; some slight modifications are needed in the state-covariance update and measurement innovation covariance equations [10].

$$\mathbf{P}_{k|k-1} = \mathbf{F}_k \mathbf{P}_{k-1|k-1} \mathbf{F}_k^T + \mathbf{W}_q \mathbf{Q}_k \mathbf{W}_q^T \tag{2.81}$$

$$\mathbf{S}_k = \mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^T + \mathbf{W}_r \mathbf{R}_k \mathbf{W}_r^T \tag{2.82}$$

The equations needs to be linearized to fit into the Kalman filter framework:

$$\mathbf{F} = \left.\frac{\partial \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{w}_k)}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_{k|k}} \tag{2.83}$$

$$\mathbf{H} = \left.\frac{\partial \mathbf{h}(\mathbf{x}_{k-1}, \mathbf{v}_k)}{\partial \mathbf{x}}\right|_{\mathbf{x}=\mathbf{x}_{k|k}} \tag{2.84}$$

$$\mathbf{W_q} = \left.\frac{\partial \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k, \mathbf{w}_k)}{\partial \mathbf{w}}\right|_{\mathbf{x}=\mathbf{x}_{k|k}} \tag{2.85}$$

$$\mathbf{W_r} = \left.\frac{\partial \mathbf{h}(\mathbf{x}_{k-1}, \mathbf{v}_k)}{\partial \mathbf{v}}\right|_{\mathbf{x}=\mathbf{x}_{k|k}} \tag{2.86}$$

[10]for a system with dynamics $\mathbf{x}_k = f(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_k$ and measurement $\mathbf{y}_k = h(\mathbf{x}_{k-1}) + \mathbf{v}_k$ this step would be unnecessary

## 2.7 Particle Filter

### 2.7.1 General introduction

A Particle Filter is a Bayesian filter and a Monte Carlo algorithm used for estimating the probability density function of the internal dynamic states of a system. The filter sets up a number of discrete guesses as to what the state of the system is at time k $\{\mathbf{x}_k^i, i = 0...n\}$ , and uses measurements to update a likelihood weight $\{w^i\}$ for each particle $\{\mathbf{x}_k^i, w^i\}$. This makes the particle filter able to represent an arbitrary probability distribution of the state but will quickly increase in complexity with larger state spaces.

### 2.7.2 Use case for a particle filter

The particle filter can, in theory, be used for any state estimation task. The algorithm has strengths compared to more classical state estimation methods such as the Kalman filter, most notable for its innate ability to represent any arbitrary probability distribution. The particle filter can also use strongly nonlinear measurement models; however, these strengths do not come without a cost. The particle filter is a very computationally expensive algorithm. Thus, the particle filter has not seen widespread use in real-time state estimation applications.

The number of samples (particles) required to give a good representation of the underlying probability density increases with the number of states included in the filter. Therefore, to be used for a real-time system, the number of states in the filter should be kept low.

#### Nonlinear measurement models

Because the particle filter estimates the state based on many discrete guesses, it is well suited to use sensors that do not directly measure the state. Sensors like LiDARs and range finders give measurements where it is difficult to provide an immediate estimate of a state given the sensor data, but it is easy to estimate the likelihood of the sensor data given a state. This holds if the environment is known; if the robot is maneuvering in an unknown environment, then some form of SLAM (Simultaneous Localization and Mapping) algorithm must be employed. SLAM will not be covered in this report.

For example, for a robot moving in the $xy$ plane a scan from a directional LiDAR might look like this:

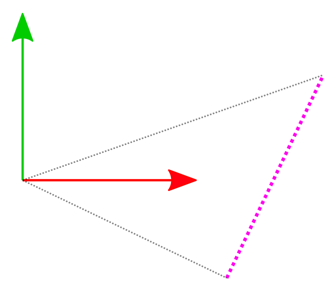


Figure 2.16: Example of what a scan from a LiDAR could look like in the XY-plane

With the arrows representing the robot axes, and the pink dots are points in the pointcloud

Given only this scan, it is hard to determine much about where the robot is located in the room. One could determine that it is a certain distance from a wall, but it provides no direct measure of location. However, if the robots position is known to some degree, one could try to match the scan to a map of the environment. With the particles in the particle filter centered around the estimated position, one would loop through the particles and update their weights based on how well their pose fits the scan in the environment. The particle with the highest weight will then contain the most likely pose.

Placing the scan into the frame of each particle could look something like figure 2.17

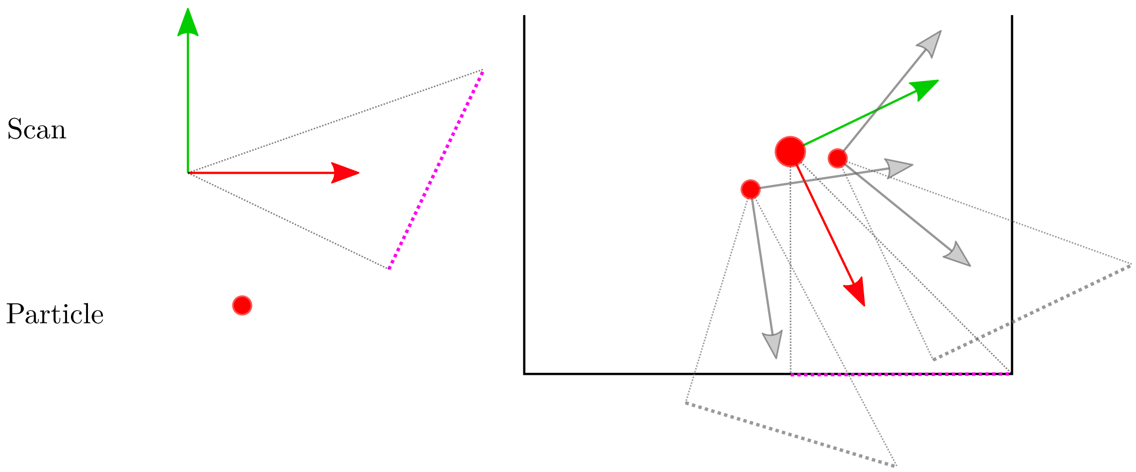



Figure 2.17: The example LiDAR scan placed in the frame of three different particles in a map

Where the size of the particle is proportional to the updated weight after the sensor update, bigger particles have larger weight.

#### Multi modal probability distributions

Since the particle filter builds the state estimate based on weighted samples in the state space, the resulting probability distribution can take on any arbitrary shape. This can be particularly helpful in localization tasks where measurements often don't give an absolute measure of position, but rather something that can support multiple different hypotheses.

Looking at the same example with the robot moving in the plane, the scan could support a plethora of different positions in a rectangular room.

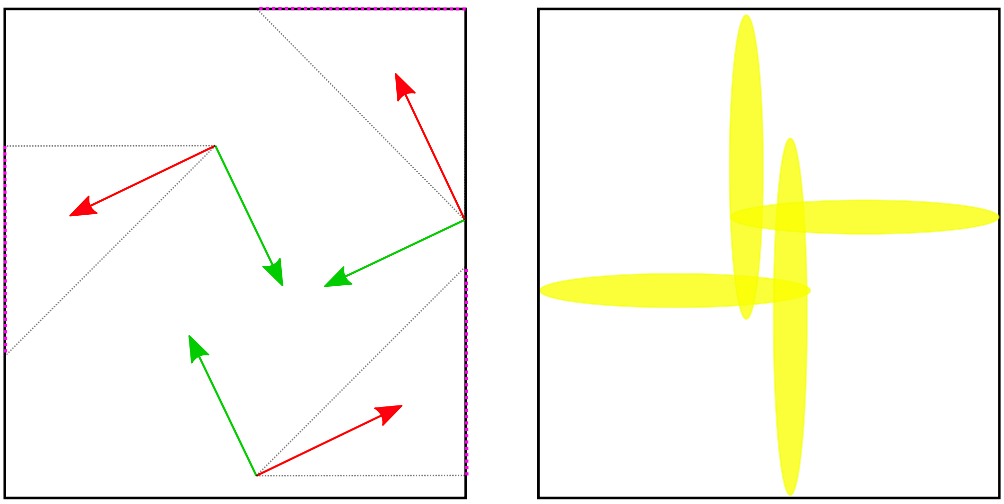


Figure 2.18: LiDAR scan placed in a rectangular room. Left: Multiple poses fit well with scan. Right: The resulting probability distribution

The resulting probability distribution is multi modal, making the filter able to track multiple hypotheses for the true state simultaneously.

### 2.7.3 Sequential Importance Sampling

Sequential Importance Sampling (*SIS*) is one implementation of the particle filter, where the particle weights are updated based on importance sampling. When implementing a SIS particle filter, choosing the correct proposal distribution $q(x)$ is essential to achieve good performance. This section will outline the workings of the SIS filter following [3].

Let $\{\mathbf{x}^i_{0:k}, w^i_k\}^{N_s}_{i=1}$ be a set of particles, with samples $\{\mathbf{x}^i_{0:k}, i = 1, ..., N_s\}$ and associated weights $\{w^i_k, i = 1, ..., N_s\}$. The set of particles describe the posterior probability distribution $p(\mathbf{x}_{0:k}|\mathbf{z}_{1:k})$ of all states from time 0 to k $\{\mathbf{x}_j, j = 0, ..., k\}$ given measurements $\{\mathbf{z}_j, j = 1, ..., k\}$ from time 1 to k. The weights of each particle is normalized so that $\Sigma_i\ w^i_k = 1$, giving an approximation of the true posterior probability distribution at time k given by:

$$p(\mathbf{x}_{0:k}|\mathbf{z}_{1:k}) \approx \sum_{i=1}^{N_s} w^i_k \delta(\mathbf{x}_{0:k} - \mathbf{x}^i_{0:k}). \tag{2.87}$$

Where $\delta(\cdot)$ is the *dirac delta* function, and the weights $w^i_k$ are calculated using importance sampling (sec 2.2.4). Given a set of samples drawn from an importance density $q(\cdot)$, we get a weighted approximation of the target density $p(\cdot)$ given by:

$$p(\mathbf{x}^i) \approx \sum_{i=1}^{N_s} w^i \delta(\mathbf{x} - \mathbf{x}^i). \tag{2.88}$$

Where $w^i$ are the normalized weights associated with each sample $x^i$, and follow:

$$w^i \propto \frac{p(\mathbf{x}^i)}{q(\mathbf{x}^i)} \tag{2.89}$$

#### Finding the weight $w$

With a desire to approximate the posterior density $p(\mathbf{x}_{0:k}|\mathbf{z}_{1:k})$, samples are drawn from a proposal distribution $q(\mathbf{x}_{0:k}|\mathbf{z}_{1:k})$, giving the weights at time k (following equation 2.89):

$$w^i_k \propto \frac{p(\mathbf{x}_{0:k}|\mathbf{z}_{1:k})}{q(\mathbf{x}_{0:k}|\mathbf{z}_{1:k})} \tag{2.90}$$

As the SIS filter is a sequential algorithm, one could at each iteration have a set of particles giving the approximation of $p(\mathbf{x}_{0:k-1}|\mathbf{z}_{1:k-1})$, and want to estimate $p(\mathbf{x}_{0:k}|\mathbf{z}_{1:k})$ using a new set of particles.

If the importance density is chosen to factorize in the following way

$$q(\mathbf{x}_{0:k}|\mathbf{z}_{1:k}) = q(\mathbf{x}_k|\mathbf{x}_{0:k-1}, \mathbf{z}_{1:k})q(\mathbf{x}_{0:k-1}|\mathbf{z}_{1:k-1}) \tag{2.91}$$

Meaning that new set of samples $\mathbf{x}^i_{0:k} \sim q(\mathbf{x}_{0:k}|\mathbf{z}_{1:k})$ can be found by augmenting the current set $\mathbf{x}^i_{0:k-1} \sim q(\mathbf{x}_{0:k-1}|\mathbf{z}_{1:k-1})$ with the new state prediction $\mathbf{x}^i_k \sim q(\mathbf{x}^i_k|\mathbf{x}_{0:k-1}, \mathbf{z}_{1:k-1})$.

And by assuming that $p(\mathbf{x}_{0:k}|\mathbf{z}_{1:k})$ can be broken down into [3]:

$$p(\mathbf{x}_{0:k}|\mathbf{z}_{1:k}) \propto p(\mathbf{z}_k|\mathbf{x}_k)p(\mathbf{x}_k|\mathbf{x}_{k-1})p(\mathbf{x}_{0:k-1}|\mathbf{z}_{1:k-1}) \tag{2.92}$$

We can derive the weight-update equation for the SIS filter by inserting equations 2.91 and 2.92 into 2.90, which gives:

$$w_k^i \propto \frac{p(\mathbf{z}_k|\mathbf{x}_k^i)p(\mathbf{x}_k^i|\mathbf{x}_{k-1}^i)}{q(\mathbf{x}_k^i|\mathbf{x}_{0:k-1}^i,\mathbf{z}_{1:k})}\ \frac{p(\mathbf{x}_{0:k-1}^i|\mathbf{z}_{1:k-1})}{q(\mathbf{x}_{0:k-1}^i|\mathbf{z}_{1:k-1})} \tag{2.93}$$

Where the fraction at the end can be identified as the set of particle weights from last filter iteration,

$$w_{k-1}^i = \frac{p(\mathbf{x}_{0:k-1}^i|\mathbf{z}_{1:k-1})}{q(\mathbf{x}_{0:k-1}^i|\mathbf{z}_{1:k-1})}$$

resulting in the expression:

$$w_k^i \propto w_{k-1}^i \frac{p(\mathbf{z}_k|\mathbf{x}_k^i)p(\mathbf{x}_k^i|\mathbf{x}_{k-1}^i)}{q(\mathbf{x}_k^i|\mathbf{x}_{0:k-1}^i,\mathbf{z}_{1:k})} \tag{2.94}$$

Furthermore, assuming the process has the *Markov Property*, meaning that the posterior state $\mathbf{x}_k$ is only dependant on the last state $\mathbf{x}_{k-1}$ and current measurement $\mathbf{z}_k$, the expression is simplified further. Resulting in the weight-update equation:

$$w_k^i \propto w_{k-1}^i \frac{p(\mathbf{z}_k|\mathbf{x}_k^i)p(\mathbf{x}_k^i|\mathbf{x}_{k-1}^i)}{q(\mathbf{x}_k^i|\mathbf{x}_{k-1}^i,\mathbf{z}_k)} \tag{2.95}$$

In the common case, where only an estimate of the posterior for the current state $\mathbf{x}_k$, and not its trajectory $\mathbf{x}_{0:k}$ is desired, the approximation of the posterior probabilty density of the state becomes:

$$p(\mathbf{x}_k|\mathbf{z}_{1:k}) \approx \sum_{i=1}^{N_s} w_k^i \delta(\mathbf{x_k} - \mathbf{x}_k^i). \tag{2.96}$$

**Weight disparity**

The SIS particle filter is often plagued by the phenomenon of weight disparity, where after a few filter iterations only a few particles retain a significant weight. This makes most of the computation done at each iteration give a negligible contribution to the approximation of $p(\mathbf{x}_k|z_{1:k})$, as most of the particles have close to no weight.

To give a measure of the disparity of the algorithm, the effective sample size $N_{eff}$ is introduced [4], [26]. $N_{eff}$ gives a measure of the number of particles in the filter which are effectively contributing to the approximation of $p(\mathbf{x}_k|z_{1:k})$, and is defined as

$$N_{eff} = \frac{N_s}{1 + \mathrm{Var}(w_k^{*i})} \tag{2.97}$$

Where $w_k^{*i}$ is referred to as the "true weight" and computed as $p(\mathbf{x}_k^i|\mathbf{z}_{1:k})/q(\mathbf{x}_k^i|\mathbf{x}_{k-1}^i,\mathbf{z}_{1:k})$. This is impossible to compute exactly, so an approximation $\hat{N}_{eff}$ of $N_{eff}$ can be found by computing

$$\hat{N}_{eff} = \frac{1}{\sum_{i=1}^{N_s}(w^i)^2} \tag{2.98}$$

over the set of normalized particle weights $w^i$. The effective sample size will always be smaller or equal to the number of particles, and a small value signify severe weight disparity in the filter.

The variance of the filter will increase over time as the particles are propagated around in the state space, so disparity is a problem that will increase with each iteration of the filter. This is obviously an issue as a lot of computation time will be devoted to samples with next to no weight associated with them. To combat the effects of disparity one could:

- Choose an importance density $q(\cdot)$ to minimize $\text{Var}(w_k^{*i})$
- Incorporate a resampling step in the filter algorigthm when the effective sample size becomes too low

#### Choosing the importance density $q(\cdot)$

When designing a SIS filter, choosing the correct importance density $q(\cdot)$ is a crucial step to ensure correct probability propagation in the state space. The importance density could be any arbitrary probability density function, but in order to draw samples in the relevant parts of the distribution the chosen proposal density should resemble the target density to some degree.

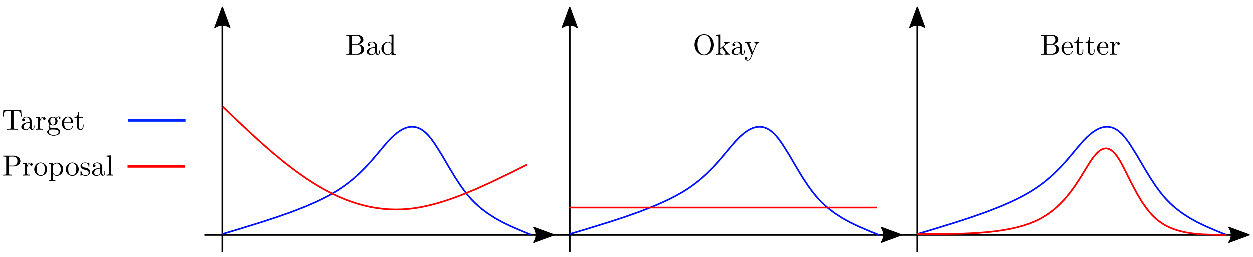



Figure 2.19: Examples of importance densities

#### Optimal importance density

The optimal importance density, which will result in zero $\text{Var}(w_k^{*i})$ conditional on $\mathbf{x}_{k-1}^i$ has been shown to be [9]

$$q(\mathbf{x}_k|\mathbf{x}_{k-1}^i, \mathbf{z}_k)_{opt} = p(\mathbf{x}_k|\mathbf{x}_{k-1}^i, \mathbf{z}_k) \tag{2.99}$$

Which when inserted into equation 2.95 yields the weight update equation

$$w_k^i \propto w_{k-1}^i p(\mathbf{z}_k|\mathbf{x}_{k-1}^i)$$

The optimal importance density may be difficult to use as sampling from $p(\mathbf{x}_k|\mathbf{x}_{k-1}^i, \mathbf{z}_k)$ is often times not straightforward. A common approximation is sampling from the prior $p(\mathbf{x}_k|\mathbf{x}_{k-1}^i)$ instead, which when inserted into equation 2.95 yield

$$w_k^i \propto w_{k-1}^i p(\mathbf{z}_k|\mathbf{x}_k^i) \tag{2.100}$$

Which is an expression that is quick to evaluate, intuitive and straightforward to implement.

### 2.7.4 The SIR particle filter

The Sequential Importance Resampling (*SIR*) particle filter is an implementation of the SIS filter with a resampling step, where the method for resampling is to be determined by the filter designer. Some of the most common approaches are outlined in [24]. The algorithm assumes that it is possible to sample from $p(\mathbf{x}_k|\mathbf{x}_{k-1})$ and that $p(\mathbf{z}_k|\mathbf{x}_k)$ is possible to evaluate (up to proportionality).

The algorithm consists of three main steps:

- Propagation
- Measurement / Re-weight
- Resampling

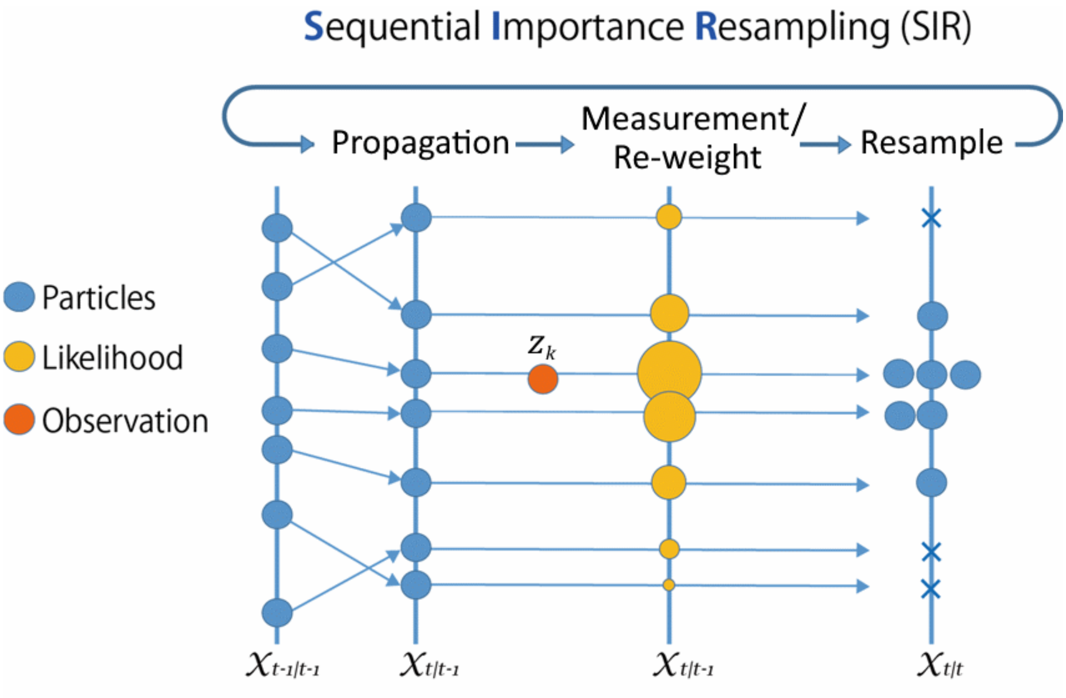



Figure 2.20: Graphic representation of the SIR algorithm, figure from [2]

---

**Algorithm 1:** The SIR particle filter in pseudocode

---

**Input:** $\{\mathbf{x}_{k-1}^n, w_{k-1}^n\}_{n=1}^{N_s}$
**begin**
  **for** $n$ = *1...N*$_s$ **do**
    $\mathbf{x}_k^n \sim q(\mathbf{x}_k^n, \mathbf{x}_{k-1}^n)$ ; `// Draw new samples (Propagate)`
  **end**
  **for** $n$ = *1...N*$_s$ **do**
    $w_k^n \propto w_{k-1}^n p(\mathbf{z}_k|\mathbf{x}_k^n)$ ; `// Update weights (equation 2.100)`
  **end**
  Normalize_Weights() ; `// Normalize sum of weights to 1`
  **if** $\hat{N}_{eff} < N_{thr}$ **then**
    Resample() ; `// Resample if` $N_{eff}$ `below threshold`
  **end**
**end**
**return** $\{\mathbf{x}_k^n, w_k^n\}_{n=1}^{N_s}$

---

**Particle propagation**

The propagation step is what "moves" the particles around in the state space, where the next sample of the particles state is drawn from the distribution $p(\mathbf{x}_k^i|\mathbf{x}_{k-1}^i)$. This distribution could in theory be any arbitrary distribution, but for robotic applications it is often based on robot motion or odometry with some added noise.

**Velocity-based motion model**

A velocity-based motion model proposes a distribution $p(\mathbf{x}_k^i|\mathbf{x}_{k-1}^i, \mathbf{u}_k)$, where the next state in the filter is drawn from a distribution based on the current state $\mathbf{x}_{k-1}^i$ and an input $\mathbf{u}_k$ containing the velocities.

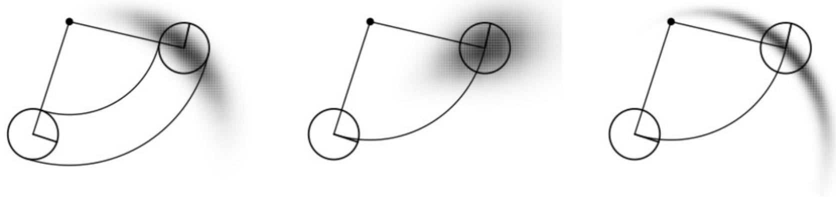


Figure 2.21: The distribution $p(\mathbf{x}_k^i|\mathbf{x}_{k-1}^i, \mathbf{u}_k)$ for different noise parameters [36]

Sampling from the motion model for can be done using the forward Euler method described in subsection 2.3.5 which gives the following expression for the next particle state.

$$\mathbf{x}_k^i = \mathbf{F}_k\mathbf{x}_{k-1}^i + \mathbf{B}_k(\mathbf{u}_k^i + \mathbf{w}_k^i) \tag{2.101}$$

With $\mathbf{F}$, $\mathbf{B}$ and $\mathbf{w}$ as outlined in section 2.3, and $\mathbf{u}_k^i$ is drawn from a Gaussian distribution

$$\mathbf{w}_k^i = \mathcal{N}(\mathbf{0}, \sigma_{\mathbf{u}}) \tag{2.102}$$

**Measurement and re-weight**

The measurement step uses sensor data to give a likelihood for each particle to be the true state of the system. This step is often the most time-consuming in the algorithm, as data from the sensor must be evaluated for all particles in the filter. Depending on the sensor and measurement strategy, this can quickly become several hundred or thousands of calculations per particle.

This step evaluates $p(\mathbf{z}_k|\mathbf{x}_k)$ for each particle, evaluating the likelihood of the scan given the current particle state and updating the weight of the particle according to equation 2.100.

Depending on the measurement model, this will often times not yield a normalized probability distribution over the particles A common approach is to add a normalizing step at the end of the measurement step, where the particle weights are normalized according to:

$$w_k^i = \frac{w_k^i}{\sum_i w_k^i} \tag{2.103}$$

**Resampling of particles**

The resampling step is vital to combat the effects of disparity in the particle filter. There are many different resampling methods, where some draw new samples around the most likely particles, and some replicate the highest and remove the lowest weighted particles. This step is executed when the effective sample size $\hat{N}_{eff}$ from equation 2.98 falls below a set threshold.

A resampling method commonly called "low variance resampling" will be shown in this section. A study of different resampling methods outlining multiple different strategies can be seen in [24].

Low variance resampling is a sequential algorithm and assumes normalized particle weights, meaning that the sum of all particle weights adds up to 1. It loops through all particles and compares the cumulative sum of particle weights $W$ to a number $u$; which consists of a random number $r$ drawn from the uniform distribution $\mathcal{U}(0, N_s^{-1})$ and an additional term increasing with each loop iteration, following the equation:

$$u = r + \frac{n-1}{N_s} \tag{2.104}$$

With $n$ being the current loop iteration $n = 1...N_s$, and $N_s$ is the number of particles in the filter.

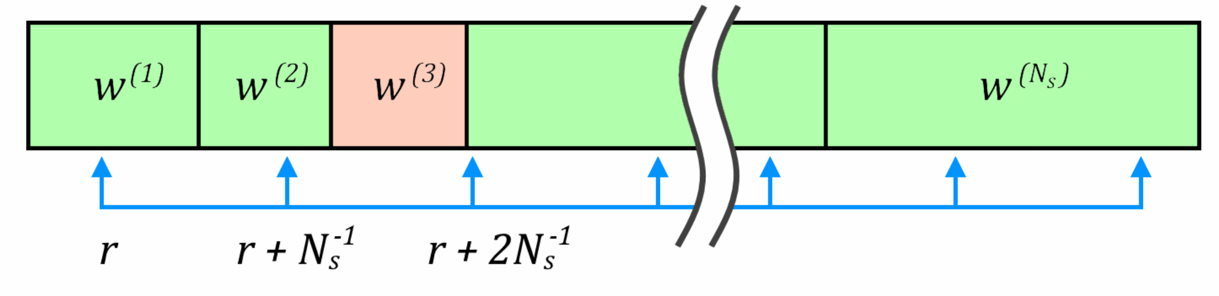



Figure 2.22: Particles picked by the low variance resampler

The algorithm replaces all particles in the filter, breaking up the most likely particles into multiple particles and removing the least likely particles. After resampling, all particles will have the same weight ($1/N_s$), while the sum of weights still equal one and approximately the same weight density is kept throughout the state-space (ref figure 2.20). Pseudocode for the algorithm can be seen in Algorithm 2.

**Algorithm 2:** Low Variance Resampling, named "systematic resampling" in [24]

```
Input: {x_k^(n_p), w_k^(n_p)}_{n_p=1}^{N_s} ;            // Set of particles
begin
    r = U(0, N_s^-1) ;
    W = w^(1) ;
    i = 1;
    for n = 1...N_s do
        u = r + (n - 1)/N_s;
        while u > W do
            i = i + 1;
            W = W + w_k^(i);
        end
        x_k^(n)* = x_k^(i);
        w_k^(n)* = N_s^-1;
    end
end
return {x_k^(n_p)*, w_k^(n_p)*}_{n_p=1}^{N_s} ;   // Resampled set of particles
```

### 2.7.5 Monte Carlo Localization

Monte Carlo Localization (*MCL*), or "particle filter localization" is a localization algorithm for robots using a particle filter. The algorithm models the process as a Hidden Markov Model (2.2.6), with measurements giving some information about the hidden state.

Given sensor inputs and a map of the environment, the algorithm estimates the position and orientation in the map based on recursive Bayesian estimation. The algorithm can be initialized with an initial guess of the robot's location, or "globally" - meaning that there is no information of the robot's start position. Global initialization spreads the particles evenly throughout the map initially, leaving each robot pose equally likely. After the robot moves around and senses the environment, the unlikely poses will be resampled, and the filter should ultimately converge to the true pose of the robot.

#### One dimensional MCL Example

A classic example of Monte Carlo Localization is a "door-sensing" robot moving in one dimension. The sensors on the robot include wheel odometry and a sensor giving readings when the robot is in front of a door.

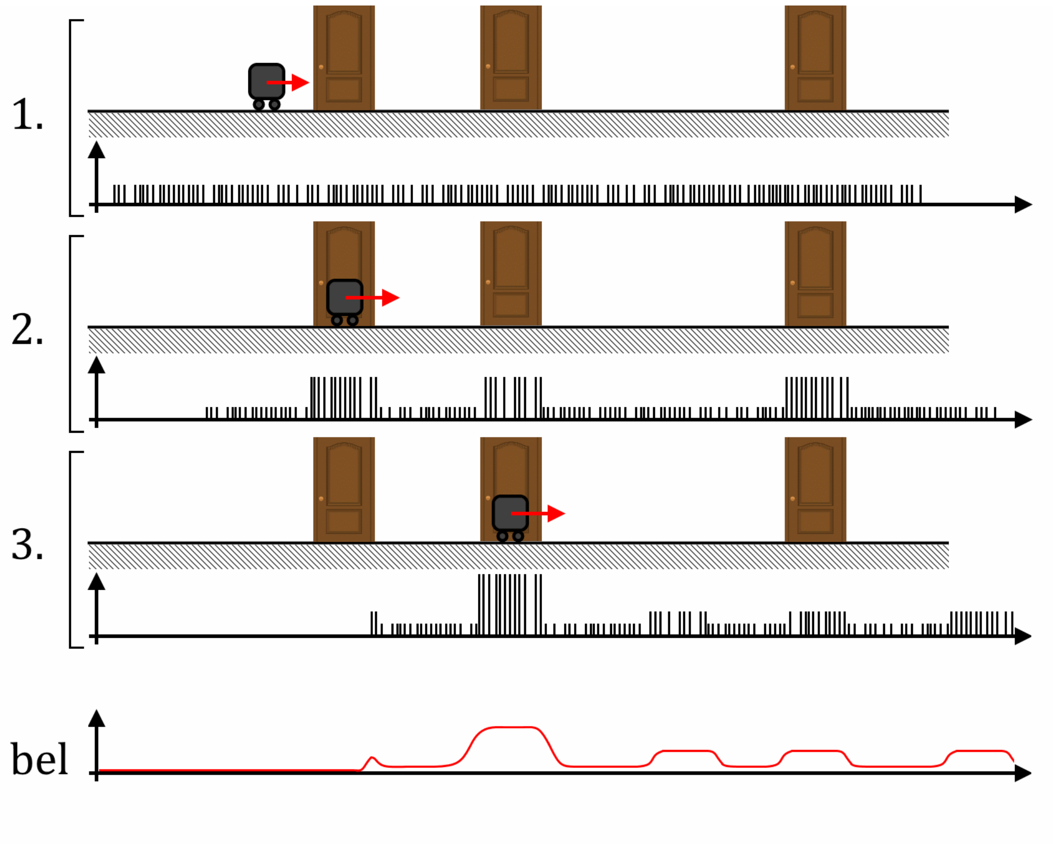



Figure 2.23: Monte Carlo Localization example: A door-sensing robot moving in 1D

1. The algorithm is first globally initialized, leaving all particles with an equal weight, and the robot starts to move to the right.
2. The robot moves to the right, shifting the particles to the right in the state space. At some point, the robot senses a door, giving all particles that are located in front of a door a higher likelihood to be the true pose.
3. The movement continues, and another door is sensed, the algorithm is now fairly certain of which door it is located in front of.

# Chapter 3

# Method

## 3.1 Concepts

In this section the different choices of hardware components will be outlined and motivated. The choice of filtering architecture will also be detailed and motivated.

Further it has been desired to keep the cost of the completed system low, making the system more approachable for further development and possible future deployment of the system.

The system design is also made with modularity in mind, the Hybrid filter consists of a Kalman filter and a Particle filter operating co-dependently. This concept allows for great modularity in design and development. This will be further discussed in this section.

### 3.1.1 Sensor selection

#### Depth perception sensor

The sensor selection for localization is a choice between camera solutions or lidar based depth perception. Due to the expense of 3D lidars they where quickly ruled out as an option for the project.

The selection was then focused on camera based solutions. On the more affordable and compact end of the spectrum is the Intel Realsense series of camera solutions and the Zed Mini stereo camera. The Intel realsense family are primarily active cameras using structured light for depth perception, the emission of light from the camera is undesired as this removes some of the flexibility of the system. Say if a plant operator objects to the emanation of structured light, or the site uses monitoring sensors that are light sensitive. Another drawback of the Intel series is that they are not natively compatible with ARM processor architectures, greatly limiting the choice of small form factor computers for use in the drone. The Zed Mini stereo camera on the other hand is a passive camera relying on stereo vision for depth perception. The Zed Mini software development kit [42] comes with ARM support and a already pre-made ROS implementation. Making is a suitable camera solution for the project.

#### Inertial sensor

The inertial measurement unit used in the project a part of the Pixhawk 4 flight controller used for controlling the drone. The IMU sensor data is available from the flight controller with the use of an alternate flight controller firmware and a ROS software development kit provided by the PX4 development team [31].

### 3.1.2 Computation on UAV

As the inspection drone is intended for indoor industrial environments, communication to a ground station can be assumed to be unreliable. An environment with concrete and metal clad walls can prove difficult for communication signals, further it can be expensive to equip, or undesirable to outfit an industrial complex with communication infrastructure like WiFi for the sole purpose of facilitating an inspection drone. Therefore it is desirable to design a solution where all the necessary navigational computations are preformed on and on-board computer.

To that end a small and compact computer is needed. Two different computers were available for use in the project from the beginning, the Nvidia Jetson TX2i module with associated carrier board and the Nvidia Jetson AGX Xavier. Both computers are equipped with the CUDA capabilities needed for the Zed Mini stereo camera.

The choice ended on the Nvidia Jetson TX2i based on size and weight constraints.

### 3.1.3 Proposed hybrid filter architecture

The proposed hybrid filter is a filter architecture that uses a loosely coupled Kalman- and particle-filter to utilize the strength of both filtering approaches while simultaneously trying to avoid their weaknesses.

The Kalman filter will be used as the primary filter; the Kalman filter shines when the underlying probability distribution is Gaussian, and the system model and measurement equations are linear or linearizable. This is the case for the IMU sensor models used to navigate, both the accelerometer and gyroscope sensor models can be linearized, and the Gaussian assumptions firmly hold. The Kalman filter is also capable of being computationally efficient with a large state vector. It is not a problem for the Kalman filter to contain the complete state vector describing the drone's position, orientation, and sensor biases. Since the Kalman filter needs a linearizable measurement equation, it is problematic to use a point cloud observation directly in the Kalman filter. Therefore this is left to the particle filter, and a most likely position and heading will be used in the Kalman filter, making for a now linear position and heading measurement model.

The particle filter will be used as a position and heading aiding filter for the Kalman filter. The particle filter relaxes the assumptions of an underlying Gaussian distribution and can handle nonlinear measurement models; that is, the measurement model does not need to produce an exact answer that relates directly to the states of the system but rather a probability of a proposed state being a true state. This is perfect for use with a point cloud-based measurement model. However, the particle filter needs to keep track of a large number of proposed solutions, the so-called particles in the filter. Therefore the state vector used to represent a particle needs to be kept to a minimum. This will allow each particle to represent a more considerable part of the state space. Therefore the particle filter's internal state will be reduced to the position, ie. X, Y, and Z coordinates and the heading (yaw) of the drone. Leaving the state vector to only contain the position and heading will require that the particle filter receives velocity information from an external source; that is, it will need to know how to propagate (move) the particles in space. The particle filter will also require the rest of the attitude (roll/pitch) for leveling the point-cloud during measurement. This information is received from the Kalman filter.

The proposed coupling of the filter can be seen in figure 3.1

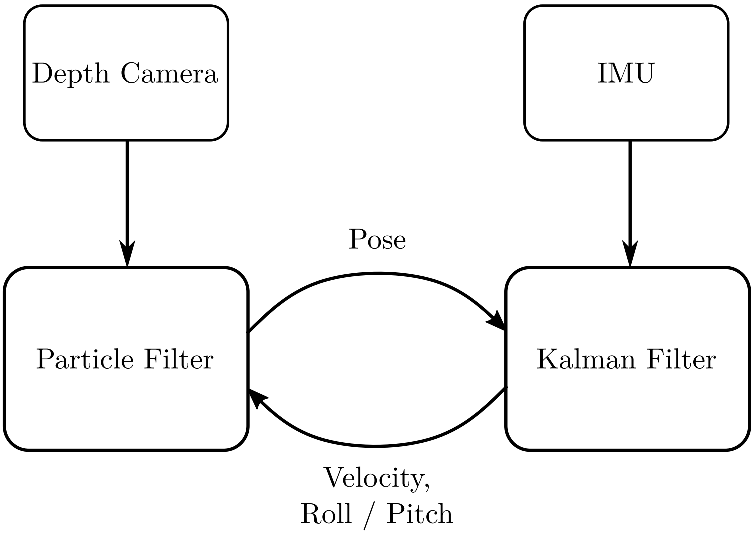


Figure 3.1: Proposed Hybrid filter coupling

### 3.1.4 Measurement model, Ray-cast vs. Likelihood-field

Because the system is going to be deployed to a single-board computer placed on the drone and executed real-time, the range finder sensor model ought to be pre-computed to reduce computational load. Pre-calculating the sensor model will turn the calculation of $p_{hit}$ into a simple indexing operation, drastically reducing computational load.

#### Ray-casting

The ray-casting algorithm described in section 2.5.2 can be pre-computed, but one needs to pre-calculate ray-casts for all possible orientations for each position. Thus, a pre-computed ray-cast sensor model will quickly increase in size, with each point $(x,\ y,\ z)$ in the map containing several pre-computed rays. The pre-cumputed ray-cast sensor model will quickly become several Gigabytes in size depending on the map size and resolution for both position and orientation.

#### Likelihood field

The likelihood field sensor model described in section 2.5.2 can also be pre-computed for the entire map, where for each position in the map contains only the probability of a sensor hit, yielding substantially smaller datasets compared to ray-cast. This will introduce some numerical errors as the map must be discretized to encode the likelihood of hits, but these errors are tiny for even relatively course maps.

#### Choice of model

The likelihood field sensor model was chosen due to its smoothness over $\mathbf{x}$ and its smaller data size when pre-computed compared to the ray cast method.

## 3.2 Software used

#### Python

Python is one of the worlds most popular programming languages and is a high-level, interpreted code language with a focus on object oriented programming and code-readability. The language is easy to pick up for new software developers as it is dynamically typed and code is grouped visually by indentations in the script.

Because of it's widespread use, python has great community support and a wide array of packages and libraries available. Notable packages used extensively in this project are:

- NumPy [15]: One of the most used libraries in Python, contains functionality for matrix- and array operations and is fully open source.
- Numba [23]: A package enabling Just in Time (JiT) compilation[1] for a subset of Python and NumPy code, enhancing performance during runtime.

#### Open3D

Open3D [43] is a modern, open source library for C++ and Python for working with 3D data. The library contains tools for point cloud manipulation and working with 3D models, and has been a vital part of creating the likelihood maps used for localization. The library can load common 3D model filetypes (.stl, .obj), which means that the environment can be modeled in for instance SolidWorks or Blender before being imported and converted in Python. Autodesk also have a tool which enables the export of Revit *Building Information Model*, or "BIM", models to the .stl file format, which would make them importable into Open3D.

#### ROS 2

"Robot Operating System" or "ROS" for short is a framework for writing robot software, where ROS 2 is the newest release. ROS is a set of tools that aims to make creating modular robot software easier by allowing programs (nodes) to communicate across multiple machines or internally using pre-defined topics. ROS 2 targets newer versions of C++ and Python, and is set up for object-oriented programming using timers and callbacks for execution of subroutines. The ROS ecosystem includes a lot of pre-compiled packages and tools to boost development of high complexity robot software.

ROS 2 "Eloquent Elusor" (codename 'eloquent') is the newest ROS 2 distribution targeting ubuntu 18.04 LTS and is the chosen distribution for this project. Although eloquent is not listed as a long term support package, both PX4 and StereoLabs had pre-made packages for this distribution, making sensor data from the hardware easily accessible on the ROS network. ROS also provides packages to interface with the Gazebo simulator. Thus enabling the creation of a simulated drone armed with the same sort of sensors which will be available on the physical system, forming a good platform for development and testing.

### 3.2.1 PX4 Development environment

The PX4 development environment consists of tools and software to control and get sensor data from the simulated drone. The simulator used is Gazebo, as this is the most popular

[1] JiT-compilation is detailed further in section 3.11.1

simulation-environment used with ROS and the PX4 *Software In The Loop* (SITL) simulation also integrates directly with Gazebo. This gives a simulation-platform with a highly customizable drone which is controlled and responds just like a physical system running the PX4 flight stack.

#### Gazebo

*Gazebo* is an open-source simulation environment focused on robotics simulation, and sees widespread use with ROS. Many pre-made plugins exist, enabling the placement of sensors such as IMUs and depth cameras in the simulation. Though packages and plugins exist for ROS2, it is under continuous development by the maintainers, and during work with the project, the documentation was a bit lackluster.

The simulated drone, sensor models, and environment will be detailed further in section 3.4

#### PX4 SITL

The PX4 SITL simulation simulates the full PX4 flight stack on a host computer, enabling testing and interface with the flight controller software in the same manner as with a physical system. The SITL simulation is also available for a custom *Real Time Publish Subscribe* (RTPS) firmware implementation, which enables publishing internal sensor data from the flight controller onto the ROS network. Running the PX4 flight stack on the simulated drone also enables controlling its autopilot through third-party software.

#### QGroundControl

*QGroundControl* (QGC) is an open-source software package enabling control and path planning for MAVLINK-enabled systems and has been the primary control interface against the simulated drone. The software enables manual control of the simulated drone running the SITL simulation using either on-screen virtual joysticks or by connecting a physical gamepad. For general testing of the system, an Xbox 360 controller has been used - which is plug-and-play with QGroundControl and fully customizable. For the recording of test results, the "mission" feature of QGroundControl is used, where the drone follows a user-defined path at a set velocity, making the flight more repeatable for consecutive tests.

#### Git

Git is a free, open-source distributed version control system and has been used extensively in the project. The entire code-base for the project is located on the GitLab group for the project, and crucial third-party software has been forked to avoid version inconsistencies in the case of new updates.

There have been set up repositories for all of the developed software packages. In addition, some main ROS2 workspaces have their repository setup with multiple of the other developed packages included as *submodules*. Submodules are repositories within repositories, where each submodule points to a specific commit in the version history of the target repository. Setting the main ROS workspaces up with submodules enables easy deployment to new locations and hardware for testing without having to clone down multiple repositories independently.

## 3.3 Choice of frames

Multiple different frames of reference is used for the system. Keeping track of the different frames and the transforms between them is critical for the correct operation of the system. In this section the different frames in the system will be shown and their choices motivated.

Table 3.1: The different frames used in the system

| Frame | Description |
|---|---|
| m | Map frame |
| n | NED frame |
| b | Body frame |
| s | Sensor frame (IMU) |
| l | Level body frame |
| p | Particle frame |

### 3.3.1 NED navigation frame

The main navigation frame is chosen to be a NED (north east down) frame, placed on a tangential plane on earths surface. That is; the x-axis pointing towards earth geographical north axis, the y-axis is pointing eastwards, leaving the z-axis to point straight down, perpendicular to the tangential plane formed by the x- and y-axis.

This choice for a main navigation frame makes it convenient for possible future integration as coordinates in a NED frame can be converted to latitude, longitude and altitude without much hassle, making integration into a GPS driven navigation system tangible.

NED frame coordinates can also be converted into navigation frames centred at the center of the earth. That is the earth centred earth inertial (ECEI) and earth centred earth fixed (ECEF) frames. Also allowing for future integration into navigation systems utilizing those frames.

### 3.3.2 Map frame

The main navigation frame in the system is the NED-frame. The reasoning for the introduction of the separate map-frame is two-fold. The main reason is flexibility; it is not given that the building the map represents is oriented in such a way that the NED-frame is a natural frame to use for the map. Having the possibility to generate the map separate from the NED-frame, increases the flexibility of the system. The secondary reason is practicality; Gazebo uses a coordinate system with Z-up, defining the map-frame similarly will make importing the maps into Gazebo easier.

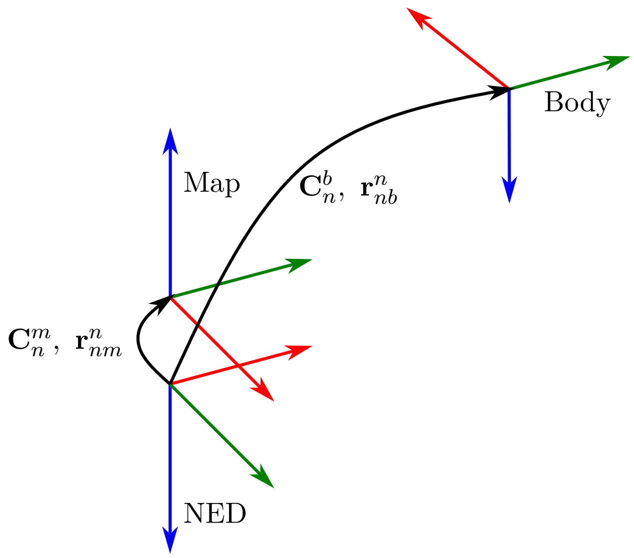



Figure 3.2: The map, NED and body-frame

### 3.3.3 Body frames

There are multiple frames on the drone. The camera and IMU sensor frames are rigidly connected to the body frame, while the level body frame shares origin with body while keeping level in NED ($z_l \parallel z_n$). Figures 3.3 and 3.4 show the different frames and their relation to each other.

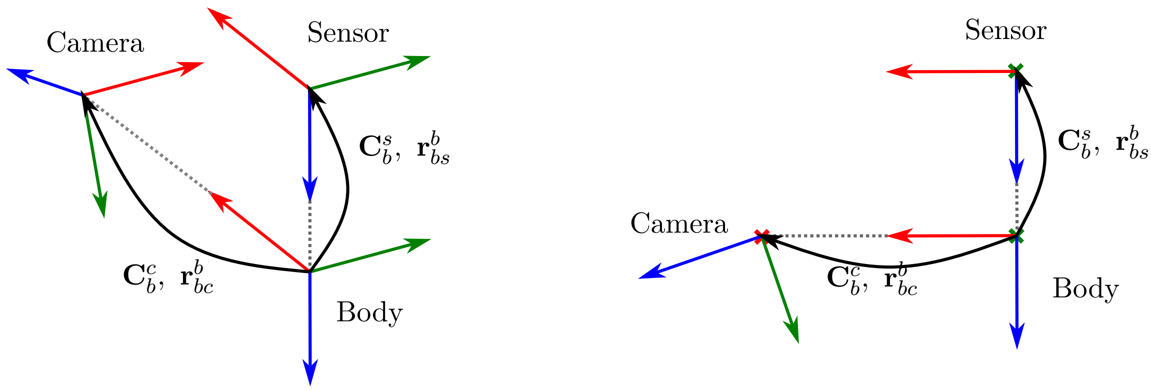



Figure 3.3: The body, sensor and camera frame

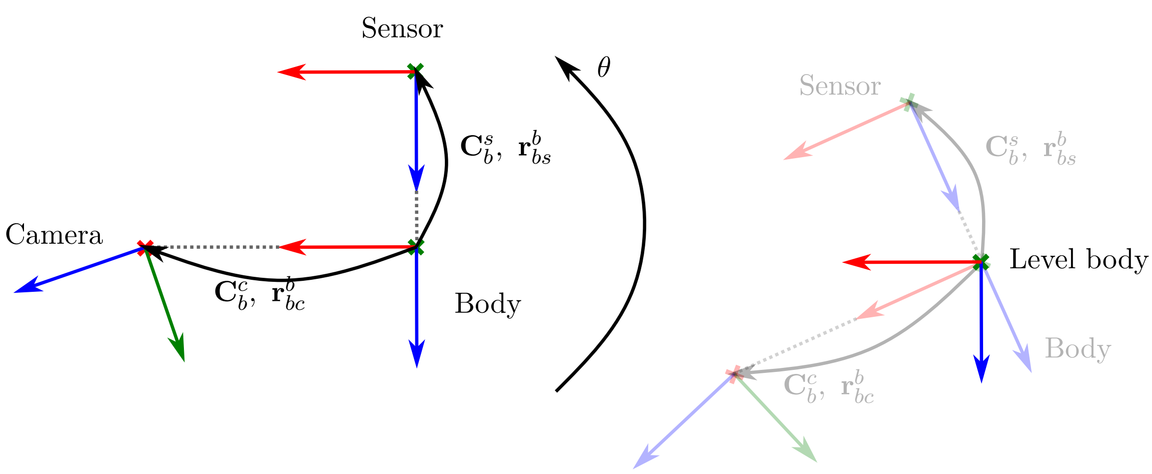



Figure 3.4: The body and level body frame

The level body frame is used to give velocity estimates from the Kalman filter to the particle filter. These velocities are used in the propagation-step, and must be in the level frame as the particle filter contain no estimate of the roll and pitch of the drone.

### 3.3.4 Particle frame

Each particle in the particle filter has its own frame, with the Z-axis parallel to the Z-axis of the map frame and x-axis oriented based on the particle heading. Each particle represents a hypothesis for the position and heading of the drone in the map.

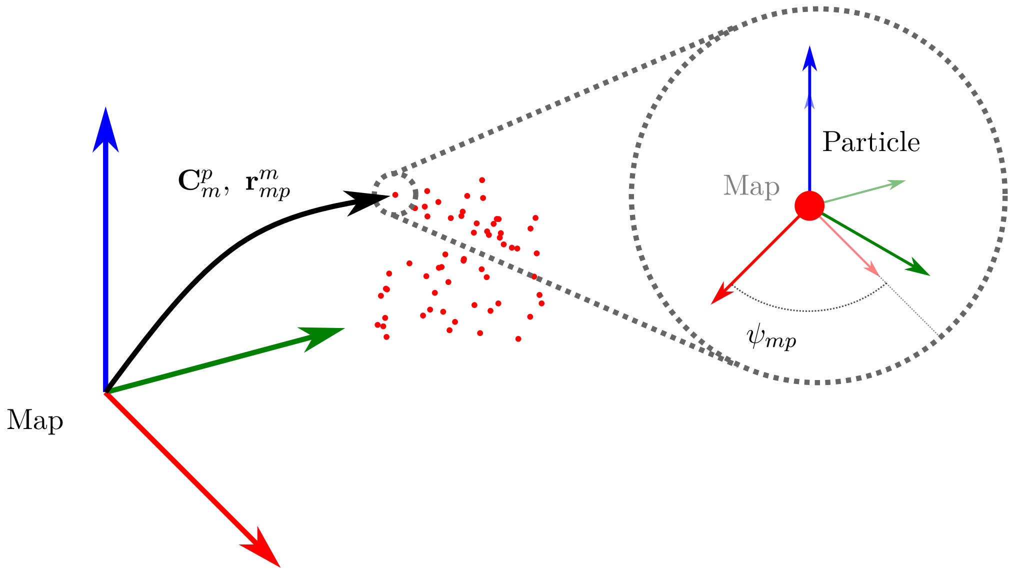



Figure 3.5: The particle frame and map frame

The transformation from the map to the particle frame is described by the particles coordinate and heading.

$$\mathbf{r}_{mp}^m = \left[x_{mp}^m, y_{mp}^m, z_{mp}^m\right] \qquad \mathbf{C}_m^p = \mathbf{C}_z(\psi_{mp}) \tag{3.1}$$

## 3.4 Simulation

### 3.4.1 Gazebo simulation environment models

#### Drone model

The drone model is based on the *IRIS 3DR* drone model modeled by the PX4/gazebo development community[6]. Some slight modifications have been made to the drone model, this includes:

- An IMU in the position where the PX4 flight controller is located
- Added a zed mini camera model

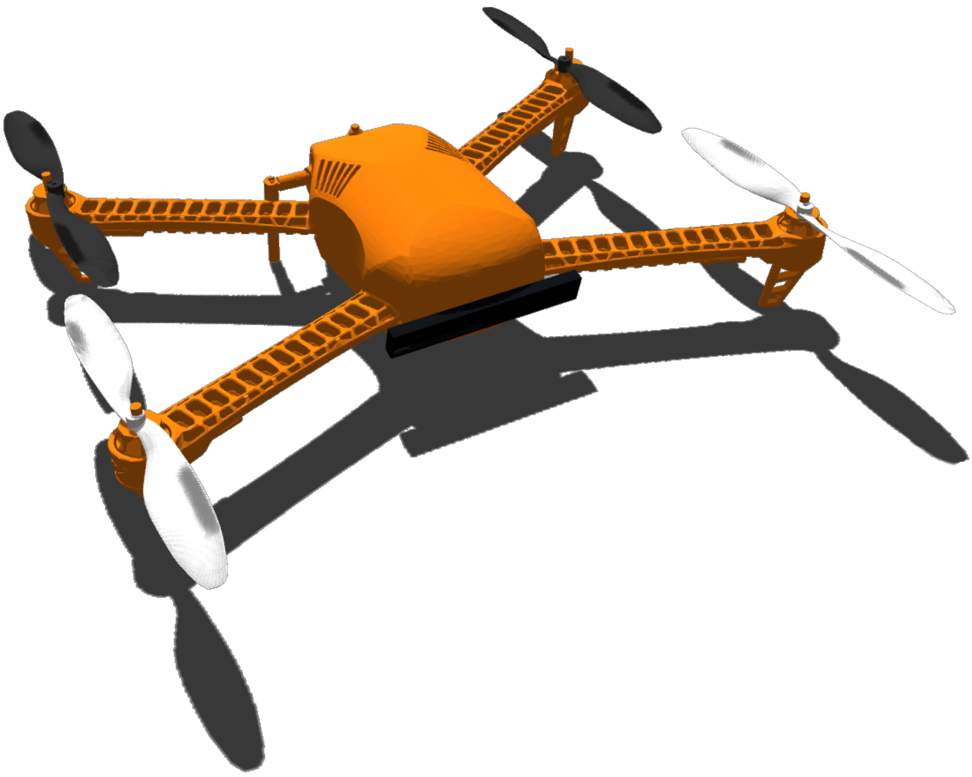


Figure 3.6: Simulation drone model based on IRIS 3DR

#### IMU sensor model

An IMU sensor model has been added to the simulated drone. This IMU is located in the same location as the pixhawk 4 IMU and serves as an IMU that is accessible natively in the Gazebo simulation and publishes data to the ROS network. This is in contrast to the IMU that is a part of the PX4 SITL flight controller in the simulation; data from the SITL IMU is only accessible when the *PX4_SITL* "micrortps_agent" is running[2].

The IMU model added is a standard model in the gazebo environment and has some key parameters that need to be filled in for the sensor model to accurately model and IMU, that is, the sensor bias and the sensor noise characteristics[12][13].

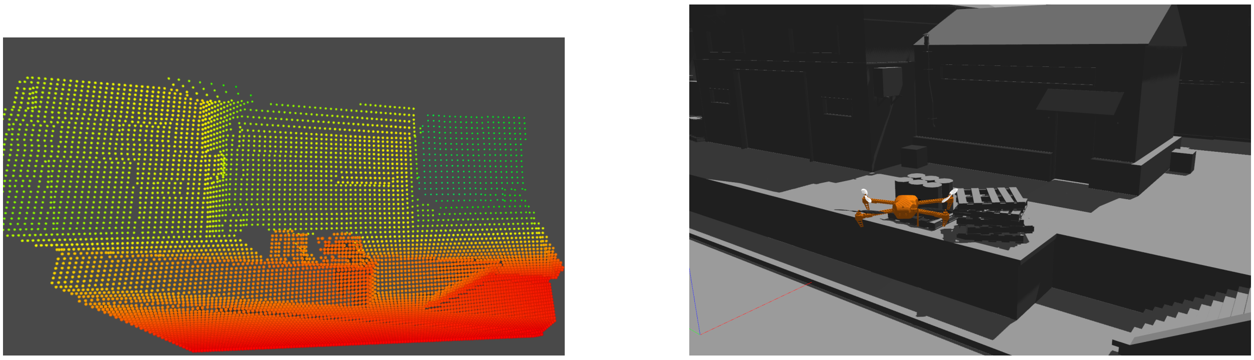
[2]This will be covered in more detail later in the report

#### Camera sensor model

The Zed mini camera chosen for depth perception of the environment is not models in the Gazebo environment. A close kin to the Zed mini is, the Intel real sens depth cameras. Therefore a model was created based on this model. An additional IMU was placed in the camera to model the IMU percent on the Zed mini.

The noise characteristics that can be added to the camera's depth model are barrel distortion, Gaussian noise, and a constant offset[13].

There is a maximum and minimum range of depth perception that can be set as well. In the simulation model, the max value is set higher than the approximately 15-meter max range of the Zed mini[35][3]. This is done to make visualization in the program *Rviz* easier. Any values that are further away than 15 meters are handled by the camera input function in the particle filter.



(a) Point cloud as seen in Rviz

(b) Environment surrounding the drone

Figure 3.7: Depth camera point cloud visualized in Rviz

Figure 3.7 displays how the depth camera perceives the environment model in Gazebo.

[3]The zed Mini can be set to perform in an *Ultra mode* mode and reach a max range of 24 meters

**Environment model**

To simulate the operating environment of the drone, three different environment models were made. One small, simple map with many distinct features for early-stage development and testing. For late-stage testing, development, and validation, two more realistic environments were modeled. One based on a free industrial game asset found online, and one based on the basement at the University of Agder Campus Grimstad

**Small, simple environment**

One simple and small model with many different features making it an idealized test environment for early-stage testing and development. The map has a $5 \times 10$ meter footprint and walls that are 5 meters tall.

