Sensor Fusion with Out-of-Sequence Measurements

Localization in an Agricultural Robot Using
Visual Odometry

Mathias Hauan Arbo

Master of Science in Cybernetics and Robotics

Submission date:	June 2015
Supervisor:	Jan Tommy Gravdahl, ITK
Co-supervisor:	Trygve Utstumo, Adigo AS

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Norges teknisk-naturvitenskaplige Universitet
IME-fakultetet
Institutt for Kybernetikk og robotikk

Master Thesis TTK4900

Student:	Mathias Hauan Arbo
Program:	M.Sc. Engineering Cybernetics
Title:	Sensor fusion with Out-Of-Sequence-Measurements Localization of an agricultural robot using visual odometry

Adigo is developing an autonomous robot for weed control in row crops, Asterix. A vision system classifies weeds and crop, and targets each weed leaf with an individual droplet of herbicide. The system requires a highly accurate position and movement estimate to accurately target individual leaves. The robot is equipped with an RTK GPS unit, IMU, wheel encoders and a camera system which can be utilized for visual odometry.

The nature of visual odometry (VO) measurements is highly different from measurements from an IMU or wheel encoder, which are near instantaneous and frequent. The processing time of VO introduces a delay, and the frequency of measurements are limited both by the imaging system and processing time. These measurements can be described as out-of-sequence measurements (OOSM).

Sensor fusion with OOSM has been presented in numerous ways in literature, using various assumptions and approaches. A central topic in this project is to evaluate solution strategies for the delayed sensor fusion problem.

The main tasks in this projects are:
- Present an overview of existing work on delay fusion for localization.
- Evaluate the Stochastic Cloning strategy in a Bayesian framework.
- Construct a delay fusion strategy for use with an unscented Kalman filter.

Advisor:	Jan Tommy Gravdahl Professor, Dept. of Engineering Cybernetics Edmund Brekke Professor, Dept. of Engineering Cybernetics
External advisor:	Trygve Utstumo Ph.D. candidate, Dept. of Engineering Cybernetics MSc. Engineering Cybernetics, Adigo AS
Project assigned:	January 2015
To be handed in by:	June 12, 2015

Trondheim, Januar 2015

Jan Tommy Gravdahl
Professor, Dept. Engineering Cybernetics

Abstract

Timing is an important aspect when working with visual odometry (VO). The time of processing and data transfer of image based measurements are significantly longer than IMU, wheel encoders, and GPS measurements. This introduces an arrival latency, causing the VO measurements to require delay fusion, a subcategory of out-of-sequence measurement (OOSM) fusion. In cooperation with Adigo AS this thesis has focused on the Asterix project, where an agricultural robot uses a downward facing camera for visual-inertial odometry to aid in localization. The main focus in this work is with the delay fusion problem.

By approaching the OOSM fusion with a Bayesian framework, the theory chapter presents a method of fusing delayed displacement measurements. This can be considered a generalization of the stochastic cloning approach. A byproduct of the investigation is an unscented multiple-point smoother capable of defining fixed-points to be smoothed on demand.

Simulations and experiments showed that the OOSM fusion methods worked, but model inconsistencies and inaccuracies in the VO measurements negatively affected the results.

Sammendrag

Timing er et viktig aspekt ved sensorfusjon av visuell odometri (VO) målinger. Prosessering og dataoverføring av bildebaserte målinger er betydelig lengre enn IMU, hjulenkodere, og GPS-målinger. Dette introduserer en forsinkelse som fører til at VO målingene må håndteres annerledes under fusjon. Denne typen sensorfusjon er en underkategori av ”out-of-sequence measurement” (OOSM) fusjon. I samarbeid med Adigo AS har denne avhandlingen fokusert på Asterix prosjektet der en landbruksrobot bruker et nedadvendt kamera for ”visual-inertial” odometri for lokalisering. Hovedfokuset har vært på det teoretiske grunnlaget for forsinket fusjon.
Teorikapittelet introduserer fusjon av forsinkede målinger i et i et Bayesiansk rammeverk hvor stokastisk kloning er et spesialtilfelle. Et produkt av denne undersøkelsen er en ”unscented multiple-point smoother”. Det er et ”fixed-point smoothing” filter hvor de glattede punktene kan bli definert etter behov mens filteret kjører.
Simuleringer og datasettene viste til at OOSM metodene fungerte, men at filteret var veldig sensitiv til forskjeller i den observerte tilstanden og den modellerte tilstanden. Ved for lav prosesstøy klarer ikke filteret følge raske endringer, men med for høy prosesstøy gir ikke VO målinger forbedring i tilstandsestimatet.

Acknowledgement

After approximately one year and 50 litres of coffee, my master thesis has come to an end. I am sincerely grateful for the discussions and guidance provided by my supervisors: Jan Tommy Gravdahl, Edmund Brekke and Trygve Utstumo. I am also grateful to Programvareverkstedet at NTNU, a constant source of good, fun, infuriating, and intriguing conversations. Gratitude must also be extended to Adigo AS who formulated this interesting problem, gave me hands-on experience with their robots during a summer internship, and provided helpful cooperation.

Mathias Hauan Arbo
Trondheim
June 12, 2015

Contents

1 Introduction

1.1 Precision Agriculture

The world's population is increasing, and so does the demand for nutrition. Projections from 2009 indicate that feeding a population of 9.1 billion people in 2050 requires raising food production by 70 percent (FAO, 2009). To be able to sustain the growing world, agriculture has to keep up the development of efficient and productive techniques.

Typical tasks encountered in the agricultural industry involve identification and sorting of various produce, picking, harvesting, mowing and crop control. Many of these tasks are easy for humans to do, but difficult to achieve with robots as there is great uncertainty in the objects with which to interact and the environment of the task.

One of the bottlenecks in automated agriculture is weed control. It requires extensive manual labor and uses pesticides and herbicides that are potentially harmful to both farmers and the environment. To use robotics in a setting as complex and varied as a field of carrots, high precision is required to ensure on demand control, only where necessary.

Generally weed control has been done by uniform application of herbicides to the entire field of crops. Precision agriculture intends to identify specific problem areas and only apply herbicide as appropriate. This requires accurate knowledge of the robot's configuration with respect to its environment, a task only possible with computer vision.

1.2 Visual Odometry

Odometry stems from *odos* meaning route and *metros* meaning measure. It is the act of using sensors to measure the configuration of the robot in a predefined map. This term will be used interchangeably with localization. In this thesis we call configuration the current state of the robot (position, heading and velocities). Pose is a combination term used for position and heading.

Visual odometry (VO) uses feature points from two or more images to reconstruct the movement between the two images. It does so by identifying the motion of the observing camera as a translation and rotation. This is similar to wheel odometry (WO) which uses the distance traversed by the wheels to reconstruct the movement of the robot.

VO techniques have different criterions for ensuring optimal results. The surface observed has to contain sufficient features, and as the operations performed are more cumbersome than wheel odometry the estimates are significantly delayed for real-time applications.

The term visual odometry (VO) was first introduced in 1996 by Srinivasan to define motion orientation in honey bees. However something akin to real-time stereo VO was first implemented in 1980 by Moraveck for NASA's Jet Propulsion Laboratory. The method of VO is especially important on Mars where GPS is not available, and other means of reliable navigation are necessary. Between 1980 and 2000, VO research was dominated by NASA and JPL. In 2004, the research proved successful when the Mars rovers Spirit and Opportunity both used VO for navigation on Mars. The term was further popularized in the academic environment by the publication of "Visual Odometry" (Nister et al., 2004) which presented a brief overview of the technique.

In recent years VO has seen a resurgence in research as an alternative or complement to simultaneous localization and mapping (SLAM). Whereas VO aims at recovering the motion of an agent with focus on local consistency, V-SLAM aims at recovering the map an agent moves in with a focus on global consistency. An example of this is Project Tango, a research project by Google that has created a development platform, a cellphone with two cameras and IMUs, for visual odometry and visual navigation. Nasa's Mars Science Laboratory is, after their successes with Curiosity and Spirit, continuing their research on VO and V-SLAM.

For fast localization filters, VO measurements are intermittent and delayed. The processing time for images is significantly longer than wheel encoders or similar sensors. This delay means that the VO measurements do not describe the current configuration of the system, but rather how it was some time ago. This thesis discusses how to handle these delayed VO measurements with respect to Asterix, an agricultural robot in development at Adigo AS.

1.3 The System and Related Work

Asterix is a research and development project at Adigo AS in cooperation with Bioforsk. It is led by Trygve Utstumo, an industry PhD candidate at Dept. of Cybernetics at NTNU. The project entails creating a high-precision crop control robot. Tasks of the robot are: to follow rows of carrot crops, identify weeds, and spray herbicide on demand.

To spray the correct weeds in a row of crops, the robot needs to maintain an estimate of its position relative to a set of identified weeds. This means it hopes to maintain a locally accurate estimate of its position. To be completely autonomous, the robot must also be able to turn into rows and follow the rows of the field. This means that one must either maintain a globally accurate position estimate or have a row following controller based on a forward facing camera.

Much of the basic framework for the robot has already been investigated.
In 2013, Alexander Tallund Klungerbo did his master thesis on the drop-on-

demand system with respect to the requirements of a nozzle system for necessary precision (Klungerbo, 2013). In 2014, Øystein Grændsen did his master thesis on automatic visual weed recognition using machine vision and artificial intelligence (Grændsen, 2014). Frode Urdal designed a precision spray matrix with focus on real-time control of the spray nozzles (Urdal, 2013). This semester Jarle Dørum implemented a row following controller that uses visual plant recognition and a forward facing camera to follow the rows.

This master thesis is a continuation of a specialization project completed last semester (Arbo, 2014). In the specialization project the main focus was on a localization filter for absolute localization. This used RTK-GPS in combination with IMU and wheel encoders.

In this thesis the VO problem is addressed in a feature-based manner. The thesis considers feature matching and estimation of relative motion based on sets of features extracted from consecutive images. The features were extracted manually, and automatic extraction of features in the row-of-crops environment is a subject for future research.

1.4 Contributions

The contributions of this thesis are:

Warm-Start of Feature Search
A method of reducing the search space for matching features in two pictures given an estimate of the pose when the pictures were taken.

Bayesian Generalization of Stochastic Cloning
A Bayesian framework for handling fusion of OOSM in the context of VO. This framework can be viewed as a generalization of stochastic cloning.

Unscented Multiple-Point Smoother
An unscented fixed-point smoother capable of formulating fixed-points to smooth on demand.

Covariance Threshold
An alternative to masking for OOSM fusing filters that improves behavior during turns.

1.5 Report Structure

Theory
The theory chapter presents localization filters from the view of probability theory. This is followed by a brief introduction into how VO measurements are considered in this thesis. With a description of a method of “warm-starting” the search for matching feature points. The main part of this chapter is on OOSM fusion. Specifically how to optimally fuse delayed measurements in a Bayesian framework and how that compares to stochastic cloning.

Implementation
The implementation chapter presents systems used for testing the methods described in this thesis, both real and simulated. First a target tracking exercise is described that was used to test the python modules and the underlying behavior of the fusion methods. This is followed by a description of the Asterix robot in terms of kinematic model used, parameters of the prototype and how the sensors on Asterix correspond to the kinematic model. The last part of the implementation chapter discusses the software used for manual feature detection and the python modules developed in this thesis.

Results
The results chapter presents data from the simulations and experiments described in the implementation chapter. Each result section is followed by a brief discussion.

Epilogue
The epilogue chapter contains discussions on the overall project and methods, important points for the further development of Asterix and OOSM fusion, and concluding remarks.

1.6 Notation

Throughout the thesis, we deal mainly with discrete dynamical systems and probability distributions. Vectors will be denoted with bold-face letters e.g.: $\boldsymbol{x}_k$, where the subscript indicates which time instance, t_k, it corresponds to.

For probability distributions, $p(\boldsymbol{x}, \boldsymbol{y}|\boldsymbol{z})$ is the joint probability of $\boldsymbol{x}$ and $\boldsymbol{y}$ given $\boldsymbol{z}$. $\mathcal{N}(\boldsymbol{x}; \hat{\boldsymbol{x}}, P)$ is the normal distribution of $\boldsymbol{x}$ described by the expected value $\hat{\boldsymbol{x}}$ and the covariance P. $\mathrm{E}(\boldsymbol{x})$ also denotes the expected value of $\boldsymbol{x}$, $\mathrm{Var}(\boldsymbol{x})$ the variance, and $\mathrm{Cov}(\boldsymbol{x}, \boldsymbol{y})$ the covariance between $\boldsymbol{x}$ and $\boldsymbol{y}$.

As the means and covariances are described in terms of available knowledge, $\hat{x}_{i|j}$ means the expected value of x_i given knowledge of observations up to time j. Similarly for covariances. Capital letters with slanted, serifed font are used for sigma-points or images. $\mathcal{X}_{i,k|k+1}$ denotes ith sigma-point of the set of sigma-points $\mathcal{X}$, evaluated for the state $\boldsymbol{x}_k$ at time t_k, given information up to time t_{k+1}.

Matrices are denoted by capital letters, P, S, F. Dot-product is denoted by $\bullet$ and cross-product by $\times$. When a variable has a subscript with a colon sign, e.g. $x_{i:j}$, it is the sequence of variables $x_i, x_{i+1}, ..., x_{j-1}, x_j$.

2 Theory

2.1 Probabilistic Localization Filter

The Kalman filter has long been used in robot localization tasks. The basic idea of which is that the robot's configuration is probabilistic. The position of the robot on a coordinate frame is a probability distribution, the certainty of which increases during periods of accurate measurements. And decreases during periods of pure propagation, or when affected by unmodeled factors. This is the basic premise of probabilistic localization.

Robots come equipped with proprioceptive and extroreceptive sensors. Proprioceptive sensors are wheel encoders, IMU and other devices that record the motion with respect to the robot's previous configurations. They maintain relative localization. For example through integration of wheel encoders. As such they are prone to drift, modeling errors, slip, etc. Extroreceptive sensors measure the robots configuration with respect to some outside reference. For example GPS, sonar, laser-range finders, visually tracking landmarks, etc. They maintain absolute localization as the outside references define the coordinate frame itself. More of this is presented in section 2.5.

The probabilistic approach to robotics is based on the Bayes filter (Thrun, 2002, p.26).

2.1.1 Bayesian Filter

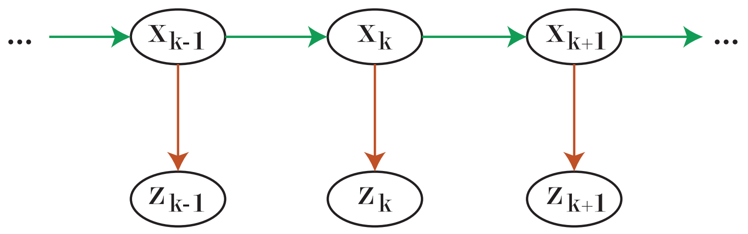

Figure 1: *Hidden Markov model of a simple robot system. The colors of the lines correspond to transition model and the sensor model.*

Suppose the robots configuration in its environment is described by the state vector $\boldsymbol{x}_k$. The robot is observed by sensors giving measurements $\boldsymbol{z}_k$. Localization means to find the distribution $p(\boldsymbol{x}_k|\boldsymbol{z}_0, .., \boldsymbol{z}_k)$ (Thrun, 2002). This is the probability that you are in a certain configuration at time t_k. For this to be possible, we assume that the system is governed by stationary processes and that the Markov assumption holds. The Markov assumption says that the current state only depends on a finite fixed number of previous states.

In this thesis the hidden Markov Model (HMM) is used as a method for representing a statistical Markov model in which the system being modeled is assumed to be a Markov process with a set of states not directly observed. It is used for graphical representation of independence and conditional independence for random variables. An example of such a graphical representation is in Fig.1.

By the first-order Markov assumption, the current configuration will only depend on the previous configuration:

$$p(\boldsymbol{x}_k|\boldsymbol{x}_0, ..., \boldsymbol{x}_{k-1}) = p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1}) \tag{2.1}$$

Note that one can make an nth-order Markov assumption into a first-order by extending the state vector to include the necessary prior states.

Similarly, in a standard HMM, the observation model only relates to the current configuration of the system:

$$p(\boldsymbol{z}_k|\boldsymbol{x}_0, ..., \boldsymbol{x}_k, \boldsymbol{z}_0, ..., \boldsymbol{z}_k) = p(\boldsymbol{z}_k|\boldsymbol{x}_k) \tag{2.2}$$

Using these assumptions, by basic probability theory, the joint pdf of a set of configurations and observations is given by:

$$p(\boldsymbol{x}_0, ..., \boldsymbol{x}_k, \boldsymbol{z}_1, ..., \boldsymbol{z}_k) = p(\boldsymbol{x}_0)\prod_{i=1}^{k} p(\boldsymbol{x}_i|\boldsymbol{x}_{i-1})p(\boldsymbol{z}_i|\boldsymbol{x}_i) \tag{2.3}$$

The optimal method of estimating the belief distribution is the recursive Bayes filter. The derivation is given in the appendix A.1. The Bayes filter consists of two major steps: prediction and update.

Predict

$$p(\boldsymbol{x}_k|\boldsymbol{z}_{0:k-1}) = \int_{\boldsymbol{x}_{k-1}} p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1})p(\boldsymbol{x}_{k-1}|\boldsymbol{z}_{0:k-1})\mathrm{d}\boldsymbol{x}_{k-1} \tag{2.4}$$

Update

$$p(\boldsymbol{x}_k|\boldsymbol{z}_{0:k}) = \mu p(\boldsymbol{z}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{z}_{0:k-1}) \tag{2.5}$$

In the predict step, the $p(\boldsymbol{x}_k|\boldsymbol{x}_{k-1})$ distribution is formed by forming a joint distribution of the current and the next configuration using the transition model and the previous configuration distribution. Then marginalizing the previous, now unnecessary, previous distribution. The update step utilizes the knowledge of our sensor model $p(\boldsymbol{z}_k|\boldsymbol{x}_k)$ and the *a priori* distribution to construct the *a posteriori* distribution $p(\boldsymbol{x}_k|\boldsymbol{z}_{0:k})$ using Bayes theorem. The terms *a priori* and *a posteriori* are with regard to the distribution before and after measurement fusion. If needed, the normalizing constant μ can be determined by the knowledge that the integral over the domain $\boldsymbol{x}_k$ must be equal to 1. The color of the equations signify the appropriate edges of the HMM in Fig.1.

These two equations form the basis of the Bayesian filter, which is computationally complicated and often practically impossible for continuous state vectors. The Bayesian filter has many approximations, such as the particle filter or the Kalman filter. The Kalman filter, and its nonlinear extensions, assumes a Gaussian distribution of the localization. Particle filters, Monte-Carlo filters, etc, are nonparametric filters that use sampling techniques to propagate the probability distributions.

2.1.2 Kalman Filters and Nonlinear Extensions

Kalman filters approximate the probability distributions of the Bayes filter with Gaussian distributions.

Consider the system:

$$\begin{aligned} \boldsymbol{x}_{k+1} &= f(\boldsymbol{x}_k) + \boldsymbol{w}_k \\ \boldsymbol{z}_k &= h(\boldsymbol{x}_k) + \boldsymbol{v}_k \\ p(\boldsymbol{w}_k) &= \mathcal{N}(\boldsymbol{w}_k; 0, Q_k) \\ p(\boldsymbol{v}_k) &= \mathcal{N}(\boldsymbol{v}_k; 0, R_k) \end{aligned} \tag{2.6}$$

where $\boldsymbol{x}_k \in \mathbb{R}^L$ are internal states, $\boldsymbol{z}_k \in \mathbb{R}^m$ are measurements, $\boldsymbol{w}_k$ is process noise, $\boldsymbol{v}_k$ is measurement noise. The mappings f and h are state transition and observation mappings respectively.

When the probability distributions are approximated by Gaussians, what we are actually looking at is maintaining good estimates of the mean and the covariance during the update and prediction steps of the filter.

Predict

$$\hat{\boldsymbol{x}}_{k+1|k} = E(\boldsymbol{x}_{k+1} | \boldsymbol{z}_1, ..., \boldsymbol{z}_k) \tag{2.7}$$

$$P_{k+1|k} = E\left((\boldsymbol{x}_{k+1} - \hat{\boldsymbol{x}}_{k+1|k})(\boldsymbol{x}_{k+1} - \hat{\boldsymbol{x}}_{k+1|k})^T \Big| \boldsymbol{z}_1, ..., \boldsymbol{z}_k \right) \tag{2.8}$$

Update

$$\hat{\boldsymbol{x}}_{k+1|k+1} = E(\boldsymbol{x}_{k+1} | \boldsymbol{z}_1, ..., \boldsymbol{z}_{k+1}) \tag{2.9}$$

$$P_{k+1|k+1} = E\left((\boldsymbol{x}_{k+1} - \hat{\boldsymbol{x}}_{k+1|k+1})(\boldsymbol{x}_{k+1} - \hat{\boldsymbol{x}}_{k+1|k+1})^T \Big| \boldsymbol{z}_1, ..., \boldsymbol{z}_{k+1} \right) \tag{2.10}$$

For linear f and h, the optimal estimates are given by a classical Kalman filter (Kalman, 1960). A brief derivation of the Kalman gain, showing that it is used for giving a minimum mean-square error evaluation of the next state of the system is given in the appendix A.1.

There are many types of nonlinear variations upon the Kalman filter. The main characteristics they share is that they assume the state, measurement, and error distributions can be described by Gaussian distributions (Tuan Pham et al., 1998) and that the evolution of these can be described in a recursive manner. In (2.7)-(2.10) the expectation and variance are formulated generally so that the two steps are descriptive of all Kalman-like filters.

The most common nonlinear version of the Kalman filter is the extended Kalman filter (EKF) (Brown and Hwang, 2012; Fossen, 2011). It uses analytic Jacobians to linearize the nonlinear functions. It is widely used in a variety of state and parameter estimation tasks, anything from traffic flow (van Lint, 2008) to oceanography (Tuan Pham et al., 1998), but it is not without flaws. The EKF can diverge if the system is not close to linear within the time scale of the update interval (Perea et al., 2007) and it requires the evaluation of Jacobians that can be computationally costly.

Unscented Kalman filters (UKF)(Merwe and Wan, 2004) use the unscented transformation to perform state estimations. The unscented transformation propagates a set of deterministically chosen points through the nonlinear functions to evaluate the resulting distribution characteristics. It has been shown to give second-order accuracy of the mean and covariance of a gaussian random variable undergoing a nonlinear transformation (Haykin, 2001, pp.269-273). The UKF was described by Merwe and Wan (2004) as a member of a set of filters termed sigma-point Kalman filters (SPKF). SPKFs do not require an analytic description of the Jacobian: they are “derivative-free” filters.

One of the benefits of using a Kalman-like filter is that marginalization is a simple process of choosing the subset of the elements in the expectation vector and covariance matrix corresponding to the states we want to keep. All other elements can be simply discarded.

2.1.3 Extended Kalman Filter

The EKF approximates (2.6) by linearization. Describing the system as:

$$\begin{aligned} \boldsymbol{x}_{k+1} &= F_k \boldsymbol{x}_k + \boldsymbol{w}_k \\ \boldsymbol{z}_k &= H_k \boldsymbol{x}_k + \boldsymbol{v}_k \end{aligned} \tag{2.11}$$

where F_k is the analytic jacobian of f and H_k is the analytic jacobian of h, at time t_k. It is easy to evaluate the predict step as the mean and variance for linear combinations of gaussian variables is well established.

The update step can be derived using least-squares arguments (see Appendix A.2) or by using the fundamental product identity for conditional Gaussian

Predict

$$\hat{\boldsymbol{x}}_{k+1|k} = f(\hat{\boldsymbol{x}}_{k|k}) \tag{2.12}$$

$$P_{k+1|k} = F_k P_{k|k} F_k^T + Q_k \tag{2.13}$$

distributions (Mahler 2007, Salmond 1989). The product identity can be summarized as the following:

$$\mathcal{N}(\boldsymbol{z}; H\boldsymbol{x}, R_k)\mathcal{N}(\boldsymbol{x}; \bar{\boldsymbol{x}}, \bar{P}) = \mathcal{N}(\boldsymbol{z}; \hat{\boldsymbol{z}}, S)\mathcal{N}(x; \hat{\boldsymbol{x}}, \hat{P}) \tag{2.14}$$

with

$$\begin{gathered} \hat{\boldsymbol{z}} = H\hat{\boldsymbol{x}}, \quad S = R + HPH^T \\ K = \bar{P}H^T S^{-1} \\ \hat{\boldsymbol{x}} = \bar{\boldsymbol{x}} + K(\boldsymbol{z} - \hat{\boldsymbol{z}}) \\ \hat{P} = (I - KH)\bar{P} \end{gathered} \tag{2.15}$$

where $\bar{\boldsymbol{x}}$ and $\bar{P}$ are our *a priori* mean and covariance estimates, and $\hat{\boldsymbol{x}}$ and $\hat{P}$ are our *a posteriori* estimates.

An interesting byproduct of this derivation is that the previous constant factor, μ, from the bayes filter is incorporated into the product identity. The pdf $\mathcal{N}(\boldsymbol{z}; \hat{\boldsymbol{z}}, S)$ describes the pdf of the measurement $\boldsymbol{z}$ given the current state estimate. This means that the pdf can be used for evaluating the likelihood of a measurement occuring, something which is important in solving the data association problem often encountered with target tracking using radar. The update step for an EKF is thusly:

Update

$$K_{k+1} = P_{k+1|k} H_{k+1}(H_{k+1} P_{k+1|k} H_{k+1}^T + R_{k+1})^{-1} \tag{2.16}$$

$$\hat{\boldsymbol{x}}_{k+1|k+1} = \hat{\boldsymbol{x}}_{k+1|k} + K_{k+1}(\boldsymbol{z}_{k+1} - H_{k+1}\hat{\boldsymbol{x}}_{k+1|k}) \tag{2.17}$$

$$P_{k+1|k+1} = (I - K_{k+1}H_{k+1})P_{k+1|k} \tag{2.18}$$

According to Thrun (2002, p.61), each iteration of the EKF has a computational complexity of $O(m^{2.4} + L^2)$ where L is the dimension of the state vector and m is the dimension of the measurement vector. This is the same as a linear Kalman filter, given that we have analytic descriptions of the Jacobians.

2.1.4 Unscented Kalman Filter

Consider:

$$\boldsymbol{y} = f(\boldsymbol{x}) \tag{2.19}$$

where $\boldsymbol{x} \in \mathbb{R}^L$ is a random variable with pdf $p(\boldsymbol{x})$ and f is an arbitrary non-linear function. The basic premise of the unscented transformation is to approximate a probability distribution rather than a nonlinear function (Julier and Uhlmann, 2002). As stated by Julier and Uhlmann, a set of $2L + 1$ sigma-points, $\mathcal{X}$, and weights, w, are deterministically chosen with a condition:

$$g(\mathcal{X}, w, p(\boldsymbol{x})) = 0$$

where $p(\boldsymbol{x})$ is the pdf of the state undergoing the transform. This condition function g is used to determine what information should be captured about $\boldsymbol{x}$ (Julier, 1998; Julier and Uhlmann, 2002). This does not necessarily result in unambiguous sigma-points, if necessary a penalty function c can be applied. The condition for choosing the sigma-points then becomes:

$$\min_{\mathcal{X}, w} c(\mathcal{X}, w, p(\boldsymbol{x})) \text{ subject to } g(\mathcal{X}, w, p(\boldsymbol{x})) = 0$$

For the UKF in this thesis one of the assumptions is that the distribution of the robot configuration is Gaussian. The necessary condition g has to ensure that the mean and variation are clearly defined. This means that g then becomes (Julier, 1998):

$$g(\mathcal{X}, w, p(\boldsymbol{x})) = \begin{bmatrix} \sum_{i=0}^{2L+1} w_i \\ \sum_{i=0}^{2L+1} w_i \mathcal{X}_i \\ \sum_{i=0}^{2L+1} w_i (\mathcal{X}_i - \mathrm{E}(\boldsymbol{x}))(\mathcal{X}_i - \mathrm{E}(\boldsymbol{x})) \end{bmatrix} - \begin{bmatrix} 1 \\ \mathrm{E}(\boldsymbol{x}) \\ \mathrm{Var}(\boldsymbol{x}) \end{bmatrix} \tag{2.20}$$

By generating the sigma-points and weights according to the rules of (2.21), the condition (2.20) will hold (Merwe, 2004, p.52). This has been shown explicitly in Appendix A.3.

$$\begin{aligned} \mathcal{X}_0 &= \mathrm{E}(\boldsymbol{x}) \\ \mathcal{X}_j &= \mathrm{E}(\boldsymbol{x}) + \sqrt{(L+\lambda)\mathrm{Var}(\boldsymbol{x})}_i \\ \mathcal{X}_k &= \mathrm{E}(\boldsymbol{x}) - \sqrt{(L+\lambda)\mathrm{Var}(\boldsymbol{x})}_i \\ w_0 &= \lambda/(L+\lambda) \\ w_i &= 1/2(L+\lambda) \end{aligned} \tag{2.21}$$

for $j = 1, ..., L$, $k = L + 1, ..., 2L + 1$ and $i = 1, ..., L$. The subscript indices on the covariance matrices corresponds with columns of the matrix. With the transformed sigma points being:

$$\mathcal{Y}_i = f(\mathcal{X}_i)$$

the mean and covariance of $\boldsymbol{y}$ can be found by:

$$\begin{aligned} \mathrm{E}(\boldsymbol{y}) &= \sum_{i=0}^{2L+1} w_i \mathcal{Y}_i \\ \mathrm{Var}(\boldsymbol{y}) &= \sum_{i=0}^{2L+1} w_i (\mathcal{Y}_i - \mathrm{E}(\boldsymbol{y}))(\mathcal{Y}_i - \mathrm{E}(\boldsymbol{y}))^T \end{aligned} \tag{2.22}$$

The cross-covariance between $\boldsymbol{y}$ and $\boldsymbol{x}$ is (Merwe, 2004, p.53):

$$\mathrm{cov}(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=0}^{2n+1} w_i (\mathcal{X}_i - \mathrm{E}(\boldsymbol{x}))(\mathcal{Y}_i - \mathrm{E}(\boldsymbol{y}))^T \tag{2.23}$$

This is intuitive considering that if $f(\boldsymbol{x}) = A\boldsymbol{x}$ then the cross-covariance is:

$$\mathrm{cov}(\boldsymbol{x}, A\boldsymbol{x}) = \mathrm{Var}(\boldsymbol{x}) A^T$$

As shown in the derivation of the Kalman gain in the Appendix A.2, other than the initial mean and covariance, the prediction-observation covariance N_k, and observation residual covariance (innovation covariance) S_k are also needed to ensure minimum-mean-square error (see Appendix for definitions). With a sensor model h, these can be easily computed according to the guidelines above.

The UKF algorithm with tuning parameters is given in Alg. 1. The parameters are added to minimize the chance of the sigma-points sampling non-local effects (Merwe, 2004, p.54). Parameter κ is chosen to guarantee positive semi-definiteness of the covariance matrix, usually set to 0. The spread of the sigma-points is determined by α, and should be small to avoid sampling non-local effects. Knowledge of higher-order moments of the initial distribution affects the choice of β, which in our gaussian case is set to 2.

An example of the unscented transformation performing better than the linearization can be seen in Fig.2. In that figure, a Monte-Carlo simulation of the nonlinear transformation of a gaussian variable is compared to the mean and covariance produced by an unscented transformation and that of a linearization. The initial distribution was:

$$p\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \mathcal{N}\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}; \begin{bmatrix} 5 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 & -2 \\ -2 & 3 \end{bmatrix}\right)$$

and the nonlinear transformation was:

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0.1 x_1 \sqrt{x_1^2 + x_2^2} + x_2 \\ -x_1 \end{bmatrix}$$

The key difference in the EKF and the UKF is the underlying assumption of what to approximate. EKF assumes the nonlinear function is linear in a time interval, the UKF assumes the resulting distribution is gaussian. The two assumptions may appear equivalent as the linear combination of a set of Gaussian distributions is Gaussian. But by only passing a single point through the nonlinear function, less of the information of the curvature is passed and the EKF might fail to take this information into account when producing its state estimate.

According to Thrun (2002, p.69), the computational complexity of the UKF algorithm is the same as the EKF, although in practice the EKF is slightly faster. However, the UKF has shown consistently improved performance both in convergence rate and long term state estimation error (Kandepu et al., 2008).

Algorithm 1: Unscented Kalman Filter

```
    **Set tunable parameters**:
    $0 \leq \alpha \leq 1$, $\beta \geq 0$ and $\kappa \geq 0$
    **Initialize filter and weights**:
        $L = length(\boldsymbol{x})$
        $\lambda = \alpha^2(L + \kappa) - L$
        $w_0^m = \frac{\lambda}{L + \lambda}$
        $w_0^c = w_0^m + (1 - \alpha^2 + \beta)$
        for $i \in 1 \ldots 2L : w_i^c = w_i^m = \frac{1}{2(L + \lambda)}$
        $\hat{\boldsymbol{x}}_{0|0} = E(\boldsymbol{x}_{0|0})$
        $P_{0|0} \succ 0$
    **Generate sigma-points**:
        $\boldsymbol{\mathcal{X}}_{0,k-1|k-1} = \hat{\boldsymbol{x}}_{k-1|k-1}$
        $\boldsymbol{\mathcal{X}}_{i,k-1|k-1} = \hat{\boldsymbol{x}}_{k-1|k-1} + \left(\sqrt{(L + \lambda)P_{k-1|k-1}}\right)_i$, for $i = 1, ..., L$
        $\boldsymbol{\mathcal{X}}_{i,k-1|k-1} = \hat{\boldsymbol{x}}_{k-1|k-1} - \left(\sqrt{(L + \lambda)P_{k-1|k-1}}\right)_i$, for $i = L + 1, ..., 2L$
    **Predict**
        $\boldsymbol{\mathcal{X}}_{k|k-1} = f\left(\boldsymbol{\mathcal{X}}_{k-1|k-1}\right)$
        $\hat{\boldsymbol{x}}_{k|k-1} = \sum_{i=0}^{2L} w_i^m \boldsymbol{\mathcal{X}}_{i,k|k-1}$
        $P_{k|k-1} = \sum_{i=0}^{2L} w_i^c \left(\boldsymbol{\mathcal{X}}_{i,k|k-1} - \hat{\boldsymbol{x}}_{k|k-1}\right)\left(\boldsymbol{\mathcal{X}}_{i,k|k-1} - \hat{\boldsymbol{x}}_{k|k-1}\right)^T + Q_{k-1}$
        $\boldsymbol{\mathcal{Z}}_{k|k-1} = h\left(\boldsymbol{\mathcal{X}}_{k|k-1}\right)$
        $\hat{z}_{k|k-1} = \sum_{i=0}^{2L} w_i^m \boldsymbol{\mathcal{Z}}_{i,k|k-1}$
    **Update**
        $S_k = \sum_{i=0}^{2L} w_i^c \left(\boldsymbol{\mathcal{Z}}_{i,k|k-1} - \hat{\boldsymbol{z}}_{k|k-1}\right)\left(\boldsymbol{\mathcal{Z}}_{i,k|k-1} - \hat{\boldsymbol{z}}_{k|k-1}\right)^T + R_k$
        $N_k = \sum_{i=0}^{2L} w_i^c \left(\boldsymbol{\mathcal{X}}_{i,k|k-1} - \hat{\boldsymbol{x}}_{k|k-1}\right)\left(\boldsymbol{\mathcal{Z}}_{i,k|k-1} - \hat{\boldsymbol{z}}_{k|k-1}\right)^T$
        $K_k = N_k S_k^{-1}$
        $\hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + K\left(\boldsymbol{z}_k - \hat{\boldsymbol{z}}_{k|k-1}\right)$
        $P_{k|k} = P_{k|k-1} - K S_k K^T$
```

2.1.5 Masking

There are cases where it might not be advantageous to update all the states in a filter. The states may be very "weakly" observable, thus rendering their state estimates unreliable, or they may have a known probability distribution which is constant with time. Deliberately preventing the filter from updating such states is known as masking (Merwe, 2004, p.204). A masked filter takes into account that these states have probability distributions without changing its estimate of that distribution (Brown and Hwang, 2012, p.188).

The use of masking is relevant for OOSM fusion and there are two different mentalities. In Roumeliotis and Burdick (2002), a cloned state is maintained with the same initial estimate as when it was cloned in hopes of reducing its exposure to modeling errors. In (Merwe, 2004, p.204) it is discussed that not using a mask, and allowing the cloned state to be "smoothed" is beneficial for OOSM fusion.

When handling EKF or KF, knowledge of the masking can reduce the computational savings, and the procedure for this is the Schmidt-Kalman filter (Brown and Hwang, 2012, p.188). The Schmidt-Kalman filter formulates a recursive description of the evolution of the cross-covariances between the masked and the unmasked states. It uses this to reduce the computational load of the filter and keep the elements of the state covariance matrix associated with the masked states constant. This requires dedicated Schmidt versions of the filters and was not used in this thesis. Instead masking has been done by retaining the previous state and state covariance estimates during the update procedure, and then resetting the masked subsets to their values prior to being updated.

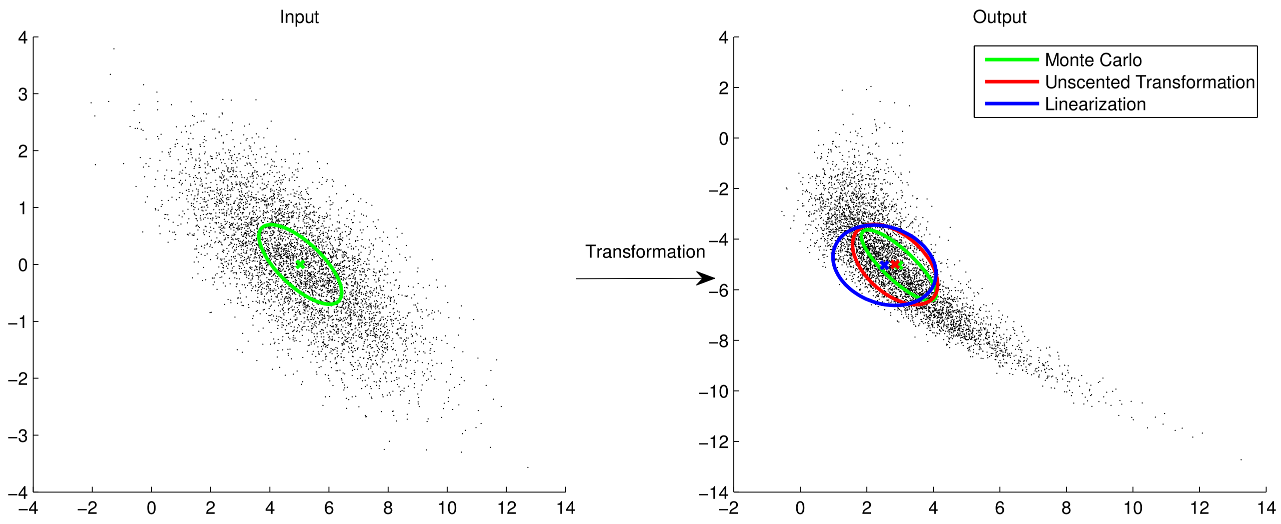

Figure 2: *The unscented transformation gives a better estimate of the mean and covariance of gaussian random variables undergoing a nonlinear transformation. Note that the UT mean is closer to the mean of the Monte Carlo simulation than the linearized mean. This is because the mean is the average of the transformed weighted sigma points, not just the transformation of the prior mean.*

2.2 Visual Odometry

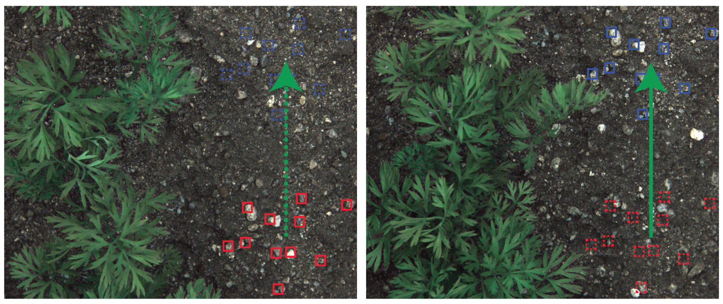

__Figure 3:__ The relative motion of feature points is in the opposite direction of the movement of the robot. The robot is heading down the page. The left image is the first, and the right is the second.

Visual odometry (VO) is a method of determining the displacement of a camera with respect to time. It is an attempt at estimating the displacement of the camera over time expressed in either 3 degrees of freedom, x, y and heading ψ, or in 6 degrees of freedom, x, y, z, roll φ, pitch θ and yaw ψ.

There are two major classes of methods to recover camera displacement from sequential sets of images (Forster et al., 2014):

Direct Methods
Given two intensity maps of sequential images with timing. Calculating the intensity gradient with respect to magnitude and direction gives an estimate of the motion of the camera (Forster et al., 2014).

Feature-Based Methods
Given two sequential images with timing, the motion of the camera can be inferred by identifying feature points common for the two images and estimating the motion necessary for that transformation. This has been used extensively on NASA's Mars exploration rovers (Maimone and Matthies, 2005).

The direct methods hope to use as much as possible of the available information in a picture to estimate the camera motion (Newcombe et al., 2011). To that effect most algorithms are computationally costly and require a high level of parallelization. The feature-based methods attempt to identify key features that match across sequential images according to a feature identification algorithm. The relative displacement of these features in the image frame are used to describe the motion of the camera. They rely on the assumption that features can be consistently detected and tracked over different pictures.

In Mammarella et al. (2012) both direct methods and feature-based methods were compared for a optical flow task and it was found that feature-based methods were better at handling large and small displacements that may be present in complex 3D motion. Mammarella et al. uses the displacement to estimate the velocities that the camera is experiencing and this is often described as optical flow. Asterix takes sequential pictures of high quality at a low frame rate, the low framerate means that the average velocities calculated from the displacement measurements is not an accurate representation of the velocities the robot has experienced. Instead, the VO displacement is considered to be a measure between two configuration estimates tracked by the filter.

