Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Juraj Figura

Probabilistic models for localization of an unmanned aerial vehicle based on real data

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Marta Vomlelová, Ph.D.
Study programme: Informatics
Specialization: Theoretical Computer Science

Prague 2014

I would like to thank Mgr. Marta Vomlelová, Ph.D. for her supervision, valuable advices, ideas, help and proposing this thesis, Dr. rer. nat. Martin Saska, Bc. Tomáš Báča and Mgr. Robert Brunetto for providing the data and kind support. Also, I would like to thank my family for the support during my studies.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Pravděpodobnostní modely pro lokalizaci bezpilotního letounu testované na reálných datech

Autor: Juraj Figura

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Marta Vomlelová, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Práca sa zaoberá problémom odhadovania stavu dynamického systému v oblasti robotiky, konkrétne bezpilotných lietajúcich robotov. Na základe dát získaných z robota navrhneme niekoľko pravdepodobnostných modelov pre odhad jeho stavu (hlavne rýchlosti a rotačných uhlov), takisto pre konfigurácie, kde jeden zo senzorov nie je dostupný. Používame Kalmanov filter a Časticový filter a zameriavame sa na učenie parametrov modelu EM algoritmom. EM algoritmus je potom upravený vzhľadom k negaussovskému rozloženiu chyby niektorých senzorov a pridaním penalizačných členov za zložitosť modelu pre lepšie fungovanie na neznámych dátach. Tieto metódy implementujeme v prostredí MATLAB a vyhodnotíme na oddelených dátach. V práci tiež analyzujeme dáta z pozemného robota a použijeme našu implementáciu Časticového filtra pre odhad jeho polohy.

Klíčová slova: pravděpodobnostní robotika, částicové filtrování, strojové učení, bezpilotní letouny

Title: Probabilistic models for localization of an unmanned aerial vehicle based on real data

Author: Juraj Figura

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Marta Vomlelová, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: The thesis addresses the dynamic state estimation problem for the field of robotics, particularly for unmanned aerial vehicles (UAVs). Based on data collected from an UAV, we design several probabilistic models for estimation of its state (mainly speed and rotation angles), including the configurations where one of the sensors is not available. We use Kalman filter and Particle filter and focus on learning the model parameters using EM algorithm. The EM algorithm is then adjusted with respect to non-Gaussian density of some sensor errors and modified using model complexity penalization terms for better generalization. We implement these methods in MATLAB environment and evaluate on separate datasets. We also analyze data from a ground robot and use our implementation of Particle filter for estimation of its position.

Keywords: probabilistic robotics, particle filtering, machine learning, drone

Contents

1. Introduction

1.1 Motivation and goals

Dynamic state estimation is an interesting and intensively researched topic due to the variety of its applications. The task is to (formally) describe a system that evolves in time and its states are not directly observable. Then we can estimate the true state value in a certain time given some measurements or make predictions about the future. Such systems can describe biological processes, economy, robotics or be used in computer graphics.

We focus on the field of robotics and use real data from unmanned aerial vehicles (UAVs) also known as drones. This kind of robots becomes increasingly popular these days. There are a few commercial models available ([27], for instance) and many low-cost, home-made ones. For autonomous control of a robot, state estimation is crucial. A state typically consists of the position including rotation angles and the current speed. These values can be measured by sensors, but they usually provide noisy data. Moreover, some of the sensors may be unavailable.

The most common approach is to maintain a probabilistic distribution of the current state estimate and update it according to the model of the system dynamics and measurement model. These models or their parameters may not be known and learning of them is referred to as *system identification*.

Our goal is to design and implement a probabilistic model for state estimation of a particular UAV. We focus on the tasks where some of the sensors are not available and their values need to be predicted, which requires more accurate model. We use EM algorithm to learn the parameters, Kalman filter and Particle filter to provide estimates. We evaluate our models on separate datasets using likelihood and mean square error (MSE) for predicting the missing values.

We design the methods to be general enough and easily applicable for other types of robots. We test our implementation of Particle filter on a ground robot position estimation task. This introduces a few more problems like asynchronous measurements and misleading GPS values.

1.2 Structure of the thesis

Chapter 2 starts with a description of the dynamic state estimation problem. It introduces three main tasks: filtering, smoothing and learning the model parameters. It briefly summarizes the most common approaches to the filtering problem and then focuses on two of them: Kalman filter and Particle filter. These algorithms are described to a level of detail sufficient for implementation. A special attention is devoted to learning the parameters by EM algorithm. Its application for both Kalman filter and Particle filter is discussed.

Chapter 3 describes our models for state estimation of UAV. It first summarizes the basic approaches of the system identification used by [7] and continues with application of the methods from the previous chapter for this specific data, describes various models and adjustments of EM algorithm. Finally, a few

evaluation experiments compare the methods we used and the final results are presented.

In chapter 4, our analysis of other robotics datasets is presented. The first dataset is from a special experiment Vicon that enables to analyze measurement model more precisely. The second is a ground robot dataset, where our implementation of Particle filter is used to estimate the position of the robot based on GPS and compass measurements.

More details of our MATLAB implementation are then discussed in chapter 5.

2. Theoretical background

In this chapter we describe two approaches for the task of a dynamic system state estimation. In the first section, we introduce the problem in general. Then we describe Kalman filter, a method that assumes a linear model with Gaussian distribution. The next part is devoted to the Particle filter approach. It is a Monte Carlo type of state estimation and therefore an approximate method. However, it is applicable for nonlinear and non-Gaussian problems. For both methods we also derive EM-type algorithms for learning their parameters.

2.1 Problem definition and notations

2.1.1 Notation remarks

Throughout the text, we use uppercase to denote a variable and lowercase to denote a value (both are usually vectors). If a time-dependant value has its time index omitted, it stands for a vector of values for all considered time steps. $x_{i:j}$ stands for the vector of values from time i to time j included. The time steps indices are typically denoted t, T denotes the number of time steps.

2.1.2 Dynamic systems description

Our goal is the state estimation of a dynamic system. We assume discrete time, which is the most common approach in this area. For each time step t, the system is described by its state X_t. It is a vector of possibly continuous values. In robotics, it usually contains position, speed and angles for each coordinate. We denote the dimension of the state vector by n. These values are not directly observable and we try to estimate them using the measurements. The measurement vector Z_t contains the observations that are usually noisy. They are for instance the values provided by the speed and position sensors. We denote the dimension of Z_t by m. Usually $m \leq n$. Sometimes, especially in robotics, we have a separate vector u_t containing the control signals.

The system is then described by the *state transition model* and the *measurement model*. The state transition model defines how the state vector evolves over time. Here, we assume the Markov property, that is, the current state contains all the information about the transition to the next state. In other words

$$p(X_{t+1}|X_1 = x_1, ..., X_t = x_t) = p(X_{t+1}|X_t = x_t). \tag{2.1}$$

The state transition is not deterministic, but we suppose a noise in the transition. Therefore, the state transition model is a function of the previous state and a noise vector.

$$x_t = f_t(x_{t-1}, w_t). \tag{2.2}$$

Here, we can see a general case where the transition function can change over time. However, in the following we suppose there is a single transition function for all the time steps. Generally, there are no constraints on this function, but each estimation method introduces some constraints. Kalman filter assumes a

linear function, for instance. The noise vector w_t, called the *process noise*, is an independent and identically distributed random variable, usually from the normal distribution.

We can imagine the state transition process in an example. The next position of a robot is determined by a formula based on some physical laws considering the previous position, rotation angle and the previous speed. This outputs a new position in the state X_{t+1}. However, there are some factors we do not model exactly, like air resistance or slippery surface, that lead the robot to a bit different (hardly predictable) position. This is modeled by adding a random noise w_t to that calculated position.

The measurement model describes how the (unknown) true state X_t is projected to the measured values.

$$z_t = h_t(x_{t-1}, v_t). \tag{2.3}$$

Similarly, various methods add some constraints to this function. The interpretation of the measurement noise v_t is that the sensors are not exact. By adjusting its distribution, we try to model the sensor errors.

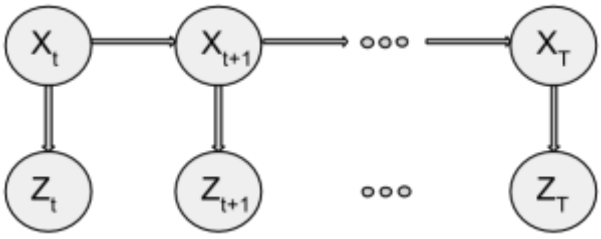

Figure 2.1: Dynamic system viewed as a Hidden Markov Model.

2.1.3 The tasks

Suppose we know both functions f and h. Then the main tasks are:

- **Filtering:** for each time step t, given a sequence of measurements up to t, estimate the state in t. This requires computing $p(X_t|Z_0 = z_0, ..., Z_t = z_t)$, referred to as the *posterior*. Note that we have to be able to compute the current state estimation online (knowing only measurements up to the current time).

- **Smoothing:** given all the measurements up to T, estimate each state, that is, $p(X_t|Z_0 = z_0, ..., Z_T = z_T)$ for each $1 \leq t \leq T$. This can be performed only offline and should provide more accurate (and smoother) results as it has more information than a filter has.

In our thesis, the goal is the filtering task. However, we cannot assume that we know the state transition and measurement models exactly. Therefore, there is a separate task of **learning** these models. More precisely: assuming the structure of the models, learning its parameters, given (offline) measurements. A structure can, for instance, mean a linear model with a Gaussian noise (Kalman filter).