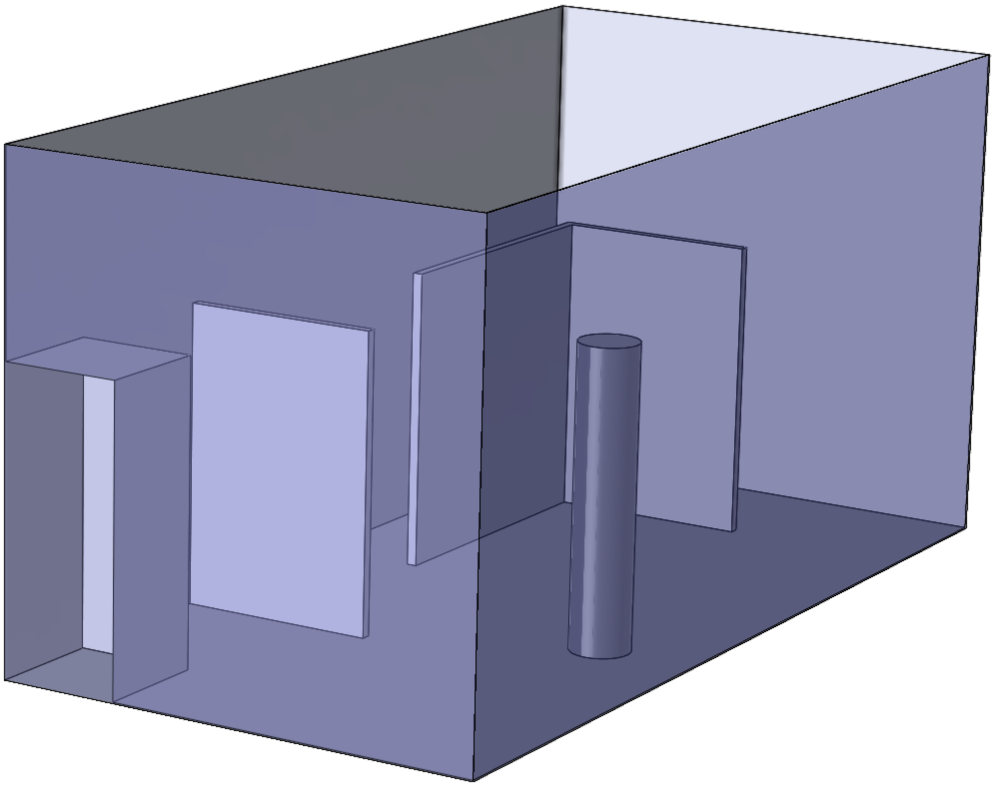


Figure 3.8: Simple environment as seen in the Gazebo simulation tool

Figure 3.8 displays the small environment; here the many "clean" features can be seen. The front two walls have been made transparent for the visualization of the features within the walls.

The environment was modeled using Solidworks and imported into Gazebo in an STL file format. The test environment has no collision model, and only appears in the Gazebo simulation as a visual entity. This is great for testing as it means that collisions with walls can not occur and makes it a comfortable environment to fly in manually during testing.

**Industrial environment**

The industrial environment is a large outdoor area resembling a multi-building industrial complex. The footprint of the environment is roughly $150 \times 150$ meters and building with features up to roughly 10 in height. Even though the primary use case for the proposed Hybrid-filter is indoor applications where GPS and magnetometer sensors are denied, this is still a real test case because it is an industrial area. The model is realistically clouted and has many long sight-lines that are longer than the max range of the sensor. This makes for a challenging environment and a good test for the proposed system.

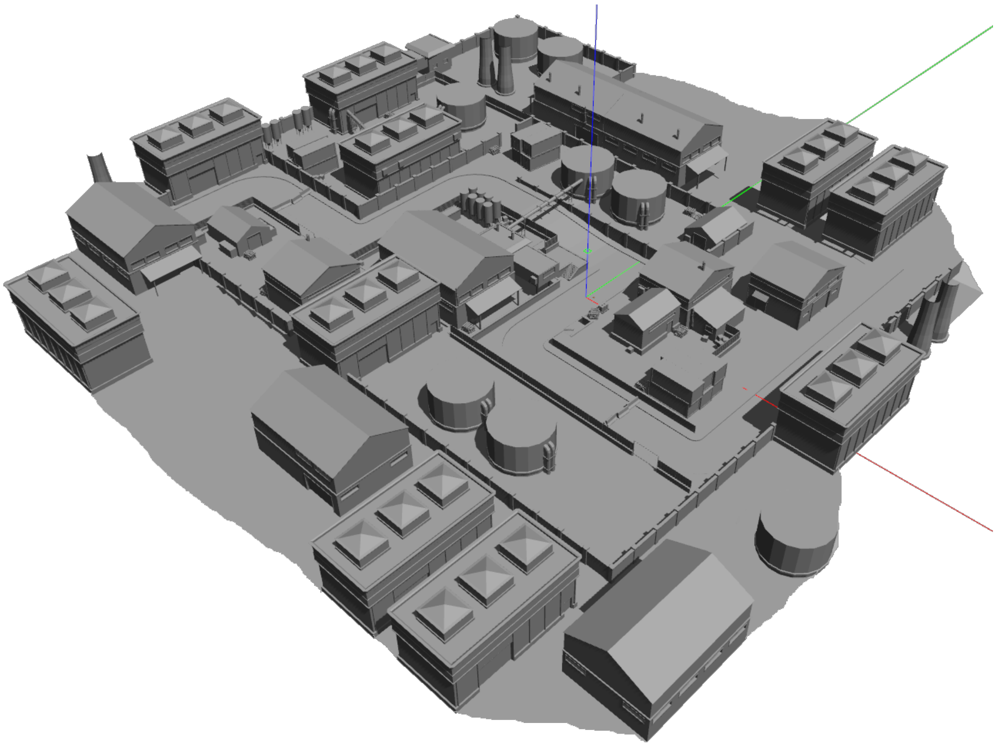


Figure 3.9: Industrial environment seen in the Gazebo simulation tool

Figure 3.9 displays the industrial environment, the model is made by *Dmitrii Kutsenko* and made available free for download under a royalty free licence[22].

**University of Agder Campus Grimstad Basement**

The model of the UiA campus basement is based on the footprint drawings of the building and is therefore suitably dimensional accurate to represent the actual environment; some features like doors and door frames have not been fully modeled. The ceiling in the Gazebo model is made visually transparent with a blue tint but is still an object the depth camera detects. This makes it easy to see where the drone is in the simulation and makes manual flying of the drone in the simulation feasible.

The environment consists of long hallways that have similar features in the length direction of the hallway. Therefore it will be a challenging test environment for the proposed system to determine its position along the length of a given hallway.

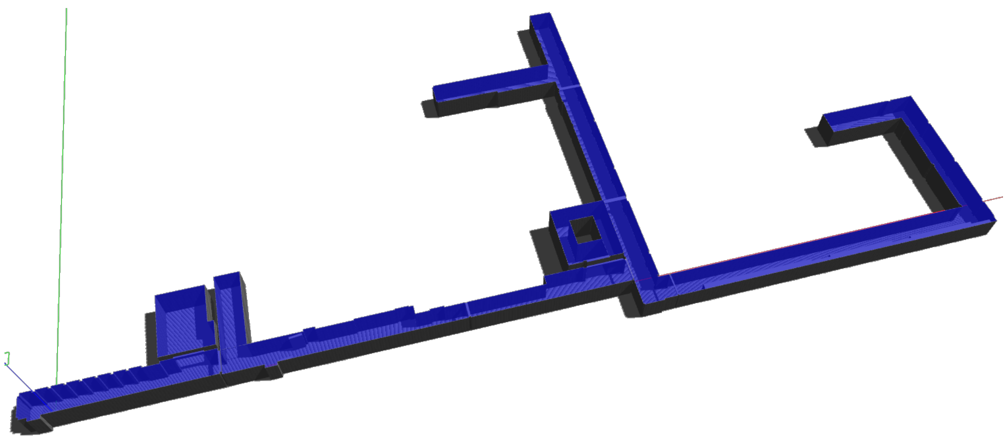


Figure 3.10: Univeristy of Agder basement environment seen in Gazebo simulation tool

### 3.4.2 Gazebo ground truth publisher

The gazebo environment does not natively provide data about the simulated drones true position, velocity or orientation, i.e. the true state of the drone. This data is important in validating the Hybrid-filter system's performance. This information is obtained trough the use of plugins in the model. These plugins have been placed in the drone's center of mass and provides information about the drone's position, orientation and linear- and angular-velocity. The data from these plugins are read into the ground truth publisher node, which was written to refactor this data to the desired format and publish it to the ROS network.

### 3.4.3 Modes of simulation

The Gazebo simulation environment has been used to develop, test, and validate the Hybrid-filter system software components and the complete Hybrid-filter. In addition, the sensor data from the drone is made available on the ROS network. This gives great flexibility for simulation and allows for both SITL and HIL simulation.

#### SITL simulation

The primary mode of simulation has been done with the developed software in the loop. In this simulation regime, the developed software runs as part of the simulation on the simulation host machine. This means that the same computer is running both the simulation and the production software. This mode of simulation is easy to set up and makes rapid prototyping and development of software possible. Another great advantage of using the production code in a simulation environment is that many of the bugs that otherwise would be present during the integration stage of the development process can be resolved during code development.

SITL simulation can be a tricky endeavor as the simulation host machine is also that machine running the production software. This results in the production software being executed on a different platform than what it will be deployed on. Further, it is dependent on the simulation host machine having the resources to process both the simulation and the deployed software. Therefore care must be taken when evaluating results regarding computation resources used to form a SITL simulation.

#### HIL simulation

Hardware in the loop simulations is a regime of simulations where the simulation is executed on a simulation host machine, and the production software is executed on the intended platform. This means that the production software is running on the Nvidia Jetson TX2. This simulation method allows for testing and evaluation of the execution rate of the developed software and making sure that it is feasible to run on the intended hardware.

## 3.5 Map

The likelihood map, introduced as a 2D-grid map of pre-computed values for the likelihood-field sensor model in [36] and extended to 3D in [28] is a discrete 3D grid-map. In the likelihood-map each cell contains the probability density for the distance $d$ taken from a gaussian distribution centered at the closest object.

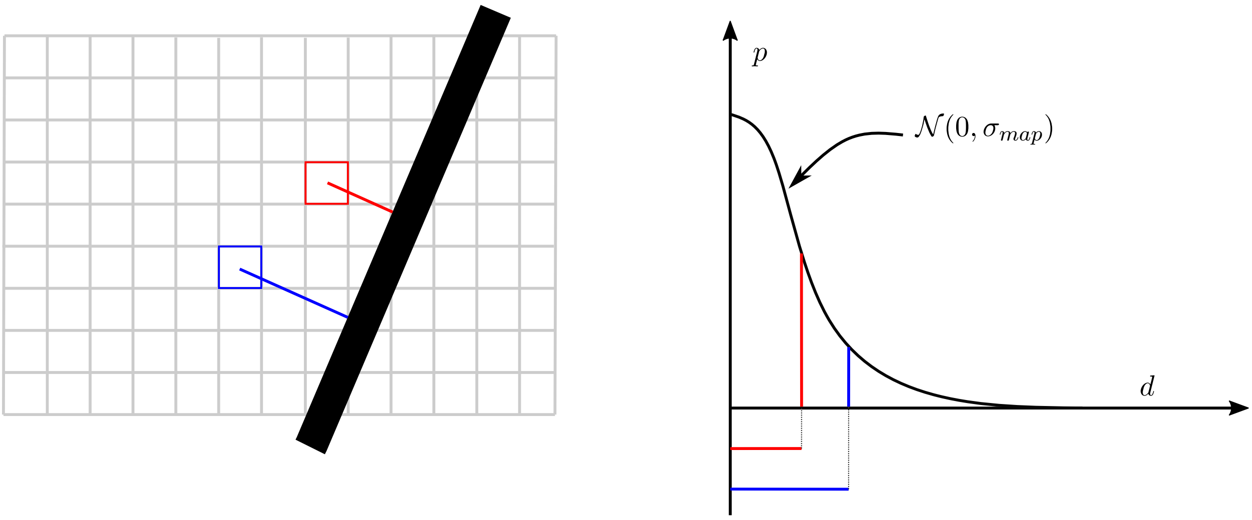



Figure 3.11: The main idea behind the likelihood map, demonstrated in 2D

The likelihood map is generated using functions defined in the Open3D library [43]. The map is a discrete pre-computation of the laser likelihood field sensor model described in section 2.5.2. A set of GUI-based tools have been created in order to simplify the generation and validation of the likelihood-map.

The map is stored as a three-dimensional NumPy array as this is known to be directly compatible with JIT-compiled Python programs using Numba.

### 3.5.1 Generating a likelihood map

The map generation script generates the full 3D likelihood map and metadata from a chosen model, where the map resolution and $\sigma_{map}$ is decided by the user at execution. The metadata include the map resolution and *origin offset* in meters, the size of the map in cells for the x- y- and z-direction, the maximum value for the gaussian used to compute the map probabilities and a bool signifying if the probabilities are encoded in unsigned 8-bit integers. The origin offset is the cartesian distance from the map origin (from the 3D-model) and cell [0, 0, 0] in the map, and is used when indexing from the map when origin is not cell [0, 0, 0].

The possibility to encode the probabilities stored in the map as UInt8's is motivated by memory usage. A 3D grid representation is not space-efficient, and it was discovered that the large industrial map used more than 1 GB of memory when created with probabilities using the float32 datatype. Changing the datatype to UInt8 results in roughly a quarter of the memory-usage. There exist more space efficient, tree-based mapping solutions, like *Octomap* [16]. At the time of writing this report these were not supported with JiT Compiled python programs, so a 3D NumPy array was used as this was known to work.

The map-generator script creates the map in two steps, the first step assigns the 3D grid over a chosen model and finds the distance from the center of each voxel to the closest point in the model, described in block-diagram form in figure 3.12. The size of the map is extended by $6\sigma_{map}$ in all directions to avoid sudden sharp changes in the likelihood-field around the edge of the model.

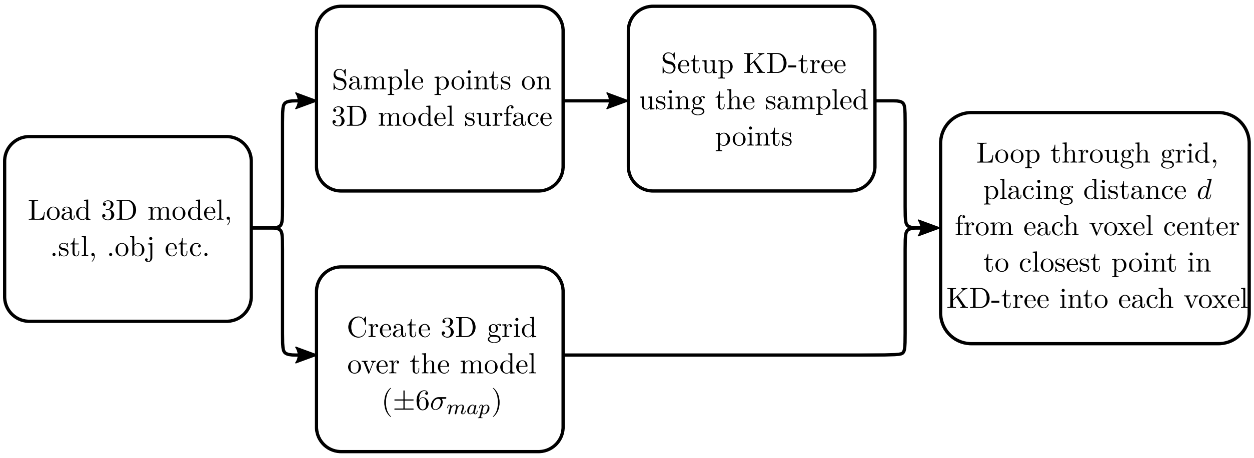



Figure 3.12: The main workflow of the map-generation script

The second step converts these distances into probabilities and encodes them into UInt8 data type if desired. The distance $d$ for each cell $(x, y, z)$ in the map is transformed to a probability of hit using a zero-mean Gaussian with the standard deviation $\sigma_{map}$ defined at program execution

$$p(x, y, z|\mathcal{M}) = \frac{1}{\sigma_{map} * \sqrt{2\pi}} \exp\left(\frac{-d(x, y, z|\mathcal{M})^2}{2\sigma_{map}^2}\right) \tag{3.2}$$

Encoding this probability into UInt8 is done by converting the probability from the range $[0,\ (\sigma_{map} \cdot \sqrt{2\pi})^{-1}] \in \mathbb{R}$ to $[0, 255] \in \mathbb{N}$. This introduces some discretization-error, but as the map is already divided into a discrete grid this error will have negligible effect. The theoretical maximum error introduced by this conversion will be half the resolution of the UInt8, which becomes:

$$e_{max} = \frac{1}{2 \cdot 255 \cdot \sqrt{2\pi} \cdot \sigma_{map}} \approx \frac{1}{1278.4 \cdot \sigma_{map}} \tag{3.3}$$

**Environment models as likelihood maps**

The three aforementioned environments (section 3.4.1) were all converted to likelihood-maps using a resolution of 10 $[cm]$ and a standard deviation $\sigma_{map}$ of 10 $[cm]$. To verify that the maps were created correctly, slices of each map was converted to images for visualization using the map slicer script. Figures 3.13 and 3.14 shows a slice of the likelihood map generated for the basement at UiA and industrial map, where the probabilities are scaled to be $\in [0, 255]$ and the colours inverted, showing more probable regions as darker colour.

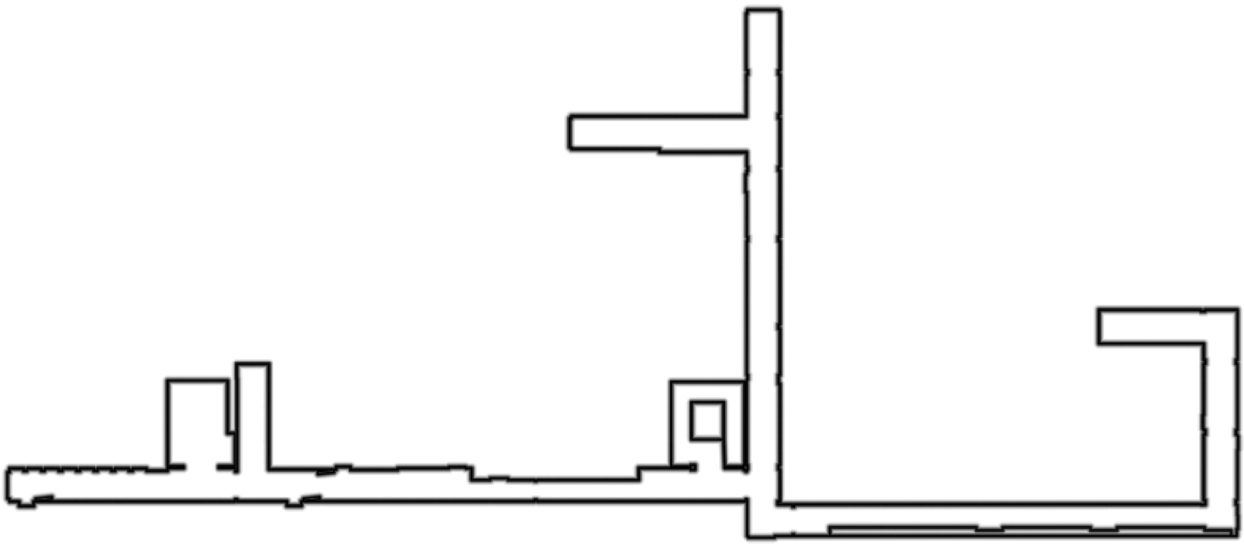


Figure 3.13: A slice at $z = 1$ $[m]$ from the generated likelihood-map for the UiA basement, darker regions are more probable hit locations

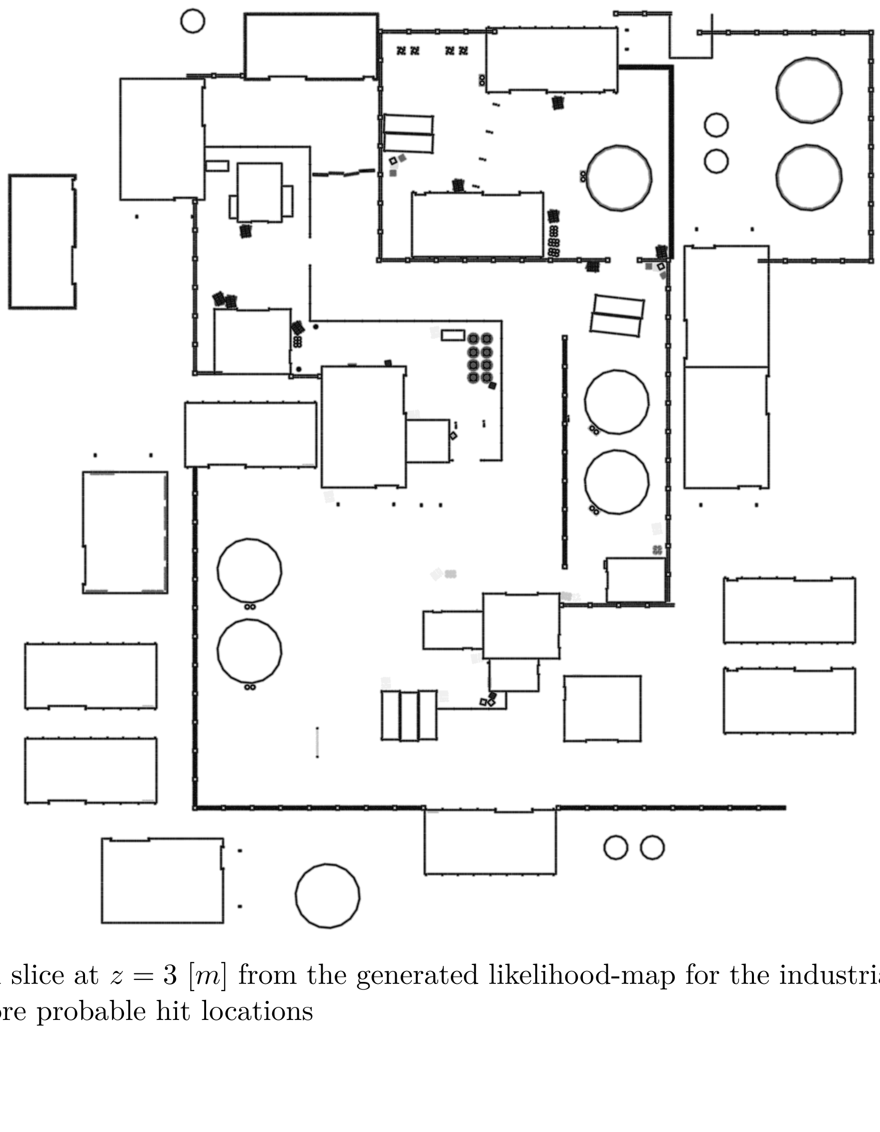


Figure 3.14: A slice at $z = 3$ $[m]$ from the generated likelihood-map for the industrial map, darker regions are more probable hit locations

## 3.6 Hybrid filter

The hybrid filter is split into a Kalman filter and a particle filter, where the architecture of the hybrid filter draws from the strengths of the different filter types. The Kalman filter is excellent at fusing IMU data, which contain Gaussian distributed noise, with position and heading measurements. Whilst the particle filter is better at dealing with non-Gaussian sensor models and multi-modal probability distributions.

As the system is designed to operate in an indoor, industrial environment, the classical UAV position and heading sensors (GPS and Magnetometer, and to some degree Barometer[4]) will not work reliably. This means that the Kalman filter will need position and heading aiding from another source - this is where the particle filter will fill in. Utilizing a likelihood map sensor model and a range-finder, the particle filter will localize the drone using the map and give position- and heading measurements to the Kalman filter. Since the drone is to be used in autonomous inspection tasks, it is believed that an initial position is known to some degree as the system is assumed to have a designated landing zone or charging station to rest between missions.

The hybrid filter architecture can be seen in figure 3.15. The position and yaw estimates ($[\mathbf{p}^n_{nb},\ \psi_{nb}]$) from the particle filter is passed to the Kalman filter, and the linear- and angular velocities in the level frame and the estimated euler-angles ($[\mathbf{v}^l_{nb},\ \omega^l_{nb},\ \mathbf{\Theta}_{nb}]$) are passed to the particle filter.

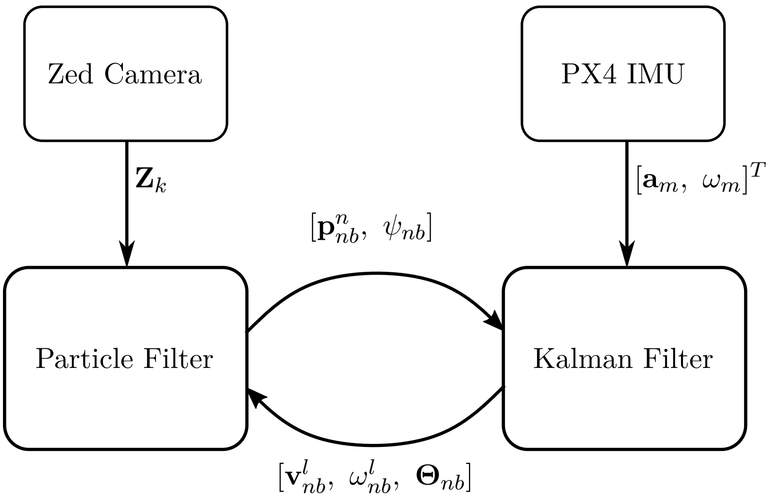



Figure 3.15: The hybrid filter architecture including sensors

### 3.6.1 Co-dependence of filters

The Hybrid filter is, as mentioned, split into a Kalman filter and a particle filter. The Kalman filter will be the "main estimator" in the system, fusing the sensor data from the IMU with the position and yaw estimates from the particle filter.

The two parts of the hybrid filter are dependant on each other. The Kalman filter receives the position and yaw estimate from the particle filter as measurements, correcting its estimates based on integrated IMU data. The particle filter uses the velocity estimate from the Kalman filter to move the particles in the state space.

[4]Due to indoor ventilation systems, the indoor barometric pressure might fluctuate

### 3.6.2 Filter separation

The filters are setup in a way that makes them separable, and the ground truth data publisher described in section 3.4.2 is used to test the filters separately before integration. The ground truth data publisher gives true data from the simulated environment in the same form that the filters output, so it will use the same interface for passing data into the filters. The interface between the system and ground truth publisher can be seen in figure 3.16.

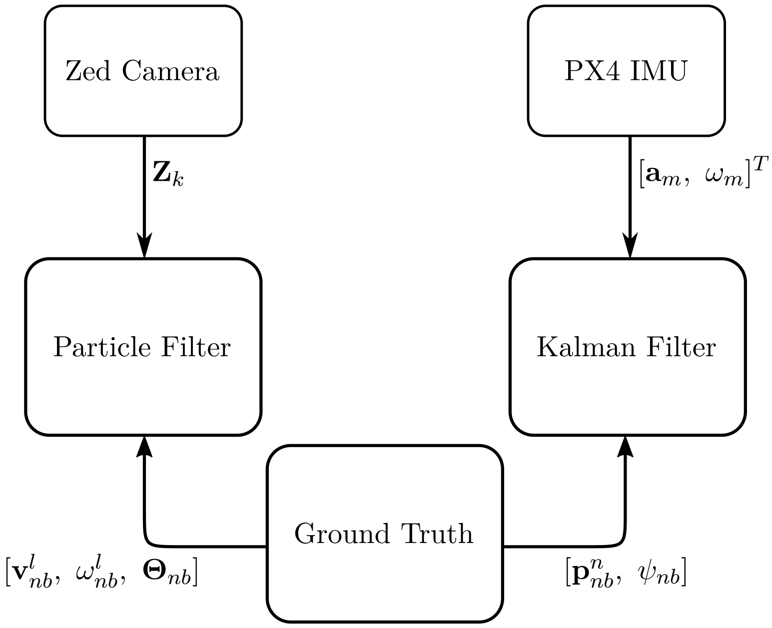



Figure 3.16: The hybrid filter architecture including sensors, with ground truth breaking dependence

Feeding the filters with ground truth data removes the problem of co-dependence during testing, and enables completely isolated development of the two filters before system integration, as long as the communication-interface is established.

## 3.7 Kalman filter

The Kalman filter is derived in such a way that it operates on the position and orientations and the linear- and angular velocity of the sensor. The state of the sensor is then later related to the state roughly at the center of mass of the drone.

The reason for choosing to use sensor-centered states is that this is the location where the actual IMU measurements are taking place; integrating the measurements in the sensor frame will avoid numerical errors introduced by translating them to the center of gravity of the drone before integration.

For later control purposes, the states at the center of gravity are the most useful. Therefore the sensor states are translated to this frame in the output equations of the filter.

As mentioned in the concepts section 3.1 the Kalman filter is based on the sensor models. That is the accelerometer and gyroscope dynamics.

The states in the filter are related to the senors states:

$$\mathbf{x} = \begin{bmatrix} \mathbf{p}^n_{ns} & \mathbf{v}^n_{ns} & \mathbf{a}_\beta & \mathbf{\Theta}_{ns} & \omega_\beta \end{bmatrix}^T$$

All but the states regarding the Euler angles are non constrained, that is to say they have a free domain over the real numbers.

The Euler angles are defined to be wrapped on the unit circle with:

$$\{\phi, \theta\} \in (-\pi, \pi], \qquad \psi \in (0, 2\pi] \tag{3.4}$$

### 3.7.1 State transition model

The state transition model for the filter is a kinematic model with the following linear dynamics:

$$\dot{\mathbf{p}}^n_{ns} = \mathbf{v}^n_{ns} \tag{3.5}$$

$$\dot{\mathbf{v}}^n_{ns} = \mathbf{a}^n_{ns} \tag{3.6}$$

The angular dynamics is based on Euler angles and a derivation of the rate transform matrix $T(\mathbf{\Theta}_{ns})$ can be found in section 2.1.3.

$$\dot{\Theta}_{ns} = \mathbf{T}(\mathbf{\Theta}_{ns})\omega^b_{ns} \tag{3.7}$$

The accelerometer and gyroscope bias dynamic are also included as states, and the state dynamics are based on the sensor model discussed in section 2.5.1

$$\dot{\mathbf{a}}_\beta = \mathbf{0} \tag{3.8}$$

$$\dot{\omega}_\beta = \mathbf{0} \tag{3.9}$$

**Accelerometer model**

Starting with equation 2.56 and remembering that the accelerometer measures the proper acceleration. Where it is the coordinate acceleration that is of interest for the navigation solution.

$$\mathbf{a}_m = \mathbf{C}_n^s[\mathbf{a}_{ns}^n + \mathbf{g}^n] + \mathbf{a}_\beta + \mathbf{a}_n \tag{3.10}$$

Solving the measurement equation 3.10 for the coordinate acceleration $\mathbf{a}_{ns}^n$ gives[5]:

$$\mathbf{a}_{ns}^n = \mathbf{C}_s^n[\mathbf{a}_m - \mathbf{a}_\beta + \mathbf{a}_n] - \mathbf{g}^n \tag{3.11}$$

A new variable $\mathbf{a}_c$ is introduced and defined as:

$$\mathbf{a}_c = \mathbf{a_m} - \mathbf{C}_n^s\mathbf{g}^n \tag{3.12}$$

The value of $\mathbf{a}_c$ is calculated every time a new acceleration measurement is received from the accelerometer. Substituting equation 3.12 into 3.11 gives the final coordinate acceleration equation:

$$\mathbf{a}_{ns}^n = \mathbf{C}_s^n[\mathbf{a}_c - \mathbf{a}_\beta + \mathbf{a}_n] \tag{3.13}$$

**Gyroscope model**

Beginning with the gyroscopes measurement equation 2.61 and solving for the body's angular velocity:

$$\omega_m = \omega_{nb}^b + \omega_\beta + \omega_n \tag{3.14}$$

$$\omega_{nb}^b = \omega_m - \omega_\beta + \omega_n \tag{3.15}$$

**Complete state transition model**

Summarizing the state transition equations and inserting the coordinate acceleration solved in equation 3.11 into 3.6, as well as inserting the body rates from equation 3.15 into 3.7 gives the complete state transition equations:

$$\frac{d}{dt}\mathbf{x} = f(\mathbf{x},\mathbf{u}) = \begin{bmatrix} \mathbf{v}_{ns}^n + \mathbf{v}_n \\ \mathbf{C}_s^n(\hat{\mathbf{\Theta}}_{ns})[\mathbf{a}_c - \mathbf{a}_\beta + \mathbf{a}_n] \\ \mathbf{a}_{n,\beta} \\ \mathbf{T}(\hat{\mathbf{\Theta}}_{ns})[\omega_m - \omega_\beta + \omega_n] \\ \omega_{n,\beta} \end{bmatrix} \tag{3.16}$$

### 3.7.2 Measurement equations

The measurement equations to the Kalman filter are detailed below

[5]Note that the zero mean Gaussian noise vector $\mathbf{a}_n$ has not switched sign, that is because a zero mean Gaussian distribution is symmetric around the *y-axis*

**Position measurement equation**

The position measurement is the best available estimated position from the particle filter and is estimated in the body frame of the UAV; the covariance of the estimate is also related to the body frame of the UAV. Therefore, they both need to be transformed into the sensor frame.

The position is transformed in the following way:

$$\mathbf{p}_{ns}^{n} = \mathbf{p}_{nb}^{n} + \mathbf{C}_{s}^{n}(\hat{\mathbf{\Theta}}_{ns})\mathbf{r}_{bs}^{s} \tag{3.17}$$

$$\mathbf{z}_{p} = \mathbf{p}_{ns}^{n} \tag{3.18}$$

Here there is uncertainty in both the position estimate given by particle filter $\mathbf{p}_{nb}^{n}$ and in the Euler parameterization of the DCM $\mathbf{C}_{b}^{n}$.

The covariance matrix $\mathbf{R}_{p}$ accompanying the position estimate $\mathbf{p}_{ns}^{n}$ can then be calculated in accordance with the covariance equation detailed in section 2.2.3.

$$\mathbf{R}_{p} = \mathbf{R}_{p,pf} + \mathbf{W}_{p}\mathbf{P}_{\mathbf{\Theta}}\mathbf{W}_{p}^{T} \tag{3.19}$$

Where the matrix $\mathbf{W}_{p}$ is the partial derivative of the position measurement equation 3.18 with respect to the Euler vector constituting the DCM $\mathbf{C}_{s}^{n}$:

$$\mathbf{W}_{p} = \frac{\partial \mathbf{z}_{p}}{\partial \mathbf{\Theta}_{ns}} \tag{3.20}$$

The covariance matrix $\mathbf{P}_{\Theta}$ is extracted from the state uncertainty matrix at the time when the measurement takes place, and the covariance matrix $\mathbf{R}_{p,pf}$ is the accompanying covariance to the position estimate from the particle filter.

**Yaw measurement equation**

The yaw measurement is also the best available estimate from the particle filter. The yaw estimate is estimated in the body centred frame. The Kalman filter operates in the sensor frame. Since both frames are attached to a rigid body and has the same orientation, the estimate from the particle filter can be used directly as a measurement in the Kalman filter.

$$z_{\psi} = \psi_{nb} \tag{3.21}$$

The accompanying measurement covariance matrix $\mathbf{R}_{\psi,pf}$ is the estimated covariance from the particle filter.

$$\mathbf{R}_{\psi} = \mathbf{R}_{\psi,pf} \tag{3.22}$$

**Leveling**

Since the gravity vector is present in the proper acceleration measured by the accelerometer, it can be used to infer the attitude of the UAV. This is the process of leveling.

Leveling is necessary in the step of identifying the gyroscope bias and also making sure that the attitude estimate of the UAV dose not drift over time.

The process of leveling is dependant on assuming that the coordinate acceleration is close to zero and that the accelerometer biases are either identified or assumed equal to zero.

Starting with the proper acceleration equation for the accelerometer 2.56

$$\mathbf{a}_m = \mathbf{C}_n^s[\mathbf{a}_{ns}^n - \mathbf{g}^n] + \mathbf{a}_\beta + \mathbf{a}_n \tag{3.23}$$

The above assumptions leads to:

$$\mathbf{a}_m = -\mathbf{C}_n^s \mathbf{g}^n \tag{3.24}$$

This equation is valid under the assumption that the coordinate acceleration $\mathbf{a}_{ns}^n$ is small, this can be applied in the Kalman filter by only doing leveling when:

$$|\mathbf{a}_m| \in [g(1-\epsilon), g(1+\epsilon)] \tag{3.25}$$

Where $g$ is the absolute value of the earths gravitational acceleration and $\epsilon$ is value that determines the narrowness of the leveling window.

To infer the roll and pitch angles equation 3.24 must be solved for them, multiplying the gravity vector $\mathbf{g}^n$ with the DCM $\mathbf{C}_n^b$ gives:

$$\mathbf{a}_m = g \begin{bmatrix} sin\theta \\ -cos\theta sin\phi \\ -cos\theta cos\phi \end{bmatrix} \tag{3.26}$$

By manipulating the elements in equation 3.26 the roll and pitch angles $\phi$ and $\theta$ can be solved for.

$$\phi = atan\left(\frac{a_{m,y}}{a_{m,z}}\right) \tag{3.27}$$

$$\theta = atan\left(\frac{a_{m,x}}{\sqrt{a_{m,y}^2 + a_{m,z}^2}}\right) \tag{3.28}$$

Both angles will involve a tangent function. This is undesirable as the tangent function is limited to half a circle in domain by definition, typically $\pm\frac{\pi}{2}$. This problem can be solved by using the $atan2$ function typically included in most math programming libraries. This function uses the signs of the arguments passed to determine what quadrant the argument is in, and thus the function has a domain that covers the full unit circle[7].

$$\phi_l = atan2\left(-a_{m,y},\ -a_{m,z}\right) \tag{3.29}$$

$$\theta_l = atan2\left(a_{m,x},\ \sqrt{a_{m,y}^2 + a_{m,z}^2}\right) \tag{3.30}$$

The signs in equation 3.29 is lost in the derivation of equation 3.27, therefore care must be taken when deriving the phi leveling function[6].

[6] the phi leveling equation when not simplifying the signs looks like: $\phi = atan\left(-a_{m,y}/-a_{m,z}\right)$, it is the numerator and denominator from this equation that is used as the arguments for the atan2 function

Equation 3.29 and 3.30 will be used as measurements for the roll and pitch angles in the Kalman Filter, the subscript $l$ is used to denote leveling. Giving the measurement equation:

$$\mathbf{z}_l = \begin{bmatrix} \phi_l \\ \theta_l \end{bmatrix} \tag{3.31}$$

The covariance matrix for the roll and pitch leveling is designed in such a way that the filter trust the measurement less the further away the measurement is from being only a measurement of the gravity vector[25].

This is accomplished by first calculating how much the measurement deviates from from the gravity vector. Then using this deviation to determine the measurement covariance.

$$\delta g = ||\mathbf{a}_m| - g| \tag{3.32}$$

$$r_l = \sigma_l \left(1 + k_l \left(\delta g + \delta g^2\right)\right) \tag{3.33}$$

$$\mathbf{R_l} = r_l \mathbf{I}_{2\times 2} \tag{3.34}$$

It can here be seen that the further the measurement is from the gravity vector, the less the measurement is trusted. This method of dynamic tuning nicely accompanies the method of only doing leveling when the measured accelerations is within a certain threshold of the gravitational acceleration. It can be seen that when there is no deviation the value is the variance $\sigma_l$ set to the covariance matrix $\mathbf{R}_l$. Further $\sigma_l$ and $k_l$ are parameters to be tuned.

### 3.7.3 Output equations

The outputs of the Kalman filter are detailed below.

#### Position output equation

The position of interest for control purposes is the body centred position. Therefore the sensor position and sensor position state uncertainty covariance must be transformed. This is done much the same way as for the position measurement input equation.

$$\hat{\mathbf{p}}^n_{nb} = \hat{\mathbf{p}}^n_{ns} - \mathbf{C}^n_s(\hat{\mathbf{\Theta}}_{ns})\mathbf{r}^s_{bs} \tag{3.35}$$

The estimated associated state covariance is calculated as:

$$\mathbf{P}_{p,b} = \mathbf{P}_{p,s} + \mathbf{W}_p \mathbf{P}_\Theta \mathbf{W}_p^T \tag{3.36}$$

Where the matrix $\mathbf{W}_p$ is the partial derivative of the position output equation with respect to the Euler parameter vector $\mathbf{\Theta}_{ns}$

$$\mathbf{W}_p = \frac{\partial \hat{\mathbf{p}}^n_{nb}}{\partial \mathbf{\Theta}_{ns}} \tag{3.37}$$

#### Angular rate output equation

The angular rate output is a bias corrected senor measurement:

$$\omega_{ns}^{b} = \omega_m - \hat{\omega}_\beta \tag{3.38}$$

Where $\omega_m$ is the measured angular rate.

#### Velocity output equation

The velocities in the Kalman filter velocity states in the Kalman filter is estimated in the sensor frame and resolved in the NED frame. These velocity components must be transformed to the body frame and resolved in the Level frame.

$$\hat{\mathbf{v}}_{nb}^{l} = \mathbf{C}_s^l(\hat{\mathbf{\Theta}}_{ns})\left[\hat{\mathbf{v}}_{ns}^{s} - \mathbf{S}(\hat{\omega}_n^s s)\mathbf{r}_{bs}^{s}\right] \tag{3.39}$$

The covariance is calculated as:

$$\mathbf{P}_{v,b} = \mathbf{P}_{v,s} + \mathbf{W}_v \mathbf{P}_\theta \mathbf{W}_v^T \tag{3.40}$$

Where:

$$\mathbf{W}_v = \frac{\partial \hat{\mathbf{v}}_{nb}^{l}}{\partial \mathbf{\Theta}_{ns}} \tag{3.41}$$

#### Remaining output equations

The remaining outputs form the filter is simply just the estimated states accompanied with their estimated state uncertainty covariance.

## 3.8 Kalman filter Implementation

### 3.8.1 Linearization of state transition equation

For implementation in the Kalman filter, the state transition equation 3.16 must be linearized:

$$\frac{d}{dt}\mathbf{x} = f(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} \mathbf{v}_{ns}^{n} \\ \mathbf{C}_s^n(\hat{\mathbf{\Theta}}_{ns})[\mathbf{a}_c - \mathbf{a}_\beta + \mathbf{a}_n] \\ \mathbf{a}_{n,\beta} \\ \mathbf{T}(\hat{\mathbf{\Theta}}_{ns})[\omega_m - \omega_\beta + \omega_n] \\ \omega_{n,\beta} \end{bmatrix}$$

Remembering that the states in the filter are:

$$\mathbf{x} = \begin{bmatrix} \mathbf{p}_{ns}^{n} & \mathbf{v}_{ns}^{n} & \mathbf{a}_\beta & \mathbf{\Theta}_{ns} & \omega_\beta \end{bmatrix}^T$$

And defining the control inputs to the filter as the accelerometer and gyroscope sensor measurements:

$$\mathbf{u} = \begin{bmatrix} \mathbf{a}_c \\ \omega_m \end{bmatrix}$$

linearizing equation 3.16 first with respect to the state vector gives the state transition matrix **A**, then linearizing the state transition equation with respect to the control inputs gives the input matrix **B**

$$\mathbf{A}(\mathbf{x}) = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{bmatrix} \mathbf{0}_{3\times3} & \mathbf{I}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & -\mathbf{C}_s^n(\hat{\mathbf{\Theta}}_{ns}) & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & -\mathbf{T}(\hat{\mathbf{\Theta}}_{ns}) \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \end{bmatrix} \tag{3.42}$$

$$\mathbf{B}(\mathbf{x}) = \frac{\partial \mathbf{f}}{\partial \mathbf{u}} = \begin{bmatrix} \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{C}_s^n(\hat{\mathbf{\Theta}}_{ns}) & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{T}(\hat{\mathbf{\Theta}}_{ns}) \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \end{bmatrix} \tag{3.43}$$

The state-transition- and input-matrix must be discretized, this is done using the forward Euler method discussed in section 2.3.5:

$$\mathbf{F}(\mathbf{x}_{k-1}) = (\mathbf{I} + \mathbf{A}dt) \tag{3.44}$$

$$\mathbf{B}_d(\mathbf{x}_{k-1}) = \mathbf{B}dt \tag{3.45}$$

For propagation of the state uncertainty matrix the state transition equations must be linearized around the noise vectors, as well as discretized, here this is done by multiplying the result by $dt$, this gives:

$$\mathbf{W}_q = \frac{\partial \mathbf{f}}{\partial \mathbf{w}} dt = \begin{bmatrix} \mathbf{I}_{3\times3}dt & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{C}_s^n(\hat{\mathbf{\Theta}}_{ns})dt & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{I}_{3\times3}dt & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{T}(\hat{\mathbf{\Theta}}_{ns})dt & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{I}_{3\times3}dt \end{bmatrix} \tag{3.46}$$

The vector **w** is the vector os noise elements from the state transition equation:

$$\mathbf{w} = \begin{bmatrix} \mathbf{v}_n & \mathbf{a}_n & \mathbf{a}_{n,\beta} & \omega_n & \omega_{n,\beta} \end{bmatrix}^T \tag{3.47}$$

#### Euler angle constraints

Since the attitude and heading of the UAV is described by Euler angles and the domain is limited to lie on the unit circle there is a need to *wrap* the angles around the unit circle.

This is, if the yaw angle is $2\pi + 0.1$, then the angle should be 0.1, the same is also the case if the angle is $-0.1$, then it should be $2\pi - 0.1$. The same applies for the roll and pitch angles, although they are both wrapped at $\pm\pi$.

To respect this constraint the angles are wrapped each time the state is predicted:

The yaw is wrapped using the modulo operator:

$$\psi = \psi \;\%\; 2\pi \tag{3.48}$$

The roll and pitch angles are wrapped using a slight modification to the modulo operator

$$\phi = ((\phi + \pi) \;\%\; 2\pi) - \pi \tag{3.49}$$

$$\theta = ((\theta + \pi) \;\%\; 2\pi) - \pi \tag{3.50}$$

**Predict algorithm**

The prediction step has been implemented using two different methods—one using a regular forward Euler integration scheme and one using a two-step Adams-Bashforth integration scheme. The reason for implementing two different strategies is that the rate at which the predict step is called determines the *dt* of the predict step and, in turn, how far the states are propagated forwards in time. Too large of a step will deteriorate the accuracy of the propagation step, hence the need for an alternative method for propagating if the *dt* becomes large. Computing the propagation step at a high rate is desired from an accuracy standpoint, but it is computationally expensive. It is here that the two-step integration scheme finds its place. It is slightly more computationally expensive compared to the forward Euler method but more accurate with larger steps in time, at the trade off that the two previous steps must be stored in memory.

The forwards Euler prediction is described in algorithm 3

**Algorithm 3:** Kalman filter predict step

get $dt$ ;
get $\mathbf{u}_k$ ;
write current state to last state ;
$\hat{\mathbf{x}}_{k-1} \leftarrow \hat{\mathbf{x}}_k$ ;
$\mathbf{P}_{k-1} \leftarrow \mathbf{P}_k$ ;
linearize $\mathbf{F}$, $\mathbf{B}_d$, $\mathbf{W}_q$ ;
predict ;
$\hat{\mathbf{x}}_k = \mathbf{F}\hat{\mathbf{x}}_{k-1} + \mathbf{B}_d\mathbf{u}_k$ ;
$\mathbf{P}_k = \mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^T + \mathbf{W}_q\mathbf{Q}\mathbf{W}_q^T$ ;
$\hat{\mathbf{x}}_k \leftarrow$ wrap_angles($\hat{\mathbf{x}}_k$) ;
**Result:** ($\hat{\mathbf{x}}_k$, $\mathbf{P}_k$)

The two-step prediction is described in algorithm 4:

---

**Algorithm 4:** Kalman filter predict step

---

get $dt$ ;
get current and last control input ;
get $\mathbf{u}_k$ ;
get $\mathbf{u}_{k-1}$ ;
get last two states ;
$\hat{\mathbf{x}}_{k-2} \leftarrow \hat{\mathbf{x}}_{k-1}$ ;
$\mathbf{P}_{k-2} \leftarrow \mathbf{P}_{k-1}$ ;
$\hat{\mathbf{x}}_{k-1} \leftarrow \hat{\mathbf{x}}_k$ ;
$\mathbf{P}_{k-1} \leftarrow \mathbf{P}_k$ ;
linearize $\mathbf{F}$, $\mathbf{B}_d$, $\mathbf{W}_q$ ;
calculate ;
$\hat{\mathbf{x}} = \frac{1}{2}(3\hat{\mathbf{x}}_{k-1} - \hat{\mathbf{x}}_{k-2})$ ;
$\mathbf{u} = \frac{1}{2}(3\mathbf{u}_k - \mathbf{u}_k)$ ;
predict ;
$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_{k-1} + (\mathbf{F} - \mathbf{I})\hat{\mathbf{x}} + \mathbf{B}_d\mathbf{u}$ ;
$\mathbf{P}_k = \mathbf{F}\mathbf{P}_{k-1}\mathbf{F}^T + \mathbf{W}_q\mathbf{Q}\mathbf{W}_q^T$ ;
$\hat{\mathbf{x}}_k \leftarrow$ wrap_angles($\hat{\mathbf{x}}_k$) ;
**Result:** ($\hat{\mathbf{x}}_k$, $\mathbf{P}_k$)

---

### 3.8.2 Measurement equations

For use in the Kalman filter, the measurement equations need to be linearized. This is done following the theory outlined in section 2.3.1

The measurements regarding the attitude and heading of the UAV need some special attention regarding the calculation of the innovation signal.

#### Position measurement

The position measurement equation 3.18 is already a linear equation and takes the form of the following matrix:

$$\mathbf{H}_p = \frac{\partial \mathbf{z}_p}{\partial \mathbf{x}} = \begin{bmatrix} \mathbf{I}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \end{bmatrix} \tag{3.51}$$

**Algorithm 5:** Kalman filter position measurement

get data from pf ;
$\mathbf{p}^n_{nb} \leftarrow \mathbf{p}_{pf}$ ;
$\mathbf{R}_{p,pf} \leftarrow \mathbf{R}_{pf}$ ;
transform measurement ;
$\mathbf{p}^n_{ns} = \mathbf{p}^n_{nb} + \mathbf{C}^n_s(\hat{\mathbf{\Theta}}_{ns})\mathbf{r}^s_{bs}$ ;
$\mathbf{z}_k \leftarrow \mathbf{p}^n_{ns}$ ;
calculate covariance ;
$\mathbf{R}_p = \mathbf{R}_{p,pf} + \mathbf{W}_p \mathbf{P}_{\mathbf{\Theta}} \mathbf{W}_p^T$ ;
get last states ;
$\hat{\mathbf{x}}_k \leftarrow \hat{\mathbf{x}}_{k-1}$ ;
$\mathbf{P}_k \leftarrow \mathbf{P}_{k-1}$ ;
calculate innovation and innovation covariance ;
$\tilde{\mathbf{y}}_k = \mathbf{z}_k - \mathbf{H}_p \hat{\mathbf{x}}_k$ ;
$\mathbf{S} = \mathbf{H}_p \mathbf{P}_k \mathbf{H}_p^T + \mathbf{R}_p$ ;
compute Kalman gain ;
$\mathbf{K}_k = \mathbf{P}_k \mathbf{H}_p \mathbf{S}^{-1}$ ;
correct ;
$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k + \mathbf{K}_k \tilde{\mathbf{y}}_k$ ;
$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_p)\, \mathbf{P}_k$ ;
**Result:** $(\hat{\mathbf{x}}_k, \mathbf{P}_k)$

#### Yaw measurement

The yaw measurement equation 3.21 takes the following matrix form:

$$\mathbf{H}_\psi = \frac{\partial z_\psi}{\partial \mathbf{x}} = \begin{bmatrix} \mathbf{0}_{1\times3} & \mathbf{0}_{1\times3} & \mathbf{0}_{1\times3} & \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} & \mathbf{0}_{1\times3} \end{bmatrix} \tag{3.52}$$

When calculating the innovation for the yaw measurement, the standard linear approach will not work. This is observed when the estimate is close to but slightly larger than 0, and the measurement is close to but slightly smaller than $2\pi$, then the innovation signal will be close to $2\pi$, when in reality, the estimate and the measurement closely agree on the state of the system.

Another method for calculating the innovation is needed to remedy the wrapping issue that takes this constraint into account. The following mini algorithm remedies this wrapping issue:

The measurement must first be checked that it is within the domain of the yaw angle, that is, within $[0,\ 2\pi)$

First calculate the innovation in the *normal* way:

$$\tilde{y}_{\psi,1} = z_\psi - \mathbf{H}_\psi \hat{\mathbf{x}}_k \tag{3.53}$$

Then a second innovation candidate is calculated depending on the sign of the first innovation candidate:

$$\tilde{y}_{\psi,2} = \begin{cases} \tilde{y}_{\psi,1} + 2\pi & , \ \tilde{y}_{\psi,1} < 0 \\ \tilde{y}_{\psi,1} - 2\pi & , \ \tilde{y}_{\psi,1} \geq 0 \end{cases} \tag{3.54}$$

Finally the innovation candidate is selected based in which has the smallest absolute value of the two possible candidates:

$$\tilde{y}_{\psi} = \begin{cases} \tilde{y}_{\psi,1} & , \ |\tilde{y}_{\psi,1}| < |\tilde{y}_{\psi,2}| \\ \tilde{y}_{\psi,2} & , \ |\tilde{y}_{\psi,1}| \geq |\tilde{y}_{\psi,2}| \end{cases} \tag{3.55}$$

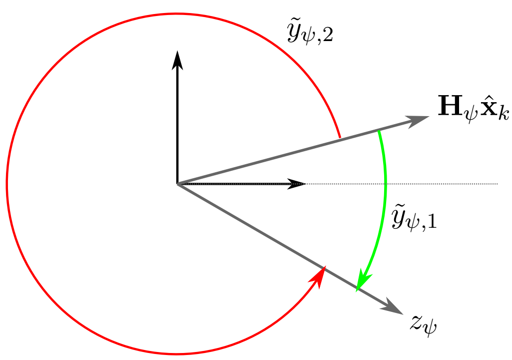



Figure 3.17: Finding the smallest angle when calculating innovation for the yaw

The now calculated innovation signal can be used to correct the state estimate and respects the domain of the yaw angle.

---

**Algorithm 6:** Kalman filter yaw measurement

---

get data from particle filter ;
$\mathbf{z}_k \leftarrow \psi_{ns} \% 2\pi$ ;
$\mathbf{R}_\psi \leftarrow \mathbf{R}_{\psi,pf}$ ;
get last states ;
$\hat{\mathbf{x}}_k \leftarrow \hat{\mathbf{x}}_{k-1}$ ;
$\mathbf{P}_k \leftarrow \mathbf{P}_{k-1}$ ;
calculate innovation ;
$\tilde{\mathbf{y}}_{k,1} = \mathbf{z}_k - \mathbf{H}_\psi \hat{\mathbf{x}}_k$ ;
**if** $\tilde{\mathbf{y}}_{k,1} < 0$ **then**
  $\tilde{\mathbf{y}}_{k,2} = \tilde{\mathbf{y}}_{k,1} + 2\pi$ ;
**else**
  $\tilde{\mathbf{y}}_{k,2} = \tilde{\mathbf{y}}_{k,1} - 2\pi$ ;
**end**
**if** $|\tilde{\mathbf{y}}_{k,1}| < |\tilde{\mathbf{y}}_{k,2}|$ **then**
  $\tilde{\mathbf{y}}_k \leftarrow \tilde{\mathbf{y}}_{k,1}$ ;
**else**
  $\tilde{\mathbf{y}}_k \leftarrow \tilde{\mathbf{y}}_{k,2}$ ;
**end**
compute innovation covariance ;
$\mathbf{S} = \mathbf{H}_\psi \mathbf{P}_k \mathbf{H}_\psi^T + \mathbf{R}_\psi$ ;
compute Kalman gain ;
$\mathbf{K}_k = \mathbf{P}_k \mathbf{H}_\psi \mathbf{S}^{-1}$ ;
correct ;
$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k + \mathbf{K}_k \tilde{\mathbf{y}}_k$ ;
$\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_\psi) \mathbf{P}_k$ ;
**Result:** $(\hat{\mathbf{x}}_k, \mathbf{P}_k)$

---

### Roll and pitch measurements

The roll and pitch measurement equation 3.31 takes the following form:

$$\mathbf{H}_l = \frac{\partial z_l}{\partial \mathbf{x}} = \begin{bmatrix} \mathbf{0}_{2\times3} & \mathbf{0}_{2\times3} & \mathbf{0}_{2\times3} & \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} & \mathbf{0}_{2\times3} \end{bmatrix} \tag{3.56}$$

The calculation of the innovation signal for both roll and pitch takes much the same form as the calculation for the yaw innovation, the only exception is that the attitude measurements must be preconditioned to lie within the interval $[-\pi,\ \pi)$. For most implementations of the $atan2()$ function this is the de facto range of the function.

**Algorithm 7:** Kalman filter roll pitch measurement

get data from accelerometer ;
calculate leveling window ;
**if** $|\mathbf{a}_m| \in [g(1-\epsilon), g(1+\epsilon)]$ **then**
  calculate covariance ;
  $\delta g = ||\mathbf{a}_m| - g|$ ;
  $\mathbf{R}_l \leftarrow \sigma_l(1 + k_l(\delta g + \delta g^2))\mathbf{I}_{2\times2}$ ;
  calculate angles ;
  $\phi_l = atan2\left(-a_{m,y},\ -a_{m,z}\right)$ ;
  $\theta_l = atan2\left(a_{m,x},\ \sqrt{a_{m,y}^2 + a_{m,z}^2}\right)$ ;
  $\mathbf{z}_k \leftarrow \begin{bmatrix} \phi_l \\ \theta_l \end{bmatrix}$ ;
  get last states ;
  $\hat{\mathbf{x}}_k \leftarrow \hat{\mathbf{x}}_{k-1}$ ;
  $\mathbf{P}_k \leftarrow \mathbf{P}_{k-1}$ ;
  calculate innovation ;
  $\tilde{\mathbf{y}}_{k,1} = \mathbf{z}_k - \mathbf{H}_l\hat{\mathbf{x}}_k$ ;
  **if** $\tilde{\mathbf{y}}_{k,1} < 0$ **then**
    $\tilde{\mathbf{y}}_{k,2} = \tilde{\mathbf{y}}_{k,1} + 2\pi$ ;
  **else**
    $\tilde{\mathbf{y}}_{k,2} = \tilde{\mathbf{y}}_{k,1} - 2\pi$ ;
  **end**
  **if** $|\tilde{\mathbf{y}}_{k,1}| < |\tilde{\mathbf{y}}_{k,2}|$ **then**
    $\tilde{\mathbf{y}}_k \leftarrow \tilde{\mathbf{y}}_{k,1}$ ;
  **else**
    $\tilde{\mathbf{y}}_k \leftarrow \tilde{\mathbf{y}}_{k,2}$ ;
  **end**
  compute innovation covariance ;
  $\mathbf{S} = \mathbf{H}_l\mathbf{P}_k\mathbf{H}_l^T + \mathbf{R}_l$ ;
  compute Kalman gain ;
  $\mathbf{K}_k = \mathbf{P}_k\mathbf{H}_l\mathbf{S}^{-1}$ ;
  correct ;
  $\hat{\mathbf{x}}_k = \hat{\mathbf{x}}_k + \mathbf{K}_k\tilde{\mathbf{y}}_k$ ;
  $\mathbf{P}_k = (\mathbf{I} - \mathbf{K}_k\mathbf{H}_l)\,\mathbf{P}_k$ ;
  **Result:** $(\hat{\mathbf{x}}_k,\ \mathbf{P}_k)$
**else**
  do nothing ;
**end**

### 3.8.3 Predict step

In this section the Kalman filter implementation will be discussed. The Kalman filter uses the propagation model discussed in section 3.8.1 and the measurement functions described in section 3.8.2.

#### Prediction strategies

Two different predict strategies were implemented.