In this project the long time between sequential images and less than 75% overlap between images suggest that the direct methods are unlikely to work as desired. Instead the feature-based methods will be used as they handle large displacements better, and are more common. The visual odometry is based on images which are also used for the crop identification algorithm(Grændsen, 2014).

In SLAM, features can be maintained over a long time horizon, enabling very good estimates of absolute position through loop-closing when features are revisited (Thrun, 2002). For our downward facing camera, we will only be able to identify the displacement that the robot has moved between the two pictures. The images only overlap for 2-3 frames at a time.

2.2.1 Motion From Features

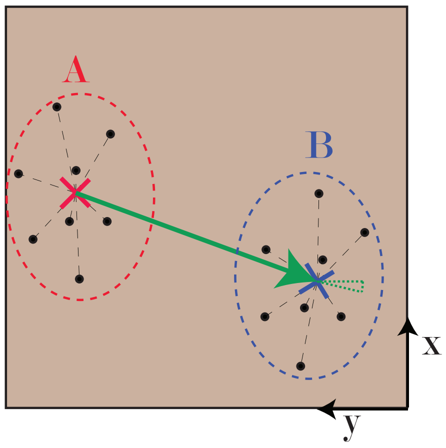

Figure 4: The motion of a set of features (Black dots) undergoing a translation and a rotation. The green arrow is translation. The stippled green arc indicates angular change. The red and blue crosses are the mean positions of the two sets of features.

A variety of methods exist to infer motion from two sets of features. In this case, the features are assumed to be in the plane, and as such, the most straightforward method is the one suggested by Horn (1987). In Fig.4, two pictures have been laid on top of each other so that they share coordinate frames, and a set of features have been found that are present for both. The set A corresponds to the oldest picture, and the set B corresponds to the newest picture. The position of each feature is described by a set of coordinates:

$$A = \left\{ \begin{bmatrix} x_i \\ y_i \end{bmatrix} \right\}$$
$$B = \left\{ \begin{bmatrix} x_i \\ y_i \end{bmatrix} \right\}$$

The number of features L is shared for both feature sets. We assume that the two sets are ordered so that the same feature has the same index. We then have:

$$\begin{bmatrix} \bar{x}_A \\ \bar{y}_A \end{bmatrix} = \frac{1}{L} \sum_{i=1}^{L} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$
$$\begin{bmatrix} \bar{x}_B \\ \bar{y}_B \end{bmatrix} \frac{1}{L} \sum_{i=1}^{L} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Robot motion is inferred from looking at the motion from the oldest set of features, A, to the newest set of features, B. The linear motion of the features is described by:

$$\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} \bar{x}_B \\ \bar{y}_B \end{bmatrix} - \begin{bmatrix} \bar{x}_A \\ \bar{y}_A \end{bmatrix}$$

Note that these displacements are considered from the perspective of the oldest picture. The rotation between the features can be calculated by defining:

$$C = \sum_{i=1}^{L} \left(\begin{bmatrix} x_{B,i} \\ y_{B,i} \\ 0 \end{bmatrix} - \begin{bmatrix} \bar{x}_B \\ \bar{y}_B \\ 0 \end{bmatrix} \right) \bullet \left(\begin{bmatrix} x_{A,i} \\ y_{A,i} \\ 0 \end{bmatrix} - \begin{bmatrix} \bar{x}_A \\ \bar{y}_A \\ 0 \end{bmatrix} \right)$$
$$S = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \sum_{i=1}^{L} \left(\begin{bmatrix} x_{B,i} \\ y_{B,i} \\ 0 \end{bmatrix} - \begin{bmatrix} \bar{x}_B \\ \bar{y}_B \\ 0 \end{bmatrix} \right) \times \left(\begin{bmatrix} x_{A,i} \\ y_{A,i} \\ 0 \end{bmatrix} - \begin{bmatrix} \bar{x}_A \\ \bar{y}_A \\ 0 \end{bmatrix} \right)$$

Giving the rotation $\Delta\psi$ in yaw as:

$$\Delta\psi = -\text{atan2}(S, C)$$

Expressed as a rotation around the up direction from the oldest picture to the newest.

2.2.2 Warm-Start of Feature Search

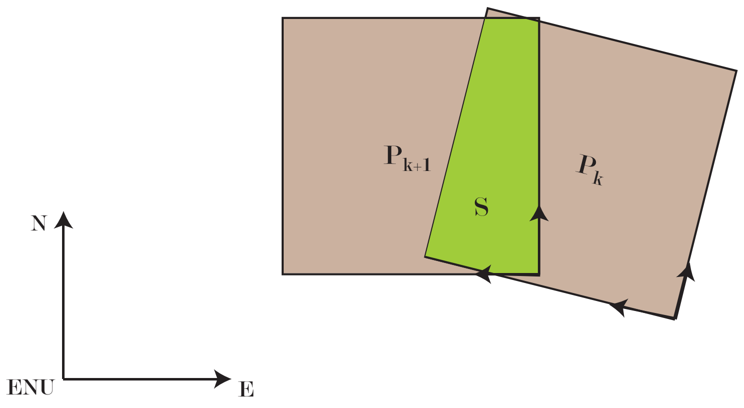

Figure 5: Picture $\mathcal{P}_k$ and picture $\mathcal{P}_{k+1}$ overlap forming the region $\mathcal{S}$. The pictures have their own coordinate systems, and they are located in the global ENU frame.

One of the main challenges facing visual odometry is the necessity to search the whole picture for feature points. Any localization filter that attempts to fuse a delayed displacement measurement must be able to describe the transformation between how the robot is localized now and how it was when the previous picture took place. This means that we are maintaining an estimate of the robot's state from when it took picture $\mathcal{P}_k$ up to the point it takes picture $\mathcal{P}_{k+1}$. This can be used to reduce the search space for matching features drastically.

Suppose at the time when picture $\mathcal{P}_k$ is taken, the robot is located at x_k, y_k with heading ψ_k in the ENU frame. And at the time when picture $\mathcal{P}_{k+1}$ is taken, the robot is located at x_m, y_m with heading ψ_m. The indices differ for $\mathcal{P}_{k+1}$ and the pose at t_m as the pictures arrives in a different sequence than the states. The pictures may arrive every 2 s while the states are estimated every 0.05 s.

We have a point x_{f1}, y_{f1} in the coordinate frame of picture $\mathcal{P}_k$. This point can be expressed as x_{f2}, y_{f2} in the coordinate frame of picture $\mathcal{P}_{k+1}$ by the transformation:

$$\begin{bmatrix} x_{f2} \\ y_{f2} \end{bmatrix} = R(\psi_m - \psi_k) \begin{bmatrix} x_{f1} \\ x_{f1} \end{bmatrix} + R(\psi_m) \begin{bmatrix} x_k - x_m \\ y_k - y_m \end{bmatrix} \tag{2.24}$$

with $R(\theta)$ being the 2D rotation matrix in $SO(2)$:

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{2.25}$$

This means that if we find a feature in picture $\mathcal{P}_{k+1}$, we can define a region we expect it to be in by the transformation (2.24). The same transformation can also be used to reduce the initial search space of picture $\mathcal{P}_k$. As the height of the camera above ground is fixed, the dimensions of the visible area is known and constant. It can be expressed as 4 coordinates in the $\mathcal{P}_{k+1}$ frame. These coordinates can be expressed in $\mathcal{P}_k$ by the inverse of the transformation (2.24). As shown in the extended project they can be used to create a searchable region defined by a set of inequalities (Arbo, 2014). This is visualized in Fig.6 as the green area $\mathcal{S}$.

Also note that if the displacement with visual odometry, $\Delta x_{VO}, \Delta y_{VO}$, is represented in the coordinate frame of picture $\mathcal{P}_k$, it will be represented in the ENU frame as:

$$\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = R(\psi_k) \begin{bmatrix} \Delta x_{VO} \\ \Delta y_{VO} \end{bmatrix} \tag{2.26}$$

As the filter will be smoothing the old state estimate, the oldest picture coordinate frame has been chosen. Angles can only take values between $-\pi$ and π, this means that one must calculate addition and subtraction differently. Wrapping the values on the interval and calculating differences in the circle rather the line. In practice, if the angle differences are sufficiently small, this can be disregarded.

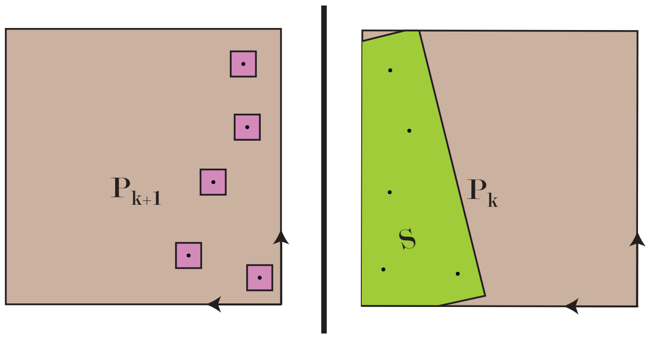

Figure 6: *Feature points found in the search space $\mathcal{S}$ for picture $\mathcal{P}_k$ can be searched for in small rectangles (the purple rectangles) around the corresponding points in the new picture region $\mathcal{P}_{k+1}$.*

In Fig.6 we see search regions for a set of features in $\mathcal{P}_{k+1}$, corresponding to a set of features found in $\mathcal{P}_k$. Using the estimated distributions of the states to define the search regions is known as validation gating (Bar-Shalom and Li, 1996; Brekke and Chitre, 2014) and allows us to define more complex search regions. From a practical perspective the search regions are easiest to implement as squares of set height and width, the predefined areas ensure a deterministic search time per feature.

2.2.3 Expected Timing

In Utstumo (2011), the timing of a feature detection algorithm that uses binocular vision was presented. The system was a BlackBox IMU VO system described in Goldberg and Matthies (2011). This system had an expected delay of 58 ms from when the two pictures are taken until the motion estimate became available. The delay is visualized in Fig.7, describing roughly which parts took what amount of time.

This was for a dedicated VO processing setup. For the Asterix system, the delay is expected to be longer. In this thesis, the delay is considered to be 500 ms from when the second image arrives and when the VO measurement becomes available. It is a rough estimate chosen to be significantly larger than the expectations to ensure that the system will still function. But this delay is not necessarily the same each time. If the robot moves over a segment where the plants have grown tall enough to obfuscate the view, the system will struggle to find an acceptable VO measurement. The previous section described warm-start allowing for a deterministic search time per feature, but the number of features is not deterministic when the plants have to be taken into consideration as the features are not necessarily consistently recognizable. A feature can be clear of any plants in one image and hidden in the next. This suggests that for the initial image, the feature identification algorithm should find more features than necessary such that the feature matching algorithm will have full opportunity to find a sufficient number of matching features.

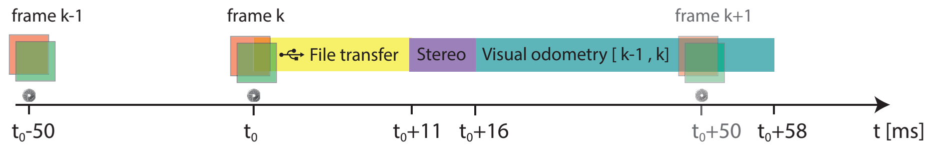

Figure 7: *Timing diagram for a binocular IMU VO system used for motion estimation (Trygve Utstumo).*

2.3 Out-of-Sequence Measurements

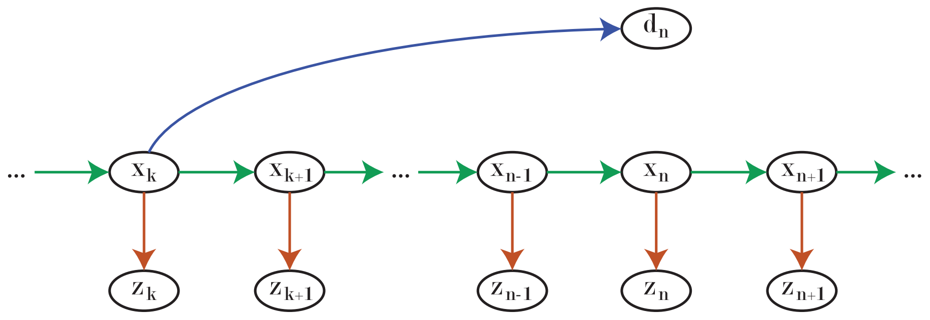

Figure 8: *Visualization of the hidden markov model where a measurement $\boldsymbol{d}_n$ is of the state of the system at time t_k, but arrives at time t_n.*

Conventional hidden Markov models of filters, visualized as in Fig.1, describe a world where observations occur instantaneously and neither observation nor filter iteration has any processing time. The real world is not so accommodating. Observations often require processing time, or are samples of continuous processes not necessarily synchronized with the filter. Out-of-sequence measurements (OOSMs) describe the scenario where measurements do not arrive in order of the states they correspond to.

If measurements can only arrive after the state they correspond to, by applying a causality assumption, there are three major parts to OOSMs. One concerns the fusion of measurements that have occured between two time instances of the filter (see Fig.9), the second concerns measurements that are delayed an arbitrary number of iterations (see Fig.8). The third discusses cases where a delayed measurement arrives, without us knowing what time instance it relates to.

For the first topic, a Bayesian formulation was proposed by Challa et al. (2003), which involves augmenting the state vector to include the previous state, and then using knowledge of the timestamp and the transition model to fuse the old measurement. A Gaussian formulation was proposed by Bar-Shalom (2002), where the inverse of the transition matrix and the measurement timestamp was used to “retrodict” the state estimate such that it accounts for the delay.

For Asterix the feature detection and processing algorithms require a lot of time, the measurements may become available after many filter cycles have already transpired. But the measurements are taken onboard the Asterix, and the timestamp of when a picture is taken is available to the filter as it is taken. Hence the second topic is of interest for this project and will be addressed in this chapter.

The two first topics are not mutually exclusive, and in fact, the reality is a combination. But if the timestep of the filter is small enough, then the timestamp of the OOSM can be approximated as coinciding with the next timestamp of the filter with negligible error.

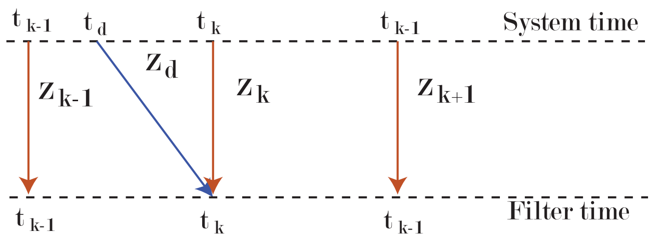

Figure 9: *The measurement z_d is fused at time t_k but is a measurement of the system at time t_d.*

2.3.1 Previous Strategies

Since the invention of the Kalman filter, a variety of strategies have been discussed to fuse OOSMs.

Slow down
: Wait for the slowest measurement. Or ignore the fact that it is delayed (Gavrilets, 2003).

Replay
: Advance the filter as normal, but store all the measurements that have occured between when the delayed measurement should have arrived and now. When the delayed measurement becomes available, replay the filter from when the measurement should have arrived.

Delay Augmentation
: If the delay is of a known length, the state vector can be augmented with a set of delayed states. The transition function then also becomes modified with a delay and the delayed measurement will be of a state in the localization filter (Challa et al., 2002). This can be considered a fixed-lag smoother approach.

Larsen's method
: Calculate the Kalman gain as though the measurement was available, then propagate the gain with the transition matrix until it is available. This is used to formulate a correction term that is added when the measurement becomes available (Larsen and Poulsen, 1998).

Stochastic Cloning
: Maintain a lagged state at the point when the lagged measurement should have been fused. During the fast measurements, evaluate the covariance and cross-covariance of the current and the lagged state. When the measurement becomes available, evaluate the current state using the covariance to send the estimate forward in time (Merwe, 2004; Roumeliotis and Burdick, 2002). This can be considered a fixed-point smoother approach.

The strategy of slowing down is not practical for most dynamical systems. It ignores the problem of the Markov assumption not holding for the delayed measurement. A high-speed drone for example, might need state estimates faster than the onboard GPS is capable of refreshing, and ignoring the problem may give erroneous state estimates Merwe and Wan (2004). Replaying the measurements from the time the delayed measurement was to have arrived is computationally costly, and is not practical for an arbitrary delay length. Similarly, delay augmentation only works when the delay length is known beforehand, and for long delay periods, can be resource intensive. Larsen's method acknowledges that there will be an accumulated error of the filter as a result of the lagged measurement and estimates this when it has arrived. This accumulation based on the transition matrix seems likely to accumulate modeling errors when handling a nonlinear system.

2.3.2 Optimal Approach

This derivation is similar to that of Challa et al. (2002), but it considers a fixed-point smoother instead of a fixed-lag smoother as the underlying method. We wish to find an expression for $p(\boldsymbol{x}_n|\boldsymbol{z}_{k:n}, \boldsymbol{d}_n)$ (see Fig.8). Available to the system are measurements $\boldsymbol{z}_k$ that arrive at each iteration, and measurements $\boldsymbol{d}_n$ that arrive intermittently and delayed from the state they describe, $\boldsymbol{x}_k$.

We have the following knowledge of the system:

- $p(\boldsymbol{x}_k|\boldsymbol{z}_k)$ (Known initial configuration estimate).
- $p(\boldsymbol{x}_{i+1}|\boldsymbol{x}_i)$ for any $i > k$ (transition model, first-order Markov assumption).
- $p(\boldsymbol{z}_i|\boldsymbol{x}_i)$ for any i (sensor model).
- $p(\boldsymbol{d}_j|\boldsymbol{x}_i)$ for any $j > i$ (sensor model for the delayed measurement).

Given a joint distribution $p(\boldsymbol{x}_i, \boldsymbol{x}_j|\boldsymbol{z}_{i:j})$ with indices $j > i$, and a sequence of measurements $\boldsymbol{z}_{j+1:n}$, we can find the distribution $p(\boldsymbol{x}_i, \boldsymbol{x}_n|\boldsymbol{z}_{i:n})$. This is done by repeating the following steps as necessary:

1. Extend the joint distribution:
$$p(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{x}_{j+1}|\boldsymbol{z}_{i:j}) = p(\boldsymbol{x}_{j+1}|\boldsymbol{x}_j)p(\boldsymbol{x}_i, \boldsymbol{x}_j|\boldsymbol{z}_{i:j}) \tag{2.27}$$
2. Fuse the latest measurement with Bayes theorem:
$$p(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{x}_{j+1}|\boldsymbol{z}_{i:j+1}) = \frac{p(\boldsymbol{z}_{j+1}|\boldsymbol{x}_{j+1})p(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{x}_{j+1}|\boldsymbol{z}_{i:j})}{p(\boldsymbol{z}_{j+1}|\boldsymbol{z}_{i:j})} \tag{2.28}$$
3. Marginalize the unnecessary state:
$$p(\boldsymbol{x}_i, \boldsymbol{x}_{j+1}|\boldsymbol{z}_{i:j+1}) = \int_{\boldsymbol{x}_j} p(\boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{x}_{j+1}|\boldsymbol{z}_{i:j+1}) d\boldsymbol{x}_j \tag{2.29}$$

As the desired distribution from these steps is $p(\boldsymbol{x}_i, \boldsymbol{x}_n|\boldsymbol{z}_{i:n})$, the intermittent states between $\boldsymbol{x}_i$ and $\boldsymbol{x}_n$ are deemed unnecessary and marginalized.

Thus, if we have an initial distribution $p(\boldsymbol{x}_k|\boldsymbol{z}_k)$, we can formulate the joint prior distribution by:

$$p(\boldsymbol{x}_k, \boldsymbol{x}_{k+1}|\boldsymbol{z}_k) = p(\boldsymbol{x}_{k+1}|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{z}_k) \tag{2.30}$$

and fuse the latest measurement by Bayes theorem:

$$p(\boldsymbol{x}_k, \boldsymbol{x}_{k+1}|\boldsymbol{z}_{k:k+1}) = \frac{p(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1})p(\boldsymbol{x}_k, \boldsymbol{x}_{k+1}|\boldsymbol{z}_k)}{p(\boldsymbol{z}_{k+1}|\boldsymbol{z}_k)} \tag{2.31}$$

Then we repeat the steps (2.27) - (2.29) until we arrive at the distribution $p(\boldsymbol{x}_k, \boldsymbol{x}_n|\boldsymbol{z}_{k:n})$. This distribution contains all the states necessary to fuse the delayed measurement $\boldsymbol{d}_n$. We can arrive at the desired posterior distribution by fusing:

$$p(\boldsymbol{x}_k, \boldsymbol{x}_n|\boldsymbol{z}_{k:n}, \boldsymbol{d}_n) = \frac{p(\boldsymbol{d}_n|\boldsymbol{x}_k)p(\boldsymbol{x}_k, \boldsymbol{x}_n|\boldsymbol{z}_{k:n})}{p(\boldsymbol{d}_n|\boldsymbol{z}_{k:n})} \tag{2.32}$$

And finally marginalizing the unnecessary state $\boldsymbol{x}_k$:

$$p(\boldsymbol{x}_n|\boldsymbol{z}_{k:n}, \boldsymbol{d}_n) = \int_{\boldsymbol{x}_k} p(\boldsymbol{x}_k, \boldsymbol{x}_n|\boldsymbol{z}_{k:n}, \boldsymbol{d}_n)d\boldsymbol{x}_k \tag{2.33}$$

This leaves us with the desired posterior distribution, purely in accordance with the Bayes filter.

2.3.3 Unscented Fixed-Point Smoother

In practical terms, this procedure can be done by applying an unscented fixed-point smoother. A key point is that this fixed-point smoother is capable of formulating the joint distribution on demand. Fixed-point smoothing considers the system where:

$$\begin{aligned}
\boldsymbol{x}_{k+1} &= f(\boldsymbol{x}_k) + \boldsymbol{w}_k \\
\boldsymbol{z}_k &= h(\boldsymbol{x}_k) + \boldsymbol{v}_k \\
p(\boldsymbol{w}_k) &= \mathcal{N}(\boldsymbol{w}_k; 0, Q_k) \\
p(\boldsymbol{v}_k) &= \mathcal{N}(\boldsymbol{v}_k; 0, R_k)
\end{aligned}$$

with $\boldsymbol{x}_k \in \mathbb{R}^n$, $\boldsymbol{z}_k \in \mathbb{R}^m$. We desire to maintain the joint distribution $p(\boldsymbol{x}_i, \boldsymbol{x}_j|\boldsymbol{z}_{i:j})$, where $j > i$ and the smoothed state is $\boldsymbol{x}_i$. The steps (2.27) - (2.29) of the previous section describes how to optimally fuse a measurement for a joint distribution in Bayesian terms and can be used directly in fixed-point smoothing. The first step is constructing the joint distribution $p(\boldsymbol{x}_k, \boldsymbol{x}_{k+1}|\boldsymbol{z}_k)$. If this is a gaussian distribution, we have:

$$p(\boldsymbol{x}_k, \boldsymbol{x}_{k+1}|\boldsymbol{z}_k) = \mathcal{N}\left(\begin{bmatrix}\boldsymbol{x}_k \\ \boldsymbol{x}_{k+1}\end{bmatrix}; \begin{bmatrix}\mathrm{E}(\boldsymbol{x}_k|\boldsymbol{z}_k) \\ \mathrm{E}(\boldsymbol{x}_{k+1}|\boldsymbol{z}_k)\end{bmatrix}, \begin{bmatrix}\mathrm{Var}(\boldsymbol{x}_k|\boldsymbol{z}_k) & \mathrm{Cov}(\boldsymbol{x}_k, \boldsymbol{x}_{k+1}|\boldsymbol{z}_k) \\ \mathrm{Cov}(\boldsymbol{x}_{k+1}, \boldsymbol{x}_k|\boldsymbol{z}_k) & \mathrm{Var}(\boldsymbol{x}_{k+1}|\boldsymbol{z}_k)\end{bmatrix}\right)$$

From (2.22) and (2.23) we know:

$$\begin{aligned}
\mathrm{E}(\boldsymbol{x}_k|\boldsymbol{z}_k) &= \sum_{i=0}^{2n+1} w_i \mathcal{X}_{i,k|k} \\
\mathrm{E}(\boldsymbol{x}_{k+1}|\boldsymbol{z}_k) &= \sum_{i=0}^{2n+1} w_i f(\mathcal{X}_{i,k|k}) \\
\mathrm{Var}(\boldsymbol{x}_k|\boldsymbol{z}_k) &= \sum_{i=0}^{2n+1} w_i(\mathcal{X}_{i,k|k} - \mathrm{E}(\boldsymbol{x}_k|\boldsymbol{z}_k))(\mathcal{X}_{i,k|k} - \mathrm{E}(\boldsymbol{x}_k|\boldsymbol{z}_k))^T \\
\mathrm{Var}(\boldsymbol{x}_{k+1}|\boldsymbol{z}_k) &= \sum_{i=0}^{2n+1} w_i(\mathcal{X}_{i,k+1|k} - \mathrm{E}(\boldsymbol{x}_{k+1}|\boldsymbol{z}_k))(\mathcal{X}_{i,k+1|k} - \mathrm{E}(\boldsymbol{x}_{k+1}|\boldsymbol{z}_k))^T + Q_k \\
\mathrm{Cov}(\boldsymbol{x}_k, \boldsymbol{x}_{k+1}) &= \sum_{i=0}^{2n+1} w_i(\mathcal{X}_{i,k|k} - \mathrm{E}(\boldsymbol{x}_k|\boldsymbol{z}_k))(\mathcal{X}_{i,k+1|k} - \mathrm{E}(\boldsymbol{x}_{k+1}|\boldsymbol{z}_k))^T
\end{aligned} \tag{2.34}$$

and $\mathrm{Cov}(\boldsymbol{x}_{k+1}, \boldsymbol{x}_k|\boldsymbol{z}_k) = \mathrm{Cov}(\boldsymbol{x}_k, \boldsymbol{x}_{k+1}|\boldsymbol{z}_k)^T$. These quantities are (in the Gaussian case) sufficient statistics for the joint prior distribution in (2.30).

For our fixed-point smoothing UKF with the joint prior distribution, the steps (2.27) - (2.29) are equivalent to running the filter with the augmented state transition and measurement functions of (2.35) and (2.36). These are augmented to accomodate the extended state vector from the joint prior distribution, and to maintain the state $\boldsymbol{x}_k$.

$$\begin{bmatrix}\boldsymbol{x}_k \\ \boldsymbol{x}_{k+n}\end{bmatrix} = f^a(\boldsymbol{x}_k, \boldsymbol{x}_{k+n-1}) = \begin{bmatrix}\boldsymbol{x}_k \\ f(\boldsymbol{x}_{k+n-1})\end{bmatrix} \tag{2.35}$$

$$\boldsymbol{z}_n = h^a(\boldsymbol{x}_k, \boldsymbol{x}_{k+n}) = h(\boldsymbol{x}_{k+n}) \tag{2.36}$$

When the delayed measurement becomes available, the measurement function is augmented to include the delayed measurement and it is fused using the typical UKF method. The unscented fixed-point smoother maintains the cross-covariance between the current state and a chosen previous state.

A requirement for this method is that there is non-zero process-noise. This is easy to see from the linear case, $\boldsymbol{x}_{k+1} = A\boldsymbol{x}_k$ where:

$$\begin{bmatrix}\mathrm{Var}(\boldsymbol{x}_k|\boldsymbol{z}_k) & \mathrm{Cov}(\boldsymbol{x}_k, \boldsymbol{x}_{k+1}|\boldsymbol{z}_k) \\ \mathrm{Cov}(\boldsymbol{x}_{k+1}, \boldsymbol{x}_k|\boldsymbol{z}_k) & \mathrm{Var}(\boldsymbol{x}_{k+1}|\boldsymbol{z}_k)\end{bmatrix} = \begin{bmatrix}\mathrm{Var}(\boldsymbol{x}_k|\boldsymbol{z}_k) & \mathrm{Var}(\boldsymbol{x}_k|\boldsymbol{z}_k)A^T \\ A\mathrm{Var}(\boldsymbol{x}_k|\boldsymbol{z}_k) & A\mathrm{Var}(\boldsymbol{x}_k|\boldsymbol{z}_k)A^T\end{bmatrix} \tag{2.37}$$

Which, by Schur's complement, is not positive definite. This is important for the Cholesky factorization in the generation of sigma-points. Intuitively we are saying that since the new state is given uniquely by the previous state, the covariance matrix will not contain enough information for the augmented state vector to be described by a Gaussian distribution.

2.3.4 Displacement Measurements

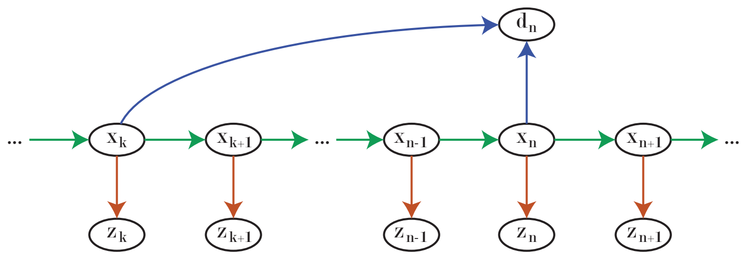

Figure 10: *Hidden markov model of a displacement measurement depending on two state instances $\boldsymbol{x}_k$ and $\boldsymbol{x}_n$.*

As discussed in the visual odometry chapter, VO measurements come from two sets of features, identified to be common for two sequential pictures. The underlying sensor model is affected by: lens distortion, camera resolution and camera noise, and stochastic traits of the underlying feature identification and matching algorithms. Evaluating the true sensor model requires difficult analysis of the underlying mechanisms and significant experimentation. When the view is obfuscated by plants, it is questionable whether a reliable VO sensor model can be constructed at all. An alternative is simplifying the sensor model as a displacement between two state estimates, see Fig.10. This is similar to what was done by Mourikis and Roumeliotis (2007). The measurement noise is modeled as additive Gaussian on top of this displacement measurement. This means that we have a description of a displacement measurement $\boldsymbol{d}_n$, dependent on the configurations $\boldsymbol{x}_n$ and $\boldsymbol{x}_k$, as:

$$\boldsymbol{d}_n = \begin{bmatrix} R_o^k(-\psi_k) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_n - x_k \\ y_n - y_k \\ \psi_n - \psi_k \end{bmatrix} + \boldsymbol{v}_n^{VO}$$

$$p(\boldsymbol{v}_n^{VO}) = \mathcal{N}(0, R_n^{VO})$$

$$R_o^k(-\psi_k) = \begin{bmatrix} \cos(-\psi_k) & -\sin(-\psi_k) \\ \sin(-\psi_k) & \cos(-\psi_k) \end{bmatrix}$$

$R_o^k(-\psi_k)$ is the rotation from the origin of the localization frame to the heading at time k, and should not be confused with the VO measurement noise covariance matrix R_n^{VO}. The way we apply Horn's method means that translations seen of the features will be represented in terms of the reference frame of the oldest picture. The rotation occuring between the two images introduces a nonlinearity to the sensor model. This means we have the nonlinear sensor model:

$$p(\boldsymbol{d}_n|\boldsymbol{x}_k, \boldsymbol{x}_n) = \mathcal{N}\left(\boldsymbol{d}_n; \begin{bmatrix} R_o^k(-\psi_k) & 0 \\ 0 & 1 \end{bmatrix} (\boldsymbol{x}_n - \boldsymbol{x}_k), R_n^{VO}\right) \tag{2.38}$$

Thus, the Bayesian measurement update (2.32) is carried out by means of (2.38). A model which is dependent on the two states we are already maintaining with the joint distribution. This is if the displacement measurement is available at the exact time the second picture is taken.

2.3.5 Delayed Displacement Measurement

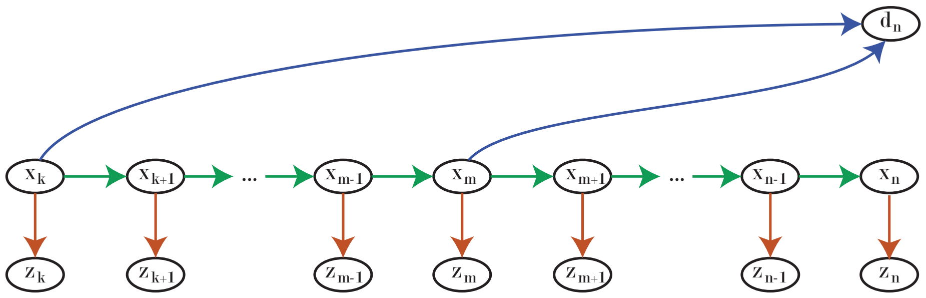

***Figure 11:** Hidden Markov model for the delayed displacement measurement.*

In reality, we often have the situation depicted in Fig.11. Here a displacement measurement, dependent on the states $\boldsymbol{x}_k$ and $\boldsymbol{x}_m$ at times t_k and t_m becomes available at time t_n. From the previous reasonings it is apparent that this means that we need to maintain three states in our joint distribution. Instead of a fixed-point smoother, we are constructing a nonstandard smoother, which we term a “multiple-point” smoother. This is simply a generalization of the procedure from (2.27) - (2.29) smoothing two states instead.

To fuse the delayed displacement measurement, we must formulate the joint probability distribution $p(\boldsymbol{x}_k, \boldsymbol{x}_m, \boldsymbol{x}_n|\boldsymbol{z}_{k:n})$.

Given a joint distribution $p(\boldsymbol{x}_h, \boldsymbol{x}_i, \boldsymbol{x}_j|\boldsymbol{z}_{h:j})$ with indices $j > i > h$, and a sequence of measurements $\boldsymbol{z}_{h:j}$, we can find the distribution $p(\boldsymbol{x}_h, \boldsymbol{x}_i, \boldsymbol{x}_m|\boldsymbol{z}_{h:m})$ for any $m > j$. This is done by repeating the following steps as necessary:

1. Extend the joint distribution:

$$p(\boldsymbol{x}_h, \boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{x}_{j+1}|\boldsymbol{z}_{h:j}) = p(\boldsymbol{x}_{j+1}|\boldsymbol{x}_j)p(\boldsymbol{x}_h, \boldsymbol{x}_i, \boldsymbol{x}_j|\boldsymbol{z}_{h:j}) \tag{2.39}$$

2. Fuse the latest measurement with Bayes theorem:

$$p(\boldsymbol{x}_h, \boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{x}_{j+1}|\boldsymbol{z}_{h:j+1}) = \frac{p(\boldsymbol{z}_{j+1}|\boldsymbol{x}_{j+1})p(\boldsymbol{x}_h, \boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{x}_{j+1}|\boldsymbol{z}_{h:j})}{p(\boldsymbol{z}_{j+1}|\boldsymbol{z}_{h:j})} \tag{2.40}$$

3. Marginalize the unnecessary state:

$$p(\boldsymbol{x}_h, \boldsymbol{x}_i, \boldsymbol{x}_{j+1}|\boldsymbol{z}_{h:j+1}) = \int_{\boldsymbol{x}_{j+1}} p(\boldsymbol{x}_h, \boldsymbol{x}_i, \boldsymbol{x}_j, \boldsymbol{x}_{j+1}|\boldsymbol{z}_{h:j+1}) d\boldsymbol{x}_j \tag{2.41}$$

This is the trivial generalization of the measurement fusion procedure given an initial joint prior distribution maintaining three states instead of two.

Just as in the fixed-point case, from a joint distribution of $p(\boldsymbol{x}_k, \boldsymbol{x}_m|\boldsymbol{z}_{k:m})$, we can construct the initial joint prior distribution for our multiple-point smoother. First extending the state vector to include the next state:

$$p(\boldsymbol{x}_k, \boldsymbol{x}_m, \boldsymbol{x}_{m+1}|\boldsymbol{z}_{k:m}) = p(\boldsymbol{x}_{m+1}|\boldsymbol{x}_m)p(\boldsymbol{x}_k, \boldsymbol{x}_m|\boldsymbol{z}_{k:m}) \tag{2.42}$$

And then fusing the latest measurement using Bayes theorem:

$$p(\boldsymbol{x}_k, \boldsymbol{x}_m, \boldsymbol{x}_{m+1}|\boldsymbol{z}_{k:m+1}) = \frac{p(\boldsymbol{x}_{m+1}|\boldsymbol{x}_m)p(\boldsymbol{x}_k, \boldsymbol{x}_m, \boldsymbol{x}_{m+1}|\boldsymbol{z}_{k:m})}{p(\boldsymbol{z}_{m+1}|\boldsymbol{z}_{k:m})} \tag{2.43}$$

This means that given an initial distribution $p(\boldsymbol{x}_k|\boldsymbol{z}_k)$: we apply (2.30)-(2.31), then repeat steps (2.27)-(2.29) until we have the distribution $p(\boldsymbol{x}_k, \boldsymbol{x}_m|\boldsymbol{z}_{k:m})$. Thereafter we apply (2.42)-(2.43), repeat steps (2.39)-(2.41) until we have the distribution $p(\boldsymbol{x}_k, \boldsymbol{x}_m, \boldsymbol{x}_n|\boldsymbol{z}_{k:n})$.

And then the delayed displacement measurement can be fused using Bayes theorem:

$$p(\boldsymbol{x}_k, \boldsymbol{x}_m, \boldsymbol{x}_n|\boldsymbol{z}_{k:n}, \boldsymbol{d}_n) = \frac{p(\boldsymbol{d}_n|\boldsymbol{x}_k, \boldsymbol{x}_m)p(\boldsymbol{x}_k, \boldsymbol{x}_m, \boldsymbol{x}_n|\boldsymbol{z}_{k:n})}{p(\boldsymbol{d}_n|\boldsymbol{z}_{k:n})} \tag{2.44}$$

Finally, the desired posterior belief of the latest state is found by marginalization:

$$p(\boldsymbol{x}_n|\boldsymbol{z}_{k:n}, \boldsymbol{d}_n) = \int_{\boldsymbol{x}_m} \int_{\boldsymbol{x}_k} p(\boldsymbol{x}_k, \boldsymbol{x}_m, \boldsymbol{x}_n|\boldsymbol{z}_{k:n}, \boldsymbol{d}_n) d\boldsymbol{x}_k d\boldsymbol{x}_m \tag{2.45}$$

The keys to this procedure is the initialization of the joint prior distribution (2.30) and (2.42), and the fusion of the delayed displacement measurement (2.44).

This procedure leaves us with the desired belief given that we have maintained accurate estimates of $\boldsymbol{x}_k$, $\boldsymbol{x}_m$ and $\boldsymbol{x}_n$. Constructing the fixed-point smoothing has already been described. The next is to be able to extend the fixed-point smoother to maintain more states on demand.

2.3.6 Multiple-Point Smoother

Suppose we are running a fixed-point smoother as previously described. It is iterating according to the system:

$$
\begin{aligned}
\boldsymbol{x}^a_{m+1} &= f^a(\boldsymbol{x}^a_m) + \boldsymbol{w}^a_m \\
\begin{bmatrix} \boldsymbol{x}_k \\ \boldsymbol{x}_{m+1} \end{bmatrix} &= \begin{bmatrix} \boldsymbol{x}_k \\ f(\boldsymbol{x}_m) \end{bmatrix} + \begin{bmatrix} 0 \\ \boldsymbol{w}_m \end{bmatrix} \\
p(\boldsymbol{w}_m) &= \mathcal{N}(\boldsymbol{w}_m; 0, Q_m)
\end{aligned}
\tag{2.46}
$$

This is an augmented representation of the underlying system, augmented so that it maintains a smoothed version of $\boldsymbol{x}_k$. We would like to extend the number of states by performing the step (2.42), so that we can smooth both $\boldsymbol{x}_k$ and $\boldsymbol{x}_m$ while maintaining the current configuration. To do so, one has to find the distribution:

$$
\mathcal{N}\left(\begin{bmatrix} \boldsymbol{x}_k \\ \boldsymbol{x}_m \\ \boldsymbol{x}_{m+1} \end{bmatrix}; \begin{bmatrix} \mathrm{E}(\boldsymbol{x}_k) \\ \mathrm{E}(\boldsymbol{x}_m) \\ \mathrm{E}(\boldsymbol{x}_{m+1}) \end{bmatrix}, \begin{bmatrix} \mathrm{Var}(\boldsymbol{x}_k) & \mathrm{Cov}(\boldsymbol{x}_k, \boldsymbol{x}_m) & \mathrm{Cov}(\boldsymbol{x}_k, \boldsymbol{x}_{m+1}) \\ \mathrm{Cov}(\boldsymbol{x}_m, \boldsymbol{x}_k) & \mathrm{Var}(\boldsymbol{x}_m) & \mathrm{Cov}(\boldsymbol{x}_m, \boldsymbol{x}_{m+1}) \\ \mathrm{Cov}(\boldsymbol{x}_{m+1}, \boldsymbol{x}_k) & \mathrm{Cov}(\boldsymbol{x}_{m+1}, \boldsymbol{x}_m) & \mathrm{Var}(\boldsymbol{x}_{m+1}) \end{bmatrix} \right)
\tag{2.47}
$$

At time t_m the fixed-point smoother has all the information except for $\mathrm{E}(\boldsymbol{x}_{m+1}|\boldsymbol{z}_{k:m})$, $\mathrm{Cov}(\boldsymbol{x}_k, \boldsymbol{x}_{m+1}|\boldsymbol{z}_{k:m})$, $\mathrm{Cov}(\boldsymbol{x}_m, \boldsymbol{x}_{m+1}|\boldsymbol{z}_{k:m})$ and $\mathrm{Var}(\boldsymbol{x}_{m+1}|\boldsymbol{z}_{k:m})$. Given the current knowledge of the system, and the current state estimates, we desire to estimate the unknown values.