Then the parameters to learn would be the parameters of that linear function, mean and covariance of process and measurement noise. Similarly, we can assume the functions to be mixtures of some certain nonlinear functions. That leads to a more complicated model, but more accurate for nonlinear systems. We can propose several structures of the model, learn the parameters for each and then compare the accuracy of the filters.

We also deal with the smoothing task, because it is used in the algorithms for learning the parameters.

2.1.4 Filtering methods overview

Kalman filter

Kalman filter assumes the posterior at each time step is Gaussian. It can be proved [13] that if the posterior in time t is Gaussian, transition and measurement models are linear and known functions with Gaussian noises of known parameters, then the posterior in time $t + 1$ is also Gaussian. In this scenario, Kalman filter provides the optimal solution. It is also very efficient (time complexity is linear in T, space requires only current mean and covariance). However, the conditions are often too restrictive.

Extended Kalman filter and Unscented Kalman filter

Extended Kalman filter (EKF, [15] [16]) does not require transition and measurement models to be linear. They can be arbitrary differentiable functions. The idea is that these functions can be approximated locally by linear functions. This approximation is obtained by calculating the Jacobian at each time step for the current predicted state. This process linearizes the nonlinear (transition or measurement) function around the current estimate. Then Kalman filter can be used.

Sometimes the linear approximation is not accurate. Unscented Kalman filter [14] instead picks a set of points (called *sigma points*) from the current distribution and applies the nonlinear function to each of them. Then, a new estimated mean and covariance are computed from the transformed points. The choice of the sigma points is deterministic and such that their mean and covariance is the same as the mean and covariance of the current estimate (and the number of points is minimal). The advantage of this method is that it does not have to compute the Jacobian at each time step.

Both of these methods still represent the posterior as a Gaussian. Although they can work with nonlinear models, the true posterior is not Gaussian in this case and these methods provide only its approximation. If the true posterior is bimodal or heavily skewed, it cannot be described well by a Gaussian [4].

Grid-based methods

Grid-based methods are optimal if the state space is discrete and final [4]. Intuitively, we can calculate the posterior as a sum over all possible states. This method can be also used for continuous state space by discretizing the space. Then it is obviously an approximate method.

Particle filters

Particle filters are Monte Carlo approximation methods that represent the posterior by a set of points in the state space called particles. This way, we can represent arbitrary probability distribution assuming enough particles. Particle filters do not have any constraints on the transition and measurement models and can easily handle nonlinear and non-Gaussian models. There are several instances of Particle filters with various methods of generating the new set of particles from the previous. The most common approach is SIR Particle filter [4]. It first transforms each particle according to the transition model and then resamples the particles according to their probability given by the measurement model.

Particle filters are less computationally efficient, their complexity is $O(MT)$ where M is the number of particles and T the number of time steps. Not enough particles, however, leads to not accurate results.

2.2 Kalman filter

2.2.1 Definition

As already mentioned, Kalman filter assumes the posterior is Gaussian and the model to is linear with Gaussian noise. The state transition model is described by

$$x_{t+1} = Ax_t + w_t, w_t \sim \ N(0, Q) \tag{2.4}$$

and the measurement model equation is

$$z_t = Hx_t + v_t, v_t \sim \ N(0, R), \tag{2.5}$$

where $A \in \mathbf{R}^{n \times n}$, $H \in \mathbf{R}^{m \times n}$, $Q \in \mathbf{R}^{n \times n}$ and $R \in \mathbf{R}^{m \times m}$.

In robotics, the transition model often considers control signals separately, that is, uses the equation

$$x_{t+1} = Ax_t + Bu_t + w_t, w_t \sim \ N(0, Q), \tag{2.6}$$

where u_t denotes the control signals vector for time t and $B \in \mathbf{R}^{n \times n_u}$ (n_u denotes the dimension of u_t). However, equation (2.4) is general enough and we can extend the state vector by variables describing the control signals. Then we must consider the control signals being a part of observations, set the corresponding part of the matrix H to be an identity and the corresponding part of Q to be small. We can utilize this also if the control signals are noisy. Moreover, the equations for the parameters learning simplify by that.

2.2.2 Algorithm

The goal of the filtering algorithm is to estimate the state for each time step. Assuming the Markov property, we estimate the next state from the previous state. Estimating a state (in Kalman filter) means to estimate its mean $\hat{x}_t = E[x_t]$ and covariance $P_t = E[(x_t - \hat{x}_t)(x_t - \hat{x_t}^T)]$. We start with an initial estimate $\hat{x}_0$

and initial covariance P_0. We either know these values or set them reasonably (e.g. mean of the state space and high covariance).

Each estimate is made in two steps. The first is called *time update* (or *predictor step*) and the second is called *measurement update* (or *corrector step*). The time update calculates the *apriori* mean and covariance (noted with minus superscript) according to the previous state, transition matrix A and process noise Q (and control signals if not represented in the state) as follows:

$$\hat{x}_t^- = A\hat{x}_{t-1} + Bu_{t-1} \tag{2.7}$$
$$P_t^- = AP_{t-1}A^T + Q. \tag{2.8}$$

The measurement update corrects the apriori estimate using the measurement z_t. It adds a weighted difference of z_t and the predicted measurement $H\hat{x}_t^-$ to the apriori mean:

$$\hat{x}_t = \hat{x}_t^- + K_t(z_t - H\hat{x}_t^-) \tag{2.9}$$
$$P_t = (I - K_tH)P_t^-. \tag{2.10}$$

The weight matrix K_t is chosen to minimize the posterior error covariance (P_t). The resulting matrix is

$$K_t = P_t^- H^T(HP_t^- H^T + R)^{-1}. \tag{2.11}$$

The smaller the measurement covariance R, the more weighted the measurement becomes (we trust the measurement if R is small). If P_t^- is small, measurement is less weighted (because the apriori estimate is confident). [2]

The filtering process is summarized in Algorithm 1. Note that the posterior estimate does not change after it is once calculated (it does not depend on future observations unlike in Kalman smoother), so the algorithm could output this value immediately if necessary.

Algorithm 1

function KALMAN-FILTER($\hat{x}_0, P_0, Z, u, \theta = A, B, H, Q, R$)
  for $t = 1$ to T **do**
    Calculate $\hat{x}_t^-$ and P_t^- according to (2.7) and (2.8).
    Calculate K_t according to (2.11).
    Calculate $\hat{x}_t$ and P_t according to (2.9) and (2.8).
  return $\hat{x}_t$ for each t

2.2.3 Kalman smoother

Smoothing methods provide better estimate of the state in a given time, because they use also the future measurements. This disables them to be used online when current estimate is needed immediately. However, they can be used in the process of learning the parameters of the model as described in Section 2.2.4.

Smoothing consists of two phases. The first is essentially the filtering (called *forward pass*). The second (*backward pass*) passes from time T back to the start

and updates the filter estimates. Again (but backwards), the state in a given time is updated only according to the nearest time step, because for estimation of X_t knowing X_{t+1}, states $X_{t+2:T}$ provide no additional information [17].

We denote the smoothed mean of state in time t by $\hat{x}_{t|T}$ and its covariance $P_{t|T}$ (emphasizing that the values are calculated using all data). The smoothed estimate in time T is equal to the filter estimate. The algorithm then continues from $t = T - 1$ downto $t = 1$ and calculates the smoothed estimate as follows:

$$L_t = P_t A^T (P_{t+1}^-)^{-1} \tag{2.12}$$

$$\hat{x}_{t|T} = \hat{x}_t + L_t(\hat{x}_{t+1|T} - \hat{x}_{t+1}^-) \tag{2.13}$$

$$P_{t|T} = P_t + L_t(P_{t+1|T} - P_{t+1}^-)L_t^T. \tag{2.14}$$

Note that all the smoothed values required by the equations are already calculated at the given time. The smoother also requires all prior estimates and covariances, so the filter has to store these values.

2.2.4 EM Algorithm

This section is based on the slides in [3]. We try to provide further explanation for individual steps mentioned there.

Likelihood

EM algorithm is the most common algorithm for learning the parameters of a model. Here, we learn the values of A, H, Q and R (collectively denoted by θ), given data $Z = z_1, ..., z_T$. The algorithm learns the parameters by maximizing their likelihood. The likelihood of θ given Z is defined as

$$L(\theta|Z) \equiv p(Z|\theta). \tag{2.15}$$

To justify why to maximize the likelihood, let us consider the Bayes formula:

$$p(\theta|Z) = \frac{p(Z|\theta)p(\theta)}{p(Z)}. \tag{2.16}$$

What we in fact would like to maximize is the left hand side of the equation, as we are given the data and searching for the best parameters that explain it. If we do not assume anything about the prior probability of θ, we just maximize the likelihood.

However, we cannot maximize the likelihood directly, because it depends on the hidden states. Therefore we consider the joint likelihood of X and Z

$$L(\theta|X, Z) = p(X, Z|\theta). \tag{2.17}$$

As we do not know the values of X, we will maximize the expectation of the joint likelihood over X, given Z. [5] proves that maximizing the expected joint likelihood also maximizes the original likelihood.