The first implementation predicts the next state of the filter every time a new IMU measurement is received. This is a natural choice as the filter is based on the sensor's kinematic model. Furthermore, this approach will ensure that all the measurements received are incorporated into the current estimated state. Here the *dt* between the IMU measurements is used as the *dt* in the predict step. The drawback of this method is that it becomes computationally expensive as the rate of IMU measurements increase. This approach gives rise to the following processing of the IMU data:

**Algorithm 8:** IMU data processing for forward Euler predict

get data from accelerometer ;
subtract g vector ;
$\mathbf{a}_c = \mathbf{a}_m - \mathbf{C}_n^s \mathbf{g}^n$ ;
calculate dt ;
$dt = t_{now} - t_{last}$ ;
$t_{last} \leftarrow t_{now}$ ;
$\mathbf{u}_k \leftarrow \begin{bmatrix} \mathbf{a}_c \\ \omega_m \end{bmatrix}$ ;

The second implemented prediction method tries to remedy the problem arising from high data rate. Here the prediction is calculated at a fixed rate, regardless of the rate of IMU measurements. A two-step prediction algorithm is used to maintain numerical accuracy when the predictions are computed at a lower rate. First, the received IMU measurements are accumulated, and then the accumulated value is used as the control input to the prediction algorithm. This is done to not miss out on valuable IMU measurements. Here the accumulated values are denoted with a $\Delta$, both the IMU measurements and the time between the measurements are accumulated, then when the time comes to run the prediction, the accumulated IMU data is divided by the accumulated time. This can be seen as a weighted average of the accumulated IMU measurements.

**Algorithm 9:** IMU data processing for two-step predict

get data from imu ;
subtract g vector ;
$\mathbf{a}_c = \mathbf{a}_m - \mathbf{C}_n^s \mathbf{g}^n$ ;
calculate dt ;
$dt = t_{now} - t_{last}$ ;
$t_{last} \leftarrow t_{now}$ ;
accumulate imu data ;
$\Delta\mathbf{v} += \mathbf{a}_c \cdot dt$ ;
$\Delta\boldsymbol{\Theta} += \omega_m \cdot dt$ ;
$\Delta t += dt$ ;
when prediction is called, calculate ;
$\mathbf{u}_k \leftarrow \frac{1}{\Delta t} \begin{bmatrix} \Delta\mathbf{v} \\ \Delta\boldsymbol{\Theta} \end{bmatrix}$ ;

### 3.8.4 Holistic filtering strategy

Using the forward Euler prediction strategy the complete filtering algorithm is described in algorithm 10.

**Algorithm 10:** Kalman filter algorithm using forward Euler

```
Initialize filter;
if IMU measurement then
    process IMU data using algorithm 8 ;
    predict using algorithm 3 ;
    level using algorithm 7
end
if Particle filter measurement then
    correct position using algorithm 5 ;
    correct yaw using algorithm 6
end
```

When using the two-step prediction strategy the prediction needs to be called by a timer. The position and measurement functions responsible for incorporating the particle filter estimate are still called as soon as a measurement is available. The two-step implementation is described in algorithm 11.

**Algorithm 11:** Kalman filter algorithm using two-step predict

```
Initialize filter;
if Predict timer elapsed then
    predict using algorithm 4 level using algorithm 7
end
if IMU measurement then
    process IMU data using algorithm 9 ;
end
if Particle filter measurement then
    correct position using algorithm 5 ;
    correct yaw using algorithm 6
end
```

#### Output equations

The retrieval of the filter outputs are only executed once called upon. For some of the states this is as simple as directly outputting a selection of the elements in the filters state vector. For the position, linear velocity and angular rate some calculations are included. These functions are outlined in section 3.7.3

### 3.8.5 Implementation specific functions

#### Offline timer

A watchdog timer is implemented, and watch the time between consecutive position measurements. If the time between measurements is greater than a certain threshold, the position estimation part of the filter is set offline. The length of the timer depends on the quality of the accelerometer used, but a realistic value is in the range of $t \in [5, 30]$ seconds, whereas the position measurements should be updated several times per second under nominal conditions.

Upon setting the position part of the filter in the offline state, the filter is also reset but leaving the roll and pitch at their current estimated values and continuing to estimate them.

**Filter reset**

Two filter reset functions were implemented. On that resets all the filter states and the state uncertainty covariance matrix, and one that resets all the states except for the roll and pitch states.

The function to reset all states is intended to be used when a complete filter reset is called for; this can be during testing when it is faster to reset the filter than to reset the filtering software.

The second reset function that resets all but the roll and pitch states is intended to be used when the filter no longer receives information about its position through the position measurement functions. In this scenario, accelerometer drift will within a short time window render the position estimate useless. However, the roll and pitch angles are primarily estimated using the gyroscope and corrected using the leveling procedure. The attitude estimation is still doable without accelerometer bias estimation. Therefore the roll and pitch angles are left untouched.

## 3.9 Particle filter

The implemented particle filter is a SIR filter, with a simple kinematic motion model for the particles and a likelihood-field sensor model for the stereo camera. The filter does not run global initialization but is rather initialized with a position and a standard deviation in each state.

There are four states included in the filter; the positions in the map frame $\mathbf{p_{mb}^m}$ and the heading (yaw) $\psi_{mb}$ of the drone. The positions are unbounded whereas the heading is wrapped

$$\mathbf{x} = \begin{bmatrix}\mathbf{p}_{mb}^m & \psi_{mb}\end{bmatrix}^T, \qquad \mathbf{p}_{mb}^m \in \mathbb{R}, \qquad \psi_{mb} \in [0,\ 2\pi) \tag{3.57}$$

These states were chosen as they are the most problematic for the Kalman filter to estimate, given the choice of using a depth camera for localization purposes. Further they are the fewest numbers of states required to represent a hypothesise of where the drone is located in 3D space and at what heading the drone is oriented in. Also, for a particle filter in a real-time application, it is desirable to keep the number of states as low as possible.

The particle filter takes in the estimates of angular ($\omega_{bn}^l$) and linear ($\mathbf{v}_{bn}^l$) velocities in the level frame as well as the orientation ($\mathbf{\Theta_{nb}}$) of the drone relative to the NED frame, in addition to their covariance estimated in the Kalman filter. The filter outputs are position and yaw of the drone in the NED frame and their associated estimated variances. In addition random set of particles with a fixed size are drawn form the complete set of particles for visualization in RViz.

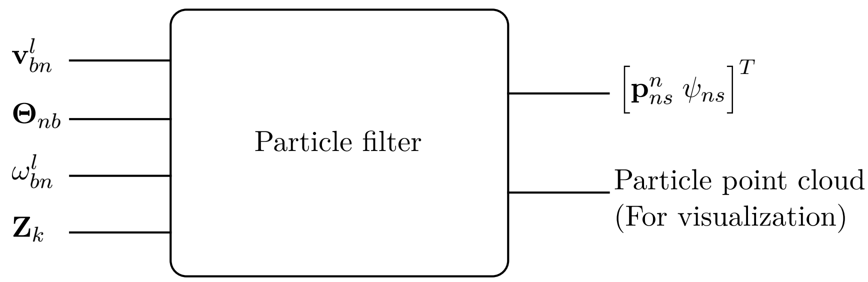



Figure 3.18: Block representation of the Particle filter with inputs and outputs.

The recursive nature of the particle filter is depicted in figure 3.19, where the inputs are used in specific steps in the algorithm.

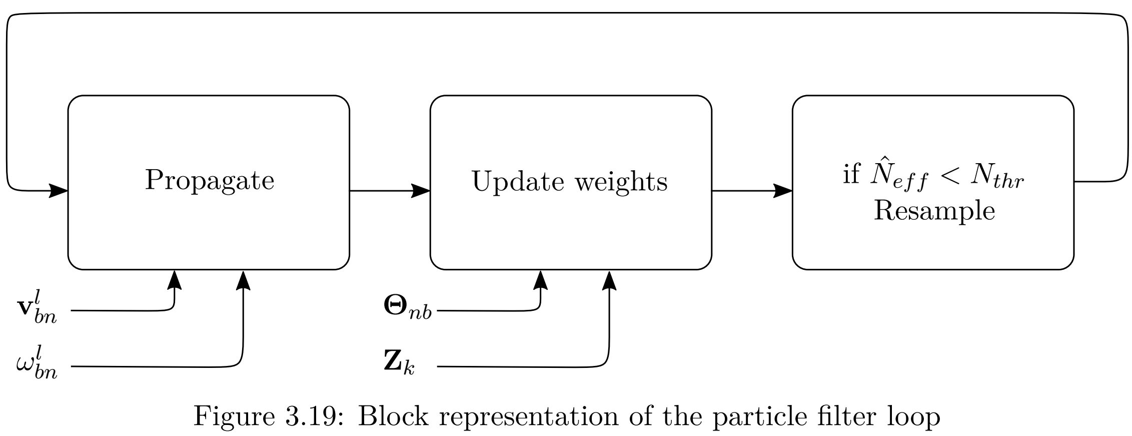



Figure 3.19: Block representation of the particle filter loop

### 3.9.1 Kinematic motion model

The kinematic motion model is used to propagate the samples in the particle filter. The motion model relies on knowledge about the velocity of the drone, as these are not included as states in the filter they must be given as input.

The particles are propagated from the prior set of particles, in practice this means that each particle is moved in the state space based on a propagation model. For each particle $i$ this is done as outlined in equation 2.101:

$$\mathbf{x}_k^i = \mathbf{F}\mathbf{x}_{k-1}^i + \mathbf{B}(\mathbf{u}_k^i + \mathbf{w}_k^i) \tag{3.58}$$

Where $\mathbf{u}_k^i$ is the linear and angular velocities $[\mathbf{v}_k, \dot{\psi}_k]^T$ for particle $i$, where the velocities are drawn from a Gaussian distribution centered around the velocity inputs from the Kalman filter; using the variance from the Kalman filter with some extra noise to ensure a good spread in the particle cloud.

$$\mathbf{u}_k^i = [\mathbf{v}_{nb}^l \dot{\psi}_{nb}^l]^T, \qquad \mathbf{w}_k^i \sim \mathcal{N}(0, \sigma_{\mathbf{v}}), \qquad \sigma_{\mathbf{v}} = \begin{bmatrix} \sigma_{\dot{x}} & \sigma_{\dot{y}} & \sigma_{\dot{z}} & \sigma_{\dot{\psi}} \end{bmatrix}^T \tag{3.59}$$

As the velocities are given as inputs and not included as states in the particle filter, $\mathbf{F}$ becomes a $4x4$ identity matrix. The $\mathbf{B}$ matrix is responsible for rotating the given velocities into the frame of each particle and is defined as

$$\mathbf{B} = \begin{bmatrix} \mathbf{C}_z(\psi^i)dt & \mathbf{0}_{3x1} \\ \mathbf{0}_{1x3} & dt \end{bmatrix}, \qquad \psi^i = \psi_{k-1}^i + \frac{\dot{\psi}_k dt}{2} \tag{3.60}$$

Where $\psi_{k-1}^i$ is the yaw of particle $i$ before propagation.

The propagation-model was tested for different noise parameters $\sigma_{\mathbf{v}}$ shown in figure 3.20, where the orange dot is 100 particles initialized to $[x,\ y] = [0,\ 0]$, and the blue is the spread after 10 seconds. The particles were propagated with a "true" velocity of $v_x = 0.1\ [m/s]$ with added noise $\sigma_{\mathbf{v}}$, showing the final shapes similar to what is seen in figure 2.21.

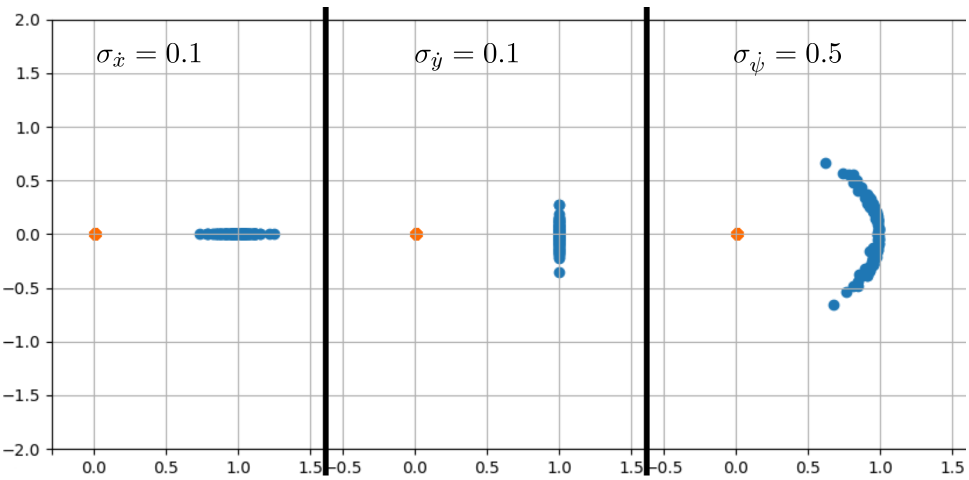



Figure 3.20: The particle propagation-model tested with different noise parameters, moving 100 particles

### 3.9.2 Weight update

The weight update equation is shown in equation 2.100, and finds the new weight of each particle based on the scan inserted into the likelihood field for each particle in the filter.

$$w_k^i = w_{k-1}^i \cdot p(\mathbf{Z}_k | x_k^i, \mathcal{M}) \tag{3.61}$$

The particle weights $w_k^i$ are updated based on the point cloud $\mathbf{Z}_k$ from the stereo camera. First, the point cloud is downsampled, picking random points from the cloud and checking that all points are within the set max range for the sensor. The points that fail this check are removed from the cloud. It is not given that the drone is level with the map frame at the time of capturing a point cloud, to compensate for this roll and pitch estimates from the Kalman filter is used to level the downsampled point cloud.

The downsampled and leveled point cloud is projected into the likelihood filed at the location and heading of each particle in the filter. For each particle the likelihood of each end point in the point cloud is multiplied together. This product of likelihoods is used to update the weight of the particle by multiplying it with the previous particle weight.

$$w_k^i = w_{k-1}^i \cdot \prod_{m=1}^{M} \left( z_{hit} \cdot p_{hit}(\mathbf{z}_k^m | \mathbf{x}_k^i, \mathcal{M}) + \frac{z_{rand}}{z_{max}} \right) \tag{3.62}$$

Where $p_{hit}(\mathbf{z}_k^m | \mathbf{x}_k^i, \mathcal{M})$ is the likelihood of a point $\mathbf{z}_k^m$ being at a detectable object in the map $\mathcal{M}$, given the particle state $\mathbf{x}_k^i$. This likelihood is "read out" from the pre-computed likelihood filed at the voxel located at the points location $\mathbf{z}_k^m$

After the weights are updated for all particles, the new weights are normalized according to equation 2.103. This step must be included each time the weight-update algorithm is run as it is guaranteed[7] to produce a weight distribution that is not normalized. After normalization, an estimate of the effective number of samples $\hat{N}_{eff}$ is calculated following equation 2.98.

A procedure for finding the mixing-constants $z_{hit}$, $z_{rand}$ and $z_{max}$ is described in [36], it is also mentioned that these values can be "eyeballed".

[7]Although the likelihood-field is created assuming normalized Gaussian distributions when calculating $p_{hit}$ for each cell, inserting points into the field and taking the product can easily produce weights greater than one for each particle; depending on the used $\sigma_{map}$ and amount of points sampled from the point cloud

### 3.9.3 Resampling

The resampling step is run whenever the effective number of samples $\hat{N}_{eff}$ becomes too low, and implements low variance resampling as shown in algorithm 2.

The threshold for resampling is set to a parameter, to allow tuning for how often the resampling-step is executed. Running the resampling step too seldom will waste a lot of compute-time on updating particles with almost zero probability, whilst running it too often might make the filter too focused on specific areas. Therefore a balance must be struck.

### 3.9.4 Getting a solution from the filter

Getting the best estimate of the state given the particles is not a straightforward task. Sometimes it is good enough to just choose the highest weighted particle, and accept that as the best estimate. This could potentially be slightly misleading if the resampling step is run often, as the highest weighted particle might bounce around with some lucky particles getting a great hit from noisy sensor data.

It was chosen to implement a histogram smoothing algorithm to get an estimate from the particle filter. The algorithm represents the weighted particle distribution as a histogram for each of the states. A Gaussian kernel is then used to smooth the histogram. The Kernel smoothing achieves two things; flatten peaks, acting as a outlier-rejection algorithm, and smoothing the histogram, making the most likely state more prominent.

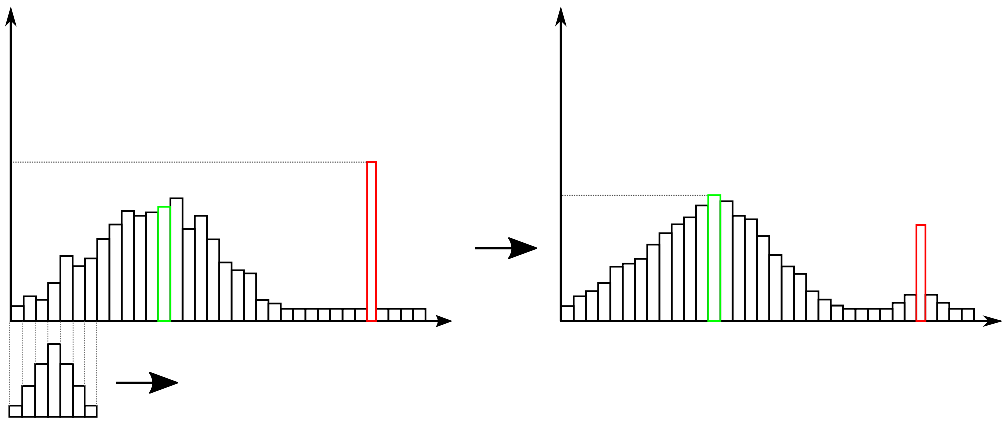


Figure 3.21: The main idea of the histogram smoothing algorithm, the red peak gets flattened.

After smoothing, the center of the highest peaking histogram bin is chosen as the best state estimates.

The get an accompanying estimate of the uncertainty of the estimated states the mean square error (MSE) of the particle set is calculated. The MSE is calculated based on the estimated state as its origin, this MSE is used as a quasi-variance for the state estimate. This is done as opposed to using the mean weighted particle position to calculate the true variance of the particle set, as the MSE centred at a position different from the mean will always be larger then the variance calculated about the mean.

This estimate and it's variance is then output to the Kalman filter.

### 3.9.5 Pseudocode

The full implemented SIR particle filter algorithm in pseudocode, the histogram smoothing algorithm is omitted as this is not strictly a part of the loop, but runs as a side-process at

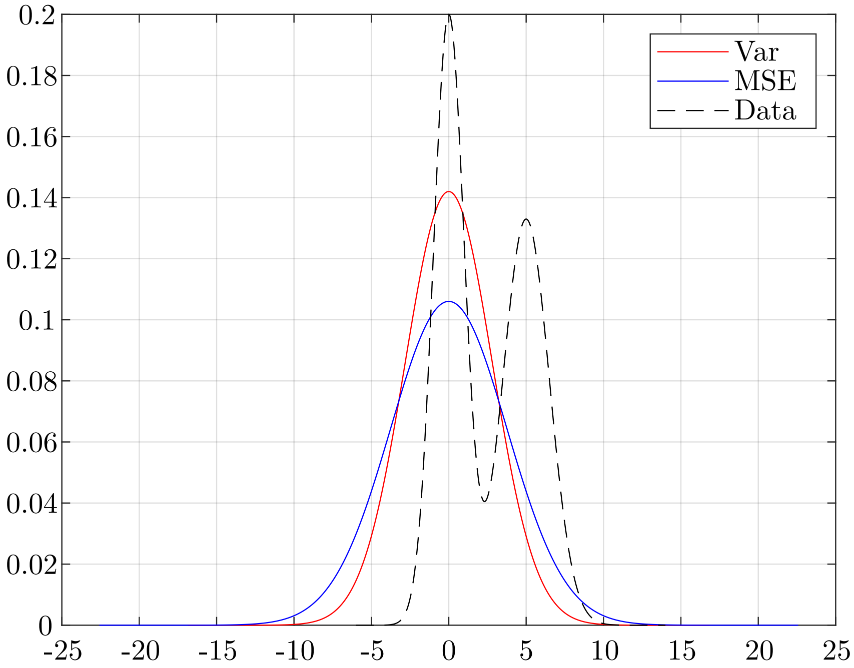



Figure 3.22: Two gaussian distributions about the peak value of a bimodal dataset, red: Var = Var of dataset, Blue: Var = MSE from peak

a fixed rate using the newest set of particles $\{\mathbf{x}_k^n, w_k^n\}_{n=1}^{N_s}$ from the filter.

---

**Algorithm 12:** The implemented SIR particle filter in pseudocode

---

**Input:** $\{\mathbf{x}_{k-1}^n, w_{k-1}^n\}_{n=1}^{N_s}$, $\mathbf{v}_{nb}^l$, $\mathbf{w}_{nb}^l$, $\mathbf{\Theta}_{nb}$, $\mathbf{Z}_k$

```
begin
    for n = 1...N_s do
        w_k^n ~ N(0, σ_v);
        x_k^n = x_{k-1}^n + B(u_k^n + w_k^n) ;          // Propagate according to (3.58)
    end
    {z_k^{*m}}_{m=1}^M = PC_Downsample(Z_k) ;          // Downsample pointcloud
    {z_k^m}_{m=1}^M = PC_Level({z_k^{*m}}_{m=1}^M, Θ_nb) ;   // Level and rotate to map frame
    for n = 1...N_s do
        w_k^n = w_{k-1}^n · ∏_{m=1}^M ( z_hit * p_hit(z_k^m | x_k^n, M) + z_rand / z_max ) ;   // Update weights
    end
    Normalize_Weights() ;                               // Normalize weights
    if N̂_eff < N_thr then
        Resample() ;                                    // Resample if N̂_eff below threshold
    end
end
return {x_k^n, w_k^n}_{n=1}^{N_s}
```

In mathematical notation:

- $\mathbf{w}_k^n \sim \mathcal{N}(0, \sigma_{\mathbf{v}})$;
- $\mathbf{x}_k^n = \mathbf{x}_{k-1}^n + \mathbf{B}(\mathbf{u}_k^n + \mathbf{w}_k^n)$ — // Propagate according to (3.58)
- $\{\mathbf{z}_k^{*m}\}_{m=1}^M = \text{PC\_Downsample}(\mathbf{Z}_k)$ — // Downsample pointcloud
- $\{\mathbf{z}_k^m\}_{m=1}^M = \text{PC\_Level}(\{\mathbf{z}_k^{*m}\}_{m=1}^M, \mathbf{\Theta_{nb}})$ — // Level and rotate to map frame
- $w_k^n = w_{k-1}^n \cdot \prod_{m=1}^M \left( z_{hit} * p_{hit}(\mathbf{z}_k^m | \mathbf{x}_k^n, \mathcal{M}) + \frac{z_{rand}}{z_{max}} \right)$ — // Update weights
- if $\hat{N}_{eff} < N_{thr}$ then Resample() — // Resample if $\hat{N}_{eff}$ below threshold

**return** $\{\mathbf{x}_k^n, w_k^n\}_{n=1}^{N_s}$

## 3.10 Particle filter Implementation

This section will describe in more detail how each step of the particle filter is implemented, some different configurations for each step as well as show some measures of execution time on the chosen hardware platform.

### 3.10.1 Propagation

The propagation step will, for each particle, pick a random propagation velocity drawn from a Gaussian distribution with a mean $\mathbf{v}^l_{nb}$ recived from the Kalman filter.

The mean value of the Gaussian distribution that the propagation-velocities will be drawn from will be denoted $\mathbf{v}_\mu$.

Three different methods for getting the velocity has been implemented and tested, where the simplest implementation uses the most resent estimate from the Kalman filter as the mean for the Gaussian.

$$\mathbf{v}_\mu = [\mathbf{v}^l_{nb}\ \psi^l_{nb}]^T_k \tag{3.63}$$

The second method uses a two-step Adams-Bashforth method for integration[8], as shown in equation 2.55. This method requires that the last velocity estimate from the Kalman filter is retained in memory, but at the cost of slightly larger memory usages an increase in numerical accuracy is gained[9].

$$\mathbf{v}_\mu = 1.5 \cdot [\mathbf{v}^l_{nb}\ \psi^l_{nb}]^T_k - 0.5 \cdot [\mathbf{v}^l_{nb}\ \psi^l_{nb}]^T_{k-1} \tag{3.64}$$

The variance from the Kalman filter is calculated using the variance propagation equation from section 2.2.3.

The third method continuously integrates velocity estimates over time as new estimates are available from the Kalman filter, calculating the drones traversed distance based on the velocity estimates from the Kalman filter. The method numerically integrates the velocities using the time $d\tau$ between each subsequent incoming velocity estimate. The integral is reset each time the propagation step is executed.

$$\Delta\mathbf{x} = \sum [\mathbf{v}^l_{nb}\ \psi^l_{nb}]^T\, d\tau \tag{3.65}$$

To keep the interface the same, the distance $\Delta\mathbf{x}$ is divided by $dt$ used in the propagation step before being used in the motion model.

$$\mathbf{v}_\mu = \frac{\Delta\mathbf{x}}{dt} \tag{3.66}$$

For this method to be an improvement over the traditional forward Euler method, the velocity estimates from the Kalman filter must arrive at a higher rate compared to the other implementations.

When testing, it became evident that the two ladder methods obtaining an improved velocity mean $\mathbf{v}_{mu}$ did not impact the estimated states from the particle filter. The lack of improvement is likely due to the constant variance added to the velocity variance to disperse

[8] As $\mathbf{F}$ in the case for the particle filter is equal to $\mathbf{I}$, the first product is not included. Making the update equation $\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{B}(1.5u_k - 0.5u_{k-1})$

[9] The velocity is not stored per particle, but only the last mean value estimated from the Kalman filter

the particle cloud. Additionally, the third method - requiring a higher rate of Kalman filter velocity estimate outputs substantially increased the processor-usage when deployed to hardware. For these reasons, the first method described was used for the mean velocity $\mathbf{v}_\mu$ of propagation when the filter was deployed to hardware.

**Propagation standard deviation**

The standard deviation $\sigma_\mathbf{v}$ used for drawing propagation-velocity is composed of three components.

$$\sigma_\mathbf{v} = \sigma_{KF} + \sigma_{const} + \sigma_k, \qquad \sigma_\mathbf{v} = \begin{bmatrix}\sigma_{\dot{x}} & \sigma_{\dot{y}} & \sigma_{\dot{z}} & \sigma_{\dot{\psi}}\end{bmatrix}^T \tag{3.67}$$

The first component $\sigma_{KF}$ is the standard deviation of the estimates in the Kalman filter. This standard deviation is calculated based on the state uncertainty matrix in the Kalman filter. The element ($\sigma_{const}$) is a constant standard deviation added in each direction and is a tunable parameter in the filter, it is introduced to disperse the particles in space. $\sigma_k$ will be zero during typical operation; this value is intended to put the filter into "search mode" if no new velocity estimates are received for set amount of time, this mode will be described in the next part.

**Search mode**

When no new velocity estimate is received, knowledge about the motion of the drone is lost. This means that the particle filter will no longer have a good guess of how to propagate the particles. On the other hand, the last received velocity will be reasonably accurate for some time, as an object in motion tends to stay in motion[10]. Therefore the last known estimate of the velocity and standard deviation from the Kalman filter is decayed each time the propagate-step is executed without a new velocity estimate.

$$\mathbf{v}_\mu = \mathbf{v}_\mu(1 - k_{wd}) \qquad \sigma_{KF} = \sigma_{KF}(1 - k_{wd}) \tag{3.68}$$

Where $k_{wd}$ is a tunable parameter in the interval $[0, 1]$. At each execution of the propagation-step, a counter $i_{wd}$ is incremented, this counter is reset when a new velocity estimate is received. The counter-value is used to calculate the added standard deviation $\sigma_k$, which increase until it reaches an upper bound.

$$\sigma_k = (i_{wd} - 1) * k_{wd} \qquad \sigma_k \in [0, \sigma_{k,max}] \tag{3.69}$$

This mode is only intended as an emergency approach for localization in case the Kalman filter shuts down unexpectedly, and will propagate the particles in all directions randomly, whilst still executing the measurement and re-sampling steps.

[10] "Vir meus!"-Isaac.N

### 3.10.2 Weight update

The weight update step takes uses the pointcloud from the depth camera, the depth information is input as a matrix containing $N_p$ number of measurement vectors (points) resolved in the camera frame.

$$\mathbf{Z}_k = \begin{bmatrix} x_k^{(1)} & x_k^{(2)} & x_k^{(3)} & ... & x_k^{(N_p)} \\ y_k^{(1)} & y_k^{(2)} & y_k^{(3)} & ... & y_k^{(N_p)} \\ z_k^{(1)} & z_k^{(2)} & z_k^{(3)} & ... & z_k^{(N_p)} \end{bmatrix}^C \tag{3.70}$$

**Downsampling the point cloud**

The point cloud is downsampled using one of three methods, chosen by setting the relevant parameters. The first method samples $M$ points from the point cloud using evenly spaced samples from the point cloud, creating the vector $\mathbf{m}$ of all indexes to sample[11].

$$\mathbf{m} = \text{linspace}(1, N_p, M) \tag{3.71}$$

Using linspace to pick the points for the down sampling can be risk-ridden if the point cloud is an ordered set. That is if the first index is say the top left of the image, then increasing in when moving to the right, and so on for the next rows. If not careful when selecting a spacing, the selected points might be on a vertical line in the image[12], or some diagonal. his would virtually guarantee that the points are not independent, as they could be picked along a wall or in a line on the floor and would give precious little information about the environment.

The second method picks $M$ random points from a uniform distribution, drawing $M$ random integers from $\mathcal{U}(1, N_p)$ *before* sampling from the point cloud. The random numbers are then checked for uniqueness in $\mathbf{m}$, deleting duplicates to avoid sampling the same point twice times.

$$m \sim \mathcal{U}(1, N_p) \tag{3.72}$$

Common for the two methods mentioned is that all the numbers in $\mathbf{m}$ are picked *before* drawing a single point from the point cloud. After the points are drawn, the downsampling algorithm checks how far away from the sensor the points are located.

$$d_k^{(m)} = \sqrt{(x_k^{(m)})^2 + (y_k^{(m)})^2 + (z_k^{(m)})^2} \tag{3.73}$$

If the distance $d$ is above a set threshold, the point $m$ is removed from the downsampled point cloud resulting in a point cloud with fewer than $M$ points.

The third method picks the points at random, just like the second method. However, here the points are picked and validated one by one *before* the point is added to the list. First, a random integer $m$ is drawn from the range $[1, N_p]$, and the distance to the point at that index is checked using equation 3.73. Then, if the point is within the threshold, and the index-vector $\mathbf{m}$ does not already contain the index $m$, $m$ is added to $\mathbf{m}$ and the process

[11] Represented as "code" - in practice numpy.linspace() was used for this operation, $\text{linspace}(1, N_p, M)$ creates a vector of $M$ values evenly spaced between 1 and $n_p$

[12] Ex: if the image resolution is 20x20, and the spacing between points is 20, then the points will lie on a vertical line in the depth image

repeats. This loop runs until $M$ points are picked or a maximum number of points have been tested.

The three methods shown is detailed in pseudocode, and can be seen in algorithm 13

**Algorithm 13:** The three different methods of downsampling the pointcloud

```
Input: Z_k, M, d_thr, M_thr
begin
    if Method 1 then
        m = linspace(1, N_p, M) ;                       // Pick M Evenly spaced integers
        for m = 1...M do
            Check d_k^(m) as in eq 3.73;
            if d_k^(m) > d_thr then
                Delete m from m;
            end
        end
        z_k = Z_k^(m) ;                                 // Index m from Z_k
    end
    else if Method 2 then
        m ~ U(1, N_p, M) ;                              // Draw M random integers
        m = unique(m);                                  // Ensure only unique indexes in m
        M' = length(m);
        for m = 1...M' do
            Check d_k^(m) as in eq 3.73;
            if d_k^(m) > d_thr then
                Delete m from m;
            end
        end
        z_k = Z_k^(m) ;                                 // Index m from Z_k
    end
    else if Method 3 then
        while m < M_thr AND length(m) < M do
            m ~ U(1, N_p) ;                             // Draw a random integer
            Check d_k^(m) as in eq 3.73;
            if d_k^(m) < d_thr AND m ∉ m then
                Add m to m ;    // Add m to m if within threshold and unique
            end
        end
        z_k = Z_k^(m) ;                                 // Index m from Z_k
    end

    M* = length(m)
end
return {z_k^m}_{m=1}^{M*} ;                             // Return downsampled point cloud
```

**Leveling the point cloud**

The downsampled point cloud is still resolved in the camera-frame, which means it needs to be transformed into the frame of each particle. This is done in multiple steps to avoid unnecessary transformations having to be done for each particle.

The first step is transforming the point cloud from the camera frame into the level body frame, which has it's Z-axis straight down and X-Y plane parallel with that of the map, regardless of the drones attitude. The transformation is two-fold, where one is based on the geometry of the drone, and is static; whereas the other one is based on the estimated pitch and roll angles from the Kalman filter, and is dynamic. With the camera rigidly mounted to the airframe, the first set of rotations and translations is given by the design of the airframe and mounting brackets. This transformation brings the pointcloud into the body-frame of the drone.

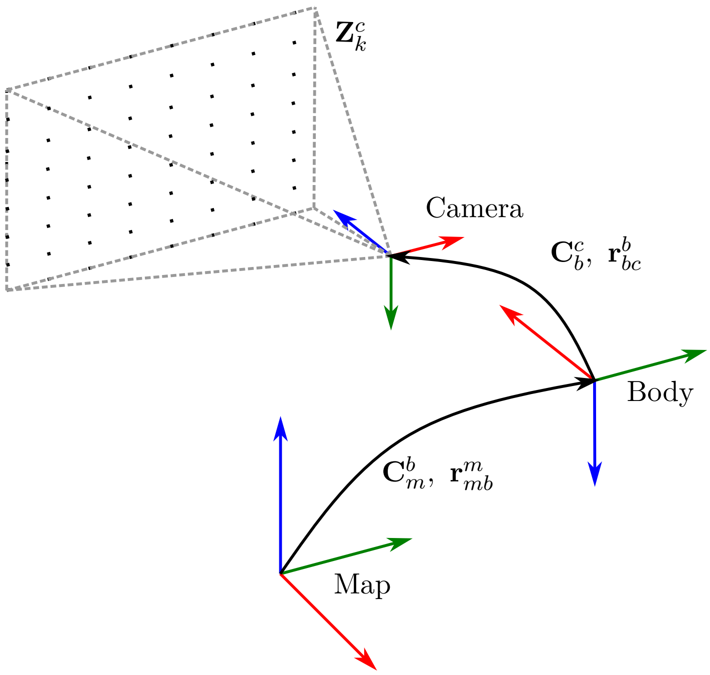



Figure 3.23: The map, body and camera frame

The transformation-process of the pointcloud will be described using the notation$\mathbf{Z}_k^x$, where $\mathbf{Z}$ is the set of points constituting the point cloud at time $k$, resolved in frame $x$.

The first step is to rotate and translate the point cloud into the drone body frame ($b$). This is a static transforms as the camera is rigidly fastened to the airframe.

$$\mathbf{Z}_k^b = \mathbf{r}_{bc}^b + \mathbf{C}_c^b \mathbf{Z}_k^c \tag{3.74}$$

The point cloud is then rotated into the level body frame ($l$) using the roll and pitch estimates from the Kalman filter

$$\mathbf{Z}_k^l = \mathbf{C}_b^l(\mathbf{\Theta}_{nb})\mathbf{Z}_k^b \tag{3.75}$$

One final rotation is done, which is rotating 180 degrees about the X-axis. This transforms the pointcloud into a "body centered map frame", with X forward and Z up.

$$\mathbf{Z}_k^{m*} = \mathbf{C}_x(\pi)\mathbf{Z}_k^l \tag{3.76}$$

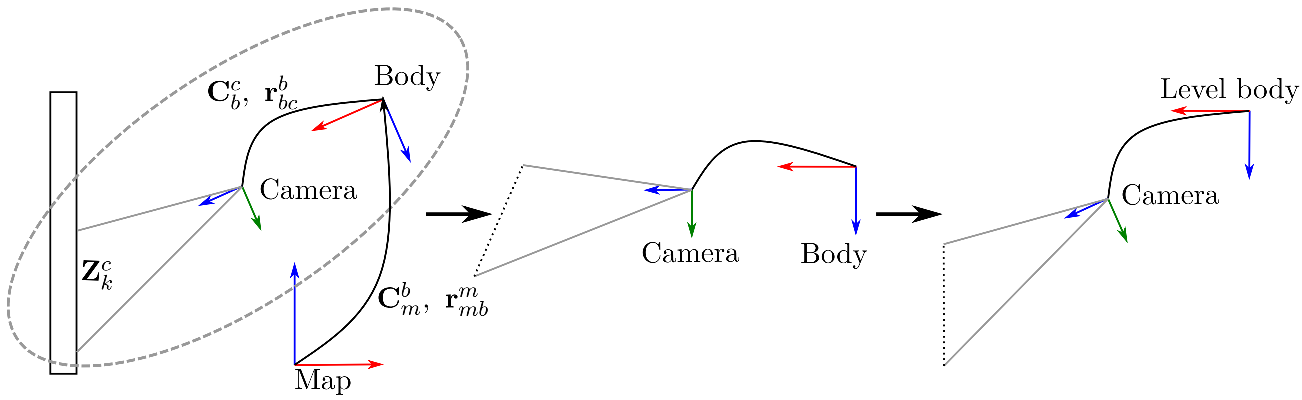



Figure 3.24: The point cloud must be leveled before insertion into the map

All the transformations above are preformed as a pre-processing step before the point cloud is handed of to the particle filter, that is to say, only preformed once per point cloud measurement from the stereo camera.

#### Updating the weight for each particle

To find the weight for each particle, the point cloud $\mathbf{Z}_k^{m*}$ must be inserted into the map. This is done by transforming the now leveled and downsampled point cloud into the frame of each particle in the following manner:

$$\mathbf{Z}_k^p = \mathbf{r}_{mp}^m + \mathbf{C}_z(\psi^p)\mathbf{Z}_k^{m*} \tag{3.77}$$

Where $\mathbf{r}_{mp}^m$ is the position and $\psi^p$ is the yaw of particle $p$. $\mathbf{Z}_k^p$ contain the $x$ $y$ and $z$ coordinates of the points from the scan in map frame, projected out from particle $p$.

The next step is to calculate what voxels the individual measurement points lie within. This is done by dividing the points $x$, $y$ and $z$ coordinates with the maps resolution(voxel size), and then rounding off to find the voxel index. Once this index is found it is impotent to check that the points index is a valid index in the map, as trying to index an invalid point is nonsensical. With the method developed for this project indexing an invalid voxel address will result in a segmentation fault, as the program will then try to read a part of the computers memory that it dose not have access to.

If a point is calculated to have an index outside the range of valid indexes for the map, it is given a probability equal to that of a random reading ($z_{rand}/z_{max}$). This is done as our test environments contain no unm odelled objects or areas outside of map bounds, whereas [36] states that a probability of $1/z_{max}$ could serve as a crude way to incorporate readings outside modeled space for a real environment containing unmapped regions.

If the likelihoods stored in the map is encoded in UInt8's laying in the interval $\in [0, 255]$, they will need to be converted back to decimal numbers after being read from the map. The map metadata (section 3.5.1) contain the maximum value for the Gaussian distribution used for map generation ($\sigma_{max}$), this value is used to decode the likelihood $p_{hit}^*$ from the map in the following manner:

$$p_{hit} = \frac{\sigma_{max}}{255} * p_{hit}^* \tag{3.78}$$

Care must be taken when selecting what data type to use for the particles weight. If the value of $z_{rand}/z_{max}$ is small and a large number of points is sampled from the point cloud, the resulting product of likelihoods can become smaller then what a 32 bit float can represent($1.175494351E-38$), causing underflow to zero. Therefor using 64 bit floats is advisable. An alternative solution to this problem, proposed in [14] shows a different approach to combine the likelihoods of measurement points from the likelihood field, proposing:

$$p(\mathbf{Z_k}|\mathbf{x}_k^i, \mathcal{M}) = \frac{(\sum_{n=1}^{N_p} p(\mathbf{z}_k^n|\mathbf{x}_k^i, \mathcal{M}))^2}{N_p} \tag{3.79}$$

And then updating the weight $w_k^i$ according to equation 3.61. This is an approximation. Intended to limit how small of a weight a particle can be given, compared to multiplying the likelihoods of each point. This weighting method was implemented as a configuration in the filter.

**Algorithm 14:** Update step

```
Input: {x_k^i, w_{k-1}^i}_{i=1}^{N_s}, Z_k^{m*}, M
begin
    for i = 1...N_s do
        q = 1 ;                                   // Initialize update-weight to 1
        Get r_mp^m, ψ^p from x_k^i;
        Z_k^p = r_mp^m + C_z(ψ^p) Z_k^{m*};
        for All points z_k^p in Z_k^p do
            if z_k^p ∈ M then
                find idx for z_k^p ;              // Find index [x, y, z] for point
                p = z_hit · M(idx) + z_rand/z_max;
            else
                p = z_rand/z_max;
            end
            q = combine_hits() ;                  // Find p(Z_k|x_k^i, M) using 3.61 or 3.79
        end
        w_k^{i*} = w_{k-1}^i · q;
    end
    w_k^i =Normalize_Weights(w_k^{i*})
end
return {x_k^i, w_k^i}_{i=1}^{N_s} ;               // Return particles with updated weights
```

### 3.10.3 Resample

The resampling step is implemented in code more or less exactly like in algorithm 2, with an added check to avoid the possibility of segmentation faults. In the final implementation the resampling step was run at every filter iteration.

**Algorithm 15:** Low Variance Resampling implementation

```
Input: {x_k^(np), w_k^(np)}_{np=1}^{Ns} ;                    // Set of particles
begin
    r = U(0, Ns^-1) ;
    W = w^(1) ;
    i = 1;
    for n = 1...Ns do
        u = r + (n - 1)/Ns;
        while u > W do
            i = i + 1;
            if i > Ns then
                W = u ;              // Avoid indexing outside of particle set
                i = Ns ;
            end
            W = W + w_k^(i);
        end
        x_k^(n)* = x_k^(i);
        w_k^(n)* = Ns^-1;
    end
end
return {x_k^(np)*, w_k^(np)*}_{np=1}^{Ns} ;              // Resampled set of particles
```

### 3.10.4 Histogram smoothing

The histogram smoothing algorithm first create a histogram for each state in the filter using the current state of every particle and its associated weight. The histogram is setup to have the same width for every bin.

As the histograms are created from the particles, they will have a lower and upper limit equal to the position of the extremal particles, as opposed to creating a histogram that covers the possible position state space. This will result in a more memory space effective histogram. Meaning that when kernel-smoothing the resulting histograms need to be padded at each end. For the positions this results in padding the histograms at each end with an amount of zeros equal to half the length of the kernel. The heading, however, is wrapped $[0, 2\pi)$ - which means that if the current solution is around 0, it must be padded with values from the opposite end of the histogram.

Because the heading is wrapped, calculating the mean square error has to be done using the smallest angle just like in the Kalman filter Yaw measurement innovation (section 3.8.2).

The mean square error around the solution is calculated much like variance, where instead of the mean of the distribution, the peaks of the smoothed histograms ($\hat{x}$) are used.

$$MSE_{\hat{x}} = \frac{V_1}{V_1^2 - V_2} \sum_{i=1}^{N_s} (x_i - \hat{x})^2 \cdot w_i \tag{3.80}$$

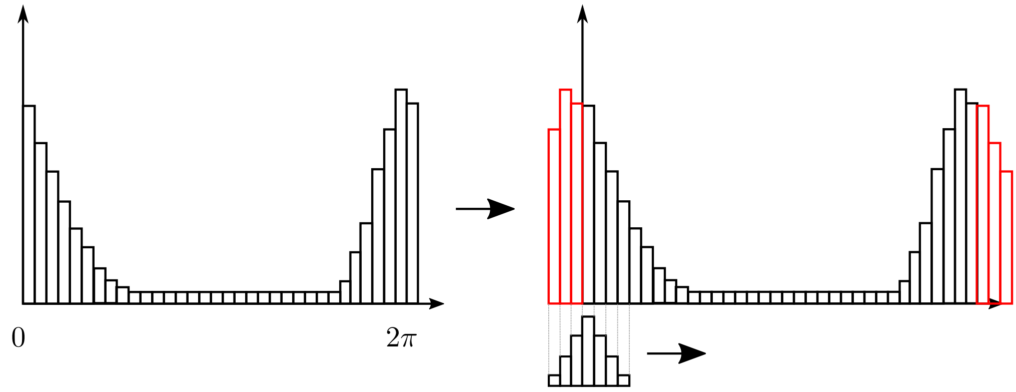



Figure 3.25: Histogram of wrapped variable padded with values from the other end

Where $V_1$, $V_2$ are the same as in 2.2.2, $w_i$ and $x_i$ is the weight and state of particle $i$.

The particle filter estimates are then output to the Kalman filter as position and yaw measurements, with the mean square errors populating the diagonal of the measurement covariance matrix.

### 3.10.5 Execution time on hardware

A simple benchmarking script was created in order to log execution-times of the different parts of the particle filter algorithm. In this script the different steps of the filter is executed using dummy-data given the same format that the filter is going to receive during operation, and the execution times are logged.

The propagation step moves the particles in the state space, and the update step first updates the weights based on the point cloud, before normalizing and resampling. Which means that the time it takes for each iteration of the filter will be the sum of the run-times from the two plots.

Running the benchmarking script on the Jetson TX2 with 1000 particles in the filter and picking 50 points from the dummy-pointcloud resulted in the following execution-times:

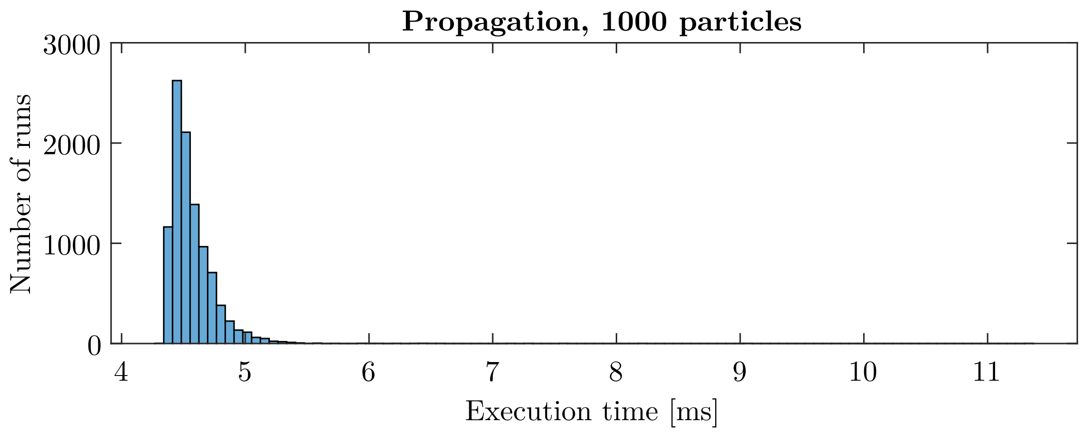



Figure 3.26: Histogram plot of exectuion times for 10000 runs of the propagation-steps

Assuming worst case from plot 3.26 and 3.27, which are $\sim 11$[ms] from the propagation step and $\sim 24$ from the update and resample step; one filter iteration takes approximately:

$$\Delta t = 11\ [ms] + 24\ [ms] = 35\ [ms] \tag{3.81}$$

Leaving 65 $[ms]$ of "overhead" for ROS2 and histogram smoothing, assuming a execution-rate

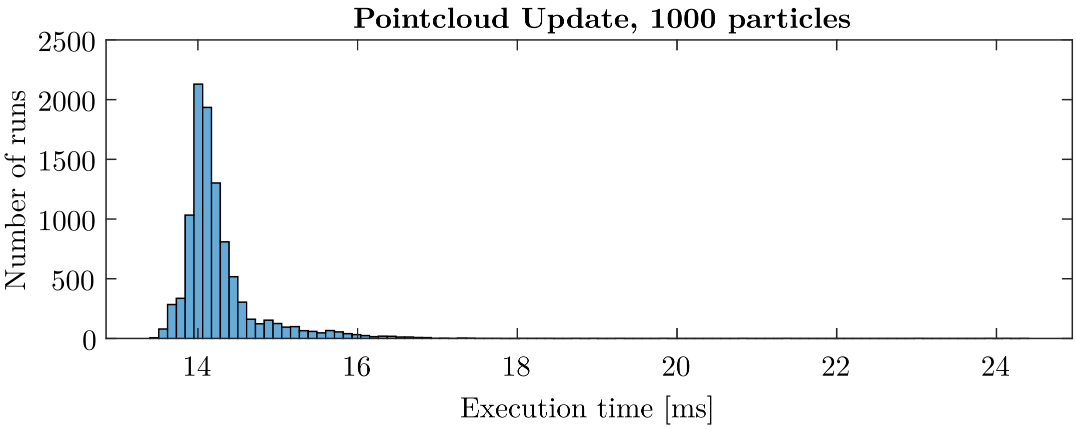



Figure 3.27: Histogram plot of exectuion times for 10000 runs of pointcloud update, normalize and resample

of 10 $[Hz]$ for the filter. The following plots show the execution times for the 99 percentile of runs.

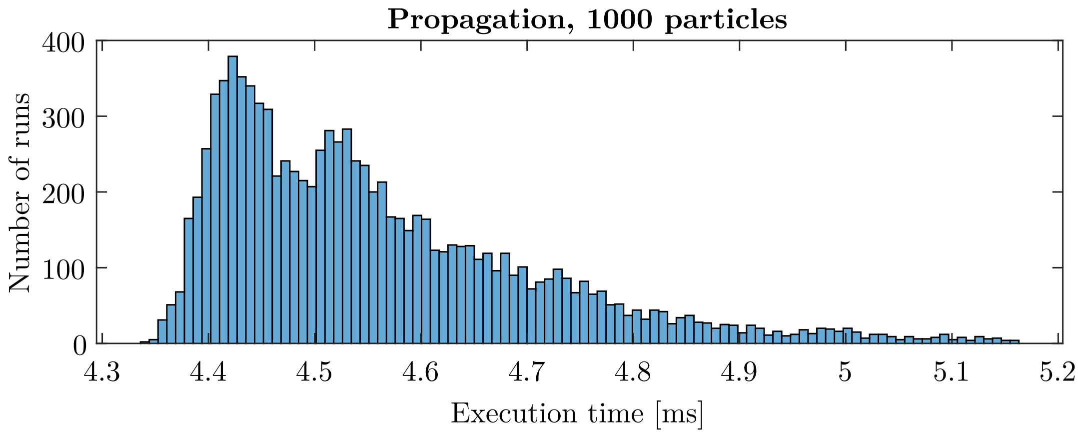



Figure 3.28: Histogram plot of exectuion times for the top 99% of the runs, propagation

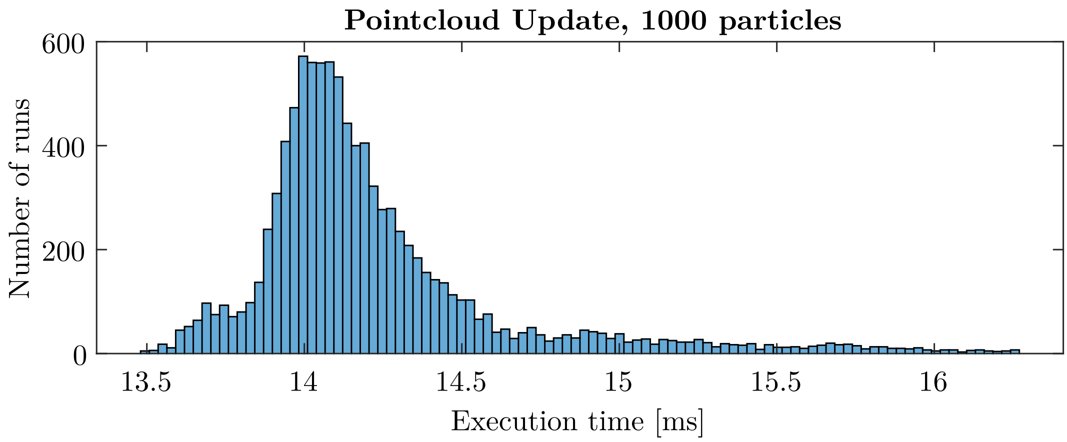



Figure 3.29: Histogram plot of exectuion times for the top 99% of the runs, pointcloud update, normalize and resample

## 3.11 Software implementation

### 3.11.1 JiT compilation

*Just-in-Time* (JiT) compilation is a way of executing a computer program where the written code is compiled during program execution. This differs from *Ahead of Time* (AoT) compilation, where the program is compiled into an executable file which can then be run; or *Interpreted* code, where the written code is parsed and run directly during run-time. In a sense, JiT compilation can be seen as a mix of AoT compilation and interpretation.

When it comes to performance, interpreted code (such as Python) does not do as well as compiled code (for example C/C++), as the compilers often optimize the code in ways regular interpreters cannot do[13]. With no compilation necessary, interpreted code is very easy to port between platforms. JiT compiled programs are also quite easily ported, as compilation happens at execution - this also enables platform-dependant optimizations, but impact the "start up" time of the program quite substantially.

Numba [23] is an open source JiT compiler for python, translating a subset of Python and NumPy code into fast machine code. Not all NumPy functions are supported 100% in Numba, which means that some reformatting might be necessary when wrapping a class or function with the JiT-decorators. That being said, the documentation is good so integration is fairly straightforward.

It was hypothesized that the desired filter architecture would not be feasible to run in a pure Python implementation. Testing the particle filter on a virtual machine running on a laptop with an Intel i7 6820HQ this hypothesis was confirmed, as the particle filter propagation step alone took in excess of 60 $[ms]$ to complete. The exact same code JiT-compiled using the Numba jitclass wrapper cut the execution-times by a factor greater than 10 to approximately 5 $[ms]$ as seen in figure 3.30. This result gave confidence that an update-rate of 10 $[Hz]$ for the particle filter should be attainable.

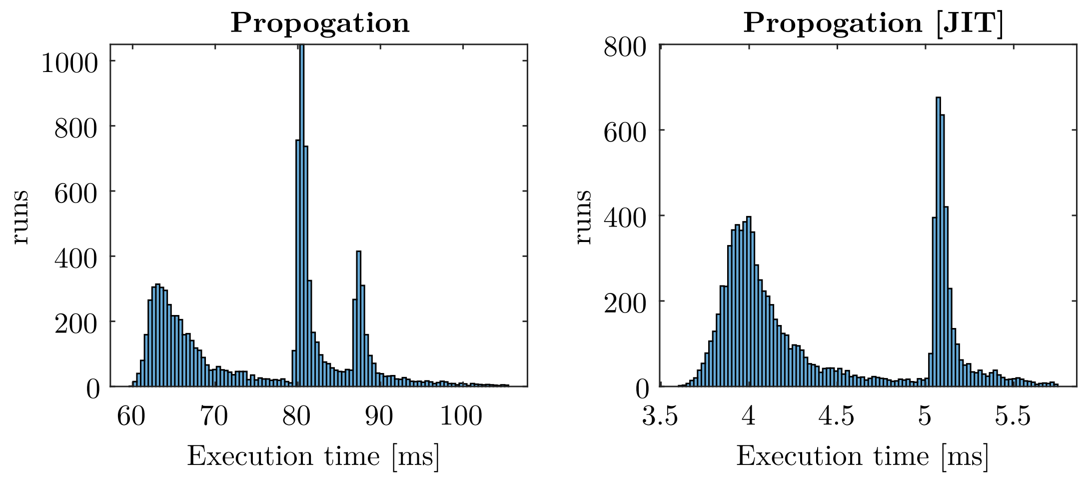



Figure 3.30: The execution-times for the 99th percentile of 10000 runs of the particle filter propagation step, on a development computer running an Intel i7 6820HQ

### 3.11.2 ROS2 implementation

The software is tightly integrated with ROS 2 (robot operating system), and ROS tools are used to handle communication between the Kalman- and Particle-filter. ROS tools are also

[13]This is dependant on the interpreter, there exist interpreters that preform some degree of optimizations on the code (for example Template- and ByteCode- Interpreters)

used to call functions in the Hybrid filter objects. The different nodes subscribe and publishes data to different topics depending on if the system is being simulated or deployed on actual hardware; the main difference in configuration is due to some of the software components outside of the designed systems having predesignated message topics.

#### Kalman filter

The Kalman filter ROS node implements the Kalman filter object. Time based callback functions are used to routinely retrieve data from the Kalman filter object and publish the data to the ROS network. Subscriber callback functions are used to parse the received data, and feed it to the filter object.

The Kalman filter subscribes to the following topics and recives the following messages for:

Position aiding:

- gazeboGT/pose_ned, PoseWithCovarianceStamped
- pf/pose_ned, PoseWithCovarianceStamped, if in hybrid-mode

IMU data:

- sensor/imu_main, Imu , if in simulation-mode
- SensorCombined_PubSubTopic, SensorCombined

The filter node publishes data to the following topics:

position estimate:

- ekf/pose_ned, PoseWithCovarianceStamped

Velocity estimate(in level and body frame):

- ekf/vel_level, TwistWithCovarianceStamped
- ekf/vel_body, TwistWithCovarianceStamped

Sensor bias estimates:

- ekf/sensor_bias, TwistWithCovarianceStamped

#### Particle filter

Like with the Kalman filter the Particle filter utilizes the same implementation methodology of creating a ROS node object that creates a filter object. The particle filter uses subscriber callback functions to parse data and timer based callback functions to publish data at a fixed rate.

The particle filter subscribes to the following topics with the following message types:

Point cloud data:

- zed_mini_depth/points, PointCloud2, if in simulation-mode
- zedm/zed_node/point_cloud/cloud_registered, PointCloud2

Velocity data:

- gazeboGT/vel_level, TwistWithCovarianceStamped
- ekf/vel_level, TwistWithCovarianceStamped if in hybrid-mode

Attitude data:

- gazeboGT/pose_ned, TwistWithCovarianceStamped
- ekf/pose_ned, TwistWithCovarianceStamped if in hybrid-mode

The node publishes data to the following topics:
Position estimate:

- pf/pose_ned, PoseWithCovarianceStamped

Particle point cloud for visualization:

- pf/pose_ned/pointcloud, PointCloud

#### Logger node

A logger node has been created, the node subscribes to desired topics and logges the data it receives on the topics to a CSV file.

#### Transform node

A transform node has been written and interfaced with ROS. The packages subscribes to the position estimate from the Kalman filter and sends the position estimate to the ROS transform server. This allows for the drones estimated position to be visualized in Rviz.

### 3.11.3 Packages

The Hybrid-filter software, supporting software, and simulation files are decomposed into smaller packages, making the software system manageable and flexible. The packages are centered around one main piece of the project each. An explanation of what the different packages contain is detailed below. Separating the project into packages makes it easy to centralize properties like initial conditions for the filters and filter configurations.

#### Filter configuration package

The "*idl_botsy_pkg*" named after the project name for the drone is where the primary configuration of the Hybrid-filter is located. This package contains the geometric data relating to the different frames, the filter initial conditions and configuration classes for the Hybrid-filter. The package also contains the ROS2 launch files for launching the ground station related nodes and the Hybrid-filter nodes.

**Kalman filter package**

The Kalman filter package is named "*idl_orientation_pkg*" in the git group[14] and contains the different implementations of the Kalman filter and the Kalman filter ROS node. The filter is primarily configured from the "*idl_botsy_pkg*"

**Particle filter package**

The particle filter package has the name "*idl_pf_pkg*" in the git group. The package contains the particle filter and the particle filter ROS node. The package also contain the relevant tools for the particle-filter implementation. The same "*idl_botsy_pkg*" is also used to configure the particle filter.

**Transform package**

The transform package contains tools related to the ROS2 environment. The packages is named "*idl_transform_pkg*" in the git group. The package contains the ground truth publisher node written to be used with the gazebo simulation to publish the drone's true state. The package also contains the node responsible for publishing data to the ROS2 transform server that enables visualization in Rviz. There is also a node responsible for configuring the point cloud data from the depth camera to a compatible format with Rviz.

**Gazebo simulation configuration package**

The gazebo simulation work-space and configuration files are located in the "*gazebo_botsy*" git repository in the git group. All the gazebo-related files are located here, including the modified sensor models and the simulation drone model. The different environment models are also located under this package.

**Logger package**

A logger node has been created and is located in the "*idl_logger_pkg*" this node is used for logging data from the ROS network during testing and exports the data to an excel friendly format.