From the definition of covariance:

$$
\begin{aligned}
\mathrm{Cov}(\boldsymbol{x}^a_m, f^a(\boldsymbol{x}^a_m)) &= \mathrm{Cov}\left(\begin{bmatrix} \boldsymbol{x}_k \\ \boldsymbol{x}_m \end{bmatrix}, \begin{bmatrix} \boldsymbol{x}_k \\ \boldsymbol{x}_{m+1} \end{bmatrix} \right) \\
&= \begin{bmatrix} \mathrm{Var}(\boldsymbol{x}_k) & \mathrm{Cov}(\boldsymbol{x}_k, \boldsymbol{x}_{m+1}) \\ \mathrm{Cov}(\boldsymbol{x}_m, \boldsymbol{x}_k) & \mathrm{Cov}(\boldsymbol{x}_m, \boldsymbol{x}_{m+1}) \end{bmatrix}
\end{aligned}
\tag{2.48}
$$

This means that accessing submatrices of an evaluation of the state to predicted state covariance will give us the necessary quantities. This cross-covariance can be evaluated both for a UKF and for an EKF. For the EKF, this is simply:

$$
\mathrm{Cov}(\boldsymbol{x}^a_m|\boldsymbol{z}_{k:m}) = P^a_{m|m} \begin{bmatrix} I & 0 \\ 0 & F^T_m \end{bmatrix}
\tag{2.49}
$$

F^T_m is the Jacobian matrix of the function f evaluated for states $\boldsymbol{x}_m$ at timestep t_m.

And for the UKF:

$$
\mathrm{Cov}(\boldsymbol{x}^a_m|\boldsymbol{z}_{k:m}) = \sum_{i=0}^{2L+1} w_i \left(\mathcal{X}^a_{i,m|m} - \hat{\boldsymbol{x}}_m \right) \left(\mathcal{X}^a_{i,m+1|m} - \hat{\boldsymbol{x}}_{m+1} \right)^T
\tag{2.50}
$$

The needed cross-covariances can be found by accessing the upper right and lower right submatrices. The matrix $\mathrm{Var}(\boldsymbol{x}_{m+1}|\boldsymbol{z}_{k:m})$ can be found by the lower right submatrix of the next prediction. Which is, for the EKF:

$$\mathrm{Var}(\boldsymbol{x}_{m+1}|\boldsymbol{z}_m) = \begin{bmatrix}0 & 0\\0 & I\end{bmatrix}\left(\begin{bmatrix}I & 0\\0 & F_m\end{bmatrix} P^a_{m|m} \begin{bmatrix}I & 0\\0 & F_m^T\end{bmatrix} + \begin{bmatrix}0 & 0\\0 & Q_m\end{bmatrix}\right) \tag{2.51}$$

and for the UKF:

$$\mathrm{Var}(\boldsymbol{x}_{m+1}|\boldsymbol{z}_m) = \begin{bmatrix}0 & 0\\0 & I\end{bmatrix}\left(\sum_{i=0}^{2L+1} w_i \left(\mathcal{X}^a_{i,m+1|m} - \hat{\boldsymbol{x}}_{m+1|m}\right)\left(\mathcal{X}^a_{i,m+1|m} - \hat{\boldsymbol{x}}_{m+1|m}\right)^T \right.$$
$$\left. + \begin{bmatrix}0 & 0\\0 & Q_m\end{bmatrix}\right) \tag{2.52}$$

Our multiple-point smoother will at time t_{m+1} create this joint prior instead of performing a normal prediction step, and augment the transition function, measurement function and transition covariance accordingly. This procedure will be referred to as *extend* in the implementation as it extends the state vector into the next time instance. Just as for the fixed-point smoother, the multiple-point *extend* also requires that there is non-zero process noise. In the rest of this thesis the method of creating the joint prior distributions on demand with *extend* will be called the “joint” method.

2.3.7 Stochastic Cloning

Stochastic cloning is outlined in Roumeliotis and Burdick (2002) and used in Merwe and Wan (2004) under the term “latency compensation”. It is similar to the machinery described in the previous section. However, stochastic cloning is, in the mentioned references, developed without any formal construction of the joint distribution. Instead the lagged state is simply “cloned” in order to create an augmented state vector. The cloning procedure is described as: Given the pair of the state estimate $\hat{\boldsymbol{x}}_{k|k}$ and of the state covariance estimate $P_{k|k}$, augment them such that:

$$\begin{aligned}\hat{\boldsymbol{x}}^a_{k|k} &= \begin{bmatrix}\hat{\boldsymbol{x}}_{k|k}\\ \hat{\boldsymbol{x}}_{k|k}\end{bmatrix}\\ P^a_{k|k} &= \begin{bmatrix}P_{k|k} & P_{k|k}\\ P_{k|k} & P_{k|k}\end{bmatrix}\end{aligned} \tag{2.53}$$

The transition function, measurement function and transition noise covariance matrix are also augmented to accomodate the smoothed fixed state. This state covariance matrix $P^a_{k|k}$ is singular and the corresponding distribution is therefore not a well-defined Gaussian distribution. However, discarding the Bayesian paradigm and working solely in terms of the EKF, this singularity does not pose any difficulties if process noise is present. With an EKF, during the next predict step, we will get:

$$\begin{aligned}\hat{\boldsymbol{x}}^a_{k+1|k} &= \begin{bmatrix} I & 0 \\ 0 & F_k \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{x}}_{k|k} \\ \hat{\boldsymbol{x}}_{k|k} \end{bmatrix} = \begin{bmatrix} \hat{\boldsymbol{x}}_{k|k} \\ \hat{\boldsymbol{x}}_{k+1|k} \end{bmatrix} \\ P^a_{k+1|k} &= \begin{bmatrix} I & 0 \\ 0 & F_k \end{bmatrix} \begin{bmatrix} P_{k|k} & P_{k|k} \\ P_{k|k} & P_{k|k} \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & F_k^T \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & Q_k \end{bmatrix} = \begin{bmatrix} P_{k|k} & P_{k|k}F_k^T \\ F_k P_{k|k} & F_k P_{k|k} F_k^T + Q_k \end{bmatrix}\end{aligned} \tag{2.54}$$

which is exactly the same as the outcome from the optimal approach presented in the previous sections. For the unscented transformation to work, a Cholesky factorization is required. Therefore, positive definiteness of the state covariance must be ensured. This can be done by the following approximation, where ϵ is a very small number:

$$\begin{aligned} p(\boldsymbol{x}_k, \boldsymbol{x}_k|\boldsymbol{z}_k) &= p(\boldsymbol{x}_k|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{z}_k) \\ p(\boldsymbol{x}_k, \boldsymbol{x}_k|\boldsymbol{z}_k) &= \mathcal{N}(\boldsymbol{x}_k; \hat{\boldsymbol{x}}_k, P_{k|k} + \epsilon I)\mathcal{N}(\boldsymbol{x}_k; \hat{\boldsymbol{x}}_k, P_{k|k}) \end{aligned} \tag{2.55}$$

This means that we approximate the cloning procedure by adding a negligible noise to the cloned state, a regularization constant. In other words, for the UKF, the cloning procedure gives the augmented pair of expected value and state covariance:

$$\begin{aligned} \hat{\boldsymbol{x}}^a_{k|k} &= \begin{bmatrix} \hat{\boldsymbol{x}}_{k|k} \\ \hat{\boldsymbol{x}}_{k|k} \end{bmatrix} \\ P^a_{k|k} &= \begin{bmatrix} P_{k|k} & P_{k|k} \\ P_{k|k} & P_{k|k} + \epsilon I \end{bmatrix} \end{aligned} \tag{2.56}$$

where ϵ is chosen to be sufficiently small to not adversely effect the filter while still ensuring positive definiteness for the Cholesky decomposition. The ϵ also allows us to use stochastic cloning when a system has no process noise. But one can always choose a process noise sufficiently small for the two methods to be practically equivalent.

2.3.8 Fixed-Lag Smoother

Another way one could maintain all the information necessary for fusion of delayed displacement measurements is to use a fixed-lag smoother. This is based on a method described in Challa et al. (2002). The fixed-lag smoother follows the system:

$$\boldsymbol{x}^a_{n+1} = \begin{bmatrix} \boldsymbol{x}_k \\ \vdots \\ \boldsymbol{x}_m \\ \vdots \\ \boldsymbol{x}_{n+1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_{k+1} \\ \vdots \\ \boldsymbol{x}_{m+1} \\ \vdots \\ f(\boldsymbol{x}_n) \end{bmatrix} + \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \vdots \\ \boldsymbol{w}_n \end{bmatrix} \tag{2.57}$$

Recall that the displacement measurement is from $\boldsymbol{x}_k$ to $\boldsymbol{x}_m$ and the current state of the system is $\boldsymbol{x}_n$. Let us say that the fixed-lag smoother has arrived at time t_n, and is about to fuse a delayed displacement measurement. For simplicity, let us consider an EKF fixed-lag smoother. If we are only fusing the delayed displacement measurement, the sensor model becomes:

$$\boldsymbol{d}_n = \begin{bmatrix} H_n^k & 0 & \dots & 0 & H_n^m & 0 & \dots & 0 \end{bmatrix} \boldsymbol{x}_n^a \tag{2.58}$$

With H_n^k being the measurement jacobian related to state $\boldsymbol{x}_k$, and similar for H_n^m. The filter has state estimate and corresponding covariance:

$$\begin{gathered} \hat{\boldsymbol{x}}_n^a = \begin{bmatrix} \hat{\boldsymbol{x}}_{k|n} \\ \vdots \\ \hat{\boldsymbol{x}}_{m|n} \\ \vdots \\ \hat{\boldsymbol{x}}_{n|n} \end{bmatrix} \\ P_{n|n}^a = \begin{bmatrix} P_{k|n} & \dots & C_n^{k,m} & \dots & C_n^{k,n} \\ \vdots & & \vdots & & \vdots \\ C_n^{m,k} & \dots & P_{m|n} & \dots & C_n^{m,n} \\ \vdots & & \vdots & & \vdots \\ C_n^{n,k} & \dots & C_n^{n,m} & \dots & P_{n|n} \end{bmatrix} \end{gathered} \tag{2.59}$$

Where $C_n^{m,k}$ means the covariance between state $\boldsymbol{x}_k$ and $\boldsymbol{x}_m$ given information up to time t_n.

With these equations, looking at the Kalman gain equation for EKF (2.16), we have that:

$$\begin{bmatrix} K_n^k \\ \vdots \\ K_n^m \\ \vdots \\ K_n^n \end{bmatrix} = \begin{bmatrix} P_{k|n} H_n^k + C_n^{k,m} H_n^m \\ \vdots \\ C_n^{m,k} H_n^k + P_{m|n} H_n^m \\ \vdots \\ C_n^{n,k} H_n^k + C_n^{n,m} H_n^m \end{bmatrix} \begin{bmatrix} H_n^k P_{k|n} H_n^{kT} + H_n^k C_n^{k,m} H_n^{mT} + H_n^m C_n^{m,k} H_n^{kT} + H_n^m P_{m|n} H_n^{mT} \end{bmatrix}^{-1} \tag{2.60}$$

where K_n^k means the Kalman gain for time t_n with respect to state k.

As our sensor model Jacobians consists of H_n^m, H_n^k and zero matrices, the Kalman gain is mainly constructed from $P_{k|n}$ and $P_{m|n}$. The rest are the cross-covariances associated with these states. This means that the essential information for fusing the delayed displacement measurement is the state covariance associated with $P_{k|n}$ and $P_{m|n}$, and the cross-covariances. The cross-covariances associated with other states are not necessary for formulating the

Kalman gain, and hence the fusion of the measurement into the current state estimate $\hat{\boldsymbol{x}}_n$.

Our multiple-point smoother contains the same information. The difference is that the the fixed-lag smoother maintains significantly more unnecessary state estimates and state covariance estimates. If there are s states in the fixed-lag smoother, we are maintaining $s-3$ more states and s^2-9 more covariance matrices than we need. These unnecessary states lead to the fixed-lag smoother being too computationally costly. For a 2D robot taking pictures every 2 s, a filter updating at 0.05 s and no processing time for the pictures, the fixed-lag smoother will have to maintain 40 state vectors containing 5 elements, and 1600 covariance matrices containing 25 elements.

2.4 Rotation

The robot is considered to be moving in 2D, but some of the sensors available give measurements in the plane, others give them with respect to some reference frame. As such it is important to present some common definitions of reference frames that will be used in this thesis. The definitions are taken from Vik (2014) with the addition of an ENU frame and modification of the BODY frame.

2.4.1 Reference Frames

ECI

The Earth Centered Inertial (ECI) frame's origin is coincident with the center of the Earth. It is defined with the x-axis pointing towards the *vernal equinox,* and the z-axis pointing along the Earth's rotation axis at some initial time. The y-axis completes the right handed orthogonal coordinate system. This frame can be considered an inertial frame for terrestrial navigation purposes. That is, Newton's laws of motion apply for this frame. All inertial sensor measurements are relative to this frame and resolved in the platform frame.

ECEF

The Earth Centered Earth Fixed (ECEF) frame also has it's origin in the in the center of the Earth. It is defined with the x-axis pointing towards the intersection of 0° longitude (Greenwich meridian) and 0° latitude (Equator). The z-axis points along the Earth's rotation axis, and the y-axis complete the right handed orthogonal coordinate system. The ECEF frame rotates relative to the ECI frame with the Earth rotation rate ω_e, and is *not* an inertial coordinate frame. Both Cartesian and ellipsoidal coordinates are used to represent position in the ECEF frame.

NED

The North East Down (NED) frame is defined relative to the Earth's reference ellipsoid. The z-axis points downward perpendiculary to the tangent plane of the ellipsoid, and the x-axis points towards true north. The y-axis points towards east to complete the orthogonal coordinate system. The ellipsoid currently used for GPS is the *WGS-84* ellipsoid.

ENU

The East North Up (ENU) frame is also defined relative to the Earth's reference ellipsoid. The z-axis points upward perpendicularly to the tangent plane of the ellipsoid, and the y-axis points towards true north. The x-axis points towards east to complete the orthogonal coordinate system. This is the reference frame most commonly used in the Robotic Operating System (ROS), which is the programming platform used in the development of Asterix.

BODY

The classic BODY frame is moving and rotating with the vehicle. The origin coincides with the origin of the NED frame, and the x-axis points in the forward direction, the y-axis to the right side, and the z-axis downward. The BODY frame is related to the NED frame through the Euler angles roll, pitch and yaw. This is the convention normally used.

ROS-BODY

The robot model that was chosen has positive rotation around the up direction, and we will therefore refer to a ROS-BODY frame with x-axis pointing forwards, y-axis to the left, and z-axis upwards. This ROS-BODY frame is related to the ENU frame by the Euler Angles. This is the BODY frame convention most commonly used in ROS.

I

I is the reference frame of strapdown instruments attached to the robot. The gyro and accelerometer instruments are, in the case of strapdown systems, nominally aligned with the platform and the ROS-BODY frame. But due to misalignments the strapdown instrument frame should not be considered perfectly orthogonal or coincident with the ROS-BODY frame.

2.4.2 Ellipsoidal to Cartesian ECEF

A GPS unit reports position in terms of the ellipsoidal ECEF coordinates longitude, l, latitude, μ, and altitude, h. The transformation from ellipsoidal to Cartesian is given in Vik (2014) as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} (N+h)\cos\mu\cos l \\ (N+h)\cos\mu\sin l \\ (N(1-\varepsilon^2)+h)\sin\mu \end{bmatrix} \tag{2.61}$$

Where:

$$N = \frac{r_e}{\sqrt{1-\varepsilon^2\sin\mu^2}} \tag{2.62}$$

And the parameters of the *WGS-84* ellipsoid are given in Tab.1.

Symbol	Description	Value
r_e	Equatorial radius of ellipsoid (Major axis)	6378137 m
r_p	Polar axis radius of ellipsoid (Minor axis)	6356752 m
ω_e	Angular velocity of Earth	$7292115 \cdot 10^{-11}$ rad/s
μ_g	Gravitational constant of Earth	$3986005 \cdot 10^{8}\text{m}^3/\text{s}^2$
ε	Eccentricity of ellipsoid	0.0818

Table 1: WGS-84 *Parameters from Vik (2014)*

2.4.3 ECEF to ENU

Cartesian ECEF coordinates can be converted to ENU coordinates by applying a rotation based on the longitude and latitude of the origin of the ENU frame. This derivation is similar to what is done in (Vik, 2014, p.11). First we construct the rotation necessary to make the x-axis of the ECEF frame align with the east axis of the local ENU frame.

$$R_{z_{ECEF}}(-l-\pi/2) = \begin{bmatrix} \sin l & \cos l & 0 \\ -\cos l & \sin l & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.63}$$

Then we construct the rotation necessary to make the z-axis of the current frame align with the up axis of the local ENU frame.

$$R_x(\mu-\pi/2) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \sin\mu & \cos\mu \\ 0 & -\cos\mu & \sin\mu \end{bmatrix} \tag{2.64}$$

Giving the final transformation from ECEF to ENU as their product:

$$R_{ECEF}^{ENU}(l,\mu) = \begin{bmatrix} -\sin l & \cos l & 0 \\ -\cos l\sin\mu & -\sin l\sin\mu & \cos\mu \\ \cos l\cos\mu & \sin l\cos\mu & \sin\mu \end{bmatrix} \tag{2.65}$$

2.4.4 The Local ENU Frame

We are interested in maintaining a local map of the movement of the robot. To do this we construct a local ENU frame from which the position is described. When the first GPS measurements arrive, we define that to be the elliptic coordinates ,$[l_0,\mu_0,h_0]$, of the origin of the local ENU frame. The origin has the Cartesian ECEF coordinates $[x_0,y_0,z_0]^T$. When a new GPS measurement arrives, to give the local ENU coordinates, the procedure is as follows:

1. Convert the measurement into ECEF coordinates $[x,y,z]^T$
2. Compute the local ENU coordinates with:

$$\begin{bmatrix} x_E \\ y_N \\ z_U \end{bmatrix} = R_{ECEF}^{ENU}(l_0,\mu_0)\left(\begin{bmatrix} x-x_0 \\ y-y_0 \\ z-z_0 \end{bmatrix}\right) \tag{2.66}$$

If the localization filters have been running for a while before the first GPS measurement arrives, there will be an offset between the location of the filters current origin and the local ENU frame. In practice the first GPS measurement arrives early in the datasets, and the localization filters are therefore reset to the ENU origin at that point.

2.5 Local Versus Global

Localization is done in a map. These maps are either defined by means of proprioceptive sensors or extroreceptive sensors. These terms are adopted from Mourikis and Roumeliotis (2006). An extroreceptive sensor, such as GPS, provides estimates of the robots position with respect to some known locations in the world. A proprioceptive sensor, such as wheel encoders, provides estimates of the robots position with respect to its previous position. This section presents some concepts for probabilistic localization in terms of proprioceptive and extroreceptive measurements.

2.5.1 Proprioceptive Measurements

Proprioceptive measurements give us an estimate of the motion of the robot relative to an initial position. We essentially integrate the measurements to arrive at the estimated position. These local measurements cannot improve the state covariance beyond what was set initially.

2.5.2 Extroreceptive Measurements

Extroreceptive measurements give us an estimate of the robot's states with respect to some known objects. For example distance to known landmarks or distance to satellites in orbit. The precision with which we know the initial location of these objects is what we use to refine our own estimates. This means that with a perfect laser range finder, we would not be able to estimate our distance from an object better than the object's known initial position. For GPS, the orbit trajectories of positioning satellites are monitored closely and are highly stable. They transmit a signal describing the time of transmission with a known pseudorandom sequence. By aligning the sequence on the receiver side and compensating for known clock errors, the time of arrival can be computed and the distance to the satellite evaluated.

2.5.3 Delay And Smoothing

A lagged state will be smoothed by extroreceptive measurements and proprioceptive measurements. But given only proprioceptive sensors, there is a limit to how much smoothing can contribute. This is visualized in Fig.12 and it was observed in simulations. Extroreceptive measurements should, in theory, be able to improve our estimate of the smoothed state significantly better than the proprioceptive measurements. In experiments with the datasets, this was observed where the heading of a lagged state was gradually shifted according to new GPS measurements.

The smoothing cannot go beyond the knowledge of the underlying system model. If the system model describes a system that is assumed to have a con-

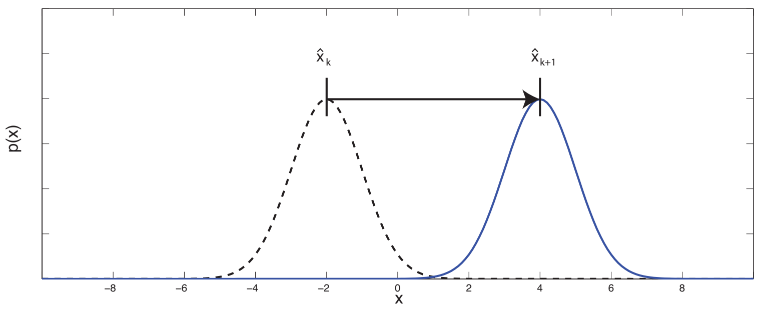

Figure 12: *Perfectly knowing the transition from a state to another will only maintain the initial state probability estimate.*

stant velocity, but our observations are of a system performing rapidly varying movement, the filter will consider the motion highly unlikely and disregard the observations. This is an argument for masking the localization filter. We might rely too heavily on the smoothed estimate of the lagged state when fusing it. Discrepancy between modeling assumptions and the actual system behavior can lead to the smoothed state having a lower state covariance than is correct. As the extroreceptive measurements are direct measurements of the system states, they will have a greater effect on the smoothed state. When smoothing extroreceptive measurements, care should be taken to ensure that potential modeling errors do not lead to loss in performance. This model inconsistency was observed in simulations of the system where our modelling choice meant that the filter was not able to accurately track the robot during a sharp turn.

2.5.4 VO Sensor Model

The VO sensor model (2.38) measures the displacement between two states with respect to our smoothed heading estimate. As with most rotated transformations, this is not accurately described by a Gaussian. In Fig.13, the sensor model has been described by a Monte-Carlo simulation of the model with Gaussian uncertainty in the VO measurements and yaw. We can see that, similar to all cases of rotated transformations, the distribution is easily affected by our estimate of the heading. When the robot is moving with only proprioceptive measurements, we will accumulate uncertainty during any periods the displacement measurements are unavailable, diminishing the gain in accuracy we have by using VO. These measurements describe motion relative to the robot and are therefore proprioceptive measurements.

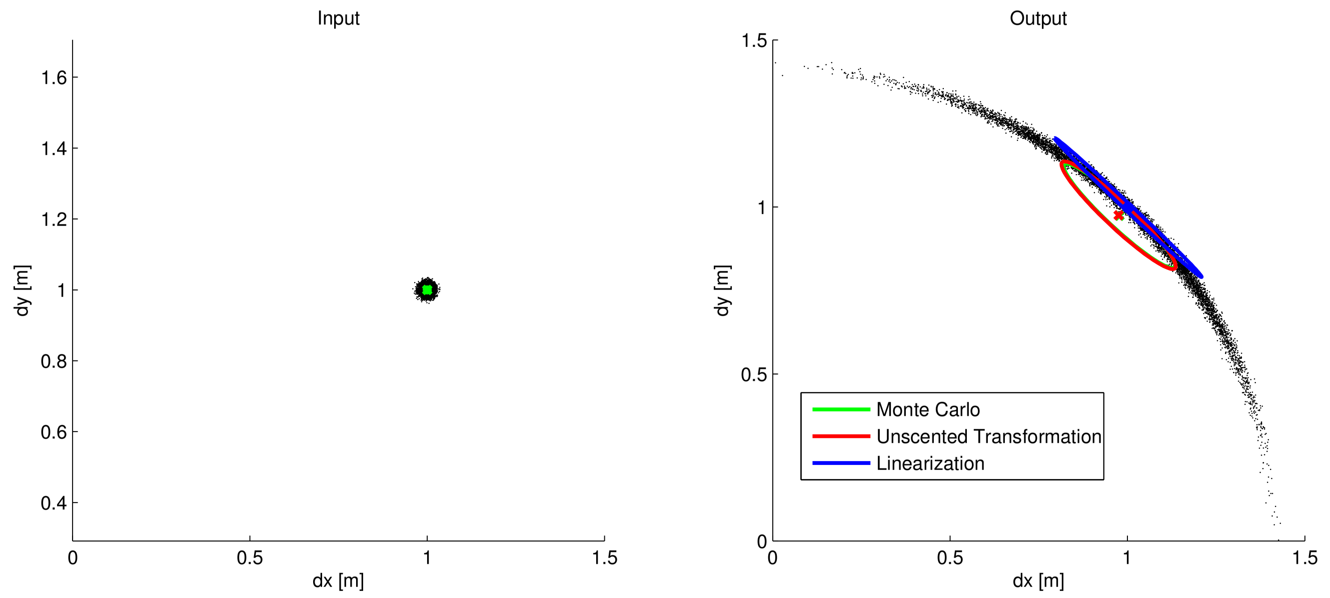

Figure 13: *Monte Carlo simulation of the VO sensor model with* $p(\Delta x_{VO}) = \mathcal{N}(\Delta x_{VO}; 1, 0.001)$, $p(\Delta y_{VO}) = \mathcal{N}(\Delta y_{VO}; 1, 0.001)$ *and* $p(\psi_k) = \mathcal{N}(0, 0.05)$. *The angle* ψ_k *greatly affects the resulting distribution.*

3 Implementation

3.1 Module Testing

To assess how the implementations of the filters and delay fusion methods work, a simple example based on target tracking was studied. This was instrumental for learning what to expect from the methods and whether the filter implementations worked correctly.

3.1.1 Target Tracking

In Challa et al. (2003), the system used to test the methods was a classical white-noise acceleration model. This has been modified to test our delayed measurement scenario and our unscented fixed-point smoother:

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix} \boldsymbol{x}_k + \boldsymbol{w}_k \\ z_k &= \begin{bmatrix} 1 & 0 \end{bmatrix} \boldsymbol{x}_k + v_k \\ z_k^d &= \begin{bmatrix} 1 & 0 \end{bmatrix} \boldsymbol{x}_{k-N} + v_k^d \\ p(\boldsymbol{w}_k) &= \mathcal{N}(\boldsymbol{w}_k; 0, Q) \\ p(v_k) &= \mathcal{N}(v_k; 0, R) \\ p(v_k^d) &= \mathcal{N}(v_k^d; 0, R^d) \\ \alpha &= \frac{R^d}{R} \end{aligned} \tag{3.1}$$

where T is the sampling interval, w_k is the process noise with covariance:

$$Q = \begin{bmatrix} T^3/3 & T^2/2 \\ T^2/2 & T \end{bmatrix} q \tag{3.2}$$

z_k^d is a measurement delayed N iterations. Measurement z_k arrives at each time instance, but z_k^d arrives at every M time instance. The scaling factor α is used to describe the comparative accuracy of the delayed measurement. The parameter q is used the process noise. A low value corresponds to a target following close to linear path, and a high q indicates highly randomized movement.

The results and the discussion regarding this system are given in section 4.1 of the Results chapter.

3.2 Asterix Robot

3.2.1 Modeling

Asterix is modeled as a unicycle-like robot with no-slip conditions. The kinematic model for the system is based on the model presented in De La Cruz and Carelli (2006) with the only change that the castor wheel is opposite the direction of travel.

The full kinematic model used for simulations is described by:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \\ \dot{u} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} u\cos\psi - a\omega\sin\psi \\ u\sin\psi + a\omega\cos\psi \\ \omega \\ \frac{\theta_3}{\theta_1}\omega^2 - \frac{\theta_4}{\theta_1}u \\ -\frac{\theta_5}{\theta_2}u\omega - \frac{\theta_6}{\theta_2}\omega \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \frac{1}{\theta_1} & 0 \\ 0 & \frac{1}{\theta_2} \end{bmatrix} \begin{bmatrix} u_{ref} \\ \omega_{ref} \end{bmatrix} + \begin{bmatrix} \delta_x \\ \delta_y \\ \delta_\psi \\ \delta_u \\ \delta_\omega \end{bmatrix} \tag{3.3}$$

The parameter a determines the placement of the rotational center with respect to the wheelbase. In simulink this is 0.2, for the real data it was assumed 0.

The control parameters $\theta_1, ..., \theta_6$ are determined by an adaptive controller. To simplify the localization filter equations, and make it independent of the adaptive controller, we disregard the velocity references. This means that we assume the change in linear and angular velocity is zero.

The kinematic model used by the filters is therefore:

$$\dot{x} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \\ \dot{u} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} u\cos\psi - a\omega\sin\psi \\ u\sin\psi + a\omega\cos\psi \\ \omega \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} w_x \\ w_y \\ w_\psi \\ w_u \\ w_\omega \end{bmatrix} \tag{3.4}$$

with Jacobian matrix:

$$\frac{\mathrm{d}\dot{\boldsymbol{x}}}{\mathrm{d}\boldsymbol{x}} = \begin{bmatrix} 0 & 0 & -u_k\sin\psi_k - a\omega_k\cos\psi_k & \cos\psi_k & a\sin\psi_k \\ 0 & 0 & u_k\cos\psi_k - a\omega_k\sin\psi_k & \sin\psi_k & a\cos\psi_k \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.5}$$

This is a noise driven model where we are assuming that a robot rolling will keep rolling from the perspective of the filter. The only thing that changes this view is the measured velocity.

The $\boldsymbol{\delta}$ vector is a vector of dynamic and parametric uncertainties pertaining to slip and modeling errors(De La Cruz and Carelli, 2006). This is modeled as Gaussian noise for the comparatively heavy, slow moving Asterix. The vector

$\boldsymbol{w}$ is process noise that is also modeled as Gaussian. With the differential equation described by (3.4) and Jacobian (3.5), the kinematic model used by the filters is readily discretized using the forward Euler method:

$$\begin{aligned}
\dot{\boldsymbol{x}} &= f_c(\boldsymbol{x}) + \boldsymbol{\delta} \\
\boldsymbol{x}_{k+1} &= f(\boldsymbol{x}_k) + \boldsymbol{w}_k = \boldsymbol{x}_k + f_c(\boldsymbol{x})(t_{k+1} - t_k) + \boldsymbol{w}_k \\
F_k &= I + \frac{df_c(\boldsymbol{x}_k)}{d\boldsymbol{x}_{\boldsymbol{k}}}(t_{k+1} - t_k)
\end{aligned} \tag{3.6}$$

Where F_k is the transition Jacobian, f_c is the continuous differential equation and f is the discrete transition function. The forward Euler method was chosen as it is easy to compute the Jacobian, allowing us to compare the behavior of the EKF with the UKF. The simulink model and controller was based on Martins (2015).

For the simulations, the sensors considered were: an IMU (yaw and yaw rate), a GPS (X, Y coordinates), wheel encoders (linear and angular velocities) and a VO displacement measurement (difference of displacement forward, left and heading with respect to the robots pose in two time instances).

The decision to use a noise driven model gives us an intuitive remark on the tuning. The model used describes linear and angular acceleration as Gaussian noise. If the elements of the diagonal process noise covariance matrix Q are large, the corresponding state responds quickly to changes. If it is smaller, the state is less noisy. This is because we are changing how we weight a measurement, large process noise covariance means we are accepting sudden changes more.

The simulink model used is given in Fig.14 and Fig.15.

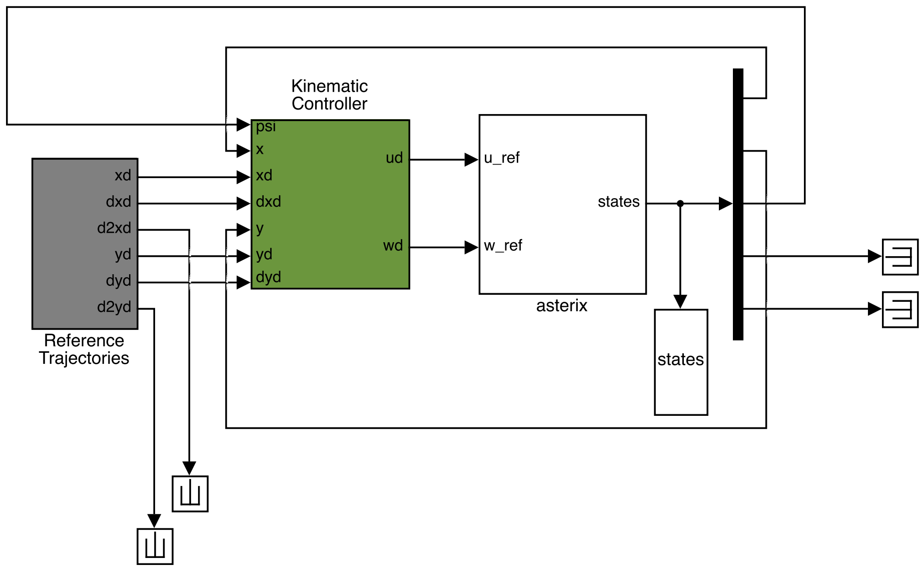

Figure 14: *Simulink diagram of the Asterix robot system with trajectory controller with simulink blocks from Martins (2015). Measurement noise was added with a script to the simulated data so that it could be made consistent for all the filters simulated.*

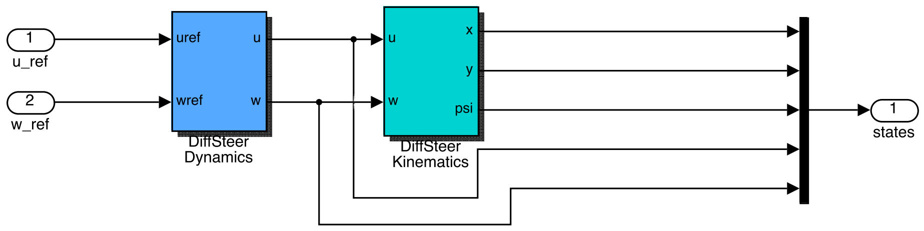

Figure 15: *Simulink diagram of Asterix subsystem in Fig.14 with simulink blocks from Martins (2015).*

3.2.2 Prototype

Adigo has developed a prototype of Asterix that was used to gather datasets. These include pictures and ROS bag files. ROS is a framework for robot software development with conventions on data representation and coordinate frames. The ROS bag files contain "message" data from the sensors in a standardized notation. ROS has an API for use with Python.

The practical specifications of the prototype are given in Tab.2. The prototype is equipped with an IMU sensor (CHR-UM6) capable of sensing orientation at a rate between 20 Hz and 300 Hz. It contains a magnetometer, MEMS gyro and accelerometer and uses these in conjunction with an internal EKF to produce orientation and angular velocities. There are two motorized wheels in the front equipped with wheel encoders that give the rotational speed of the wheels. The prototype also has an RTK-GPS (Septentrio Polarx 2e) which pro-

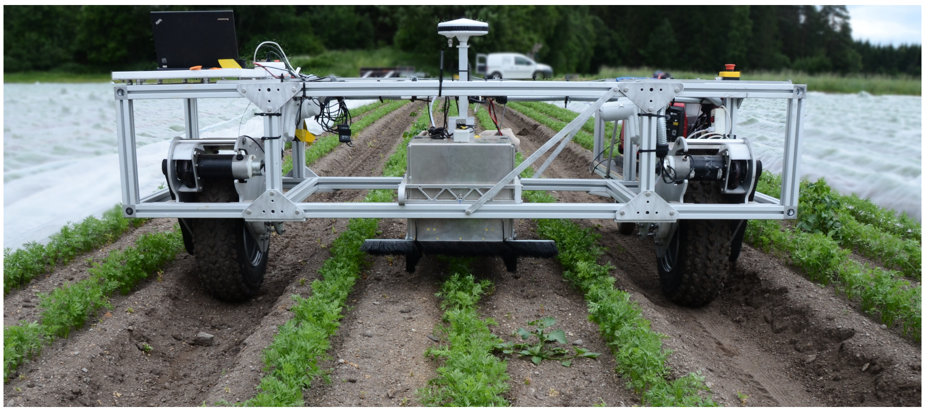

Figure 16: *Asterix prototype gathering data from a carrot field (Rygge 2014).*

vides position estimates at rates between 1 Hz and 20Hz. The camera (IDS USB 2.0 uEye LE) is a 5MPix color camera capable of transmitting at most 6.3 fps.

Description	Symbol	Value
Wheel base	b	$1.7[m]$
Wheel diameter	D	$0.533[m]$
Drive belt ratio	r_g	1/4.4
Solenoid gear ratio	r_c	1/20
Encoder speed to tangential wheel speed	r_e	$\pi D r_g r_c$

Table 2: *Specifications of the prototype*

3.2.3 Measurement model

The wheel encoders provide measurements of the rotational velocity of the wheels. By wheel odometry (WO) equations and using the state notation in (3.4), the measurement mapping between states and rotational velocity of the wheels is:

$$\begin{bmatrix} \omega_{right} \\ \omega_{left} \end{bmatrix} = \begin{bmatrix} (u + \frac{1}{2}b\omega)\frac{1}{r_e} \\ (u - \frac{1}{2}b\omega)\frac{1}{r_e} \end{bmatrix} \tag{3.7}$$

where $b = 1.7$m is the wheel spacing, $r_e = 0.00308$ is the encoder to tangential speed ratio. This was calculated as in Tab.2 but tuned based on the comparison between wheel odometry and visual odometry for dataset B where a tape measure was visible for a portion of the dataset.

The IMU provides measurements of the Euler angles, roll, pitch, and yaw. The CHR-UM6 provides these in the conventional BODY frame, which are converted internally by the ROS interface to ROS-BODY. The angular rates provided by the IMU are of the I frame with respect to the ECI frame expressed in

I ($\boldsymbol{\omega}_I^{I,ECI}$). If the I frame is aligned with the ROS-BODY frame, the mapping between IMU measurements and states is:

$$\begin{bmatrix} \psi_{IMU} \\ \omega_{IMU} \end{bmatrix} = \begin{bmatrix} \psi \\ \omega \end{bmatrix} \tag{3.8}$$

As stated in (2.26), visual odometry, calculated as a displacement between observed features has the following measurement mapping:

$$\begin{bmatrix} \Delta x_{VO} \\ \Delta y_{VO} \\ \Delta \psi_{VO} \end{bmatrix} = \begin{bmatrix} R(-\psi_k) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_m - x_k \\ y_m - y_k \\ \psi_m - \psi_k \end{bmatrix} \tag{3.9}$$

Where t_k is when the oldest picture was taken, and t_m is the newest picture. In the datasets, the pictures arrived approximately every 2.5 s. The processing delay has been added by making the VO measurement available some time after the latest picture has been taken.

Simulations of the Asterix robot and test on the available datasets are given in the Results chapter.

3.3 Manual Feature Software

Feature detection is a large research field outside the scope of this thesis. Designing an algorithm capable of finding acceptable features is a major undertaking worthy of its own thesis. Instead, for this project, the features were chosen manually from picture subsets of the two datasets.

3.3.1 OpenCV

OpenCV was used to process the images from the datasets. OpenCV is an open source computer vision platform with interfaces for C, C++, Python and Java. When using Python, images are stored as NumPy arrays. With NumPy, pixels are accessed in terms of row and column indices of the image with the origin set in the upper left corner. However, points are represented in OpenCV in terms of X and Y coordinates corresponding to column and row. These are the coordinate conventions used when drawing shapes. This is something to be mindful of when working on the actual data in the image arrays.

3.3.2 Software

In Fig.17 a set of features have been chosen, and their positions estimated by using the wheel odometry measurements between the two picture times. In Fig.18 the features' coordinates have been corrected by brute-force search.

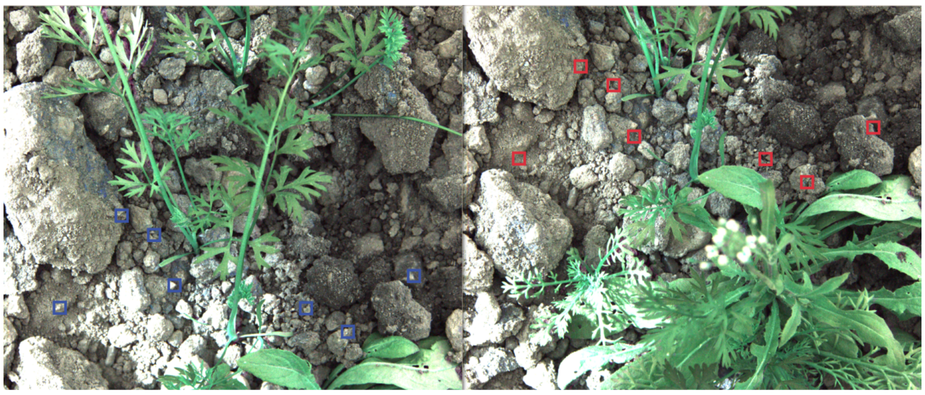

Figure 17: *Example use of the feature software on real images from a dataset before brute-force search correction. Left image is the newest, right image is the oldest. The wheel odometry has underestimated the distance traversed.*

The brute-force search takes the color values of a 60x60 pxs feature in the current image and compares it to all 60x60 pxs segments in the vicinity of where it is expected to be. The location which minimizes the $L2$ norm of the difference between the matrices is the corrected location. It was observed that using the full RGB pictures did not always yield the desired results. If a feature is too

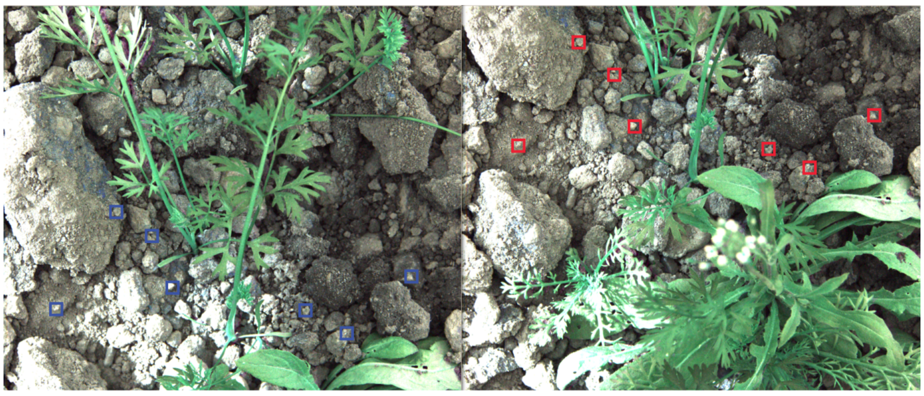

Figure 18: *Example use of the feature software on real images from a dataset. Left image is the newest, right image is the oldest. The brute-force search has corrected the underestimated translations.*

dark, there might not be enough information to differentiate from any other dark segment. The software was therefore expanded to include the ability to search in the intensity values only.