Log-likelihood for Kalman filter

We compute the logarithm of the likelihood and because log is growing monotonically, the maximum is the same for both likelihood and log-likelihood.

$$\log L(\theta|x,z) = \log p(x,z|A,C,Q,R) = \log \prod_{t=1}^{T} p(x_t|x_{t-1})p(z_t|x_t). \tag{2.18}$$

We also suppose the prior state x_0 is not known and define $p(x_1|x_0) \equiv 1$. Using the rule that the logarithm of a product is the sum of logarithms, we get

$$\sum_{t=1}^{T-1} \log p(x_{t+1}|x_t) + \sum_{t=1}^{T} \log p(z_t|x_t) \tag{2.19}$$

Our model assumes that x_{t+1} is drawn from a normal distribution with mean Ax_t and covariance Q. Similarly, $z_t \sim N(Hx_t, R)$. Multivariate (k-dimensional) normal distribution $N(\mu, \Sigma)$ is defined by its probability density function

$$p(x) = (2\pi)^{-\frac{k}{2}} |\Sigma^{-1}|^{\frac{1}{2}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)}. \tag{2.20}$$

To calculate the logarithms in (2.19), we use this density function, that is a product of 3 factors. Therefore the logarithm is a sum of 3 addends, one of which is a constant ($\log(2\pi)^{-k/2}$). The exponential factor transforms to its argument. We get

$$\begin{aligned} &\sum_{t=1}^{T} \left[\frac{1}{2} \log |Q^{-1}| - \frac{1}{2}(x_{t+1} - Ax_t)^T Q^{-1}(x_{t+1} - Ax_t) \right] \\ &+ \sum_{t=1}^{T} \left[\frac{1}{2} \log |R^{-1}| - \frac{1}{2}(z_t - Hx_t)^T R^{-1}(z_t - Hx_t) \right] + const. \end{aligned} \tag{2.21}$$

The first part of each sum is not dependant on the sum, so we take it out $T-1$ (and T, respectively) times. Then we use a trick of adding a trace[1]. $(x_{t+1} - Ax_t)^T Q^{-1}(x_{t+1} - Ax_t)$ is in fact a scalar, despite x_t is a vector. $Tr(x) = x$ when x is a scalar, therefore we have

$$\begin{aligned} &\frac{T-1}{2} \log |Q^{-1}| - \sum_{t=1}^{T-1} \left(\frac{1}{2} Tr \left[(x_{t+1} - Ax_t)^T Q^{-1}(x_{t+1} - Ax_t) \right] \right) \\ &+ \frac{T}{2} \log |R^{-1}| - \sum_{t=1}^{T} \left(\frac{1}{2} Tr \left[(z_t - Hx_t)^T R^{-1}(z_t - Hx_t) \right] \right) + const. \end{aligned} \tag{2.22}$$

In the following, we use these two basic rules: $Tr(AB) = Tr(BA)$ and $Tr(A) + Tr(B) = Tr(A+B)$. Although the argument is a scalar, it is a product of matrices, too.

$$\begin{aligned} &\frac{T-1}{2} \log |Q^{-1}| - \frac{1}{2} Tr \left[Q^{-1} \sum_{t=1}^{T-1} (x_{t+1} - Ax_t)(x_{t+1} - Ax_t)^T \right] + \\ &\frac{T}{2} \log |R^{-1}| - \frac{1}{2} Tr \left[R^{-1} \sum_{t=1}^{T} (z_t - Hx_t)(z_t - Hx_t)^T \right] + const. \end{aligned} \tag{2.23}$$

[1] Trace of a matrix M, denoted $Tr(M)$ is the sum of the elements on the diagonal of M.

Finally, we just expand.

$$\frac{T-1}{2}\log|Q^{-1}| - \frac{1}{2}Tr\left[Q^{-1}\sum_{t=1}^{T-1}\left(x_{t+1}x_{t+1}^T - x_{t+1}x_t^TA^T - Ax_tx_{t+1}^T + Ax_tx_t^TA^T\right)\right] + \frac{T}{2}\log|R^{-1}| - \frac{1}{2}Tr\left[R^{-1}\sum_{t=1}^{T}\left(z_tz_t^T - z_tx_t^TH^T - Hx_tz_t^T + Hx_tx_t^TH^T\right)\right] + const. \tag{2.24}$$

Maximizing of the log-likelihood

Maximizing the likelihood means to compute the partial derivatives of (2.24) with respect to A, H, Q and R, set them equal to zero and solve the equations.

A derivative of a function $f\colon \mathbf{R}^{n\times m} \to \mathbf{R}$ is a generalization of the standard derivative of a function on the real numbers. The important identity here is

$$\frac{\partial Tr(AB)}{\partial A} = \frac{\partial Tr(B^TA^T)}{\partial A} = B^T. \tag{2.25}$$

To calculate the new value for A, we consider all the parts of (2.24) containing A (the rest differentiates to zero):

$$\frac{\partial}{\partial A}\log L(\theta|x,y) = \frac{\partial}{\partial A}\left(-\frac{1}{2}Tr\left[Q^{-1}\sum_{t=1}^{T-1}\left(-x_{t+1}x_t^TA^T - Ax_tx_{t+1}^T + Ax_tx_t^TA^T\right)\right]\right). \tag{2.26}$$

We can split the argument of the trace into 3 parts and compute the sum of 3 derivatives. We also know that Q^{-1} is contained in each.

$$\frac{\partial}{\partial A}\left[Tr\left(\sum_{t=1}^{T-1} -x_{t+1}x_t^TA^T\right)\right] = -\sum_{t=1}^{T-1}x_{t+1}x_t^T \tag{2.27}$$

$$\frac{\partial}{\partial A}\left[Tr\left(\sum_{t=1}^{T-1} -Ax_tx_{t+1}^T\right)\right] = -\sum_{t=1}^{T-1}\left(x_t^Tx_{t+1}\right)^T = -\sum_{t=1}^{T-1}x_{t+1}x_t^T \tag{2.28}$$

$$\frac{\partial}{\partial A}\left[Tr\left(\sum_{t=1}^{T-1} Ax_tx_t^TA^T\right)\right] = \sum_{t=1}^{T-1}2Ax_tx_t^T. \tag{2.29}$$

Therefore the final derivative is equal to

$$Q^{-1}\sum_{t=1}^{T-1}\left(x_{t+1}x_t^T - Ax_tx_t^T\right). \tag{2.30}$$

We set this equal to zero and solve for A.

$$A = \sum_{t=1}^{T-1}(x_{t+1}x_t^T)(x_tx_t^T)^{-1}. \tag{2.31}$$

The case of H is calculated very similarly. The derivative is equal to

$$R^{-1}\sum_{t=1}^{T}\left(z_tx_t^T - Hx_tx_t^T\right) \tag{2.32}$$

and the maximizing H is

$$H = \sum_{t=1}^{T} (z_t x_t^T)(x_t x_t^T)^{-1}. \tag{2.33}$$

Note, that the calculations so far can be viewed as a proof for a general rule

$$\frac{\partial}{\partial A}\left[(x - As)^T W (x - As)\right] = -2W(x - As)s^T, \tag{2.34}$$

which we use also in Section 2.3.7.

The maximizing covariance matrices are

$$Q = \frac{1}{T-1} \sum_{t=1}^{T-1} \left(x_{t+1}x_{t+1}^T - x_{t+1}x_t^T A^T - A x_t x_{t+1}^T + A x_t x_t^T A^T\right), \tag{2.35}$$

$$R = \frac{1}{T} \sum_{t=1}^{T} \left(z_t z_t^T - z_t x_t^T H^T - H x_t z_t^T + H x_t x_t^T H^T\right). \tag{2.36}$$

Expected likelihood

Now, we have to substitute all terms containing X by their expectations given Z. We use the Kalman smoother here, because it outputs more precise results and we are not limited by the online requirement in the process of learning the parameters.

We assume that at each time step the probability distribution of the state is Gaussian and the Kalman smoother outputs its mean and covariance. The expected value of a normal distribution is its mean:

$$E[X_t] = \hat{x}_t^s \tag{2.37}$$

To compute the expectation of $x_t x_t^T$, we use the definition of covariance matrix:

$$cov(X_i, X_j) \equiv E[(X_i - \mu_i)(X_j - \mu_j)] = E[X_i X_j] - E[X_i]E[X_j]. \tag{2.38}$$

From that we directly have

$$E[X_i X_j] = cov(X_i, X_j) + E[X_i]E[X_j]. \tag{2.39}$$

Our covariance matrix was denoted $P_{t|T}$ and we have already derived the expectation of x_t. Therefore the expectation is

$$E[X_t X_t^T] = P_{t|T} + \hat{x}_{t|T}(\hat{x}_{t|T})^T. \tag{2.40}$$

The replacement for $x_t x_{t+1}^T$ is [3]

$$E[X_t X_{t+1}^T] = \hat{x}_t \hat{x}_{t+1|T}^T + L_t \left[P_{t+1|T} + (\hat{x}_{t+1|T} - \hat{x}_{t+1}^-)\hat{x}_{t+1|T}^T\right]. \tag{2.41}$$

Finally, as $cov(A, B) = cov(B, A)^T$,

$$E[X_{t+1} X_t^T] = E[X_t X_{t+1}^T]^T. \tag{2.42}$$

Algorithm

As usual in EM-type of algorithms, we set $\theta = \{A, H, Q, R\}$ to some initial estimates and then repeat the 2 steps until θ converges.

1. **E-step:** using Kalman smoother with parameters θ and given measurements $Z = z_1, ..., z_T$, compute the distributions for $X_1, ..., X_T$.

2. **M-step:** update θ using equations in (2.31), (2.33), (2.35) and (2.36) with expectations used instead of the true state values (according to equations (2.37), (2.40), (2.41) and (2.42)).

The algorithm converges to a local optimum.

2.3 Particle filter

2.3.1 Introduction

Particle filter has recently been one of the most popular approaches to the dynamic state estimation problem. Unlike Kalman filter, it does not constrain the model to be linear nor Gaussian. The basic idea is to represent the posterior by a set of particles. Each particle is a point of the state space (a vector) and has a weight associated to itself. The more particles we use, the more computationally complex the algorithm becomes, but the representation of the posterior is more accurate.