[14] The name "orientation" poorly describes what the packages contains as the Kalman filter also does localization in 3d space, but the name stuck during development

## 3.12 Hardware implementation

### 3.12.1 Drone platform desired properties

The drone is designed using rapid-prototyping techniques, and as such, should not be seen as a final product but rather a platform for software- and system testing. Some desired properties were thought out before the design started:

#### Easy access to the hardware components

Since the platform intends to serve as a vessel for development and testing, it is highly desirable to have easy access to the hardware components like the flight controller and the Jetson TX2 compute module. These are components that should be located inside the drone for their protection, yet be accessible and easy to detach from the drone for desktop testing.

#### Protection of the hardware components

The hardware components are relatively fragile and need protection from the wear and tear that will be put on a drone used for inspection purposes. This is not just in the unfortunate case of a crash but also to protect the components during transportation and general handling.

#### Flexible for testing other hardware components

It is also desired that the drone design is flexible for allowing the exchange of components like the camera, flight controller, and onboard computer. This will allow further testing and evaluation of different components without the need for a different drone design.

### 3.12.2 Drone platform selection

The selected base platform is the Holybro S500 drone kit, as it contained all the essential parts for flying a drone. It is also relatively inexpensive, making it great for prototyping. Conveniently, the supplier has provided motor characteristic data. This data is used to make a rough estimate of the drone's flight time and hover throttle setting.

| Item No. | Propeller | Throttle | Voltage (V) | Torque (N*m) | Thrust (g) | Current (A) | RPM | Input power (W) | Efficiency (g/W) | Operating Temperature |
|---|---|---|---|---|---|---|---|---|---|---|
| AIR2216 KV880 | T1045 | 50% | 16 | 0.07 | 435 | 3.5 | 6015 | 56 | 7.77 | 53.5°C |
| | | 55% | 16 | 0.08 | 527 | 4.6 | 6620 | 73.6 | 7.16 | |
| | | 60% | 16 | 0.09 | 608 | 5.6 | 7113 | 89.6 | 6.79 | |
| | | 65% | 16 | 0.11 | 702 | 6.8 | 7563 | 108.8 | 6.45 | |
| | | 75% | 16 | 0.13 | 888 | 9.5 | 8545 | 152 | 5.84 | |
| | | 85% | 16 | 0.15 | 1076 | 12.3 | 9442 | 196.8 | 5.47 | |
| | | 100% | 16 | 0.18 | 1293 | 16.2 | 10464 | 259.2 | 4.99 | |

Notes:Motor temperature is motor surface tenperature @100% throttle running 10 mins.
(Data above based on benchtest of 2018 are for reference only. Comparison with that of other motor types is not recommended.)

Figure 3.31: Motor characteristics for S500 drone kit

Figure 3.31 shows the motor data from the supplier of the kit.

The data from figure 3.31 was used to curve fit polynomials for the relationships *Thrust to Current*, *Thrust to Input power* and *Thrust to Throttle.* Then a candidate lithium polymer battery was selected, and a rough power density was calculated based on this battery.

$$\rho_{capacity} = \frac{m_{battery}}{c_{battery}} = \frac{0.375[kg]}{4000[mAh]} = 9.4 \cdot 10^{-5} \left[\frac{kg}{mAh}\right] \tag{3.82}$$

The selected candidate battery was a candidate battery from a local hobby store and had a capacity of 4000 $[mAh]$ and weight of 0.375 $[kg]$. This is chosen to be roughly representative of the batteries available for the project.

The listed weight for the S500 kit is $1\ kg$, the total weigh of the drone is then said to be:

$$m_{total} = \rho_{capacity} \cdot c_{battery} + m_{kit} + m_{payload} \tag{3.83}$$

Here, $m_{kit}$ is the listed weight of the S500 kit, $m_{payload}$ is the weight for all the parts to be designed as well as computer and camera system, $c_{battery}$ is the capacity for the battery selection.

The nominal power draw is found by evaluating the fitted polynomial from the figure 3.31 using the total mass of the drone for the calculation:

$$p_{nom} = f(m_{tot}) \tag{3.84}$$

The flight time of the drone is then calculated as:

$$t_{flight} = \lambda_{safety} \frac{v_{battery} \cdot c_{battery}}{p_{nom}} \tag{3.85}$$

The factor $\lambda_{safety}$ is set as a safety factor; for the calculations, 0.75 is used, which means that 75% of the battery capacity is available to use before needing to land, leaving some overhead.

The nominal throttle setting of the drone is then calculated based on the fitted polynomial. The nominal throttle setting must not be too high, as this can lead to actuator saturation, that is if the controller commands a set-point above the maximum available actuation effort. This is obviously undesirable. To lessen the chances of a saturation event from happening, a component selection and design that keeps the nominal throttle setting as low as possible is desired.

The above functions have been evaluated for a range of battery capacities and payload masses, giving the following curves in figure 3.32:

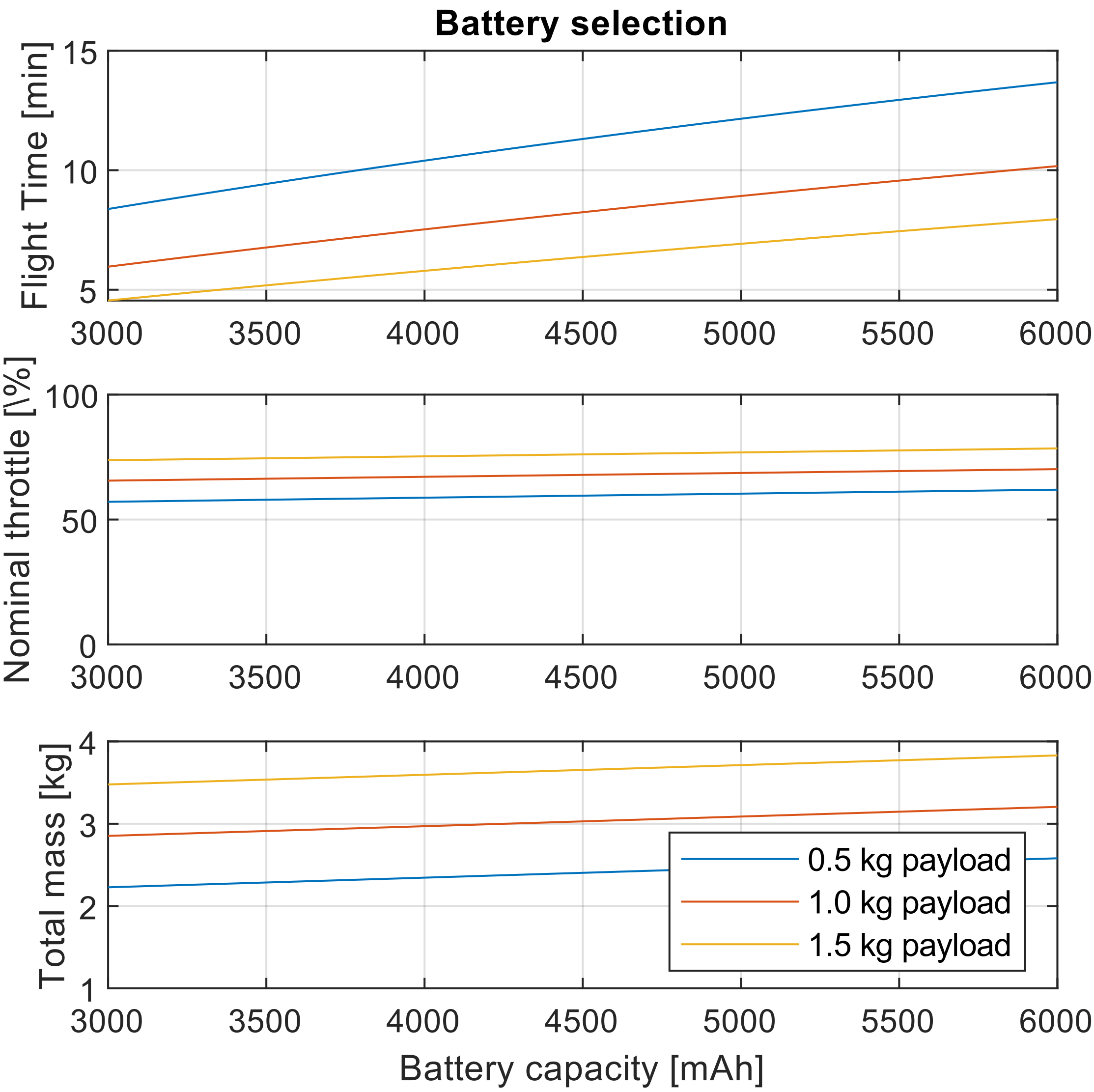



Figure 3.32: Drone design, battery and payload design graph

In figure 3.32 shows how different battery capacities will affect the flight time and the nominal throttle setting of the drone.

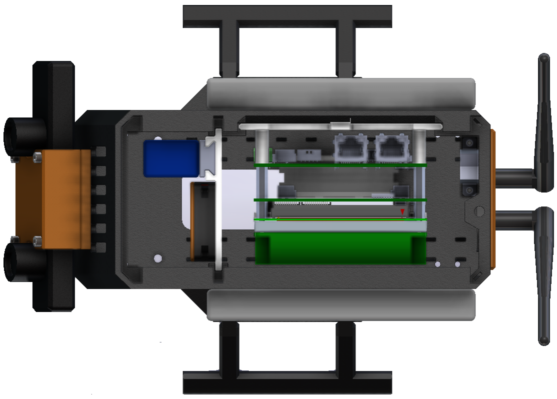
### 3.12.3 Drone design

The drone assembly was split into two major assemblies; the bottom part, named the *Electronics bay* as it contains the majority of the hardware used for localization, and the top part of the drone, named the *Dome*. The Dome houses the flight controller, telemetry radio, and the drone's GPS module.

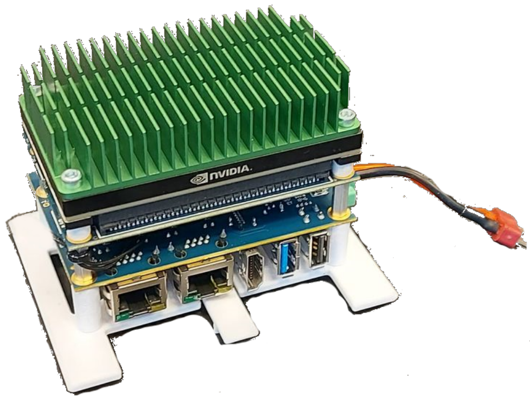
#### Electronics bay

The electronics bay primarily consists of one large tub-like main hull that most electronics mounts inside of. This is designed as an outer shell which will protect the electronics from impacts during transport and give some resilience against crashes.

The Nvidia Jetson TX2 is mounted on a mounting plate that slides down into the main hull, making it easily accessible and a flexible mounting solution for alternative computer candidates. The slide-in bracket is also held in place by a cover plate mounted to the bottom of the electronics bay.



(a) Inside electronics bay



(b) Jetson TX2 on mounting bracket

Figure 3.33: Drone design Jetson TX2 mounting

Mounting of the Zed Mini stereo camera is done with a screw-in-place bracket. This makes it simple to make a similar mounting solution for an alternative depth camera or LiDAR. The mounting backing plate that the camera screws into is also designed as a separate part from the main hull, making it possible to change the camera's angle. Figure 3.34 displays the mounting bracket.

The batteries are also mounted to either side of the drone's main hull. The drone uses a four-cell lithium battery pack, so a pair of two-cell battery packs are used, one on either side wired in series to create a four-cell battery pack. Mounting the batteries on either side keeps the drone from growing too tall in the vertical direction. There is also ample space here to use battery packs in a wide range of capacities. Figure 3.35 shows the intended mounting location for the batteries. It is also possible to adjust the location of the battery packs to the left and right direction in the figure, making it simple to place the center of gravity under the center of lift of the drone.

Figure 3.34: Drone design Zed Mini stereo camera mount

Figure 3.35: Drone design battery mounting

### Dome

The Dome assembly consists of a dome-like part covering the flight controller and cabling between the different hardware components.

The Dome serves as a protective cover but also a mounting location for the GPS module and telemetry radio. A recessed mounting location is designed for the GPS module. Figure 3.36 shows the dome assembly and a section view of the assembly.

(a) Dome assembly

(b) Done assembly section view

Figure 3.36: Drone design Dome assembly

The Pixhawk 4 flight controller is mounted on a tray that slides into a mounting bracket. This is done so that the flight controller can be taped in place using vibration-isolating foam pads but still be easily removed from the drone without the risk of destroying the vibration isolating pads. This can be seen in figure 3.37.

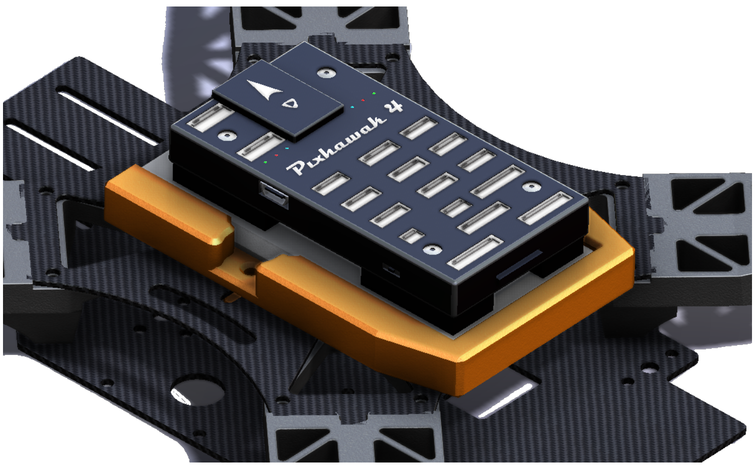



Figure 3.37: Drone design Pixhawk 4 flight controller mounting tray

The complete drone assembly is shown in figure 3.38

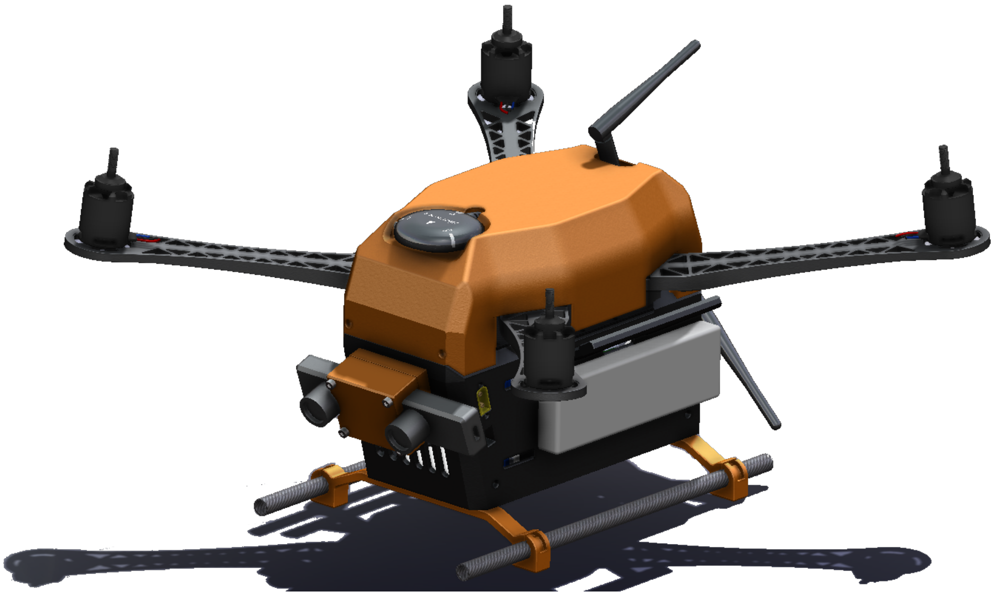
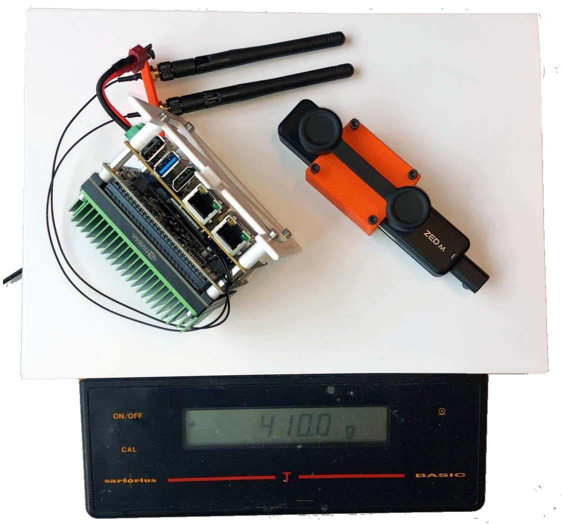


Figure 3.38: Drone design complete assembly

#### Dummy parts

To emulate the TX2 and Zed Mini camera during test flight where the computation and stereo camera is not needed, some dummy parts have been made that replicates them in mass and shape.




(a) Real components, mass = 410.0 g


(b) Dummy components, mass = 421.1 g

Figure 3.39: Drone design real vs. dummy components

### 3.12.4 Completed drone

The assembled drone consists mainly of 3D printed parts and the parts from the S500 drone kit. The total mass of the drone is roughly 2 $[kg]$, as the kit weighed 1 $[kg]$ the payload weight is then 1 $[kg]$. Looking back at figure 3.32 a battery can be selected based on the desired flight time.

The battery pack selected was a set of two, two cell 4000 $[mAh]$ lithium polymer batteries. Resulting in an estimated flight time of approximately $7,5$ $[min]$. The reason for selecting this battery configuration came down to the accessibility of batteries and a desire to be conservative with the drone's weight.

(a) S500 kit assembled (b) Assembled drone with designed parts

Figure 3.40: Drone design assembled

Figure 3.40 displays the built drone next to the assembled S500 kit as delivered by Holybro.

The drone was test flown and had a flight time of approximately $7min$; this is close to the estimated flight time based on the drone's motor data, battery capacity density, and mass. The flight time and distance flown can be seen in the logged track from the program QGroundControl in figure 3.41

It should be noted that the battery status indicator in figure 3.41 was not correctly calibrated at the time of the flight. Instead, a simple battery alarm was used to indicate when the batteries were exhausted.

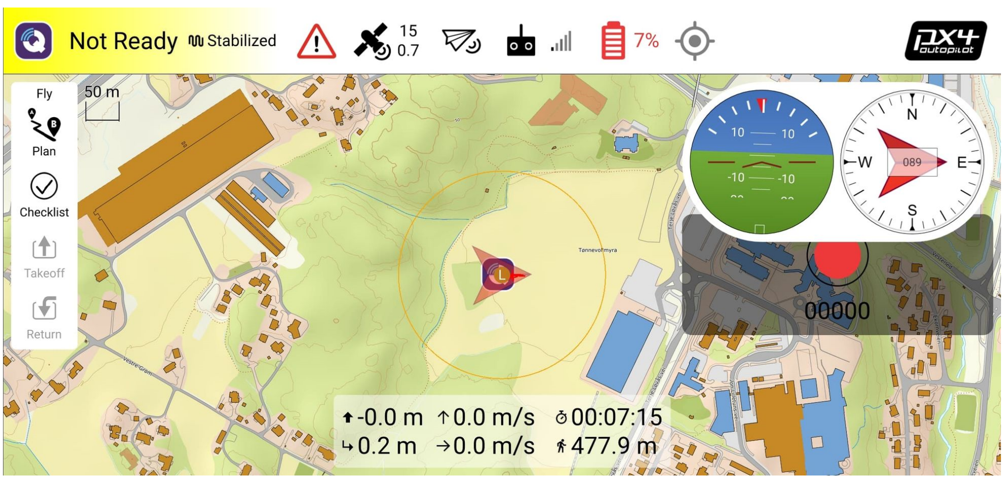



Figure 3.41: Test flight of the drone, flight time: 7 $min$ 15 $sec$, distance: 478 $meters$

Page intentionally left blank

# Chapter 4

# Results

## 4.1 Hybrid filter performance

To test the hybrid filter several test scenarios have been preformed. All in the Gazebo simulation environment, using the drone model devised for the project in the Industrial- and the UiA Basement test environments.

Three main scenarios have been tested.

- Case 1: A simple hop test in the Industrial test environment. The aim of this test is to compare the Hybrid filter to a pure Kalman- and a pure particle-filter solution. The drone has preformed a simple hop up to an altitude of 5 meters, preformed a complete 360 degree revolution and then landed where it took off from. The test is intended to be a simple test case that will show case the weaknesses of the individual filters and demonstrate the improvements made by combining them to a Hybrid-filter.
- Case 2: A longer test scenario where the drone flies a closed circuit in the Industrial test environment. The test is intended to demonstrate that the system is capable of navigation in a representative Industrial setting. The flight path is set up in a closed circuit to mimic that of an inspection flight.
- Case 3: Simulated test in the UiA campus Grimstad Basement. The test is intended to demonstrate that the Hybrid filters performance in an environment ridden with long narrow hallways. The tests in the UiA Basement will also act as a good comparison for a future full scale deployment of the Hybrid filter.

Analyzing the stability of the filter is difficult due to the inherent random events occurring in the sampling, propagation and re-sampling steps in the particle filter. Therefore the test cases have been preformed several times and a statistical analysis of the results have been preformed.

The system can be seen executing Case 2 and 3 in the YouTube video, available online at `https://youtu.be/pDOOLkh2-aE`.

### 4.1.1 Simulation setup

All the simulations have been preformed as HIL (hardware in the loop) simulations, that is, the filters used in the test cases presented have been executed on the Nvidia Jetson TX2i. The simulation have been executed on a simulation host computer and the two computes have communicated via Ethernet.

The filter parameters used during the tests can be seen in appendix B.2.1.

### 4.1.2 Case 1: Hop test

The hop test have been performed for both filter individually, and then combined in the Hybrid filter solution. The hop test have been preformed for the filters separately to highlight their weaknesses, then combined to demonstrate in a simple test case that the Hybrid filter is capable of localization.

#### Particle filter test

The below results are Hop tests performed with a pure particle filter solution.

Figure 4.1 displays the true and estimated trajectory of one of the test hops. In the error plots in the same figure it can be seen that the deviation in $\psi$ is quite large while turning (4-12 seconds).

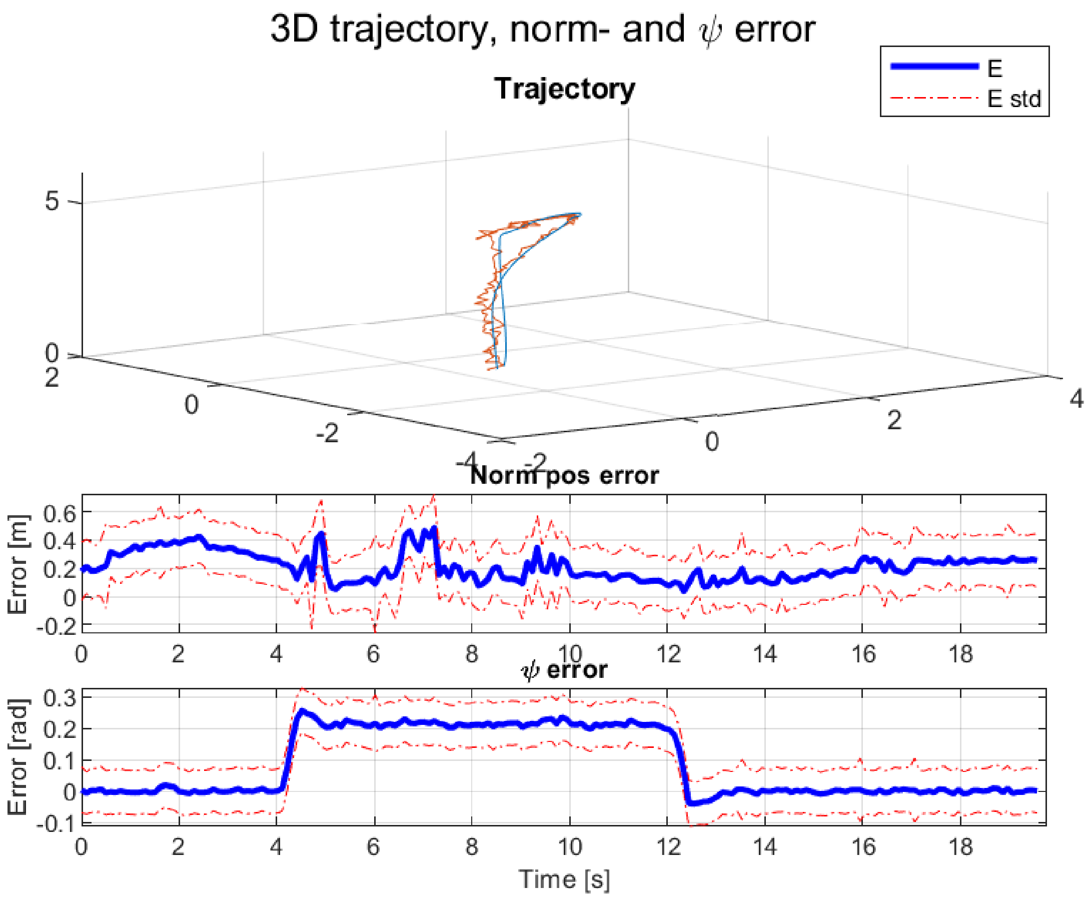



Figure 4.1: 3D Trajectory of true position and estimate from particle filter, with norm error over time

From the trajectories in the X, Y and Z directions as well as the estimated yaw angle seen in figure 4.2 it can be seen that the filter solution is quite "jagged". This is due to the random dispersion of the particles. This "Jaggedness" makes for a navigation solution that is problematic to navigate the drone by.

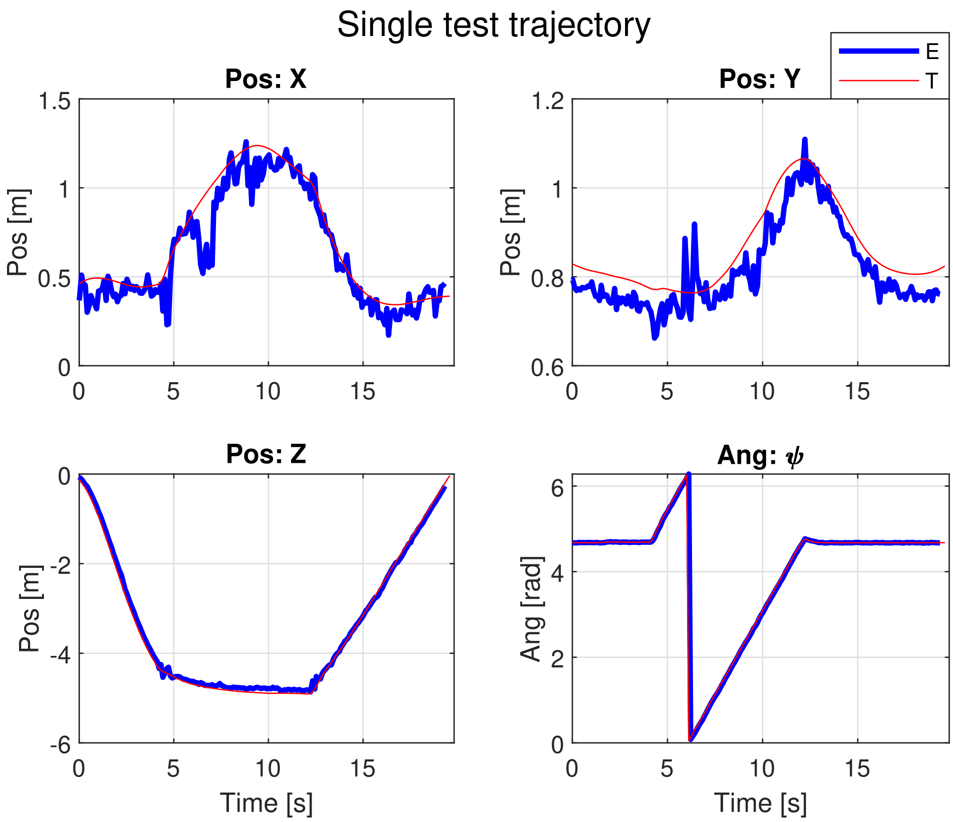



Figure 4.2: True position and estimate from particle filter in X, Y, Z and Psi states

Figure 4.3 displays the mean error and the error standard deviations (red dashed lines) in the X, Y and Z directions and the mean error in the heading estimate of 10 test hops.

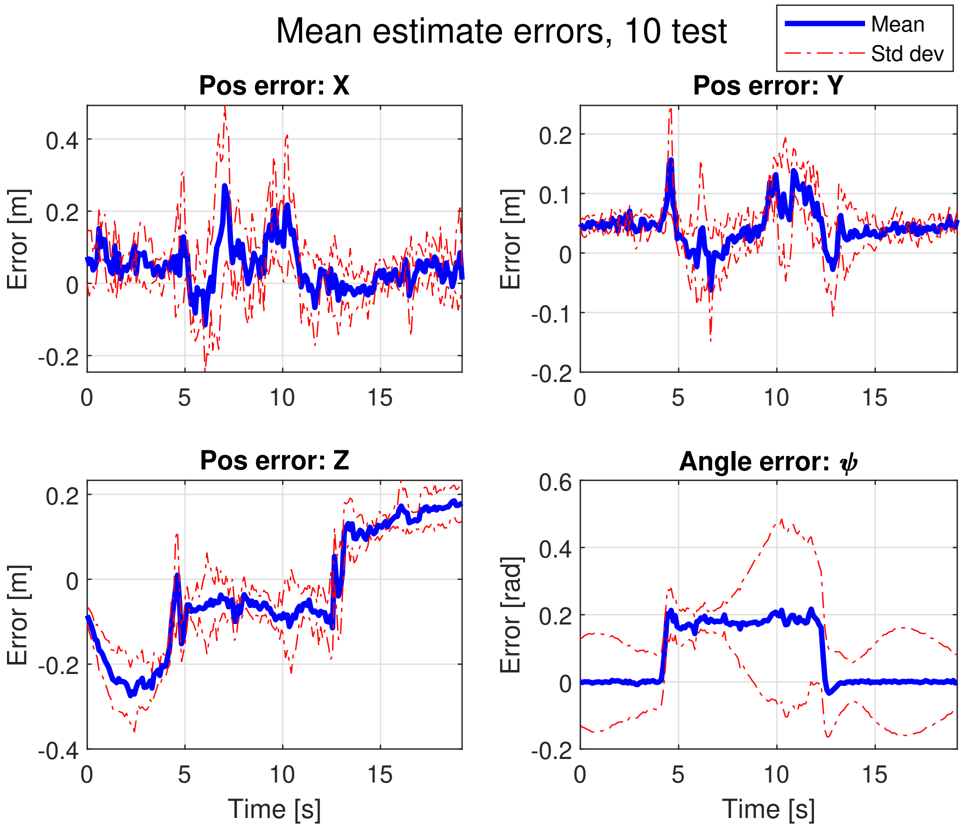



Figure 4.3: Mean estimate errors with standard deviation from particle filter in X, Y, Z and Psi

The errors and the covariance in the particle filter is also larger than desired, this is due to the filter having to disperse the cloud of particles over a large area in its state space, this is needed as the particle filter has no information about the velocities of the drone, and therefor no information about what direction it is best to propagate the particles.

It should also be remembered that the particle filter does not estimate the roll and pitch angles of the drone and can therefor not be used as a stand alone estimator for the drone. This means that the point cloud is not properly leveled when the Kalman filter is not in the loop, this will cause a problem if the drone is commanded to roll or pitch any significant amount. This is a significant problem, as the drone needs to roll and pitch to maneuver it the horizontal plane.

### Kalman filter

The same series of hops where performed with a pure Kalman filter solution, the filter quickly looses its position estimate due to the accelerometer biases. Figure 4.4 displays the true and estimated trajectory of one of the test hops. In the normal error plot it can be seen that the filter quickly drifts away from the true position. This can also be seen in figure 4.5 where the estimated X, Y and Z tracks along with the estimated Yaw angle can be seen compared to the true track during simulation.

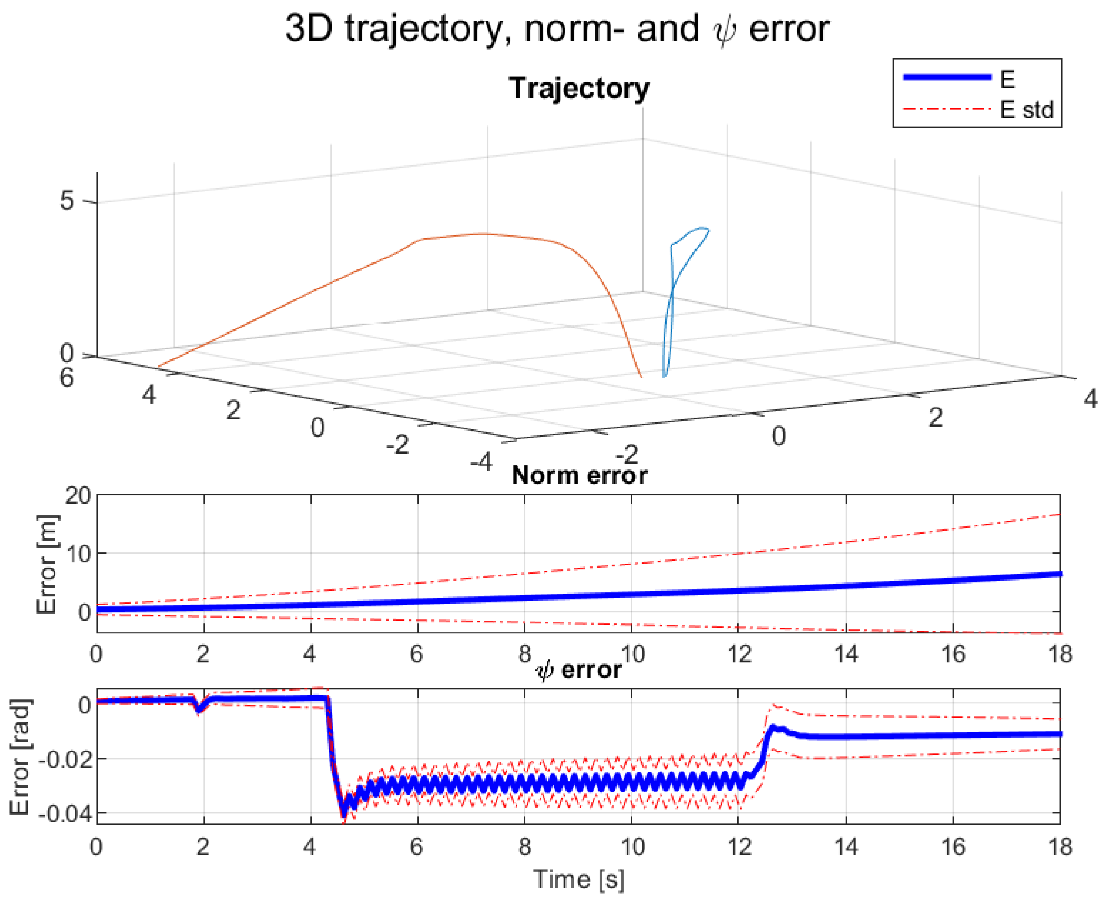



Figure 4.4: 3D Trajectory of true position and estimate from Kalman filter, with norm error over time

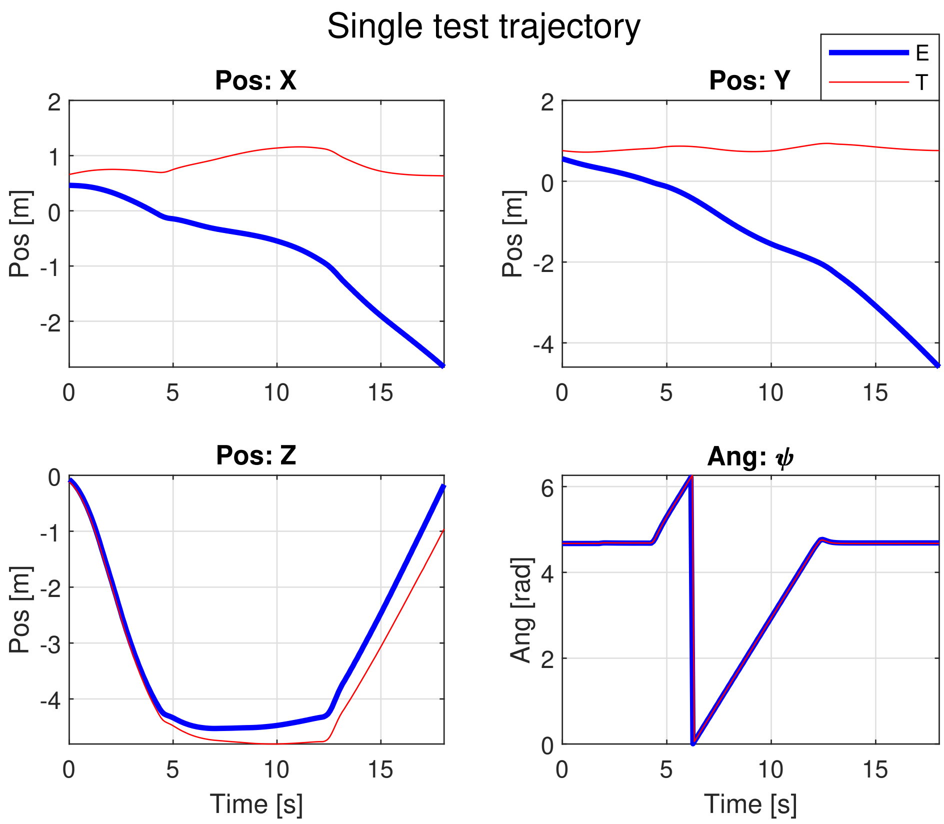



Figure 4.5: True position and estimate from Kalman filter in X, Y, Z and Psi

As mentioned the estimation error quickly grows due to the accelerometer biases not being estimated properly, as well as small errors in the attitude estimates. These errors are left unchecked since the Kalman filter is not receiving position or heading aiding.

Figure 4.6 displays the mean error and the error standard deviation in the X, Y and Z directions along with the mean error in the heading estimate of 10 test hops.

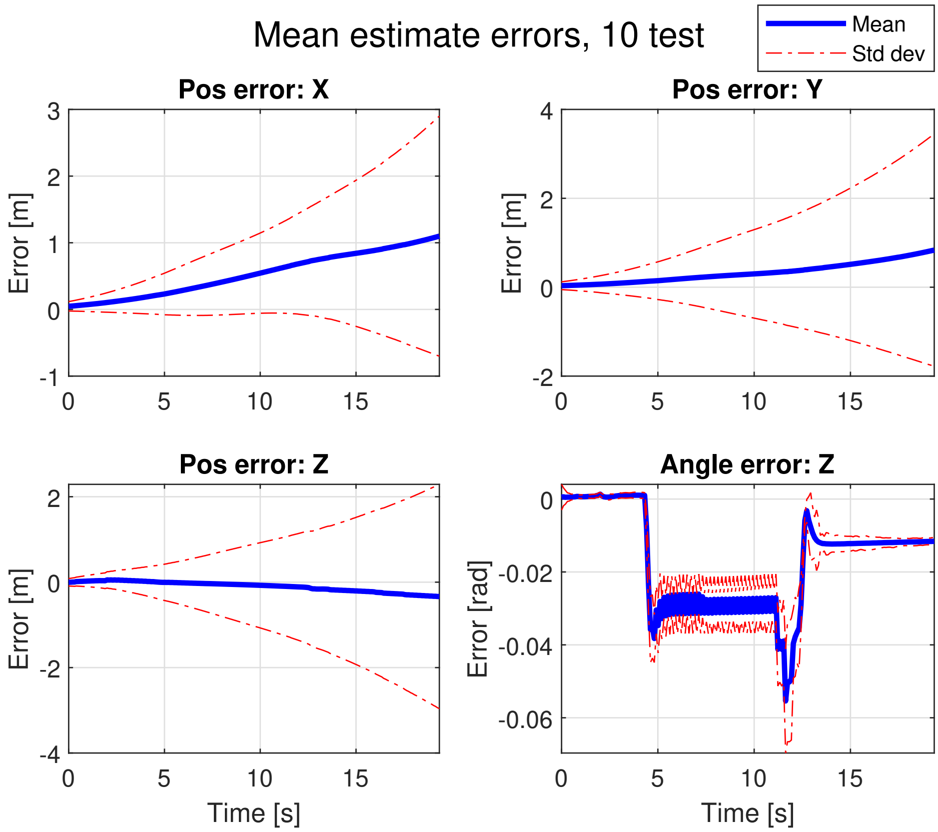



Figure 4.6: Mean estimate errors with standard deviation from Kalman filter in X, Y, Z and Psi

Here it can be seen that the error grows quickly, and the position estimate quickly becomes unusable even during a short test hop. This clearly illustrates that the Kalman filter needs position and heading aiding[1].

[1]The mean value stays close to 0, this is because the filter drifts away in random directions each test, resulting in a zero mean, however it can be seen that the standard deviation grows with time, indicating that the individual test hops have a large error at the end of each test

#### Hybrid filter

Finally the filters are tested in conjunction as a Hybrid filter. The filter performance for a single hop test can be seen in figure 4.7 where the trajectory of the true and estimated position can be seen. The accompanying error in the estimates are smaller than that of the stand alone Kalman- and Particle filter solution. The estimated trajectories can be seen in figure 4.8 with the true position shown in red, here again an improvement over the separate filters can be seen.

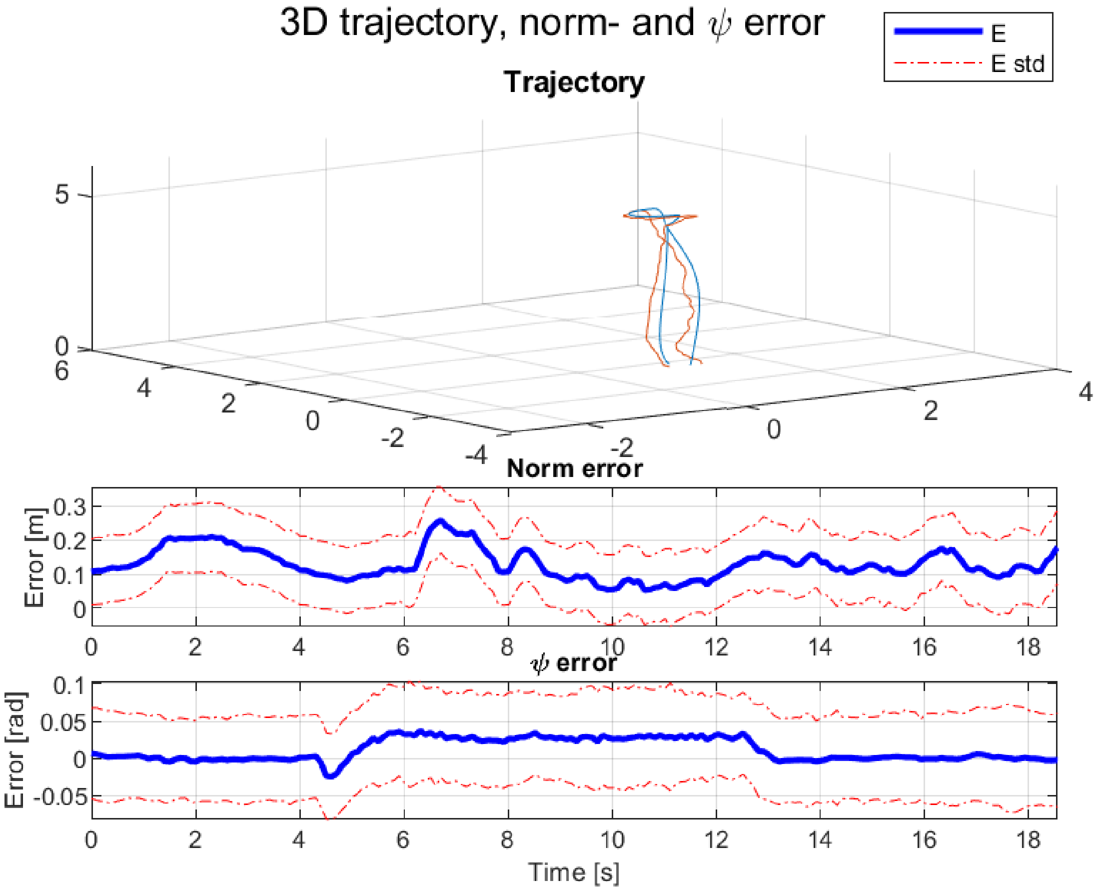



Figure 4.7: 3D Trajectory of true position and estimate from Hybrid filter, with norm error over time

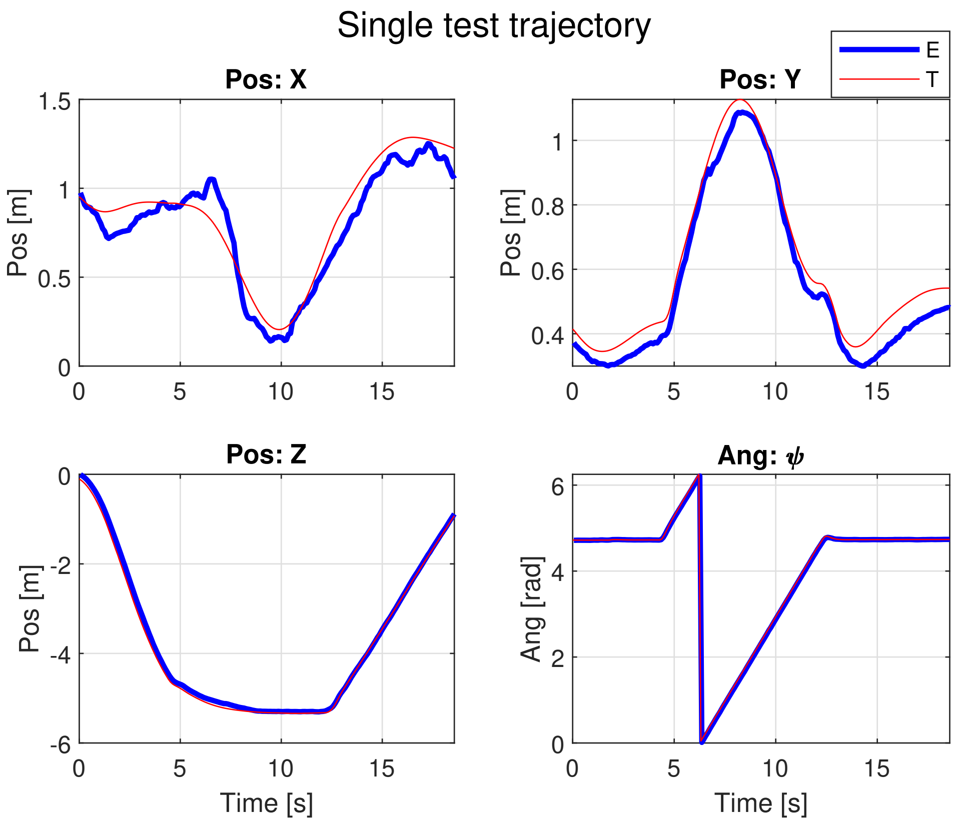



Figure 4.8: True position and estimate from Hybrid filter in X, Y, Z and Psi

The Hybrid filter also estimates the roll and pitch angles of the drone, the estimated angles during the hop test can be seen in figure 4.9, these angle estimates are from the same test as displayed in figure 4.7.

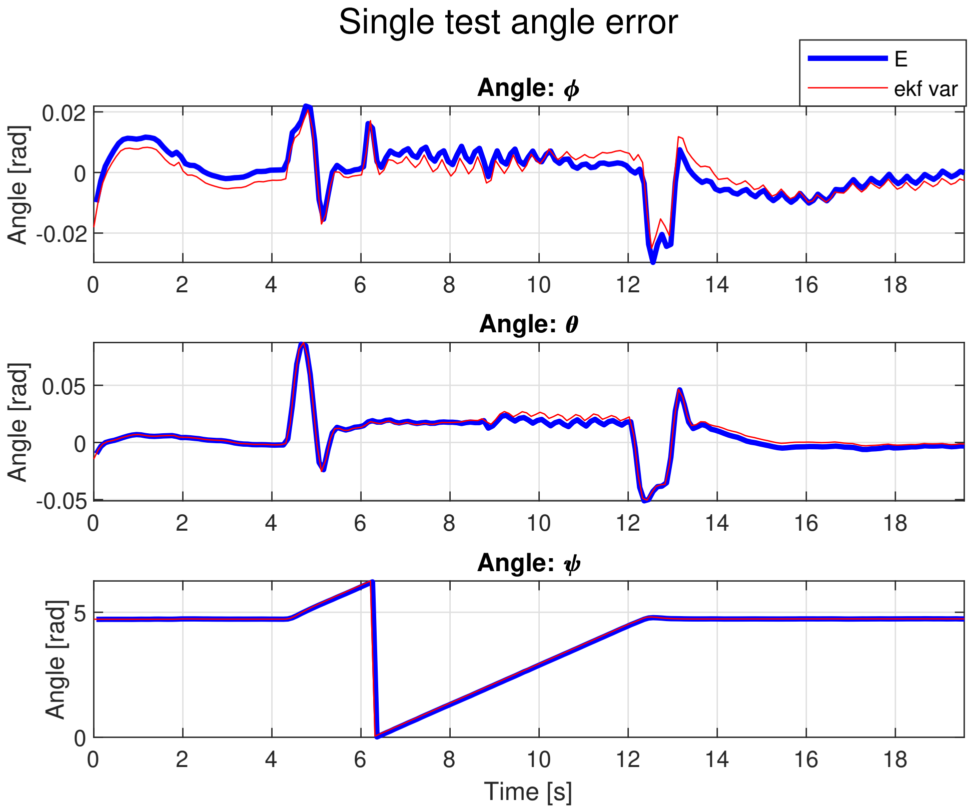



Figure 4.9: Roll pitch and yaw estimate from Hybrid filter and true value

The errors in the angle estimates over the 10 hop tests can be seen in figure 4.10. It can be seen that the errors in the angle estimates are fairly small and consistent over the test runs. This is due to the angle primarily being estimated based on the gyroscope and aided by presence of the G vector in the accelerometer.

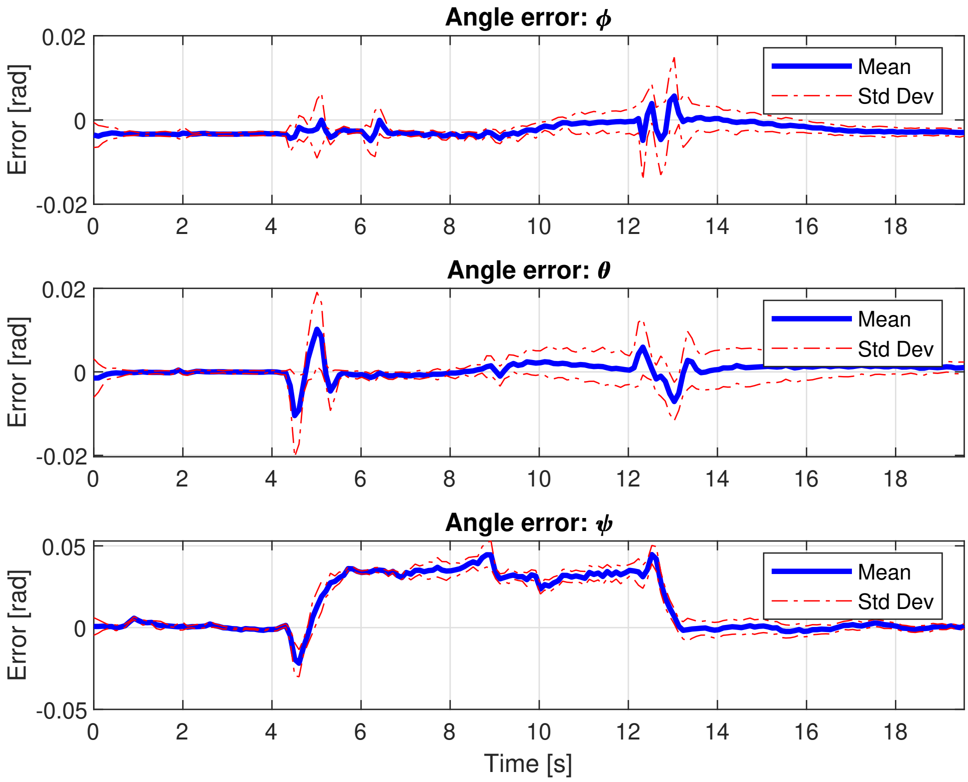



Figure 4.10: Roll pitch and yaw error from Hybrid filter and standard deviation

Figure 4.8 displays the errors in the estimate in the X, Y and Z directions as well as the error in the heading estimate. The Figure displays the mean value over 10 test hops and the red dashed line is the standard deviation of the test series.

From these plots it can be seen that the Hybrid filter clearly out-preforms the separate filters. This is as expected and the intent of the Hybrid filter design.

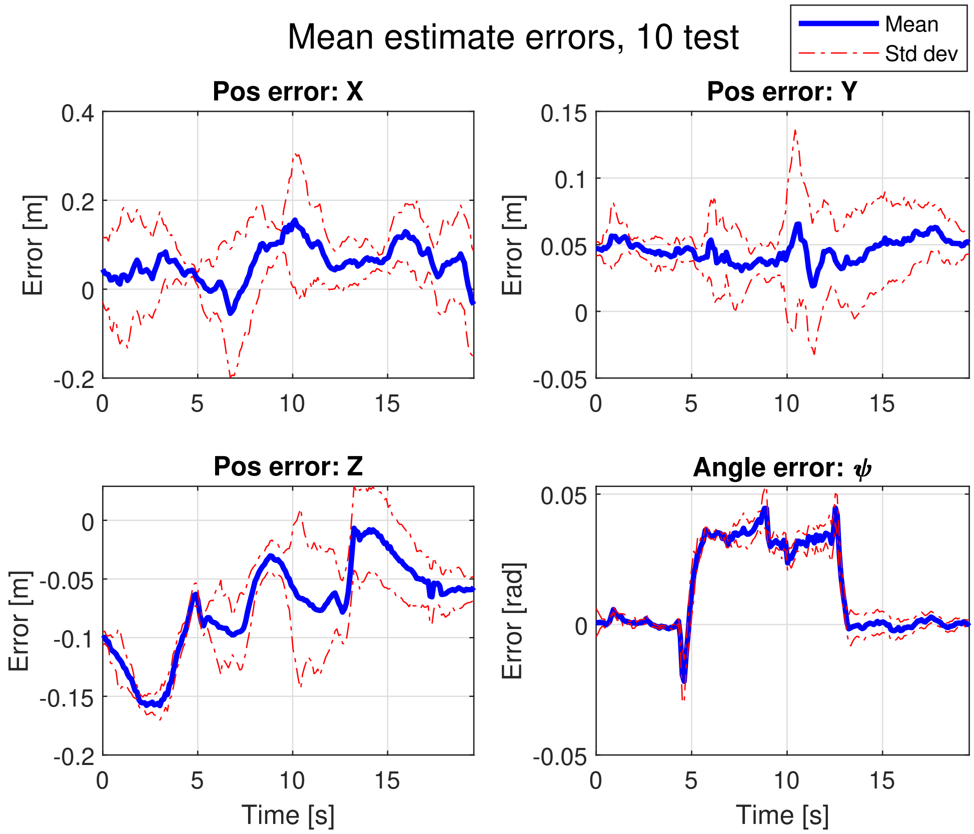



Figure 4.11: Mean estimate errors with standard deviation from Hybrid filter in X, Y, Z and Psi

### 4.1.3 Case 2: industrial environment

The test path in the industrial map has been flown several times and the results logged.

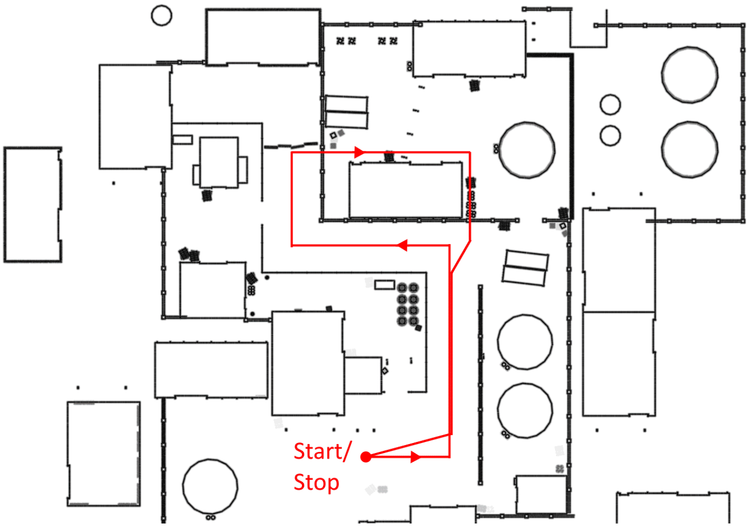



Figure 4.12: The path flown in the industrial map

Figure 4.13 displays the true and estimated trajectory for one run in the industrial map, as well as the normal error in the estimate and the error in the heading estimate.

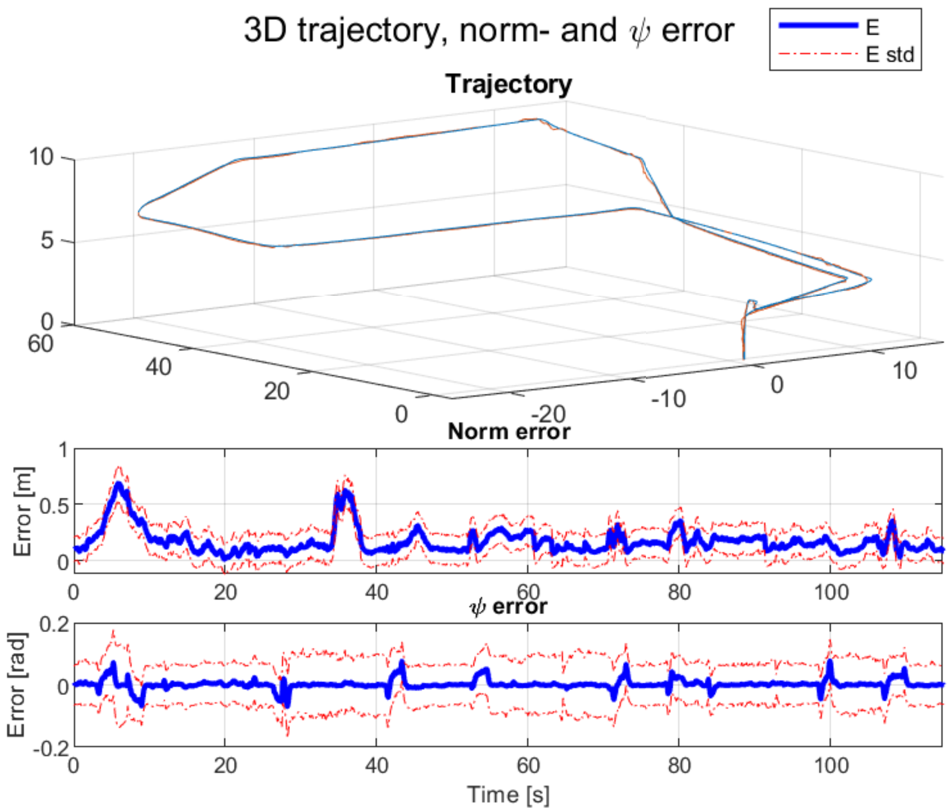



Figure 4.13: Industrial map 3D Trajectory of true position and estimate from Hybrid filter, with norm and $\psi$ error over time

The estimated angles is displayed in figure 4.14, and the estimation errors in the angles with the variance from the filter displayed in figure 4.15, the filter tracks the angles well.