Applying two different gaussian blurs to a picture and subtracting is a method often used in image processing for edge detection. There was only one case found where this performed better than the grayscale intensity map. This was when a tape measure had been placed on the ground and the edges of the sharp, clean numbers were in the pictures.

It was also observed that the height difference in features can yield different displacement estimates. An example of a potentially problematic feature is the topmost feature of Fig.18. It is located on top of a rock of unknown height, that appears to be higher up than the other features. When choosing features, this was minimized by attempting to only choose features in approximately the same plane.

The brute-force search has a glaring flaw in the fact that it cannot test different rotations of the 60x60 pxs feature blocks. The datasets are of the robot moving down a row of crops and this was not a major issue. When rotation was apparent, the features were chosen to minimize the false positives. This was done by choosing features that were equivalent before and after rotation (e.g. small circular pebbles or holes).

3.4 Python Modules

The project was done using Python, running simulink models using the Matlab API, reading ROS bag files using the ROS API, and handling images using OpenCV. As this is a research project on fusing OOSM, Python was chosen as the main programming language. Python has pseudo-code like syntax, is object oriented, uses duck typing extensively, and can be debugged with an interactive shell. This means that one can quickly prototype and combine things when necessary, with little interfacing between modules and objects.

NumPy is a well established numerical library for Python that was used extensively in this project. It is optimized for array and matrix operations and has comparable efficiency to MATLAB. This is partially because both environments use LAPACK and BLAS for linear algebra computations.

For the ease of testing various systems and scenarios, a set of modules were created. A probabilisticFilter module was created with KF, EKF, UKF, and a particle filter based on Thrun (2002). The particle filter goes beyond the scope of this thesis and was not used during simulations. The modules were not designed with optimal resource usage in mind, but rather modularity and descriptiveness. The code used is given in appendix A.5. For completeness, an implementation of Horn's method and a module for conversion from elliptical coordinates to local cartesian ENU coordinates are also in the appendix.

3.4.1 Probabilistic Filter Module

The probabilisticFilter module contains classes BaseFilter, KalmanFilter, ExtendedKalmanFilter, UnscentedKalmanFilter and ParticleFilter. In Fig.19 the class diagram is shown, any parameter whose datatype is not specified is a NumPy matrix. All filters are based on the recursive bayesian filter and have a *predict* and an *update* function. *Iterate* is a shorthand that calls predict and then update. This module was created to be conceptually correct, with verbose naming and inheritance based on underlying assumptions rather than direct necessity. E.g. both EKF and UKF inherit from the KF class as the underlying assumptions are the same. For conceptual clarity it could have been better to create abstract "KalmanLike" and "ParticleLike" classes.

CreateMasked creates a filter class with a "wrapper" on the *update* function. This takes a predetermined mask_size which defines the section of state estimates that are to remain constant. This wrapper is general enough to handle any Kalman-like filter classes.

3.4.2 Delay Fusion Module

The main functionality of the delayFusion module is the *patchFilter* function. The *patchFilter* function takes a filter object and augments the object with three new functions: *clone*, *extend* and *marginalize*. Stochastic cloning requires the use of *clone* and *marginalize*. The joint method uses *extend* and *marginalize*. As outlined in the theory chapter, the extend function is different for EKF which uses the equations (2.49) and (2.51), and UKF which uses (2.50) and (2.52). *Clone* and *marginalize* are the same for both EKF and UKF if the transition function, transition Jacobian function and measurement function is designed to take an arbitrarily large augmented state vector $\boldsymbol{x}^a$ and only evaluating the transformation for the latest state. E.g:

```
def transition_function(states):
    states[:-STATE_SIZE] = A*states[:-STATE_SIZE]
    return states
```

In the theory chapter it was shown that the cloning procedure for an EKF will not require the use of the approximation noise associated with the regularization constant ϵ. In this module however, the *clone* function for an EKF has ϵ. This is to better understand how the ϵ in the SC method affects the results.

In Fig.20, a basic flowchart is given for both stochastic cloning and the joint method describing typical usage. When a picture is taken, stochastic cloning performs a clone step while the joint method waits until the next predict step and substitutes it with an extend step. When a VO measurement has arrived, the oldest cloned state is marginalized.

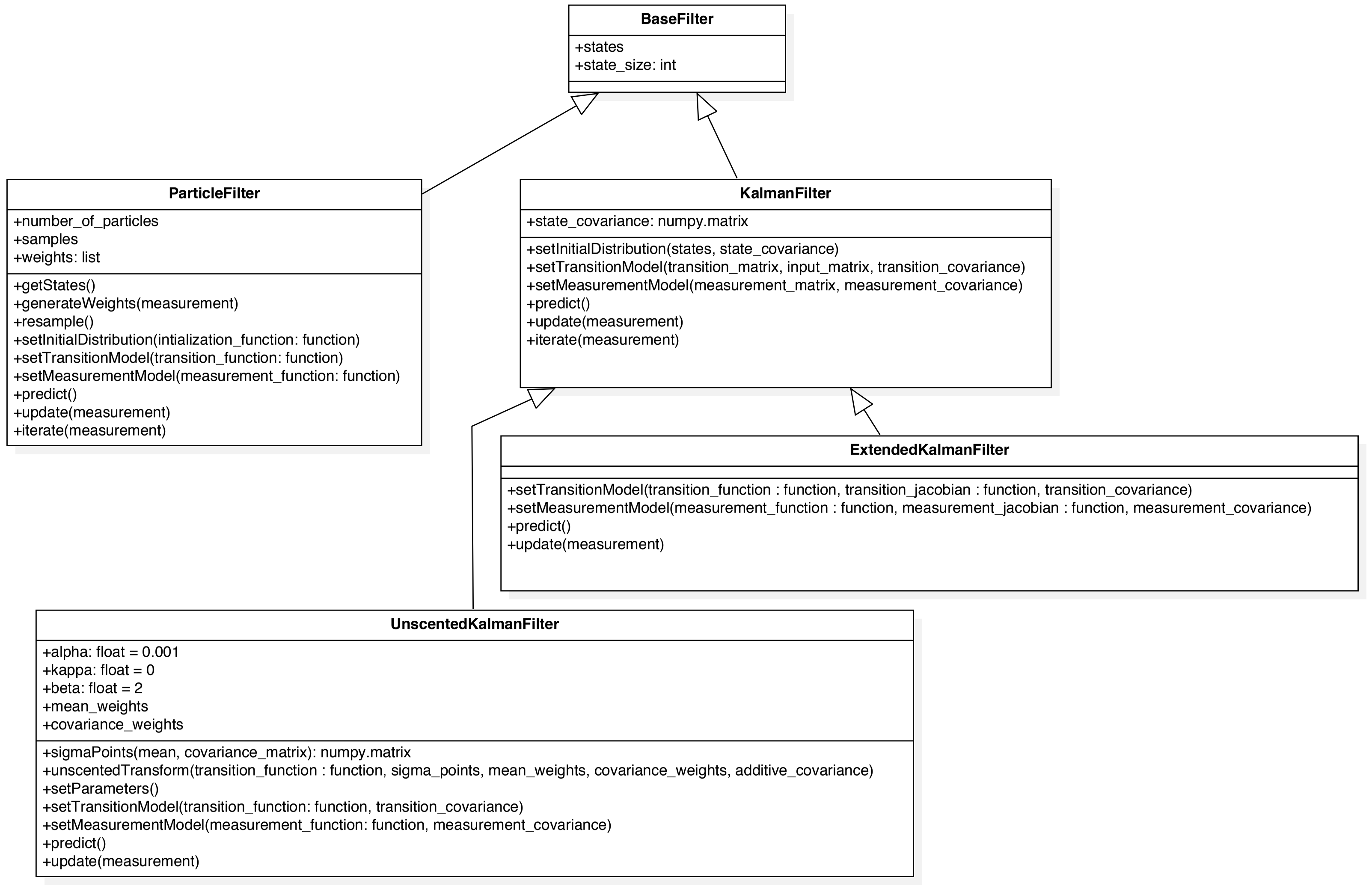
BaseFilter
+states
+state_size: int
ParticleFilter
+number_of_particles
+samples
+weights: list
+getStates()
+generateWeights(measurement)
+resample()
+setInitialDistribution(intialization_function: function)
+setTransitionModel(transition_function: function)
+setMeasurementModel(measurement_function: function)
+predict()
+update(measurement)
+iterate(measurement)
KalmanFilter
+state_covariance: numpy.matrix
+setInitialDistribution(states, state_covariance)
+setTransitionModel(transition_matrix, input_matrix, transition_covariance)
+setMeasurementModel(measurement_matrix, measurement_covariance)
+predict()
+update(measurement)
+iterate(measurement)
ExtendedKalmanFilter
+setTransitionModel(transition_function : function, transition_jacobian : function, transition_covariance)
+setMeasurementModel(measurement_function : function, measurement_jacobian : function, measurement_covariance)
+predict()
+update(measurement)
UnscentedKalmanFilter
+alpha: float = 0.001
+kappa: float = 0
+beta: float = 2
+mean_weights
+covariance_weights
+sigmaPoints(mean, covariance_matrix): numpy.matrix
+unscentedTransform(transition_function : function, sigma_points, mean_weights, covariance_weights, additive_covariance)
+setParameters()
+setTransitionModel(transition_function: function, transition_covariance)
+setMeasurementModel(measurement_function: function, measurement_covariance)
+predict()
+update(measurement)

Figure 19: *Class diagram for the probabilisticFilter module. If not specified, functions return void and parameters are NumPy matrices.*

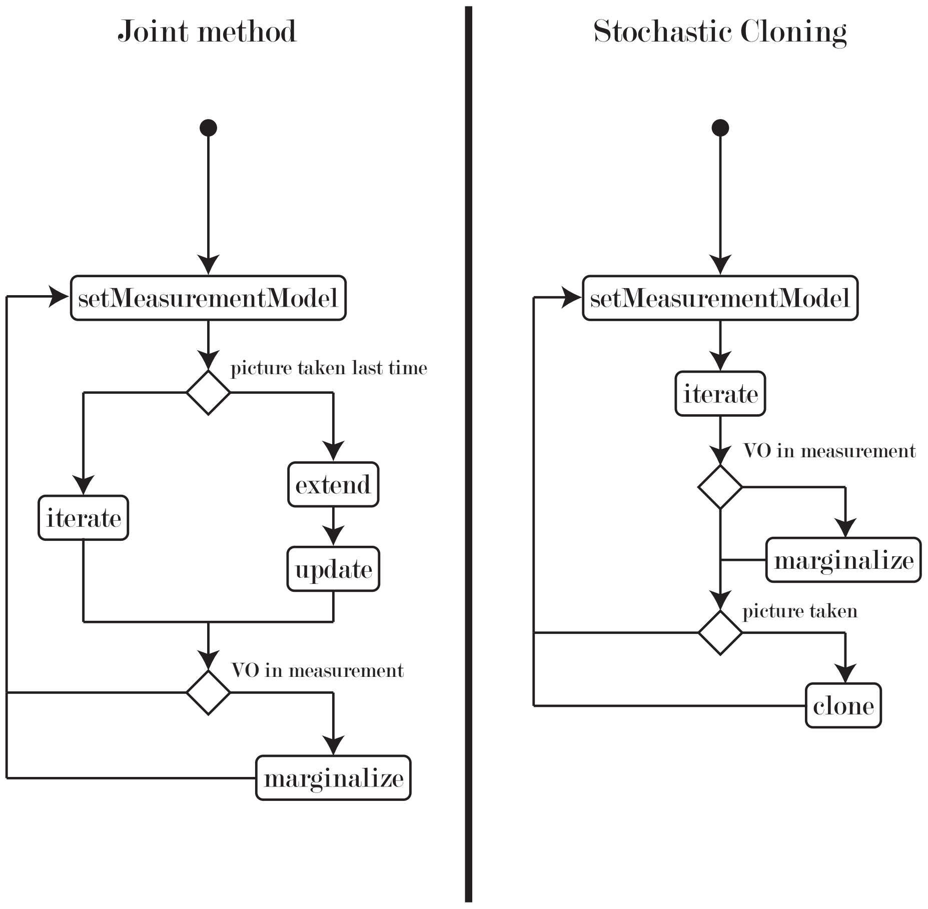

Figure 20: *Typical usage of a filter augmented for delay fusion.*

3.4.3 Kinematic Module

The kinematic module was created to simplify the initialization of filter objects. The main function in this module is *patchFilter*. It takes a filter object as instantiated from the probabilisticFilter module and patches the filter with the transition and transition Jacobian functions necessary for the filter to follow the modified model of De La Cruz. The base differential function is as described by (3.4). The Jacobian was found analytically and verified with SymPy. The discrete filter transition function and transition Jacobian is evaluated by using forward euler on the underlying differential equation with a timestep specified in the filter. This method of implementing a transition function is conceptually clearer and allows one to change the underlying model if the chosen one does not fit, but has an unnecessary amount of function calls and memory allocation. The actual system should be implemented with only the required functions specified from the beginning.

3.4.4 Measurement Handler Module

When accessing topics of ROS bag files in Python, the built-in method of the ROS API returns a generator object that can give an iterator for a sequence of messages between two rostime instances. To keep the modularity chosen for the research of this project, there is no reason for the filter module to know of ROS bags or other ROS types. The measurementHandler module was designed to simplify the construction of the measurement function, measurement Jacobian function, and measurement vector. The measurementHandler is an interface to the real datasets.

Everything in Python is an object in the pythonic sense, the only difference between an object and a function is that it has an internal __*call*__ function. This means that one can construct a class whose instances are callable: a function generating class. This simplifies the process of making measurement functions on demand that only handle the sensors that have currently arrived. The function behavior can be instantiated with different functonality as needed. The internal variables of our function generating objects can be thought of as static variables in a C function.

The measurementHandler module uses many tricks that are relatively unique to Python, e.g. accessing local variables with strings to dynamically create the measurement function. This is the least descriptive module used in this project, but it was very important in designing proper control flows for the system and filters without getting lost in the measurements arrival order and availability.

4 Results

The object-oriented implementations allow us to create a variety of filters. The prefixes describing the functionality of a specific filter is given in Tab.3, e.g. MSCEKF denotes a masked EKF that uses stochastic cloning for delay fusion.

Prefix	Description
SC*	Indicates the filter fuses VO with stochastic cloning
J*	Indicates the filter fuses VO with the joint method
M*	Indicates the lagged states are not smoothed

Table 3: *Filter abbreviations used in the simulations. If no fusion method is specified, VO measurements are ignored.*

4.1 Module Testing

This section shows simulations of the target tracking system (3.1). The intention of these simulations was to test the probabilisticFilters and delayFusion modules. Description of the parameters of the system are recreated from the implementation description, and given in Tab.4.

Symbol	Description
R_k	Measurement noise covariance
T	Sampling interval or timestep
q	Process noise
M	Frequency of measurement z_k^d (in iterations)
N	Delay of measurement z_k^d (in iterations)
α	Comparative accuracy of delayed measurements (lower is better)

Table 4: *Parameters for the delay testing based on (3.1).*

4.1.1 Target Tracking

The simulation parameters are given in Tab.5. It was observed that the EKF implementations were less numerically stable than the UKF implementations. The state covariance matrix did not remain symmetric. Matlab performs a robust matrix “division” based on tests of positive-definiteness and other characteristics. If all of these tests fail, and the matrix is square, Matlab uses the LU solver from LAPACK. Numpy does not perform these tests and uses the LU solver directly. This leads to numerical errors in the off-diagonal elements in Python implementations of the filters. To compensate the numerical trick $P = 0.5(P + P^T)$ was added at the end when updating state covariance for all filters in the probabilisticFilters module.

Fig.21 shows the filter tracking the target. BASEKF is a Kalman filter where the delayed measurement z_k^d arrives as though there was no delay. KF is a Kalman filter that ignores the delayed measurement. The difference in behavior of the delay fusion strategies are indiscernible in their graphs. The masked versions are not depicted to reduce clutter in the graph. In Tab.6 a description of the filters is given with the root-mean-square (RMS) error of the position estimate. For these simulations ϵ was set to 10^{-10}. Lower values did not ensure positive-definiteness consistently for all system scenarios tested.

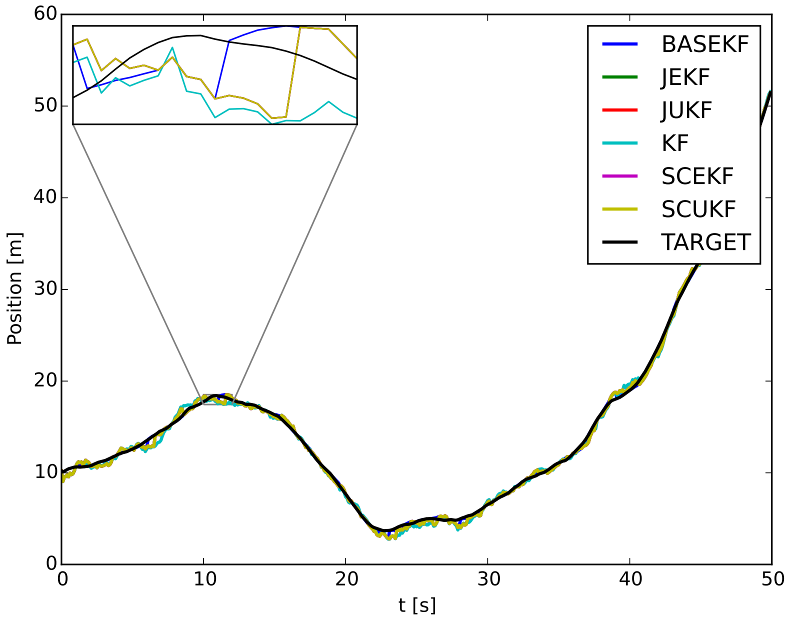

Figure 21: *Basic target tracking according to the system (3.1), with parameters in Tab.5. The filters are described in Tab.6. The zoomed inset shows BASEKF fusing z_k^d (Blue line), then 5 iterations later the other filters fuse the measurement ("Brown" line). KF does not fuse z_k^d (Cyan line). The masked versions of the filters are not shown to reduce the clutter in the graph.*

Parameter	Value
R_k	1
T	0.1
q	0.5
M	10
N	5
α	0.001

Table 5: *Parameters used for the basic tracking simulation.*

Name	RMS
BASEKF	0.25784926
SCUKF	0.37536253
JUKF	0.37536699
SCEKF	0.37536253
JEKF	0.37536699
MSCUKF	0.43339973
MJUKF	0.43340372
MSCEKF	0.43340173
MJEKF	0.43340572
KF	0.45773917

Table 6: *RMS of position estimate. BASEKF is a Kalman filter where z_k^d arrives without delay. KF is a Kalman filter that ignores z_k^d.*

4.1.2 Discussion: Basic Tracking

The RMS values in Tab.6 show that the smoothing filters have better estimates than the masked filters. Furthermore, the masked filters have better estimates than the KF. The EKF and UKF performs similarly, which is not surprising as we are handling a linear system.

The RMS values in Tab.6 suggest that the SC method was better than the joint method. This was not a consistent result. With the same system and tuning parameters, but different random seeds for the noise, this changed. Increasing the simulation time made the difference decrease. With a simulation time of 500 iterations the difference was, on average, in the order of 10^{-6}. With a simulation time of 20000 iterations the difference was, on average, in the order of 10^{-8}. For long enough simulations, SCUKF was consistently better than JUKF, and JEKF was consistently better than SCEKF. This suggests that the numerical issues of performing the higher number of Cholesky factorizations needed for JUKF induced larger errors than the regularization constant ϵ induced in SCUKF.

With large enough delay or process noise, there were cases observed where ignoring the lagged measurements was better than fusing them incorrectly.

4.2 Asterix Simulation

This section shows simulations of the Asterix system in Fig.14 with the intention of investigating the SC method and the joint method for robot localization. The parameters of the filters and the system are given in Tab.7. The trajectory planner from Martins (2015) has 4 reference trajectories: straight line, circle, figure eight, and moving to a fixed point. The simulations used wheel odometry (WO), linear and angular velocity, instead of direct wheel speeds, left wheel speed and right wheel speed, as these were readily available from the simulation.

For each of the simulations we have the opportunity to run with any permutation of the sensors: WO, IMU, GPS and VO. The sensors are assumed to be affected by additive gaussian noise. The noise covariance matrices are described in Tab.7.

Symbol	Description
Q	Process noise (from perspective of the filters)
R_{GPS}	Noise covariance of GPS measurements
R_{VO}	Noise covariance of VO measurements
R_{WO}	Noise covariance of WO measurements
R_{IMU}	Noise covariance of IMU measurements
T	Sampling interval or Timestep
P_0	Initial state covariance
$\boldsymbol{x}_0$	Initial states
M	Frequency of VO measurements (iterations)
N	Delay of VO measurements (iterations)

Table 7: *Parameters for the Asterix simulink model.*

Parameter	Value
T	0.05 s
Q	$diag([10^{-9}, 10^{-9}, 10^{-9}, 5 \cdot 10^{-7}, 5 \cdot 10^{-7}])$
R_{WO}	$diag([10^{-2}, 10^{-2}])$
R_{IMU}	$diag([5 \cdot 10^{-2}, 5 \cdot 10^{-2}$
R_{GPS}	$diag([5 \cdot 10^{-1}, 5 \cdot 10^{-1}])$
R_{VO}	$diag([10^{-6}, 10^{-6}, 10^{-5}])$

Table 8: *Tuned parameters used with filters.*

4.2.1 Localization Without GPS

This section shows simulations of the filters following the robot with VO frequency $M = 60$ and VO delay $N = 20$. Equivalent to 0.33 fps and 1 second processing time. Significantly slower than what is expected of the real system. With this timing and tuning, the delay fusing EKFs became unstable and are not shown in these figures. The two UKFS using stochastic cloning and the joint method are practically indistinguishable in the figures.

In Fig.22 the robot turns into a straight path. The pose with respect to time is given in Fig.23. The velocities are given in Fig.24.

In Fig.25 the robot takes the first picture when it has arrived on the straight path. The pose with respect to time is given in Fig.26. The velocities are given in Fig.27.

In Fig.28 the robot moves in a figure eight. The pose with respect to time is given in Fig.29. The velocities are given in Fig.30. The RMS errors are shown in Tab.10.

In Fig.31 the robot moves in a circle. The trajectory controller used requires the robot to move in a straight line out to the perimeter of its circle before moving in the circle. It causes small spikes in the angular rate when it passes the point the robot started moving in a circle. To compensate for the sharp turn when moving to the perimeter, the first image was taken after the robot arrived on the circle. For this simulation the process noise corresponding with the linear velocity and angular velocity was increased by a factor of 10. This was done to ensure that the estimation does not drift during the sharp turn caused by the trajectory controller. This drift would result in our estimates drawing a circle a distance away from the actual. The pose with respect to time is given in Fig.32. The velocities are given in Fig.33. The RMS errors are shown in Tab.11.

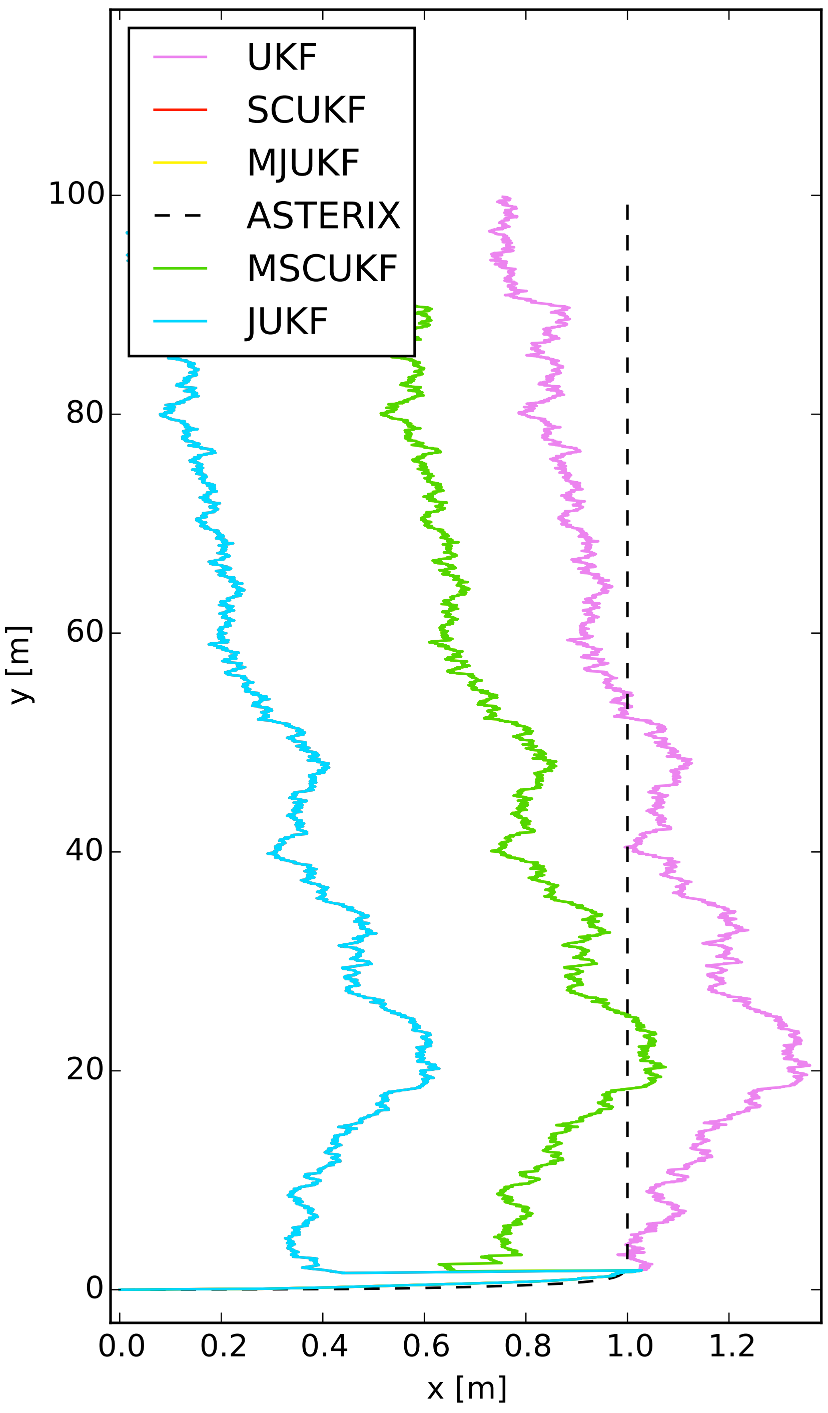

Figure 22: *Simulation of Asterix performing a sharp turn into a straight path. In xy-coordinates. Note that the x-axis is of a significantly smaller scale than the y-axis to emphasize the error induced by the sharp turn.*

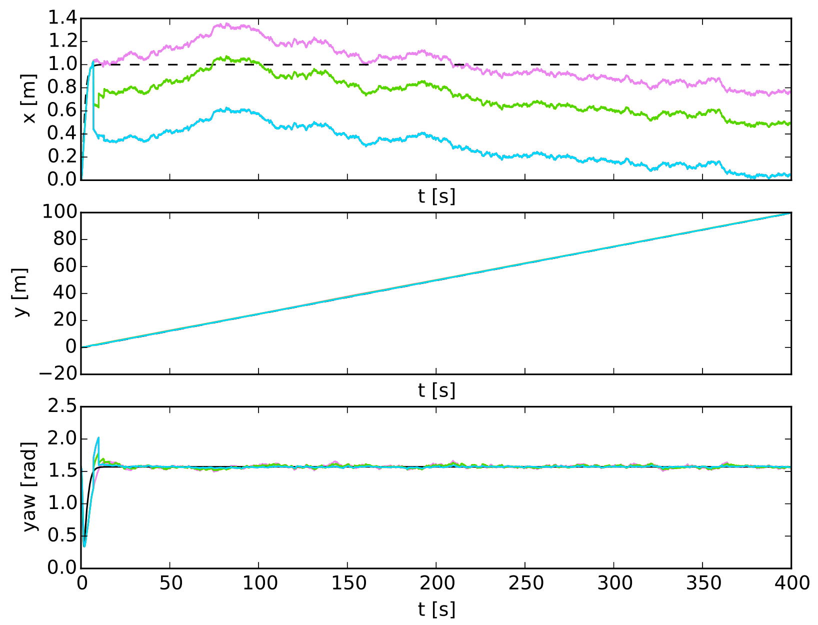

Figure 23: *Simulation of Asterix performing a sharp turn into a straight path. Pose with respect to time.*

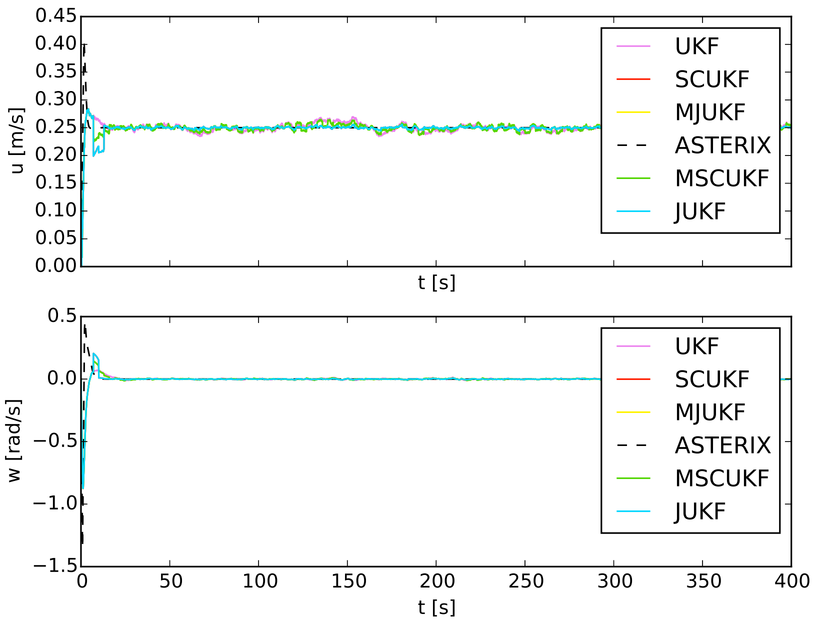

Figure 24: *Simulation of Asterix performing a sharp turn into a straight path. Velocities with respect to time.*

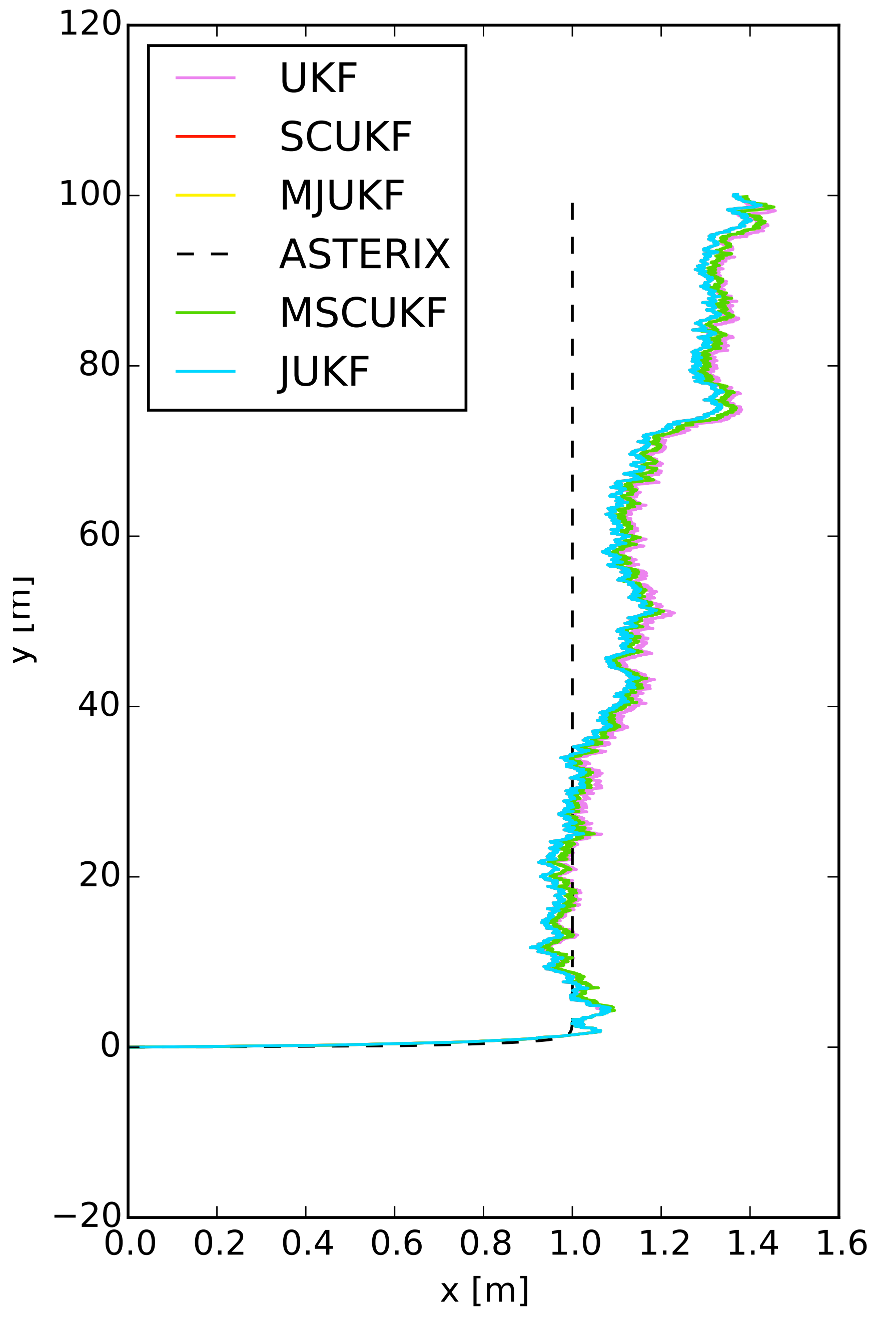

Figure 25: *Simulation of Asterix turning to move into a straight path where the pictures for VO measurements first arrive after the sharp turn. In xy-coordinates.*

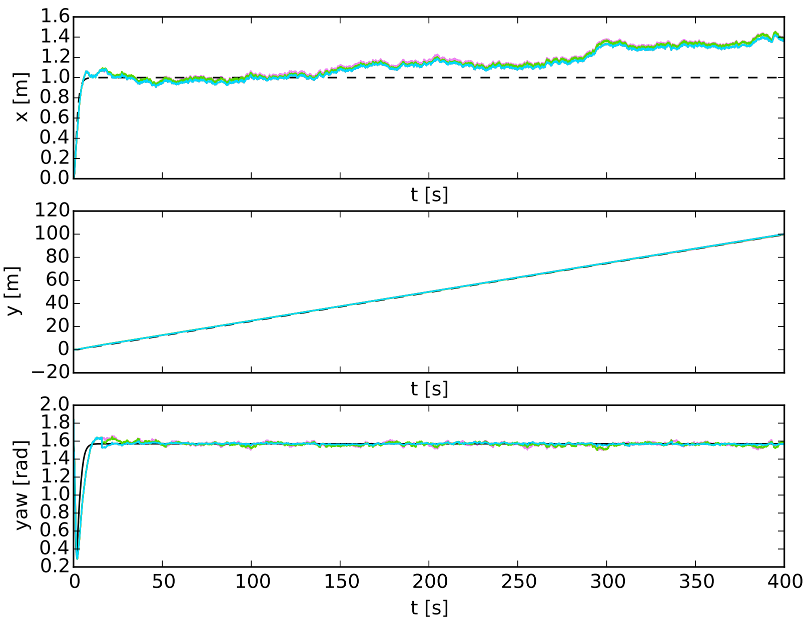

Figure 26: *Simulation of Asterix turning to move into a straight path where the pictures for VO measurements first arrive after the sharp turn. Pose with respect to time.*

	x	y	ψ	u	ω
UKF	0.36700297	0.09870972	0.02209709	0.00592651	0.0035072
SCUKF	0.3351425	0.48283523	0.00730707	0.00188329	0.00174407
JUKF	0.33514391	0.48283339	0.00730673	0.00188331	0.00174407
MSCUKF	0.36055355	0.27727286	0.02043303	0.00474811	0.00312327
MJUKF	0.36055471	0.27727006	0.020433	0.00474807	0.00312326

Table 9: *RMS errors from Asterix moving into a straight path and VO taken after arriving on the straight line. Based on state estimates after arriving on the straight path.*

	x	y	ψ	u	ω
UKF	1.61529791	0.7773032	0.08627775	0.09775693	0.05800764
SCUKF	1.12839804	0.73753893	0.07100451	0.06256853	0.05423402
JUKF	1.12846113	0.73756414	0.07100509	0.06256849	0.05423403
MSCUKF	1.34300359	0.6057546	0.0653399	0.09725445	0.05541977
MJUKF	1.34300564	0.60575607	0.06533995	0.09725453	0.05541978

Table 10: *RMS errors from Asterix moving a figure eight.*

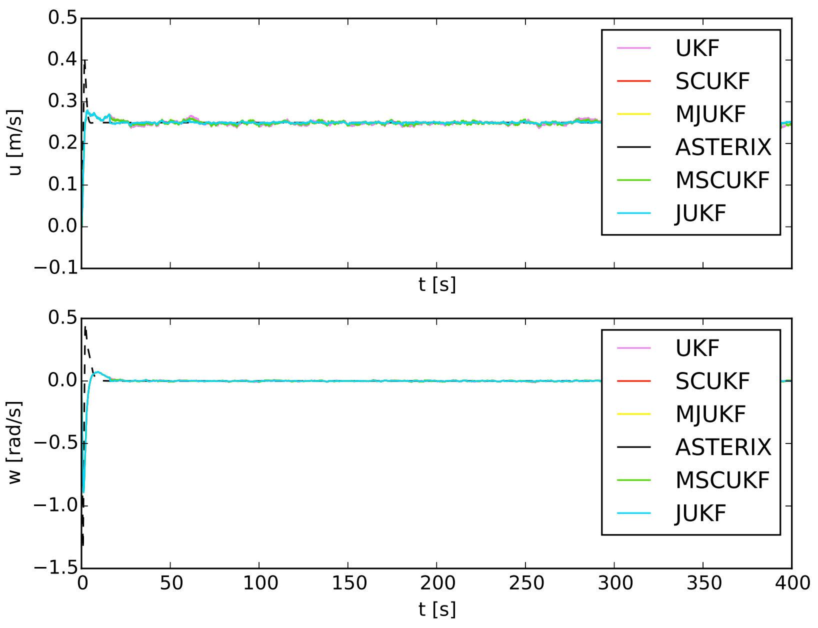

Figure 27: *Simulation of Asterix turning to move into a straight path where the pictures for VO measurements first arrive after the sharp turn. Velocities with respect to time.*

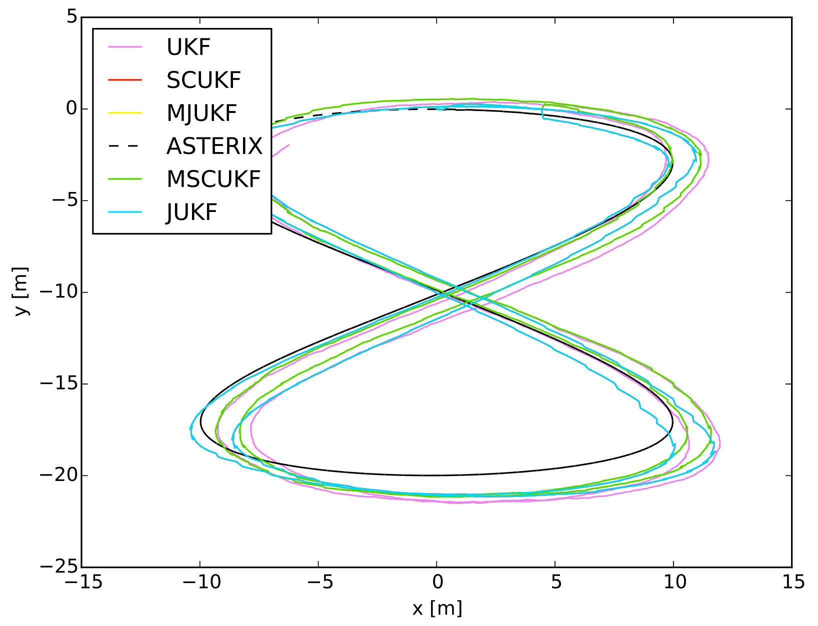

Figure 28: *Simulation of Asterix moving in a figure eight. In xy-coordinates.*

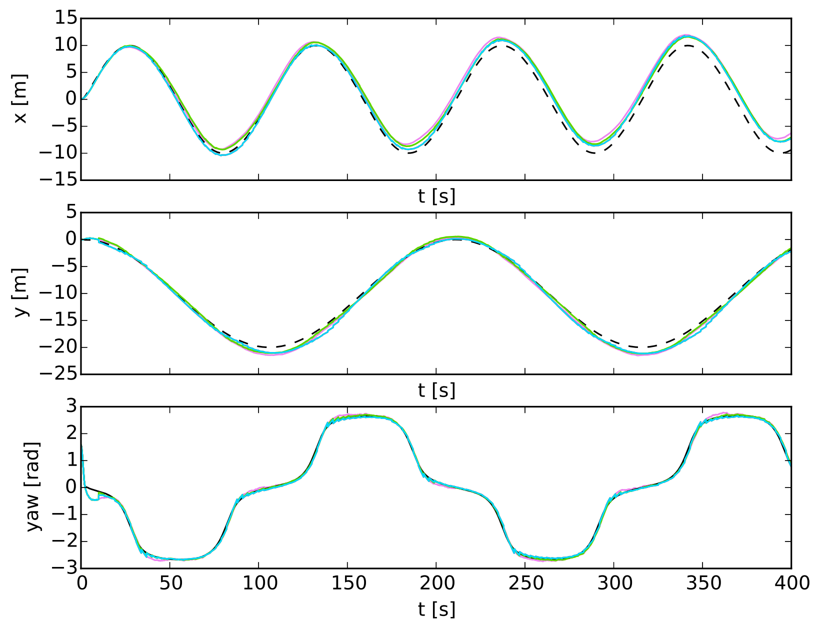

Figure 29: *Simulation of Asterix moving in a figure eight. Pose with respect to time.*