Particle filter can be viewed rather like a family of algorithms. There are multiple ways how to generate new set of particles from the previous one and from the new measurements.

2.3.2 General algorithm derivation [4]

We denote the i-th particle in time step t by x_t^i. In each time step, we have a set of N particles. To derive the algorithm we assume each particle to have assigned the history of the points it has represented since $t = 0$ (particles are initialized according to prior x_0). We denote it by $x_{0:t}^i$. Similarly, all measurements up to time t are denoted by $z_{1:t}$.

Let us assume we have a probability density $q(x_{0:t}|z_{1:t})$, called *importance density*, from which we can generate samples easily. We want to draw the next time step's samples $x_{0:t}^i \sim q(x_{0:t}|z_{1:t})$ from the previous set of samples $x_{0:t-1}^i \sim q(x_{0:t-1}|z_{1:t-1})$, therefore we choose the importance density to satisfy

$$q(x_{0:t}|z_{1:t}) = q(x_t|x_{0:t-1}, z_{1:t})q(x_{0:t-1}|z_{1:t-1}). \tag{2.43}$$

We use a discrete approximation of the posterior, defined as

$$p(x_{0:t}|z_{1:t}) \approx \sum_{i=1}^{N} w_t^i \delta(x_{0:t} - x_{0:t}^i), \tag{2.44}$$

where $\delta(.)$ is the Dirac delta function[2] and the weights are normalized such that they sum up to one. As the samples $x^i_{0:k}$ were drawn from the importance density, the weights must be

$$w^i \propto \frac{p(x^i_{0:t}|z_{1:t})}{q(x^i_{0:t}|z_{1:t})}. \tag{2.45}$$

Now, we want to derive the update rule for the weights. Suppose we have the previous weights w^i_{t-1}. We express $p(x_{0:t}|z_{1:t})$ in terms of the previous posterior $p(x_{0:t-1}|z_{1:t-1})$, measurement density $p(z_t|x_t)$ and transition density $p(x_t|x_{t-1})$. First, we use the Bayes' rule.

$$p(x_{0:t}|z_{1:t}) = \frac{p(z_{1:t}|x_{0:t})p(x_{0:t})}{p(z_{1:t})}. \tag{2.46}$$

We can view $p(z_{1:t}|x_{0:t})$ as a probability of a conjunction and use the fundamental rule in its conditioned form [3], splitting $z_{1:t}$ into z_t and $z_{1:t-1}$. We apply the same (unconditioned) rule for $p(z_{1:t})$ and get

$$\frac{p(z_t|z_{1:t-1}, x_{0:t})p(z_{1:t-1}|x_{0:t})p(x_{0:t})}{p(z_t|z_{1:t-1})p(z_{1:t-1})}. \tag{2.47}$$

Then, again the Bayes' rule is applied, followed by the conditioned fundamental rule.

$$= \frac{p(z_t|z_{1:t-1}, x_{0:t})p(x_{0:t}|z_{1:t-1})}{p(z_t|z_{1:t-1})} \tag{2.48}$$

$$= \frac{p(z_t|z_{1:t-1}, x_{0:t})p(x_t|x_{0:t-1}, z_{1:t-1})p(x_{0:t-1}|z_{1:t-1})}{p(z_t|z_{1:t-1})}, \tag{2.49}$$

which can be simplified using the Markov property to

$$\frac{p(z_t|x_t)p(x_t|x_{t-1})p(x_{0:t-1}|z_{1:t-1})}{p(z_t|z_{1:t-1})} \tag{2.50}$$

$$\propto p(z_t|x_t)p(x_t|x_{t-1})p(x_{0:t-1}|z_{1:t-1}). \tag{2.51}$$

Using (2.45) and (2.43) we have

$$w^i_t \propto \frac{p(x^i_{0:t}|z_{1:t})}{q(x^i_{0:t}|z_{1:t})} \tag{2.52}$$

$$\propto \frac{p(z_t|x^i_t)p(x^i_t|x^i_{t-1})p(x^i_{0:t-1}|z_{1:t-1})}{q(x^i_t|x^i_{0:t-1}, z_{1:t})q(x^i_{0:t-1}|z_{1:t-1})} \tag{2.53}$$

$$= w^i_{t-1}\frac{p(z_t|x^i_t)p(x^i_t|x^i_{t-1})}{q(x^i_t|x^i_{0:t-1}, z_{1:t})}. \tag{2.54}$$

We can choose the importance density such that it depends only on the previous state and the current measurement, i.e,

$$q(x_t|x_{0:t-1}, z_{1:t}) = q(x_t|x_{t-1}, z_t). \tag{2.55}$$

[2]Dirac delta function is defined such that it is zero everywhere except from at 0 and its integral is 1.

[3]$p(A, B|C) = p(A|B, C)p(B|C)$

Then we no longer need the history of the particles.

The general Particle filter algorithm, also called Sequential Importance Sampling Particle Filter (SIS PF) is described in Algorithm 2. For each time step, a new set of particles is generated from the previous in the way that the new $i-th$ particle is drawn from the importance density of the old i-th particle. Then the weights are updated according to (2.54). Specific algorithms depend on the choice of the importance density and other modifications.

Algorithm 2

function SIS-PF-STEP($\{x_{t-1}^i, w_{t-1}^i : 1 \leq i \leq N\}$)
  for $i = 1$ to N **do**
    draw $x_t^i \sim q(x_t|x_{t-1}^i, z_t)$
    $w_t^i := w_{t-1}^i \frac{p(z_t|x_t^i)p(x_t^i|x_{t-1}^i)}{q(x_t^i|x_{t-1}^i, z_t)}$
  return $\{x_t, w_t\}$

In each time step, the expected state is (for normalized weights)

$$\hat{x}_t = \sum_{i=1}^{N} w_t^i x_t^i. \tag{2.56}$$

2.3.3 Resampling

Definition

The major problem of Particle filters is the *degeneracy problem*, where the most particles have very small weights. This typically happens after a few time steps. One method to deal with this is *good choice of importance density* described in [4]. More common approach is *resampling*. This method eliminates the particles with the smallest weights and adds more particles with higher weights. It is basically generating a new set of particles by sampling with replacement from the current set with probability of choosing x_i equal to w_i.

Algorithm

There are several possible ways how to implement resampling. The straightforward algorithm is the following. Let us assume the weights are normalized. First, we calculate cumulated weights, which split the interval $[0, 1]$ into N intervals. We generate N times a random number between 0 and 1 and each time choose the particle whose associated interval surrounds the generated number. Finding the appropriate interval takes $O(N)$ time, so this is $O(N^2)$ algorithm. Sorting the particles according to their weights improves the complexity, but the worst case remains the same.

There is a linear time resampling algorithm described in [20] and another in [19]. We describe *systematic resampling* [18], which is a linear algorithm, too. It moves along the $[0, 1]$ interval in $\frac{1}{N}$-long steps, starting at a random position in $[0, \frac{1}{N}]$. It keeps track of the current interval of the cumulated weights and as it moves only forward, the interval does not have to be searched from start each time. The random selection of the initial position guarantees a chance for each

particle to be selected and the equal steps along the cumulated weights assure proportional sampling. The pseudocode is shown in Algorithm 3.

Algorithm 3

function RESAMPLE($\{x_{t-1}^i, w_t^i : 1 \leq i \leq N\}$)
  $c_1 := 0$
  for $i = 2$ to N **do**
    $c_i := c_{i-1} + w_t^i$
  $i := 1$
  $step := \frac{1}{N}$
  $u_1 := random\_uniform([0, step])$
  for $j = 1$ to N **do**
    $u_j = u_1 + step(j-1)$
    while $u_j > c_i$ **do**
      $i := i + 1$
    $x_t^{j*} = x_t^i$
    $w_t^j = \frac{1}{N}$
  return $\{x_t^*, w_t\}$

When to resample

A measure usually used to estimate degeneracy of particles is

$$\hat{N}_{eff} = \frac{1}{\sum_{i=1}^{N} (w_t^i)^2}. \tag{2.57}$$

If $\hat{N}_{eff}$ (called *effective sample size*) falls below a certain threshold (it is always $\leq N$) resampling is performed.

Sample impoverishment

Resampling leads to another problem called *sample impoverishment*. The problem is lack of diversity among the particles, often all the particles collapse into a single point in the state space. The most severe is the case of small process noise. There are several techniques to avoid sample impoverishment. One is the *resample-move* algorithm [21], another is *regularization* [22]. [4]

2.3.4 SIR Particle filter

Sampling Importance Resampling (SIR) Particle filter is one of the most common instances of the general Particle filter due to its straight-forward implementation. Its choice of importance density is

$$q(x_t | x_{t-1}, z_{1:t}) = p(x_t | x_{t-1}), \tag{2.58}$$

which means that we draw new samples from the prior (transition model). This is sometimes considered as the only way to implement Particle filter, but it is useful to think about it as a particular choice of the importance density.

Using the prior as the importance density leads to the following simplification of the weights update.

$$w_t^i \propto w_{t-1}^i \frac{p(z_t|x_t^i)p(x_t^i|x_{t-1}^i)}{q(x_t^i|x_{0:t-1}^i, z_{1:t})} = w_{t-1}^i p(z_t|x_t^i) \tag{2.59}$$

Moreover, the SIR PF performs resampling in each step, therefore the weights are just $p(z_t|x_t^i)$.