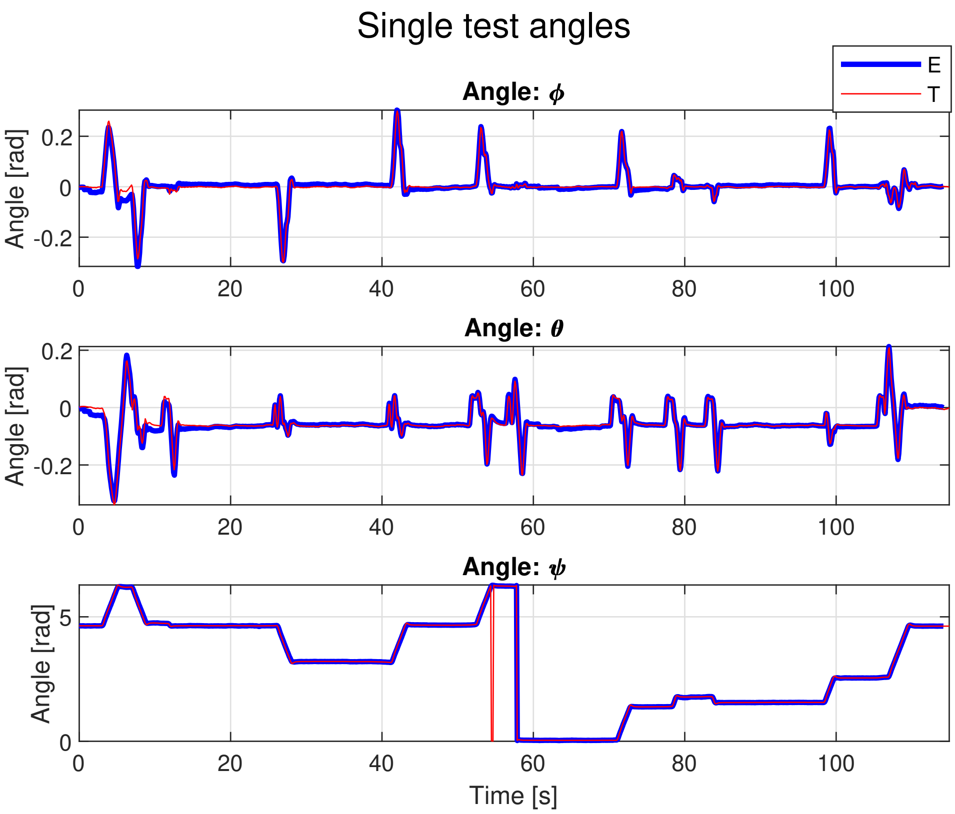



Figure 4.14: Single test estimates of roll, pitch, yaw and from filter with true values

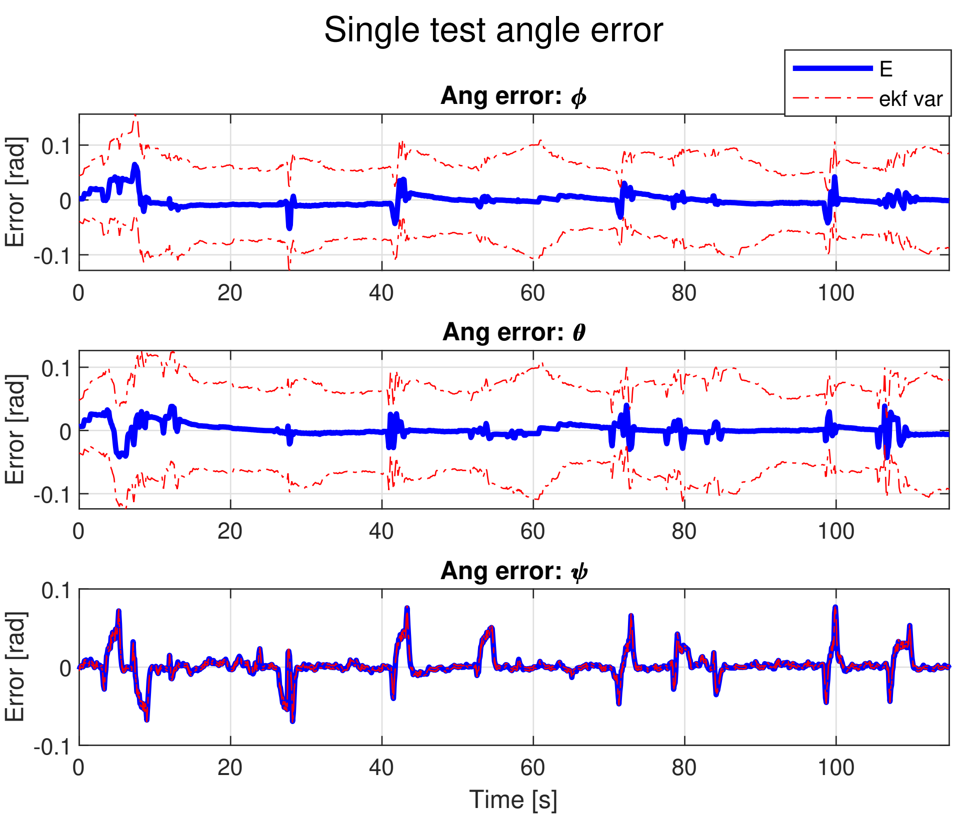



Figure 4.15: Single test estimates of roll, pitch, yaw error and uncertainty from filter

The estimated velocities in the level frame can be seen in figure 4.16. The estimated velocity in the level frame X-direction is close on 2 $[m/s]$, which is the commanded velocity set in the path planer.

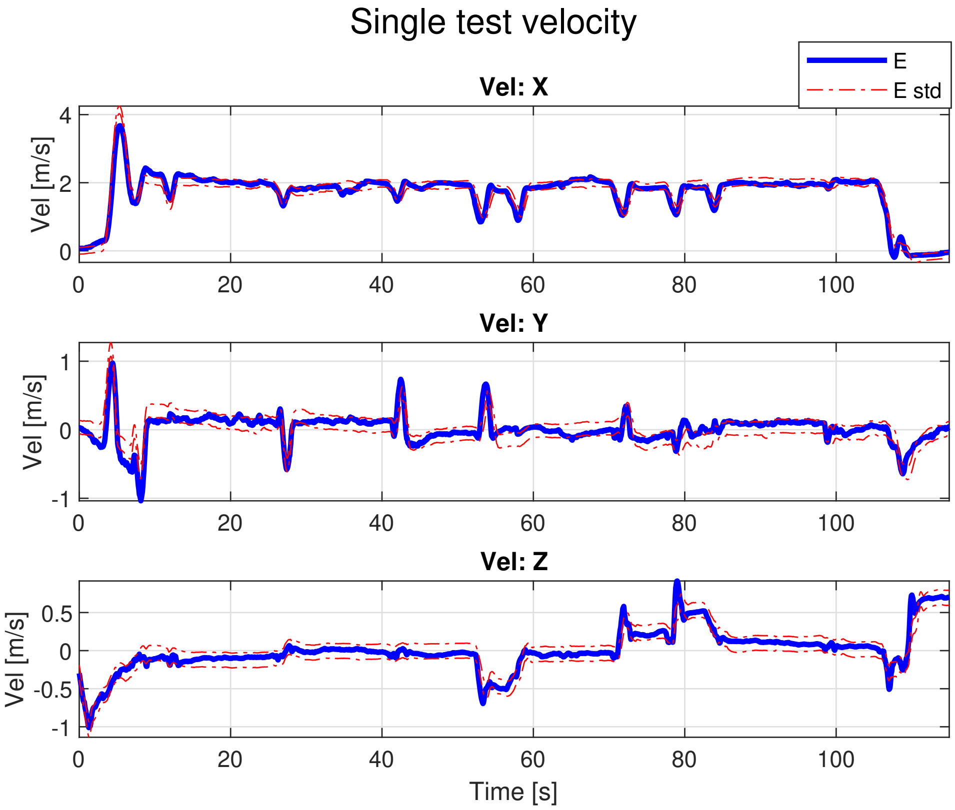



Figure 4.16: Single test velocity estimates in X, Y and Z

As mentioned the tests have been preformed several times and the estimation errors in the X, Y and Z direction along with the error in the heading over 10 tests can be seen in figure 4.17

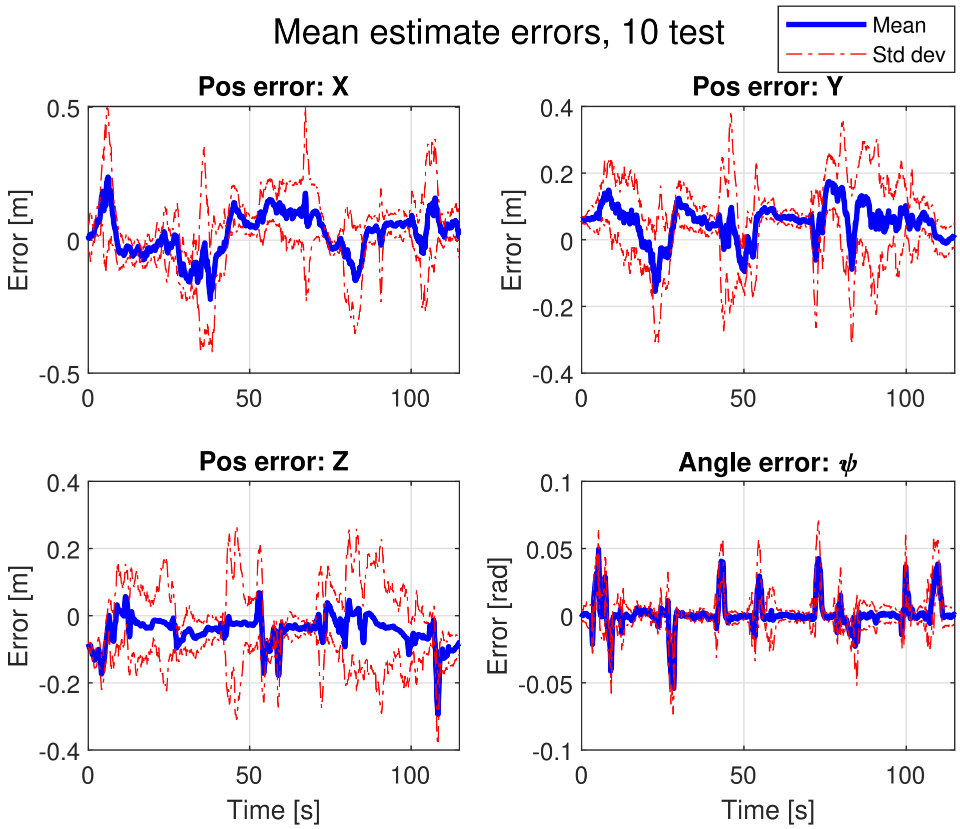



Figure 4.17: X, Y, Z and yaw error from Hybrid filter and standard deviations in the industrial map

It can be seen that the standard deviations in the errors over the multitude of tests are fairly low, indicating that the filter is preforming stably.

The mean angle errors can be seen in figure 4.18, again the filter is preforming consistently.

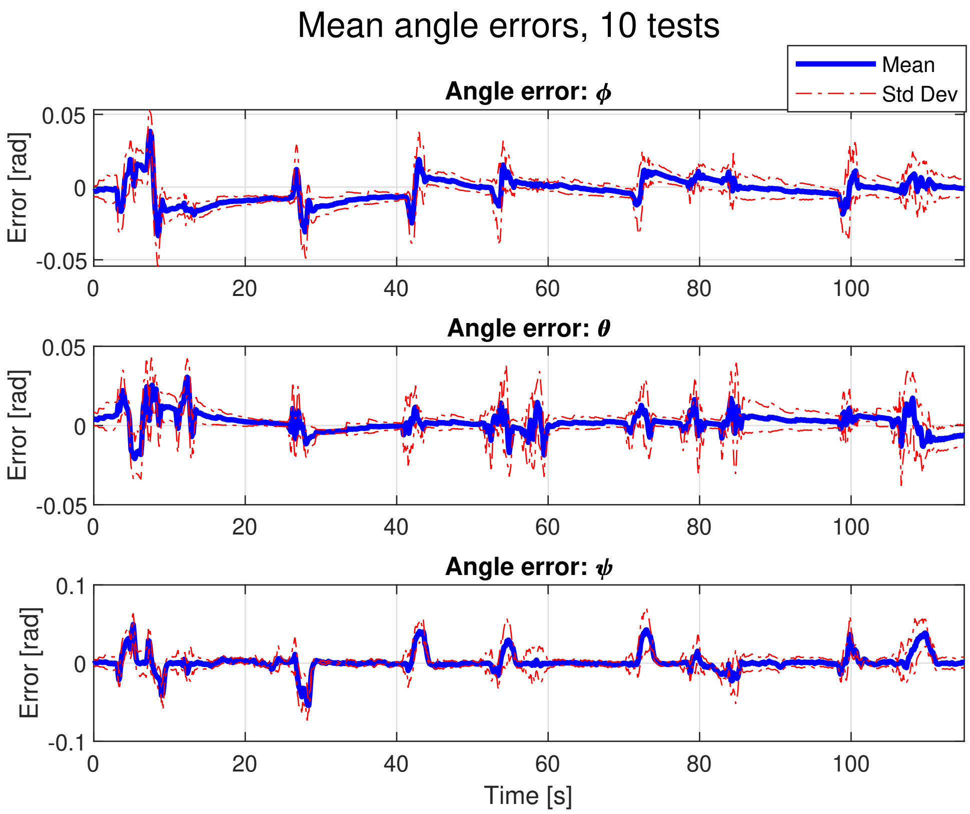



Figure 4.18: Mean values of roll, pitch and yaw error from Hybrid filter and standard deviations in the industrial map

Figure 4.19 shows the linear velocity estimate of the drone in the level frame, that is, the x-axis is oriented in the forwards direction of the drone and level with the horizontal plane. Figure 4.20 displays the bias corrected angular rate measurements

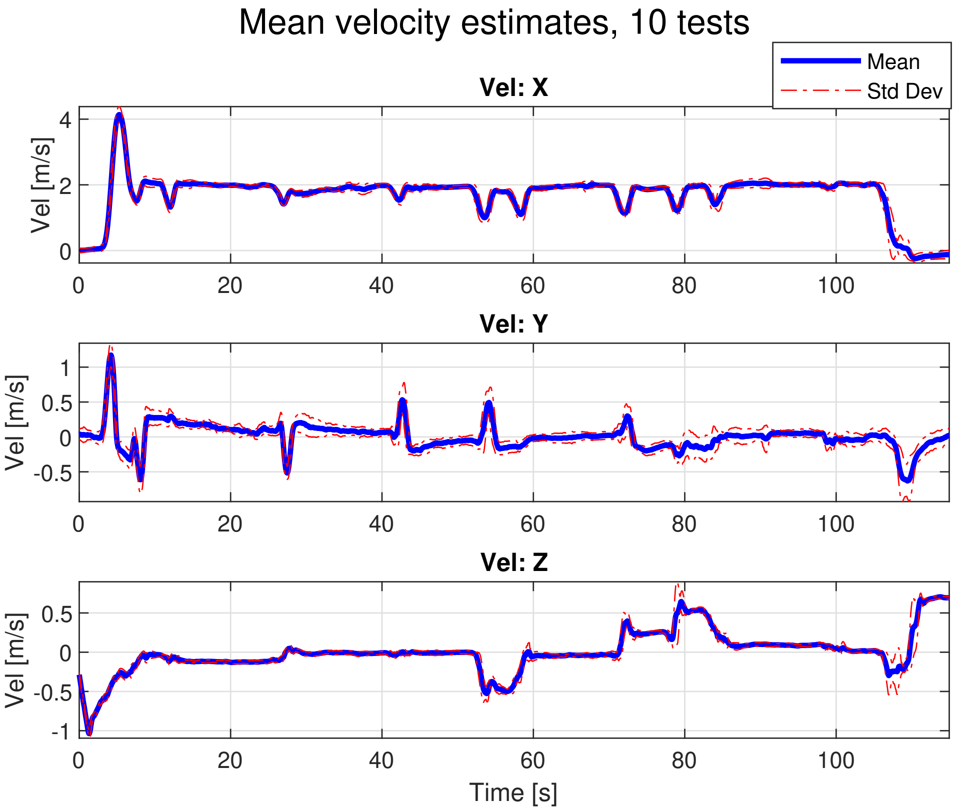



Figure 4.19: Mean values of linear velocity estimates and their standard deviations over time for industrial map

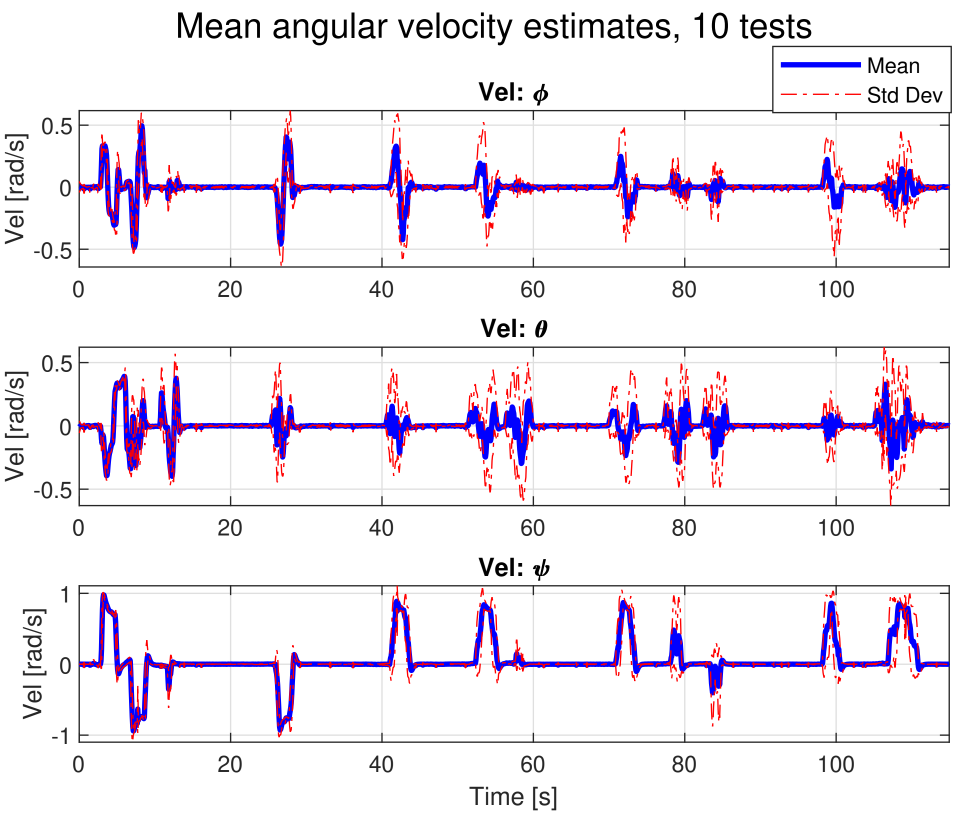



Figure 4.20: Mean values of angular velocity estimates and their standard deviations over time for industrial map

The filter is also estimating the sensor biases, the mean accelerometer biases estimated by the filter over the 10 tests can be seen in figure 4.21. The value for the biases set in the gazebo IMU model fot eh test was $0.0[m/s^2]$ for the accelerometer and $0.1[rad/s]$ for the gyroscope.

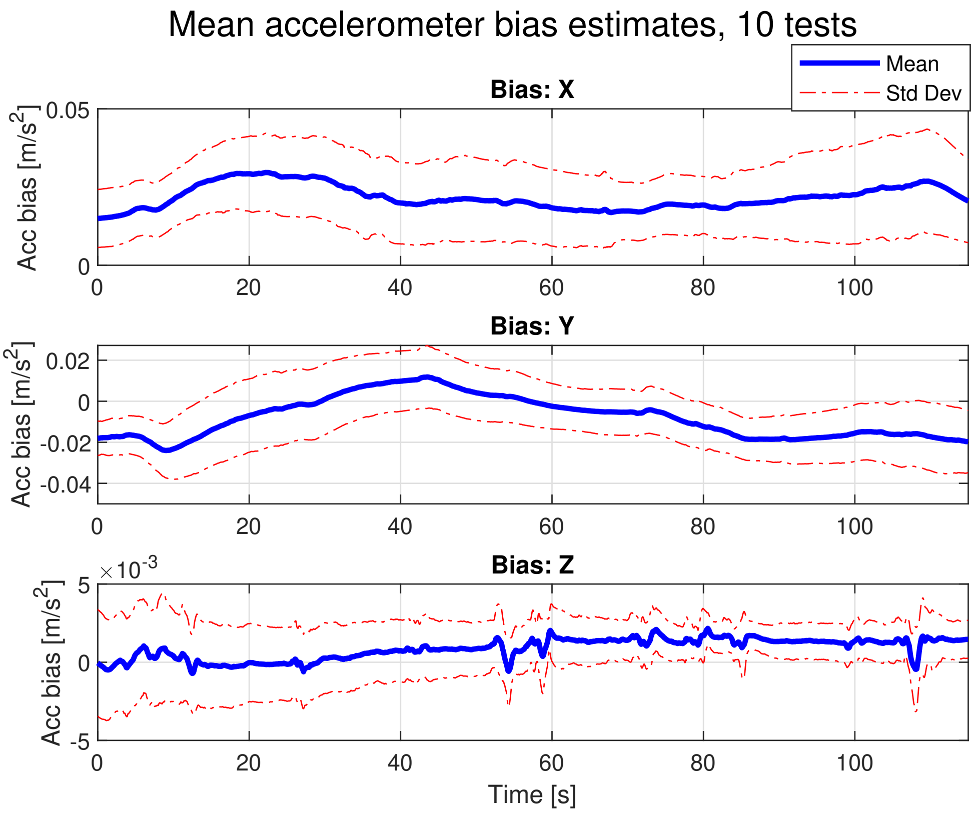



Figure 4.21: Mean values of accelerometer bias estimates from Hybrid filter with standard deviations

The accelerometer biases in the X and Y directions becomes observable as the drone rolls and pitches, the biases are not optimally estimated, and leaves some performance to be desired.

The estimated gyroscope biases can be seen in figure 4.22. The gyroscope biases are estimated to a higher degree of accuracy then the accelerometer biases.

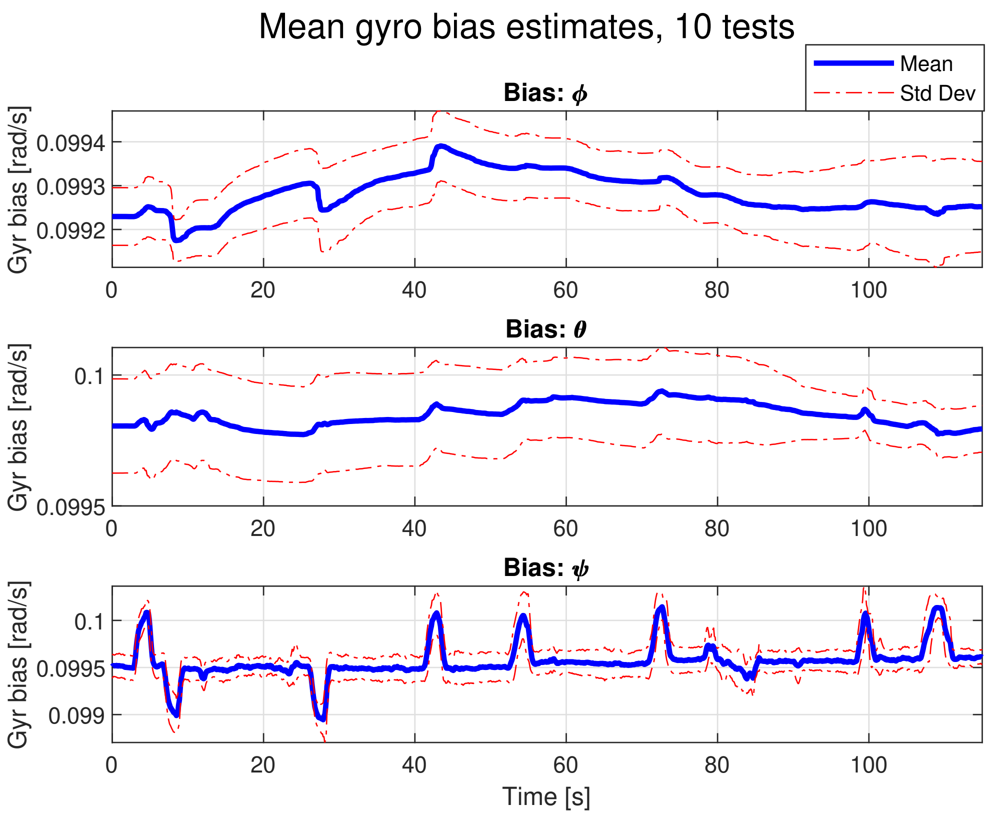



Figure 4.22: Mean values of gyro bias estimates from Hybrid filter with standard deviations

The gyro biases are nicely estimated, this can be seen by the tightness of the standard deviations in the plots.

### 4.1.4 Case 3: Real environment

The path flown in the basement map can be seen in figure 4.23, where the long straight corridors will be referred to as "A" and "B" as seen in the figure.

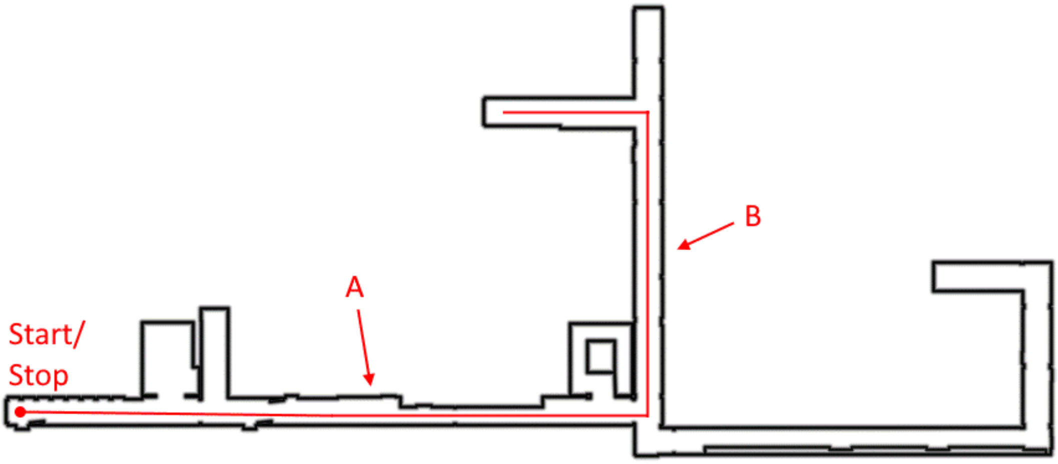



Figure 4.23: The path flown in the basement map

The basement environment test was preformed multiple times, and underlines the filters problem of navigating in long, featureless hallways. 3 out of 10 tests were successful at keeping track of the drone during the whole flight, while the rest lost track. The 3D trajectory of one flight can be seen in figure 4.24.

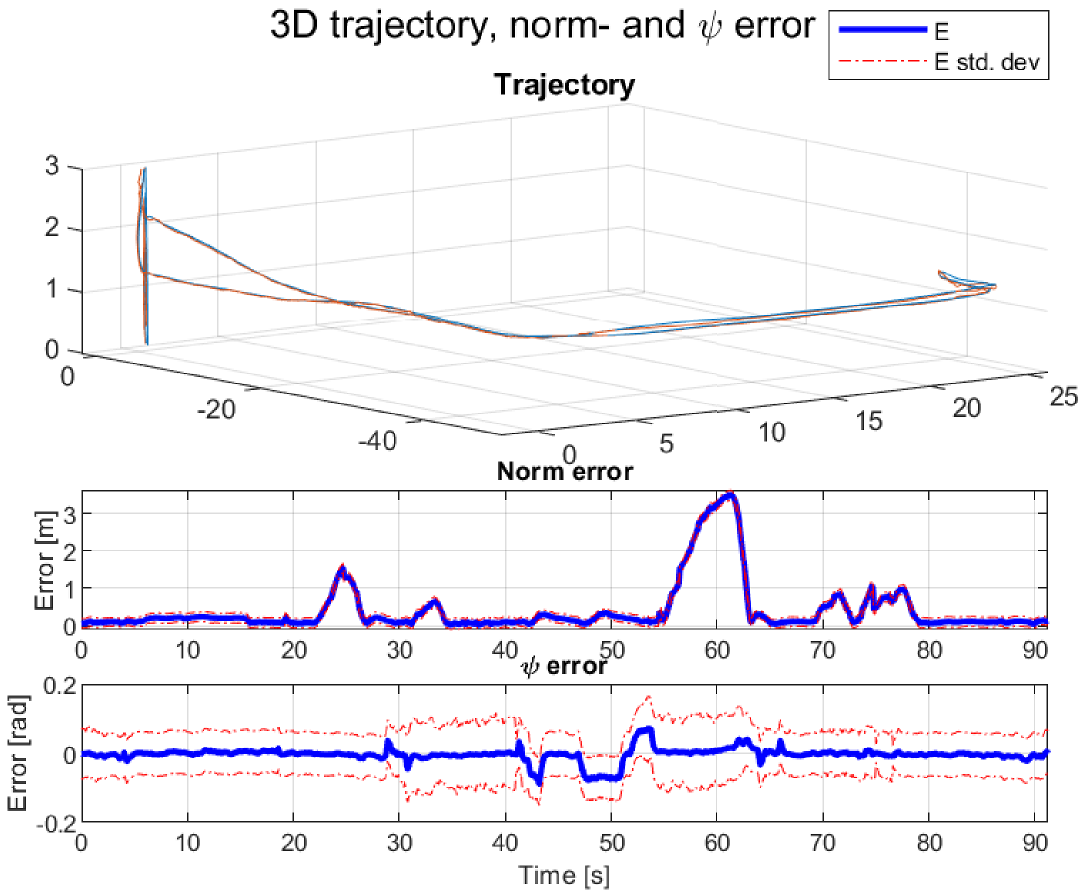



Figure 4.24: 3D Trajectory of true position and estimate from Hybrid filter in the basement environment, with norm- and $\psi$ error over time

It can be seen in the normal error plot in figure 4.24 that the error is much greater in certain parts of the test ($t = 25s,\ t = 60s$), which correspond to movement in the X-direction seen in figure 4.25. This is when the drone flies through corridor B.

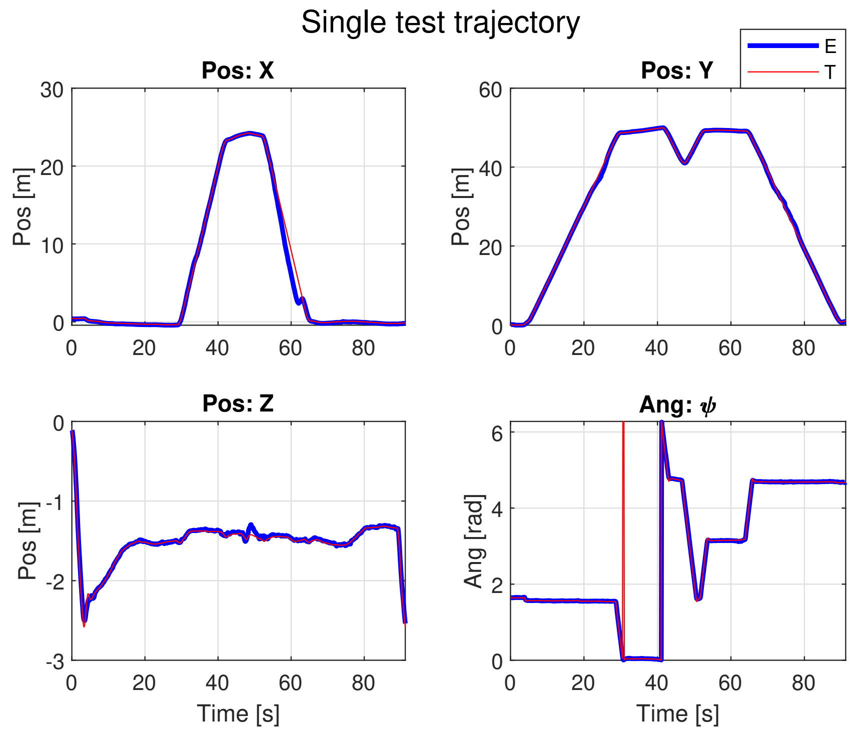



Figure 4.25: Single test position, yaw estimate and true value over time in basement map

Figure 4.26 shows the errors in the angle-estimates for the basement test, displaying low errors in the estimated angles.

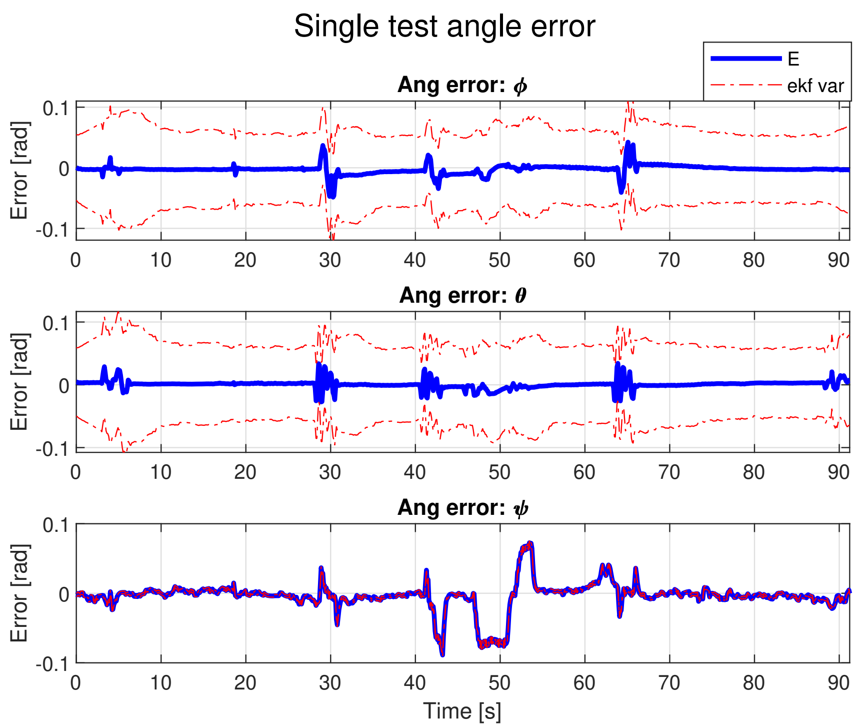



Figure 4.26: Single test angle estimate errors and filter variance for the basement test

Having a closer look at the movement in X, taking the mean and standard deviations for the errors of all successful runs, shows that hallway B is very problematic for the filter, resulting in large errors in estimate shown in figure 4.27.

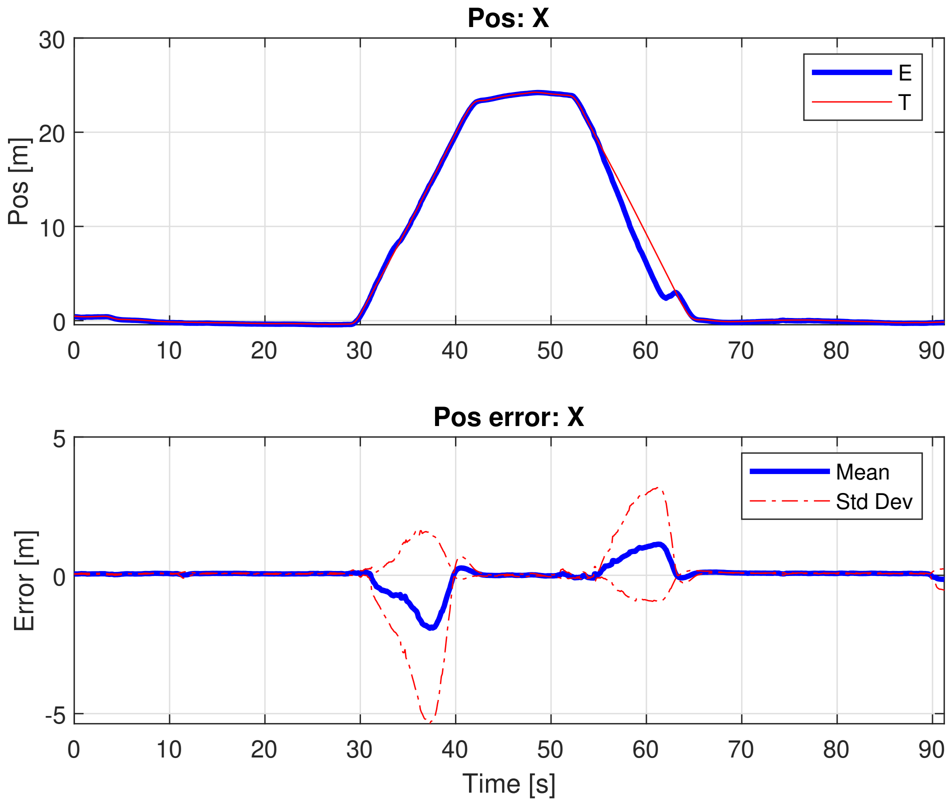



Figure 4.27: The system has difficulties estimating X-position in the long, featureless hallway "B" in the basement map

## 4.2 Hardware platform

A drone platform has been designed and a prototype created, this platform is based on an established PX4 flight controller and a off the shelf drone kit. The Zed mini depth camera was selected as it is a passive "depth sensor" and an off the self component.

### 4.2.1 Drone platform

A drone platform has been designed and built, the platform is based on the S500 drone kit supplied by Holybro. A flight time analysis was preformed based on the manufacturers motor data and the drones design weight. The resulting flight time matched within a good margin, giving confidence in the figure presented for selecting a battery for a desired flight time for the platform.

The designed drone platform is modular in the sense that the both the companion computer and the Zed Mini stereo camera are mounted by use of exchangeable brackets. That is, if a new computer module is to be tested only a redesign of a single bracket is needed, given that the new module has a comparative size to the TX2i. The same idea is implemented for the stereo camera mount, and again, only a single mounting bracket needs to be redesigned to allow for the attachment of a new depth perception sensor.

The drone design also offers the companion computer and flight controller good protection from general handling of the drone, and some crash protection should a crash occur.

(a) Holybro S500 kit

(b) Drone render

(c) Drone assembled

Figure 4.28: Drone kit, design and implementation

The drone was also test flown with dummy payload and an image from the flight can be seen in figure 4.29.

Figure 4.29: Image from test flight

### 4.2.2 Roll and pitch estimation

Even though the complete Hybrid filter has not been tested with data from the real sensors, the filter has been deployed to the TX2 and tested in the simulation. As the filter was already deployed and communication with the flight controller established, testing the roll and pitch estimation in the filter using real IMU data was fairly straight forwards, and the results can be seen in figure 4.30

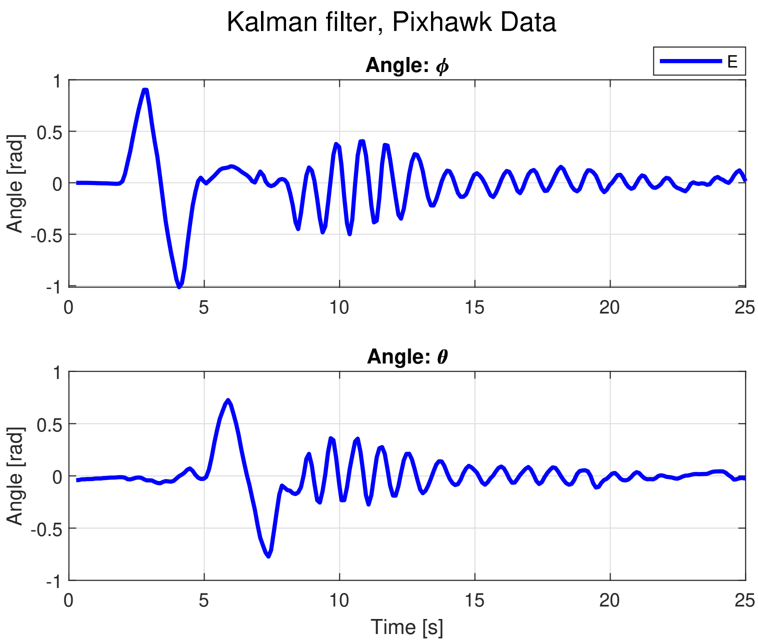
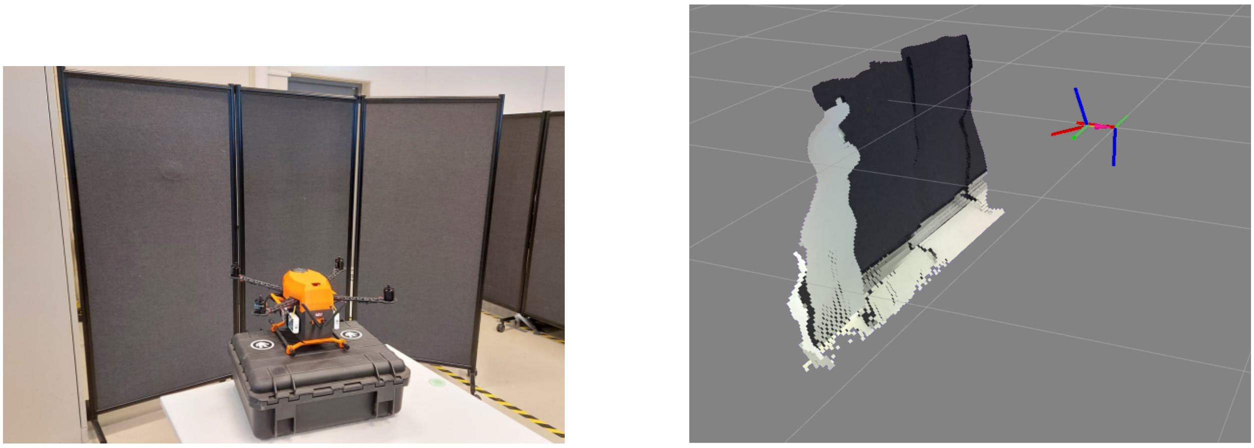



Figure 4.30: Hybrid filter angle estimates using data from Pixhawk flight controller

The tests were performed by rolling and pitching the drone by hand. Therefore true value was not available for the tests, and no conclusions can be drawn about the accuracy of the estimates. However the estimated roll and pitch angles are in the correct directions and have approximately the correct magnitudes, and return to zero when placed on the table.

### 4.2.3 Zed Mini camera point cloud

The Zed mini stereo camera has been connected to the drones computer and a point cloud has been visualized in RViz. The point cloud is correctly leveled using the static transformations. As the kalman filter was not online for the testing of the Zed Mini camera, the dynamic leveling of the point cloud has not yet been tested. But the roll and pitch estimates showed promising results giving confidence that the point cloud would have been correctly leveled on the deployed hardware platform, had all systems been online for the test.



(a) Drone as it captured the point cloud

(b) Point cloud seen in RViz with rgb Values

Figure 4.31: Point cloud captured with Zed Mini stereo camera

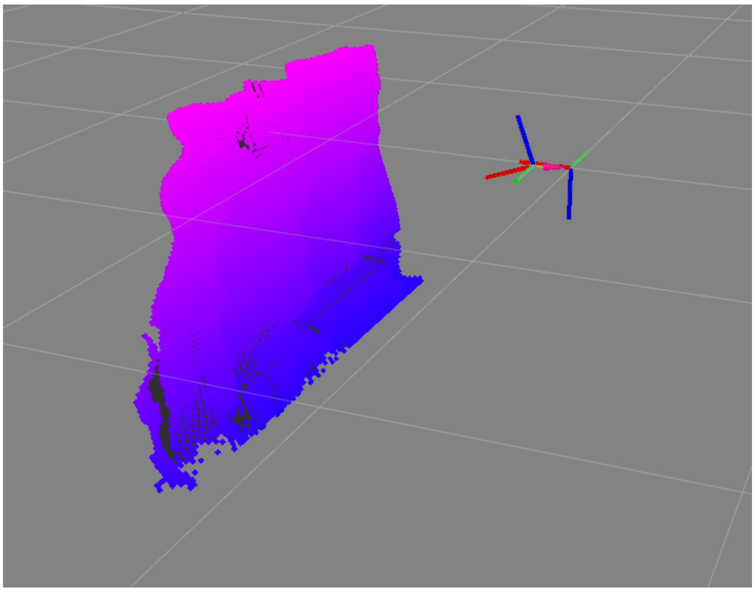


Figure 4.32: Intensity map of point cloud captured with Zed Mini

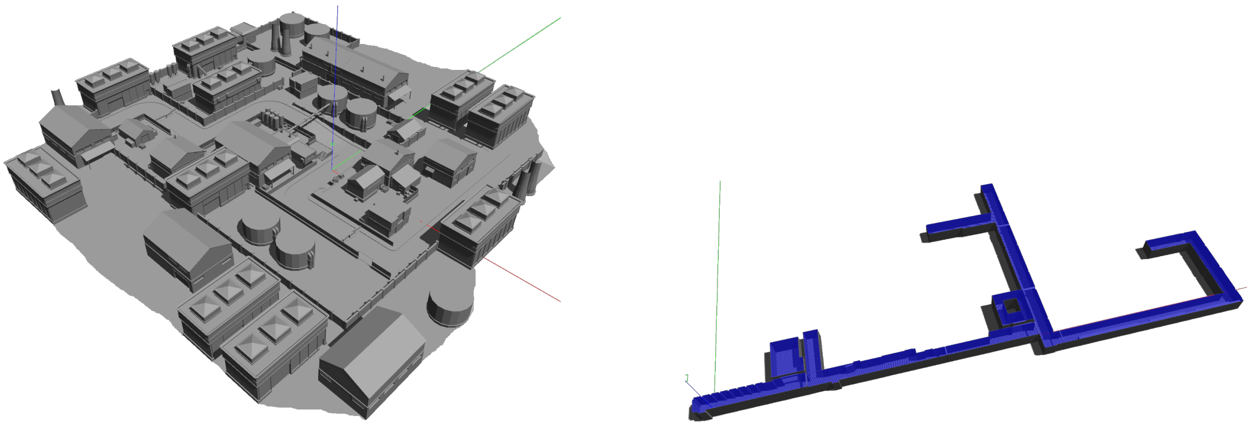
Figure 4.31 and 4.32 displays the point cloud captured form the Zed Mini visualized in RViz, here it can be seen that even though the camera is mounted at an angle, the vertical wall in front of the drone appears level in RViz, demonstrating that the static part of the point cloud leveling functions as intended.

## 4.3 Simulation environment

Two realistic test environments have been created and are available for further use.

The Industrial environment is a good emulation of an industrial multi building complex where drone testing and development can be preformed in a simulated environment. The map is large enough to allow for longer drone flights emulating real inspection flights.

A model of the hallways of the UiA campus Grimstad basement is also created and available for further use in other projects. The Model can also serve as a platform for development of navigation systems not only limited for drone applications.



(a) Industrial environment

(b) UiA Basement environment

Figure 4.33: Gazebo simulation environments

A drone model has been adapted to serve as a simulation drone that is outfitted with the same senors as the designed drone platform. This drone model is also available for further use in other projects and serves a true to life digital twin of the drone platform created.

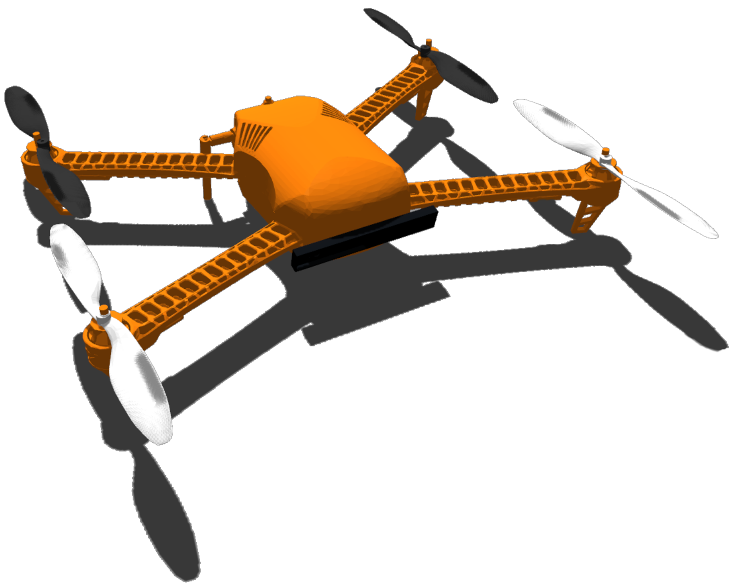


Figure 4.34: Gazebo model of the simulation drone

Page intentionally left blank

# Chapter 5

# Discussions

## 5.1 Singularity in filter

As mentioned in the theory section (2.1) about rotations, the Euler angle orientation representation contains a singularity. This singularity is for the chosen rotation sequence located at $\theta \pm 90[deg]$. This means that an angle estimate based on the IMU leveling can not be acquired around this angle. Further, the transformation matrix used to transform the drones angular rates reported by the gyroscope can not be integrated and give sensible results about the drones Euler angles. For an inspection drone this is acceptable as it is unlikely to ever have to execute a controlled maneuver involving pitching the drone at such an extreme angle. There are ways around this singularity, one approach is proposed in [18], here an Euler angle approach is still used, although an alternative method of integration is proposed when the filter is close to its singularity. A more elegant and perhaps more up-to-date method is to use an Indirect Kalman filter, often also referred to as a Multiplicative Kalman filter. A singularity free filter is developed and demonstrated in the thesis *Singularity-Free Navigation System for an Autonomous Unmanned Aerial Vehicle* [40]. Adopting a similar filtering approach and implementing it with ROS would solve the singularity issue; and as long as the ROS interface is kept the same, swapping the current filter for a singularity-free filter is a simple task[1].

## 5.2 Feedback loop between filters

The chosen strategy of hybrid filtering does not come without problems, as the filters are co-dependent; they can, in some cases, take each other "on a trip". For instance, if the particle filter makes a faulty estimate, causing the solution to jump, the Kalman filter might blame the jump in position on an error in its estimated velocity. This will, in turn, propagate the particles in the wrong direction, forming the basis of a nasty feedback loop; sending the solution off in a completely wrong direction. The system is particularly susceptible to this in featureless environments such as long, straight hallways or open spaces where the particle filter can find multiple positions to fit the sensor's point cloud. This problem is the root cause of the big errors in the X-direction occurring in hallway B illustrated in figure 4.27

The authors propose two methods for fixing the issue.

[1]Developing one on the other hand; is not

### 5.2.1 "Leash method"

The first proposed method is to introduce a modification to the mean propagation velocity use in the particle filter.

The implemented method solely relies on the velocity estimate from the Kalman filter:

$$\mathbf{v}_{\mu} = \mathbf{v}_{ekf} \tag{5.1}$$

Where $\mathbf{v}_{\mu}$ is the average velocity the particle are propagated with, and $\mathbf{v}_{ekf}$ is the velocity estimate from the Kalman filter.

The suggested modification is adding on extra term that will pull the particles towards the estimate in the Kalman filter, in a sense keeping the particle filter propagation step in a leash.

$$\mathbf{v}_{\mu} = \mathbf{v}_{ekf} + k_{leash} \cdot (\mathbf{p}_{kf} - \mathbf{p}_{pf}) \tag{5.2}$$

Where $\mathbf{p}_{kf}$ is the position estimate from the Kalman filter and likewise $\mathbf{p}_{pf}$ is the last outputted position from the Particle filter. The factor $k_{leash}$ will determine how much the particles are pulled towards the solution from the Kalman filter.

### 5.2.2 "Particle based measurement model"

Another option proposed is to eliminate the process of particle propagation completely. That is, every step in the Particle filter, the particles are dispersed around the current position and heading estimate of the Kalman filter. The measurement step and histogram smoothing steps in the particle filter are kept the same. A solution is found in the Particle filter, and this solution is given as a measurement to the Kalman filter, just like in the implemented Hybrid filter.

This will, in a sense, turn the particle filter into a *particle-based measurement model* for the Kalman filter as the particle filter no longer keeps track of the last states in the form of the previous particles. This method would also eliminate the need for re-sampling the particles, as they are now dispersed around the Kalman position estimate for every camera measurement.

## 5.3 Base station sensor package

A focus has been to keep all sensors used for navigation on the drone. This has been done for practical reasons like reducing the needed infrastructure needed to deploy the system and eliminating the need for communication with a ground station in environments where this can sometimes be a challenge. Another reason for this decision was to keep the scope of the project limited.

For *future work* creating and implementing a *base station* could help to stabilize the problem of the filters "going on a trip".

The proposed base station would include a transponder capable of inferring how far away the drone is from the base station; this distance could then be implemented as a probabilistic measurement in the particle filters measurement model. The introduction of such a measurement would reduce the possible positions of the drone to a sphere centered at the *base station*. Alternatively, if a WiFi system is already installed in the industrial area, these transmitters can be utilized in much the same manner. Such a system would also improve the positioning accuracy of the system in long hallways or areas where there are relatively few features within the range of the stereo camera.

It has been discussed in the report that using a barometer for height determination can be risk-ridden in indoor environments where the air pressure can fluctuate due to air-conditioning or the use of a ventilation system. A way around this problem can be to use differential barometry; then, a barometer would be mounted on the base station at a known height. The difference between the two barometer readings can then be used to infer the elevation difference between the drone and the base station, in turn determining the altitude of the drone. This system would be more robust against the use of ventilation systems and fluctuations in the air pressure due to industrial operations as the pressure disturbance would affect both sensors equally.

## 5.4 Proposed alternative sensor package

The current sensor for depth perception only used a single stereo camera pointing forwards and tilted slightly down to get a better estimate of the drone's height. This sensor configuration might be adequate for navigation in *highly featured* environments. But it has been demonstrated that it lacks performance in long hallways or *featureless environments*. Furthermore, only having one forwards facing sensor also makes the drone incapable of perceiving its immediate surroundings on its side and rear; this results in a fairly blind drone and can pose a safety hazard if the drone operates around humans.

Switching to another senor configuration can help alleviate these issues. Switching to a LiDAR-based sensor suite would be beneficial if the operating environment allows for it. The proposed combination of sensors is:

- One front-facing solid-state LiDAR, advancements in mems mirrors have made solid-state LiDARs more available and affordable. A sensor like the Velodyne Velabit would serve this purpose nicely and have a reported cost of approximately 100 USD[2]. The sensor has a narrow field of view of 60 by 10 degrees but a detection range up to 100 meters; this will serve nicely as a front scanning LiDAR.
- One top-mounted 360-degree scanning LiDAR. A *scanning LiDAR* is a device that only contains *one* LiDAR sensor but continuously rotates and scans the distances to objects lying on the scanning disc. This LiDAR will give a good perception of the drones surrounding environment. Crucially it will make collision avoidance possible from the side and rear of the drone, and not only the direction the current stereo camera is observing. Scanning LiDARs range in price but are relatively affordable.
- One down facing single point measurement device, this can be either a LiDAR or an Ultrasonic sensor. Since the new proposed sensor package only senses in a narrow front-facing direction and scans a disc in the horizontal plane, observations of the drone's height will be challenging. This is solved by having a LiDAR (or Ultrasonic sensor) facing down. This will continuously observe the drone's height.

All the observations from the three LiDARs can be combined into one point cloud and handled in much the same way as the point cloud from the stereo camera. It would, however, be wise to rethink what points are sampled and not blindly combine the different sensors into one point cloud and then random draw from it using the methods proposed in this report. A method that guarantees that the down-facing LiDAR is used in every iteration

[2] At the time of writing, reported by blogs and tech news sites, the figure gives a ballpark estimate

would be vise, and the reminding points sampled evenly from the 360 and the front-facing mems LiDAR could be one strategy.

This sensor packages would be comparable in cost to the Zed Mini used in this project and could have some nice benefits as outlined above.

# Chapter 6

# Conclusion

Through the work done in this thesis, an Indoor navigation system has been proposed, deployed to software, and simulated in a HIL simulation. The Hybrid filter utilizes the strengths of both the Kalman filter and the Particle filter and combines their properties to produce a filtering solution that out-preforms the individual filters constituting the system. In addition, all sensors used for navigation and the computer executing the filtering are located on the drone, making it a self-contained autonomous navigation system.

The Hybrid filtering solution proposed is not, however, without its flaws. Namely, the feedback loop between the filters; can lead them both astray. Therefore, two methods have been proposed to solve this issue using the current sensor selection. Alternatively, the proposed solutions in combination with a *base station* or other *rough position estimation* like a WiFi-based system could also aid in resolving the feedback loop.

The navigation system struggles in *feature-poor* environments. This problem was demonstrated in the simulations in the UiA basement environment. Here again the proposed *base station* or a system for *rough position estimation* could resolve this issue. Alternatively, the proposed Lidar-based sensor package could help alleviate these issued by having a longer sensing range.

A hardware platform has also been designed, and a prototype made. The platform was test flown and preforms as expected. The hardware platform will serve as a vessel for further hardware development and a testbed for the Hybrid filter.

Page intentionally left blank

# List of Figures

# List of Tables

Page intentionally left blank

# Bibliography


[1] Markov Andrej Andreevíc and Nagorny Nikolai Makarovíc. *The theory of algorithms*. Kluwer academic, 1988.

[2] Hazuki Arakida et al. "Non-Gaussian data assimilation of satellite-based Leaf Area Index observations with an individual-based dynamic global vegetation model". In: *Nonlinear Processes in Geophysics Discussions* (May 2016), pp. 1–19. DOI: 10.5194/npg-2016-30.

[3] M.S. Arulampalam et al. "A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking". In: *IEEE Transactions on Signal Processing* 50.2 (2002), pp. 174–188. DOI: 10.1109/78.978374.

[4] N. Bergman. "Recursive Bayesian Estimation : Navigation and Tracking Applications". In: 1999.

[5] Steven L. Brunton and J. Nathan Kutz. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*. 1st. USA: Cambridge University Press, 2019. ISBN: 1108422098.

[6] PX4 community. *PX4 User Guide - Gazebo Vehicles*. URL: https://docs.px4.io/master/en/simulation/gazebo_vehicles.html.

[7] The SciPy community. *numpy.arctan2*. URL: https://numpy.org/doc/stable/reference/generated/numpy.arctan2.html.

[8] Andrew J. Davison et al. "MonoSLAM: Real-Time Single Camera SLAM". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 29.6 (2007), pp. 1052–1067. DOI: 10.1109/TPAMI.2007.1049.

[9] Arnaud Doucet, S.J. Godsill, and Christophe Andrieu. "On Sequential Monte Carlo Sampling Methods for Bayesian Filtering". In: *Statistics and Computing* 10 (Apr. 2003). DOI: 10.1023/A:1008935410038.

[10] YINGZHI NING EILIV HÄGG. "Map Representation and LIDAR-Based Vehicle Localization". Master's thesis in the Signal and Systems department, Chalmers. CHALMERS UNIVERSITY OF TECHNOLOGY, Department of Signal and System, 2016.

[11] Felix Endres et al. "An evaluation of the RGB-D SLAM system". In: *Proceedings - IEEE International Conference on Robotics and Automation* (May 2012), pp. 1691–1696. DOI: 10.1109/ICRA.2012.6225199.

[12] Open Source Robotics Foundation. *Gazebo plugins in ROS*. URL: http://gazebosim.org/tutorials?tut=ros_gzplugins.

[13] Open Source Robotics Foundation. *Sensor Noise Model*. URL: http://gazebosim.org/tutorials?tut=sensor_noise.

[14] Oussama El Hamzaoui. "Localisation et cartographie simultanées pour un robot mobile équipé d'un laser à balayage : CoreSLAM. (Simultaneous Localization and Mapping for a mobile robot with a laser scanner : CoreSLAM)". In: 2012.

[15] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[16] Armin Hornung et al. “OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees”. In: *Autonomous Robots* (2013). Software available at `http://octomap.github.com`. DOI: `10.1007/s10514-012-9321-0`. URL: `http://octomap.github.com`.

[17] Aleš Janota et al. “Improving the Precision and Speed of Euler Angles Computation from Low-Cost Rotation Sensor Data”. In: *Sensors* 15.3 (2015), pp. 7016–7039. ISSN: 1424-8220. DOI: `10.3390/s150307016`. URL: `https://www.mdpi.com/1424-8220/15/3/7016`.

[18] Chul Kang. “Euler Angle Based Attitude Estimation Avoiding the Singularity Problem”. In: Aug. 2011, pp. 2096–2102. ISBN: 9783902661937. DOI: `10.3182/20110828-6-IT-1002.01993`.

[19] M. G. Kendall, A. Stuart, and J. K. Ord. *Kendall’s Advanced Theory of Statistics*. USA: Oxford University Press, Inc., 1987. ISBN: 0195205618.