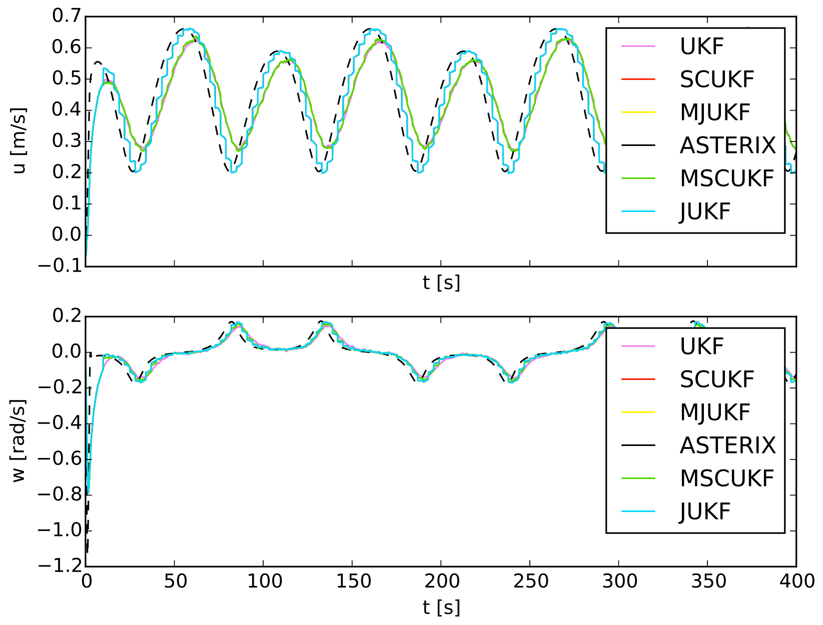

Figure 30: *Simulation of Asterix moving in a figure eight. Velocities with respect to time.*

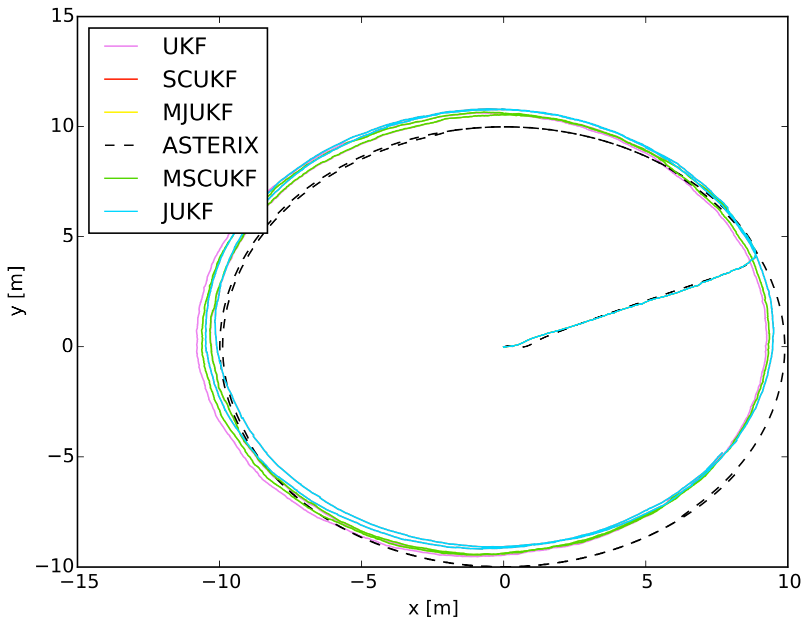

Figure 31: *Simulation of Asterix moving in a circle. In xy-coordinates.*

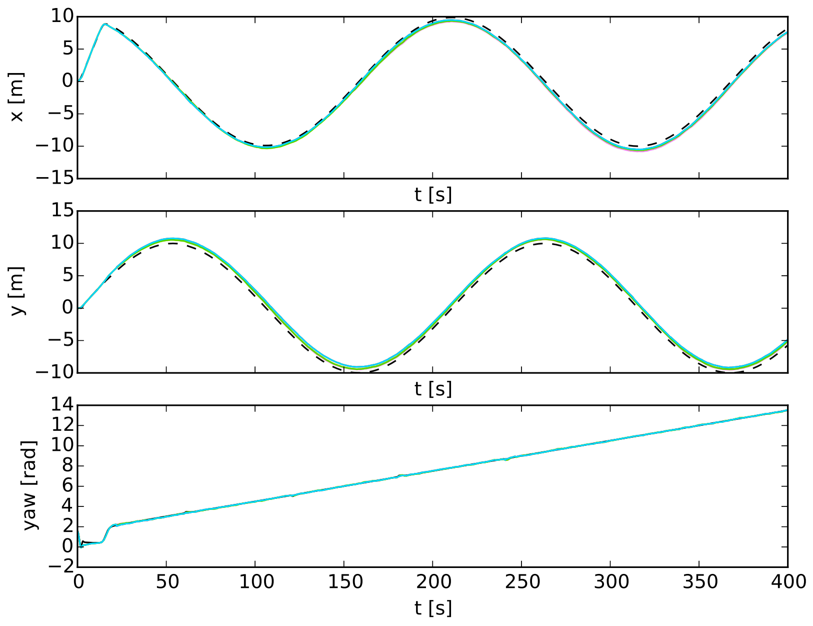

Figure 32: *Simulation of Asterix moving in a circle. Pose with respect to time.*

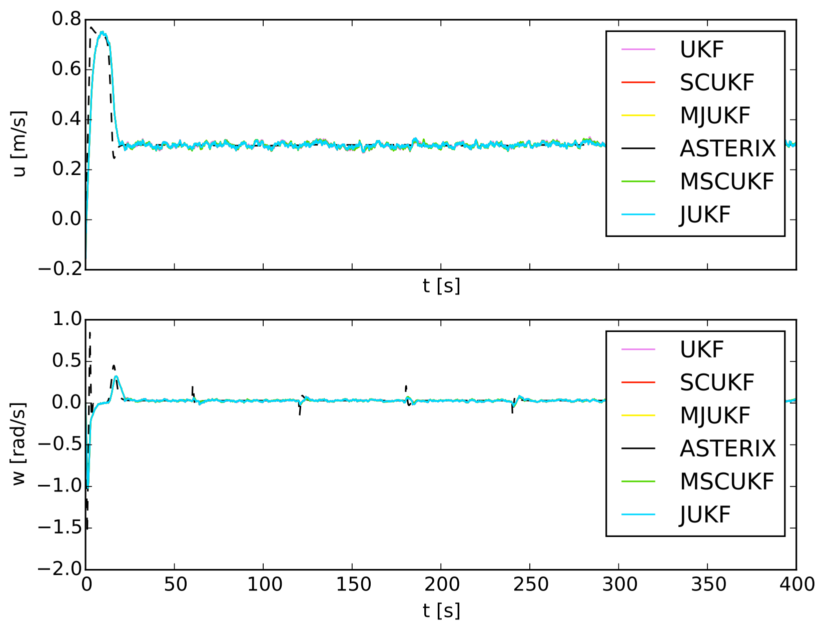

__Figure 33:__ Simulation of Asterix moving in a circle. Velocities with respect to time.

	x	y	ψ	u	ω
UKF	0.58564466	0.50873096	0.04312988	0.03452777	0.06666587
SCUKF	0.38142596	0.82124117	0.03975624	0.03396135	0.06651956
JUKF	0.38142293	0.82123973	0.03975627	0.03396135	0.06651956
MSCUKF	0.49389481	0.5538976	0.04152201	0.03434701	0.06665945
MJUKF	0.49389486	0.55389762	0.04152201	0.03434701	0.06665945

__Table 11:__ RMS errors from Asterix moving in a circle. First measurement taken after arriving on the circle.

4.2.2 Discussion: Localization Without GPS

Stochastic cloning and the joint method are practically equal in performance when moving in a straight path. When turning, the masked versions outperform the smoothing filters. During straight motion, however, the heading estiamtes are significantly better for the smoothing filters than for the others. In general the heading and velocities were better estimated when using VO than without, for all trajectories available. This is indicative of improved local accuracy.

In Fig.34, the first smoothed states of JUKF are shown for a simulation where Asterix turns into the straight path with pictures taken from the start. We see that neither x, y, nor ψ is correctly smoothed during the entirety of the turn. Investigating the state covariance corresponding to the smoothed states x, y, and ψ revealed that the masked filters had approximately the same covariance associated with x and y as the smoothing filters. The covariance associated with heading ψ in the smoothing filters was less than half of the covariance in the masking filters. This suggests that the smoothing filter has assumed it is more certain of the previous heading than it should have for optimal performance. This has resulted in the filter falsely fusing the VO measurement from the fast turn.

If the state covariance has converged too far and we have very low process noise, the filter will see a rapid change in states as highly unlikely and almost disregard the measurement. This sort of behavior can be seen affecting the linear velocity estimations of JUKF and SCUKF in Fig.30, making the WO measurements practically discarded while the velocity is only updated by the VO measurement.

This suggests a potential trick for improving the following during turns. By setting a lower threshold on the pose covariances, the VO measurements will not reduce the current state covariance to the same degree. This is a practical trick used to combat model inconsistencies which is present in many localization tasks. Others have approached these model inconsistencies via nonlinear optimization (Brekke and Chitre, 2014), or by reducing the number of model approximations used as they may cause a false information gain (Huang et al., 2010). In the investigations, the main culprit in turning issues was the state covariance associated with yaw, therefore placing a lower threshold on the smoothed yaw produced better results. This is slightly different from increasing additive measurement covariance of VO.

In Fig.35, a simulation of Asterix performing a figure eight with covariance threshold is given. It can be seen that after converging, the JUKF and SCUKF with covariance threshold have significantly less drift than the masked filters or the non-VO UKF. We see this from the path of UKF, MJUKF and MSCUKF diverging from its previous estimate.

The same can be seen when moving in a circle, Fig.36. Without the VO the filter has diverged more than with the VO. There is also a marginal improvement in the smoothing filters with covariance threshold over the masked filters.

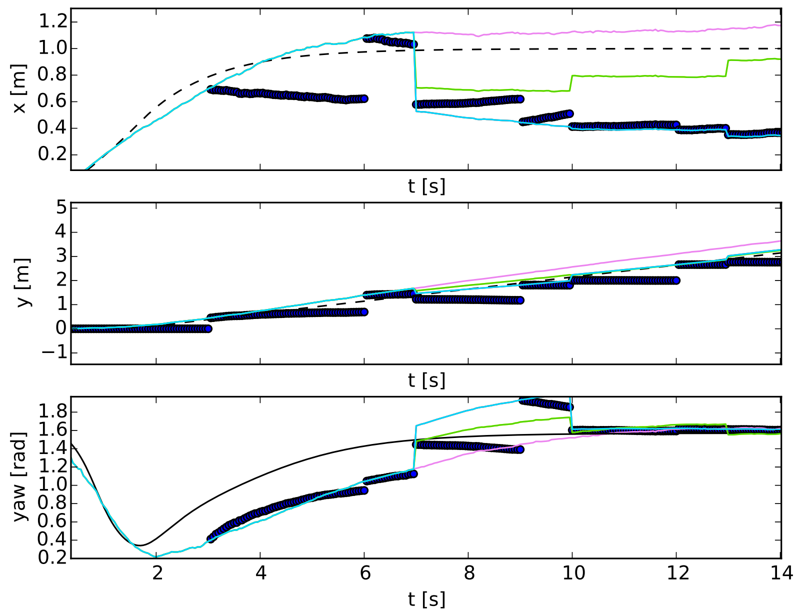

Figure 34: *Simulation of Asterix moving into a straight path, with focus on the initial turn. The thick dark lines are estimates of the first smoothed state of JUKF.*

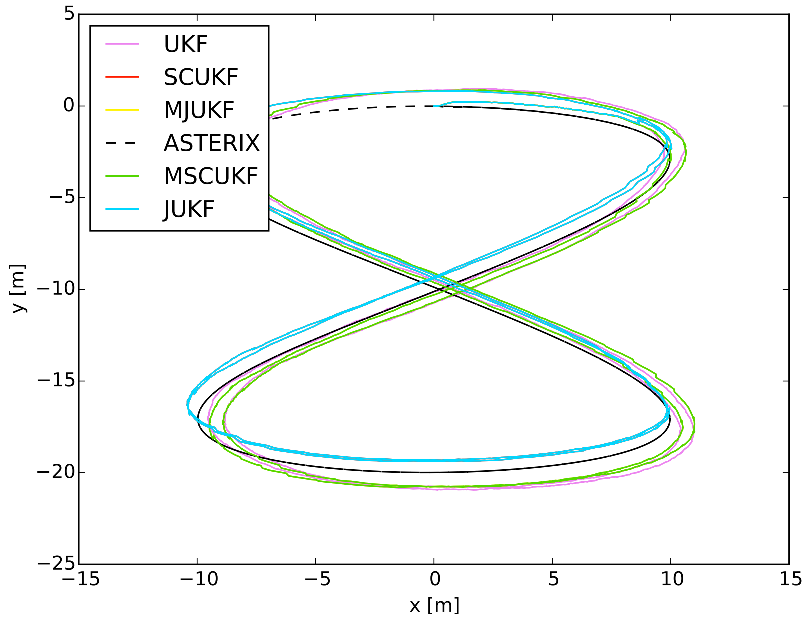

Figure 35: *Simulation of Asterix moving in a figure eight with covariance threshold. In xy-coordinates.*

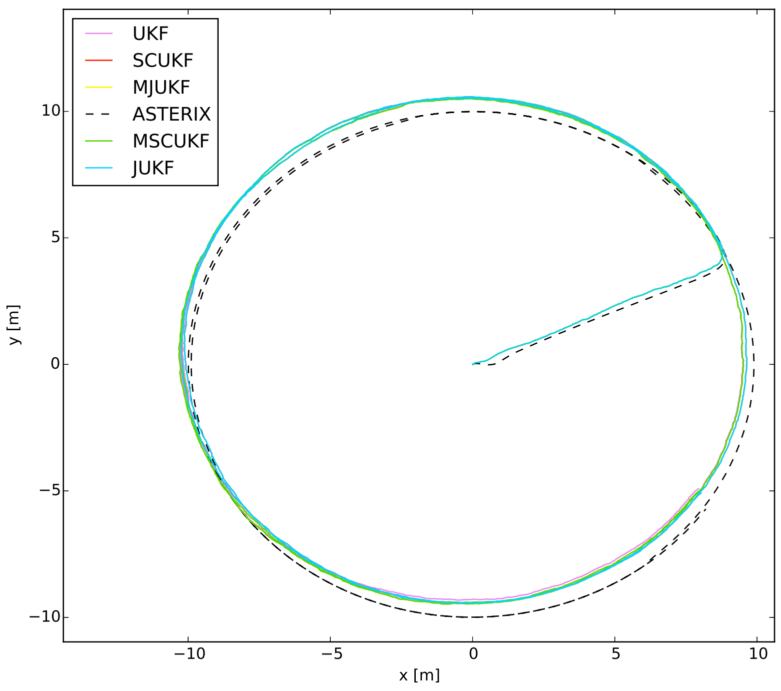

Figure 36: *Simulation of Asterix moving in a circle with covariance threshold. In xy-coordinates.*

4.2.3 Localization With GPS

With the GPS, the pose and velocities match quite accurately for all the filters. Therefore, we only look at the RMS errors, and not the pose and velocity values themselves.

In Fig.37 Asterix turns into a straight path. RMS errors are given in Tab.12.

In Fig.38 the robot moves in a figure eight. RMS errors are given in Tab.13.

In Fig.39 the robot moves in a circle. RMS errors are given in Tab.14.

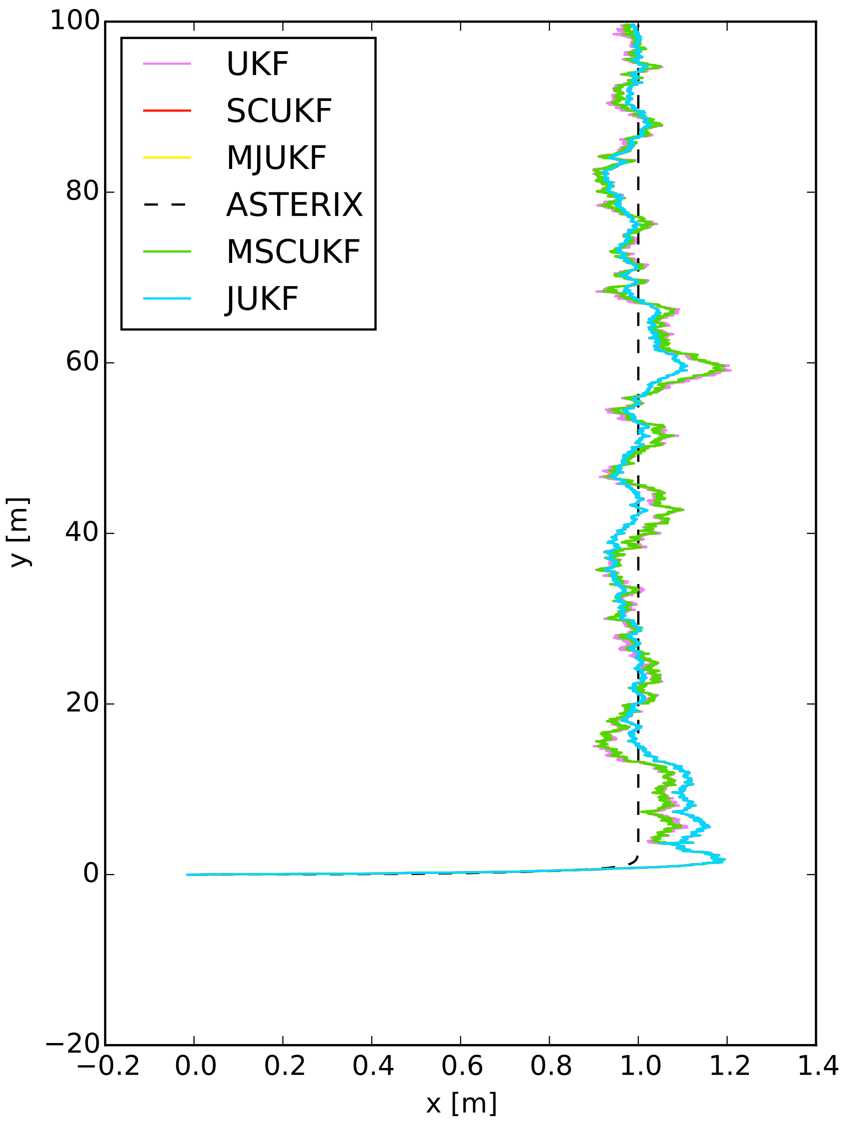

Figure 37: *Simulation of Asterix moving into a straight line with GPS enabled. In xy-coordinates.*

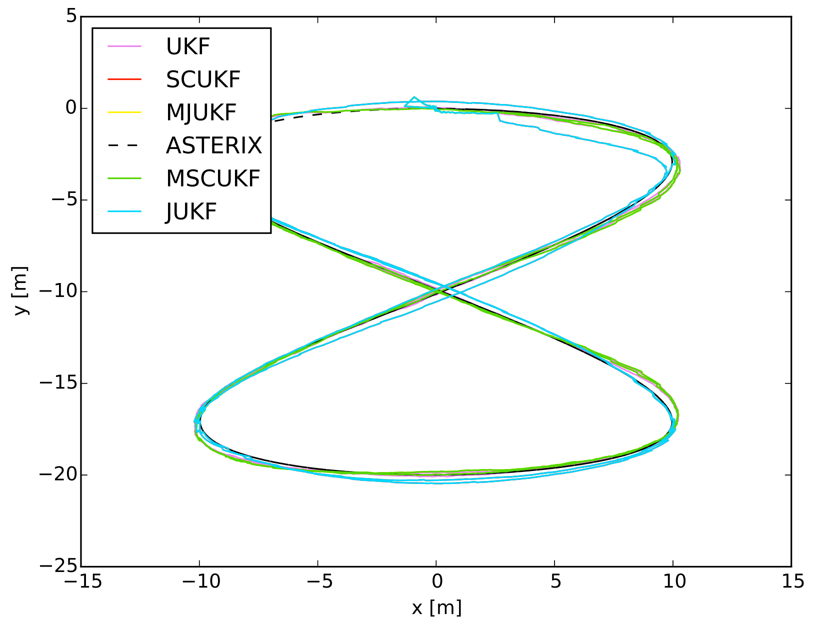

Figure 38: *Simulation of Asterix moving in a figure eight with GPS enabled. In xy-coordinates.*

	x	y	ψ	u	ω
UKF	0.05791371	0.05651774	0.05615649	0.01280591	0.06193944
SCUKF	0.0533361	0.02332885	0.05156995	0.01198779	0.0618413
JUKF	0.05333616	0.02332892	0.05156996	0.01198779	0.0618413
MSCUKF	0.05575388	0.04412132	0.05417581	0.0124805	0.06190236
MJUKF	0.05575388	0.04412123	0.05417581	0.0124805	0.06190236

Table 12: *RMS errors from Asterix moving into a straight path with GPS. First VO measurement taken after arriving on the straight path.*

	x	y	ψ	u	ω
UKF	0.26660216	0.2407436	0.09886145	0.10119604	0.05892125
SCUKF	0.14126572	0.22154109	0.07746124	0.06177537	0.05518257
JUKF	0.14126897	0.22154985	0.07746164	0.06177533	0.05518258
MSCUKF	0.31016586	0.2533244	0.08682799	0.09597943	0.05706502
MJUKF	0.31016636	0.25332488	0.08682808	0.09597952	0.05706503

Table 13: *RMS errors from Asterix moving in a figure eight with GPS.*

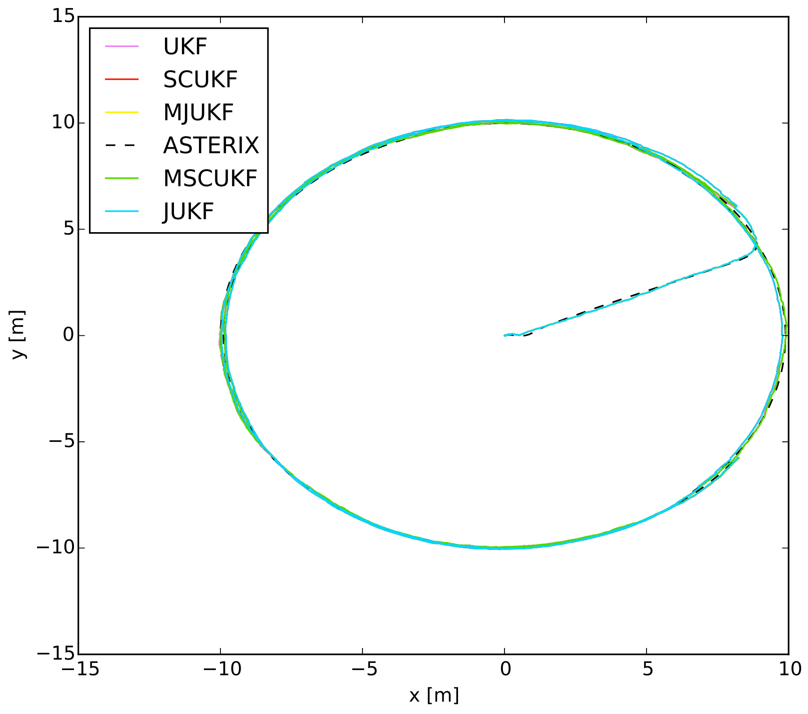

Figure 39: *Simulation of Asterix moving in a circle with GPS enabled. In xy-coordinates.*

	x	y	ψ	u	ω
UKF	0.08100364	0.16768807	0.07793271	0.04701396	0.07449521
SCUKF	0.53171966	0.4838474	0.16914433	0.04668859	0.07667171
JUKF	0.53173145	0.48386166	0.16914696	0.04668868	0.07667173
MSCUKF	0.08743931	0.14491779	0.07391027	0.0460379	0.07468705
MJUKF	0.08743941	0.14491795	0.07391034	0.04603792	0.07468706

Table 14: *RMS errors from Asterix moving in a circle with GPS. First VO measurement taken after arriving on the circle.*

	x	y	ψ	u	ω
UKF	0.0491472	0.0695823	0.04458652	0.03388542	0.06559486
SCUKF	0.0344218	0.06573378	0.03964613	0.03336424	0.06541137
JUKF	0.0344218	0.06573377	0.03964614	0.03336424	0.06541137
MSCUKF	0.04890236	0.06984123	0.04208915	0.0337513	0.06555642
MJUKF	0.04890236	0.06984123	0.04208915	0.0337513	0.06555642

Table 15: *RMS errors from Asterix moving in a circle with GPS and yaw covariance threshold. Threshold trick was only used on JUKF and SCUKF, masked filter remain unchanged. First VO measurement taken after arriving on the circle.*

4.2.4 Discussion: Localization With GPS

Under unmodeled motion, the masking filters perform better than the smoothing filters. This can be seen in the simulations with the circle trajectory. With the straight line and figure eight trajectories the smoothing filters perform better than the masked filters. As seen in Tab.15, a smoothing filter with yaw covariance threshold performs better than a masked filter without yaw covariance threshold.

Including the GPS does not manage to remove the error during the fast turn in the beginning of the circle trajectory and the straight path trajectory. This is partially a result of the tuning of the process noise. With the low process noise needed for accurate pose estimate while moving in a straight path, the filter will not be able to track the states during the fast turn. The VO measurements will show a very different velocity than what the WO measurements will. However, when using GPS, the filters are able to correct the error over time.

4.2.5 Effect of VO Frequency

In this section, the delay from VO processing remains a constant 500 ms and the camera's fps is changed. The sensors enabled are WO, IMU and VO. The figure eight trajectory was chosen for this as it varies in both x, y and ψ. The system was simulated with M ={8,16,32,64,128,256}. This corresponds to a camera with seconds delay between pictures of $\{0.4, 0.8, 1.6, 3.2, 6.4, 12.8\}$.

Note that the filtering strategies are capable of handling a delay greater than the VO frequency.

In Fig.40 trajectories for the different simulations are given side by side. The VO fusing UKF filters are stable for all simulations. For VO frequencies $M > 64$ the filters perform significant jumps resulting from the corrections associated with the VO measurements.

Note that the noise affecting each simulation is from a different random seed so behavior is captured, but comparing RMS values is not sufficiently representative.

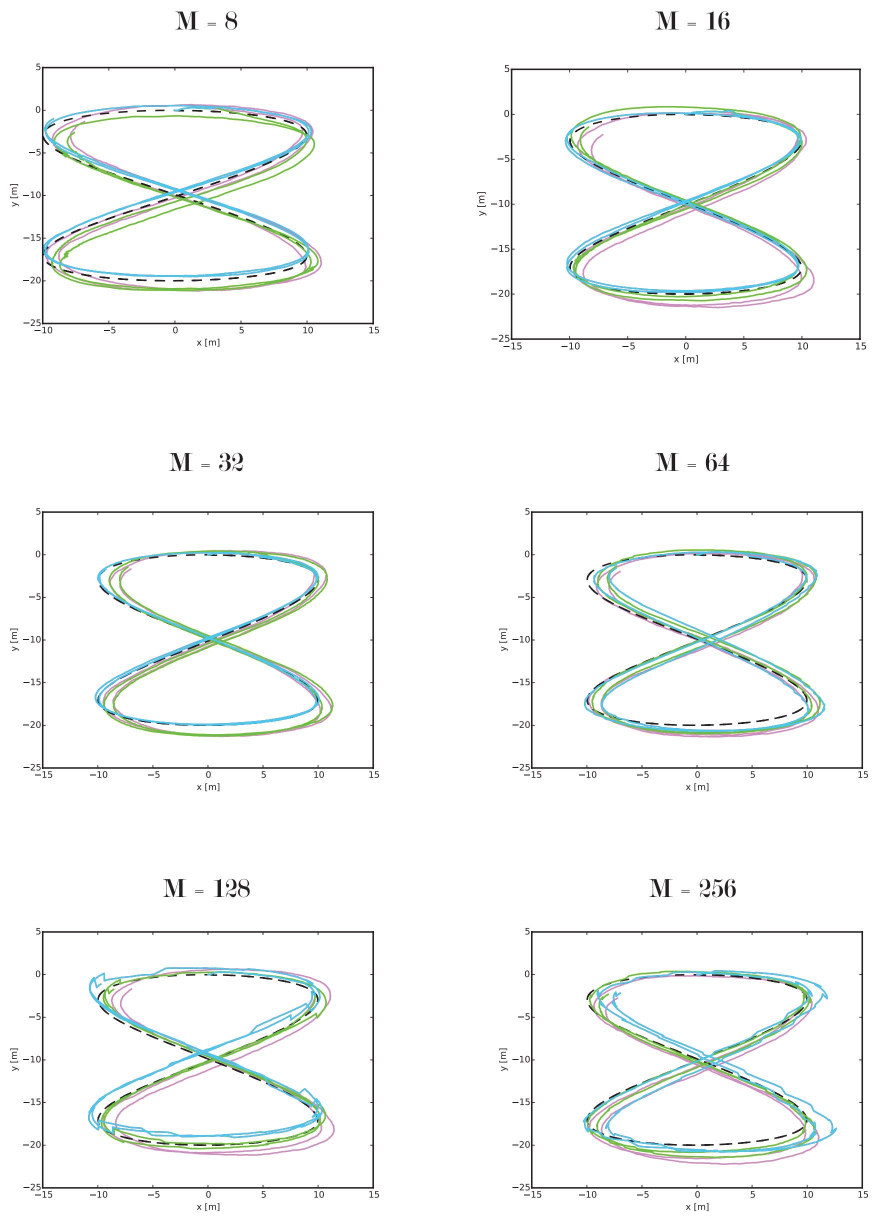

Figure 40: *Simulation of the asterix performing the figure eight movement with various VO frequencies M. In xy-coordinates.*

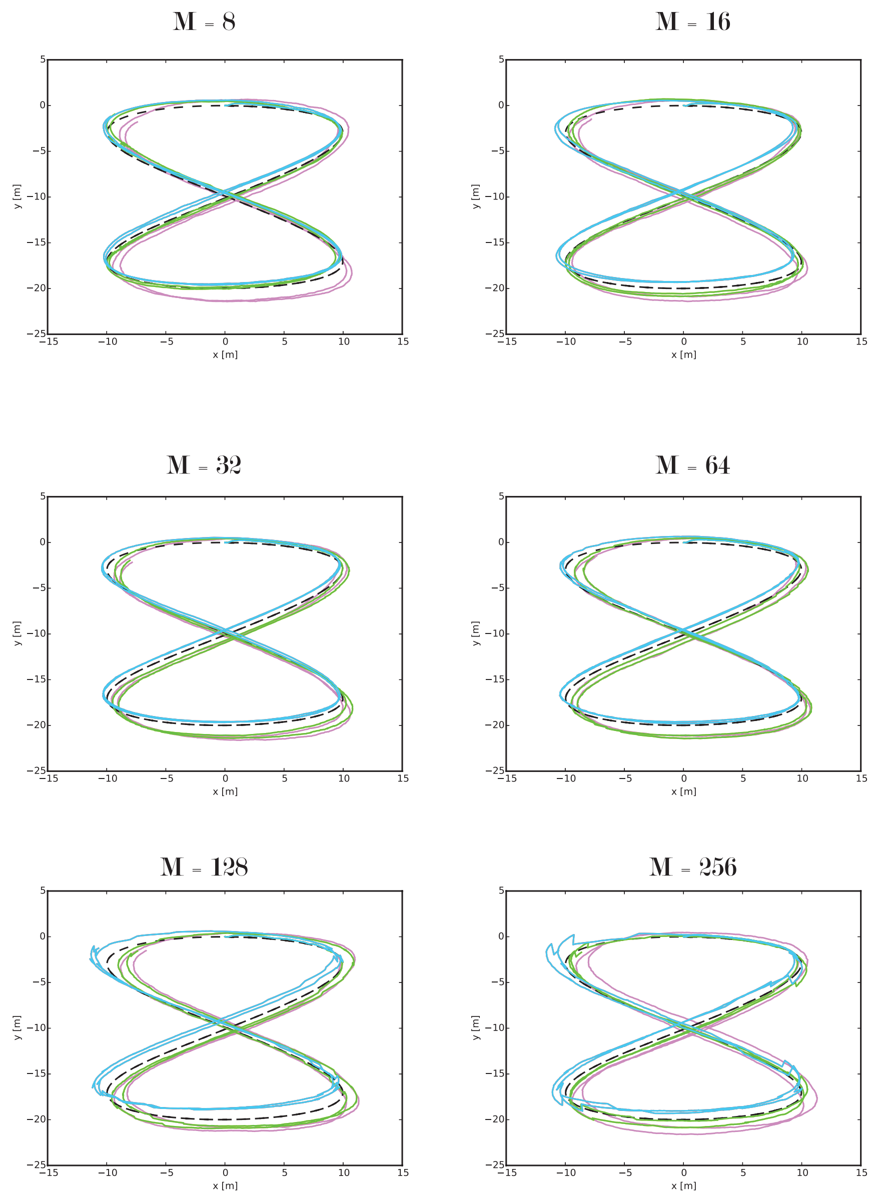

Figure 41: Simulation of Asterix performing the figure eight movement with various VO frequencies M and yaw covariance threshold. In xy-coordinates.

4.2.6 Discussion: Effect of VO Frequency

We can see that there is a limit to the usefulness of VO measurements. If the frequency is very low, the filter will not have managed to accurately maintain the cross-covariances and state covariances associated with the smoothed states, this results in the basic UKF performing better than the rest.

We see that the masked filters are better at tackling the lower VO frequencies than the smoothing filters. Furthermore, the covariance threshold technique resulted in better performance than the simple masking filters. This can be observed in Fig.41. This allowed the smoothing filters to be relevant for lower VO frequencies, but there was a limit to how much this could help. A rough estimate is to say that for the VO frequencies of $M > 64$, the corrections in the smoothing filters caused jumps that could prove problematic if present in Asterix.

4.3 Experimental Data

This section presents experimental data that was collected by Adigo for development of the Asterix robot, and tests of the localization filter run on this data. An artificial 500 ms delay was added to the VO measurements.

4.3.1 Datasets

For this thesis, two datasets were used. They will from here on be referred to as dataset A and dataset B. Some traits of dataset A are given in Tab.16, and some traits of dataset B are given in Tab.17.

Dataset A	
Duration	30 min 43 s
GPS Frequency	21.0 Hz
IMU Frequency	19.8 Hz
Wheel Encoder Frequency	20.4 Hz
Pictures Processed	67

Table 16: *Specifications of dataset A. Pictures processed are the number of pictures for which VO was manually found.*

Dataset B	
Duration	24 min 9 s
GPS Frequency	18.5 Hz
IMU Frequency	Not available
Wheel Encoder Frequency	20.4 Hz
Pictures Processed	89

Table 17: *Specifications of dataset B. Pictures processed are the number of pictures for which VO was manually found.*

In Fig.43 the GPS East-North data from dataset A is given. We see that Asterix moves down a row of crops and turns at the end. When Asterix is stopped for a while, the GPS loses RTK fix and gives erratic measurements. This is shown in 44. For this test we restrict attention to a subset when the robot is moving straight down the row.

In Fig.45 the GPS East-North data from dataset B is given. Here Asterix moves down a row of crops. In the beginning Asterix is standing still for a while, during which the GPS again loses RTK fix and gives erratic measurements. This is shown in Fig.46. For this test we restrict attention to a subset when the robot is moving straight down the row.

The IMU was available in dataset A, but not available in dataset B. The Euler angles taken directly from the data are shown in Fig.42. The inital pitch is 180° and yaw is between 90° and 100°. In the Fig.43 we see that the robot is moving down the row at an angle between -90° and -100°. The ROS driver for CHR-UM6 has not converted the angles from NED to ENU. This conversion is equivalent to switching pitch and roll, and inverting yaw. The IMU appeared to have been installed upside down as the accelerometer reports -1 g in its NED like I frame. This NED like I frame has angular rates in the oppposite direction of a ENU like I frame. The transformation between them is a 3D rotation of 180° around I_x.

In Fig.47 an example of the pictures taken during dataset A is shown. The crops are close to fully grown and often obfuscate the view of the camera. There are many rocks in the soil that are larger than 3cm x 3cm. The tips of these rocks and the bottom of the holes between them have shifted differently than the even soil around them when comparing two pictures. If the features chosen for the 2D VO displacement estimation are not on the same level, the measurement will be affected by a 3D environment error. The subset of pictures processed were from when the robot was performing a straight motion along the row.

In Fig.48 an example of the pictures during dataset B is shown. The crops are smaller, and placed consistently to the left side of the image. The soil has few rocks larger than 1.5cm x 1.5cm and appears flat and even. The subset of pictures processed were from when the robot was performing a straight motion along the row. This suggests that the VO measurements estimated from this dataset are likely more correct than dataset A. However, the crops take up most of theleft part of the pictures and throw a shadow on the middle, making the VO measurements more easily affected by errors caused lens distortions.