In the SIR Particle filter we need two functions modeling the problem:

- **State transition model:** given a particle, generate a particle for the next step according to $p(x_t^i|x_{t-1}^i)$.
- **Measurement model:** given a particle and a measurement for the corresponding time step, return the probability of the measurement, i.e., $p(z_t|x_t^i)$ or at least up to proportionality (because the weights are normalized either).

There are no constraints on these functions.

Due to resampling at each step, SIR Particle filter often encounters loss of diversity. Moreover, generating particles without knowledge of the observation may lead to inefficient performance [4].

2.3.5 Alternative methods

There are several algorithms based on the Particle filter. *Auxiliary SIR particle filter* proposed by [23] is less sensitive to outliers and has more even weights than SIR in case of small process noise. However, with larger process noise it performs worse.

[24] introduces a filtering method using *Particle Swarm Optimization* (PSO, a general optimization method originally proposed by [25]). The idea is to move the particles towards to regions with high likelihood and simultaneously not allow them to move far from the prior. Multiobjective fitness function is used to achieve these two goals. This function is then optimized by PSO at each time step and the result is used for sampling and assigning the weights. Resampling is performed when $\hat{N}_{eff}$ reaches a threshold.

It is important to mention that all these PF variants are applicable for EM learning of parameters as long as they provide the "interface" of weighted particles for each time step.

2.3.6 Particle smoother

Similarly to Kalman smoother, particle-based methods can also provide smoothed estimates using backward pass. In the backward pass, the weights are updated for each particle, but the particles are not changed nor generated. The update is performed according to equations

$$w_{t|T}^i = w_t^i \sum_{k=1}^{N} \frac{p(_{t+1}^k|x_t^i)}{v_t^k}, \tag{2.60}$$

$$v_t^k = \sum_{j=1}^{N} w_t^j p(x_{t+1}^k|x_t^j). \tag{2.61}$$

2.3.7 EM Algorithm

Likelihood and its expectation

Similarly as in Section 2.2.4, we would like to maximize the expected joint likelihood of the measured data Z and hidden states X, over all possible parameters θ. We denote it by $Q(\theta, \theta_k)$, where θ is the free parameter for maximization and θ_k is a fixed value (obtained from the previous EM iteration) over which we condition the expectation (we use θ_k in the smoother). The space for the possible θ values depends on how we model the problem (see Section 2.3.7).

In Kalman EM algorithm, we used the posterior mean and covariance calculated by Kalman smoother to express the expectation of a state. Now, we cannot assume Gaussian posterior and need to provide the expectation in a general form. Expected value of a function $g(x)$ is an integral over x, which can be approximated by Particle smoother:

$$E\left[g(x)\right] = \int g(x)p(x)dx \approx \sum_{i=1}^{N} w^i g(x^i). \tag{2.62}$$

In our case, the function g is the joint log likelihood, i.e.,

$$\sum_{t=1}^{T-1} \log p(x_{t+1}|x_t) + \sum_{t=1}^{T} \log p(z_t|x_t). \tag{2.63}$$

The second sum's expectation can be expressed as

$$\begin{aligned} E_{\theta_k}\left[\sum_{t=1}^{T} \log p_\theta(z_t|x_t)|z_{1:T}\right] &= \sum_{t=1}^{T} \int \log p_\theta(z_t|x_t) p_{\theta_k}(x_t|z_{1:T})dx_t \\ &\approx \sum_{t=1}^{T}\sum_{i=1}^{N} w_{t|T}^i \log p_\theta(z_t|x_t^i). \end{aligned} \tag{2.64}$$

It is more complicated to express the expectation of $\log p_\theta(x_{t+1}|x_t)$, because it leads to integration over all x_t and x_{t+1} simultaneously. [5] proves that

$$\begin{aligned} E_{\theta_k}\left[\log p_\theta(x_{t+1}|x_t)|z_{1:T}\right] &\approx \sum_{i=1}^{N}\sum_{j=1}^{N} w_{t|T}^{ij} \log p_\theta(x_{t+1}^j|x_t^i), \\ w_{t|T}^{ij} &= \frac{w_t^i w_{t+1|T}^j p_{\theta_k}(x_{t+1}^j|x_t^i)}{\sum_{l=1}^{N} w_t^l p_{\theta_k}(x_{t+1}^l|x_t^l)}. \end{aligned} \tag{2.65}$$

Our approximation of $Q(\theta, \theta_k)$ is the sum of the terms in (2.64) and (2.65). Maximization of this function over θ can be performed either

- when having a closed form of the equation, solving it directly (as in Kalman filter EM), or
- iterating using gradient as proposed in [5].

Nonlinear models

[6] proposes an approach where nothing is known about the structure of the measurement model. The function $h(x(t), \theta)$ is then modeled as a mixture of Gaussians:

$$h(x_t, \theta) = \sum_{p=1}^{P} m_p g_p(x_t) + Hx_t + b. \tag{2.66}$$

The parameters of this model are $m_p \in \mathbf{R}^{\mathrm{m}}$ for P being fixed, $H \in \mathbf{R}^{\mathrm{m}\times\mathrm{n}}$ and $b \in \mathbf{R}^{\mathrm{m}}$. The functions g_p are Gaussians with fixed means and covariance matrices. The means are distributed over the range of the states and the covariances are chosen to be identities for simplicity. The measurements are assumed to be drawn from a normal distribution with mean $h(x_t, \theta)$ and unknown covariance R.

The state transition model is assumed to be completely known (but nonlinear with a non-Gaussian process noise). Then the EM algorithm estimates only m_p, H, b and R.

We denote

$$\Phi_t \equiv \left[g_1(x_t), g_2(x_t), ..., g_P(x_t), x_t^T, 1\right]^T \tag{2.67}$$

$$\theta \equiv \left[m_1, m_2, ..., m_P, A, b\right]. \tag{2.68}$$

Then

$$h(x_t, \theta) = \theta\Phi_t. \tag{2.69}$$

Given the Gaussian distribution of the measurement model, we have

$$\log p_\theta(z_t|x_t) = -\frac{1}{2}(z_t - \theta\Phi_t)^T Q^{-1}(z_t - \theta\Phi_t) + \frac{1}{2}\log|Q| + const. \tag{2.70}$$

Maximizing the mixture of Gaussians measurement model

We want to maximize the expectation of (2.70) over θ. Using rule (2.34) leads to its derivative with respect to θ being

$$\sum_{t=1}^{T} Q^{-1}(z_t - \theta\Phi_t)\Phi_t^T. \tag{2.71}$$

Solving for θ with the derivative equal to zero leads to the maximizing θ:

$$\theta_{k+1} = E_{\theta_k}\left[\sum_{t=1}^{T}(z_t\Phi_t)(\Phi_t\Phi_t^T)^{-1}\right] = \sum_{i=1}^{N}\sum_{t=1}^{T} w_t^i(z_t\Phi_t^i)\left[\Phi_t^i(\Phi_t^i)^T\right]^{-1}. \tag{2.72}$$

Similarly, for the covariance matrix R:

$$R_{k+1} = \sum_{i=1}^{N}\sum_{t=1}^{T} w_t^i\left[z_t z_t^T - \theta_{k+1}\Phi_t^i z_t^T\right]. \tag{2.73}$$

A similar model could be used for estimation of state transition parameters. Generally, this method is computationally impractical as its complexity grows exponentially with the dimension of the state space (if no prior knowledge about where to place the means of the Gaussian kernels).

Algorithm overview

The algorithm iterates the expectation (E) and maximization (M) steps, until the parameter θ converges, starting with an arbitrary initialization $\theta_k, k = 0$.

- **E-step:** calculate $Q(\theta, \theta_k)$. This value is parametrized by an unknown value of θ, so by calculating Q, we mean estimating the probabilities of the hidden states by the Particle smoother, which enables to express Q.
- **M-step:** assign $\theta_{k+1} = \underset{\theta}{\operatorname{argmax}}\, Q(\theta, \theta_k)$ using either closed form equations or a gradient method.

The two steps are independent in sense that we can employ any particular method of Particle filtering for the E-step.

3. Application for unmanned aerial vehicles

In this chapter we describe our design of probabilistic models for tasks based on unmanned aerial vehicle (UAV) data. We describe several methods with focus on learning the model parameters. We evaluate these methods and analyze the results.

3.1 Data and tasks

The data was provided by the Czech Technical University in Prague (ČVUT). We have received 4 files, each describing a single flight. The flights have various length, approximately from 5 to 20 thousands records for a flight, 66 000 together.

All the flights were performed by the same UAV in the same conditions. A static point (so called blob) was marked on the floor and the aim was to stay above that point. Naturally, the UAV was oscillating around that spot and the overall view of the speeds seems like a non-perfect sinusoid.

Currently, the flights are not performed in more complicated conditions (outside, for instance), because "the system is unstable, dangerous and easily damageable" [7].

3.1.1 Data description

A single record has the following format:

- **pitch** - rotation angle in the forward/backward axis (B_x)
- **roll** - rotation angle in the left/right axis (B_y)
- **elevator** - control signal for the forward motion, desired angle (B_y)
- **aileron** - control signal for the sideways motion, desired angle, (B_y)
- **throttle** - control signal for the elevation (height)
- **x** - position in the x axis (G_x)
- **y** - position in the y axis (G_y)
- **yaw** - rotation angle around the vertical axis (coordinate system I)
- **xdot** - speed in the forward axis (B_x)
- **ydot** - speed in the sideways axis (B_y)
- **z** - height from the ground measured in axes perpendicular to the plane of the UAV
- **time** - time elapsed from the beginning.

There are multiple coordinate systems used, as described on Figure 3.1. The most important for us (regarding our tasks, see Section 3.1.2) is that *pitch*, *roll*, *elevator*, *aileron*, *xdot* and *ydot* are all measured in the same coordinate system, the one depending on the current pose of the UAV.