[20] Christian Kerl, Jürgen Sturm, and Daniel Cremers. “Robust odometry estimation for RGB-D cameras”. In: *2013 IEEE International Conference on Robotics and Automation*. 2013, pp. 3748–3754. DOI: `10.1109/ICRA.2013.6631104`.

[21] H.H. Ku. “Notes on the use of propagation of error formulas”. In: *National Bureau of Standards* 70C (1966).

[22] Dmitrii Kutsenko. *RPG FPS Game Assets for PC Mobile Industrial Set v2 Free low-poly 3D model*. URL: `https://www.cgtrader.com/free-3d-models/exterior/industrial/rpg-fps-game-assets-for-pc-mobile-set-v1-rpg-fps-game-assets-for`.

[23] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. “Numba: A LLVM-Based Python JIT Compiler”. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM ’15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052. DOI: `10.1145/2833157.2833162`. URL: `https://doi.org/10.1145/2833157.2833162`.

[24] Tiancheng Li, Miodrag Bolic, and Petar M. Djuric. “Resampling Methods for Particle Filtering: Classification, implementation, and strategies”. In: *IEEE Signal Processing Magazine* 32.3 (2015), pp. 70–86. DOI: `10.1109/MSP.2014.2330626`.

[25] Cheng Liu, Zhaoying Zhou, and Xu Fu. “Attitude determination for MAVs using a Kalman filter”. In: *Tsinghua Science and Technology* 13.5 (2008), pp. 593–597. DOI: `10.1016/S1007-0214(08)70097-X`.

[26] Jun Liu and Rong Chen. “Sequential Monte Carlo Methods for Dynamic Systems”. In: *Journal of the American Statistical Association* 93 (Apr. 1998). DOI: `10.1080/01621459.1998.10473765`.

[27] Yang Liu et al. “A Survey of Depth Estimation Based on Computer Vision”. In: *2020 IEEE Fifth International Conference on Data Science in Cyberspace (DSC)*. 2020, pp. 135–141. DOI: `10.1109/DSC50466.2020.00028`.

[28] Pierre Merriaux et al. “Robust Robot Localization in a Complex Oil and Gas Industrial Environment”. In: *Journal of Field Robotics* 35 (June 2017). DOI: `10.1002/rob.21735`.

[29] Marius Strand Ødven. “SLidar-Based SLAM for AUtonomous Ferry”. Master of Science in Cybernetics and Robotics. Norwegian University of Science and Technology, Department of Engineering Cybernetics, 2019.

[30] Francisco Perez-Grau et al. “Multi-sensor three-dimensional Monte Carlo localization for long-term aerial robot navigation”. In: *International Journal of Advanced Robotic Systems* 14 (Sept. 2017), p. 172988141773275. DOI: `10.1177/1729881417732757`.

[31] *PX4 ROS2 User Guide*. `https://docs.px4.io/master/en/ros/ros2_comm.html`. Accessed: 2021-05-26.

[32] Open Robotics. *navigation*. URL: `http://wiki.ros.org/action/fullsearch/navigation?action=fullsearch&context=180&value=linkto%3A%22navigation%22`.

[33] P. Savage. “REDEFINING GRAVITY AND NEWTONIAN NATURAL MOTION”. In: 2015.

[34] Joan Solà. “Quaternion kinematics for the error-state KF”. In: (Mar. 2015).

[35] Stereolabs3d. *Sensor Noise Model.* URL: https://support.stereolabs.com/hc/en-us/articles/206953229-What-is-the-range-of-the-ZED-and-ZED-Mini-.

[36] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics.* Cambridge, Mass.: MIT Press, 2005. ISBN: 0262201623. URL: http://www.amazon.de/gp/product/0262201623/102-8479661-9831324?v=glance&n=283155&n=507846&s=books&v=glance.

[37] D. Titterton and J. Weston. "Strapdown inertial navigation technology - 2nd edition". In: *Aerospace and Electronic Systems Magazine, IEEE* 20 (Aug. 2005). DOI: 10.1109/MAES.2005.1499250.

[38] Perez Tristan and Thor Fossen. "Kinematic Models for Manoeuvring and Seakeeping of Marine Vessels". In: *Modeling, Identification and Control* 28 (Jan. 2007). DOI: 10.4173/mic.2007.1.3.

[39] Stephan Weiss, Davide Scaramuzza, and Roland Siegwart. "Monocular-SLAM-Based Navigation for Autonomous Micro Helicopters in GPS-Denied Environments". In: *J. Field Robotics* 28 (Nov. 2011), pp. 854–874. DOI: 10.1002/rob.20412.

[40] Erik Falmår Wilthil. "Singularity-Free Navigation System for an Autonomous Unmanned Aerial Vehicle". Master of Science in Cybernetics and Robotics. Norwegian University of Science and Technology, Department of Engineering Cybernetics, 2015.

[41] Zheng-shi Yu and John Crassidis. "Accelerometer Bias Calibration Using Attitude and Angular Velocity Information". In: *Journal of Guidance, Control, and Dynamics* 39 (Jan. 2016), pp. 1–13. DOI: 10.2514/1.G001437.

[42] *Zed SDK.* https://www.stereolabs.com/developers/release/. Accessed: 2021-05-26.

[43] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. "Open3D: A Modern Library for 3D Data Processing". In: *arXiv:1801.09847* (2018).

Page intentionally left blank

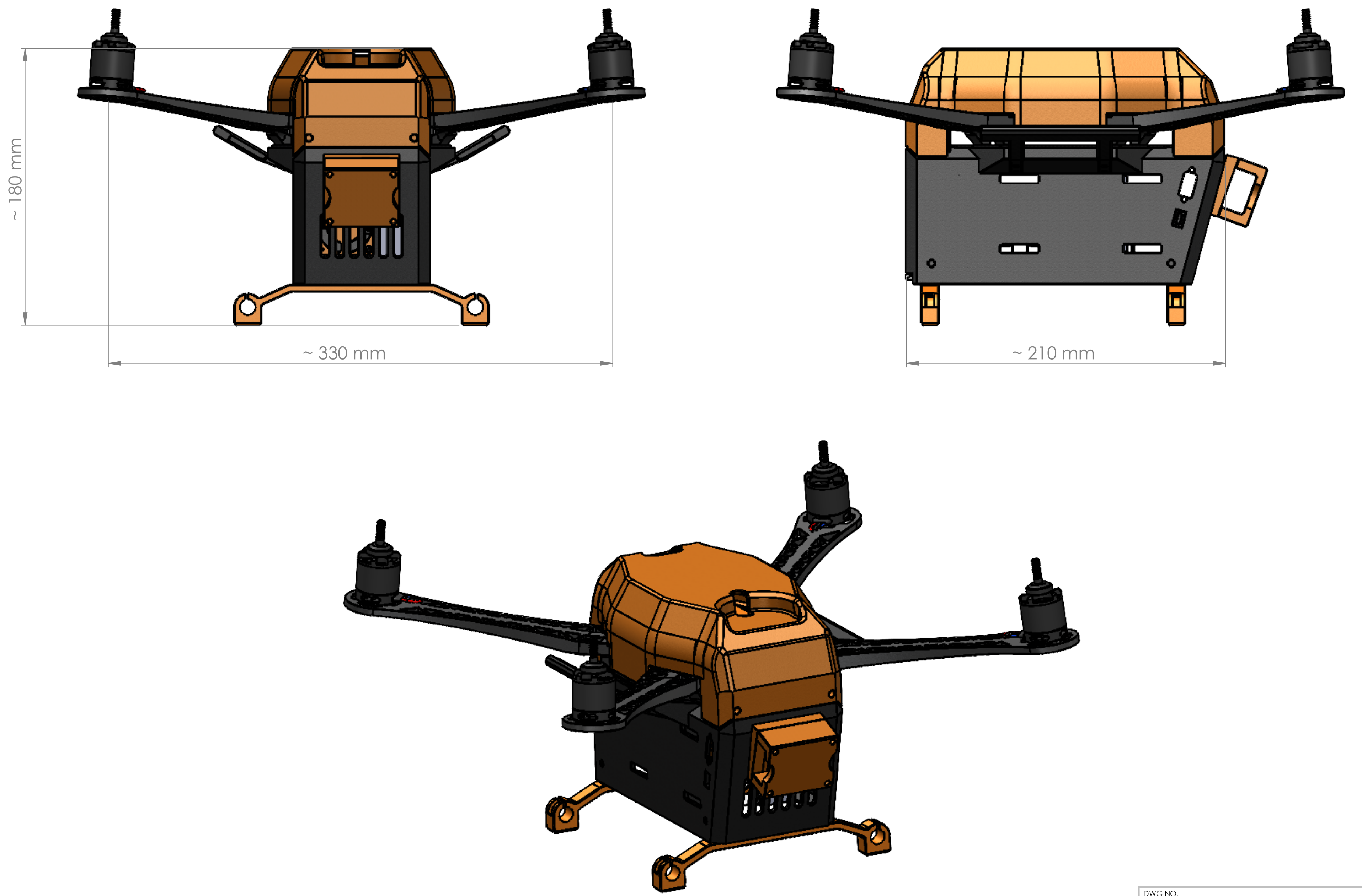
# Appendix A

# Drone drawings

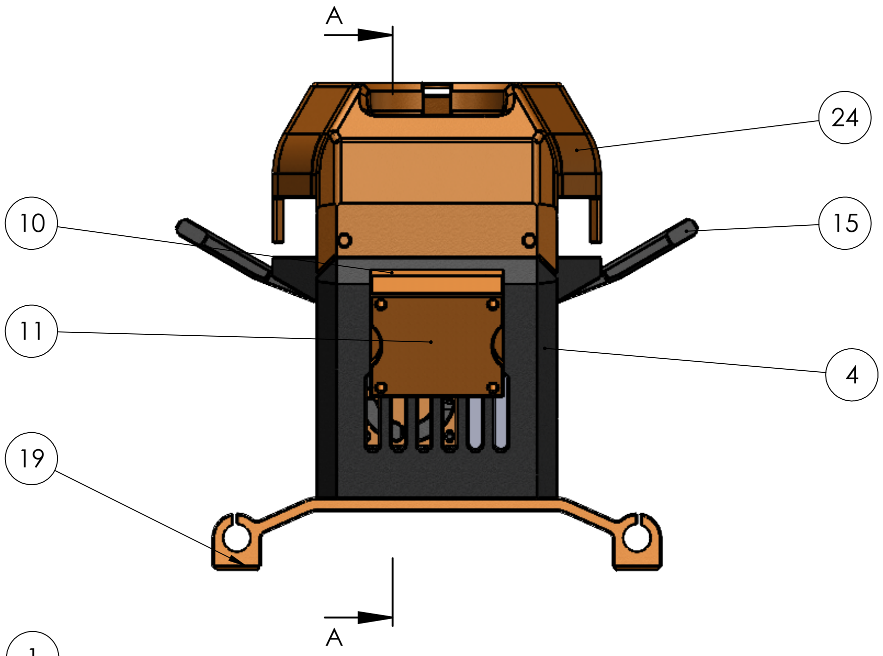
The drones printed parts consists of:

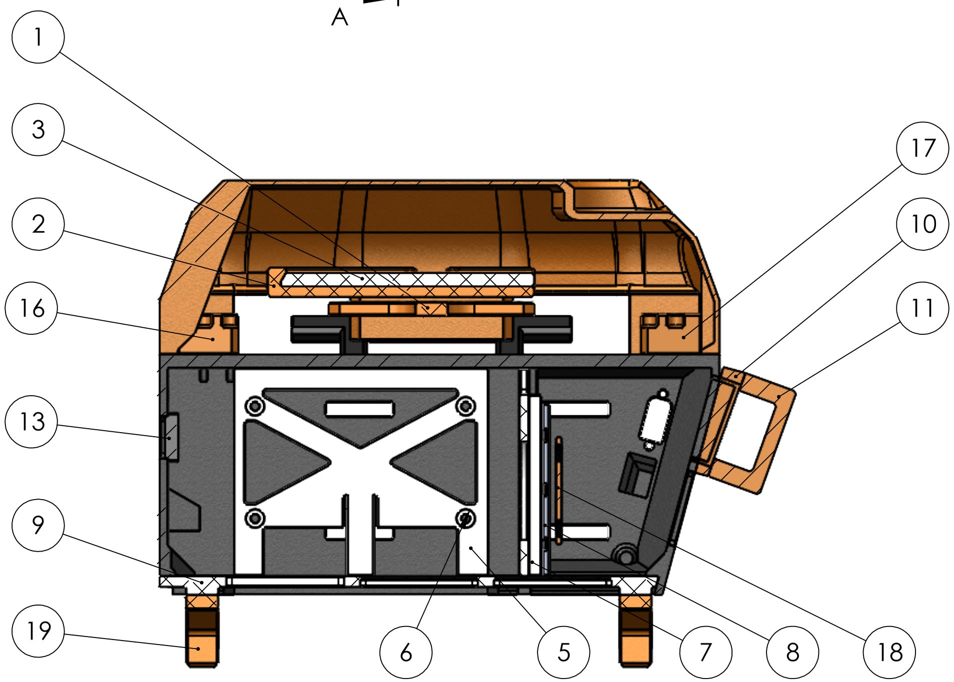
- Dome assembly:
  - dome
  - domeMountingBlock
  - px4MountingFemale
  - px4MountingMale
  - px4MountingStar
- Electronics Tub assembly:
  - antennaMount
  - avionicsBatteryHolder
  - carryHandle
  - electronicsTub
  - electronicsTubBottomPlate
  - electronicsTubDivider
  - fanCover
  - landingSkid
  - sensorMountingInterface
  - tx2MountingSpacer
  - tx2MountingTray
  - zedMiniMountingBracket

Two drawings are presented, one where the main dimensions of the drone can be seen, and a second drawing that describes how the parts are placed in relation to each other and how they are intended to be mounted.




| DWG NO. | Botsy | A3 |
|---|---|---|
| SCALE:2:5 | SHEET 1 OF 2 | |




SECTION A-A

| ITEM NO. | PART NUMBER | DESCRIPTION | QTY. |
|---|---|---|---|
| 1 | px4MountingStar | | 1 |
| 2 | px4MountingFemale | Skrew to S500 Kit | 1 |
| 3 | px4MountingMale | Slides into female | 1 |
| 4 | electronicsTub | Skrew to S500 Kit | 1 |
| 5 | tx2MountingTray | Slide into el-Tub, mount TX2 using spacers | 1 |
| 6 | tx2MountingSpacer | | 4 |
| 7 | electronicsTubDivider | Slide into el-Tub, mount fan and av-bat holder | 1 |
| 8 | avionicsBatteryHolder | Slide into el-Tub-Divider | 1 |
| 9 | electronicsTubBottom Plate | Slide into el-Tub, locks TX2 and el-Tub Divider in place | 1 |
| 10 | sensorMountingInter Face | ZipTie + Mounting tape to el-tub | 1 |
| 11 | zedMiniMounting Bracket | Intefaces with sensorMount, holds Zed mini using 4 screw-pattern | 1 |
| 13 | antennaMount | Screw to el-Tub, holds TX2 WiFi antennas | 1 |
| 14 | carryHandle | Screw to el-Tub | 1 |
| 15 | carryHandle Mirror | Screw to el-tub | 1 |
| 16 | domeMountingBlock | Screw to el-Tub | 2 |
| 17 | domeMountingBlock Mirror | Screw to el-Tub | 2 |
| 18 | FanCover | Screw to el-Tub, into el-Tub Divider to hold fan | 1 |
| 19 | landingSkid | Screw to el-Tub Bottom plate | 2 |
| 24 | dome | Use mounting tape for PX4 GPS + Telem, screws to domeMountingBlock | 1 |

DWG NO.
# Botsy assembly
A3
SCALE:1:2 SHEET 2 OF 2

Page intentionally left blank

# Appendix B

# Source code

## B.1 Software structure and overview

Below is an overview of the file and folder structure of the software packages written for the project.

```
    idl_botsy_pkg
        idl_botsy_pkg
            droneConfiguration.py
            droneGeometricData.py
            filterConfig.py
            __init__.py
            JITdroneConfiguration.py
            softwareConfigutarion.py
        launch
            groundStation_launch.py
            localization_launch.py
        package.xml
        resource
            idl_botsy_pkg
        rviz
            botsy_PointCloud_config.rviz
        setup.cfg
        setup.py
        test
            test_copyright.py
            test_flake8.py
            test_pep257.py
    idl_logger_pkg
        idl_logger_pkg
            __init__.py
            ros_node_logger.py
        package.xml
        resource
            idl_logger_pkg
        setup.cfg
        setup.py
        test
            test_copyright.py
            test_flake8.py
            test_pep257.py
    idl_map_tools
        maps
            map.npy
            metadata.npy
        mapSlices
            HeightSlice0m3.png
        models
            Map1_Origo_InZero.stl
            SimpleMapWithObstacles10x5STL.STL
        README.md
        tools
            likelihoodFieldGenerator.py
            mapSlicer.py
    idl_orientation_pkg
        idl_orientation_pkg
            ekfNode.py
            extendedKalmanFilterCombined.py
            extendedKalmanFilterOrientationControl.py
            extendedKalmanFilterOrientation.py
            extendedKalmanFilterParameters.py
            extendedKalmanFilterPositionControl.py
            extendedKalmanFilterPosition.py
            extendedKalmanFilterSplit.py
            __init__.py
            JITextendedKalmanFilterCombined.py
            JITextendedKalmanFilterOrientationControl.py
            JITextendedKalmanFilterOrientation.py
            JITextendedKalmanFilterParameters.py
            JITextendedKalmanFilterPositionControl.py
            JITextendedKalmanFilterPosition.py
            JITextendedKalmanFilterSplit.py
            testFile.py
        package.xml
```

```
        resource
            idl_orientation_pkg
        setup.cfg
        setup.py
        test
            test_copyright.py
            test_flake8.py
            test_pep257.py
    idl_pf_pkg
        benchmark_scripts
            pfTimerBenchmarks.py
        idl_pf_pkg
            __init__.py
            JitParticleFilterClass.py
            LocalizationFilter.py
            map
                map_IndustrialU8_10cm_10cm.npy
                map_SimpleU8_10cm_10cm.npy
                map_SimpleU8_5cm_5cm.npy
                map_UiABasementU8_10cm_10cm.npy
                metadata_IndustrialU8_10cm_10cm.npy
                metadata_SimpleU8_10cm_10cm.npy
                metadata_SimpleU8_5cm_5cm.npy
                metadata_UiABasementU8_10cm_10cm.npy
            ParticleFilterClass.py
            PF_ros_node.py
            PFTools.py
        package.xml
        resource
            idl_pf_pkg
        setup.cfg
        setup.py
        test
            test_copyright.py
            test_flake8.py
            test_pep257.py
    idl_transform_pkg
        idl_transform_pkg
            droneConfiguration_legacy.py
            earthTransforms.py
            gazeboGroundTruthPublisher.py
            __init__.py
            JITdroneConfiguration_legacy.py
            pointCloudRestamper.py
            pointCloudRestamperZedMini.py
            transformPublisher.py
        package.xml
        README.md
        resource
            idl_transform_pkg
        setup.cfg
        setup.py
        test
            test_copyright.py
            test_flake8.py
            test_pep257.py
```

## B.2 idl_botsy_pkg

### B.2.1 Filter configuration file

This is the filter parameters used for testing the filter.

```python



'''
File is only intended to hold drone configration data
'''

import numpy as np

from idl_botsy_pkg.droneConfiguration import DroneGeometry


### Global variables ###
kalmanFilterConfigurationJitCompile = True
kalmanFilterConfigurationSplit = False



## Structs to hold data
class Vec3(object):

    def __init__(self, x = 0.0,y = 0.0 ,z = 0.0):
        self.x = x
        self.y = y
        self.z = z

    def asNpArray(self, shape = (3,1)):
        return np.array([self.x,self.y,self.z], dtype = np.float32).reshape(shape)


class FilterInitialStates(object):

    def __init__(self):

        ### Initial states
        # Position Related
        self.pos        = Vec3(x=0.0, y=0.0, z=-0.05)
        self.tHeta      = Vec3(x=0.0, y=0.0, z=1.57)

        # Velocity Related
        self.linVel     = Vec3(x=0.0, y=0.0, z=0.0)
        self.angVel     = Vec3(x=0.0, y=0.0, z=0.0)

        # Acceleration Related
        self.linAcc     = Vec3(x=0.0, y=0.0, z=0.0)

        # Sensor Biases
        self.accBias    = Vec3(x=0.0, y=0.0, z=0.0)
        self.omgBias    = Vec3(x=0.0, y=0.0, z=0.0)

        ### Uncertainties
        # Position Related
        self.posCov     = Vec3(x=1.0, y=1.0, z=0.1)
        self.tHetaCov   = Vec3(x=0.01, y=0.01, z=0.25)

        # Velocity Related
        self.linVelCov  = Vec3(x=1.0, y=1.0, z=1.0)
        self.angVelCov  = Vec3(x=0.01, y=0.01, z=0.01)

        # Acceleration Related
        self.linAccCov  = Vec3(x=0.01, y=0.01, z=0.01)

        # Sensor Biases
        self.accBiasCov = Vec3(x=0.00001, y=0.00001, z=0.00001)
        self.omgBiasCov = Vec3(x=0.00001, y=0.00001, z=0.00001)


class FilterConfiguration(object):

    def __init__(self):

        # Configures filter to use gazebo ground truth data, only available in simulation mode
        self.gazeboGT = False
        # Delta imu msg means that the IMU data is integrated between predicts, if False the last received
            imu data is used for predict
        self.deltaImuCum = True
        # Using second order predict, can be beneficial if predict rate is low
        self.secondOrderPredict = True
        # Fixed rate predict, dose predicts at a timer, insted of in IMU callback function
        self.fixedRatePredict = True


### EKF imu settings ###

# Gazebo IMU tuning
class ImuGazeboMain(object):

    def __init__(self):
        # System state predict uncertainty matrix


        self.qPosition    = 0.5
        self.qAngles      = 0.5
        self.qLinVel      = 0.5
```

```
        self.qAngVel      = 0.5
        self.qLinAcc      = 0.01
        self.qAngAcc      = 0.1
        self.qBiasAcc     = 0.0001
        self.qBiasGyro    = 0.00003
        self.qGravity     = 0.00001

        # Measurement uncertainties covariance
        self.rYaw   = 0.1
        self.rPos   = 0.1
        self.rAcc   = 0.01
        self.rGyro  = 0.01
        self.rLevel = 250.0

        # Acc related
        self.levelingWindow = 0.05

        # Sensor localization
        droneGeom = DroneGeometry()
        self.rotMat_bs = droneGeom.rotMat_bu
        self.pos_b_bs = droneGeom. pos_b_bu

        # Position and yaw threshold
        self.posThreshold = 5.0
        self.yawThreshold = 1.57

        # Time stuff
        self.timeMaxDelayPose = 10.0

# Gazebo PX4 imu tuning
class ImuPX4SimMain(object):

    def __init__(self):
        # System state predict uncertainty matrix


        self.qPosition    = 0.5
        self.qAngles      = 0.5
        self.qLinVel      = 0.5
        self.qAngVel      = 0.5
        self.qLinAcc      = 0.01
        self.qAngAcc      = 0.1
        self.qBiasAcc     = 0.0001
        self.qBiasGyro    = 0.00003
        self.qGravity     = 0.00001

        # Measurement uncertainties covariance
        self.rYaw   = 0.1
        self.rPos   = 0.1
        self.rAcc   = 0.01
        self.rGyro  = 0.01
        self.rLevel = 250.0

        # Acc related
        self.levelingWindow = 0.05

        # Sensor localization
        droneGeom = DroneGeometry()
        self.rotMat_bs = droneGeom.rotMat_bu
        self.pos_b_bs = droneGeom. pos_b_bu

        # Position and yaw threshold
        self.posThreshold = 5.0
        self.yawThreshold = 1.57

        # Time stuff
        self.timeMaxDelayPose = 10.0

# Physical PX4 IMU tuning
class ImuPX4RealMain(object):

    def __init__(self):
        # System state predict uncertainty matrix


        self.qPosition    = 0.5
        self.qAngles      = 0.5
        self.qLinVel      = 0.5
        self.qAngVel      = 0.5
        self.qLinAcc      = 0.01
        self.qAngAcc      = 0.1
        self.qBiasAcc     = 0.0005
        self.qBiasGyro    = 0.0001
        self.qGravity     = 0.00001

        # Measurement uncertainties covariance
        self.rYaw   = 0.1
        self.rPos   = 0.1
        self.rAcc   = 0.01
        self.rGyro  = 0.01
        self.rLevel = 50.0

        # Acc related
        self.levelingWindow = 0.10

        # Sensor localization
        droneGeom = DroneGeometry()
        self.rotMat_bs = droneGeom.rotMat_bu
        self.pos_b_bs = droneGeom. pos_b_bu

        # Position and yaw threshold
        self.posThreshold = 5.0
        self.yawThreshold = 1.57

        # Time stuff
```

```
        self.timeMaxDelayPose = 10.0

# Ekf config
class EkfRates(object):

    def __init__(self):
        self.ekfInsPredictHz = 75.0
        self.ekfServiceHz = 2.0
        self.ekfVelBodyPubHz = 2.0
        self.ekfVelLevelPubHz = 15.0
        self.ekfPosNedPubHz = 10.0
        self.ekfOdomNedPubHz = 10.0
        self.ekfSensorBiasPubHz = 10.0


### PF setup ###
class ParticleFilterSetup(object):

    def __init__(self):

        ''' PF operation specific params '''
        # Number of particles to use in PF
        self.numParticles = 1500

        # Threshold for resampling, resampling occurs if effective sample size is lower than threshold
        self.resamplingThreshold = self.numParticles

        # Integration methods, propagation
        self.secondOrderIntegration = False
        self.deltaPositionIntegration = False

        # Parameters for PF covariance
        # Watchdog gain for decay of velocity when no message received
        self.wd_K                = 0.05

        # Maximum value for added propogation std.deviation when no message is received
        self.maxVelStdCtr        = 3.0

        # Const covariance added to propogation
        self.constVar            = Vec3(x=1.0, y=1.0, z=1.0)
        self.constVarPsi         = 0.5

        # Method for pointcloud update step, use square sum method over product
        self.pcUpdateSqSum = False


        ''' Particle pointcloud publisher '''
        # Bools to signify whether or not to publish
        self.pubPFParticlePC     = True

        # Number of particles to publish
        self.pfPCSize            = 50

        # Rate of publish [Hz]
        self.pfPCPublisherRate   = 5


        ''' Sensor parameters '''
        # Helpers
        z_hit = 0.8
        z_rand = 0.2
        z_max = 1.0

        # The sum: z_hit + z_rand/z_max is defined to be equal 1.0
        z_sum = z_hit + z_rand/z_max

        # Setting sensor parameter values
        self.pf_z_hit            = z_hit / z_sum
        self.pf_z_rand           = z_rand / z_sum
        self.pf_z_max            = z_max / z_sum

        # Max range of sensor
        self.pf_sensMaxRange     = 15.0

        # Number of points to use from pointcloud
        self.nPts_PC             = 90


        ''' Parameters for configuring pointcloud downsampling '''
        # Use random points when downsampling
        # Selects nPts randomly, and then checks for duplicates and max_range measurements
        # deletes dupes and max_range measurements from pointcloud being passed on
        self.pcdsRandPoints = True

        # Select particles in a loop, checking each point for validity (range, dupe) before adding to an
    array
        # IF BOTH pcdsLoopSelect AND pcdsRandPoints IS SET TRUE, DEFAULTS TO LOOP CHECK MODE
        self.pcdsLoopSelect = False
        self.pcdsLoopSelMaxLoops = 2*self.nPts_PC


        ''' Histogram smoothing '''
        # Resolution
        self.histRes             = 0.01

        # std deviation
        self.histGaussStdDev     = 0.1


class MapConfig(object):

    def __init__(self):
        '''
            Maps that are currently available:
                UiABasementU8      -   Basement hallways at UiA, uint8 prob
```

```
                IndustrialU8        -    Industrial map found online, uint8 prob
                SimpleU8            -    Simple map with shapes made in solidworks, uint8 prob
        '''

        self.mapName = 'UiABasementU8'
```

### B.2.2 Drone configuration file

```

'''
File is only intended to hold drone configration data

Drone geometry class is intended for static transforms and dynamic transforms between frames
'''

import numpy as np
from idl_botsy_pkg.softwareConfigutarion import simulation

from numba.experimental import jitclass
from numba import int32, float32, boolean, jit, types, typed, typeof  # import the types

'''
https://www.fossen.biz/wiley/ed2/Ch2.pdf
'''

RotSpec = []

@jitclass(RotSpec)
class Rot(object):

    def __init__(self):
        pass

    def rotX(self, arg):
        '''
        Rotates the frame

        input   arg is a scalar
        output  is a np mat with shape (3,3)
        '''

        # pre calculates cos and sine
        c = np.cos(arg)
        s = np.sin(arg)

        return np.array([   [ 1.0, 0, 0],
                            [ 0   , c,-s],
                            [ 0   , s, c]],dtype=np.float32)

    def rotY(self, arg):
        '''
        Rotates the frame

        input   arg is a scalar
        output  is a np mat with shape (3,3)
        '''

        # pre calculates cos and sine
        c = np.cos(arg)
        s = np.sin(arg)

        return np.array([   [ c,    0, s],
                            [ 0, 1.0, 0],
                            [-s,    0, c]],dtype=np.float32)

    def rotZ(self, arg):
        '''
        Rotates the frame

        input   arg is a scalar
        output  is a np mat with shape (3,3)
        '''

        # pre calculates cos and sine
        c = np.cos(arg)
        s = np.sin(arg)

        return np.array([   [c,  -s, 0  ],
                            [s,   c, 0  ],
                            [0,   0, 1.0]],dtype=np.float32)

    def rotXYZ(self, arg):

        '''
        Rotates Rx*Ry*Rz (xyz)

        input   arg is numpy array with shape (3,1)
        output  is a np mat with shape (3,3)
        '''

        # Parses data
        phi     = arg[0, 0]
        theta   = arg[1, 0]
        psi     = arg[2, 0]

        return self.rotX(phi) @ self.rotY(theta) @ self.rotZ(psi)

    def rotZYX(self, arg):

        '''
```

```python
        Rotates Rz*Ry*Rx (zyx)

        input   arg is numpy array with shape (3,1)
        output  is a np mat with shape (3,3)
        '''

        # Parses data
        phi     = arg[0, 0]
        theta   = arg[1, 0]
        psi     = arg[2, 0]

        return self.rotZ(psi) @ self.rotY(theta) @ self.rotX(phi)


# Numba specs for droneGeom class
RotNumbaType = Rot.class_type.instance_type
droneGeomSpec = [
    ('__rotation', RotNumbaType),
    ('tHeta_nm',  float32[:,:]),
    ('tHeta_mn',  float32[:,:]),
    ('pos_n_nm', float32[:,:]),
    ('camera_tilt', float32),
    ('pos_b_bu', float32[:,:]),
    ('pos_b_bv', float32[:,:]),
    ('pos_b_bw', float32[:,:]),
    ('pos_b_bc', float32[:,:]),
    ('tHeta_bu', float32[:,:]),
    ('tHeta_bv', float32[:,:]),
    ('tHeta_bc', float32[:,:]),
    ('tHeta_bw', float32[:,:]),
    ('rotMat_bu', float32[:,:]),
    ('rotMat_bv', float32[:,:]),
    ('rotMat_bc', float32[:,:]),
    ('rotMat_bw', float32[:,:]),
    ('rotMat_nm', float32[:,:]),
]

@jitclass(droneGeomSpec)
class DroneGeometry(object):

    def __init__(self):
        '''
        convention:
        trans_a_bc is a vector resolved in a describing some measure from b to c

        tHeta_bc is a vector of euler angle describing the relation from frame b to frame c

        rotMat_bc is a DCM matrix describing the relation ship from frame b to frame c... i.e c(tHeta_bc) gives a dcm from frame b to c

        rotFun_xy returns a rot mat


        Imu_main frame is denoted u
        Imu_aux frame is denoted w
        Imu_aux_virtual frame is denoted v

        Camera frame is denoted c

        Level frame is denoted l (this frame is a frame with it origin in body, and z in global ned z direction)

        Body (base_link) frame is denoted b
        '''

        # Creates instance of rotation class
        self.__rotation = Rot()


        ########################### World stuff ###########################

        # from global to map
        self.tHeta_nm = np.array([[0.0],[np.pi],[-np.pi/2]], dtype = np.float32)
        self.tHeta_mn = np.array([[0.0],[np.pi],[-np.pi/2]], dtype = np.float32)
        self.pos_n_nm = np.array([[0.0],[0.0],[0.0]], dtype = np.float32)

        if simulation:
            ########################### Drone stuff ###########################
            ### offsets and misc

            # Camera tilt is the angle the camera is tilted downwards, i.e. if pointing towards the sky the tilt angle is negative
            # Also remember to change in .sdf file... not automated... :-( yet
            self.camera_tilt = 15*np.pi/180


            ### Geometric vectors
            # body to imu main
            self.pos_b_bu = np.array([[0.0],[0.0],[-0.1]], dtype = np.float32)

            # body to imu aux
            self.pos_b_bv = np.array([[0.1],[0.0],[0.0]], dtype = np.float32)
            self.pos_b_bw = self.pos_b_bv

            # body to camera
            self.pos_b_bc = np.array([[0.1],[0.0],[0.0]], dtype = np.float32)

            ### Rotation "vector" in sequence zyx with elements [phi theta psi]
            # body to imu main
            self.tHeta_bu = np.array([[0.0],[0.0],[0.0]], dtype = np.float32)

            # body to imu aux (virtual frame)
            self.tHeta_bv = np.array([[0.0],[0.0],[0.0]], dtype = np.float32)

```

```python
            # Body to camera (assumes zed mini imu is in same orientation as camera frame)
            self.tHeta_bc = np.array([[np.pi/2.0-self.camera_tilt],[0.0],[np.pi/2]], dtype = np.float32)
            self.tHeta_bw = self.tHeta_bc

        else:
            ############################ Drone stuff ############################
            ### offsets and misc

            # Camera tilt is the angle the camera is tilted downwards, i.e. if pointing towards the sky the tilt angle is negative
            # Also remember to change in .sdf file... not automated... :-( yet
            self.camera_tilt = 20*np.pi/180


            ### Geometric vectors
            # body to imu main
            self.pos_b_bu = np.array([[-0.01],[0.0],[-0.08]], dtype = np.float32)

            # body to imu aux
            self.pos_b_bv = np.array([[0.1],[0.0],[0.0]], dtype = np.float32)
            self.pos_b_bw = self.pos_b_bv

            # body to camera
            self.pos_b_bc = np.array([[0.135],[-0.03],[0.0]], dtype = np.float32)

            ### Rotation "vector" in sequence zyx with elements [phi theta psi]
            # body to imu main
            self.tHeta_bu = np.array([[0.0],[0.0],[0.0]], dtype = np.float32)

            # body to imu aux (virtual frame)
            self.tHeta_bv = np.array([[0.0],[0.0],[0.0]], dtype = np.float32)


            # Body to camera (assumes zed mini imu is in same orientation as camera frame)
            self.tHeta_bc = np.array([[0+np.pi],[0-self.camera_tilt],[0.0]], dtype = np.float32) # Added 0 + and 0 - to the first two rows because numba was acting up and this apparently fixed it.
            self.tHeta_bw = self.tHeta_bc


        # Associated dcm
        self.rotMat_bu = self.__rotation.rotZYX(self.tHeta_bu)
        self.rotMat_bv = self.__rotation.rotZYX(self.tHeta_bv)
        self.rotMat_bc = self.__rotation.rotZYX(self.tHeta_bc)
        self.rotMat_bw = self.rotMat_bc

        self.rotMat_nm = self.__rotation.rotZYX(self.tHeta_nm)


    def rotFun_nb(self, tHeta_nb):
        '''
        Function to return dcm from ned to body given euler angle vector

        Input with np shape (3,1)
        '''

        # Parsing data
        phi     = tHeta_nb[0, 0]
        theta   = tHeta_nb[1, 0]
        psi     = tHeta_nb[2, 0]


        return self.__rotation.rotX(-phi) @ self.__rotation.rotY(-theta) @ self.__rotation.rotZ(-psi)

    def rotFun_nl(self, tHeta_nb):
        '''
        Function to return dcm from ned to level given euler angle vector

        Input with np shape (3,1)
        '''

        # Parsing data
        psi     = tHeta_nb[2, 0]


        return self.__rotation.rotZ(-psi)

    def rotFun_lb(self, tHeta_nb):
        '''
        Function to return dcm from level to body given euler angle vector

        Input with np shape (3,1)
        '''

        # Parsing data
        phi     = tHeta_nb[0, 0]
        theta   = tHeta_nb[1, 0]


        return self.__rotation.rotX(-phi) @ self.__rotation.rotY(-theta)

    def rotFun_bn(self, tHeta_nb):
        '''
        Function to return dcm from body to ned given euler angle vector

        Input with np shape (3,1)
        '''

        # Parsing data
        phi     = tHeta_nb[0, 0]
        theta   = tHeta_nb[1, 0]
        psi     = tHeta_nb[2, 0]


        return self.__rotation.rotZ(psi) @ self.__rotation.rotY(theta) @ self.__rotation.rotX(phi)

```

```
    def rotFun_bl(self, tHeta_nb):
        '''
        Function to return dcm from body to level given euler angle vector

        Input with np shape (3,1)
        '''

        # Parsing data
        phi     = tHeta_nb[0, 0]
        theta   = tHeta_nb[1, 0]


        return self.__rotation.rotY(theta) @ self.__rotation.rotX(phi)

    def rateTransform_nb(self, arg):
        '''
        Input is a np mat of shape (3,1)

        Output is a np mat of shape (3,3)

        Transforms body rates to global rates
        '''

        # Parses data
        phi     = arg[0][0]
        theta   = arg[1][0]

        # Pre calculating cos, sin and tan
        cx = np.cos(phi)
        cy = np.cos(theta)
        sx = np.sin(phi)
        ty = np.tan(theta)

        # Defines transform
        t = np.array([  [1, sx*ty,  cx*ty],
                        [0,    cx,    -sx],
                        [0, sx/cy,  cx/cy]], dtype = np.float32)

        return t



def main():
    pass




if __name__ == '__main__':
    main()
```

### B.2.3 Software configuration

```



### Simulation ###
simulation = False

### Use PX4 ###
pX4Sensor = True
```

## B.3 idl_orientation_pkg

### B.3.1 Kalman filter node

```python

from llvmlite import binding
binding.set_option("tmp", "-non-global-value-max-name-size=8192")


# Ros imports
import rclpy
from rclpy.node import Node
from rclpy.qos import qos_profile_sensor_data
from rclpy.time import Time

from std_msgs.msg import Bool
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Vector3, TwistWithCovarianceStamped, PoseWithCovarianceStamped



# Configuration
from idl_botsy_pkg.softwareConfigutarion import *

# Filter Specifications
from idl_botsy_pkg.filterConfig import FilterInitialStates
from idl_botsy_pkg.filterConfig import FilterConfiguration
from idl_botsy_pkg.filterConfig import EkfRates

# Selecting what Imu msg definition to use
if pX4Sensor:
    from px4_msgs.msg import SensorCombined as Imu
else:
    from sensor_msgs.msg import Imu as Imu

# Selecting what filter tuning to load
if simulation == True:
    if pX4Sensor == False:
        from idl_botsy_pkg.filterConfig import ImuGazeboMain as InsImuSensorConfig
    else:
        from idl_botsy_pkg.filterConfig import ImuPX4SimMain as InsImuSensorConfig
else:
    from idl_botsy_pkg.filterConfig import ImuPX4RealMain as InsImuSensorConfig



# Selecting wether to use JIT or not (DO NOT! try to run real time without JIT)
from idl_botsy_pkg.filterConfig import kalmanFilterConfigurationJitCompile
from idl_botsy_pkg.filterConfig import kalmanFilterConfigurationSplit

if kalmanFilterConfigurationJitCompile == True:
    if kalmanFilterConfigurationSplit == True:
        from idl_orientation_pkg.JITextendedKalmanFilterSplit import InsEkf
    else:
        from idl_orientation_pkg.JITextendedKalmanFilterCombined import InsEkf
    from idl_orientation_pkg.JITextendedKalmanFilterParameters import InsParameters
    from idl_botsy_pkg.JITdroneConfiguration import DroneGeometry
else:
    if kalmanFilterConfigurationSplit == True:
        from idl_orientation_pkg.extendedKalmanFilterSplit import InsEkf
    else:
        from idl_orientation_pkg.extendedKalmanFilterCombined import InsEkf
    from idl_orientation_pkg.extendedKalmanFilterParameters import InsParameters
    from idl_botsy_pkg.droneConfiguration import DroneGeometry


# Math stuff
from scipy.spatial.transform import Rotation as R
import numpy as np




class EkfOrientationNode(Node):

    def __init__(self):
        super().__init__('ekfOrientationNode')

        ## Setting up INS params
        droneGeom = DroneGeometry()
        insMainParam = InsParameters()
        initStates = FilterInitialStates()
        insConfig = FilterConfiguration()
        imuSensor = InsImuSensorConfig()
        insRates = EkfRates()

        # System state predict uncertainty matrix
        insMainParam.qPosition   = imuSensor.qPosition
        insMainParam.qAngles     = imuSensor.qAngles
        insMainParam.qLinVel     = imuSensor.qLinVel
        insMainParam.qAngVel     = imuSensor.qAngVel
        insMainParam.qLinAcc     = imuSensor.qLinAcc
        insMainParam.qAngAcc     = imuSensor.qAngAcc
        insMainParam.qBiasAcc    = imuSensor.qBiasAcc
        insMainParam.qBiasGyro   = imuSensor.qBiasGyro
        insMainParam.qGravity    = imuSensor.qGravity

        # Measurement uncertainties covariance
        insMainParam.rYaw   = imuSensor.rYaw
        insMainParam.rPos   = imuSensor.rPos
        insMainParam.rAcc   = imuSensor.rAcc
```

```python
        insMainParam.rGyro  = imuSensor.rGyro
        insMainParam.rLevel = imuSensor.rLevel

        # Time delay for pose msg for filter to go offline
        insMainParam.timeMaxDelayPose = imuSensor.timeMaxDelayPose

        # FilterConfig
        insMainParam.deltaImuCum         = insConfig.deltaImuCum
        insMainParam.secondOrderPredict = insConfig.secondOrderPredict
        insMainParam.fixedRatePredict    = insConfig.fixedRatePredict

        # Acc related
        insMainParam.levelingWindow = imuSensor.levelingWindow

        # Pos related
        insMainParam.posThreshold = imuSensor.posThreshold
        insMainParam.yawThreshold = imuSensor.yawThreshold

        # Geometry
        insMainParam.rotMat_bs = imuSensor.rotMat_bs.astype(np.float32)
        insMainParam.pos_b_bs = imuSensor.pos_b_bs.astype(np.float32)

        ## Initial state
        # Geting states from global definition and populating kalman filter specific definition class

        # Pos related
        insMainParam.initState[0:3] = initStates.pos.asNpArray()
        insMainParam.initState[3:6] = initStates.linVel.asNpArray()
        insMainParam.initState[6:9] = initStates.linAcc.asNpArray()
        insMainParam.initState[9:12] = initStates.accBias.asNpArray()
        # Angle related
        insMainParam.initState[12:15] = initStates.tHeta.asNpArray()
        insMainParam.initState[15:18] = initStates.angVel.asNpArray()
        insMainParam.initState[18:21] = initStates.omgBias.asNpArray()

        ## Initial cov
        # Pos related
        insMainParam.initCovVec[0:3] = initStates.posCov.asNpArray()
        insMainParam.initCovVec[3:6] = initStates.linVelCov.asNpArray()
        insMainParam.initCovVec[6:9] = initStates.linAccCov.asNpArray()
        insMainParam.initCovVec[9:12] = initStates.accBiasCov.asNpArray()
        # Angle related
        insMainParam.initCovVec[12:15] = initStates.tHetaCov.asNpArray()
        insMainParam.initCovVec[15:18] = initStates.angVelCov.asNpArray()
        insMainParam.initCovVec[18:21] = initStates.omgBiasCov.asNpArray()


        ## Selecting position msg source for the kalman filter
        if insConfig.gazeboGT:
            posMsgSource = 'gazeboGT/'
        else:
            posMsgSource = 'pf/'

        ## Selecting imu topic to subscribe to
        if pX4Sensor:
            imuMsgSource = '/SensorCombined_PubSubTopic'
            imuMsgQosProfile = qos_profile_sensor_data
        else:
            imuMsgSource = 'sensor/imu_main'
            imuMsgQosProfile = qos_profile_sensor_data


        ## Publish body rates
        self.__pubBodyVel = False

        ## INS Main object
        # Predict rate
        ekfInsPredictHz = insRates.ekfInsPredictHz
        ekfInsPredictDt = 1/ekfInsPredictHz

        # Creating object
        self.__insMain = InsEkf(ekfInsPredictDt)
        self.get_logger().info('Ins Object Created')

        # Calls member function to run all member functions in classes to ge them jit compiled
        self.get_logger().info('Ins Object Jit Compilation Started')
        self.__insMain.jitInit()
        self.get_logger().info('Ins Object Jit Compilation Complete')

        # Sets filter parameters
        self.__insMain.setFilterParameters(insMainParam)
        self.get_logger().info('Ins Parameters set')
        self.get_logger().info('Starting Node work!')


        ## Timers

        # EKF predict timer
        self.__ekfPredictTimer = self.create_timer( ekfInsPredictDt,
                                                    self.__insMainPredictCallback)

        # Service routine timer for ins main
        ekfServiceHz = insRates.ekfServiceHz
        ekfServiceDt = 1.0/ekfServiceHz
        self.__ekfServiceRoutineTimer = self.create_timer(  ekfServiceDt,
                                                            self.__insMainCheckTimers)

        ## Publish timers
        # Velocity body publish timer
        ekfVelBodyPubHz = insRates.ekfVelBodyPubHz
        ekfVelBodyPubDt = 1.0/ekfVelBodyPubHz
        if self.__pubBodyVel == True:
            self.__ekfVelBodyPubTimer = self.create_timer(  ekfVelBodyPubDt,
                                                            self.__insPublishVelBodyTimerCallback)

```

```python
        # Velocity level publish timer
        ekfVelLevelPubHz = insRates.ekfVelLevelPubHz
        ekfVelLevelPubDt = 1.0/ekfVelLevelPubHz
        self.___ekfVelLevelPubTimer = self.create_timer( ekfVelLevelPubDt,
                                                         self.___insPublishVelLevelTimerCallback)

        # Pose publish timer
        ekfPosNedPubHz = insRates.ekfPosNedPubHz
        ekfPosNedPubDt = 1.0/ekfPosNedPubHz
        self.___ekfPosNedPubTimer = self.create_timer(    ekfPosNedPubDt,
                                                         self.___insPublishPosNedTimerCallback)

        # Odom publish timer
        ekfOdomNedPubHz = insRates.ekfOdomNedPubHz
        ekfOdomNedPubDt = 1.0/ekfOdomNedPubHz
        self.___ekfPosNedPubTimer = self.create_timer(    ekfOdomNedPubDt,
                                                         self.___insPublishOdomNedTimerCallback)

        # SensorBias publish timer
        ekfSensorBiasPubHz = insRates.ekfSensorBiasPubHz
        ekfSensorBiasPubDt = 1.0/ekfSensorBiasPubHz
        self.___ekfAccBiasPubTimer = self.create_timer(   ekfSensorBiasPubDt,
                                                         self.___insPublishSensorBiasTimerCallback)


        ## Subscribers
        # Main imu subscriber
        self.___ekfImuSubscriber = self.create_subscription( Imu,
                                                             imuMsgSource,
                                                             self.___insMainImuMeasurementCallback,
                                                             imuMsgQosProfile)

        # Pose subscription
        self.___ekfPoseSubscriber = self.create_subscription(PoseWithCovarianceStamped,
                                                             posMsgSource + 'pose_ned',
                                                             self.___insMainPoseMeasurementCallback,
                                                             10)

        # Pose subscription
        self.___insResetSubscriber = self.create_subscription(    Bool,
                                                                  'ins/system/reset',
                                                                  self.___insResetStatesCallback,
                                                                  10)

        ## Publisher
        # Velocity body publisher
        self.___ekfVelBodyPublisher = self.create_publisher( TwistWithCovarianceStamped,
                                                             'ekf/vel_body',
                                                             10)

        # Velocity level publisher
        self.___ekfVelLevelPublisher = self.create_publisher( TwistWithCovarianceStamped,
                                                             'ekf/vel_level',
                                                             10)

        # Position publisher
        self.___ekfPosNedPublisher = self.create_publisher(   PoseWithCovarianceStamped,
                                                             'ekf/pose_ned',
                                                             10)

        # Position publisher
        self.___ekfOdomNedPublisher = self.create_publisher( Odometry,
                                                             'botsy/odom/body',
                                                             qos_profile_sensor_data)

        # Position publisher
        self.___insStatePublisher = self.create_publisher(    Bool,
                                                             'ins/system/ekf_online',
                                                             10)

        # Accelerometer bias publisher
        self.___insSensorBiasPublisher = self.create_publisher(   TwistWithCovarianceStamped,
                                                                  'ekf/sensor_bias',
                                                                  10)

    ## Timer callback functions
    # Predict
    def __insMainPredictCallback(self):
        '''
        Function to run predict on timer callback
        '''
        # Predicting state
        self.___insMain.predict()

    # Service routine for ins filter
    def __insMainCheckTimers(self):
        '''
        Function to check elapse of timers in ins filter object
        '''

        # Checking timings in filter
        timeOfCall = self.get_clock().now().to_msg()
        timeNsec = Time.from_msg(timeOfCall).nanoseconds
        timeSec = timeNsec*10**(-9)

        self.___insMain.checkTiming(timeSec)


    ## subscriber callbacks

    # Pose
    def __insMainPoseMeasurementCallback(self, msg):
        '''
        Function to set insMain position from pf
        '''
```

```python

        # Gets position
        pos_n_nb = msg.pose.pose.position

        # Gets Theta angle
        theta = msg.pose.pose.orientation.z

        # Stitching together to a pose vector
        pose = np.array([[pos_n_nb.x],[pos_n_nb.y],[pos_n_nb.z],[theta]], dtype = np.float32)

        # Gets covariance
        varFromMsg = msg.pose.covariance.reshape(6,6)

        # Formats to "filter from"
        rPose = np.zeros((4,4), dtype = np.float32)
        # Position related
        rPose[0:3, 0:3] = varFromMsg[0:3, 0:3]
        # Theta
        rPose[3, 3] = varFromMsg[5, 5]

        # gets time
        timeNsec = Time.from_msg(msg.header.stamp).nanoseconds
        timeSec = timeNsec*10**(-9)

        self.__insMain.setPoseMeasurementWithCovariance(pose, rPose, timeSec)

    # main imu
    if pX4Sensor:
        def __insMainImuMeasurementCallback(self, msg):
            '''
            Function to set insMain imu data form sensor imu
            '''

            imuData, timeSec = self.__sensorCombinedMsg2VecAndTime(msg)

            # Sending to ins system

            self.__insMain.setImuMeasurement(imuData, timeSec)
    else:
        def __insMainImuMeasurementCallback(self, msg):
            '''
            Function to set insMain imu data form sensor imu
            '''

            imuData, timeSec = self.__imuMsg2vecAndTime(msg)

            # Sending to ins system

            self.__insMain.setImuMeasurement(imuData, timeSec)

    # main imu IMU mag unpacker function
    def __imuMsg2vecAndTime(self, msg):
        '''
        Function to unpack gazbo/ros Imu msg data to vector and time format
        '''
        ## Parsing data
        # Gets sensor data
        acc = msg.linear_acceleration
        omg = msg.angular_velocity

        # gets time
        timeOfCall = self.get_clock().now().to_msg()
        timeNsec = Time.from_msg(timeOfCall).nanoseconds
        timeSec = timeNsec*10**(-9)


        # Populating np array
        imuData = np.array([[acc.x],[acc.y],[acc.z],[omg.x],[omg.y],[omg.z]], dtype = np.float32)

        return imuData, timeSec

    def __sensorCombinedMsg2VecAndTime(self, msg):
        '''
        Function to unpack PX4 sensorCombined msg data to vector and time format
        '''

        ## Parsing data
        # Gets sensor data
        acc = msg.accelerometer_m_s2
        omg = msg.gyro_rad

        # Gets time
        timeMsec = msg.timestamp
        timeSec = timeMsec*10**(-6)

        imuData = np.array([[acc[0]],[acc[1]],[acc[2]],[omg[0]],[omg[1]],[omg[2]]], dtype = np.float32)

        return imuData, timeSec

    # Ins state reset
    def __insResetStatesCallback(self, msg):
        '''
        Function to reset states in kalman filter
        '''

        if msg.data:
            self.__insMain.resetFilter()
            self.get_logger().info('Filter reset')


    ## Publish timer callbacks
    # Vel body
    def __insPublishVelBodyTimerCallback(self):
        '''
        Function to publish velocity msg on timer callback
```

```
        '''

        # Gets linear velocity and associated covariance
        if self.___insMain.getPosFilterOnlineState():
            vel_b_bn, rVel_b_bn = self.___insMain.getLinearVelocityBodyWithCovariance()
        else:
            vel_b_bn = np.zeros((3,1), dtype = np.float32)
            rVel_b_bn = np.zeros((3,3), dtype = np.float32)

        # Gets linear velocity and associated covariance
        omg_b_bn, rOmg_b_bn = self.___insMain.getAngularVelocityBodyWithCovariance()

        # Creates vel and omg for msg
        velForMsg_b_bn = vel_b_bn.astype(np.float64)
        omgForMsg_b_bn = omg_b_bn.astype(np.float64)

        # Creates covariance vector for msg format
        cov = np.zeros((6,6), dtype = np.float64)
        cov[0:3, 0:3] = rVel_b_bn
        cov[3:6, 3:6] = rOmg_b_bn
        covVec = cov.reshape(36).astype(np.float64)

        # Creating empty msg to populate
        msg = TwistWithCovarianceStamped()

        # Populating msg
        msg.header.stamp = self.get_clock().now().to_msg()
        msg.header.frame_id = 'body'

        msg.twist.covariance = covVec

        msg.twist.twist.linear.x = velForMsg_b_bn[0,0]
        msg.twist.twist.linear.y = velForMsg_b_bn[1,0]
        msg.twist.twist.linear.z = velForMsg_b_bn[2,0]

        msg.twist.twist.angular.x = omgForMsg_b_bn[0,0]
        msg.twist.twist.angular.y = omgForMsg_b_bn[1,0]
        msg.twist.twist.angular.z = omgForMsg_b_bn[2,0]

        # Publishes msg
        self.___ekfVelBodyPublisher.publish(msg)

    # Vel level
    def ___insPublishVelLevelTimerCallback(self):
        '''
        Function to publish velocity msg on timer callback
        '''

        # Gets linear velocity and associated covariance
        if self.___insMain.getPosFilterOnlineState():
            vel_l_bn, rVel_l_bn = self.___insMain.getLinearVelocityLevelWithCovariance()
        else:
            vel_l_bn = np.zeros((3,1), dtype = np.float32)
            rVel_l_bn = np.zeros((3,3), dtype = np.float32)

        # Gets linear velocity and associated covariance
        omg_l_bn, rOmg_l_bn = self.___insMain.getAngularVelocityLevelWithCovariance()

        # Creates vel and omg for msg
        velForMsg_l_bn = vel_l_bn.astype(np.float64)
        omgForMsg_l_bn = omg_l_bn.astype(np.float64)

        # Creates covariance vector for msg format
        cov = np.zeros((6,6), dtype = np.float64)
        cov[0:3, 0:3] = rVel_l_bn
        cov[3:6, 3:6] = rOmg_l_bn
        covVec = cov.reshape(36).astype(np.float64)

        # Creating empty msg to populate
        msg = TwistWithCovarianceStamped()

        # Populating msg
        msg.header.stamp = self.get_clock().now().to_msg()
        msg.header.frame_id = 'level'

        msg.twist.covariance = covVec

        msg.twist.twist.linear.x = velForMsg_l_bn[0,0]
        msg.twist.twist.linear.y = velForMsg_l_bn[1,0]
        msg.twist.twist.linear.z = velForMsg_l_bn[2,0]

        msg.twist.twist.angular.x = omgForMsg_l_bn[0,0]
        msg.twist.twist.angular.y = omgForMsg_l_bn[1,0]
        msg.twist.twist.angular.z = omgForMsg_l_bn[2,0]

        # Publishes msg
        self.___ekfVelLevelPublisher.publish(msg)

    # Pose ned
    def ___insPublishPosNedTimerCallback(self):
        '''
        Function to publish velocity msg on timer callback
        '''

        # Gets position and associated covariance
        if self.___insMain.getPosFilterOnlineState():
            pos_n_bn, rPos_n_bn = self.___insMain.getPositionWithCovariance()
        else:
            pos_n_bn = np.zeros((3,1), dtype = np.float32)
            rPos_n_bn = np.zeros((3,3), dtype = np.float32)

        # Gets orientation and associated covariance
        tHeta_nb, rTHeta_nb = self.___insMain.getOrientationWithCovariance()

        # Creates vel and omg for msg
```

```
        posForMsg_n_bn = pos_n_bn.astype(np.float64)
        tHetaForMsg_nb = tHeta_nb.astype(np.float64)

        # Creates covariance vector for msg format
        cov = np.zeros((6,6), dtype = np.float64)
        cov[0:3, 0:3] = rPos_n_bn
        cov[3:6, 3:6] = rTHeta_nb
        covVec = cov.reshape(36).astype(np.float64)

        # Creating empty msg to populate
        msg = PoseWithCovarianceStamped()

        # Populating msg
        msg.header.stamp = self.get_clock().now().to_msg()
        msg.header.frame_id = 'ned'

        msg.pose.covariance = covVec

        msg.pose.pose.position.x = posForMsg_n_bn[0,0]
        msg.pose.pose.position.y = posForMsg_n_bn[1,0]
        msg.pose.pose.position.z = posForMsg_n_bn[2,0]

        msg.pose.pose.orientation.x = tHetaForMsg_nb[0,0]
        msg.pose.pose.orientation.y = tHetaForMsg_nb[1,0]
        msg.pose.pose.orientation.z = tHetaForMsg_nb[2,0]
        if self.__insMain.getPosFilterOnlineState():
            msg.pose.pose.orientation.w = -2.0
        else:
            msg.pose.pose.orientation.w = -3.0

        # Publishes msg
        self.__ekfPosNedPublisher.publish(msg)

    # Odom ned
    def __insPublishOdomNedTimerCallback(self):
        '''
        Function to publish location to odom msg to tf listener node
        '''

        # Cerates a msg and populates
        msg = Odometry()

        msg.header.stamp = self.get_clock().now().to_msg()


        # Populating Pose part of msg
        # Gets position and associated covariance
        if self.__insMain.getPosFilterOnlineState():
            pos_n_bn, rPos_n_bn = self.__insMain.getPositionWithCovariance()
        else:
            pos_n_bn = np.zeros((3,1), dtype = np.float32)
            rPos_n_bn = np.zeros((3,3), dtype = np.float32)

        # Gets orientation and associated covariance
        tHeta_nb, rTHeta_nb = self.__insMain.getOrientationWithCovariance()
        tHeta_nb = np.array([tHeta_nb[2,0], tHeta_nb[1,0], tHeta_nb[0,0]])
        rotObj = R.from_euler('ZYX', tHeta_nb.reshape(3), degrees= False)
        quat_nb = rotObj.as_quat()

        # Creates vel and omg for msg
        posForMsg_n_bn = pos_n_bn.astype(np.float64)

        # Creates covariance vector for msg format
        cov = np.zeros((6,6), dtype = np.float64)
        cov[0:3, 0:3] = rPos_n_bn
        cov[3:6, 3:6] = rTHeta_nb
        covVec = cov.reshape(36).astype(np.float64)

        msg.pose.covariance = covVec

        msg.pose.pose.position.x = posForMsg_n_bn[0,0]
        msg.pose.pose.position.y = posForMsg_n_bn[1,0]
        msg.pose.pose.position.z = posForMsg_n_bn[2,0]

        msg.pose.pose.orientation.x = quat_nb[0]
        msg.pose.pose.orientation.y = quat_nb[1]
        msg.pose.pose.orientation.z = quat_nb[2]
        msg.pose.pose.orientation.w = quat_nb[3]

        '''
        # Populating Twist Part of msg
        # Gets linear velocity and associated covariance
        vel_b_bn, rVel_b_bn = self.__insMain.getLinearVelocityBodyWithCovariance()

        # Gets linear velocity and associated covariance
        omg_b_bn, rOmg_b_bn = self.__insMain.getAngularVelocityBodyWithCovariance()

        # Creates vel and omg for msg
        velForMsg_b_bn = vel_b_bn.astype(np.float64)
        omgForMsg_b_bn = omg_b_bn.astype(np.float64)

        # Creates covariance vector for msg format
        cov = np.zeros((6,6), dtype = np.float64)
        cov[0:3, 0:3] = rVel_b_bn
        cov[3:6, 3:6] = rOmg_b_bn
        covVec = cov.reshape(36).astype(np.float64)

        # Populating msg
        msg.twist.covariance = covVec

        msg.twist.twist.linear.x = velForMsg_b_bn[0,0]
        msg.twist.twist.linear.y = velForMsg_b_bn[1,0]
        msg.twist.twist.linear.z = velForMsg_b_bn[2,0]

        msg.twist.twist.angular.x = omgForMsg_b_bn[0,0]
```

```
        msg.twist.twist.angular.y = omgForMsg_b_bn[1,0]
        msg.twist.twist.angular.z = omgForMsg_b_bn[2,0]
        '''

        # Publishes msg
        self.__ekfOdomNedPublisher.publish(msg)

    # Acc bias
    def __insPublishSensorBiasTimerCallback(self):
        '''
        Function to publish acc bias
        '''

        # Gets linear velocity and associated covariance
        if self.__insMain.getPosFilterOnlineState():
            accBias, rAccBias = self.__insMain.getAccBiasWithCovariance()
        else:
            accBias = np.zeros((3,1), dtype = np.float32)
            rAccBias = np.zeros((3,3), dtype = np.float32)

        # Gets linear velocity and associated covariance
        gyroBias, rGyroBias = self.__insMain.getGyroBiasWithCovariance()

        # Creates vel and omg for msg
        accBiasMsg = accBias.astype(np.float64)
        gyroBiasMsg = gyroBias.astype(np.float64)

        # Creates covariance vector for msg format
        cov = np.zeros((6,6), dtype = np.float64)
        cov[0:3, 0:3] = rAccBias
        cov[3:6, 3:6] = rGyroBias
        covVec = cov.reshape(36).astype(np.float64)

        # Creating empty msg to populate
        msg = TwistWithCovarianceStamped()

        # Populating msg
        msg.header.stamp = self.get_clock().now().to_msg()
        msg.header.frame_id = 'bias'

        msg.twist.covariance = covVec

        msg.twist.twist.linear.x = accBiasMsg[0,0]
        msg.twist.twist.linear.y = accBiasMsg[1,0]
        msg.twist.twist.linear.z = accBiasMsg[2,0]

        msg.twist.twist.angular.x = gyroBiasMsg[0,0]
        msg.twist.twist.angular.y = gyroBiasMsg[1,0]
        msg.twist.twist.angular.z = gyroBiasMsg[2,0]

        # Publishing msg
        self.__insSensorBiasPublisher.publish(msg)



def main(args=None):
    rclpy.init(args=args)

    # Creates INS node
    insSystem = EkfOrientationNode()

    # Spins node to keep it alive
    rclpy.spin(insSystem)

    # Destroys node on shutdown
    insSystem.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

### B.3.2 Kalman filter object