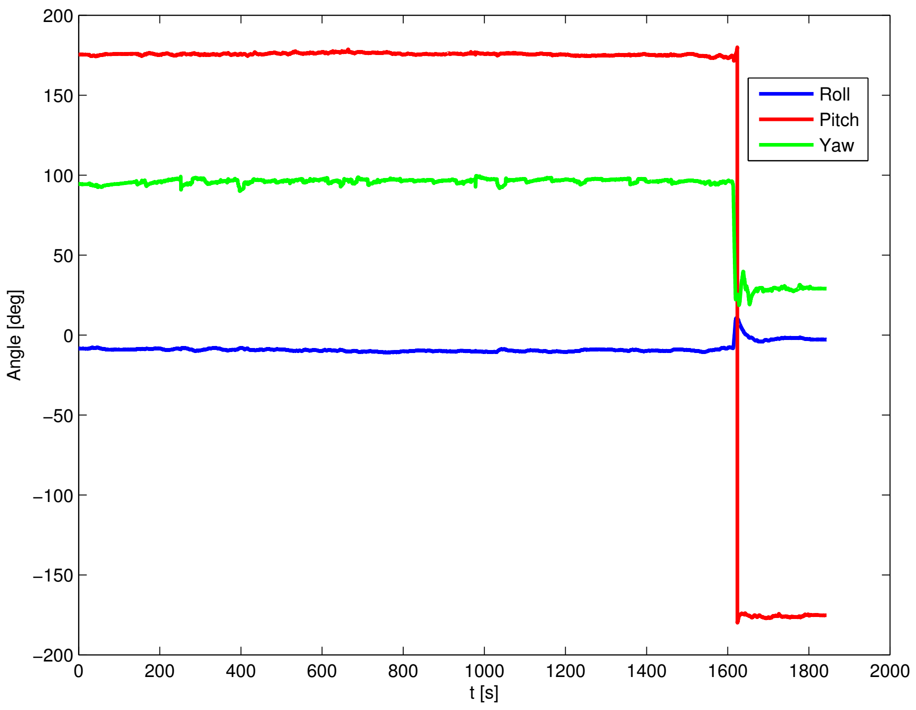

Figure 42: *Roll, pitch and yaw as reported by the IMU for dataset A.*

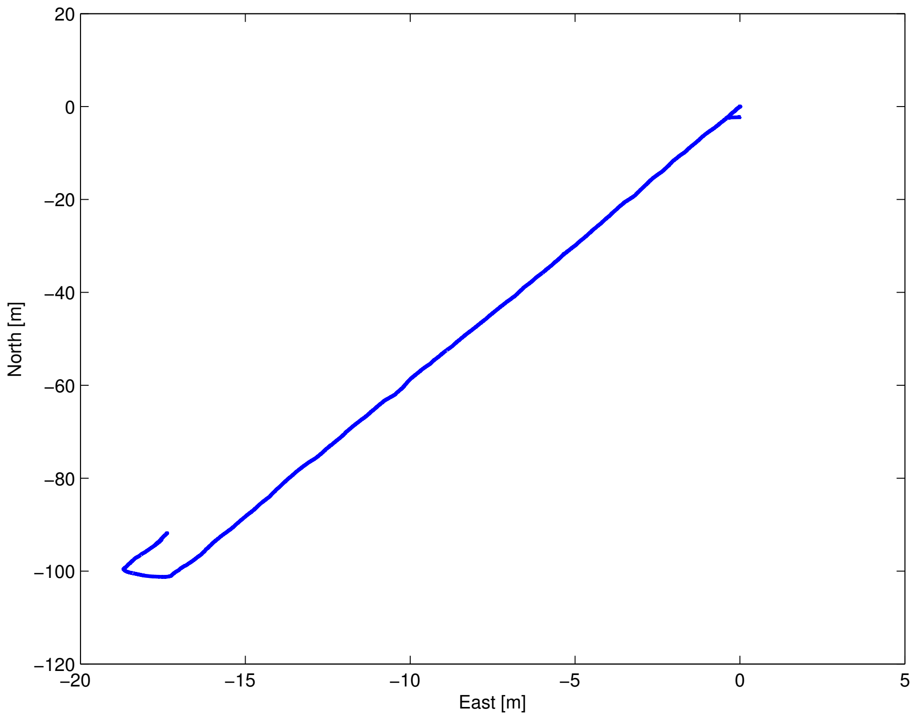

Figure 43: *GPS from dataset A.*

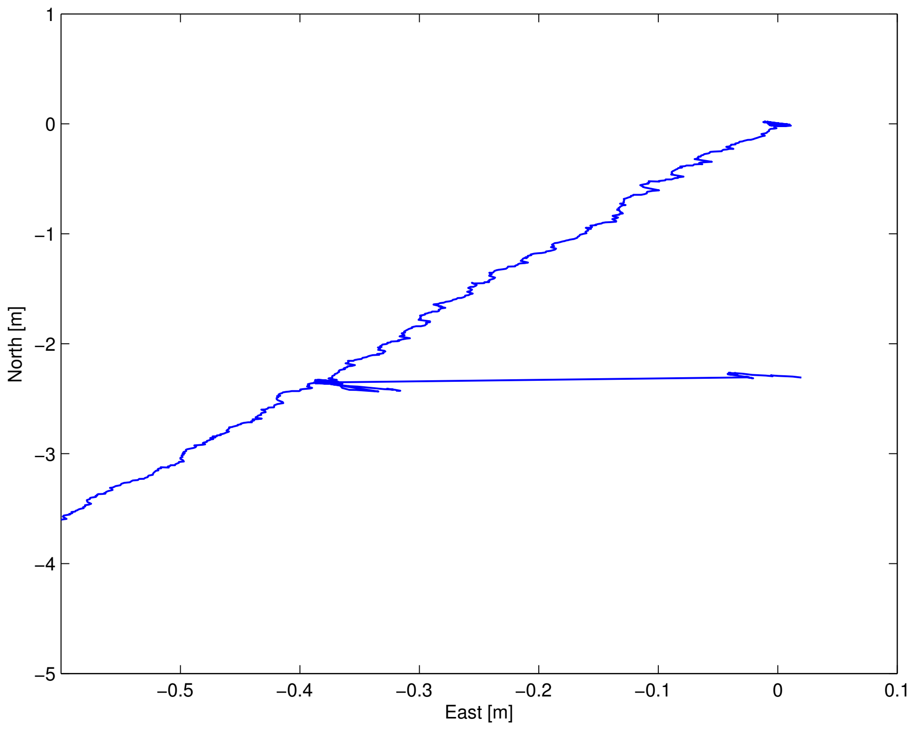

Figure 44: *GPS losing RTK fix when standing still in dataset A.*

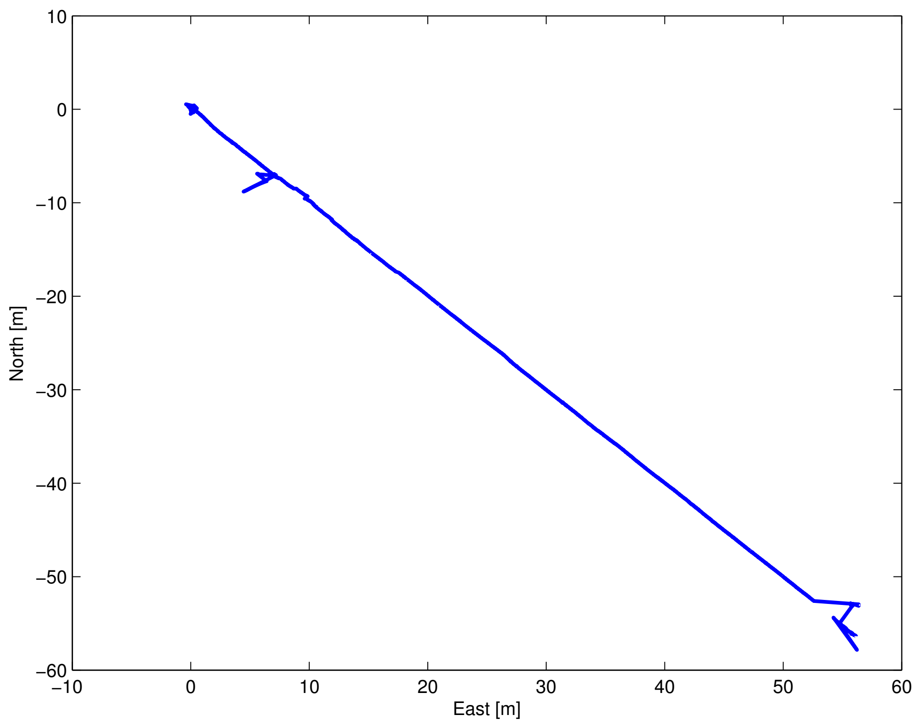

Figure 45: *GPS from dataset B.*

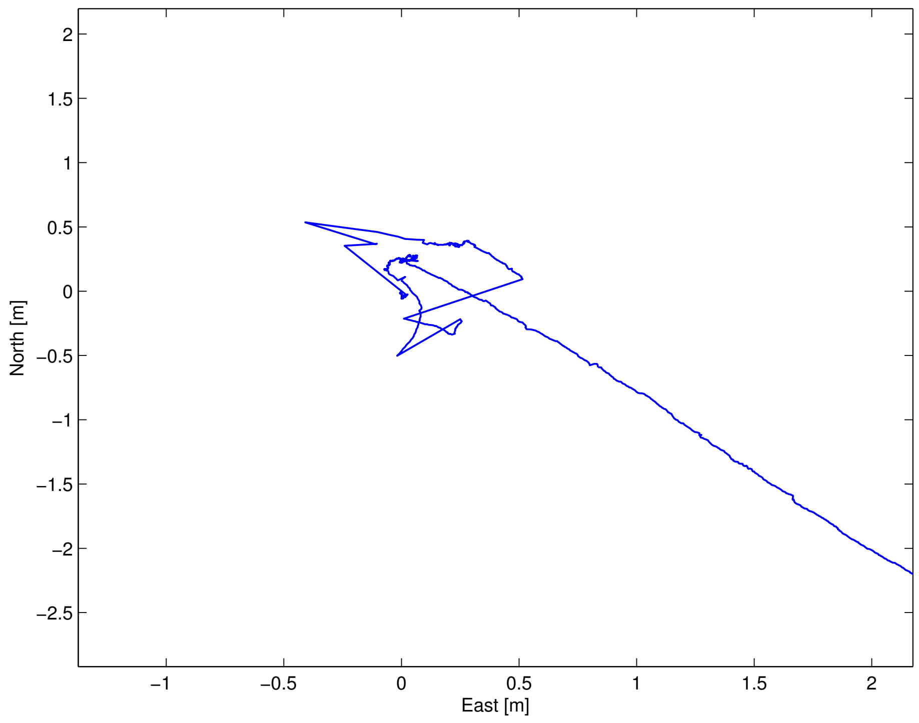

Figure 46: *GPS losing RTK fix when standing still in dataset B.*

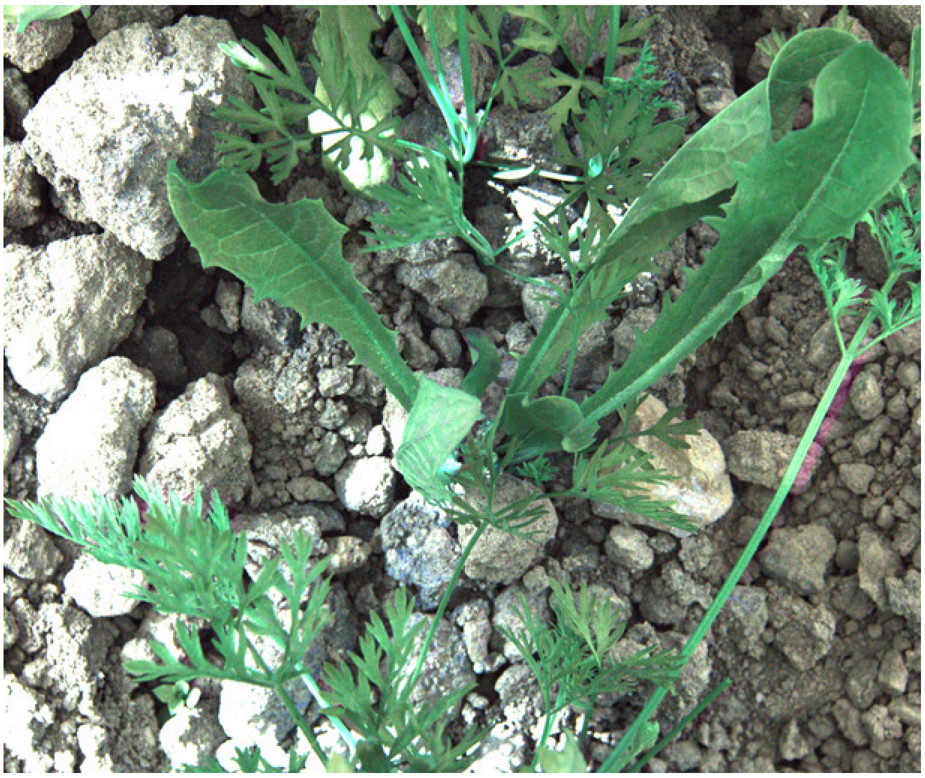

Figure 47: *Example of the pictures from dataset A.*

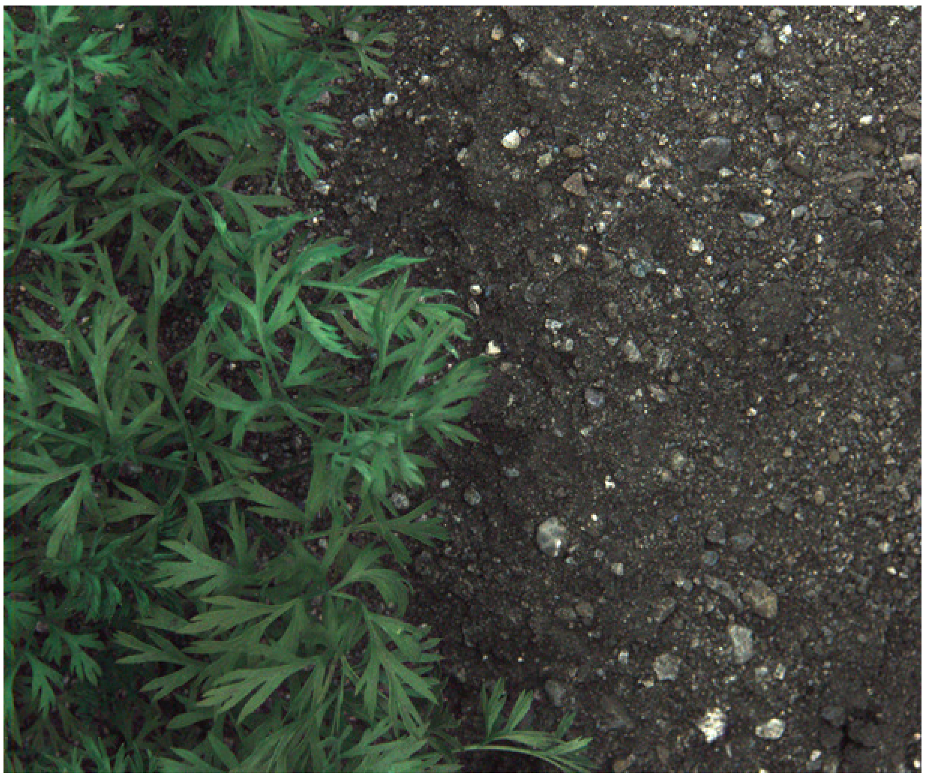

Figure 48: *Example of the pictures from dataset B.*

4.3.2 Localization with Dataset A

In this scenario the EKF and UKF are run without VO measurements and perform practically the same. SCEKF, SCUKF, JEKF, and JUKF fuse VO measurements and also behave approximately the same. The robot is moving slowly and steadily enough for the filters to have almost overlapping estimates. We first note that if GPS is considered extremely accurate (order of 10^{-5} in measurement noise covariance), we can observe oscillations in the position estimate (see Fig.49).

In Fig.50 we have ignored the GPS measurements to investigate the effect of the VO measurements. From inspection VO measurements themselves, it is apparent that the technique has overestimated sideways motion. In this case it has been consistently half a centimeter to the right relative the ROS-BODY frame for quite some time. When there is no GPS measurements, this error causes a gradual drift in position as the filter has assumed VO measurements to be highly accurate (R_{VO} in the order of 10^{-5}).

In Fig.52 the localization filter was run up to the same point, but now with GPS measurements available. The drift from the VO measurements are still present in the filters, and affects the smoothing filters more than the masked filters.

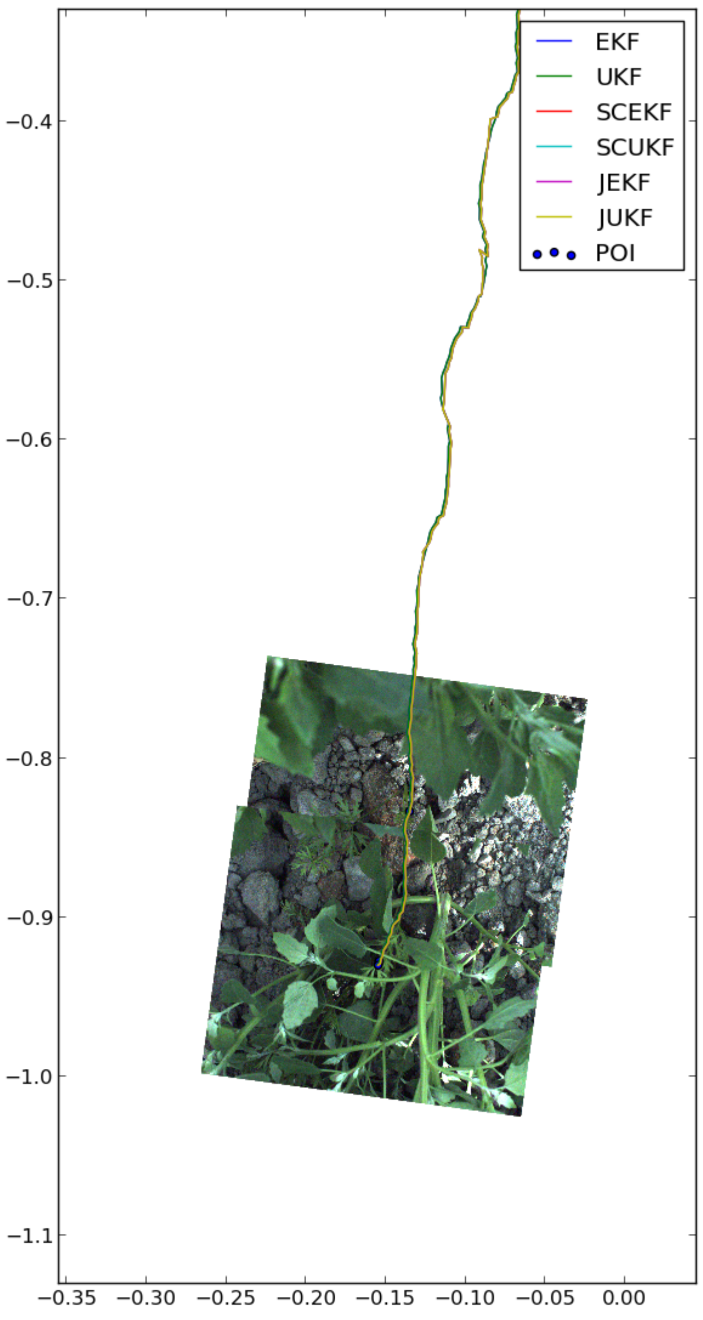

Figure 49: *Oscillations resulting from the GPS measurements. This is only presented with the smoothing filters and the VO ignoring filters.*

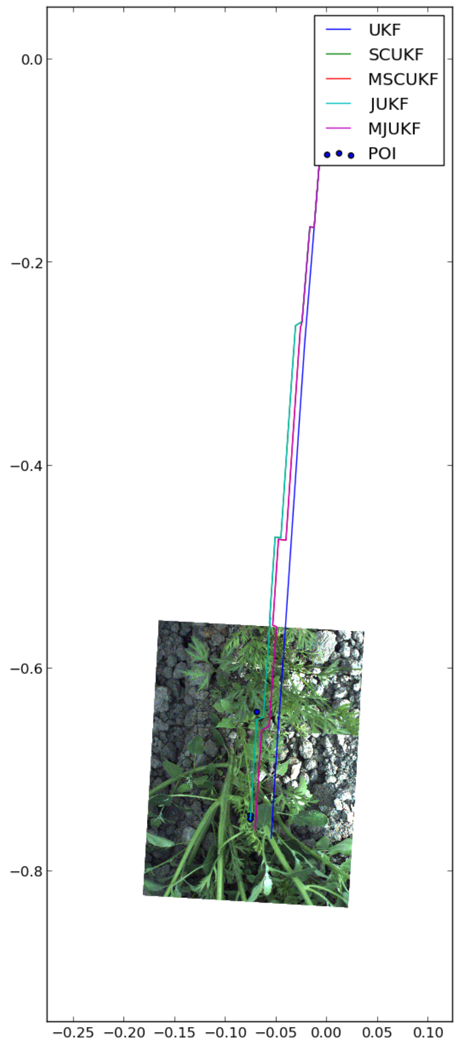

Figure 50: *Localization of dataset A without GPS, emphasizing the bias error in the VO measurements.*

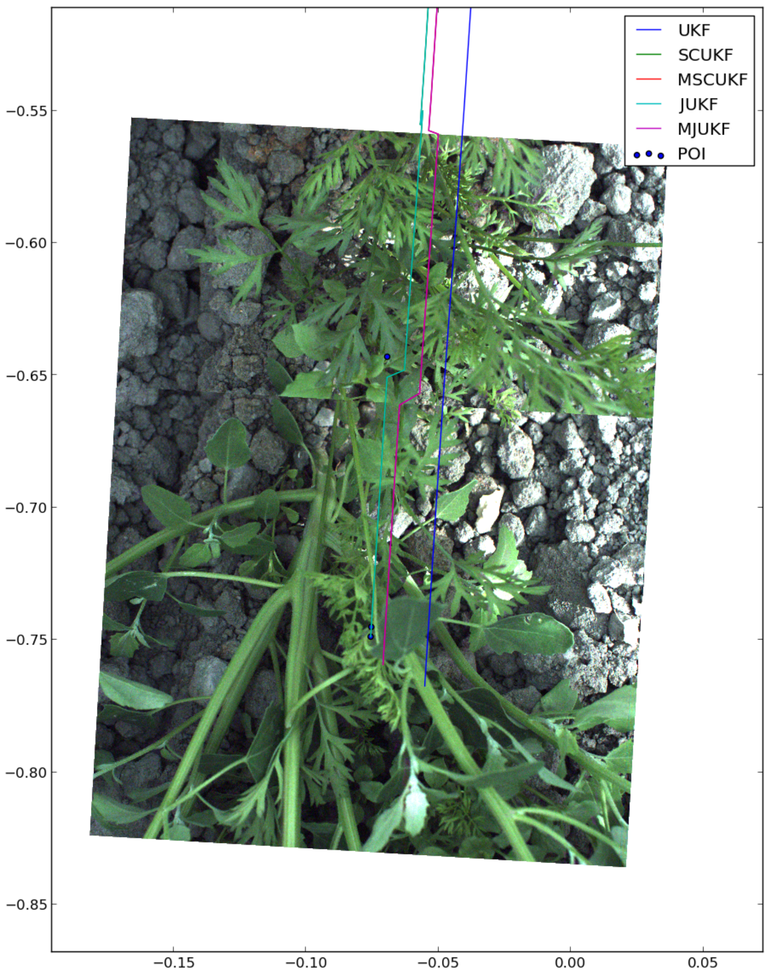

Figure 51: *Localization of dataset A without GPS, zoomed in to show the overlap of the images.*

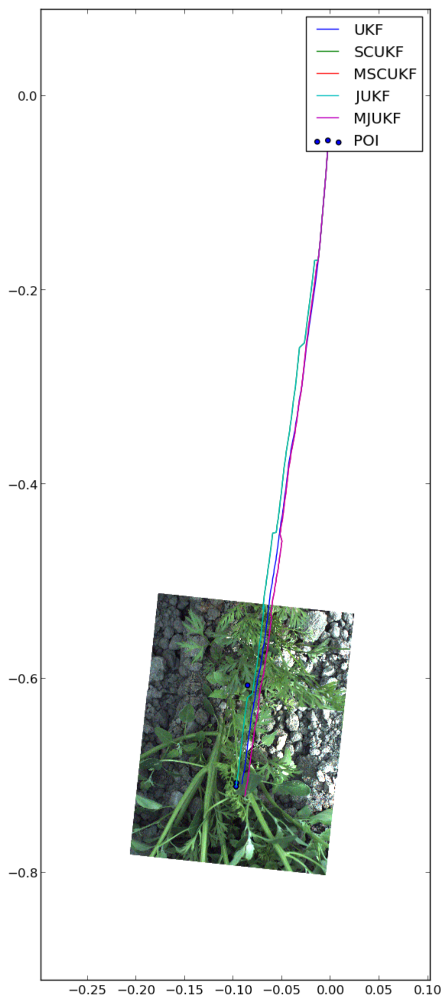

Figure 52: *Localization with dataset A with GPS, emphasizing the VO bias having less of an effect when the GPS is available.*

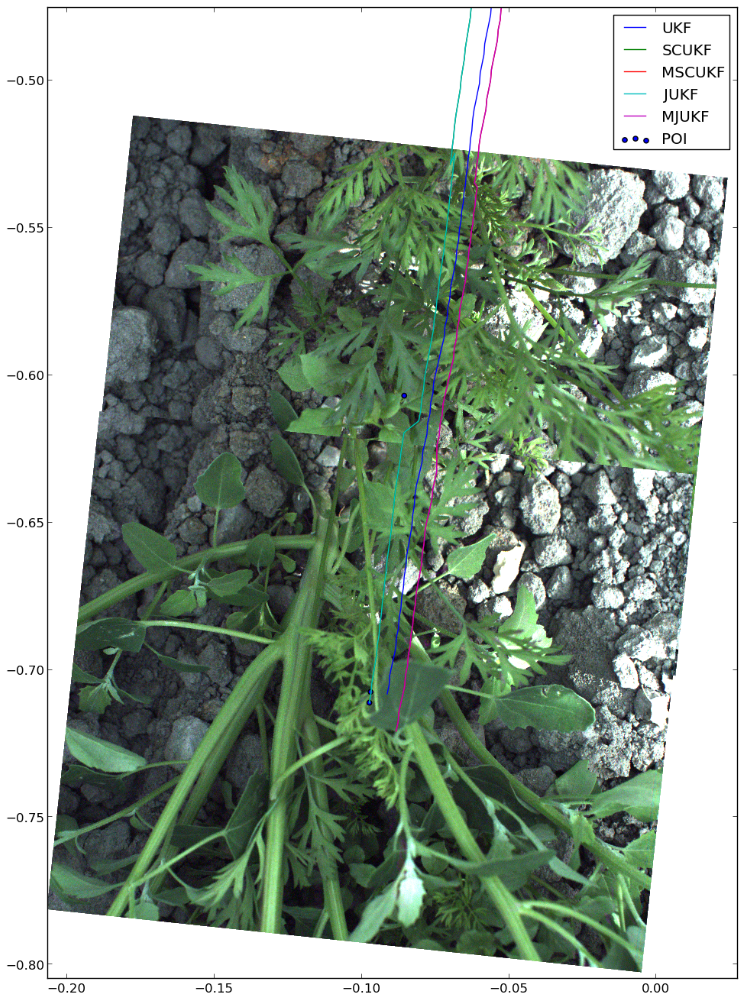

Figure 53: *Localization with dataset A with GPS enabled, zoomed in to show the overlap of the images.*

4.3.3 Discussion: Localization with Dataset A

The oscillations observed in the GPS measurement are in the order of centimeters. This could adversely affect the spraying mechanism if not handled appropriately. In this dataset a person was following the prototype, controlling it manually to follow the row. If the person had to perform continuous corrections in heading, and the GPS was placed offset from the center of rotation, a lever arm effect may be the culprit. The lever arm effect would have to be compensated for when gathering the GPS data if subcentimeter precision is desired. If the oscillations are in the actual data, this is neither apparent in the IMU nor the VO measurements.

To counteract the oscillations, the GPS measurements were considered less useful than they potentially could be, this was done by increasing the noise covariance R_{GPS} to an order of 10^{-2}. The oscillations still affect the filter but increasing the covariance further will result in the filter practically ignoring the GPS measurements.

The errors observed in the VO measurements are problematic, they indicate that the manual feature search method did not work as well as expected. Horn's method does not account for potential curvature of the lens or height disparity in the features. The error these introduce can be reduced by increasing the noise covariance associated with the VO measurements, but doing so would make the effect of VO measurements on the filters unnoticeable. As this thesis attempts to evaluate the methods for future implementations, highlighting such effects is of the essence.

When comparing Fig.51 and Fig.53 it is apparent that including GPS measurements has not improved our overlap significantly. This can be observed consistently for all places we have images overlapping. As such, it is likely that our image sizes are slightly off. The error in overlap is equivalent to the error in our picture sizes being wrong by approximately 0.5 cm in both width and height. Such a size difference is small enough that the curvature of the lens in combination with unaccounted for 3D effects may also be the culprit. We see that the quality of the image closer to the edges is lower than near the center, implying that the curvature of the lens is affecting the image. Without further testing of Asterix moving over a known surface with known distances, the source of the error cannot be exactly determined.

4.3.4 Localization with Dataset B

In Fig.54 the GPS measurement has allowed us to estimate the heading without the need for an IMU. However, without it, the smoothing filter has not readjusted the position of the previous picture enough to align them accurately. In this dataset as well, the VO measurements are affected by the same errors.

In Fig.55 the GPS measurements were ignored. The errors caused by the false

VO measurements are more prominent in this particular part of the trajectory. The tracking using only wheel encoders, segments between pictures, appear better than with the erratic VO measurements. The lack of IMU and GPS has caused significant drift for all the delay fusing filters.

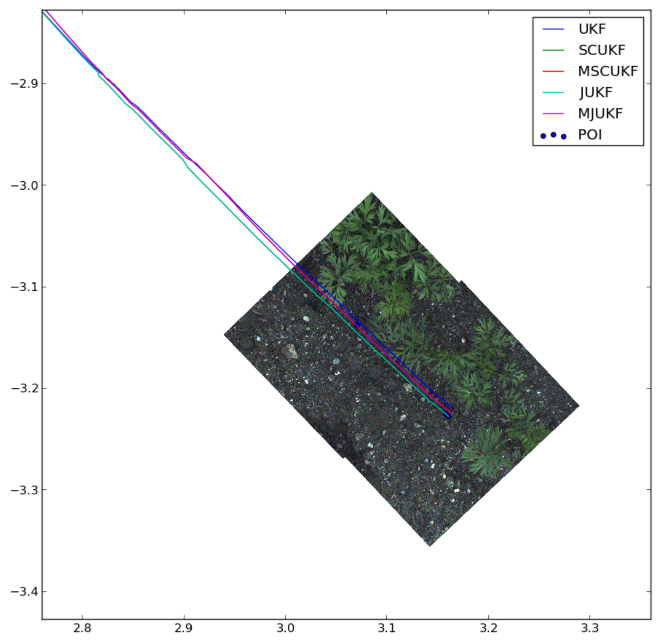

Figure 54: *Localization of dataset B with GPS, emphasizing the convergence in heading.*

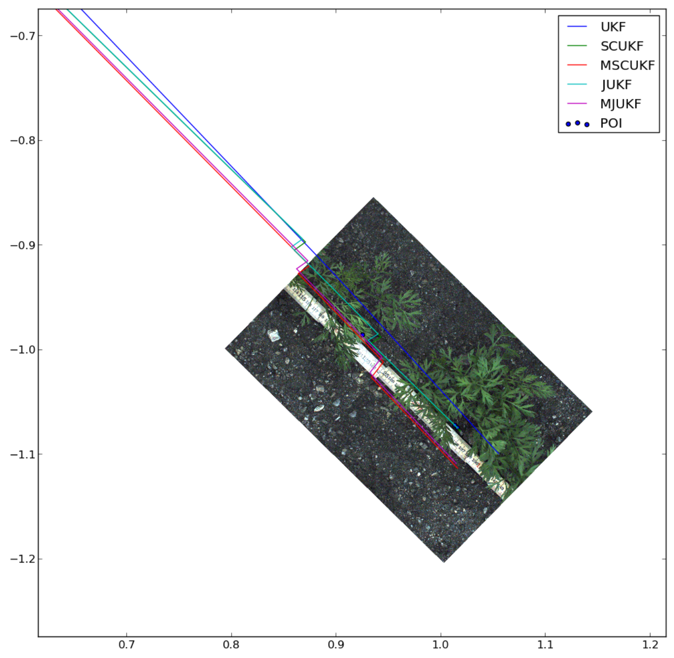

Figure 55: *Localization of dataset B without GPS, emphasizing the error caused by false VO measurements.*

4.3.5 Discussion: Localization with Dataset B

With GPS and VO available, the localization filters were able to reconstruct the heading even though there were no IMU measurements available. For this dataset, the VO measurements were also affected by unmodelled effects. The surface appears flat and even, but the features can only be chosen off-center to the right. This suggests that the curvature of the lens is the major contributor to the error.

When simulated without an IMU and ignoring the GPS, the errors from the VO measurements became cumulative. This means that measurements may be weighted incorrectly after only a short period of time. And without significant devaluing of the VO measurements compared to the previous dataset, the errors have a greater effect on the localization procedure.

4.3.6 Picture Frame Tracking

The multiple-point smoother framework is capable of tracking a set of picture centerpoints with respect to the current state of the system. This allows Asterix to maintain an estimate of the position of the pictures relative to its current position. If the spray nozzle are situated a distance away from camera module, the oldest states in the filters need only be marginalized after they are further away than the spray nozzles. From a practical perspective, this requires more bookkeeping on the control flow as the VO measurement will not be related to the oldest smoothed state, but to a specific smoothed state.

In Fig.56 we have set the VO delay to be 10 s, this causes us to have 7 tracked states that are smoothed. We are observing the behavior of the JUKF filter. This is an example of the picture frame tracking behavior. In the picture we can see that the smoothing filters have placed the picture frames (POI) away from the previous state estimate (purple line). It has done this as: the GPS measurement oscillates slightly, the IMU has reported a straight heading during these oscillations, and there is a minor bias in the heading estimate compared to the GPS direction that has not been removed. It is also apparent that even with a 10 s delay, no instability is introduced to the system from the VO measurements. They have merely become much lower weighted in the estimation of the current state.

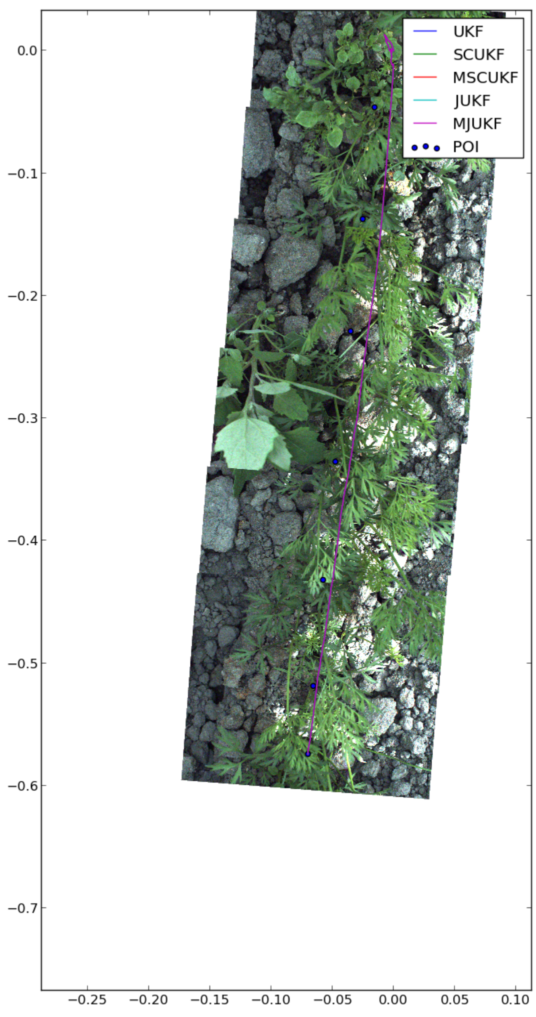

Figure 56: *Localization in dataset A using GPS and a delay of 10 s. This was to emphasize the picture frame tracking capabilities of the multiple-point smoother.*

5 Epilogue

5.1 Discussion

There is a limit to how much a downward facing monocular camera can contribute. Monocular cameras have to rely on many assumptions that are not necessarily true for real life scenarios. In dataset A, the rocks were large enough to negatively affect the VO measurements even when chosen manually. With binocular vision, the robot would be able to know more about the surface height of the terrain and could, at least theoretically achieve significantly better displacement measurements. An alternative is to increase the framerate of the camera, and track the features movements in the filter. This would allow one to infer their height by their relative motion to each other, and a higher precision would be possible.

For a robot moving at a constant velocity down a straight path, any VO measurements is approximately equal to any other. This means that it was difficult to evaluate whether the artificial delay had any positive or negative effect on the fusion. There were no observed ill effects for a delay of up to 3 s.

Displacement measurements related to a local previous frame can cause problems if not properly tuned. In the “Effects of VO Frequency” section it was found that there is a limit to the how useful a VO measurement is. As the frequency decreased, there came a point where the filter with VO measurement no longer worked better than the filter only using WO and IMU.

The tuning of delay fusing VO filters is more difficult than normal filters. These are intermittent measurements that have to have high accuracy to be useful. With the chosen model the intermittent measurements can adversely affect the state estimate. This was seen where the linear velocity converged to the point where the WO measurements were not weighted high enough during the figure eight movement and only the VO measurements were used for evaluating the velocity.

The joint method is a more generalized form of the SC method, with a clear Bayesian basis. This will allow the joint method to be incorporated into more complex scenarios than the SC method. Regarding this specific scenario, the joint method implementation performed slightly worse than the SC implementation. With very low process noise, numerical issues appear in Cholesky factorization and matrix multiplication. As it is implemented in this thesis, the joint method performs more of these steps than stochastic cloning and will therefore be more affected by numerical issues. In terms of runtime, the SC implementation is faster than the joint method. This comes from the joint method performing more Cholesky factorizations in the extend step than SC does in the predict step. In the theory section it was shown that the issue with stochastic cloning that arises when approximating a joint distribution of the current state and the current state will not become a significant issue. In practice SC is an acceptable method. However, for further theoretical development of OOSM techniques, the basis that the joint method gives is more useful.

The benefits of smoothing the lagged states depends on how well the model represents reality. The model used in this thesis required a slow moving system to smooth the lagged states correctly. For faster moving systems, with discrepancies betwe estimation and reality, masking appears to increase robustness. Masking will, however, adversely affect the estimation when the model in the filter accurately represents the system. As was seen for the straight line motion where the pictures were taken after the turn.

Angular differences can become an issue with 2D localization filters. Angles close to $\pm\pi$ can cause difficulties unless properly accounted for. It can be resolved by comparing the values on the interval 0 to 2π and changing the direction depending on whether the subtrahend is greater than or lesser than the minuend.

A possible objection to using VO measurements as they are described in this thesis is because they cause small discrete jumps in position and heading. These discrete jumps are an issue in applications where local accuracy is of the essence, and is an argument for ignoring GPS measurements that will cause distinct jumps in position. A GPS will cause the position estimate to jump a distance from its current location. This happens for all measurements directly observing position. In the case of Asterix, if the camera is located the same distance above ground as in this case, for at least 50% overlap the robot will have moved 10 cm between each image. At such distances, without GPS measurements, a 5° estimation error will result in 1 cm offset in the position estimate between one image and the next. With more reliable feature identification techniques, the VO measurement will be able to counteract such an error. And if the spray nozzles are located further back than the latest picture, a smoothing filter will be able to track multiple pictures back in time and adjust the pictures according to the most recent measurements.

5.2 Future Work

For the construction of Asterix, further development of the feature identification methods are necessary. OpenCV provides tools such as SURF, ORB and other feature identification methods that may prove useful in this. With the techniques described in this thesis, the feature identification should focus on accuracy more than efficiency. In the manual feature identification the single-channel grayscale images proved most useful.

Adopting the delay fusion methods to error-state Kalman filter (ESKF) may give better results. One of the issues encountered was in the underlying model. Error-state Kalman filters are practical in localization tasks as they are generally less sensitive to the knowledge of the underlying robot motion(Groves, 2013, pp.661-663). Following the Bayesian derivation of OOSM techniques in this thesis with an ESKF as the underlying approach should be possible. Such a localization filter would be general enough to be applied on a variety of platforms with similar sensors available (Roumeliotis et al., 1999). Development in that area can greatly benefit Adigo in the long run.

One of the ideas that had to be discontinued due to lack of time was to evaluate the methods on particle filters. As the methods were derived from probability distributions, they should prove useful in OOSM fusion with particle filters. In this area it is likely that stochastic cloning performs worse than the joint method as stochastic cloning is an approximation technique useful for filters described in terms of means and covariances. A major difficulty for particle filters lies in marginalization. This is a costly process, and investigation into how to do this in an efficient, optimal manner may yield interesting results.

It was observed that delayed measurements were not always useful. Further analysis of the limitations of the delay fusion techniques will be necessary to create a more rigorous understanding of them. The probabilisticFilters and delayFusion modules created for this thesis can provide a basis for this investigation. However, the results from the fast turn into the straight path revealed that model inconsistency is an issue. It is therefore important to test with different underlying models. As the modules created in this thesis considered undriven system models, patching of the modules to include an input $\boldsymbol{u}$ must be done if they are to be used in that regard.

High-precision agriculture requires high-precision evaluation of the filter. In order to properly evaluate the localization filter on the actual robot platform, one needs more testing in a controlled environment. One way of doing this is to print out a centimeter grid over which the robot moves and takes pictures. If placed on a flat parking lot the images will give the accurate VO measurements necessary to evaluate how artificial delay affects the localization. Discrepancies caused by errors in the cross-covariance and smoothed state will be easily identifiable in the grid between two pictures not aligning properly. The centimeter grid may also reveal whether it is lens distortion, errors in the image size, or height disparity of the features that was the cause of the problems with the VO measurements.

Robust methods of handling erratic sensor readings is also necessary if the GPS and IMU used on the prototype is to be used further down the line. The RTK fix appeared to be dependent on whether the robot was in motion or not. This suggests a simple method of handling the RTK fix problem: if the estimated velocity is below a certain threshold, the GPS measurements can be discarded.

The picture frame tracking described in the Results chapter suggests a method by which one can fuse VO measurements even when the VO measurements do not arrive in a first-in-first-out queue as was the case for our system. This requires keeping track of which lagged states are associated with which VO measurements, and whether there still are VO measurements to be processed that are dependent on the tracked states. This is a scenario where the measurements are truly out of sequence. Theoretically, the framework described in this thesis, with the bookkeeping suggested in this paragraph, should solve this problem, but further testing is necessary for experimental validation.

By approaching the situation with a SLAM mentality, the 3D effects negatively affecting the VO measurements can be counteracted. If the image rate is increased, a filter may be designed that tracks individual features. The relative motion of these features can be used to recreate the motion of the robot in 6DOF, allowing us to account for the scaling effect that height differences cause, and arrive at more accurate VO measurements. The surfaces observed in the datasets have sufficient features to be tracked, the issue lies in correctly identifying them in a 3D environment, and removing potential lens distortions.

5.3 Conclusion

Fusion of OOSMs were approached from a Bayesian point of view. This provided an improved understanding of stochastic cloning as a method, and led to the joint method described in this thesis. A byproduct of this investigation was the unscented multiple-point smoother. This filtering technique may prove useful in other tasks where smoothing has to be performed on-demand.

VO for the Asterix robot was discussed with measurements from a monocular downward facing camera. Warm-start of the feature search proved helpful in the manual feature search and will undoubtedly prove useful in the final implementation. The underlying delay fusing filter methods provide estimates of the lagged state which can be used to reduce the search space drastically per feature.

The implementations proved useful in investigating how to tune the filters. An unorthodox trick was applied that can help future uses of the techniques investigated in this thesis. The trick of placing a lower threshold on the smoothed state covariance was proposed as an alternative to masking the smoothed states. This helped combating filter inconsistency during turns.

The localization filters with delayed VO measurements worked on the datasets given. The precision was not as high as desired as the VO measurements were dissatisfactory. The delay fusion techniques outlined in this thesis, will be able to help against skidding in the wheels, maintaining accurate position estimates when standing still, and keeping an estimate of where the previous picture was taken compared to the current state of the robot. For the experimental data, it was possible to localize Asterix at an accuracy of approximately 3 cm when disregarding the erratic VO measurements. For the purpose of weed control, higher precision is needed than can be provided with the available IMU, wheel encoder and GPS unit. Without 6DOF pose estimation or 3D feature tracking, it is questionable whether sufficient accuracy is possible.

5 References

Arbo, M. H. (2014). Sensor Fusion with Out-of-Sequence Measurements: Localization in an Agricultural Robot. Final Project.

Bar-Shalom, Y. (2002). Update with out-of-sequence measurements in tracking: Exact solution. *IEEE Transactions on Aerospace and Electronic Systems*, 38.

Bar-Shalom, Y. and Li, X.-R. (1996). *Multitarget-multisensor tracking: Principles and techniques*. Yaakov Bar-Shalom, 1st edition.

Brekke, E. and Chitre, M. (2014). A multi-hypothesis solution to data association for the two-frame SLAM problem. *The International Journal of Robotics Research*, 34.

Brown, R. G. and Hwang, P. Y. (2012). *Introduction to Random Signals and Applied Kalman Filtering*. John Wiley & Sons, Inc., 4th edition.

Challa, S., Evans, R. J., and Wang, X. (2003). A Bayesian solution and its approximations to out-of-sequence measurement problems. *Information Fusion*, 4.

Challa, S., Evans, R. J., Wang, X., and Legg, J. (2002). A fixed-lag smoothing solution to out-of-sequence information fusion problems. *Communications in Information and Systems*, 2.

De La Cruz, C. and Carelli, R. (2006). Dynamic Modeling and Centralized Formation Control of Mobile Robots. In *IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics*, pages 3880–3885.

FAO (2009). Global agriculture towards 2050 The challenge. Technical report, Food and Agriculture Organization of the United Nations.

Forster, C., Pizzoli, M., and Scaramuzza, D. (2014). SVO: Fast semi-direct monocular visual odometry. In *Proceedings of the 2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 15–22.

Fossen, T. I. (2011). *Handbook of Marine Craft Hydrodynamics and Motion Control*. John Wiley and Sons.

Gavrilets, V. (2003). *Autonomous Aerobatic Maneuvering of Miniature Helicopters*. PhD thesis, Massachusetts Institute of Technology.

Goldberg, S. B. and Matthies, L. (2011). Stereo and IMU assisted visual odometry on an OMAP3530 for small robots. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*.

Grændsen, Ø. W. (2014). Automatic Visual Weed Recognition. Master thesis, Norwegian University of Science and Technology.

Groves, P. D. (2013). *Principles of GNSS, inertial, and multisensor integrated navigation systems*. Artech House, Inc., Boston, 2nd edition.