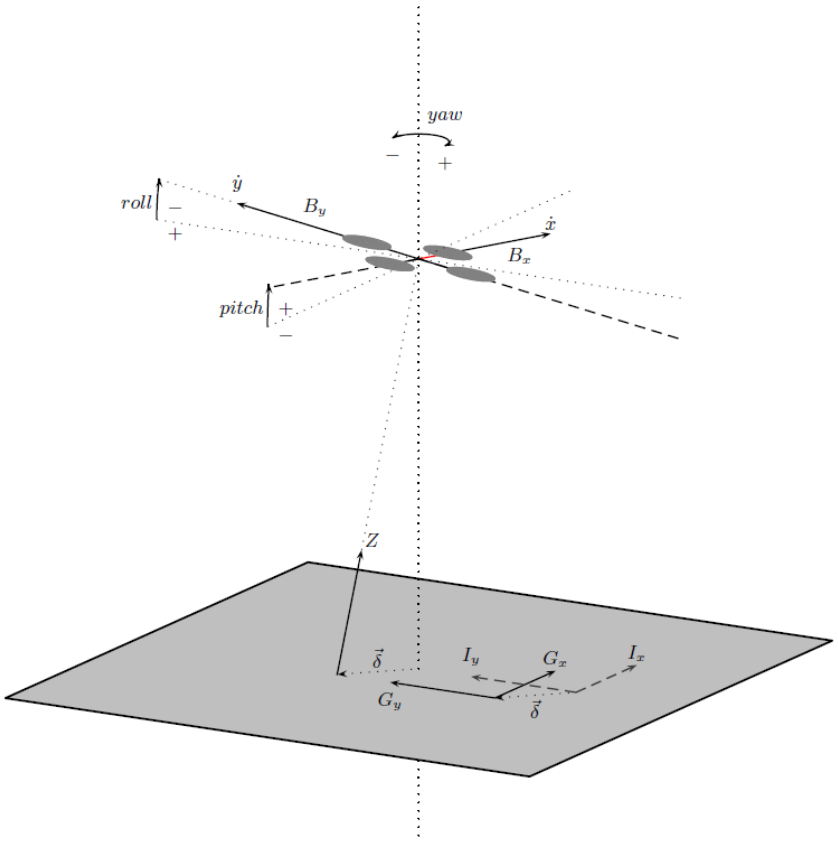

Figure 3.1: The coordinate systems of UAV. Credits: Tomáš Báča.

The records are sampled each 0.014 second (with minor variations). That enables us to use discrete time.

The values are from multiple sensors and not perfectly synchronized. It is assumed that the speed and angle are synchronized quite well position data is about 0.5 second delayed.

It is further assumed that the axes x and y are independent [7]. Therefore we can separately solve the problem for $pitch, elevator, x, xdot$ and separately for $roll, aileron, y, ydot$. Moreover, the axes should act very similarly, so we focus on just one of them mostly.

Speed measurements are more noisy than others. Position and angle measurements have instead small precision (values only with one decimal place) and probably smaller sample rate. This causes the plots to consist of sequences of the same values. Each sequence has from 3 to 10 identical records.

3.1.2 Tasks

For all tasks, the important state variables are $pitch, xdot$ and x considering the B_x axis. Control signals (i.e. *elevator*) are always available. The tasks vary in the measurement vectors.

Task	Input	Output
Task I	$pitch, xdot, x, elevator$	$pitch, xdot, x$
Task IIa	$xdot, elevator$	$pitch$
Task IIb	$pitch, elevator$	$xdot$

Table 3.1: Summary of the required tasks.

The first task (referred to as Task I) is the basic filtering task to estimate position, speed and angles. Here, the most important is the speed estimation, as its measurements are the noisiest. For all the variables in the state vector, there is a measurement value in the measurement vector.

The second, more complex task, is the estimation of the state when one of the sensors is not available (referred to as *prediction*). There is a configuration currently used at ČVUT where there is not possible to measure the angles and their estimate would be helpful. Moreover, in some cases the position is not measured, too. Another configuration is missing the speed sensor.

Therefore, we define the Task IIa to consider the measurement vector of only $xdot$ and control signals of $elevator$. In Task IIb, the measurement vector consists of $pitch$ and x, control signals contain the $elevator$ value.

The performance of the prediction can be measured comparing to the values we have in the data.

We should design a single model capable of the state estimation or prediction given either of the described inputs.

3.1.3 Currently used methods

For filtering the values, the exponential filter is currently used at ČVUT. Standard exponential filter would be the following:

$$xdot_{t+1} = \alpha\, xdot_t + (1-\alpha) Z(xdot)_{t+1} \tag{3.1}$$

for α being typically 0.95 and Z denoting the measurement.

Taking the pitch value into account and knowing the model parameters (discussed in 3.2.2) enables us to get a slightly better performance. Therefore, the best current solution (by ČVUT) is the following:

$$xdot_{t+1} = \alpha\, [p_3\, xdot_t + p_2 Z(pitch)_t] + (1-\alpha) Z(xdot)_{t+1}. \tag{3.2}$$

The performance of this model compared to our models is show on Figure 3.18 in Section 3.3.4.

In [7], simplified Kalman filter for a single variable ($xdot$) is also used.

There is currently no method used to predict the speed or angle without knowing their measurements.

3.2 Applied methods

We describe a set of methods that can be used and configured easily for each of the tasks and are general enough to be applied for other problems. The approach

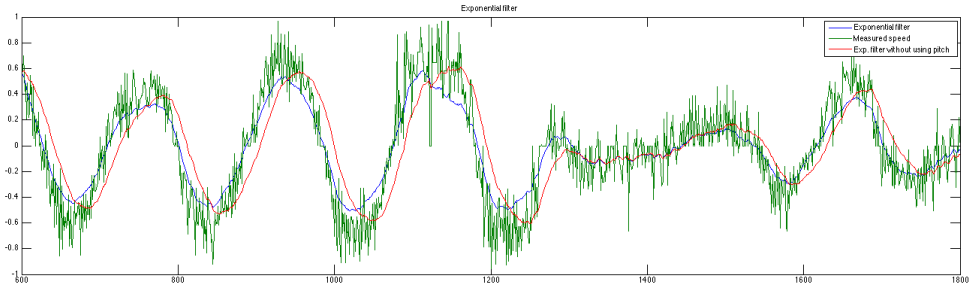

Figure 3.2: Using exponential filter for speed filtering. The blue line shows the approach considering the pitch, red line is a simple exponential filter.

consists of multiple steps like preprocessing, design of the model structure, learning the parameters, filtering and smoothing. In each step we have a few options to choose from. Some of the combinations are compatible and some are not. For instance, if we choose a linear model, we can choose either Kalman filter or Particle filter. This is not possible for a nonlinear model. Similarly, any variation of Particle smoother is compatible with EM algorithm as long as it provides the particles and their weights.

3.2.1 Preprocessing

Missing values

Sometimes the sensors output an out of range value. A preprocessing step that filters extreme values was already proposed by ČVUT. The preprocessing step simply replaces an extreme value by the previously measured value. Online implementation with preprocessing step would use fixed data ranges (learnt on the offline data or user defined) and replace the outliers according to that. In the following, we use the preprocessed data, which also makes the plots more readable.

Another approach is to treat the outliers as missing values. The filtering algorithm should be modified not to perform the correction step for these values and likelihood calculation should not take them into account, too. However, the portion of the outliers in our data is only about 0.2%.

Scaling

Another preprocessing step is scaling all the values to $[-1, 1]$ interval. This improves the performance of EM algorithm. The reason is that likelihood value (which is a value of the probability density function) depends on the covariance matrix. If one of the variables has significantly smaller range (and therefore the covariance), the likelihood is affected more by this variable and EM algorithm improves it at the expense of the other variables. The effect of scaling is tested experimentally in Section 3.3.8.

3.2.2 Dynamic system description

From the underlying physical point of view, we know the causal relations between the variables.

Elevator affects *pitch* as it is a control signal defined as the desired pitch. The *pitch* value however, does not respond immediately, nor exactly to the value of *elevator*.

The speed is affected by the corresponding angle (*xdot* by *pitch*). This may be not intuitive, but it is true for UAVs.

Finally, the position is determined by the previous position, speed and the time difference, trivially from the definition of speed.

Therefore, we should find the parameters for the following equations:

$$pitch_t = p_1\, pitch_{t-1} + p_0\, elevator_{t-1} \tag{3.3}$$

$$xdot_t = p_3\, xdot_{t-1} + p_2\, pitch_{t-1} \tag{3.4}$$

$$x_t = p_4\, xdot_{t-1} + x_{t-1} \tag{3.5}$$

The equation for x is only an approximation because $xdot$ and x are measured in a different coordinate system. This leads to negative values of $p4$ learnt by our algorithms. Despite this fact, the position can be easily filtered using Kalman filter (Figure 3.3).