```



import numpy as np


# Import Drone config class
from idl_botsy_pkg.JITdroneConfiguration import DroneGeometry
from idl_orientation_pkg.JITextendedKalmanFilterParameters import InsParameters


# import Numba stuff
from numba.experimental import jitclass
from numba import int32, float32, float64, boolean, jit, types, typed, typeof  # import the types

droneGeomNumbaType = DroneGeometry.class_type.instance_type
InsEkfSpecs = [
        ('___nStates', int32),
        ('online', boolean),
        ('___secondOrderPredict', boolean),
        ('___fixedPredictRate', boolean),
        ('___fixedPredictRateDt', float32),
        ('___stateInitialization', boolean),
        ('___posThreshold', float32),
      ('___accHighThreshold', float32),
        ('___gravity', float32),
        ('___gVec', float32[:,:]),
        ('___gravityState', boolean),
        ('___lowThreshold', float32),
        ('___highThreshold', float32),
        ('___yawThreshold', float32),
        ('___x', float32[:,:]),
        ('___xInit', float32[:,:]),
        ('___a', float32[:,:]),
        ('___w', float32[:,:]),
        ('___aCum', float32[:,:]),
        ('___wCum', float32[:,:]),
        ('___accMeasure_e', float32[:,:]),
        ('___xLast', float32[:,:]),
        ('___aLast', float32[:,:]),
        ('___wLast', float32[:,:]),
        ('___P', float32[:,:]),
        ('___PInit', float32[:,:]),
        ('___Q', float32[:,:]),
        ('___R_pos', float32[:,:]),
        ('___R_yaw', float32),
        ('___R_leveling', float32),
        ('___hPosition', float32[:,:]),
        ('___hYaw', float32[:,:]),
        ('___hLeveling', float32[:,:]),
        ('___droneGeom', droneGeomNumbaType),
        ('___rotMat_bs', float32[:,:]),
        ('___pos_b_bs', float32[:,:]),
        ('___rotMat_bn', float32[:,:]),
        ('___t_nb', float32[:,:]),
        ('___timeMaxDiffPose', float64),
        ('___timeNewPoseUpdate', float64),
        ('___timeDiffPose', float64),
        ('___timeLastImuMsg', float64),
        ('___maxDt', float64),
        ('___imuPredictDt', float32),
        ('___deltaImuCounter', int32),
        ('___deltaImuDt', float32),
        ('___deltaImuCum', boolean),
        ('___filterIsNotReSet', boolean),
]



# Holistic INS ekf class
@jitclass(InsEkfSpecs)
class InsEkf(object):

    def __init__(self, dt):

        # Kalman filter
        ## System Number of states
        self.___nStates = 18

        ## Filter settings

        # Filter state
        self.online = True

        # Second order propagation
        self.___secondOrderPredict = True

        # Fixed predict rate
        self.___fixedPredictRate = True
        self.___fixedPredictRateDt = np.float32(dt)

        # Initialization of filter
        self.___stateInitialization = 0

        # Position and ang threshold
        self.___posThreshold = 1.0

        self.___accHighThreshold = 0.01     # percent of g
        self.___gravity = np.float(9.81)
        self.___gVec = np.array([[np.float32(0.0)],[np.float32(0.0)],[-self.___gravity]], dtype = np.float32
    )
        self.___gravityState = False
```

```python

        self.__lowThreshold = self.__gravity*(1-self.__accHighThreshold)
        self.__highThreshold = self.__gravity*(1+self.__accHighThreshold)

        self.__yawThreshold = 1.57

        ## System state vector
        # [0, 1, 2,  3, 4, 5,    6,   7,   8,    9,    10,  11,   12,  13,  14,  15, 16, 17]
        # [x, y, z,  u, v, w,  abx, aby, abz,  phi, theta, psi,  wbx, wby, wbz,  gx, gy, gz]
        self.__x = np.zeros((self.__nStates,1), dtype=np.float32)
        # Initializes gravity vector
        self.__x[17:0] = -self.__gravity
        self.__xInit = self.__x

        # Acceleration
        self.__a = np.zeros((3,1), dtype = np.float32)
        self.__w = np.zeros((3,1), dtype = np.float32)
        self.__aCum = np.zeros((3,1), dtype = np.float32)
        self.__wCum = np.zeros((3,1), dtype = np.float32)

        # Acce measurement
        self.__accMeasure_e = np.array([[np.float32(0.0)],[np.float32(0.0)],[-self.__gravity]], dtype = np.float32)

        # variables for second order predict
        self.__xLast = self.__x
        self.__aLast = self.__a
        self.__wLast = self.__w

        ## Predict and state covariance
        # System state covariance matrix
        self.__P = np.eye(self.__nStates, dtype=np.float32)
        self.__PInit = self.__P

        ## System state predict uncertainty matrix
        self.__Q = np.eye(self.__nStates, dtype=np.float32)

        ## Measurement uncertainties covariance

        # Matrixes
        self.__R_pos        = np.eye(3, dtype=np.float32)
        self.__R_yaw        = 1.0
        self.__R_leveling   = 1.0

        # Measure functions
        I = np.eye(3, dtype = np.float32)
        self.__hPosition = np.zeros((3,self.__nStates), dtype = np.float32)
        self.__hPosition[0:3, 0:3] = I

        self.__hYaw = np.zeros((1,self.__nStates), dtype = np.float32)
        self.__hYaw[0,11] = 1.0

        self.__hLeveling = np.zeros((2,self.__nStates), dtype = np.float32)
        self.__hLeveling[0,9] = 1.0
        self.__hLeveling[1,10] = 1.0


        # Geometry data
        self.__droneGeom = DroneGeometry()
        self.__rotMat_bs = np.eye(3, dtype = np.float32)
        self.__pos_b_bs = np.zeros((3,1), dtype = np.float32)

        # Transform from body to ned, this is computed and written to self each predict step
        self.__rotMat_bn = np.eye(3, dtype = np.float32)
        self.__t_nb = np.zeros((3,3), dtype = np.float32)

        # Time stuff
        self.__timeMaxDiffPose = 10.0
        self.__timeNewPoseUpdate = 0.0
        self.__timeDiffPose = self.__timeMaxDiffPose +2.0

        self.__timeLastImuMsg = 0.0
        self.__maxDt = 1.0/50.0

        self.__imuPredictDt = np.float32(self.__maxDt)

        # Delta imu related variables
        self.__deltaImuCounter = 0
        self.__deltaImuDt = 0.001
        self.__deltaImuCum = False

        # Bool
        self.__filterIsNotReSet = True

        print('INS combined alive!')

    def predict(self):
        '''
        If fixed rate predict is active then the predict function is called here
        '''
        if self.__fixedPredictRate == True:
            self.__predictImuDataParsing()

    def checkTiming(self, currentTime):
        '''
        Function to routinely call to check if timers has expired
        '''

        # Finds delta time
        self.__timeDiffPose = currentTime - self.__timeNewPoseUpdate

        # If timer expires, then filter is set offline
        if self.__timeDiffPose > self.__timeMaxDiffPose:
            self.online = False
            # Reseting filter once
```

```
            if self.___filterIsNotReSet == True:
                self.___filterIsNotReSet = False
                self.resetFilter(keepAngles=True)
                # Printing that filter is offline
                print('Warn: orientation node: pos filter offline')
                print('timer expired by:')
                print(self.___timeDiffPose)
        else:
            self.online = True
            # Reseting filter once
            if self.___filterIsNotReSet == False:
                self.___filterIsNotReSet = True
                print('Status: orientation node: pos filter online!')


    # Predict functions
    def __predict(self, dt):
        '''
            Used to predict the next filter state

            Input   dt is a scalar
        '''
        # Calculates non linear therms and stores them to selfs
        tHeta_nb = self.___x[9:12]
        self.___rotMat_bn = self.___droneGeom.rotFun_bn(tHeta_nb)
        self.___t_nb = self.___droneGeom.rateTransform_nb(tHeta_nb)

        # Calculates curret linearization of the state transition equation
        F = self.___stateTransitionLin(dt, strangeTherms=False)

        B = self.___controlInput(dt)

        W = self.___covariancePredictWeight(dt)

        ## Predicts state
        # If second order, can be used in cases where compute power is limited, ie. predict must be run more seldomly
        if self.___secondOrderPredict == True:
            # Calculates propagation step and control input
            x = np.float32(1.5)*self.___x - np.float32(0.5)*self.___xLast
            a = np.float32(1.5)*self.___a - np.float32(0.5)*self.___aLast
            w = np.float32(1.5)*self.___w - np.float32(0.5)*self.___wLast

            # Calculates correct state propagation matrix for second order step
            f = F - np.eye(self.___nStates, dtype = np.float32)

            # Control input
            u = np.zeros((6,1), dtype = np.float32)
            u[0:3] = a
            u[3:6] = w

            # Propagates
            self.___x = self.___x + f @ x + B @ u

            # Sets last values to current values
            self.___xLast = self.___x
            self.___aLast = self.___a
            self.___wLast = self.___w
        else:
            # Control input
            u = np.zeros((6,1), dtype = np.float32)
            u[0:3] = self.___a
            u[3:6] = self.___w

            # Std filter implementation
            self.___x = F @ self.___x + B @ u

        # Predicts state covariance matrix
        self.___P = F @ self.___P @ F.T + W @ self.___Q @ W.T

        # Wrapping euler angles
        self.___wrapEulerAngles()

    def __stateTransitionLin(self, dt, strangeTherms = False):
        '''
            Function to return state transition matrix

            Returns
                np shape (18, 18)
        '''

        # Identity matrix
        I = np.eye(3, dtype = np.float32)

        ## State transition
        # Setsup state transition matrix
        F = np.eye(self.___nStates, dtype=np.float32)

        # Position related
        if self.online:
            # Pos related
            # Integration of lin vel
            F[0:3, 3:6] = I*dt

            # Vel related
            # Subtraction og bias
            F[3:6, 6:9] = -self.___rotMat_bn*dt
            # Strange therm
            if strangeTherms:
                # Gets data for comp of strange therm
                tHeta = self.___x[9:12]
                am = self.___a
                ab = self.___x[6:9]
                F[3:6,9:12] = self.___velThetaStateTransTherm(tHeta, am, ab)*dt
            # Subtraction of grav acc (specific force from gravity, tho gravity is not a force, but that
```

```python
construct is okay for this filter)
            if self.__gravityState == True:
                F[3:6, 15:18] = -I*dt

        # Angle related
        # Subtraction of omg bias
        F[9:12, 12:15] = -self.__t_nb*dt

        # When filter is disabled the yaw is kept constant
        if self.online == False:
            F[11,12:15] = np.zeros((1,3), dtype = np.float32)

        if strangeTherms:
            # Gets data for comp of strange therm
            tHeta = self.__x[9:12]
            wm = self.__w
            wb = self.__x[12:15]
            F[9:12, 9:12] = I + self.__thetaThetaStateTransTherms(tHeta, wm, wb)*dt

        return F

    def __velThetaStateTransTherm(self, tHeta, am, ab):
        '''
        Function to compute strange therm to do with velocity update
        '''
        # Parsing data
        phi = tHeta[0,0]
        theta = tHeta[1,0]
        psi = tHeta[2,0]

        amx = am[0,0]
        amy = am[1,0]
        amz = am[2,0]

        abx = ab[0,0]
        aby = ab[1,0]
        abz = ab[2,0]

        #Pre computes
        sx = np.sin(phi)
        cx = np.cos(phi)
        sy = np.sin(theta)
        cy = np.cos(theta)
        sz = np.sin(psi)
        cz = np.cos(psi)

        mat = np.array([    [ -((sx*sz + cx*cz*sy)*(aby - amy) + (cx*sz - cz*sx*sy)*(abz - amz)), -(cx*cy*cz*(abz - amz) - cz*sy*(abx - amx) + cy*cz*sx*(aby - amy)),  ((cx*cz + sx*sy*sz)*(aby - amy) - (cz*sx - cx*sy*sz)*(abz - amz) + cy*sz*(abx - amx))],
                            [  ((cz*sx - cx*sy*sz)*(aby - amy) + (cx*cz + sx*sy*sz)*(abz - amz)), -(cx*cy*sz*(abz - amz) - sy*sz*(abx - amx) + cy*sx*sz*(aby - amy)), -((sx*sz + cx*cz*sy)*(abz - amz) - (cx*sz - cz*sx*sy)*(aby - amy) + cy*cz*(abx - amx))],
                            [                              -(cx*cy*(aby - amy) - cy*sx*(abz - amz)),   (cy*(abx - amx) + cx*sy*(abz - amz) + sx*sy*(aby - amy)),
                                 np.float32(0.0)]], dtype = np.float32)

        return mat

    def __thetaThetaStateTransTherms(self, tHeta, wm, wb):
        '''
        Function to compute strange therm to do with tHeta update
        '''

        # Parsing data
        phi = tHeta[0,0]
        theta = tHeta[1,0]
        psi = tHeta[2,0]

        wmx = wm[0,0]
        wmy = wm[1,0]
        wmz = wm[2,0]

        pb = wb[0,0]
        qb = wb[1,0]
        rb = wb[2,0]

        #Pre computes
        sx = np.sin(phi)
        cx = np.cos(phi)
        sy = np.sin(theta)
        cy = np.cos(theta)
        ty = np.tan(theta)
        sz = np.sin(psi)
        cz = np.cos(psi)

        # Guards agains divide by zero
        if np.abs(cy) < 0.01:
            cy = 0.01*np.sign(cy)

        mat = np.array([    [      -(cx*ty*(qb - wmy) - sx*ty*(rb - wmz)), -(cx*(ty**2 + 1)*(rb - wmz) + sx*(ty**2 + 1)*(qb - wmy)), 0],
                            [              (cx*(rb - wmz) + sx*(qb - wmy)),
                           0, 0],
                            [ -((cx*(qb - wmy))/cy - (sx*(rb - wmz))/cy),   -((cx*sy*(rb - wmz))/cy**2 + (sx*sy*(qb - wmy))/cy**2), 0]], dtype = np.float32)

        return mat

    def __controlInput(self, dt):
        '''
            Function to return control input matrix

            Returns
                np shape (18, 6)
        '''
```

```
        # Predefining B matrix
        B = np.zeros((self.___nStates,6), dtype = np.float32)

        # Populates B matrix
        # Acc related
        B[3:6,0:3] = self.___rotMat_bn*dt
        # Omg related
        B[9:12,3:6] = self.___t_nb*dt

        if self.online == False:
            B[11,3:6] = np.zeros((1,3), dtype = np.float32)

        return B

    def __covariancePredictWeight(self, dt):
        '''
            Function to return covariance weight update

            return W np shape (18,18)
        '''

        # Creates eye mat
        W = np.eye(self.___nStates, dtype = np.float32)
        # Sets nonlinear therm
        W[3:6,3:6] = self.___rotMat_bn
        W[9:12,9:12] = self.___t_nb

        # Weights by dt
        W = W*dt

        return W

    def __wrapEulerAngles(self):
        '''
        Function to wrap states
        phi is wrapped to [-pi, pi)
        theta is wrapped to [-pi, pi)
        psi is wrapped to [0, 2*pi)
        '''

        ## Setting to state vector
        # Phi
        self.___x[0, 0] = self.___modPiToPi(self.___x[0,0])
        # Theta
        self.___x[1, 0] = self.___modPiToPi(self.___x[1,0])
        # Psi
        self.___x[2, 0] = np.mod(self.___x[2,0], 2*np.pi)

    # Euler angle specifics
    def __wrapEulerAngles(self):
        '''
        Function to wrap states
        phi is wrapped to [-pi, pi)
        theta is wrapped to [-pi, pi)
        psi is wrapped to [0, 2*pi)
        '''

        ## Setting to state vector
        # Phi
        self.___x[9, 0] = self.___modPiToPi(self.___x[9,0])
        # Theta
        self.___x[10, 0] = self.___modPiToPi(self.___x[10,0])
        # Psi
        self.___x[11, 0] = np.mod(self.___x[11,0], 2*np.pi)

    def __modPiToPi(self, ang):
        '''
        Function to map a variable to [-pi to pi)

        TODO: Needs a way to handle cases where the angle is grossly wrong
        '''

        # Predefining variable
        angWrapped = 0.0

        # Wrapping
        if ang >= np.pi:
            angWrapped = ang-2*np.pi
        elif ang < -np.pi:
            angWrapped = 2*np.pi+ang
        else:
            angWrapped = ang

        return angWrapped

    # Innovation functions
    def __innovationLin(self, xMeasure, H):
        '''
        Finds the error between the measurement and the predicted value, given a linear measurement
    function

        Input is     X_measure np shape (m,1)
                     H np shape (m,nStates)

        Output is    Y np shape (m,1)
        '''

        return xMeasure - H @ self.___x

    def __innovationYaw(self, yawMeasure, yawPredict):
        '''
        Function to return "geodesic" innovation on 0 to 2pi maping

        Takes two scalars as input and returns np shape (1,1)
```

```
        '''

        # Predefines error
        e = np.zeros((1,1), dtype = np.float32)

        # wraps measurement
        yawMeasure = np.remainder(yawMeasure, 2*np.pi)

        # Computes the two possible solutions
        e1 = yawMeasure - yawPredict

        if e1 < 0:
            e2 = 2*np.pi + e1
        else:
            e2 = e1 - 2*np.pi

        # Finds the shortest path
        if np.abs(e1) < np.abs(e2):
            e[0][0] = e1
        else:
            e[0][0] = e2

        return e

    def __innovationPhiTheta(self,angMeasure, angPredict):
        '''
        Function to return "geodesic" innovation on -pi to pi maping

        Takes two scalars as input and returns scalar
        '''

        # Wraps measurement
        angMeasure = self.__modPiToPi(angMeasure)

        # Predefines error
        e = 0.0

        e1 = angMeasure - angPredict

        if e1 < 0:
            e2 = 2*np.pi + e1
        else:
            e2 = e1 - 2*np.pi

        # Finds the shortest path
        if np.abs(e1) < np.abs(e2):
            e = e1
        else:
            e = e2

        return e

    # Vanilla kf functions
    def __computeKalman(self,H,R):
        '''
        Function to compute kalman gain

        Input       H np shape (n,m)
                    R np shape (n,n)

        Output      K np shape (m,m)
        '''

        S = H @ self.__P @ H.T + R

        return self.__P @ H.T @ np.linalg.inv(S)

    def __update(self,K,y,H):
        '''
        Function to update estimates

        Input       K np shape (m,n)
                    y np shape (n,1)
                    H np shape (m,n)
        '''


        # Updates estimate
        self.__x = self.__x + K @ y
        self.__P = (np.eye(self.__nStates, dtype=np.float32) - K @ H) @ self.__P

    # Position measuremen functions
    def __posMeasurement(self, posMeasure, R):
        '''
        Function to set pose measurement

        Input       pose np shape (3,1) on form [x, y, z]'
        '''

        # Gets H matrices and populates a bigger matrix
        H = self.__hPosition

        ## Kalman routine

        # Innovation
        y = self.__innovationLin(posMeasure, H)

        # Kalman gain
        K = self.__computeKalman(H,R)

        # Updates
        self.__update(K,y,H)

    def __posSet(self, posMeasure):
        '''
```

```
        Function to set the position directly, this is to be used when there is a "large" jump in the pf solution

        input posMeasure np shape (3,1)
        '''

        # Sets position to measured position and kills velocity
        self.__x[0:3] = posMeasure
        self.__x[3:6] = np.zeros((3,1), dtype = np.float32)
        self.__x[6:9] = np.zeros((3,1), dtype = np.float32)
        # Sets cov to init cov
        self.__P[0:9] = self.__PInit[0:9]

    def __posEvaluation(self, posMeasure, R):
        '''
        Function to decide if position is to be passed as a measurement or a direct replacement of the position in the filter

        If position from pf filter jumps the prediction of the pf filter is passed directly to the states to prevent a false velocity spike
        This velocity spike will then move the particles in a wrong direction, further worsening the problem

        Input       pose np shape (3,1) on form [x y z]'
                    R    np shape (3,3)
        '''

        # Gets predicted value
        posPredict = self.__x[0:3]

        # calculates norm predicted to "measurement"
        norm = np.linalg.norm(posMeasure-posPredict)

        if norm <= self.__posThreshold:
            self.__posMeasurement(posMeasure, R)
        else:
            self.__posSet(posMeasure)
            print('PoseSet')

    # Yaw measurement stuff
    def __yawMeasurement(self, yawMeasure, R):
        '''
        Function to do kalman routine on imu data

        Input       Yaw np shape (1,1)
                    R       scalar
        '''

        ## Kalman routine
        # H matrix
        H = self.__hYaw

        # Defines yaw as np array with shape (1,1)

        # computing innovation
        yawPredict = self.__x[11,0]
        y = self.__innovationYaw(yawMeasure, yawPredict)

        # computes the Kalman gain
        K = self.__computeKalman(H,R)

        # Updates states
        self.__update(K,y,H)

    def __yawSet(self, yawMeasure):
        '''
        Function to set the yaw directly, this is to be used when there is a "large" jump in the pf solution

        input yawMeasure np shape (3,1)
        '''

        # Sets yaw to measured yaw
        self.__x[11,0] = yawMeasure
        self.__x[14,0] = np.float32(0.0)

        # Sets cov to init cov
        self.__P[11,11] = self.__PInit[11,11]
        self.__P[14,14] = self.__PInit[14,14]

    def __yawEvaluation(self, yawMeasure, R):
        '''
        Function to decide if position is to be passed as a measurement or a direct replacement of the position in the filter

        If position from pf filter jumps the prediction of the pf filter is passed directly to the states to prevent a false velocity spike
        This velocity spike will then move the particles in a wrong direction, further worsening the problem

        Input       pose np shape (3,1) on form [x y z]'
                    R    np shape (3,3)
        '''

        # Gets predicted value
        yawPredict = self.__x[11,0]

        # calculates norm predicted to "measurement"
        norm = np.abs(self.__innovationYaw(yawMeasure, yawPredict))

        if norm <= self.__yawThreshold:
            self.__yawMeasurement(yawMeasure, R)
        else:
            self.__yawSet(yawMeasure)
            print('YawSet')
```

```python

    # Leveling function
    def __angleLeveling(self, acc, subBias = False):
        '''
        Function to do leveling based on imu acceleration data

        Input   acc np shape (3,1)
        '''

        ## Subtracting bias
        if subBias == True:
            accLeveling = acc - self.__x[6:9]
        else:
            accLeveling = acc


        ## Calculating phi and theta based on atan2
        # phi
        phiMeasure = np.arctan2(-accLeveling[1,0],-accLeveling[2,0])

        # theta
        den = np.sqrt(accLeveling[1,0]**2 + accLeveling[2,0]**2)
        thetaMeasure = np.arctan2(accLeveling[0,0], den)

        # gets predict value
        phiPredict = self.__x[9,0]
        thetaPredict = self.__x[10,0]

        ## Covariance, dynamic tuning of filter
        # Finds deviation from G
        devFromG = np.abs(np.linalg.norm(accLeveling) - self.__gravity)
        # Calculates the absolute covariance
        rScalar = self.__R_leveling*(1.0 + 5000.0*(devFromG + devFromG**2))
        R = rScalar*np.eye(2, dtype = np.float32)
        R = R.astype(np.float32)

        ## Kalman stuff
        # Measurement H matrix
        H = self.__hLeveling

        # Innovation
        y = np.zeros((2,1), dtype = np.float32)
        y[0,0] = self.__innovationPhiTheta(phiMeasure, phiPredict)
        y[1,0] = self.__innovationPhiTheta(thetaMeasure, thetaPredict)

        # Kalman
        K = self.__computeKalman(H,R)

        # Update
        self.__update(K,y,H)

    # Set Imu related function
    def __predictImuDataParsing(self):
        '''
        Function to unite acctions that are to be taken based on IMU data
        '''

        # Uses data for predict
        if self.__deltaImuCum == True and self.__fixedPredictRate == True:
            # Guards against decide by zero
            if self.__deltaImuDt < 0.001:
                self.__deltaImuDt = 0.001

            # Devides delta imu msg by accumulated time to bring it back to original imu units
            self.__a = self.__aCum / self.__deltaImuDt
            self.__w = self.__wCum / self.__deltaImuDt
            # Sets delta imu time to dt for predict
            dt = self.__deltaImuDt

            # Resets values
            self.__aCum = np.zeros((3,1), dtype = np.float32)
            self.__wCum = np.zeros((3,1), dtype = np.float32)
            self.__deltaImuDt = np.float32(0.0)
            self.__deltaImuCounter = 0
        else:
            dt = self.__imuPredictDt

        # If fixed predict rate is true, then dt is set to the fixed predict rate delta time
        if self.__fixedPredictRate == True:
            dt = self.__fixedPredictRateDt

        # Predicts based on new control data
        self.__predict(dt)

        ## Leveling, only when there is low accelerations
        # Abs of sensor acceleration
        absAcc = np.linalg.norm(self.__accMeasure_e)

        # Acceleration thereshold
        if self.__lowThreshold < absAcc and absAcc < self.__highThreshold:
            self.__angleLeveling(self.__accMeasure_e, subBias= False)

    # Helper functions
    def __skew(self, vec):
        '''
        Function to return matrix form of cross product

        Output:     skew mat np shape (3,1)
        '''
        x = vec[0,0]
        y = vec[1,0]
        z = vec[2,0]

        skew = np.array([   [ 0,-z, y],
                            [ z, 0,-x],
```

```python
                            [-y, x, 0]], dtype = np.float32)

        return skew

    def __rMatBNCovariancePropagation(self, tHeta, pos):
        '''
        Function to return the derivative of the propagation function

        Input       tHeta np shape (3,1)
                    pos   np shape (3,1)
        '''
        # Parsing data
        phi     = tHeta[0,0]
        theta   = tHeta[1,0]
        psi     = tHeta[2,0]

        rx = pos[0,0]
        ry = pos[1,0]
        rz = pos[2,0]

        # Pre computing
        sx = np.sin(phi)
        cx = np.cos(phi)
        sy = np.sin(theta)
        cy = np.cos(theta)
        sz = np.sin(psi)
        cz = np.cos(psi)

        deltaF = np.array([ [  ry*(sx*sz + cx*cz*sy) + rz*(cx*sz - cz*sx*sy), rz*cx*cz*cy - rx*cz*sy + ry*cz*cy*sx, rz*(cz*sx - cx*sz*sy) - ry*(cx*cz + sx*sz*sy) - rx*cy*sz],
                            [- ry*(cz*sx - cx*sz*sy) - rz*(cx*cz + sx*sz*sy), rz*cx*cy*sz - rx*sz*sy + ry*cy*sx*sz, rz*(sx*sz + cx*cz*sy) - ry*(cx*sz - cz*sx*sy) + rx*cz*cy],
                            [                                    ry*cx*cy - rz*cy*sx,         - rx*cy - rz*cx*sy - ry*sx*sy,                                   np.float32(0.0)]], dtype = np.float32)

        return deltaF


    # Sensor and aiding functions
    def setImuMeasurement(self, imuData, timeImu):
        '''
        Function to set imu data

        Input imu data np shape (6,1) on form [ax ay az wx wy wz]'
        '''

        ## Parsing data
        # Splits data to acc and omg part
        accMeasure = imuData[0:3]
        omgMeasure = imuData[3:6]

        # Transforming data to estimation frame
        accMeasure_e = self.__rotMat_bs @ accMeasure
        self.__accMeasure_e = accMeasure_e

        # If gravity is a state then it is subtracted in the state transition matrix, othervice it is subtracted here
        if self.__gravityState == True:
            acc = accMeasure_e
        else:
            acc = accMeasure_e - self.__rotMat_bn.T @ self.__gVec

        omg = self.__rotMat_bs @ omgMeasure


        ## Calculates dt

        # time
        timeImu = np.float64(timeImu)

        # If timeLastImuMsg is zero then time is set to last time, (lazy method of initializations the time)
        if self.__timeLastImuMsg <= 0.01:
            self.__timeLastImuMsg = timeImu

        # calculates dt
        dt = timeImu - self.__timeLastImuMsg

        # if dt is to large, then sets dt to max dt
        if np.abs(dt) > self.__maxDt:
            dt = self.__maxDt

        # Casting dt to float32
        dt = np.float32(dt)

        # Sets time now to timeLast
        self.__timeLastImuMsg = timeImu


        ## If delta imu is configured, then IMU data is accumulated
        if self.__deltaImuCum == True and self.__fixedPredictRate == True:
            # Accumulates imu data
            self.__aCum += acc*dt
            self.__wCum += omg*dt
            self.__deltaImuDt += dt
            # Increments counter
            self.__deltaImuCounter += 1
        else:
            # Sets control param to self variable
            self.__a = acc
            self.__w = omg
            self.__imuPredictDt = dt

        # If predict is not called at a fixed rate, then it is called here
        if self.__fixedPredictRate == False:
```

```python
            # Calls actions
            self.___predictImuDataParsing()

    def setPoseMeasurementWithCovariance(self, pose, R, timePose, covCal = False):
        '''
        Function to set pose in ned frame

        Input   pose np shape (4,1) in ned frame
                R np shape (4,4)
                time scalar
        '''

        # Stores time of current msg
        self.___timeNewPoseUpdate = timePose

        # Splits pose to pos and yaw
        pos_n_nb = pose[0:3]
        yaw_nb = pose[3,0]

        # Transforms position measurement to sensor frame

        pos_n_ns    = pos_n_nb + self.___rotMat_bn @ self.___pos_b_bs

        # Splits covariance
        rPos = R[0:3, 0:3]
        rYaw = R[3,3]

        ## Sets pose to position filter
        # Calculates covariance
        if covCal == True:
            # Gets theta and cov of theta
            tHeta_nb    = self.___x[9:12]
            rtHeta      = self.___P[9:12,9:12]
            # Calculates cov prop function
            deltaF = self.___rMatBNCovariancePropagation(tHeta_nb, self.___pos_b_bs)
            rPos = rPos + deltaF @ rtHeta @ deltaF.T
        else:
            rPos = rPos

        # Calls update functions

        # if filter is offline then pos is not calculated
        if self.online:
            self.___posEvaluation(pos_n_ns, rPos)

        self.___yawEvaluation(yaw_nb, rYaw)

    # Set filter params
    def setFilterParameters(self, params):
        '''
        Function to set filter parameters
        '''

        # Passes to member variables
        # Measurement uncertainty covariance
        self.___R_pos        = params.rPos*np.eye(3, dtype=np.float32)

        # Gravity
        self.___gravity = params.gravity
        self.___x[17,0] = -self.___gravity

        self.___gVec = np.array([[np.float32(0.0)],[np.float32(0.0)],[-self.___gravity]], dtype = np.float32)

        # Pos threshold
        self.___posThreshold = params.posThreshold
        self.___yawThreshold = params.yawThreshold

        self.___accHighThreshold = params.levelingWindow

        self.___lowThreshold = self.___gravity*(1-self.___accHighThreshold)
        self.___highThreshold = self.___gravity*(1+self.___accHighThreshold)

        # State transition covariance matrix
        Q = np.eye(self.___nStates, dtype = np.float32)
        Q[0:3,0:3]      = params.qPosition*np.eye(3, dtype = np.float32)
        Q[3:6,3:6]      = params.qLinVel*np.eye(3, dtype = np.float32)
        Q[6:9,6:9]      = params.qBiasAcc*np.eye(3, dtype = np.float32)
        Q[9:12, 9:12]   = params.qAngVel*np.eye(3, dtype = np.float32)
        Q[12:15, 12:15] = params.qBiasGyro*np.eye(3, dtype = np.float32)
        Q[15:18, 15:18] = params.qGravity*np.eye(3, dtype = np.float32)

        # Sets to self variable
        self.__Q = Q

        # Sensor to estimation frame transform
        self.___rotMat_bs = params.rotMat_bs
        self.___pos_b_bs = params.pos_b_bs

        # Initial condition
        # Postion and velocity
        self.___x[0:6]   = params.initState[0:6]
        # Acc Bias
        self.___x[6:9]   = params.initState[9:12]
        # tHeta
        self.___x[9:12] = params.initState[12:15]
        # Omg Bias
        self.___x[12:15] = params.initState[18:21]

        self.___xInit = self.___x

        # If second order predict
        if self.___secondOrderPredict == True:
            self.___xLast = self.___x

```

```
        # Postion and velocity
        self.__P[0:6,0:6] = params.initCov[0:6,0:6]
        # Acc Bias
        self.__P[6:9,6:9] = params.initCov[9:12,9:12]
        # tHeta
        self.__P[9:12,9:12] = params.initCov[12:15,12:15]
        # Omg Bias
        self.__P[12:15,12:15] = params.initCov[18:21,18:21]

        self.__PInit = self.__P

        # Sets to self variables
        self.__timeMaxDiffPose = params.timeMaxDelayPose

        # Filter Config
        self.__deltaImuCum = params.deltaImuCum
        self.__secondOrderPredict = params.secondOrderPredict
        self.__fixedPredictRate = params.fixedRatePredict

    # Public get functions
    def getPositionWithCovariance(self, calCov = False):
        '''
        Function to get position in ned frame

        Output  position np shape (3,1)
                rPos     np shape (3,3)
        '''

        # Gets position of sensor
        pos_n_ns = self.__x[0:3]
        rPos = self.__P[0:3,0:3]

        # Gets orientation
        tHeta_nb = self.__x[9:12]
        rtHeta = self.__P[9:12,9:12]

        # Transforms position measurement
        rotMat_bn = self.__droneGeom.rotFun_bn(tHeta_nb)
        pos_n_nb = pos_n_ns - rotMat_bn @ self.__pos_b_bs

        # Calculates covariance
        if calCov == True:
            deltaF = self.__rMatBNCovariancePropagation(tHeta_nb, self.__pos_b_bs)
            rTot = rPos + deltaF @ rtHeta @ deltaF.T
        else:
            rTot = rPos

        return pos_n_nb, rTot

    def getOrientationWithCovariance(self):
        '''
        Function to get orientation

        Output  orientation np shape (3,1)
                rOri        np shape (3,3)
        '''

        return self.__x[9:12], self.__P[9:12,9:12]

    def getLinearVelocityBodyWithCovariance(self, covCal = False):
        '''
        Function to get linear velocity in body frame

        This function is a combination of the linear velocity from posFilter, and the angular velocity of the oriFilter

        Output:     vel_b_nb np shape (3,1)
                    rBody    np shape (3,3)
        '''

        # Gets states to use for calculation
        # Gets linear velocity in ned frame
        vel_n_ns = self.__x[3:6]

        # Transforms to body
        rotMat_nb = self.__rotMat_bn.T
        vel_b_ns = rotMat_nb @ vel_n_ns

        # Transforms covariance
        rVel_n = self.__P[3:6,3:6]
        rVel_b = rotMat_nb @ rVel_n @ rotMat_nb.T

        # Calculating covariance
        if covCal:
            # Gets variables
            rOmg = self.__P[12:15,12:15]
            pos = self.__pos_b_bs.copy()
            omg_b_ns = self.__w - self.__x[12:15]
            # propper vel
            vel_b_nb = vel_b_ns - self.__skew(omg_b_ns) @ pos
            # propper cov
            deltaFMat = self.__skew(self.__pos_b_bs)
            rTot = rVel_b + deltaFMat @ rOmg @ deltaFMat.T
        else:
            # Dirty vel
            rTot = rVel_b
            vel_b_nb = vel_b_ns

        return vel_b_nb, rTot

    def getAngularVelocityBodyWithCovariance(self):
        '''
        Function to get angular velocity of body frame
```

```
        Output:     omg_b_nb np shape (3,1)
                    rBody    np shape (3,3)
        '''

        return self.__w - self.__x[12:15], self.__P[12:15,12:15]

    def getLinearVelocityLevelWithCovariance(self, covCal = False):
        '''
        Function to get linear velocity in level frame

        This function is a combination of the linear velocity from posFilter, and the angular velocity of
     the oriFilter

        Output:     vel_n_nb np shape (3,1)
                    rLevel   np shape (3,3)
        '''

        # Gets states to use for calculation
        vel_b_nb, rBody = self.getLinearVelocityBodyWithCovariance(covCal = covCal)

        # Transforms to level frame
        theta_nb = self.__x[9:12]
        rotMat_bl = self.__droneGeom.rotFun_bl(theta_nb)

        vel_l_nb = rotMat_bl @ vel_b_nb

        # Calculating covariance
        if covCal == True:
            deltaF = rotMat_bl
            rLevel = deltaF @ rBody @ deltaF.T
        else:
            rLevel = rBody

        return vel_l_nb, rLevel

    def getAngularVelocityLevelWithCovariance(self, covCal = False):
        '''
        Function to get angular velocity of level frame

        Output:     omg_b_nb np shape (3,1)
                    rLevel   np shape (3,3)
        '''

        # Gets angular velocity
        omg_b_nb = self.__w - self.__x[12:15]
        rOmg_b = self.__P[12:15,12:15]

        # Transforms to level frame
        theta_nb = self.__x[9:12]
        rotMat_bl = self.__droneGeom.rotFun_bl(theta_nb)

        # Transforms rates and cov
        omg_l_nb = rotMat_bl @ omg_b_nb

        if covCal == True:
            rOmg_l = rotMat_bl @ rOmg_b @ rotMat_bl.T
        else:
            rOmg_l = rOmg_b

        return omg_l_nb, rOmg_l

    def getPosFilterOnlineState(self):
        '''
        Function to check if kalman filter is online or not
        '''

        return self.online

    def getAccBiasWithCovariance(self):
        '''
        Function to get accelerometer bias
        '''

        return self.__x[6:9], self.__P[6:9, 6:9]

    def getGyroBiasWithCovariance(self):
        '''
        Function to get gyro bias
        '''

        return self.__x[12:15], self.__P[12:15, 12:15]

    # Jit Init function
    def resetFilter(self, keepAngles = False):
        '''
        Function to reset all parameters after runing JitInit function
        '''

        # Reseting state and state covariance
        if keepAngles == True:
            # In some cases there is only a need to reset the linear states, then angular states is kept
            self.__x[0:9] = self.__xInit[0:9]
            self.__P[0:9, 0:9] = self.__PInit[0:9, 0:9]
            self.__x[11] = self.__xInit[11]
            self.__P[11,11] = self.__PInit[11,11]
            self.__x[15:18] = self.__xInit[15:18]
            self.__P[15:18] = self.__PInit[15:18]
            # Resets last variables
            if self.__secondOrderPredict:
                self.__xLast = self.__x
                self.__aLast = np.zeros((3,1), np.float32)
                self.__wLast = np.zeros((3,1), np.float32)
        else:
            self.__x = self.__xInit
            self.__P = self.__PInit
```

```
            # Resets last variables
            if self.___secondOrderPredict:
                self.___xLast = self.___x
                self.___aLast = np.zeros((3,1), np.float32)
                self.___wLast = np.zeros((3,1), np.float32)

        # Resets time stuff
        self.___timeNewPoseUpdate = 0.0
        self.___timeDiffPose = self.___timeMaxDiffPose +2.0

    def ___dryRun(self):
        '''
        Function to call all functions in the class
        '''

        # Predict function
        self.predict()
        dummyTimeF64 = np.float64(0.0)
        self.checkTiming(dummyTimeF64)
        dummyDt = np.float32(0.0)
        self.___predict(dummyDt)
        self.___stateTransitionLin(dummyDt)
        dummyVec3 = np.ones((3,1), dtype = np.float32)
        self.___velThetaStateTransTherm(dummyVec3, dummyVec3, dummyVec3)
        self.___thetaThetaStateTransTherms(dummyVec3, dummyVec3, dummyVec3)
        self.___controlInput(dummyDt)
        self.___covariancePredictWeight(dummyDt)
        self.___wrapEulerAngles()
        self.___modPiToPi(np.float32(0.0))
        # Innovation lin
        dummyXMeasure = np.zeros((1,1), dtype = np.float32)
        dummyHmat = np.zeros((1,self.___nStates), dtype = np.float32)
        dummyHmat[0,0] = 1.0
        dummyY = self.___innovationLin(dummyXMeasure,dummyHmat)
        # InnovationYaw
        self.___innovationYaw(0.0, 0.0)
        # InnovationPi
        self.___innovationPhiTheta(0.0, 0.0)
        # KalmanCompute
        dummyR = np.eye(1, dtype = np.float32)
        dummyK = self.___computeKalman(dummyHmat, dummyR)
        # Update
        self.___update(dummyK, dummyY, dummyHmat)
        # AngleLeveling
        dummyAcc = np.zeros((3,1), dtype = np.float32)
        self.___angleLeveling(dummyAcc)
        # Predict Data Parsing
        self.___predictImuDataParsing()
        # YawMeasure
        rYaw = np.float32(1.0)
        self.___yawMeasurement(0.0, rYaw)
        self.___yawSet(0.0)
        self.___yawEvaluation(0.0, rYaw)
        # PosMeasure
        dummyPos = np.zeros((3,1), dtype = np.float32)
        dummyR = np.eye(3, dtype = np.float32)
        self.___posMeasurement(dummyPos, dummyR)
        self.___posSet(dummyPos)
        self.___posEvaluation(dummyPos, dummyR)
        # SetImuMeasurement
        dummyIMU = np.zeros((6,1), dtype = np.float32)
        dummyTime = np.float32(0.0)
        self.setImuMeasurement(dummyIMU, dummyTime)
        # SetPosMeasurement
        dummyPose = np.zeros((4,1), dtype = np.float32)
        dummyR = np.zeros((4,4), dtype = np.float32)
        self.setPoseMeasurementWithCovariance(dummyPose,dummyR, dummyTime)
        # Skew function
        dummyVec = np.zeros((3,1), dtype = np.float32)
        self.___skew(dummyVec)
        # Cov function
        self.___rMatBNCovariancePropagation(dummyVec, dummyVec)
        # Set Filter parameters
        dummyFilterParams = InsParameters()
        self.setFilterParameters(dummyFilterParams)
        # Get Position
        self.getPositionWithCovariance()
        # Get orientation
        self.getOrientationWithCovariance()
        # Get linear velocity
        self.getLinearVelocityBodyWithCovariance()
        self.getLinearVelocityLevelWithCovariance()
        # Get Angular velocity
        self.getAngularVelocityBodyWithCovariance()
        self.getAngularVelocityLevelWithCovariance()

        self.getPosFilterOnlineState

    def jitInit(self):
        '''
        Function to call all functions in the class

        This function should be ran right after setting up the class, this is tu ensure that everything is
      JIT compiled before run time
        '''

        # INS main class
        print('DryRun: INS Main')
        self.___dryRun()
        self.resetFilter()



def main():
    pass
```

```python



if __name__ == '__main__':
    main()
```

## B.4 idl_pf_pkg

### B.4.1 Particle filter node

```python
# Other imports
import numpy as np

# Own stuff
# Pf filter stuff
from idl_pf_pkg.JitParticleFilterClass import *
from idl_pf_pkg.LocalizationFilter import *
# Geometric data
from idl_botsy_pkg.droneConfiguration import DroneGeometry, Rot

# Filter Specifications
from idl_botsy_pkg.filterConfig import FilterInitialStates, FilterConfiguration, ParticleFilterSetup
from idl_botsy_pkg.softwareConfigutarion import *

# ROS stuff
import rclpy
import ros2_numpy as rnp
import ros2_numpy.point_cloud2 as pc2
import ament_index_python
from std_msgs.msg import Bool
from sensor_msgs.msg import PointCloud2, PointCloud, ChannelFloat32
from geometry_msgs.msg import PoseWithCovarianceStamped, TwistWithCovarianceStamped, Point32
from rclpy.node import Node
from rclpy.qos import qos_profile_sensor_data
from rclpy.time import Time

class PF_ros_node(Node):
    '''
        ROS2 Particle filter node
        Subscribers and publishers are defined at the bottom of __init__ so as to not queue a lot of messages as the JIT class compiles
    '''

    def __init__(self):
        super().__init__('PF_ros_node')

        # Filter configuration class
        filterConfig = FilterConfiguration()

        # Filter initial values
        initStatesNED = FilterInitialStates()

        # Particle filter setup params
        pfSetup = ParticleFilterSetup()

        # Dronegeom for rotation matrices etc
        droneGeom = DroneGeometry()

        # filterConfig.gazeboGT, for defining if you're going to be using velocity and roll/pitch data
        # from groundTruth publisher or from Kalman Filter, set in config file to "synchronize" filters

        # Set message source
        if filterConfig.gazeboGT:
            messageSource = 'gazeboGT/'

        else:
            messageSource = 'ekf/'

        #### Filter params ####
        filter_params                       = LocalizationFilterParams()

        ### Sensor model Params ###
        # Sensor model data [in lack of a better name], NOTE: z_hit + z_rand/z_max = 1
        filter_params.pf_z_hit              = pfSetup.pf_z_hit
        filter_params.pf_z_rand             = pfSetup.pf_z_rand
        filter_params.pf_z_max              = pfSetup.pf_z_max
        filter_params.max_range             = pfSetup.pf_sensMaxRange

        ### Particle Filter ###
        # PF Params
        filter_params.number_of_particles   = pfSetup.numParticles

        # Init position for particle generation
        initPos                             = droneGeom.rotMat_nm @ initStatesNED.pos.asNpArray()
        initPsi                             = np.pi/2 - initStatesNED.tHeta.z
        filter_params.init_pose             = np.array([[initPos[0,0]],
                                                        [initPos[1,0]],
                                                        [initPos[2,0]],
                                                        [initPsi]  ],dtype=np.float32)

        # Covariance for initial particle spread
        initPosCovNED                       = initStatesNED.posCov # In NED frame, switch X and Y to get into map frame
        inittHetaCovNED                     = initStatesNED.tHetaCov
        filter_params.sigma_pose            = np.array([[initPosCovNED.y],
                                                        [initPosCovNED.x],
                                                        [initPosCovNED.z],
                                                        [inittHetaCovNED.z]],dtype=np.float32)

        # Number of points to use from pointcloud in update step
        filter_params.nPts_PC               = pfSetup.nPts_PC   # Number of points to sample from pointcloud

        # Wether or not to use the square sum method to update the weight in the pointcloud-update step
        filter_params.pcUpdateSqSum         = pfSetup.pcUpdateSqSum

        # PC downsampling configuration
```

```python
        filter_params.pcdsRandPoints = pfSetup.pcdsRandPoints
        filter_params.pcdsLoopSelect = pfSetup.pcdsLoopSelect
        filter_params.pcdsLoopSelMaxLoops = pfSetup.pcdsLoopSelMaxLoops

        # Threshold for resampling
        # (resample if effective sample size is less than threshold)
        filter_params.resamplingThreshold    = pfSetup.resamplingThreshold

        ### Propogation velocity ###
        # Const cov to be added to propogation velocity
        filter_params.constVelVariance       = np.array([[pfSetup.constVar.x],
                                                         [pfSetup.constVar.y],
                                                         [pfSetup.constVar.z],
                                                         [pfSetup.constVarPsi]], dtype = np.float32)

        # Max value of added velocity std.dev when no new velocity message received
        maxVelStdCtr                         = pfSetup.maxVelStdCtr
        filter_params.maxVelStdCtr           = np.array([[maxVelStdCtr],
                                                         [maxVelStdCtr],
                                                         [maxVelStdCtr],
                                                         [maxVelStdCtr]], dtype = np.float32)

        # Watchdog counter gain K for velocity std.dev
        filter_params.wd_K                   = pfSetup.wd_K

        # Init propagation noise
        filter_params.velStd_l               = np.array([[initStatesNED.linVelCov.y],
                                                         [initStatesNED.linVelCov.x],
                                                         [initStatesNED.linVelCov.z],
                                                         [initStatesNED.angVelCov.z]], dtype = np.float32)

        ### Histogram Smoothing ###
        histRes                              = pfSetup.histRes #  0.01 [m], 0.01 [rad] (~ 0.57 degrees)
        smoothingGaussianStdDev              = pfSetup.histGaussStdDev
        filter_params.histogramResolution    = np.array([[histRes],
                                                         [histRes],
                                                         [histRes],
                                                         [histRes]], dtype=np.float32)
        histKernel                           = filter_params.computeGaussianKernel(histRes,
    smoothingGaussianStdDev)
        filter_params.histSmoothingKernel    = histKernel


        ##### Init filter #####
        self.get_logger().info("Filter init start...")
        self.__localizationFilter            = LocalizationFilter(params = filter_params,
                                                                  secondOrder = pfSetup.
    secondOrderIntegration,
                                                                  deltaPosition = pfSetup.
    deltaPositionIntegration)
        self.get_logger().info("Filter init Completed!")