Haykin, S. S. (2001). *Kalman filtering and neural networks*, volume 5. John Wiley & Sons, Inc., New York.

Horn, B. K. P. (1987). Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America*, 4.

Huang, G. P., Mourikis, A. I., and Roumeliotis, S. I. (2010). Observability-based rules for designing consistent EKF SLAM estimators. *The international Journal of Robotics Research*, 29.

Julier, S. J. (1998). A Skewed Approach to Filtering. In *SPIE Proceedings Signal and Data Processing of Small Targets*, volume 3373.

Julier, S. J. and Uhlmann, J. K. (2002). Reduced Sigma Point Filters for the Propagation of Means and Covariances Through Nonlinear Transformations. In *Proceedings of the 2002 American Control Conference*.

Kalman, R. (1960). A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*.

Kandepu, R., Foss, B., and Imsland, L. (2008). Applying the unscented Kalman filter for nonlinear state estimation. *Journal of Process Control*, 18.

Klungerbo, A. T. (2013). Drop-on-Demand i Presisjonsjordbruk. Master thesis, Norwegian University of Science and Technology.

Larsen, T. D. and Poulsen, N. K. (1998). Incorporation of Time Delayed Measurements in a Discrete-Time Kalman Filter. In *Proceedings of the 37th IEEE Conference on Decision and Control*, number 4.

Mahler, R. P. S. (2007). *Statistical Multisource-Multitarget Information Fusion*. Artech House, Inc., Norwood, MA, USA.

Maimone, M. and Matthies, L. (2005). Visual Odometry on the Mars Exploration Rovers. *2005 IEEE International Conference on Systems, Man and Cybernetics*, 1.

Mammarella, M., Campa, G., Fravolini, M. L., and Napolitano, M. R. (2012). Comparing Optical Flow Algorithms Using 6-DOF Motion of Real-World Rigid Objects. *IEEE Transactions on Systems, Man, and Cybernetics*, 42.

Martins, F. (2015). Velocity-based dynamic model and adaptive controller for diffential steered mobile robot. http://www.mathworks.com/matlabcentral/fileexchange/44850-velocity-based-dynamic-model-and-adaptive-controller-for-differential-steered-mobile-robot.

Merwe, R. V. D. (2004). *Sigma-point Kalman filters for probabilistic inference in dynamic state-space models*. PhD thesis, OGI School of Science and Engineering.

Merwe, R. V. D. and Wan, E. (2004). Sigma-point Kalman filters for integrated navigation. In *Proceedings of the 60th Annual Meeting of The Institute of Navigation*.

Mourikis, A. I. and Roumeliotis, S. I. (2007). SC-KF mobile robot localization: a stochastic cloning Kalman filter for processing relative-state measurements. *IEEE Transactions on Robotics*, 23.

Mourikis, A. L. and Roumeliotis, S. (2006). On the treatment of relative-pose measurements for mobile robot localization. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, pages 2277–2284.

Newcombe, R. a., Lovegrove, S. J., and Davison, A. J. (2011). DTAM: Dense tracking and mapping in real-time. In *2011 International Conference on Computer Vision*, pages 2320–2327.

Nister, D., Naroditsky, O., and Bergen, J. (2004). Visual odometry. *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1.

Perea, L., How, J., Breger, L., and Elosegui, P. (2007). Nonlinearity in sensor fusion: Divergence issues in EKF, modified truncated SOF, and UKF. In *Proceedings of AIAA Guidance, Navigation and Control Conference*.

Roumeliotis, S. and Burdick, J. (2002). Stochastic cloning: a generalized framework for processing relative state measurements. In *Proceedings of the 2002 IEEE International Conference on Robotics and Automation*, volume 2, pages 1788–1795.

Roumeliotis, S., Sukhatme, G., and Bekey, G. (1999). Circumventing dynamic modeling: evaluation of the error-state Kalman filter applied to mobile robot localization. In *Proceedings of the 1999 IEEE International Conference on Robotics and Automation*, volume 2, pages 10–15.

Salmond, D. J. (1989). Tracking in Uncertain Environments. Technical report, Royal Aerospace Establishment, Farnborough.

Thrun, S. (2002). *Probabilistic robotics*. The MIT Press.

Tuan Pham, D., Verron, J., and Roubaud, M. C. (1998). A singular evolutive extended Kalman filter for data assimilation in oceanography. *Journal of Marine Systems*, 16.

Urdal, F. (2013). Design of a Precision Spray Matrix. Master thesis, Norwegian University of Science and Technology.

Utstumo, T. (2011). Attitude Estimation in Agricultural Robotics: Design and Implementation. Master's thesis.

van Lint, H. (2008). Dual EKF State and Parameter Estimation in Multi-Class First-Order Traffic Flow Models. In *Proceedings of the 17th World Congress of the International Federation of Automatic Control*, pages 14078–14083.

Vik, B. (2014). *Integrated Satellite and Inertial Navigation Systems*. Department of Engineering Cybernetics, Norwegian University of Science and Technology.

A Appendix

A.1 Bayes Filter Derivation

Consider the hidden Markov model of Fig.1, with first order Markov assumption, the sensor Markov assumption and stationary processes. As shown in the theory chapter, the joint probability distribution for a sequence of configurations $\boldsymbol{x}$ is given by (2.3), let us look at one step in this sequence.

$$p(\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{z}_1) = p(\boldsymbol{x}_0)p(\boldsymbol{z}_1|\boldsymbol{x}_1)p(\boldsymbol{x}_1|\boldsymbol{x}_0)$$

by the marginal probability property:

$$p(\boldsymbol{x}_1, \boldsymbol{z}_1) = \int_{\boldsymbol{x}_0} p(\boldsymbol{x}_0)p(\boldsymbol{z}_1|\boldsymbol{x}_1)p(\boldsymbol{x}_1|\boldsymbol{x}_0)\mathrm{d}\boldsymbol{x}_0 \tag{A.1}$$

Bayes theorem states that our desired probability distribution $p(\boldsymbol{x}_1|\boldsymbol{z}_1)$ is given by:

$$p(\boldsymbol{x}_1|\boldsymbol{z}_1) = \frac{p(\boldsymbol{x}_1, \boldsymbol{z}_1)}{p(\boldsymbol{z}_1)}$$

Then the equation (A.1) can be used to form:

$$p(\boldsymbol{x}_1|\boldsymbol{z}_1) = \frac{p(\boldsymbol{z}_1|\boldsymbol{x}_1)\displaystyle\int_{\boldsymbol{x}_0} p(\boldsymbol{x}_0)p(x_1|\boldsymbol{x}_0)\mathrm{d}\boldsymbol{x}_0}{p(z_1)}$$

And that leaves us with the desired probability distribution. We see that given any initial distribution $p(\boldsymbol{x}_0|\boldsymbol{z}_0)$, we can arrive at any $p(\boldsymbol{x}_k|\boldsymbol{z}_{0:k})$, given the sequence of observations $\boldsymbol{z}_{0:k}$, transition model $p(\boldsymbol{x}_{k+1}|\boldsymbol{x}_k)$, and sensor model $p(\boldsymbol{z}_k|\boldsymbol{x}_k)$. In the Bayes filter formulation, this process has been split into two steps: predict and update.

The prediction step estimates the next probability distribution with observations up to and including the previous step. This is referred to as an *a priori* probability distribution. The predict step is given by:

$$p(\boldsymbol{x}_{k+1}|\boldsymbol{z}_{0:k}) = p(\boldsymbol{x}_{k+1}|\boldsymbol{x}_k)p(\boldsymbol{x}_k|\boldsymbol{z}_{0:k}) \tag{A.2}$$

The update step adjusts the estimate of the probability distribution by using the latest measurement $\boldsymbol{z}_{k+1}$ and the sensor model. This is referred to as an *a posteriori* probability distribution. The update step is given by:

$$p(\boldsymbol{x}_{k+1}|\boldsymbol{z}_{0:k+1}) = \frac{p(\boldsymbol{z}_{k+1}|\boldsymbol{x}_{k+1})p(\boldsymbol{x}_{k+1}|\boldsymbol{z}_{0:k})}{p(\boldsymbol{z}_{k+1}|\boldsymbol{z}_{0:k})} \tag{A.3}$$

The Bayes filter equations are the essential tools for probabilistic sensor fusion.

A.2 Kalman gain derivation

The following is a brief derivation of the kalman filter gain. We have a system:

$$\begin{aligned}\boldsymbol{x}_{k+1} &= f(\boldsymbol{x}_k, \boldsymbol{w}_k)\\ \boldsymbol{z}_k &= h(\boldsymbol{x}_k, \boldsymbol{v}_k)\end{aligned} \tag{A.4}$$

where $\boldsymbol{w}_k$ is process noise and $\boldsymbol{v}_k$ is measurement noise, f is the transition function, and h is the measurement function.

We will approach this with the assumption that the best estimate of a state given a measurement is the prediction plus a linear combination of the measurement residual. See Kalman (1960). Meaning that we are seeking the best linear unbiased estimator (BLUE). This is described by some K_k according to:

$$\hat{\boldsymbol{x}}_{k|k} = \hat{\boldsymbol{x}}_{k|k-1} + K_k(\boldsymbol{z}_k - \hat{\boldsymbol{z}}_k) \tag{A.5}$$

with

$$\hat{\boldsymbol{x}}_{k|k-1} = \mathrm{E}\left(f(\boldsymbol{x}_{k-1}, \boldsymbol{w}_k)\right) \hat{\boldsymbol{z}}_k = \mathrm{E}\left(h(\hat{\boldsymbol{x}}_{k-1}, \boldsymbol{v}_k)\right) \tag{A.6}$$

If one defines the measurement residual as

$$\tilde{\boldsymbol{z}}_k = \boldsymbol{z}_k - \hat{\boldsymbol{z}}_k \tag{A.7}$$

The state estimation error is the difference in reality and our estimation, given as:

$$\tilde{\boldsymbol{x}}_k = \boldsymbol{x}_k - \hat{\boldsymbol{x}}_{k|k} \tag{A.8}$$

$$\tilde{\boldsymbol{x}}_k = \boldsymbol{x}_k - \hat{\boldsymbol{x}}_{k|k-1} - K_k\tilde{\boldsymbol{z}}_k \tag{A.9}$$

For easier notation, we also define the prediction error:

$$\tilde{\boldsymbol{s}}_k = \hat{\boldsymbol{x}}_k - \hat{\boldsymbol{x}}_{k|k-1} \tag{A.10}$$

Then define the matrix $P_{k|k}$ to be the covariance of the state error (also referred to as state covariance):

$$P_{k|k} \triangleq \mathrm{E}\left((\tilde{\boldsymbol{x}}_k - \mathrm{E}(\tilde{\boldsymbol{x}}_k))(\tilde{\boldsymbol{x}}_k - \mathrm{E}(\tilde{\boldsymbol{x}}_k))^T\right) \tag{A.11}$$

by inserting our assumption (A.5) and our definition of prediction error (A.10) into our state covariance, we get:

$$P_{k|k} = \mathrm{E}\left((\tilde{\boldsymbol{s}}_k - K_k\tilde{\boldsymbol{z}}_k - \mathrm{E}(\tilde{\boldsymbol{s}}_k - K_k\tilde{\boldsymbol{z}}_k))(\tilde{\boldsymbol{s}}_k - K_k\tilde{\boldsymbol{z}}_k - \mathrm{E}(\tilde{\boldsymbol{s}}_k - K_k\tilde{\boldsymbol{z}}_k))^T\right)$$

$$P_{k|k} = \mathrm{E}\left((\tilde{\boldsymbol{s}}_k - \mathrm{E}(\tilde{\boldsymbol{s}}_k))(\tilde{\boldsymbol{s}}_k - \mathrm{E}(\tilde{\boldsymbol{s}}_k))^T\right) - \mathrm{E}\left((\tilde{\boldsymbol{s}}_k - \mathrm{E}(\tilde{\boldsymbol{s}}_k))(\tilde{\boldsymbol{z}}_k - \mathrm{E}(\tilde{\boldsymbol{z}}_k))^T\right)K_k^T$$

$$-K_k\mathrm{E}\left((\tilde{\boldsymbol{z}}_k - \mathrm{E}(\tilde{\boldsymbol{z}}_k))(\tilde{\boldsymbol{s}}_k - \mathrm{E}(\tilde{\boldsymbol{s}}_k))^T\right) + K_k\mathrm{E}\left((\tilde{\boldsymbol{z}}_k - \mathrm{E}(\tilde{\boldsymbol{z}}_k))(\tilde{\boldsymbol{z}}_k - \mathrm{E}(\tilde{\boldsymbol{z}}_k))^T\right)K_k^T$$

To reduce clutter, we define the prediction error covariance $P_{k|k-1}$, the prediction-observation covariance N_k and observation residual covariance (often called innovation residual) S_k:

$$\begin{aligned} P_{k|k-1} &= \mathrm{E}\left((\tilde{\boldsymbol{s}}_k - \mathrm{E}(\tilde{\boldsymbol{s}}_k))(\tilde{\boldsymbol{s}}_k - \mathrm{E}(\tilde{\boldsymbol{s}}_k))^T\right) \\ N_k &= \mathrm{E}\left((\tilde{\boldsymbol{s}}_k - \mathrm{E}(\tilde{\boldsymbol{s}}_k))(\tilde{\boldsymbol{z}}_k - \mathrm{E}(\tilde{\boldsymbol{z}}_k))^T\right) \\ S_k &= \mathrm{E}\left((\tilde{\boldsymbol{z}}_k - \mathrm{E}(\tilde{\boldsymbol{z}}_k))(\tilde{\boldsymbol{z}}_k - \mathrm{E}(\tilde{\boldsymbol{z}}_k))^T\right) \end{aligned} \tag{A.12}$$

This gives us a shorter expression for the covariance of the state error (A.11):

$$P_k = P_{k|k-1} - N_k K_k^T - K_k N_k^T + K_k S_k K_k^T$$

The optimal Kalman gain K_k attempts to minimize the mean square error of the state estimate. Given that we have formed unbiased estimates of the predicted state, a method of doing so would be to minimize the trace of P_k with respect to K_k. As P_k is the outer product of the state estimation error, the trace is representative of the inner product of the state estimation error. Minimizing the inner product of the state estimation error is to minimize the Euclidean distance of the state estimation error vector.

$$\frac{\partial \mathrm{trace}(P_k)}{\partial K_k} = -2N_k + 2K_k S_K = 0$$

The optimal Kalman gain is then:

$$K_k = N_k S_k^{-1} \tag{A.13}$$

The key here is noting that the optimal Kalman gain is defined separately from how the expectations (A.6) and covariances (A.12) are evaluated. This means that they can be evaluated by linearized versions of the transition function and measurement function, by unscented transformations, or other means. It also shows that the process and measurement noise does not have to be additive Gaussian for the Kalman gain to work.

A.3 Sigma-Point Selection

Sigma-points have to be selected according to a condition g to ensure that the desired information is captured (Julier, 1998). In this thesis, we described the same condition as Julier and Uhlmann (2002):

$$g(\mathcal{X}, w, p(\boldsymbol{x})) = \begin{bmatrix} \sum_{i=0}^{2L+1} w_i \\ \sum_{i=0}^{2L+1} w_i \mathcal{X}_i \\ \sum_{i=0}^{2L+1} w_i(\mathcal{X}_i - \mathrm{E}(\boldsymbol{x}))(\mathcal{X}_i - \mathrm{E}(\boldsymbol{x}))^T \end{bmatrix} - \begin{bmatrix} 1 \\ \mathrm{E}(\boldsymbol{x}) \\ \mathrm{Var}(\boldsymbol{x}) \end{bmatrix} = 0 \quad \text{(A.14)}$$

With sigma-points chosen according to:

$$\begin{aligned} \mathcal{X}_0 &= \mathrm{E}(\boldsymbol{x}) \\ \mathcal{X}_j &= \mathrm{E}(\boldsymbol{x}) + \sqrt{(L+\lambda)\mathrm{Var}(\boldsymbol{x})}_i \\ \mathcal{X}_k &= \mathrm{E}(\boldsymbol{x}) - \sqrt{(L+\lambda)\mathrm{Var}(\boldsymbol{x})}_i \\ w_0 &= \lambda/(L+\lambda) \\ w_i &= 1/2(L+\lambda) \end{aligned} \quad \text{(A.15)}$$

Where L is the dimension of the state vector $\boldsymbol{x}$. We first look at the weights:

$$\sum_{i=0}^{2L+1} w_i = \frac{\lambda}{(L+\lambda)} + \frac{L}{L+\lambda} = 1 \quad \text{(A.16)}$$

This means that the first part of the condition holds.

For the second part of the condition, evaluating expectation, we insert sigma-points according to the selection scheme (A.15):

$$\begin{aligned} \sum_{i=0}^{2L+1} w_i \mathcal{X}_i &= w_0 \mathrm{E}(\boldsymbol{x}) + \sum_{i=1}^{L} w_i \left(\mathrm{E}(\boldsymbol{x}) + \sqrt{(L+\lambda)\mathrm{Var}(\boldsymbol{x})}_i \right) \\ &\quad + \sum_{j=L+1}^{2L+1} w_j \left(\mathrm{E}(\boldsymbol{x}) - \sqrt{(L+\lambda)\mathrm{Var}(\boldsymbol{x})}_{j-L} \right) \\ &= \mathrm{E}(\boldsymbol{x}) \sum_{i=0}^{2L+1} w_i = \mathrm{E}(\boldsymbol{x}) \end{aligned} \quad \text{(A.17)}$$

This means that the second part of the condition holds given that the first condition holds.

For the third part, evaluating variance, we insert sigma-points and the weights according to the selection scheme (A.15):

$$\begin{aligned}
&\sum_{i=0}^{2L+1} w_i(\mathcal{X}_i - \mathrm{E}(\boldsymbol{x}))(\mathcal{X}_i - \mathrm{E}(\boldsymbol{x})) = \\
&\qquad \sum_{i=1}^{L} \frac{1}{2(L+\lambda)}(\sqrt{(L+\lambda)\mathrm{Var}(\boldsymbol{x})}_i)(\sqrt{(L+\lambda)\mathrm{Var}(\boldsymbol{x})}_i)^T \\
&\qquad + \sum_{j=L+1}^{2L+1} \frac{1}{2(L+\lambda)}(\sqrt{(L+\lambda)\mathrm{Var}(\boldsymbol{x})}_j)(\sqrt{(L+\lambda)\mathrm{Var}(\boldsymbol{x})}_j)^T \\
&\qquad = \frac{1}{2}\mathrm{Var}(\boldsymbol{x}) + \frac{1}{2}\mathrm{Var}(\boldsymbol{x}) = \mathrm{Var}(\boldsymbol{x})
\end{aligned} \tag{A.18}$$

This means that the third part of the condition holds.

Hence for the sigma-point selection scheme (A.15), the condition g holds. This means that if the condition g is sufficient to capture the expectation and variance of the random variable in terms of sigma-points (which it is according to Julier and Uhlmann 2002) , the sigma-point selection scheme (A.15) is a valid selection scheme for that condition.

A.4 Additional Graphs: Effect of VO Frequency

This section contains the pose with respect to time of the simulations investigating the effect of VO frequency, both with and without yaw covariance threshold.

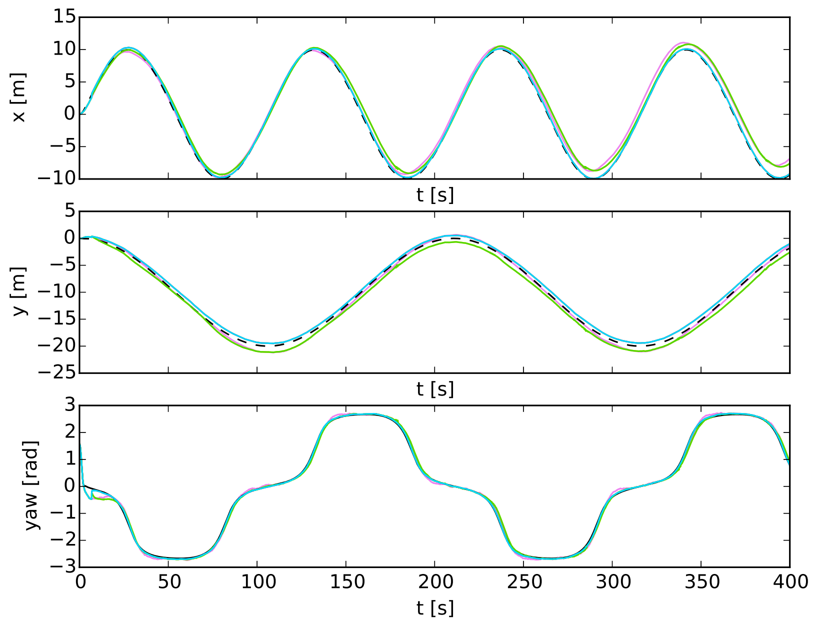

Figure 57: *Simulation of Asterix performing the figure eight movement with VO frequency $M = 8$ and VO delay $N = 10$. Pose with respect to time.*

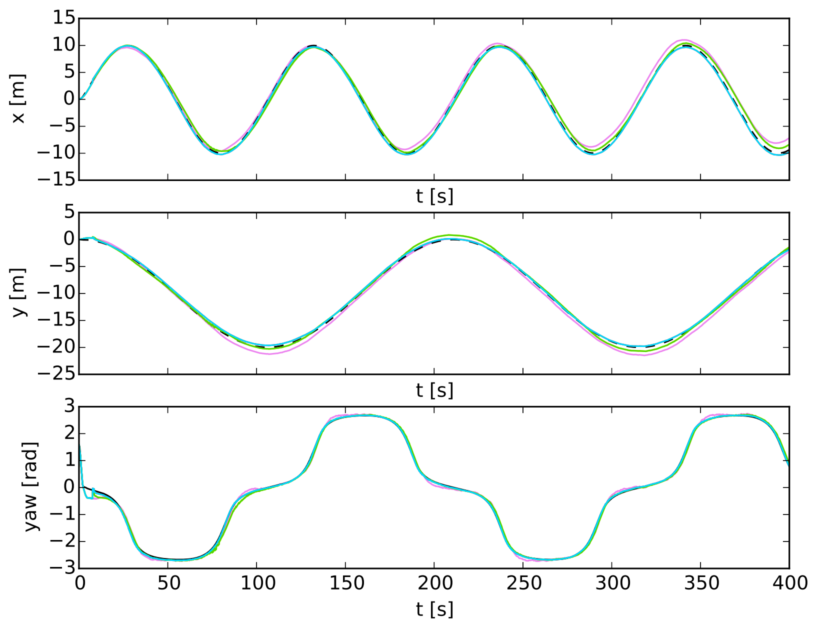

Figure 58: *Simulation of Asterix performing the figure eight movement with VO frequency $M = 16$ and VO delay $N = 10$. Pose with respect to time.*

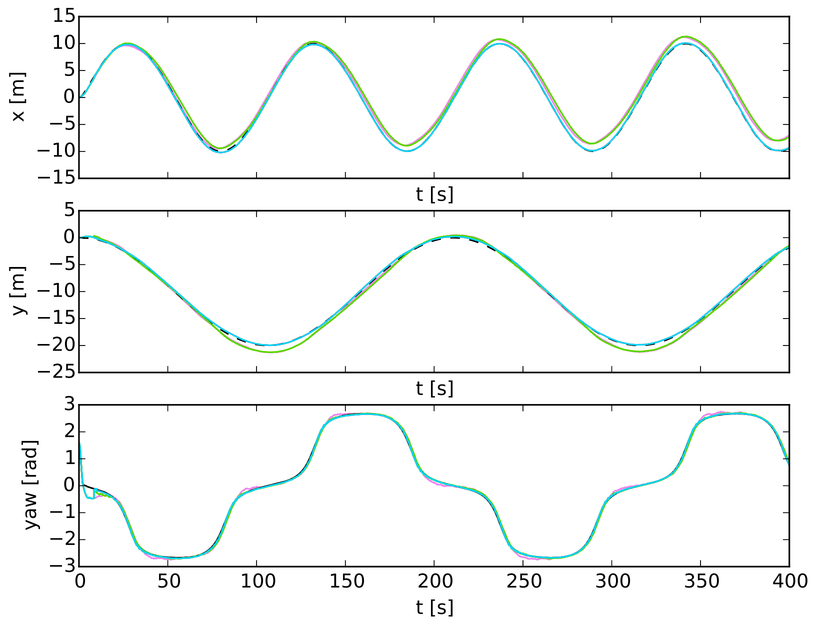

Figure 59: *Simulation of Asterix performing the figure eight movement with VO frequency $M = 32$ and VO delay $N = 10$. Pose with respect to time.*

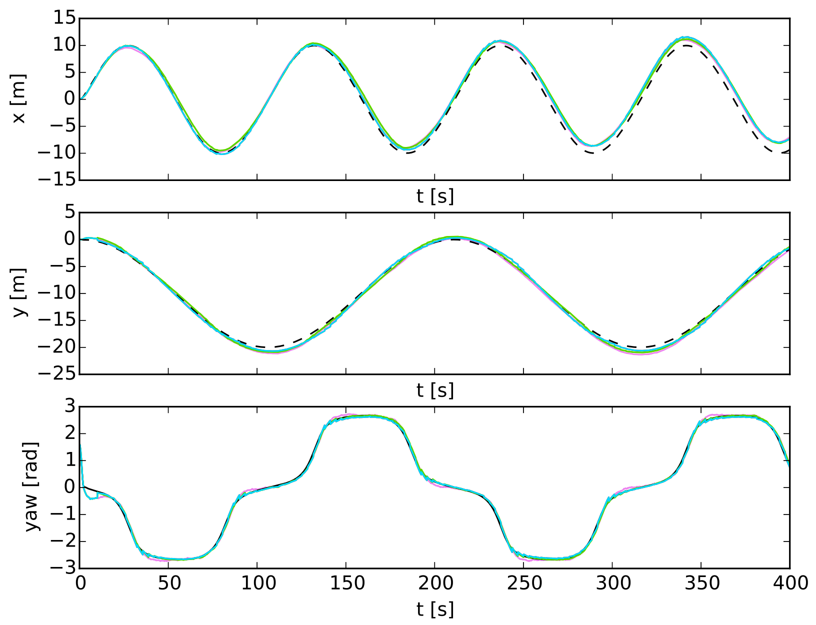

Figure 60: *Simulation of Asterix performing the figure eight movement with VO frequency* $M = 64$ *and VO delay* $N = 10$*. Pose with respect to time.*

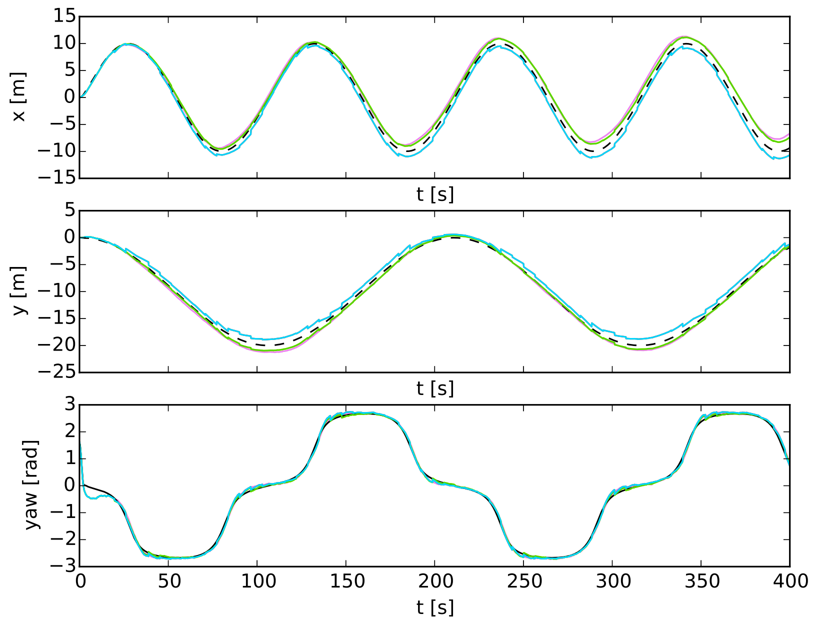

Figure 61: *Simulation of Asterix performing the figure eight movement with VO frequency* $M = 128$ *and VO delay* $N = 10$*. Pose with respect to time.*

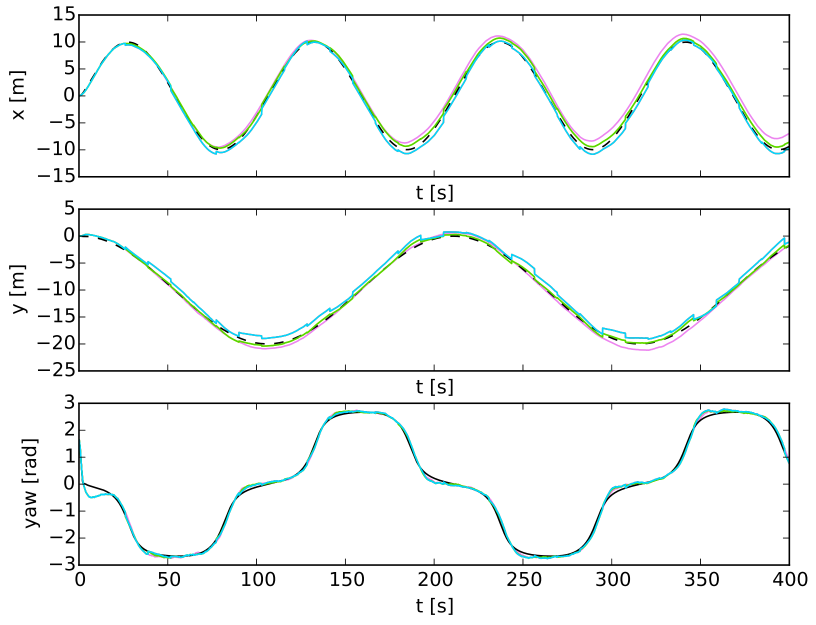

Figure 62: *Simulation of Asterix performing the figure eight movement with VO frequency $M = 256$ and VO delay $N = 10$. Pose with respect to time.*

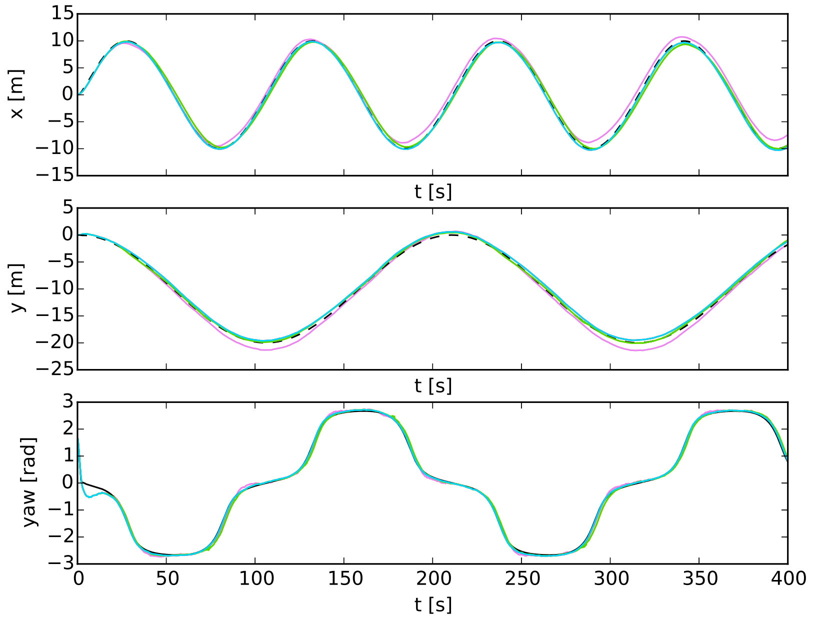

Figure 63: *Simulation of Asterix performing the figure eight movement with VO frequency $M = 8$, VO delay $N = 10$, and yaw covariance threshold. Pose with respect to time.*

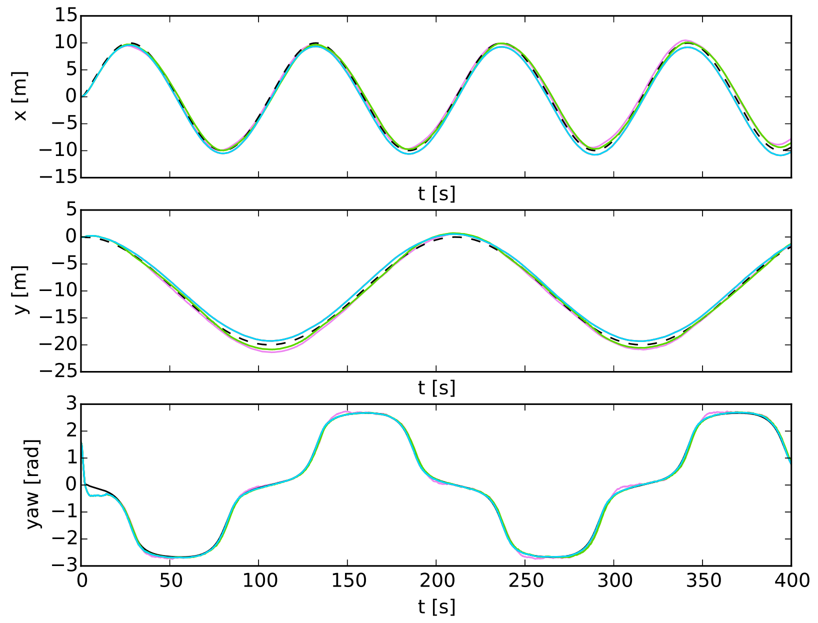

Figure 64: *Simulation of Asterix performing the figure eight movement with VO frequency $M = 16$, VO delay $N = 10$, and yaw covariance threshold. Pose with respect to time.*

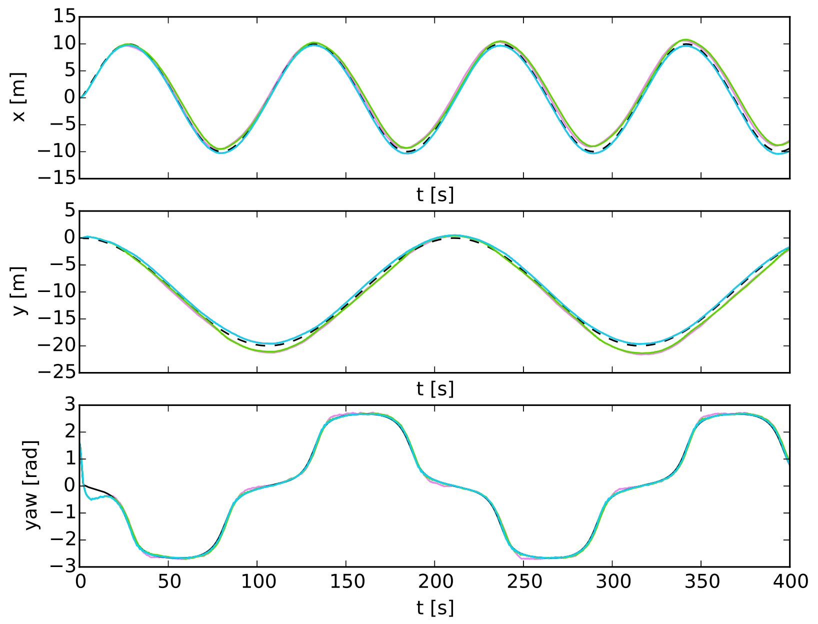

Figure 65: *Simulation of Asterix performing the figure eight movement with VO frequency $M = 32$, VO delay $N = 10$, and yaw covariance threshold. Pose with respect to time.*

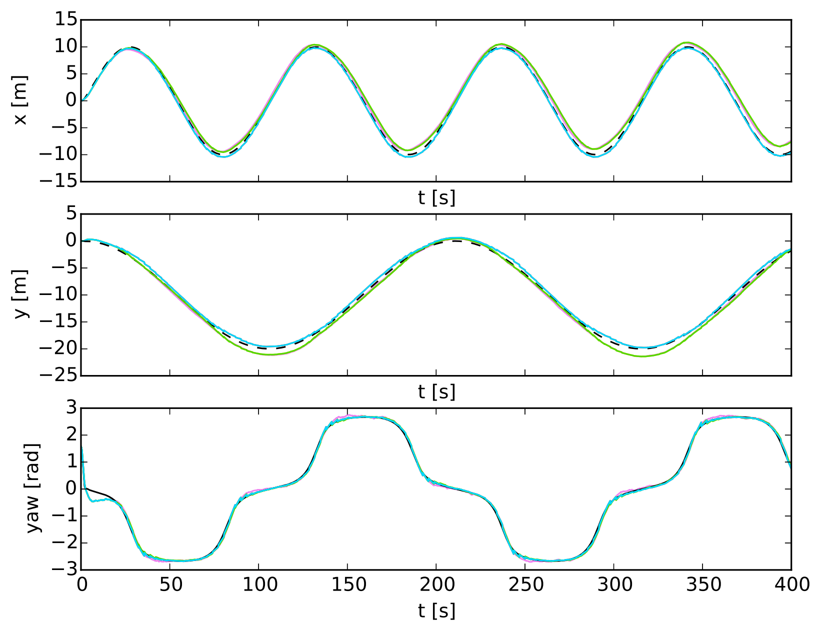

Figure 66: *Simulation of Asterix performing the figure eight movement with VO frequency $M = 64$, VO delay $N = 10$, and yaw covariance threshold. Pose with respect to time.*

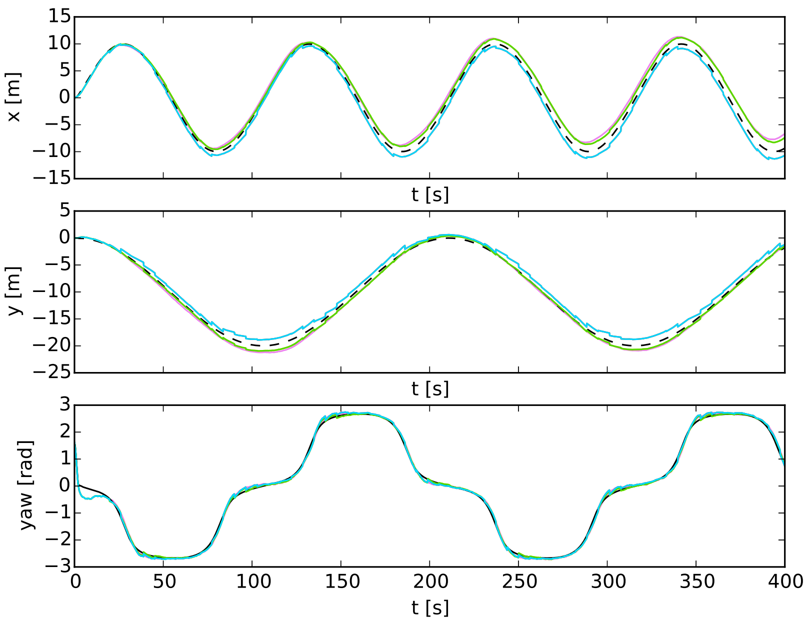

Figure 67: *Simulation of Asterix performing the figure eight movement with VO frequency $M = 128$, VO delay $N = 10$, and yaw covariance threshold. Pose with respect to time.*

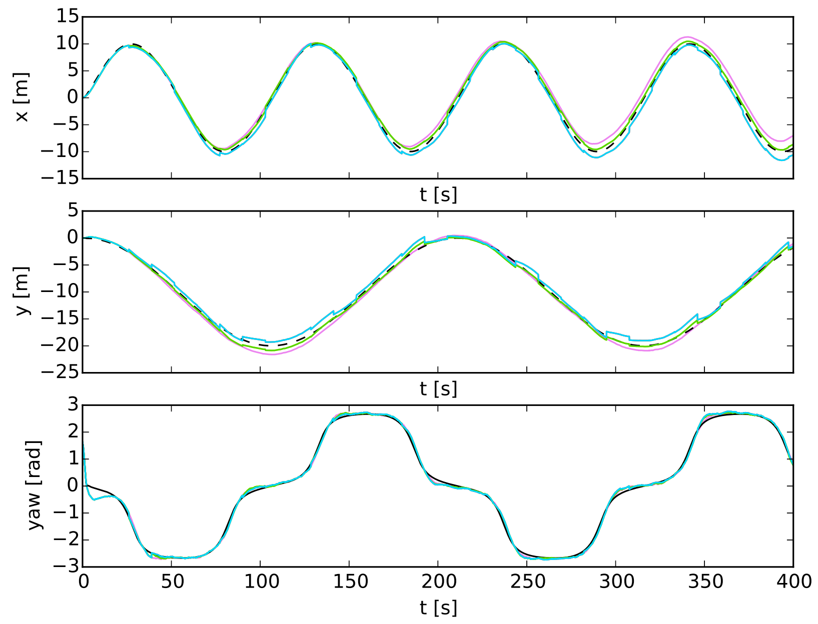

Figure 68: *Simulation of Asterix performing the figure eight movement with VO frequency $M = 256$, VO delay $N = 10$, and yaw covariance threshold. Pose with respect to time.*