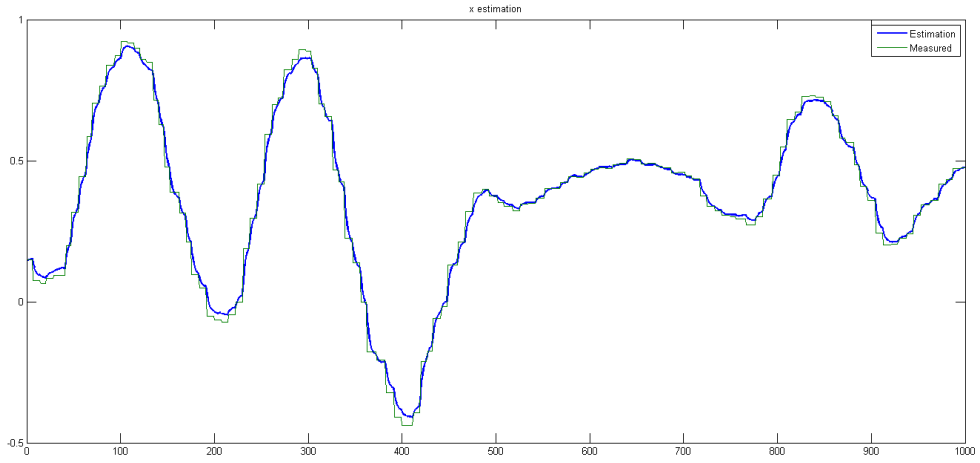

Figure 3.3: Kalman filter for position.

Considering *elevator* as the last element of the state, the transition matrix A would be

$$A = \begin{bmatrix} p_1 & 0 & 0 & p_0 \\ p_2 & p_3 & 0 & 0 \\ 0 & p_4 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

As we focus on the task of prediction without x measurement, we decided to omit the third row and column.

Moreover, we propose a model with a new variable *xdotdot*. The motivation is to avoid the problem of learning algorithms outputting $p_3 > 1$ (which leads to unstable model as discussed further). The new variable stands between *pitch* and

xdot in the causal relationship, i.e.,

$$pitch_t = r_1\, pitch_{t-1} + r_0\, elevator_{t-1} \tag{3.7}$$
$$xdot_t = r_3\, xdot_{t-1} + r_2\, xdotdot_{t-1} \tag{3.8}$$
$$xdotdot_t = r_4\, xdotdot_{t-1} + r_5\, pitch_{t-1}. \tag{3.9}$$

In conclusion, we mainly use the transition matrix

$$A = \begin{bmatrix} p_1 & 0 & p_0 \\ p_2 & p_3 & 0 \\ 0 & 0 & 1, \end{bmatrix} \tag{3.10}$$

and in Experiment C (Section 3.3.5) we use

$$A = \begin{bmatrix} r_1 & 0 & r_0 & 0 \\ 0 & r_3 & 0 & r_2 \\ 0 & 0 & 1 & 0 \\ r_5 & 0 & 0 & r_4 \end{bmatrix}. \tag{3.11}$$

3.2.3 Learning parameters by linear regression

The first estimation of the parameters for equations (3.3), (3.4) and (3.5) can be performed by least squares method on the data as used by [7]. For instance, to estimate parameters in (3.4), we solve

$$\begin{bmatrix} xdot_1 & pitch_1 \\ xdot_2 & pitch_2 \\ \vdots & \vdots \\ xdot_{T-1} & pitch_{T-1} \end{bmatrix} \begin{bmatrix} p_3 \\ p_2 \end{bmatrix} = \begin{bmatrix} xdot_2 \\ xdot_3 \\ \vdots \\ xdot_T \end{bmatrix} \tag{3.12}$$

using linear regression.

The problem is that the equations describe the transition model over the hidden states and therefore using only the measured data can lead to wrong results. If we had the information about the hidden states, the least squares method would output the best possible parameters given Gaussian transition noise. This information can be obtained by special experiments with more reliable (typically external) sensors. An example of such an experiment is described in 4.1.

Preprocessing by smoothing the data

In the most cases, the hidden state values are not available and so it is with our data. However, we can improve this method of parameter estimation. We can try to define the hidden state and use it for learning the parameters. For instance, a smoothed sequence of values will perform better in the least squares method then the raw data. We can obtain smoothed data in the learning process, because we can make use of the whole data.

The method used by [7] was polynomial fitting for noisy measurements. It was used there for estimation of parameters for speed equation (3.4) in the way that the input for the regression was the fitted function values instead of the

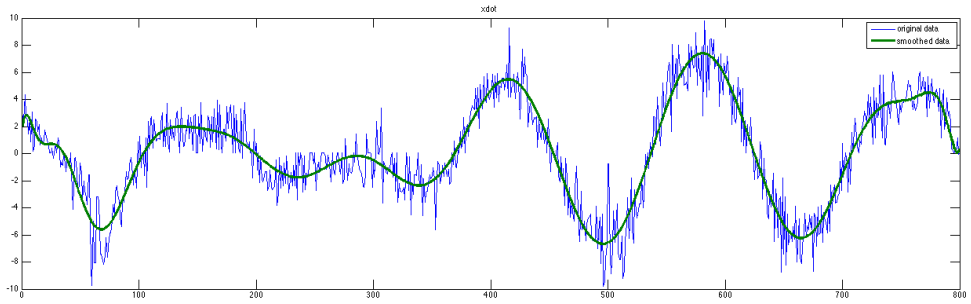

Figure 3.4: Polynomial fit for the speed ($xdot$) measurement.

measurements. The speed measurements and their polynomial fit are shown in Figure 3.4.

Although the measurements of pitch are not noisy, we try using the fitting here, too. The approach is to replace the sequences of repeated measurements.

Polynomial fitting can be reasonably used only for small portions of training data. In our implementation, we split the data into smaller parts and perform the fitting for each part[1]. At the splitting points our fitting function is discontinuous, but it does not affect the regression significantly. To address this problem, we could use spline fitting [12]. However, our experiments showed that more than a few hundred data points do not have much impact on the results.

As the speed measurements are very noisy, learning parameters directly from them has poor results. However, the results of the regression using the fitted function cannot be used either, because it sets p_3 to be slightly higher than one. That leads to exponential growing of the estimations over time, as also noted in [7]. If correction step is not present (the configuration without the speed sensor), setting p_3 to 1.00 or smaller is necessary.

Conclusion

Standard machine learning methods (like linear regression) fit a function $f(X)$ from the given data points $[X, y]$. The points y may be noisy, but X are given with a high confidence. In our case, the inputs X (previous $xdot$ and $pitch$) are noisy, too. Therefore, we need an approximation of them. Polynomial fitting can be viewed as an approach to this. Another approach is the EM algorithm, which calculates its own approximation (or smoothing) of the inputs. The approximation made by EM algorithm is the state estimate given the current parameters estimate. Then, similarly to the regression, it minimizes the error. The smoothing by EM algorithm is more plausible and performed iteratively.

[1]The coefficients of the fitted polynomials are very different for each part of data. We use polynomials of degree 10 for sequences of 400 points. The function `smooth_data_polyfit` can be found in the attached implementation.

3.2.4 Parameters of the model

Linear models

The simplest approach in this step is a linear model as described in 3.2.2. Here we have to decide whether the parameters $p_0, ...p_3$ should be the only model parameters, or each position in the matrix A should be free to tune. The first option is more plausible, the second tends to provide an overfitted model as shown in 3.3.

However, learning more positions in A can make the model more robust. Parameters $p_0, ..., p_3$ describe the causal relationships between the state variables, $A(1, 2)$, for instance, represents the influence of *xdot* on *pitch*, which is not causal (as we know from the physical background), but these variables are still coupled. Knowing these non-causal relationships can improve the performance in scenarios with higher uncertainties like missing sensor values (see the final learnt values of A in equation 3.19 in Section 3.4).

Linear models with penalization

Another approach is to introduce penalization terms for model complexity. That is, maximizing

$$L(A|Z) - \kappa \sum_{(i,j)\in O} A(i,j)^2 \tag{3.13}$$

where $L(A|Z)$ is likelihood of transition matrix A given measurements Z, O is a set of positions in A different from the positions of $p_0, ...p_3$ and κ is a tradeoff constant.

Calculating the derivative of (3.13)[2], setting equal to zero and solving for A leads to the following update of A in M-step of the modified EM algorithm (compare to (2.31):

$$A = \sum_{t=1}^{T-1} (x_{t+1}x_t^T)(x_t x_t^T)^{-1} - \kappa A_O \sum_{t=1}^{T-1} Q^{-1}(x_t x_t^T)^{-1}, \tag{3.14}$$

where A_O is the matrix A with zeros on each position not in set O.

Similarly, we have to consider the covariance matrices Q and R. Here, the problem with our data is that EM algorithm would set the *pitch* variance too small because the noise is not Gaussian (data contain sequences of identical values). We would like to keep the variance sufficiently large and introduce the penalization term

$$\kappa_2(Q_{def} - Q)^2, \tag{3.15}$$

where Q_{def} is our estimate of Q that we do not want to go too far from. We maximize Q as follows:

$$Q_{new} = Q_{LLmax} + \frac{1}{T}\kappa_2(2Q_{def} - Q_{old}), \tag{3.16}$$

[2]Derivative of a scalar value in a matrix with respect to that matrix is a matrix with 1 at the corresponding position, derivative of a squared value is a matrix with the value on the corresponding position.

where Q_{LLmax} is the maximizing term from standard EM algorithm (equation (2.35)).

The values for κ have to be determined empirically by comparing the performance on validation sets.

Regarding the measurement matrix H, we always set it to identity because we know that there is a direct mapping from states to sensor values (e.g. *xdot* to *xdot* measurement).