        # Indicator to be set when kf is ready, initiates propagation
        self.__systemReadyIndicator          = True


        ##### Particle pointcloud #####
        # Bool to disable publishing if desired
        self.__publishPFParticlePointcloud   = pfSetup.pubPFParticlePC

        # Rate of which to publish pointcloud
        pfPointcloudPublisherRate            = pfSetup.pfPCPublisherRate
        pfPointcloudPublisherDt              = 1.0 / pfPointcloudPublisherRate

        # Publish poitcloud containing __pfPointcloudSize number of particles
        self.__pfPointcloudSize              = pfSetup.pfPCSize

        ##### Timers #####
        ## Rates ##
        propogation_rate                     = 10
        localization_rate                    = 10

        ## dt ##
        self.prop_dt                         = 1.0/propogation_rate
        localization_dt                      = 1.0/localization_rate

        ## Timers ##
        self.__PF_propogation_timer          = self.create_timer(self.prop_dt, self.__propogation_callback)
        self.__localization_timer            = self.create_timer(localization_dt, self.
    __localization_callback)

        if self.__publishPFParticlePointcloud:
            # Only create timer if bool is set
            self.__PFPCPublishTimer          = self.create_timer(pfPointcloudPublisherDt, self.
    __publish_particlePointCloud)

        ## Init subscribers ##
        if simulation is True:
            self.__pointcloud_subscriber     = self.create_subscription(PointCloud2, '/zed_mini_depth/
    points', self.__pointcloud_callback, 10)

        else:

            self.__pointcloud_subscriber     = self.create_subscription(PointCloud2, '/zedm/zed_node/
    point_cloud/cloud_registered', self.__pointcloud_callback, 10)


        self.__velocity_subcriber            = self.create_subscription(TwistWithCovarianceStamped,
    messageSource + 'vel_level', self.__velocity_callback, 10)

        self.__rollPitch_subcriber           = self.create_subscription(PoseWithCovarianceStamped,
    messageSource + 'pose_ned', self.__roll_pitch_callback, 10)

        self.__systemReset_subscriber        = self.create_subscription(Bool, 'ins/system/reset', self.
    __systemReset_callback, 10)
```

```python

        self.__systemStart_subscriber       = self.create_subscription(Bool, 'ins/system/start', self.__systemStart_callback, 10)

        ## Publisher to publish position and yaw ##
        self.__pose_publisher               = self.create_publisher(PoseWithCovarianceStamped, 'pf/pose_ned', 10)

        ## Publisher to publish pointcloud of particle positions ##
        self.__pose_pointcloud_publisher    = self.create_publisher(PointCloud, 'pf/pose_ned/pointcloud', 10)

    def __publish_particlePointCloud(self):
        '''
            Function to publish particle point cloud for visualization in rviz
        '''

        # Get vector of particle poses
        pf_poseVec = self.__localizationFilter.getPFParticlePoseVector()

        # Define message
        pf_particlePCMsg = PointCloud()
        channel = ChannelFloat32()
        point = Point32()

        # Fill header
        pf_particlePCMsg.header.stamp = self.get_clock().now().to_msg()
        pf_particlePCMsg.header.frame_id = 'map_idl'

        # Fill channel name of message
        channel.name = 'intensity'
        intensity = 128

        # Initialize empty lists
        ptList = []
        chList = []

        # Fill point data
        for ii in range(self.__pfPointcloudSize):
            # Fill channel data
            chList.append(float(intensity))

            # Append new point to list of points
            ptList.append(Point32())

            # Fill Point data
            ParticlePose = pf_poseVec[:, ii].copy()
            ptList[ii].x = float(ParticlePose[0])
            ptList[ii].y = float(ParticlePose[1])
            ptList[ii].z = float(ParticlePose[2])

        # Set channel values to chList
        channel.values = chList

        # Populate pf message channels
        pf_particlePCMsg.channels = [channel]

        # Set points to list of Point32
        pf_particlePCMsg.points = ptList

        # Publish message
        self.__pose_pointcloud_publisher.publish(pf_particlePCMsg)

    def __publish_pose(self):
        '''
            Gets filter pose and variance from localizationFilter then publishes them as a PoseWithCovarianceStamped message
        '''

        # Only publish if systemReady bool is set
        if self.__systemReadyIndicator == True:
            # Get pose and variance
            pose_n_nb = self.__localizationFilter.getFilterPoseNED()
            var = self.__localizationFilter.getFilterVarianceNED()

            # Extract positions and angle from pose
            pos_n_nb = pose_n_nb[:3,0].reshape(3,1)
            psi_n_nb = pose_n_nb[3,0]

            # Setup diagonal matrix with variances of X-, Y-, Z-, and rotZ
            var_n_nb = np.zeros((6,6), dtype = np.float32)
            var_n_nb[0,0] = var[0,0]
            var_n_nb[1,1] = var[1,0]
            var_n_nb[2,2] = var[2,0]
            var_n_nb[5,5] = var[3,0]

            # Create empty message
            bodyPoseMessage = PoseWithCovarianceStamped()

            # Assign timestamp
            timeNow = self.get_clock().now().to_msg()
            bodyPoseMessage.header.stamp = timeNow

            #### Populate Pose ####
            ### Positions ###
            posToMsg_n_nb = pos_n_nb.astype(np.float64)
            bodyPoseMessage.pose.pose.position.x = posToMsg_n_nb[0, 0]
            bodyPoseMessage.pose.pose.position.y = posToMsg_n_nb[1, 0]
            bodyPoseMessage.pose.pose.position.z = posToMsg_n_nb[2, 0]
            bodyPoseMessage.pose.covariance = var_n_nb.reshape(36).astype(np.float64)

            ### Quaternions ###
            ## Not actually a quaternion, but roll/pitch/yaw ##
            psiToMsg_n_nb = psi_n_nb.astype(np.float64)
            bodyPoseMessage.pose.pose.orientation.x = 0.0
```

```python
            bodyPoseMessage.pose.pose.orientation.y = 0.0
            bodyPoseMessage.pose.pose.orientation.z = psiToMsg_n_nb
            bodyPoseMessage.pose.pose.orientation.w = -2.0

            self.__pose_publisher.publish(bodyPoseMessage)

    # Callbacks

    def __localization_callback(self):
        '''
            Runs localize method of localizationFilter, creates pose histograms from particle poses and
 weights
            and smooths them with a gaussian kernel (histSmoothingKernel -> param of localizationFilter),
            saves most likely pose and the particle variance around this pose to filter variables
        '''

        if self.__systemReadyIndicator == True:
            # Run  localize
            self.__localizationFilter.localize()

            # Publish pose
            self.__publish_pose()

    def __propogation_callback(self):
        '''
            Function to call pf.propagate with set rate
        '''

        # Only propagate if kf has sent ready signal
        if self.__systemReadyIndicator:
            # Propogate particles
            self.__localizationFilter.propagate(self.prop_dt)

    def __pointcloud_callback(self, pc_msg):
        '''
            Callback function to run each timne a new pointcloud is received from the camera
        '''

        if self.__systemReadyIndicator == True:
            # Parse pointcloud
            pointcloud_from_msg = pc2.pointcloud2_to_xyz_array(pc_msg)

            # Call update with new pointcloud
            self.__localizationFilter.pointcloud_update(pointcloud_from_msg)

    def __velocity_callback(self, vel_msg):
        '''
            Callback function to run when a new velocity from the KF is received
        '''

        # Get velocities from message
        lin_vel_from_msg = vel_msg.twist.twist.linear
        ang_vel_from_msg = vel_msg.twist.twist.angular
        cov_from_msg = vel_msg.twist.covariance

        # Set up velocity-vector from message
        velVec_l = np.array([[lin_vel_from_msg.x],
                             [lin_vel_from_msg.y],
                             [lin_vel_from_msg.z],
                             [ang_vel_from_msg.z]], dtype=np.float32)

        # Calculate vector of std.deviations from covariance-matrix
        velCov_l = np.array([[cov_from_msg[0]],
                             [cov_from_msg[7]],
                             [cov_from_msg[14]],
                             [cov_from_msg[35]]], dtype=np.float32)

        # Gets time from msg
        timeNsec = Time.from_msg(vel_msg.header.stamp).nanoseconds
        timeSec = timeNsec*10**(-9)


        # Set velocities and std dev to filter
        self.__localizationFilter.setPropagationVelocityWithCovariance( velVec_l,
                                                                        velCov_l,
                                                                        timeSec)

    def __roll_pitch_callback(self, rp_msg):
        '''
            Callback function to update roll and pitch values from KF
        '''
        # Get and set orientation from GT message.
        orient_from_msg = rp_msg.pose.pose.orientation
        roll = orient_from_msg.x
        pitch = orient_from_msg.y
        statusMsg = orient_from_msg.w

        # Construct vector
        tHeta_bl = np.array([[roll],[pitch],[0.0]], dtype=np.float32)

        # If w from quat is -3, set bool ekfOnline false
        if statusMsg < -2.5:
            ekfOnline = False

        else:
            ekfOnline = True

        # Pass to filter
        self.__localizationFilter.setEKFLinearOnlineState(ekfOnline)
        self.__localizationFilter.setCurrentRollPitchAngles(tHeta_bl)

    def __systemReset_callback(self, reset_msg):
        '''
            Callback to reset LocalizationFilter to initial state
        '''
```

```python
        # If msg.data is true, reset filter
        if reset_msg.data == True:
            # Call function to reset filter to initial pose
            self.__localizationFilter.resetParticleFilterToInitPose()
            self.get_logger().info("Particle filter reset")

    def __systemStart_callback(self, systemStart_msg):
        '''
            Callback to set kf indicator
        '''

        # Sets the indicator at first true message
        if systemStart_msg.data == True:
            self.__systemReadyIndicator = True


def main(args=None):
    rclpy.init(args=args)

    pointCloudSubscriber = PF_ros_node()

    rclpy.spin(pointCloudSubscriber)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    pointCloudSubscriber.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

### B.4.2 Particle filter class

```python
import numpy as np

from numba import int32, float32, float64, jit, types, typed, typeof  # import the types
from numba.experimental import jitclass

@jit
def coordinateTransform(theta):
    return np.array([[np.cos(theta), -np.sin(theta), 0.0, 0.0],
                     [np.sin(theta), np.cos(theta), 0.0, 0.0],
                     [0.0, 0.0, 1.0, 0.0],
                     [0.0, 0.0, 0.0, 1.0]],dtype=np.float32)

particleSpec = [
    ('weight', float64),            # a simple scalar field
    ('pose', float32[:,:]),         # an array field
] # Numba spec for particle class

@jitclass(particleSpec)
class Particle(object):

    def __init__(self, pose, initWeight):
        # Particle pose
        self.pose = pose
        self.weight = initWeight

    def move(self, V, dt):
        # Rotation about z
        R = coordinateTransform(self.pose[3][0]+ V[3][0]*dt/2.0)

        # Update pose
        self.pose += R @ (V*dt).astype(np.float32)

        # Wrap yaw 0 - 2*pi
        self.pose[3] = np.mod(self.pose[3], 2.0*np.pi)


# Getting the type of one instance of the class Particle
particleNumbaType = Particle.class_type.instance_type
particleFilterSpec = [
    ('__nParticles', int32),
    ('__particles', types.ListType(particleNumbaType)),                 # Particle Class
    ('__effectiveParticles', float32),
    ('__z_hit', float64),
    ('__z_rand', float64),
    ('__z_max', float64),
    ('__poseVec', float32[:,:]),
    ('__weightVec', float64[:,:]),          # an array field
] # Numba spec for particle filter class

@jitclass(particleFilterSpec)
class ParticleFilter(object):

    def __init__(self, z_hit, z_rand, z_max, nParticles, initPose, sig_pose):
        '''
            Creates a Particle Filter object

            input:
                z_hit, z_rand, z_max,   :   Sensor parameters                              (floats)
                nParticles              :   Number of particles in filter                  (int)
                initPose                :   Initial Pose of Particles                      ((4x1) vector, [x, y, z, yaw])
                sig_pose                :   Std deviation of particles around initial pose ((4x1) vector, [x, y, z, yaw])

        '''
        # Parameters for scan matching
        self.__z_hit = z_hit
```

```
        self.___z_rand = z_rand
        self.___z_max = z_max

        # Create list of particles
        self.___nParticles = nParticles                                         # Number of particles in filter
        self.___particles = typed.List.empty_list(particleNumbaType)             # List of particle objects
        self.___effectiveParticles = nParticles                                  # Effective sample size of the particle filter (measure of degeneracy)
        self.___poseVec = np.zeros((4,self.___nParticles), dtype = np.float32)    # Vector containing particle poses
        self.___weightVec = np.zeros((1, self.___nParticles), dtype = np.float64) # Vector containing particle weights

        # Uniform initial weight for all particles
        initWeight = np.float32(1.0/nParticles)

        for ii in range(nParticles):
            # Add random noise to initial pose with std.dev sig_pose
            initPose_with_noise = self.normalDistVector(initPose, sig_pose)
            self.___particles.append(Particle(initPose_with_noise, initWeight))
            self.___poseVec[:,ii] = self.___particles[ii].pose.copy().reshape(4)
            self.___weightVec[0,ii] = initWeight

        print("Particle filter object created!")

    # Function to reset all particles to have pose around [pose] with std.dev [sig_pose]
    def reset_filter(self, pose, sig_pose):
        '''
            Method to reset filter to a chosen pose with a set std deviation
        '''

        # Calculate normalized uniform weight
        particleWeight = np.float32(1.0/self.___nParticles)

        # Reset all particles
        for ii in range(self.___nParticles):
            # Add random noise to pose with std.dev sig_pose
            initPose_with_noise = self.normalDistVector(pose, sig_pose)
            self.___particles[ii].pose = initPose_with_noise.copy()
            self.___particles[ii].weight = particleWeight*1.0
            self.___poseVec[:,ii] = self.___particles[ii].pose.copy().reshape(4)
            self.___weightVec[0,ii] = particleWeight

    # Function to dry-run all filter methods, to not have to JIT-compile during operation
    def dry_run(self, initPose, sig_pose, likelihoodMap):
        '''
            Does a dry run of all functions in the filter, only really used in the JIT-compiled version to have all functions in the
            class compile at init to avoid long delays during runtime.
        '''

        # For propogation dry-run
        dry_run_vector4 = np.ones((4,1), dtype=np.float32)             # 3x1 vector of zeros to pass into filter functions
        dry_run_dt = 0.0001

        # For update
        dry_run_map = likelihoodMap
        dry_run_map_origin_offset = np.zeros((3,1), dtype = np.float32)
        dry_run_map_resolution = np.float32(0.1)
        dry_run_map_size_in_cells = np.ones((3,1), dtype = np.int32)
        dry_run_pointcloud = np.ones((3,100), dtype = np.float32)

        # Call functions in filter
        self.propagate(dry_run_vector4, dry_run_vector4, dry_run_dt)

        self.normalDistVector(dry_run_vector4, dry_run_vector4)

        # Pointcloud Update
        self.pointcloud_update(dry_run_map,
                               dry_run_map_origin_offset,
                               dry_run_map_resolution,
                               dry_run_map_size_in_cells,
                               1.0,
                               False,
                               dry_run_pointcloud,
                               False)

        # Resampling functs
        self.getEffectiveSampleSize()
        self.systematicResample()

        # Normalize
        self.normalize_particle_weight(1.0)

        # Reset
        self.reset_filter(initPose, sig_pose)


    # Function to create a [4x1] vector drawn from a gaussian distribution around vec_mu, with ve_sig std.dev
    def normalDistVector(self, vec_Mu, vec_Sig):
        '''
            Since numba does not support np.random.normal with vector inputs, we make our own
        '''
        return np.array([[np.random.normal(vec_Mu[0][0], vec_Sig[0][0])],
                         [np.random.normal(vec_Mu[1][0], vec_Sig[1][0])],
                         [np.random.normal(vec_Mu[2][0], vec_Sig[2][0])],
                         [np.random.normal(vec_Mu[3][0], vec_Sig[3][0])]],dtype=np.float32)

    # Function to propagate the particles
    def propagate(self, V, sig_v, dt):
        '''
            Function that propogates the particles in space
```

```
            according to the motion model in particle class

            TODO: Move on from function-based propogation
        '''

        # Make sure data is in correct format
        V = V.astype(np.float32)
        sig_v = sig_v.astype(np.float32)
        dt = np.float32(dt)

        # Propagate particles
        for particle in self.__particles:
            # Add some random noise to propogation velocity
            V_with_noise = self.normalDistVector(V, sig_v)

            # Propogate particles
            particle.move(V_with_noise, dt)

    # Function to resample low-weight particles
    def systematicResample(self):
        '''
            Systematic resampling (Page 5, Table 2, Code block 3: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7079001&tag=1)
        '''
        # Draw random number between 0 and 1/nParticles as initial upper limit for cumulative sum of weight
        r = np.random.uniform(0, 1/self.__nParticles)

        # Initiate cumulative sum of weights as weight of first particle
        W = self.__particles[0].weight
        highestWeightIndex = 0
        highestWeight = W*1.0

        # Initiate indexing variable
        # Used to index particles to replicate in the loop
        i = 0

        for n in range(self.__nParticles):
            # Add 1/N to upper limit of cumulative sum of weights
            u = r + (n)/self.__nParticles

            # While cumulative sum of weights is lower than u
            while W < u:
                # Increment indexing variable
                i += 1

                # If i exceeds range of list, set i = last index to not index outside of bounds, and set W to u to break loop
                if i >= self.__nParticles:
                    i = self.__nParticles - 1
                    W = u

                # Update cumulative sum of weights with weight of particle at index i
                W += self.__particles[i].weight


            # Replace particle pose at index n with particle pose at index i
            # multiply with 1.0 to break reference with self.__particles[i].pose
            # omitting *1.0 makes the pose referenced, and any change to particle[i]'s pose
            # will be the same in all particles referenced.
            self.__particles[n].pose = np.copy(self.__particles[i].pose)

            # Give all particle weight 1/N
            self.__particles[n].weight = np.float32(1.0/self.__nParticles)

    # Function to compute the variance of the pose
    def computePoseVariance(self, pose):
        '''
            Compute variance of all particles from a given particle pose
        '''

        sqError = np.zeros((4,1), dtype=np.float32)
        V2 = 0
        V1 = 0

        # For loop through and add square error
        for ii in range(self.__nParticles):
            # Cumulative sum of square errors

            # Positions
            sqError[:3,0] += self.__particles[ii].weight*(self.__particles[ii].pose[:3,0] - pose[:3,0])**2

            # Yaw, make sure to pick shortest dist e
            yawError = self.__innovationYaw(self.__particles[ii].pose[3,0], pose[3,0])

            sqError[3,0] += self.__particles[ii].weight*(yawError**2)

            # Sum up V1 and V2
            V1 += self.__particles[ii].weight
            V2 += self.__particles[ii].weight**2


        ### Find vaiance ###
        # var = V1/(V1^2 - V2) * sqError
        # Helper variables
        num = V1
        denom = V1**2 - V2

        # Minimum value for denominator in variance calculation
        minDenom = 0.00001

        # Make sure denom does not get too small
        if denom < minDenom:
            denom = minDenom
            print("Denominator in variace calc too small, setting minimum value")
```

```python

        # Calculate weighted variance
        varPose = (num/denom)*sqError

        return varPose.astype(np.float32)

    # Function to find shortest "geodesic" distance around circle (for yaw)
    def __innovationYaw(self, yawMeasure, yawPredict):
        '''
        Function to return "geodesic" innovation on 0 to 2pi mapping

        Takes two scalars as input and returns np shape (1,1)
        '''

        # wraps measurement
        yawMeasure = np.remainder(yawMeasure, 2*np.pi)

        # Computes the two possible solutions
        e1 = yawMeasure - yawPredict

        if e1 < 0:
            e2 = 2*np.pi + e1
        else:
            e2 = e1 - 2*np.pi

        # Finds the shortest path
        if np.abs(e1) < np.abs(e2):
            e = e1
        else:
            e = e2

        return e

    # Function to update weights of particles using measured pointcloud
    def pointcloud_update(self, likelihood_map, map_origin_offset, map_resolution, map_size_in_cells, mapMaxGaussVal, mapUsingUint8Prob, pointcloud_map_frame_body_centered, sqSum=False):
        '''
            Function to transform pointcloud into particle frames and check against map
            Updates weight as product of all map hit probabilities

            TODO:   Implement different weight update if i find the source
        '''
        # for all particles:
        #   rotate and translate san into particle frame
        #   convert scan coordinates to indexes to get prob. from map
        #   check that all indices are valid
        #   update weight of all particles
        #   normalize weights

        # Initialize sum of weights to zero, used in normalizing step
        sum_weights = np.float64(0.0)

        # Calculate z_rand/z_max before loop
        z_misc = self.___z_rand/self.___z_max

        # initialize factor to multiply map prob with, gives possibility to use more data-types
        mapProbabilityFactor = np.float64(1.0)

        # if map uses uint8 as probabilities
        if mapUsingUint8Prob:
            # Update factor
            mapProbabilityFactor = np.float64(mapMaxGaussVal) / (np.float64(255.0))

        for ii in range(self.___nParticles):
            # Get rotation-matrix from particle to map:
            rotmat_pm = coordinateTransform(self.___particles[ii].pose[3,0])[:3,:3]

            # Rotate pointcloud into particle frame
            pointcloud_particle_frame = rotmat_pm @ pointcloud_map_frame_body_centered

            # Translate pointcloud to particle position
            pointcloud_map_frame = self.___particles[ii].pose[:3].copy() + pointcloud_particle_frame

            # Offset the pointcloud and round to find map indices, then cast to int32
            indices = np.empty_like(pointcloud_map_frame)
            np.round_((pointcloud_map_frame - map_origin_offset)/map_resolution, 0, indices)
            indices = indices.astype(np.int32)


            ###### Check validity of points ######
            # Find all points where there are no negative indices
            positive_indices_logical_raw = (indices >= 0) # Check if indices (x-, y- and z-) are positive [True / False]
            # Messy because JIT does not allow .all() with defined axis, the following ANDs along the column of the vector
            positive_indices_logical = (positive_indices_logical_raw[0,:]*positive_indices_logical_raw[1,:]*positive_indices_logical_raw[2,:]) # And through each column to check validity of point

            # Check if in map
            indices_in_map_logical_raw = (indices < map_size_in_cells.copy().reshape((3,1))) # Check if coordinates (x-, y- and z-) are inside map [True / False]
            #Messy because JIT does not allow .all() with defined axis
            indices_in_map_logical = (indices_in_map_logical_raw[0,:]*indices_in_map_logical_raw[1,:]*indices_in_map_logical_raw[2,:]) # And through each column to check validity of point
            #indices_in_map = indices_in_map_logical.nonzero()[0] # Find index of point

            # AND the vectors element-wise to find valid points, then get indices of all non-zero (True) values
            valid_point_indexes = (indices_in_map_logical*positive_indices_logical).nonzero()[0] # Get index of points that are both positive AND inside map


            #### Get probabilities from map ####
            # initialize list of probabilities to z_misc
            probabilities_from_points = np.ones(indices.shape[1], dtype=np.float64)*z_misc
```

```
            # Loop only through valid points
            for jj, point_index in enumerate(valid_point_indexes):
                # Get probability from map
                probFromMap = likelihood_map[indices[0,point_index], indices[1,point_index], indices[2,point_index]]

                # Multiply with probability factor and z_hit
                probabilities_from_points[point_index] = mapProbabilityFactor * self.__z_hit * np.float64(probFromMap)

                # Add z_misc
                probabilities_from_points[point_index] += z_misc


            # If sqSum, set prob from map as: sum(prob_i^2)/nPts_PC
            if sqSum:
                probability_from_map = (probabilities_from_points**2).sum()/indices.shape[1]
            else:
                # Find probability by taking product of all probabilities in array
                probability_from_map = probabilities_from_points.prod()

            # Take product of probabilities from map and assign weight to particle
            self.__particles[ii].weight *= probability_from_map

            # Add to sum of weight for normalization
            sum_weights += self.__particles[ii].weight


        # Normalize weights, also sets effective particles
        self.normalize_particle_weight(sum_weights)

    # Function to normalize particle weights
    def normalize_particle_weight(self, sum_weights):
        # Normalize particle weights and update effective particle set
        # initialize sum of squared weights
        sumSquaredWeights = 0

        for ii in range(self.__nParticles):
            # Normalize particle weights
            self.__particles[ii].weight = self.__particles[ii].weight/sum_weights
            #print(self.__particles[ii].weight)

            # Keep track of sum of squared weights (Neff = 1/(sum(weights^2)))
            sumSquaredWeights += self.__particles[ii].weight**2

            # Update vector with particle poses & weights
            self.__poseVec[:,ii] = self.__particles[ii].pose.copy().reshape(4)
            self.__weightVec[0, ii] = self.__particles[ii].weight

            # Print for debugging.
            #print(self.__particles[ii].weight)

        # Get effective number of particles
        self.__effectiveParticles = 1.0/(sumSquaredWeights)

    # Func to print particle position
    def printParticlePos(self):
        '''
            Function to Print particle positions in terminal
            Only used fo simplicity.

        '''
        for i in range(len(self.__particles)):
            print("Particle has pose: " )
            print(self.__particles[i].pose)
            print(" ")

    # Getters
    def getParticlePoseVector(self):
        return self.__poseVec.copy()

    def getParticleWeightVector(self):
        return self.__weightVec.copy()

    def getEffectiveSampleSize(self):
        '''
            Returns number of effective particles
        '''
        return self.__effectiveParticles
```

### B.4.3 Localization filter

```python
# Other imports
import numpy as np

# Importing pathlib
from pathlib import Path

# Own stuff
from idl_pf_pkg.JitParticleFilterClass import *
from idl_pf_pkg.PFTools import HistogramTools, PointCloudTools
from idl_botsy_pkg.droneConfiguration import DroneGeometry, Rot
from idl_botsy_pkg.filterConfig import MapConfig


class LikelihoodMap():
    def __init__(self):
        ##### Load map & Metadata #####

        # Make map config class, get name of map
        mapConfig = MapConfig()
        mapName = mapConfig.mapName

        # Get path to folder with this file
        home = str(Path.home())

        # Read map metadata
        map_metadata = np.load( home + "/colcon_botsy_idl/src/idl_pf_pkg/idl_pf_pkg/map/metadata_" + mapName + "_10cm_10cm.npy", allow_pickle=True) # Metadata is array of objects, allow_pickle must be true

        # Bool to signify if map uses uint8's for probability.
        self.mapUsingUint8Prob = np.bool(map_metadata[4,1])

        # Read map
        if self.mapUsingUint8Prob:
            self.map = np.load( home + "/colcon_botsy_idl/src/idl_pf_pkg/idl_pf_pkg/map/map_" + mapName + "_10cm_10cm.npy").astype(np.uint8)
        else:
            self.map = np.load( home + "/colcon_botsy_idl/src/idl_pf_pkg/idl_pf_pkg/map/map_" + mapName + "_10cm_10cm.npy").astype(np.float32)


        ## Parse metadata and save to variables
        # Cartesian offset of cell [0,0,0] from origin
        self.origin_offset = np.array([map_metadata[0,1]]).reshape(3,1)

        # Resolution of cells (side length) [m]
        self.resolution = np.float32(map_metadata[1,1])

        # Size of map in cells [x,y,z]
        self.size_in_cells =  np.int32(map_metadata[2,1])

        # Maximum value possible in map (from gaussian)
        self.mapMaxGaussVal = np.float32(map_metadata[3,1])


class LocalizationFilterParams():
    '''
        Class to hold parameters of Localization Filter
    '''

    def __init__(self):


        # Init drone params and rotator-class
        self.___drone_params = DroneGeometry()
        self.___rot = Rot()

        ############## Const rotations ##############
        self.rotMat_lf = self.___rot.rotX(np.pi)                    # Rotmat from level body to filter frame
        self.rotMat_nm = self.___drone_params.rotMat_nm             # Rotmat from ned to map
        #############################################

        ################## Camera ##################
        self.max_range = 15.0                                       # Max range reading of the depth sensor [m]
        self.pos_b_bc = self.___drone_params.pos_b_bc               # Translation from body frame to cam-frame
        self.rotMat_bc = self.___drone_params.rotMat_bc             # Rotmat from camera to body (camera pitch)

        # Sensor model data [in lack of a better name], NOTE: z_hit + z_rand/z_max = 1
        self.pf_z_hit = 0.8
        self.pf_z_rand = 0.2
        self.pf_z_max = 1.0
        ##############################################

        ############## Particle Filter ##############
        # PF Params
        self.number_of_particles = 1000
        self.init_pose = np.array([[0.0],[0.0],[0.0],[0.0]],dtype=np.float32)
        self.sigma_pose = np.array([[1.00],[1.00],[1.00],[6.00]],dtype=np.float32)
        self.nPts_PC = 30                # Number of points to sample from pointcloud
        self.pcUpdateSqSum = False       # Use Square sum method to update weights from pointcloud data

        # Use random points when downsampling
        # Selects nPts randomly, and then checks for duplicates and max_range measurements
        # deletes dupes and max_range measurements from pointcloud before passing on
        self.pcdsRandPoints = True

        # Select particles in a loop, checking each point for validity (range, dupe) before adding to an array
        # IF BOTH pcdsLoopSelect AND pcdsRandPoints IS SET TRUE, DEFAULTS TO LOOP CHECK MODE
        self.pcdsLoopSelect = False
        self.pcdsLoopSelMaxLoops = 2*self.nPts_PC

```

```python
        # Threshold for resampling (resample if effective sample size is less than threshold)
        self.resamplingThreshold = self.number_of_particles



        # Watchdog variables for when no new velocity message arrives, and the filter keeps predicting
        self.wd_counter = 0
        self.wd_counter_decayVelocityTresh = 3
        self.wd_K = 0.01

        ### Maximum values ##
        # Max value for velocity std dev
        self.maxVelStdCtr = np.float32(np.ones((4,1), dtype=np.float32)*2.0)

        # Const variance to add to propogation
        self.constVelVariance = np.array([[0.1],[0.1],[0.1],[0.1]], dtype = np.float32)

        ###############################################

        ############ Histogram smoothing ############

        # For histogram smoothing
        self.histogramResolution = np.array([[0.01],[0.01],[0.01],[0.01]], dtype=np.float32)
        kerLen = 5
        self.histSmoothingKernel = np.ones((1,kerLen), dtype = np.float32) / np.float32(kerLen)        # Simple averaging kernel with length 5

        ###############################################

        # initiate Vectors of Velocities, std.deviations and angles from "outside" of filter
        self.velVec_l = np.zeros((4,1), dtype=np.float32)
        self.velStd_l = np.zeros((4,1), dtype=np.float32)
        self.tHeta_bl = np.zeros((3,1), dtype=np.float32)

    def computeGaussianKernel(self, res, sigma):
        '''
            Function so compute a gaussian kernel to the filter, kernel will be 12*(sigma/resolution) + 1 long, (Center value + 6 sigma in each direction)

            Input:
                res      -   Resolution, length between different indices on kernel, float - unit: [m]
                sigma    -   Std deviation of kernel, float - unit: [m]

            Output:
                kernel   -   Normalized gaussian kernel, np.array - size: 1x(12*(sigma/resolution) + 1)
        '''
        # init kernel
        kernelLength = np.int32(12*sigma/res) + 1
        kernelCenter = np.int32((kernelLength-1)/2)
        kernel = np.zeros((1,kernelLength), np.float32)

        # Compute un-normalized kernel values
        for ii in range( np.int32( (kernelLength+1)/2 ) ):
            # Making use of symmety in kernel around center
            exponent = 1.0/2.0 * (res*ii)**2/(sigma**2)
            kernelValue = np.exp(-exponent)
            kernel[0,kernelCenter + ii] = kernelValue
            kernel[0,kernelCenter - ii] = kernelValue

        # Normalize kernel
        kernelSum = kernel.sum()
        kernel = kernel / kernelSum

        # Set gaussian kernel
        return kernel


class LocalizationFilter():

    def __init__(self, params = LocalizationFilterParams(), secondOrder = False, deltaPosition = False):
        '''
            Creates an instance of the localizationFilter

            input:
                Params      -   Class with filter parameters, should be of type LocalizationFilterParams()
        '''

        # Init drone params and rotator-class
        self.__drone_params = DroneGeometry()
        self.__rot = Rot()

        ###### Tools / manipulator classes ########

        self.__histTools = HistogramTools()
        self.__pcTools = PointCloudTools()

        ##########################################

        ####### Likelihood map ########

        self.__likelihood_map = LikelihoodMap()

        ##############################

        ############# Const rotations ##############
        self.__rotMat_lf = params.rotMat_lf                     # Rotmat from level body to filter frame
        self.__rotMat_nm = params.rotMat_nm                     # Rotmat from ned to map
        ##########################################

        ################## Camera ##################
        self.__max_range = params.max_range                     # Max range reading of the depth sensor [m]
        self.__pos_b_bc = params.pos_b_bc                       # Translation from body frame to cam-frame
        self.__rotMat_bc = params.rotMat_bc                     # Rotmat from camera to body (camera pitch)
```

```python

        # Sensor model data [in lack of a better name], NOTE: z_hit + z_rand/z_max = 1
        self.__pf_z_hit = params.pf_z_hit
        self.__pf_z_rand = params.pf_z_rand
        self.__pf_z_max = params.pf_z_max
        ##############################################

        ########### Pointcloud Operations ###########
        # Use random points when downsampling
        # Selects nPts randomly, and then checks for duplicates and max_range measurements
        # deletes dupes and max_range measurements from pointcloud before passing on
        self.__pcdsRandPoints = params.pcdsRandPoints

        # Select particles in a loop, checking each point for validity (range, dupe) before adding to an array
        # IF BOTH pcdsLoopSelect AND pcdsRandPoints IS SET TRUE, DEFAULTS TO LOOP CHECK MODE
        self.__pcdsLoopSelect = params.pcdsLoopSelect
        self.__pcdsLoopSelMaxLoops = params.pcdsLoopSelMaxLoops
        ###############################################

        ############## Particle Filter ##############
        # PF Params
        self.__number_of_particles = params.number_of_particles
        self.__init_pose = params.init_pose
        self.__sigma_pose = params.sigma_pose
        self.__nPts_PC = params.nPts_PC                   # Number of points to sample from pointcloud
        self.__pcUpdateSqSum = params.pcUpdateSqSum      # Use Square sum method to update weights from pointcloud data

        # Threshold for resampling (resample if effective sample size is less than threshold)
        self.__resamplingThreshold = params.resamplingThreshold

        # Watchdog variables for when no new velocity message arrives, and the filter keeps predicting
        self.__wd_counter = params.wd_counter
        self.__wd_counter_decayVelocityTresh = params.wd_counter_decayVelocityTresh
        self.__wd_K = params.wd_K
        self.__ekfLinearOnline = True

        # Const variance to add to the pose variance
        self.__constVelVariance = params.constVelVariance

        ### Maximum values ##
        # Max value for velocity std dev
        self.__maxVelStdCtr = params.maxVelStdCtr

        # Create instance of particle filter
        self.__particle_filter = ParticleFilter(    self.__pf_z_hit,
                                                    self.__pf_z_rand,
                                                    self.__pf_z_max,
                                                    self.__number_of_particles,
                                                    self.__init_pose,
                                                    self.__sigma_pose)

        self.__particle_filter.dry_run(self.__init_pose, self.__sigma_pose, self.__likelihood_map.map)
        ###############################################

        # For histogram smoothing
        self.__histogramResolution = params.histogramResolution
        self.__histSmoothingKernel = params.histSmoothingKernel

        # initiate Vectors of Velocities, std.deviations and angles from "outside" of filter
        self.__velVec_l_nb = params.velVec_l
        self.__velStd_l = params.velStd_l

        #### Second order propagation ####
        self.__secondOrder = secondOrder
        if secondOrder == True:
            self.__velVecLast_l_nb = params.velVec_l
            self.__velStdLast_l = params.velStd_l

        self.__tHeta_bl = params.tHeta_bl

        ###### Filter Pose and Variance ######
        self.__pose = self.__init_pose
        self.__poseVariance = self.__sigma_pose**2

        #### Delta position configuration
        self.__deltaPostition = deltaPosition
        if deltaPosition == True:
            self.__lastVelMsgTime = 0.0


    # Callers for PF
    def propagate(self, dt):
        '''
            Function to run propogation step of PF
            Rotates velocities from NED to PF propogation frame (Z-Up, X-Forward)

            input:
                dt      -   dt of propogation [float, - unit: s]
        '''

        ### Watch dog stuff ###
        # Increment watch dog counter
        self.__wd_counter += 1

        # If predicts since last velocity update > thresh, decay velocity
        if self.__wd_counter > self.__wd_counter_decayVelocityTresh:
            self.__velVec_l_nb *= (1 - self.__wd_K)
            self.__velStd_l *= (1 - self.__wd_K)

        # Delta position specific configuration
        if self.__deltaPostition == True:
            # If delta position, this converts back to velocity and resets the integrated value
            velVec_l_nb = self.__velVec_l_nb/dt
```

```python
            self.___velVec_l_nb = np.zeros((4,1))
        else:
            velVec_l_nb = self.___velVec_l_nb

        # Second order specific configuration
        if self.___secondOrder == True:
            # Velocity calculation for second order accuracy
            velVec_l_nb = 1.5*velVec_l_nb - 0.5*self.___velVecLast_l_nb
            self.___velVecLast_l_nb = velVec_l_nb

            # Covariance calculation to reflect second order integration
            velCov_l = 1.5*self.___velStd_l**2 + 0.5*self.___velStdLast_l**2
            velStd_l = np.sqrt(velCov_l)
            self.___velStdLast_l = velStd_l
        else:
            velVec_l_nb = velVec_l_nb
            velStd_l = self.___velStd_l

        # Add value to std.dev from number of predicts since last velocity update, up to a maximum
        # Std dev from message + const + (counter-1)*K
        stdDevFromCtr = np.minimum((self.___wd_counter-1)*self.___wd_K, (self.___maxVelStdCtr), dtype=np.float32)
        propStdDev = velStd_l + np.sqrt(self.___constVelVariance) + stdDevFromCtr


        # Rotate velocities into PF propogation frame (Z-Up, X-Forward)
        vel_PF_nb = np.zeros((4,1))
        vel_PF_nb[0:3,0] = self.___rotMat_lf @ velVec_l_nb[0:3,0]
        vel_PF_nb[3,0] = -velVec_l_nb[3,0]

        # Call propagate method of PF
        self.___particle_filter.propagate(vel_PF_nb, propStdDev, dt)

        if self.___deltaPostition == True:
            self.___velVec_l_nb = np.zeros((4,1))

    def pointcloud_update(self, pointcloud):
        '''
            Function to run pointcloud update step of PF.
            Downsamples and adjusts pointcloud into level body frame using roll/pitch angles from Kalman Filter.


            input:
                pointcloud      -   Pointcloud from ROS message, converted to np.array(3,N) with [X, Y, Z] along the columns
        '''

        # Downsample pointcloud
        pc_downsampled = self.___pcTools.downsample_pc_arr(  pointcloud,
                                                             max_range       = self.___max_range,
                                                             nPts            = self.___nPts_PC,
                                                             randPts         = self.___pcdsRandPoints,
                                                             loopSelect      = self.___pcdsLoopSelect,
                                                             maxLoopCount    = self.___pcdsLoopSelMaxLoops)

        # If pointcloud has size 1, no valid points were chosen
        if pc_downsampled.size != 1:
            # Transform pc to level body
            rotMat_bl = self.___drone_params.rotFun_bl(self.___tHeta_bl)
            pc_level = self.___pcTools.transform_pointcloud_to_level_body(   pc_downsampled,
                                                                             self.___pos_b_bc,
                                                                             self.___rotMat_bc,
                                                                             rotMat_bl,
                                                                             left_hand = False)

            # Transform pc to PF frame
            pc_PF = (self.___rotMat_lf @ pc_level).astype(np.float32)

            # Call update
            self.___particle_filter.pointcloud_update(   self.___likelihood_map.map,
                                                         self.___likelihood_map.origin_offset,
                                                         self.___likelihood_map.resolution,
                                                         self.___likelihood_map.size_in_cells,
                                                         self.___likelihood_map.mapMaxGaussVal,
                                                         self.___likelihood_map.mapUsingUint8Prob,
                                                        pc_PF,
                                                         self.___pcUpdateSqSum)

            # Resample
            if self.getPFEffectiveSampleSize() < self.___resamplingThreshold:

                self.___particle_filter.systematicResample()

    # Histogram smoothing and localization
    def localize(self):
        '''
            Function to find most likely localization from PF and its variance
            Uses kernel smoothing of pose histograms using a gaussian kernel
            Saves pose and pose variance to filter variables
        '''

        # Get histograms
        histogramX, histogramY, histogramZ, histogramPsi = self.___createPoseHistograms()

        # Smooth histograms
        histX, binsX = self.___histTools.smoothPoseHistogram(histogramX, self.___histSmoothingKernel, False)
        histY, binsY = self.___histTools.smoothPoseHistogram(histogramY, self.___histSmoothingKernel, False)
        histZ, binsZ = self.___histTools.smoothPoseHistogram(histogramZ, self.___histSmoothingKernel, False)
        histPsi, binsPsi = self.___histTools.smoothPoseHistogram(histogramPsi, self.___histSmoothingKernel, True)

        # Get indexes of max values from histograms
        histXMaxIdx = np.argmax(histX)
        histYMaxIdx = np.argmax(histY)
```

```python
        histZMaxIdx = np.argmax(histZ)
        histPsiMaxIdx = np.argmax(histPsi)

        # Take average of associated bin edges
        xPose = (binsX[histXMaxIdx] + binsX[histXMaxIdx + 1])/2.0
        yPose = (binsY[histYMaxIdx] + binsY[histYMaxIdx + 1])/2.0
        zPose = (binsZ[histZMaxIdx] + binsZ[histZMaxIdx + 1])/2.0
        psiPose = (binsPsi[histPsiMaxIdx] + binsPsi[histPsiMaxIdx + 1])/2.0

        # Collect most likely pose into an array
        pose = np.array([[xPose],[yPose],[zPose],[psiPose]], dtype = np.float32)

        # Find variance from pose
        poseVariance = self.__particle_filter.computePoseVariance(pose)

        # Set to filter variables, add some const variance to the poseVariance
        self.__pose = pose
        self.__poseVariance = poseVariance

    # Histogram func
    def __createPoseHistograms(self):
        '''
            Function to get histograms of the poses in the different directions,
            ** Not in separate JIT-class as numba (0.50) does not allow for weighted histograms
        '''

        # Get pose and weight vector from particle filter
        poseVec = self.__particle_filter.getParticlePoseVector()
        weightVec = self.__particle_filter.getParticleWeightVector()

        # Find number of bins needed to get the wanted resolution
        nBins = np.ones((4,1), dtype = np.int32)
        for ii in range(nBins.shape[0]):
            # Calculate number of bins for current axis from max and min value of poseVec
            nBins[ii,0] = np.int32(np.round_(((poseVec[ii,:].max() - poseVec[ii,:].min())/self.__histogramResolution[ii,0]) , decimals=0))

            if nBins[ii,0] < 1:
                # If number of bins less than one, add one to not break histogram
                # should only happen if data has essentially no spread, where one bin would be the correct amount
                nBins[ii,0] = 1

        # Create histograms
        histogramX = np.histogram(poseVec[0,:], bins = nBins[0,0], weights = weightVec[0,:])
        histogramY = np.histogram(poseVec[1,:], bins = nBins[1,0], weights = weightVec[0,:])
        histogramZ = np.histogram(poseVec[2,:], bins = nBins[2,0], weights = weightVec[0,:])
        histogramPsi = np.histogram(poseVec[3,:], bins = nBins[3,0], weights = weightVec[0,:])

        # Return histograms
        return histogramX, histogramY, histogramZ, histogramPsi

    # Reset func
    def resetParticleFilterToInitPose(self):
        '''
            Function to reset Particle filter to initial pose
        '''
        # Call PF resetter func with initial pose
        self.__particle_filter.reset_filter(self.__init_pose, self.__sigma_pose)

    # Public setters
    def setEKFLinearOnlineState(self, status):
        # Sets status of ekf
        self.__ekfLinearOnline = status

    def setPFParams(self, params):
        # Method to set params to PF from params "struct"
        test = 0

    def setHistogramSmoothingKernel(self, kernel):
        # Set kernel for histogram smoothing
        self.__histSmoothingKernel = kernel

    def setPropagationVelocityWithCovariance(self, vel_vec, cov_vec, time):
        '''
            Set velocity to be used in particle propogation, velocity should be in level body frame, [x, y, z, Psi] column vector
            Set velocity standard deviation to be used in particle propogation, cov should be in level body frame, [x, y, z, Psi] column vector

            input:
                vel_vec     -   [4x1] Vector of floats, containing velocities in [x, y, z, Psi] in level frame
                cov_vec     -   [4x1] Vector of floats, containing cov in [x, y, z, Psi] in level frame
        '''

        if self.__deltaPostition == True:
            # Calculates msg dt
            # gets time
            dt = time - self.__lastVelMsgTime
            self.__lastVelMsgTime = time
            if dt >= 1.0:
                dt = 0.0
                print('wrn: PF_Ros_noe: deltaPose dt not valid')

            self.__velVec_l_nb += vel_vec*dt
        else:
            self.__velVec_l_nb = vel_vec

        # Sets std
        self.__velStd_l = np.sqrt(cov_vec)

        # Resets wd counter if linear part of EKF is online
        if self.__ekfLinearOnline:
            self.__wd_counter = 0
```

```
    def setPropagationStdDev(self, std_vec):
        '''
            Set velocity standard deviation to be used in particle propogation, std.dev should be in level
  body frame, [x, y, z, Psi] column vector

            input:
                std_vec     -   [4x1] Vector of floats, containing std.dev in [x, y, z, Psi] in level
 frame
        '''
        self.__velStd_l = std_vec

    def setCurrentRollPitchAngles(self, tHeta_bl):
        '''
            Sets angles tHeta_bl (from body to level) shape [3,1], (Roll, Pitch, Yaw) to filter
        '''
        self.__tHeta_bl = tHeta_bl

    # Public getters
    def getPFEffectiveSampleSize(self):
        # Return PF effective sample size
        return self.__particle_filter.getEffectiveSampleSize()

    def getPFParticlePoseVector(self):
        return self.__particle_filter.getParticlePoseVector()

    def getPFParticleWeightVector(self):
        return self.__particle_filter.getParticleWeightVector()

    def getFilterPoseNED(self):
        # Convert to NED frame
        pose_n_nb = np.zeros((4,1))
        pose_n_nb[:3,0] = self.__rotMat_nm.T @ self.__pose[:3,0]
        pose_n_nb[3,0] =  (np.pi/2 - self.__pose[3,0]) % (2.0*np.pi)

        return pose_n_nb.astype(np.float32)

    def getFilterVarianceNED(self):

        # Convert to NED frame, X- and Y- changes place
        nedVar = np.zeros((4,1), dtype=np.float32)
        nedVar[0,0] = self.__poseVariance[1,0]
        nedVar[1,0] = self.__poseVariance[0,0]
        nedVar[2:,0] = self.__poseVariance[2:,0]

        return nedVar
```

### B.4.4 Particle filter tools

```python
# Other imports
import numpy as np

# Numba
from numba import int32, float32, jit, types, typed, typeof  # import the types
from numba.experimental import jitclass


@jitclass([])
class HistogramTools():
    '''
        Class containing methods to manipulate histograms
    '''
    def __init__(self):
        # Dry run to test functions
        self.dry_run()

    def dry_run(self):
        # Test params
        testPose = np.ones((1,10), dtype = np.float32)
        testPose[0,5] = np.float32(2.0)
        testKernel = np.ones((1,3),dtype = np.float32)

        # Run funcs
        hX = np.histogram(testPose)
        test1, test2 = self.smoothPoseHistogram(hX, testKernel, False)

    def smoothPoseHistogram(self, hist, kernel, wrapped = False):
        '''
            Function to smooth histogram

            inputs:
                hist        -   Histogram, [hist, bins] as given from hist = np.histogram()
                kernel      -   Kernel to smooth with, should be a np.array of size [1xN] where N is odd
                wrapped     -   True/False if the histogram is wrapped (i.e 0 - 2pi)

            output:
                smoothedHist-   smoothed histogram
                binEdges    -   Edges of the bins in the histogram
        '''

        # Get histogram and bin edges
        histogram = hist[0].reshape((1, hist[0].shape[0]))
        binEdges = hist[1]

        # Get length of histogram & kernel
        histLen = np.int32(histogram.shape[1])
        kernelLen = np.int32(kernel.shape[1])

        kernelPad = kernelLen - 1       # Odd kernel length -> (kernelLen - 1)/2 extra on each side

        # Init smoothed and padded hist as zeros
        smoothedHist = np.zeros((1, histLen), dtype=np.float32)
        paddedHist = np.zeros((1, histLen + kernelPad), dtype = np.float32)

        # Find offset from kernel size
        kernelOffset = np.int32((kernelPad)/2)

        # insert histogram into padded histogram
        paddedHist[0, kernelOffset:histLen + kernelOffset] = histogram[0,:]

        # If wrapped 0...2pi and histogram endpoints are within some % of edge values,
        # pad histogram with values on opposite ends of original histogram
        if wrapped:

            # Calculate thresholds for wrapping
            yawWrapWindow = 0.05                            # 5 % of 2*pi
            upperThresh = np.pi*2.0*(1-yawWrapWindow)
            lowerThresh = np.pi*2.0*yawWrapWindow

            if binEdges.max() > upperThresh and binEdges.min() < lowerThresh:
                # If extreme-values of histogram are within a certain threshold of % 2pi, pad with values
    from opposite side of histogram
                paddedHist[0, 0:kernelOffset] = histogram[0, -kernelOffset:]
                paddedHist[0, -kernelOffset:] = histogram[0, 0:kernelOffset]


        # For loop to dot kernel with part of histogram
        for ii in range(histLen):
            # Get slice of histogram data
            histData = (paddedHist[0,ii:ii+kernelLen]).reshape(1,kernelLen)

            kernel = kernel

            # Dot product between kernal and histogram slice
            smoothedHist[0,ii] = (kernel*histData).sum()

        return smoothedHist, binEdges

class PointCloudTools():
    '''
        Class containing methods to manipulate pointclouds
    '''
    def __init__(self):
        # Default contructor

        # Run dry_run func
        self.dry_run()

    def dry_run(self):
        # Dry running funcs
        pc_test = np.ones((1,3))
```

```
        # Run funcs
        dry_run_downsample = self.downsample_pc_arr(pc_test)
        test = self.transform_pointcloud_to_level_body(dry_run_downsample)

    def downsample_pc_arr(self, pc_arr, max_range=15.0, nPts=0, randPts=False, loopSelect=False, maxLoopCount=None):
        '''
            Input:
                pc_arr          -   np.array with pointcloud data
                nPts            -   Number of points from the cloud to "keep" (linspaced through all points which are not NaN) 0 keeps all points
                randPts         -   Picks points randomly using a uniform distribution
                loopSelect      -   Bool to signify that we want to use a loop to check for valid points
                maxLoopCount    -   Maximum number of times to loop when finding points using loopSelect

            Output:
                pointcloud_filtered     -   Np array with coordinated of all points in PC, size: (3 x n)
                                            where n is amount of points in PC

        '''

        # Initialize data to 0
        pointcloud_filtered = 0

        # Init bool to signify if the values are verified before reaching the end, if the
        # values are verified, then you won't need to loop and check for uniqueness and that dist < max_range
        isChecked = False

        # If loopSelect is true and maxLoopCount not set, set maxLoopCount equal to nPts
        if loopSelect and (maxLoopCount is None):
            maxLoopCount = nPts

        # Only run if there are valid points in xyz_data
        if pc_arr.size != 0:
            if nPts == 0:
                # If input nPts to use is zero, use entire pointcloud
                pointcloud_filtered = pc_arr

            elif not randPts and not loopSelect:
                # Else if randPts is not set to true, use linspace to pick points
                indexes = np.linspace(0, pc_arr.shape[0], nPts, endpoint=False, retstep=False, dtype=np.int32)
                pointcloud_filtered = pc_arr[indexes,:]

            elif loopSelect:

                # Init vector of indexes
                idxVec = np.empty(0, dtype=np.int32)
                loopCounter = 0

                # Loop untill while is broken by either:
                # Enough points are found [idxVec.shape[0] < nPts], or
                # The number of tested points exceed maxLoopCount [loopCounter < maxLoopCount]
                while (loopCounter < maxLoopCount) and (idxVec.shape[0] < nPts):
                    # Increment counter
                    loopCounter += 1

                    # Get a random index from the cloud
                    idx = np.random.randint(0, pc_arr.shape[0], dtype=np.int32)

                    # Get point from cloud
                    point = pc_arr[idx,:]

                    # Check if closer than max-range
                    dist = np.sqrt(np.sum(point**2))

                    if dist < max_range:
                        # Check if idx is in idxVec
                        isIn = np.isin(idx, idxVec)

                        # If it's not in, add to idxVec
                        if not isIn:
                            idxVec = np.append(idxVec, idx)

                # Set isChecked True, as all indexes are unique and closer than max_range
                isChecked = True

                # Extract values
                pointcloud_filtered = pc_arr[idxVec,:]


            else:
                # Get nPts randomly selected points from PC array
                randIdx = np.random.randint(0, pc_arr.shape[0], nPts)

                # Ensure only unique points selected
                randIdx = np.unique(randIdx)

                # Slice array to get points
                pointcloud_filtered = pc_arr[randIdx,:]


            if isChecked is False:
                # Initialize list to keep indexes of points at max range
                points_at_max_range = []

                for ii in range(pointcloud_filtered.shape[0]):
                    # Check if reading is at max dist
                    point_dist = np.sqrt(np.sum(pointcloud_filtered[ii,:]**2))

                    # If distance is greater than max range
                    if point_dist > max_range:
```

```python
                        points_at_max_range.append(ii)

                # Delete max range readings
                # this simply removes the max range points, meaning the pointcloud is no longer nPts big
                pointcloud_filtered = np.delete(pointcloud_filtered, points_at_max_range, axis=0)


        # Return transposed data, to get a [3xN] list
        return np.transpose(pointcloud_filtered).astype(np.float32)

    def transform_pointcloud_to_level_body(self, pointcloud_array, pos_b_bc=np.zeros((3,1)), rotMat_bc=np.eye(3), rotMat_bl=np.eye(3), left_hand=False):
        '''
            Input:
                pointcloud_array        -   pointcloud in np.array form [x, y, z]... camera frame (Z-Forward, X-Down)
                pos_b_bc                -   translation from body to camera [x_t, y_t, z_t]
                rotMat_bc               -   Rotation matrix from body to camera
                rotMat_bl               -   Rotation matrix from body to level
                left_hand               -   if the coordinate frame is lefthanded

            Output:
                pointcloud_transformed  -   pointcloud transformed into level body frame [X-Forward, Z-Down]

            Function to transform pointcloud into "artificially level body frame" where the roll and pitch angle from the
            kalman filter is used to straighten up the pointcloud for use with the map.

        '''
        if left_hand:
            # If lefthanded coordinate system, flip X-Coordinates
            #print(pointcloud_array[0][:])
            pointcloud_array[0][:] = -pointcloud_array[0][:]

        # Rotate pointcloud into body frame [X-Forward, Z-Down]
        pointcloud_rotated = rotMat_bc @ pointcloud_array

        # Translate pointcloud from camera frame to level frame
        pointcloud_transformed = rotMat_bl @ (pos_b_bc + pointcloud_rotated)

        return pointcloud_transformed.astype(np.float32)
```

## B.5 idl_map_pkg

### B.5.1 Likelihood field generation tool

```python
import numpy as np
import os
#import octomap
import open3d as o3d
import easygui

# Get path to current file directory
cwd = os.path.dirname(os.path.abspath(__file__))


def getMeshFromModel_GUI():
    '''
        Input:

        Output:
            mesh    -   instance of mesh read from model

        Opens a GUI instance, takes in a mesh in the form of .stl, .ply or other supported mesh files
        (http://www.open3d.org/docs/release/tutorial/geometry/file_io.html)
    '''

    # Use easygui to get file
    meshFile = easygui.fileopenbox(msg=None, title=None, default=cwd + "/../models/")

    # Import ply model as triangular mesh
    mesh = o3d.io.read_triangle_mesh(meshFile)

    return mesh

def createPCDFromModel_GUI(nPoints):
    '''
        Input:
            nPoints -   Number of points to uniformally sample the mesh with

        Output:
            pcloud  -   Pint cloud with nPoints sampled uniformly over chosen model

        Opens a GUI instance, takes in a mesh in the form of .stl, .ply or other supported mesh files
        (http://www.open3d.org/docs/release/tutorial/geometry/file_io.html)
    '''
    # Open gui instance to get mesh
    mesh = getMeshFromModel_GUI()

    # Sample point off the mesh uniformally
    pcloud = o3d.geometry.TriangleMesh.sample_points_uniformly(mesh, nPoints)

    return pcloud

def createKDTreeFromMesh_GUI(nPoints):
    '''
        Input:
            nPoints -   Number of points to uniformally sample the mesh with

        Output:
            kdtree  -   Instance of kd-tree

        Opens a GUI instance, takes in a mesh in the form of .stl, .ply or other supported mesh files
        (http://www.open3d.org/docs/release/tutorial/geometry/file_io.html)
    '''

    # Get Point cloud
    pcloud = createPCDFromModel_GUI(nPoints)

    # Create KD-Tree
    kdtree = o3d.geometry.KDTreeFlann(pcloud)

    return kdtree

def createKDTreeFromMesh(mesh, nPoints):
    '''
        Input:
            mesh    -   Mesh to create KD-tree from
            nPoints -   Number of points to uniformally sample the mesh with

        Output:
            kdtree  -   Instance of kd-tree

        Opens a GUI instance, takes in a mesh in the form of .stl, .ply or other supported mesh files
        (http://www.open3d.org/docs/release/tutorial/geometry/file_io.html)
    '''

    # Sample points off the mesh uniformally
    pcloud = o3d.geometry.TriangleMesh.sample_points_uniformly(mesh, nPoints)

    # Create KD-Tree
    kdtree = o3d.geometry.KDTreeFlann(pcloud)

    return kdtree

def createOctomapFromMesh_GUI(res, nPoints):

    '''
        Input:
            res     -   desired map resolution
            nPoints -   Number of points to uniformally sample the mesh with

        Returns
```

```
            omap    -   Instance of octomap

        Opens a GUI instance, takes in a mesh in the form of .stl, .ply or other supported mesh files
        (http://www.open3d.org/docs/release/tutorial/geometry/file_io.html)
    '''

    # Setup the tree with desired resolution
    omap = octomap.OcTree(res)

    # Sample points off the mesh uniformally
    pcloud = createPCDFromModel_GUI(nPoints)

    # Convert pointcloud to array
    pcloud_np = np.asarray(pcloud.points)

    # Insert point cloud into the octree (array is "sensor origin")
    omap.insertPointCloud(pcloud_np ,np.array([0.0, 0.0, 0.0]))

    return omap

def createOctomapFromMesh(mesh, res, nPoints):
    '''
        Input:
            res     -   desired map resolution
            nPoints -   Number of points to uniformally sample the mesh with

        Returns
            omap    -   Instance of octomap

        Opens a GUI instance, takes in a mesh in the form of .stl, .ply or other supported mesh files
        (http://www.open3d.org/docs/release/tutorial/geometry/file_io.html)
    '''

    # Setup the tree with desired resolution
    omap = octomap.OcTree(res)

    # Sample points off the mesh uniformally
    pcloud = o3d.geometry.TriangleMesh.sample_points_uniformly(mesh, nPoints)

    # Convert pointcloud to array
    pcloud_np = np.asarray(pcloud.points)

    # Insert point cloud into the octree (array is "sensor origin")
    omap.insertPointCloud(pcloud_np ,np.array([0.0, 0.0, 0.0]))

    return omap

def createGridMap(mesh, resolution, nPoints, sig_sens):
    '''
        Input:
            mesh        -   mesh to base map on (Open3D triangle mesh)
            resulotion  -   Map resolution (float)
            nPoints     -   number of points to sample the map with

        Returns:
            mapArray    -   3D array containing distances to closest points in model
            minBound    -   array with the offset from model origin to cell [0, 0, 0]


        Function to create a gridmap (3D-Array) where each voxel contains the distance to the closest point in the map
        (Prelude to Likelihood-map)

        TODO: Automatically calculate nPoints from resolution and size of mesh.
    '''
    # Get boundary box of model
    maxBound = np.array(mesh.get_max_bound(), dtype=np.float32) + 6*sig_sens
    minBound = np.array(mesh.get_min_bound(), dtype=np.float32) - 6*sig_sens
    [x_size, y_size, z_size] = np.round_(maxBound - minBound, 3)
    boundaryBox = np.array([x_size, y_size, z_size])
    print("Maxbound: " + str(maxBound))
    print("Minbound: " + str(minBound))
    print(boundaryBox)

    # Find size of array to cover the boundary-box with voxels at given resolution
    [x_idxSize, y_idxSize, z_idxSize] = np.int32(np.ceil(boundaryBox/resolution))
    gridSize = np.array([x_idxSize, y_idxSize, z_idxSize], dtype=np.int32)
    print("Size of map in [cells]" + str(gridSize))

    # Get KD-tree from mesh
    kdTree = createKDTreeFromMesh(mesh, nPoints)

    # Setup 3D array for map with given resolution
    mapArray = np.ones(gridSize)

    # Create vectors of all voxel center coordinates
    '''
        Linspace(l, u, n) splits [l, u] into n equally sized pieces, add resolution/2 to get voxel center coordinate
        np.round_(num, dec) rounds num to dec decimal places
    '''
    linspaceXVoxelCenter = np.round_(np.linspace(minBound[0], maxBound[0], x_idxSize, endpoint=False) + resolution/2, 3)
    linspaceYVoxelCenter = np.round_(np.linspace(minBound[1], maxBound[1], y_idxSize, endpoint=False) + resolution/2, 3)
    linspaceZVoxelCenter = np.round_(np.linspace(minBound[2], maxBound[2], z_idxSize, endpoint=False) + resolution/2, 3)

    print("Beginning loop, this might take a while...")

    # Get distances from center of each voxel to nearest point in space
    for idx_z, z in enumerate(linspaceZVoxelCenter):
        print("Status: " + str(np.round_((idx_z / z_idxSize * 100),2)) + " Percent finished")
        for idx_y, y in enumerate(linspaceYVoxelCenter):
            for idx_x, x in enumerate(linspaceXVoxelCenter):
```

```python
                # Search KD-tree for closest neighbour in all voxels

                [k, idx, sqdist] = kdTree.search_hybrid_vector_3d([[x], [y], [z]], 6*sig_sens, 1)

                # If nothing is found withing 6*sig_sens, sqdist will be an empty vector
                # Appending a value of ((6*sig_sens)**2) to the end of the list will fix the problem arising
                # when no points are found within max search range, 6 sigma is chosen because then the probability
                # of hit will be essentially 0
                sqdist.append((6*sig_sens)**2)

                # Appending is done because then no check has to be made, if something is found within 6*sigma,
                # The appended value will have index [1], and not be used, if nothing is found, it will
                # have index [0], and be used.

                # Find distance to closest point by taking the square root of sqdist[0]
                dist = np.sqrt(sqdist[0])
                mapArray[idx_x][idx_y][idx_z] = dist
                #print("X: " + str(x) + "    Y: " + str(y) + "    Z: " + str(z) + "    Dist: " + str(dist))

    print("Loop finished!")

    #print(gridSize)

    mapMetaData = np.array([["Coordinate of cell [0, 0, 0] relative to origin [in m]", minBound],
                            ["Resolution of map [in m]", resolution],
                            ["Size of Gridmap [in cells]", gridSize]],dtype=object)

    return mapArray, mapMetaData

def generateLikelihoodMap(distmap, sigma_sens, uint8Map = False):
    '''
        Input:
            distmap         -   Map with distances to closest point in mesh (3D np.array)
            sigma_sens      -   Standard deviation of sensor used (float)
            uint8Map        -   Bool to say if probabilities are to be saved as uint8's (0-255, to be parsed)

        Output:
            likelihoodMap   -   Gridmap with likelihoods [3D np.array]
            maxVal          -   Max value of gaussian

        Function to generate a likelihood-map from a distance-map
    '''
    # Find maximal possible value of gaussian
    maxVal = 1.0/(sigma_sens*np.sqrt(2.0*np.pi))

    # Create likelihood map
    likelihoodMap = maxVal * np.exp(-1.0/2.0*(distmap/sigma_sens)**2)

    if uint8Map:
        # If the data type specified is uint8, scale map to the interval [0, 255]
        likelihoodMapScaled = likelihoodMap * 255/maxVal

        # Round off to 0 decimals
        likelihoodMapScaled = np.round_(likelihoodMapScaled, decimals=0)

        # Cast to uint8
        likelihoodMap = likelihoodMapScaled.astype(np.uint8)


    return likelihoodMap, maxVal

def saveOctoMap(tree, name):
    '''
        Input:
            tree    -   Octomap to save
            name    -   Filename to save the map as
    '''
    if tree.write(name.encode('utf-8')): # Write binary OctoMap file (Only encode "occupied", "free" and "unknown")
        print("Octomap Created from file at chosen path")
    else:
        print("Cannot create octree file.")

if __name__ == "__main__":

    # Get mesh from model
    mesh = getMeshFromModel_GUI()

    # Setting resolution and number of points to sample mesh with
    resolution = np.float32(easygui.enterbox(msg="Resolution of map (voxel side-length), [m]", default=0.1))
    nPoints = 10000000
    sig_sens = np.float32(easygui.enterbox(msg="Std deviation of map gaussian, [m]", default=0.1))
    mapDataType = np.float32(easygui.enterbox(msg="Datatype to use (float32 / uint8): [0: float32, 1: uint8]", default=0))
    mapDataType = (mapDataType > 0.5) # As easygui only returns strings, a check is done to get the desired bool

    # Name files
    mapName = str(easygui.enterbox(msg="Desired map name", default="map"))
    metaDataName = str(easygui.enterbox(msg="Desired metadata name", default="metadata"))

    # Create grid map
    arrayMap, mapMetaData = createGridMap(mesh, resolution, nPoints, sig_sens)

    # Convert to likelihood map
    likmap, maxVal = generateLikelihoodMap(arrayMap, sig_sens, uint8Map=mapDataType)

    # Append items to mapMetaData
    maxValEntry = np.array(["Maximum value of map gaussian", maxVal], dtype=object)
    dataTypeEntry = np.array(["Uint8 used in map (True / False :: uint8 / float32)", mapDataType], dtype=
```

```
     object)
    appendArr = np.array([   [maxValEntry],
                             [dataTypeEntry]], dtype=object).reshape((2,2))

    mapMetaData = np.append(mapMetaData, appendArr, axis=0)

    print(mapMetaData)

    #octomap = createOctomapFromMesh(mesh, resolution, nPoints)
    #saveOctoMap(octomap, "likelihoodMap.ot")
    #arrayMap = np.load("test.npy")

    # Save 3D numpy array
    np.save(cwd + "/../maps/" + mapName ,likmap)
    np.save(cwd + "/../maps/" + metaDataName, mapMetaData)
```

### B.5.2 Map slicer tool

```
import numpy as np
import easygui
import png
import os

# Path to current folder
cwd = os.path.dirname(os.path.abspath(__file__))

# Load map
mapFile = easygui.fileopenbox(msg="Choose map", title=None, default=cwd + "/../maps/")
likMap = np.load(mapFile)

# Load metadata
metaDataFile = easygui.fileopenbox(msg="Choose metaData", title=None, default=cwd + "/../maps/")
metaData = np.load(metaDataFile, allow_pickle=True)

# Parse needed stuff from metadata
mapOffset = metaData[0,1]
mapRes = metaData[1,1]

# Set height of desired slice
sliceHeight = np.float32(easygui.enterbox(msg="Height at which to slice map, [m]"))

# Get slice
sliceIdx = np.int32(np.round_((sliceHeight-mapOffset[2])/mapRes, decimals = 0))
mapSlice = np.sqrt(likMap[:,:,sliceIdx])
maxVal = np.amax(mapSlice)
mapSlice = mapSlice/maxVal * 255

# Convert to uint8
mapSliceImgArray = mapSlice.astype(np.uint8)

imageName = str(easygui.enterbox(msg="Name of .png image [do not add .png at the end]"))

# Save image
png.from_array(mapSliceImgArray, mode="L").save(cwd + "/../mapSlices/" + imageName + ".png")
```

Page intentionally left blank