A.5 Modules

A.5.1 ProbabilisticFilters

```python
import types
import numpy as np
import numpy.linalg as linalg
import random
import scipy.sparse

class Filter(object):
    '''Generalized Filter class.'''

    def __init__(self, state_size = 2):
        self.states = np.asmatrix(np.zeros( (state_size, 1)
            ↪  ))
        self.I = np.asmatrix(np.identity( state_size ))
        self.state_size = state_size
    def __print__(self):
        return str(self.states)


class KalmanFilter(Filter):
    '''Kalman filter. Todo: fix input.'''
    def setInitialDistribution(self, states,
        ↪ state_covariance):
        self.states = states
        self.state_covariance = state_covariance

    def setTransitionModel(self, transition_matrix,
        ↪ input_matrix, transition_covariance):
        self.transition_matrix = transition_matrix
        self.input_matrix = input_matrix
        self.input_size = input_matrix.shape[1]
        self.transition_covariance = transition_covariance

    def setMeasurementModel(self, measurement_matrix,
        ↪ measurement_covariance):
        self.measurement_matrix = measurement_matrix
        self.measurement_covariance =
            ↪ measurement_covariance
        self.measurement_size = measurement_matrix.shape[0]

    def predict(self, transition_covariance=None):
        if transition_covariance is None:
            transition_covariance = self.
                ↪ transition_covariance
        self.states = self.transition_matrix*self.states
        self.state_covariance = self.transition_matrix*self
            ↪ .state_covariance*self.transition_matrix.T +
            ↪  transition_covariance

    def update(self, measurement, measurement_covariance=None):
        if measurement_covariance is None:
            measurement_covariance = self.measurement_covariance
        residual = measurement - self.measurement_matrix*self.states
        residual_covariance = self.measurement_matrix*self.state_covariance*self.measurement_matrix.T + measurement_covariance
        if not scipy.sparse.issparse(residual_covariance):
            kalman_gain = (self.state_covariance*self.measurement_matrix.T)*linalg.inv(residual_covariance)
        else:
            kalman_gain = (self.state_covariance*self.measurement_matrix.T)*scipy.sparse.linalg.inv(residual_covariance)
        self.states = self.states + kalman_gain*residual
        KP = (np.identity(self.states.size) - kalman_gain*self.measurement_matrix)
        self.state_covariance = KP*self.state_covariance*KP.T + kalman_gain*measurement_covariance*kalman_gain.T
        #Numerical stability hack
        self.state_covariance = 0.5*(self.state_covariance.T + self.state_covariance)

    def iterate(self, measurement, transition_covariance=None, measurement_covariance=None):
        self.predict(transition_covariance)
        self.update(measurement, measurement_covariance)


class ExtendedKalmanFilter(KalmanFilter):
    '''Extended kalman filter. No support for input.'''
    def setTransitionModel(self, transition_function, transition_jacobian, transition_covariance):
        self.transitionFunction = transition_function
        self.transitionJacobian = transition_jacobian
        self.transition_covariance = transition_covariance

    def setMeasurementModel(self, measurement_function, measurement_jacobian, measurement_covariance):
        self.measurementFunction = measurement_function
        self.measurementJacobian = measurement_jacobian
        self.measurement_covariance = measurement_covariance

    def predict(self, transition_covariance = None):
        if transition_covariance is None:
            transition_covariance = self.transition_covariance
        F = self.transitionJacobian(self.states)
        self.states = self.transitionFunction(self.states)
        self.state_covariance = F*self.state_covariance*F.T + transition_covariance

    def update(self,measurement, measurement_covariance=None):
        if measurement_covariance is None:
            measurement_covariance = self.measurement_covariance
        residual = measurement - self.measurementFunction(self.states)
        H = self.measurementJacobian(self.states)
        residual_covariance = H*self.state_covariance*H.T + measurement_covariance
        if not scipy.sparse.issparse(residual_covariance):
            kalman_gain = (self.state_covariance*H.T)*linalg.inv(residual_covariance)
        else:
            kalman_gain = (self.state_covariance*H.T)*scipy.sparse.linalg.inv(residual_covariance)
        self.states = self.states + kalman_gain*residual
        KP = (np.identity(self.states.size)  - kalman_gain*H)
        self.state_covariance = KP*self.state_covariance*KP.T + kalman_gain*measurement_covariance*kalman_gain.T
        #Numerical stability hack
        self.state_covariance = 0.5*(self.state_covariance + self.state_covariance.T)


class UnscentedKalmanFilter(KalmanFilter):
    '''Unscented kalman filter. No support for input.'''
    def __init__(self, state_size = 2, alpha = 1e-3, kappa = 0, beta = 2):
        super(UnscentedKalmanFilter, self).__init__(state_size)
        self.alpha = alpha
        self.kappa = kappa
        self.beta = beta

    def sigmaPoints(self, mean=None, covariance_matrix=None, scaling_factor=None):
        if mean is None:
            mean = self.states
        if covariance_matrix is None:
            covariance_matrix = self.state_covariance
        if scaling_factor is None:
            scaling_factor = self.scaling_factor
        root_matrix = np.sqrt(scaling_factor)*linalg.cholesky(covariance_matrix)
        repeated_mean = np.repeat(mean, mean.size, axis = 1)
        sigma_points = np.concatenate( (mean, repeated_mean + root_matrix, repeated_mean - root_matrix), axis = 1)
        return sigma_points

    def unscentedTransform(self, transition_function=None, sigma_points=None, mean_weights=None, covariance_weights=None, additive_covariance=None):
        if transition_function is None:
            transition_function = self.transitionFunction
        if sigma_points is None:
            sigma_points = self.sigmaPoints()
        if mean_weights is None:
            mean_weights = self.mean_weights
        if covariance_weights is None:
            covariance_weights = self.covariance_weights
        if additive_covariance is None:
            additive_covariance = self.transition_covariance
        transformed_size = additive_covariance.shape[0]
        sigma_size = sigma_points.shape[1]
        sigma_points_t = np.asmatrix(np.zeros( (transformed_size, sigma_size) ))
        for i in xrange(sigma_size):
            sigma_points_t[:,i] = transition_function(sigma_points[:,i])
        if mean_weights.shape[0] < mean_weights.shape[1]:
            mean_t = sigma_points_t * mean_weights.T
        else:
            mean_t = sigma_points_t * mean_weights
        transformed_deviation = sigma_points_t - np.repeat(mean_t, sigma_size, axis = 1)
        covariance_t = transformed_deviation*covariance_weights*transformed_deviation.T + additive_covariance
        return (mean_t, covariance_t, sigma_points_t, transformed_deviation)

    def setParameters(self, alpha = None, kappa = None, beta = None):
        '''Set parameters of the unscented transformation. Default values are for true distribution being gaussian.'''
        if alpha is None:
            alpha = self.alpha
        if kappa is None:
            kappa = self.kappa
        if beta is None:
            beta = self.beta
        lam = alpha**2 * (self.states.size + kappa) - self.states.size
        self.scaling_factor = self.states.size + lam
        mean_weight0 = [lam/self.scaling_factor]
        mean_weightsi = [1/(2*self.scaling_factor) for i in range(2*self.states.size)]
        covariance_weight0 = [lam/self.scaling_factor + (1 - alpha**2 + beta)]
        covariance_weightsi = [1/(2*self.scaling_factor) for i in range(2*self.states.size)]
        self.mean_weights = np.asmatrix(mean_weight0 + mean_weightsi)
        self.covariance_weights = np.asmatrix(np.diagflat(covariance_weight0 + covariance_weightsi))

    def setTransitionModel(self, transition_function, transition_covariance):
        '''Transition function x_{k+1} = f(x_k) + w_k, w_k ~ N(0, transition_covariance).'''
        self.transitionFunction = transition_function
        self.transition_covariance = transition_covariance

    def setMeasurementModel(self, measurement_function, measurement_covariance):
        '''Measurement function z_k = h(x_k) + v_k, v_k ~ N(0,measurement_covariance).'''
        self.measurementFunction = measurement_function
        self.measurement_covariance = measurement_covariance
        self.measurement_size = measurement_covariance.shape[0]

    def predict(self,transition_covariance=None):
        if transition_covariance is None:
            transition_covariance = self.transition_covariance
        if self.states.size is not int((self.mean_weights.size-1)/2):
            self.setParameters() #Someone changed state size without updating parameters
        utres = self.unscentedTransform(additive_covariance = transition_covariance)
        self.states = utres[0]
        self.state_covariance = utres[1]

    def update(self, measurement, measurement_covariance=None):
        if measurement_covariance is None:
            measurement_covariance = self.measurement_covariance
        utres = self.unscentedTransform(transition_function=self.measurementFunction, additive_covariance = measurement_covariance)
        expected_measurement = utres[0]
        measurement_sigma = utres[2]
        measurement_deviation = utres[3]
        residual = measurement - expected_measurement
        residual_covariance = measurement_deviation*self.covariance_weights*measurement_deviation.T + measurement_covariance
        state_deviation = self.sigmaPoints() - np.repeat(self.states, measurement_sigma.shape[1], axis = 1)
        state_to_measurement_covariance = state_deviation*self.covariance_weights*measurement_deviation.T
        if not scipy.sparse.issparse(residual_covariance):
            kalman_gain = state_to_measurement_covariance*linalg.inv(residual_covariance)
        else:
            kalman_gain = state_to_measurement_covariance*scipy.sparse.linalg.inv(residual_covariance)
        self.states = self.states + kalman_gain*residual
        self.state_covariance = self.state_covariance - kalman_gain*residual_covariance*kalman_gain.T
        #Numerical stability hack
        self.state_covariance = (self.state_covariance.T + self.state_covariance)*0.5


class ParticleFilter(Filter):
    '''Particle filter based on Thrun. Not at all efficient , or useable for large scale systems.'''
    def __init__(self, state_size = 2, number_of_particles =50):
        super(ParticleFilter, self).__init__(state_size)
        self.number_of_particles = number_of_particles
        self.samples = np.asmatrix(np.zeros( (state_size, number_of_particles) ))
        self.weights = [1/number_of_particles]*number_of_particles
        self.getStates()

    def getStates(self):
        self.states = np.mean(self.samples, axis = 1)

    def generateWeights(self, measurement):
        self.weights = [1/self.number_of_particles] * self.number_of_particles
        for i in xrange(self.number_of_particles):
            self.weights[i] = self.measurementFunction(self.samples[:,i], measurement)

    def resample(self, number_of_samples = None):
        '''Sample particles based on the weights. Algorithm taken from Thrun.'''
        if number_of_samples == None:
            number_of_samples = self.number_of_particles
        index = int(random.random() * number_of_samples)
```

```python
        max_weight = max(self.weights)
        beta = 0.0
        resampled_indices = []
        for i in xrange(number_of_samples):
            beta += random.random()*2*max_weight
            while beta > self.weights[index]:
                beta -= self.weights[index]
                index = (index + 1) % self.number_of_particles
            resampled_indices.append(index)
        new_samples = np.asmatrix(np.zeros( (self.state_size, number_of_samples) ))
        for i, prev_index in enumerate(resampled_indices):
            new_samples[:,i] = self.samples[:,prev_index]
        self.samples = new_samples

    def setInitialDistribution(self, initialization_function):
        '''Initial sample distribution based on p(x_0)'''
        self.initializationFunction = initialization_function
        for i in xrange(self.number_of_particles):
            self.samples[:,i] = self.initializationFunction()
        self.getStates()

    def setTransitionModel(self, transition_function):
        '''transition function based on p(x_{k+1} | x_k).'''
        self.transitionFunction = transition_function

    def setMeasurementModel(self, measurement_function):
        ''' measurement function based on p(z_k | x_k).'''
        self.measurementFunction = measurement_function

    def predict(self):
        for i in xrange(self.number_of_particles):
            self.samples[:,i] = self.transitionFunction(self.samples[:,i])

    def update(self, measurement):
        self.generateWeights(measurement)
        self.resample()

    def iterate(self, measurement):
        self.predict()
        self.update(measurement)
        self.getStates()

def maskedUpdate(self, measurement, measurement_covariance = None):
    '''Masked update equation, based on previous update. Lower performance than the actual as of yet.'''
    old_states = self.states
    old_state_covariance = self.state_covariance
    self.update_pure(measurement, measurement_covariance)
    self.states[:self.mask_size] = old_states[:self.mask_size]
    self.state_covariance[:self.mask_size, :self.mask_size] = old_state_covariance[:self.mask_size, :self.mask_size]

def CreateMasked(FilterClass, state_size = 2, mask_size = 0):
    '''Creates a masked version of a given filter. States[:mask_size] remains the same.State covariance[:mask_size] also remain the same.'''
    supported_filters = ["KalmanFilter","ExtendedKalmanFilter", "UnscentedKalmanFilter"]
    if FilterClass.__name__ not in supported_filters:
        raise TypeError("Chosen filter not supported for masking.")
    masked_filter = FilterClass(state_size)
    masked_filter.mask_size = mask_size
    masked_filter.update_pure = masked_filter.update
    masked_filter.update = types.MethodType(maskedUpdate, masked_filter)
    return masked_filter
```

A.5.2 DelayFusion

```python
import types
import numpy as np
from probabilistic_filters import *

def patchFilter(dfilter):
    '''Patch the given filter with functions necessary for OOSM fusion. Gives both SC and J methods.'''
    kalman_types = ["ExtendedKalmanFilter", "UnscentedKalmanFilter", "KalmanFilter"]
    supported_filters = kalman_types
    filter_patch = { "KalmanFilter" : KFPatch,
                     "ExtendedKalmanFilter" : EKFPatch,
                     "UnscentedKalmanFilter" : UKFPatch}
    filtername = dfilter.__class__.__name__

    if filtername not in filter_patch.keys():
        raise TypeError("Chosen filtertype is not supported for delay_fusion. Supported filter classes are:\n{}".format(supported_filters))

    #patch appropriate filter with appropriate functions
    filter_patch[filtername](dfilter)
    return True

def KFPatch(kfilter):
    kfilter.epsilon = 1e-10
    kfilter.cloned_states = 0
    kfilter.extend = types.MethodType(_extendKF, kfilter)
    kfilter.clone = types.MethodType(_cloneKF, kfilter)
    kfilter.marginalize = types.MethodType(_marginalizeKF, kfilter)

def EKFPatch(ekfilter):
    ekfilter.epsilon = 1e-10
    ekfilter.cloned_states = 0
    ekfilter.extend = types.MethodType(_extendEKF, ekfilter )
    ekfilter.clone = types.MethodType(_clone, ekfilter)
    ekfilter.marginalize = types.MethodType(_marginalize, ekfilter)

def UKFPatch(ukfilter):
    ukfilter.epsilon =1e-10
    ukfilter.cloned_states = 0
    ukfilter.extend = types.MethodType(_extendUKF, ukfilter )
    ukfilter.clone = types.MethodType(_clone, ukfilter)
    ukfilter.marginalize = types.MethodType(_marginalize, ukfilter)

def _clone(self):
    state_size = self.state_size
    cloned_states = self.cloned_states
    self.states = np.vstack((self.states, self.states[-state_size:]))
    state_cov_side = self.state_covariance[:,-state_size:]
    self.transition_covariance = np.bmat([[np.zeros( (state_size, state_size) ), np.zeros( (state_size, state_size*(1 + cloned_states)) )], [np.zeros( (state_size*(1 + cloned_states), state_size ) ), self.transition_covariance]])
    self.state_covariance = np.bmat([[self.state_covariance, state_cov_side], [state_cov_side.T, self.state_covariance[-state_size:,-state_size:] + self.epsilon*self.I]]) #+ self.epsilon*np.identity(state_size*(1+cloned_states))
    self.cloned_states += 1

def _marginalize(self):
    state_size = self.state_size
    if self.cloned_states > 0:
        self.states = self.states[state_size:]
        self.transition_covariance = self.transition_covariance[state_size:, state_size:]
        self.state_covariance = self.state_covariance[state_size:, state_size:]
        self.cloned_states -= 1

def _extendKF(self):
    state_size = self.state_size
    cloned_states = self.cloned_states
    self.states = np.vstack((self.states, self.transition_matrix*self.states[-state_size:]))
    self.transition_covariance = np.bmat([[np.zeros( (state_size, state_size) ), np.zeros( (state_size, state_size*(1 + cloned_states)) )], [np.zeros( (state_size*(1 + cloned_states), state_size ) ), self.transition_covariance]])

    self.transition_matrix = np.bmat([[self.I, np.zeros( (state_size, state_size*(1+cloned_states)) )], [np.zeros( (state_size*(1+cloned_states), state_size) ), self.transition_matrix]])
    state_cov_side = self.state_covariance[:, -state_size:]
    self.state_covariance = np.bmat([[ self.state_covariance, state_cov_side], [state_cov_side.T, self.state_covariance[-state_size:, -state_size:]]])
    self.state_covariance = self.transition_matrix* self.state_covariance * self.transition_matrix.T + self.transition_covariance
    self.cloned_states += 1

def _extendEKF(self):
    self.epsilon = 0
    self.clone()
    self.predict()

def _extendUKF(self):
    self.setParameters()
    state_size = self.state_size
    cloned_states = self.cloned_states
    sigma_points = self.sigmaPoints()
    sigma_size = sigma_points.shape[1]
    states = sigma_points*self.mean_weights.T
    unscented_res = self.unscentedTransform()
    deviations = (sigma_points - np.repeat(states, sigma_size, axis = 1))
    deviations_t = unscented_res[3]
    temporary_covariance = deviations*self.covariance_weights*deviations_t.T
    state_cov_side = temporary_covariance[:,-state_size:]
    covariance_t = unscented_res[1]
    states_t = unscented_res[0]

    self.state_covariance = np.bmat([[self.state_covariance, state_cov_side],
                                     [state_cov_side.T, covariance_t[-state_size:,-state_size:]]])
    self.transition_covariance = self.transition_covariance = np.bmat([[np.zeros( (state_size, state_size)
        ↪ ), np.zeros( (state_size, state_size*(1 +
        ↪ cloned_states)) )], [np.zeros( (state_size*(1 +
        ↪ cloned_states), state_size ) ), self.
        ↪ transition_covariance]])
    self.states = np.vstack( (self.states, states_t[-
        ↪ state_size:]) )
    #Numerical compensation:
    self.state_covariance = 0.5*(self.state_covariance +
        ↪ self.state_covariance.T)
    self.cloned_states += 1
    self.setParameters()

def _cloneKF(self):
    state_size = self.state_size
    cloned_states = self.cloned_states
    self.state = np.vstack((self.states, self.states[-
        ↪ state_size:]))
    state_cov_side = self.state_covariance[:,-state_size:]
    self.transition_covariance = np.bmat([[np.zeros( (
        ↪ state_size, state_size) ), np.zeros( (state_size
        ↪ , state_size*(1 + cloned_states)) )], [np.zeros(
        ↪  (state_size*(1 + cloned_states), state_size ) )
        ↪ , self.transition_covariance]])
    self.state_covariance = np.bmat([[self.state_covariance
        ↪  + self.epsilon*self.I, state_cov_side], [
        ↪ state_cov_side.T, self.state_covariance + self.
        ↪ epsilon*self.I]])
    self.transition_matrix = np.bmat([[self.I, np.zeros( (
        ↪ state_size, state_size*(cloned_states + 1)) )],
                                      [np.zeros( (
                                          ↪ state_size,
                                          ↪ state_size*(
                                          ↪ cloned_states
                                          ↪ + 1))), self.
                                          ↪ transition_matrix
                                          ↪ ]])
    self.measurement_matrix = np.bmat([[np.zeros(self.
        ↪ measurement_matrix.shape), self.
        ↪ measurement_matrix]])
    self.cloned_states += 1

def _marginalizeKF(self):
    state_size = self.state_size
    self.states = self.states[state_size:]
    self.transition_covariance = self.transition_covariance
        ↪ [state_size:, state_size:]
    self.state_covariance = self.state_covariance[
        ↪ state_size:, state_size:]
    self.measurement_matrix = self.measurement_matrix
    self.cloned_states -= 1
```

A.5.3 Kinematic

```python
from probabilistic_filters import *
import types
import numpy as np

#Global state indices
X_IND = 0
Y_IND = 1
YAW_IND = 2
X_VEL_IND = 3
YAW_RATE_IND = 4

#Robot model from De La Cruz:
def _differentialFunction(self,states):
    '''dx/dt = f(x)'''
    fx = states[X_VEL_IND]*np.cos(states[YAW_IND]) - self.a*states[YAW_RATE_IND]*np.sin(states[YAW_IND])
    fy = states[X_VEL_IND]*np.sin(states[YAW_IND]) + self.a*states[YAW_RATE_IND]*np.cos(states[YAW_IND])
    fpsi = states[YAW_RATE_IND]
    states = np.vstack( (fx, fy, fpsi, 0, 0) )
    return states

def _differentialJacobian(self,states):
    '''df(x)/dx'''
    state_size = states.size
    jacobian = np.asmatrix(np.zeros( (state_size, state_size) ))
    jacobian[X_IND, YAW_IND] = -states[X_VEL_IND]*np.sin(states[YAW_IND]) - self.a*states[YAW_RATE_IND]*np.cos(states[YAW_IND])
    jacobian[X_IND, X_VEL_IND] = np.cos(states[YAW_IND])
    jacobian[X_IND, YAW_RATE_IND] = -self.a*np.sin(states[YAW_IND])
    jacobian[Y_IND, YAW_IND] = states[X_VEL_IND]*np.cos(states[YAW_IND]) - self.a*states[YAW_RATE_IND]*np.sin(states[YAW_IND])
    jacobian[Y_IND, X_VEL_IND] = np.sin(states[YAW_IND])
    jacobian[Y_IND, YAW_RATE_IND] = self.a*np.cos(states[YAW_IND])
    jacobian[YAW_IND, YAW_RATE_IND] = 1
    return jacobian

def _transitionFunction(self, states, timestep = None):
    '''Eulers method. With cloned states.'''
    if timestep is None:
        timestep = self.timestep
    states[-self.state_size:] = states[-self.state_size:] + timestep*self.differentialFunction(states[-self.state_size:])
    return states

def _transitionJacobian(self, states, timestep = None):
    '''Transition jacobian using Eulers method. With cloned states.'''
    if timestep is None:
        timestep = self.timestep
    if self.cloned_states is 0:
        return self.I + timestep*self.differentialJacobian(
           ↪ states)
    else:
        differential_jacobian = np.identity(self.state_size
           ↪ *(1+self.cloned_states))
        differential_jacobian[-self.state_size:, -self.
           ↪ state_size:] += self.differentialJacobian(
           ↪ states[-self.state_size:])*timestep
        return differential_jacobian

def patchFilter(tfilter):
    '''Patch the given filter with functions necessary for
       ↪ 2D asterix localization.'''
    kalman_classes = ["ExtendedKalmanFilter", "
       ↪ UnscentedKalmanFilter"]
    supported_classes = kalman_classes
    filter_name = tfilter.__class__.__name__
    filter_patch = {"ExtendedKalmanFilter": EKFPatch,
                    "UnscentedKalmanFilter": UKFPatch}
    if filter_name not in filter_patch.keys():
        raise TypeError("Chosen filter class is not
           ↪ supported. Supported types are: {}".format(
           ↪ supported_filters))

    filter_patch[filter_name](tfilter)
    return True

def EKFPatch(tfilter):
    tfilter.transitionFunction = types.MethodType(
       ↪ _transitionFunction, tfilter)
    tfilter.transitionJacobian = types.MethodType(
       ↪ _transitionJacobian, tfilter)
    tfilter.differentialFunction = types.MethodType(
       ↪ _differentialFunction, tfilter)
    tfilter.differentialJacobian = types.MethodType(
       ↪ _differentialJacobian,tfilter)
    tfilter.cloned_states = 0
    tfilter.a = 0.2

def UKFPatch(tfilter):
    tfilter.transitionFunction = types.MethodType(
       ↪ _transitionFunction, tfilter)
    tfilter.differentialFunction = types.MethodType(
       ↪ _differentialFunction, tfilter)
    tfilter.cloned_states = 0
    tfilter.a = 0.2
```

A.5.4 Geodetic

A module for conversion from elliptical to ENU.

```python
'''Contains functions necessary for converting geodetic
    ↪ coordinates.
By Mathias Hauan Arbo, 24.04.2015
Based on Wikipedia and matlabs geodetic package.'''
import math

def referenceEllipsoid(ellipse_type):
    '''Takes an ellipse type and returns the ellipsoid
        ↪ parameters.'''
    if (ellipse_type=='CLK66' or ellipse_type=='NAD27'):
        major_axis=6378206.4;
        finv=294.9786982;
    elif ellipse_type=='GRS67':
        major_axis=6378160.0;
        finv=298.247167427;
    elif (ellipse_type=='GRS80' or ellipse_type=='NAD83'):
        major_axis=6378137.0;
        finv=298.257222101;
    elif (ellipse_type=='WGS72'):
        major_axis=6378135.0;
        finv=298.26;
    elif (ellipse_type=='WGS84'):
        major_axis=6378137.0;
        finv=298.257223563;
    elif ellipse_type=='ATS77':
        major_axis=6378135.0;
        finv=298.257;
    elif ellipse_type=='KRASS':
        major_axis=6378245.0;
        finv=298.3;
    elif ellipse_type=='INTER':
        major_axis=6378388.0;
        finv=297.0;
    elif ellipse_type=='MAIRY':
        major_axis=6377340.189;
        finv=299.3249646;
    elif ellipse_type=='TOPEX':
        major_axis=6378136.3;
        finv=298.257;
    flattening=1/finv;
    minor_axis = major_axis*(1-flattening);
    eccentricity_squared = 1-(1-flattening)**2;
    return major_axis, minor_axis, eccentricity_squared,
        ↪ flattening

def geodetic2ecef(latitude, longitude, altitude,
    ↪ ellipse_type = "WGS84"):
    '''Converts geodetic coordinates (latitude, longitude,
        ↪ altitude) to ECEF coordinates (x,y,z).'''
    major_axis, minor_axis, eccentricity_squared,
        ↪ flattening = referenceEllipsoid(ellipse_type)
    #normal = major_axis / (math.sqrt(1 -
        ↪ eccentricity_squared *math.sin(latitude)**2))
    normal = (major_axis**2)/math.sqrt((math.cos(latitude)**2)*(major_axis**2) + (minor_axis**2)*math.sin(latitude)**2)
    x_ecef = (normal+altitude)*math.cos(latitude)*math.cos(longitude)
    y_ecef = (normal+altitude)*math.cos(latitude)*math.sin(longitude)
    z_ecef = (normal*(1 - eccentricity_squared) + altitude)*math.sin(latitude)
    return x_ecef, y_ecef, z_ecef

def ecef2enu(x,y,z, latitude0, longitude0, altitude0, ellipse_type = "WGS84"):
    x0, y0, z0 = geodetic2ecef(latitude0, longitude0, altitude0, ellipse_type)
    x_enu = -math.sin(longitude0)*(x-x0) + math.cos(longitude0)*(y-y0)
    y_enu = -math.sin(latitude0)*math.cos(longitude0)*(x-x0) - math.sin(latitude0)*math.sin(longitude0)*(y-y0) + math.cos(latitude0)*(z-z0)
    z_enu = math.cos(latitude0)*math.cos(longitude0)*(x-x0) + math.cos(latitude0)*math.sin(longitude0)*(y-y0) + math.sin(latitude0)*(z-z0)
    return x_enu, y_enu, z_enu

def geodetic2enu(latitude, longitude, altitude, latitude0, longitude0, altitude0, ellipse_type = "WGS84"):
    x, y, z = geodetic2ecef(latitude, longitude, altitude, ellipse_type)
    x, y, z = ecef2enu(x, y, z, latitude0, longitude0, altitude0, ellipse_type)
    return x, y, z

def ecef2ned(x,y,z, latitude0, longitude0, altitude0, ellipse_type = "WGS84"):
    x0, y0, z0 = geodetic2ecef(latitude0, longitude0, altitude0, ellipse_type)
    x_ned = -math.cos(longitude0)*math.sin(latitude0)*(x-x0) - math.sin(longitude0)*(y-y0) -math.cos(longitude0)*math.cos(latitude0)*(z-z0)
    y_ned = -math.sin(longitude0)*math.sin(latitude0)*(x-x0) + math.cos(longitude0)*(y-y0) -math.sin(longitude0)*math.cos(latitude0)*(z-z0)
    z_ned = math.cos(latitude0)*(x-x0) - math.sin(latitude0)
    return x_ned, y_ned, z_ned

def geodetic2ned(latitude, longitude, altitude, latitude0, longitude0, altitude0, ellipse_type = "WGS84"):
    x, y, z = geodetic2ecef(latitude,longitude, altitude, ellipse_type)
    x, y, z = ecef2ned(x,y,z, latitude0, longitude0, altitude0, ellipse_type)
    return x, y, z
```

A.5.5 MeasurementHandler

```python
import numpy as np
import bisect

import geodetic
#System parameters
VO_DELAY = 1#s

#Filter parameters
STATE_SIZE = 5
X_IND = 0
Y_IND = 1
YAW_IND = 2
X_VEL_IND = 3
YAW_RATE_IND = 4

#Wheel speed parameters
WHEEL_SPACING = 1.7#m
ENCODER_SPEED_RATIO = 0.00308

#Sensors and sizes
IMU_SIZE = 2
VO_SIZE = 3
GPS_SIZE = 2
WHEEL_SPEEDS_SIZE = 2
SENSOR_SIZES = [IMU_SIZE, VO_SIZE, GPS_SIZE,
    ↪ WHEEL_SPEEDS_SIZE]
KNOWN_SENSORS = ["imu", "vo", "gps", "wheel_speeds"]

#Diagonal of measurement noise covarianc
R_IMU = [1e-1 1e-1]
R_VO= [1e-7, 1e-7, 1e-5]
R_GPS = [1e-2, 1e-2]
R_WHEEL_SPEEDS = [1e-3, 1e-3]

def IndexSmallerThan(search_list, value):
    '''Find index of first value in search_list less than
        ↪ or equal to value.
    Assumes sorted search_list. Returns False if none found
        ↪ . Used for vo_dict.'''
    i = bisect.bisect_right(search_list, value)
    if i > 0:
        return i-1
    else:
        return False

class MeasurementHandler(object):
    '''The main measurement handler. Generates z vectors,
        ↪ functions, jacobians and covariances. Contains
        ↪ messages such as picture taken, gps requested
        ↪ frame reset, and what measurements are available
        ↪  in the z vector.'''
    def __init__(self, rbag, vo_dict, enabled_sensors):
        self.bag = rbag
        vo_dict["avail_time"] = [t + VO_DELAY for t in vo_dict["time"]]
        self.vo_dict = vo_dict
        self.enabled_sensors = enabled_sensors
        self.geodetic_origin = None
        self.picture_taken = False
        self.gps_reset = False

    def getMeasurement(self, t_last, t_now):
        '''Returns the measurement vector. Sets measurements_available.'''
        measurements_available = []
        self.vo_nan = False
        if "wheel_speeds" in self.enabled_sensors:
            left_msgs = self.bag.readTopic('/left/feedback', t_last, t_now)
            right_msgs = self.bag.readTopic('/right/feedback', t_last, t_now)
            if not (left_msgs == [] or right_msgs == []):
                left_speed = left_msgs[-1].measured_velocity
                right_speed = -right_msgs[-1].measured_velocity
                z_wheel_speeds = np.asmatrix([[right_speed],[left_speed]])
                measurements_available.append("wheel_speeds")

        if "imu" in self.enabled_sensors:
            imurpy_msgs = self.bag.readTopic('/imu/rpy', t_last, t_now)
            imudata_msgs = self.bag.readTopic('/imu/data', t_last, t_now)
            if not (imurpy_msgs == [] or imudata_msgs == []):
                imu_yaw = imurpy_msgs[-1].vector.z
                imu_yaw_rate = -imudata_msgs[-1].angular_velocity.z
                z_imu = np.asmatrix([[imu_yaw],[imu_yaw_rate]])
                measurements_available.append("imu")

        if "gps" in self.enabled_sensors:
            gpsfix_msgs = self.bag.readTopic('/fix', t_last, t_now)
            if not gpsfix_msgs == []:
                latitude = np.pi*gpsfix_msgs[-1].latitude/180.0
                longitude = np.pi*gpsfix_msgs[-1].longitude/180.0
                altitude = gpsfix_msgs[-1].altitude
                if not np.any(np.isnan([latitude, longitude, altitude])):
                    if self.geodetic_origin == None:
                        '''This can be problematic. Either the filters have to
                        reset their origin to this ENU frame. Or one has to backtrack.'''
                        measurements_available.append("gps")
                        self.gps_reset = True
                        self.geodetic_origin = [latitude, longitude, altitude]
                        z_gps = np.asmatrix([[0],[0]])
                    else:
                        self.gps_reset = False
                        geo0 = self.geodetic_origin
                        x_gps, y_gps, h_gps = geodetic.geodetic2enu(latitude, longitude, altitude, geo0[0], geo0[1], geo0[2])
                        z_gps = np.asmatrix([[x_gps],[y_gps]])
                        measurements_available.append("gps")

        if "vo" in self.enabled_sensors:
            vo_ind = IndexSmallerThan(self.vo_dict["time"], t_now)
            if self.vo_dict["time"][vo_ind] > t_last and self.vo_dict["time"][vo_ind] <= t_now:
                self.picture_taken = True
                self.current_picture = self.vo_dict["name"][vo_ind]
            else:
                self.picture_taken = False
            vo_ind = IndexSmallerThan(self.vo_dict["avail_time"], t_now)
            if self.vo_dict["avail_time"][vo_ind] > t_last and self.vo_dict["avail_time"][vo_ind] <= t_now:
                if not np.any(np.isnan([self.vo_dict["dx"][vo_ind], self.vo_dict["dy"][vo_ind], self.vo_dict["dyaw"][vo_ind]])):
                    z_vo = np.asmatrix([[self.vo_dict["dx"][vo_ind]],
                                             [self.vo_dict["dy"][vo_ind]],
                                             [self.vo_dict["dyaw"][vo_ind]]])
                    measurements_available.append("vo")
                else:
                    self.vo_nan = True

        #Allocate measurement vector
        sensor_indices = [KNOWN_SENSORS.index(sname) for sname in measurements_available]
        measurement_size = sum([SENSOR_SIZES[i] for i in sensor_indices])
        measurement = np.asmatrix(np.zeros( (measurement_size, 1) ))
        measurement_index = 0
        for sensor_name in measurements_available:
            sensor_index = KNOWN_SENSORS.index(sensor_name)
            min_ind = measurement_index
            max_ind = measurement_index + SENSOR_SIZES[sensor_index]
            measurement[min_ind : max_ind, :] = locals()["z_"+sensor_name]
            measurement_index = max_ind
        #Store information
        self.measurements_available = measurements_available
        self.measurement_size = measurement_size
        self.measurement = measurement
        return measurement

    def getFunction(self, measurements_available = None):
        '''Returns an function instance of the measurement function.'''
        if measurements_available == None:
            measurements_available = self.measurements_available
        return MeasurementFunctionGenerator(measurements_available)

    def getJacobian(self, measurements_available = None):
        '''Returns a function instance of the measurement jacobian.'''
        if measurements_available == None:
            measurements_available = self.measurements_available
        return MeasurementJacobianGenerator(measurements_available)

    def getCovariance(self, measurements_available = None):
        '''Returns a numpy matrix of the measurement covariance.'''
        if measurements_available == None:
            measurements_available = self.measurements_available
        return measurementCovariance(measurements_available)

class MeasurementFunctionGenerator(object):
    '''Generates a function z=h(x) that is the measurement function.'''
    def __init__(self, ordering):
        self.ordering = ordering
        self.measurement_size = 0
        for sensor_name in ordering:
            if sensor_name in KNOWN_SENSORS:
                sensor_ind = KNOWN_SENSORS.index(sensor_name)
                self.measurement_size += SENSOR_SIZES[sensor_ind]

    def __call__(self, states):
        '''Call the appropriate function according to ordering.'''
        z_est = np.asmatrix(np.zeros( (self.measurement_size, 1) ))
        measurement_index = 0
        for sensor_name in self.ordering:
            sensor_index = KNOWN_SENSORS.index(sensor_name)
            min_ind = measurement_index
            max_ind = measurement_index + SENSOR_SIZES[sensor_index]
            z_est[min_ind:max_ind,:] = getattr(self, sensor_name)(states)
            measurement_index = max_ind
        return z_est

    def imu(self, states):
        return np.vstack( (states[-(STATE_SIZE-YAW_IND),:],  states[-(STATE_SIZE-YAW_RATE_IND),:]) )

    def gps(self, states):
        '''Assumed arrives as ENU.'''
        return np.vstack( (states[-(STATE_SIZE-X_IND),:], states[-(STATE_SIZE-Y_IND),:]) )

    def vo(self, states):
        cold = np.cos(-states[YAW_IND,:])
        sold = np.sin(-states[YAW_IND,:])
        dx = states[STATE_SIZE+X_IND,:] - states[X_IND,:]
        dy = states[STATE_SIZE+Y_IND,:] - states[Y_IND,:]
        return np.vstack( (cold*dx - sold*dy, sold*dx + cold*dy, states[STATE_SIZE+YAW_IND,:] - states[YAW_IND,:]) )

    def wheel_speeds(self, states):
        right_speed = (states[-(STATE_SIZE-X_VEL_IND),:] + 0.5*WHEEL_SPACING*states[-(STATE_SIZE-YAW_RATE_IND),:])/ENCODER_SPEED_RATIO
        left_speed = (states[-(STATE_SIZE-X_VEL_IND),:] - 0.5*WHEEL_SPACING*states[-(STATE_SIZE-YAW_RATE_IND),:])/ENCODER_SPEED_RATIO
        return np.vstack( (right_speed, left_speed) )
```

```

class MeasurementJacobianGenerator(object):
    '''Generates a function J=H(x) that is the analytic
        ↪ jacobian of the measurement function.'''
    def __init__(self, ordering):
        self.ordering = ordering
        self.measurement_size = 0
        for sensor_name in ordering:
            if sensor_name in KNOWN_SENSORS:
                sensor_index = KNOWN_SENSORS.index(
                    ↪ sensor_name)
                self.measurement_size += SENSOR_SIZES[
                    ↪ sensor_index]

    def __call__(self, states):
        measurement_size = self.measurement_size
        J = np.asmatrix(np.zeros( (measurement_size, states
            ↪ .size) ))
        measurement_index = 0
        for sensor_name in self.ordering:
            if sensor_name in KNOWN_SENSORS:
                sensor_index = KNOWN_SENSORS.index(
                    ↪ sensor_name)
                min_ind = measurement_index
                max_ind = measurement_index + SENSOR_SIZES[
                    ↪ sensor_index]
                J[min_ind:max_ind, :] = getattr(self,
                    ↪ sensor_name)(states)
                measurement_index = max_ind
        return J

    def imu(self,states):
        J_imu = np.asmatrix(np.zeros( (IMU_SIZE, states.
            ↪ size) ))
        J_imu[0, -(STATE_SIZE - YAW_IND)] = 1
        J_imu[1, -(STATE_SIZE - YAW_RATE_IND)] = 1
        return J_imu

    def gps(self, states):
        J_gps = np.asmatrix(np.zeros( (GPS_SIZE, states.
            ↪ size) ))
        J_gps[0, -(STATE_SIZE - X_IND)] = 1
        J_gps[1, -(STATE_SIZE - Y_IND)] = 1
        return J_gps

    def vo(self, states):
        J_vo = np.asmatrix(np.zeros( (VO_SIZE, states.size)
            ↪  ))
        cold = np.cos(states[YAW_IND])
        sold = np.sin(states[YAW_IND])
        dx = states[STATE_SIZE+X_IND]  - states[X_IND]
        dy = states[STATE_SIZE+Y_IND]  - states[Y_IND]

        J_vo[0, X_IND] = -cold
        J_vo[0, Y_IND] = -sold
        J_vo[0, YAW_IND] = -sold*dx +cold*dy
        J_vo[0, STATE_SIZE+X_IND] = cold
        J_vo[0, STATE_SIZE+Y_IND] = sold
        J_vo[1, X_IND] = sold
        J_vo[1, Y_IND] = -cold
        J_vo[1, YAW_IND] = -cold*dx - sold*dy
        J_vo[1, STATE_SIZE+X_IND] = -sold
        J_vo[1, STATE_SIZE+Y_IND] = cold
        J_vo[2, YAW_IND] = -1
        J_vo[2, STATE_SIZE+YAW_IND] = 1
        return J_vo

    def wheel_speeds(self, states):
        J_wheel_speeds = np.asmatrix(np.zeros( (
            ↪ WHEEL_SPEEDS_SIZE, states.size) ))
        J_wheel_speeds[0, -(STATE_SIZE-X_VEL_IND)] = 1/
            ↪ ENCODER_SPEED_RATIO
        J_wheel_speeds[0, -(STATE_SIZE-YAW_RATE_IND)] =
            ↪ 0.5*WHEEL_SPACING/ENCODER_SPEED_RATIO
        J_wheel_speeds[1, -(STATE_SIZE-X_VEL_IND)] = 1/
            ↪ ENCODER_SPEED_RATIO
        J_wheel_speeds[1, -(STATE_SIZE-YAW_RATE_IND)] =
            ↪ -0.5*WHEEL_SPACING/ENCODER_SPEED_RATIO
        return J_wheel_speeds

def measurementCovariance(ordering):
    '''Generates the measurement noise covariance matrix Q.
        ↪ '''
    r_diag = []
    for sensor_name in ordering:
        if sensor_name in KNOWN_SENSORS:
            r_diag+=globals()["R_"+sensor_name.upper()]
    return np.diagflat(r_diag)
```

A.5.6 Horns Method

```
  def hornsMethod(feature_list1, feature_list2):
    '''Takes two lists of features and calculates the pixel
        ↪ movement between the two sets.
    Output: shiftEst, numpy matrix of dc dr, rotEst,
        ↪ estimate of angular rotation.
    '''
    if feature_list1 == [] or feature_list2 == []:
        return ( float('NaN'), float('NaN') ), float('NaN')
    if not isinstance(feature_list1, np.matrix):
        feature_list1 = np.asmatrix(feature_list1).T
    if not isinstance(feature_list2, np.matrix):
        feature_list2 = np.asmatrix(feature_list2).T

    mean1 = np.mean(feature_list1, axis = 1)
    mean2 = np.mean(feature_list2, axis = 1)

    difference1 = np.asmatrix(np.zeros( (feature_list1.
        ↪ shape[0]+1, feature_list1.shape[1]) ))
    difference2 = np.asmatrix(np.zeros( (feature_list2.
        ↪ shape[0]+1, feature_list2.shape[1]) ))
    difference1[0:2,:] = feature_list1 - mean1
    difference2[0:2,:] = feature_list2 - mean2

    C = np.sum(np.diag( difference1*difference2.T ))
    S =  np.matrix([[0, 0, 1]])*np.asmatrix(np.sum(np.cross
        ↪ (difference1, difference2, axisa = 0, axisb = 0)
        ↪ , axis = 0)).T

    shiftEst = np.asmatrix(np.zeros( (feature_list1.shape
        ↪ [0] + 1 ,1) ))
    shiftEst[0:2,:] = mean1 - mean2
    rotEst = - np.arctan2(S,C)
    return shiftEst, rotEst
```