Nonlinear extension

Sometimes we know the formula for the transition or measurement model, for instance that a certain value in state changes with square of another. In our data, we do not know this. We know that the linear model performs quite well, but sometimes it is not absolutely accurate. Nonlinearity do not have to be the reason behind this, however. [6] proposes a measurement model for cases where the structure is completely unknown. We try to apply this idea to the transition model.

$$x_{t+1} = \sum_{p=1}^{P} m_p g_p(x_t) + Ax_t, \tag{3.17}$$

where each g_p is a Gaussian with a fixed mean and covariance. We spread the means across the state space uniformly. The advantage of this model is that it can be based on the linear part (Ax_t) that is known from the previous approaches and shows which parts of the state space do not act linearly. If each m_p is set to 0, we have a linear model. The nonlinear parts are modeled by the sum of Gaussians. The parameters of this model are the coefficients m_p and A if we want to further tune it.

3.2.5 Noise models

Transition and measurement noise is typically assumed to be Gaussian. This perfectly fits for Kalman filter and can be easily implemented in Particle filter. However, it is not always a correct assumption. We can see the measurement noise distribution when we know the true values of hidden states (see 4.1 for such an experiment). In our case, we can estimate the measurement error distribution by first performing a filtering method and estimating the true states. Figure 3.5 shows a histogram of *xdot* measurement error using the filtered state estimates. The noise is indeed Gaussian. However, that is not the case for *pitch* (see Figure 3.6.

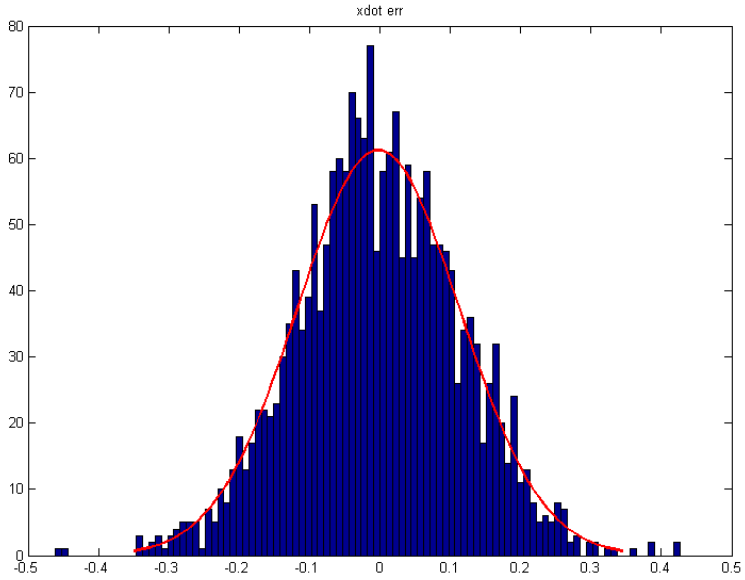

Figure 3.5: Measurement error histogram of *xdot* regarding to the best filtering estimate of true state.

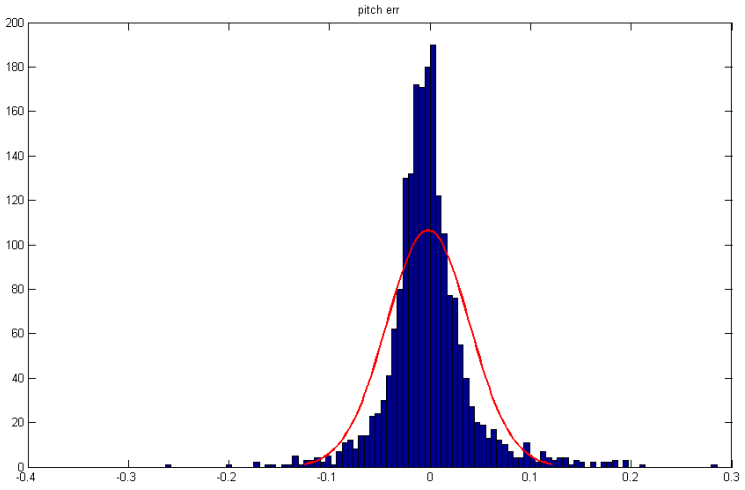

Figure 3.6: Measurement error histogram of *pitch* regarding to the best filtering estimate of true state.

The advantage of Particle filter is that we can model the probability of a measurement by an arbitrary function, but our models still approximate the error by a Gaussian distribution and perform well.

3.2.6 Filter, smoother and EM algorithm

We implement both Kalman filter and SIR Particle filter (and smoothers) as described in sections 2.2.2, 2.2.3, 2.3.3 and 2.3.6. The problem seem to have a

linear structure so we focus on Kalman filter, which is computationally faster and more accurate for this setting. For learning the parameters, we implement Kalman EM algorithm with penalization model as described in 3.2.4. We can also choose which parameters of the model should remain fixed.

In Section 3.3.6 we compare the performance of Kalman filter and Particle filter both using the same linear model.

3.3 Experiments

3.3.1 Evaluation methods

Introduction

In our experiments, we compare various methods of modeling and learning the parameters described in the previous section.

To compare the results of various learning methods, we need to compare the performance on a validation set. That is, a set of data unseen by the learning algorithm. For a better estimation of the performance, k-fold cross-validation is typically used. However, for the sequential (time-dependant) data, we cannot shuffle the records. The solution for this is known as *forward chaining* [28]. This method splits the (sequential) data into k distinct folds so that concatenation of the folds $1, 2, ..., k$ forms the original data. Then we use $1, 2$ for training and 3 for validation, $1, 2, 3$ for training and 4 for validation and so on.

Similar method is to use a fixed size training set. For instance $1, 2, 3$ for training, 4 for validation, $2, 3, 4$ for training, 5 for validation and so on. To avoid overfitting as much as possible, our approach is to choose the validation set starting at a nonzero offset from the end of the training set. This should prevent the cases where a data follows a pattern that may continue in the validation set. Choosing the right training set size and validation set offset is discussed in Section 3.3.2.

It is also important to choose sufficiently large validation sets to test the *stability* of the model. Some models may predict well at the beginning and encounter problems after a few hundreds of time steps and then cannot recover to the true state (see Figure 3.7).

Summary of the used evaluation methods

The following procedure is performed for each learning method and repeated 5 times using different training sets.

1. We select a training set of a fixed size and learn a model from all the data $(pitch, xdot, elevator)$ in this set. The learning method also defines its filtering method, which is Kalman filter in most of the cases and Particle filter in Experiment D.

2. We take a validation set separate from the training set, starting at a given offset from the end of the training set.

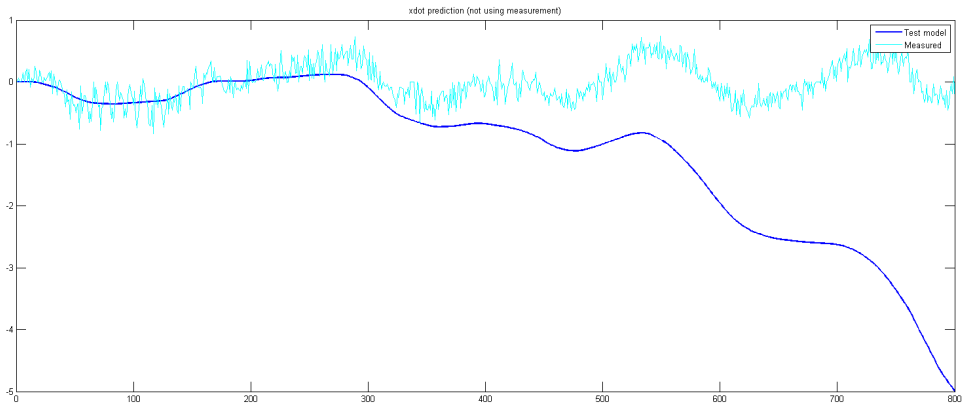

Figure 3.7: Demonstration of a model that predicts well at the beginning, but gets lost after a few hundreds of time steps. Model learnt by $B5$ was used with p_3 changed to 1.005. A similar scenario happens, for instance, for models of Experiment A if p_3 is not manually set lower. Therefore, this parameter is set to 0.997 for the most models.

3. We run the filtering method on a validation set, still with all measurements available and calculate the expected likelihood according to equation (2.24) and expectation according to (2.37), (2.40), (2.41) and (2.42). The likelihood is normalized by the size of the validation set. [3].

4. We generate a plot of the filtering performance given all measurements.

5. We run the filtering algorithm on the validation set with *pitch* measurements removed and calculate the prediction error as MSE, i.e.,

$$MSE_{pitch} = \frac{1}{T}(pitch_{prediction} - pitch_{measured})^2. \tag{3.18}$$

6. Similarly, we run the filtering algorithm with *xdot* measurements removed and calculate the *xdot* MSE. We still use the same learnt model with the correspondent rows of matrix H and rows and columns of R removed.

7. We generate a plot showing the prediction of *pitch* and a plot showing the prediction of *xdot*.

For a few reasons, log likelihood serves only as an orientation measure. First, it assumes Gaussian distribution, which is not satisfied by some of our variables. Second, it cannot be compared between models with different size of the state vector nor for Particle filter. Third, the filtering task can be solved easily (see Section 3.3.3) and it is not so important as the prediction of missing measurements. On the other hand, prediction errors can be compared across all the models.

All the training and validation sets are taken from a single flight (number 3). We compare the average performance (among 5 iterations) of the methods and

[3]The values of log likelihood are usually negative. However, positive values are also correct, because the likelihood is not calculated as a probability but as a value of the probability density function, which can be greater than one for small covariances.

select the best methods for the final evaluation. The final evaluation is performed on the rest of the flights (1, 2 and 4). There is no learning from these flights, just filtering and prediction.

3.3.2 Validation parameters

Training size

To estimate the best size for training, we performed our validation procedure with various training set sizes (3 separate runs for each size from 80 to 4000). Based on the resulting prediction errors (Figure 3.8 and 3.9) we decided to use the training size of 1000 data records.

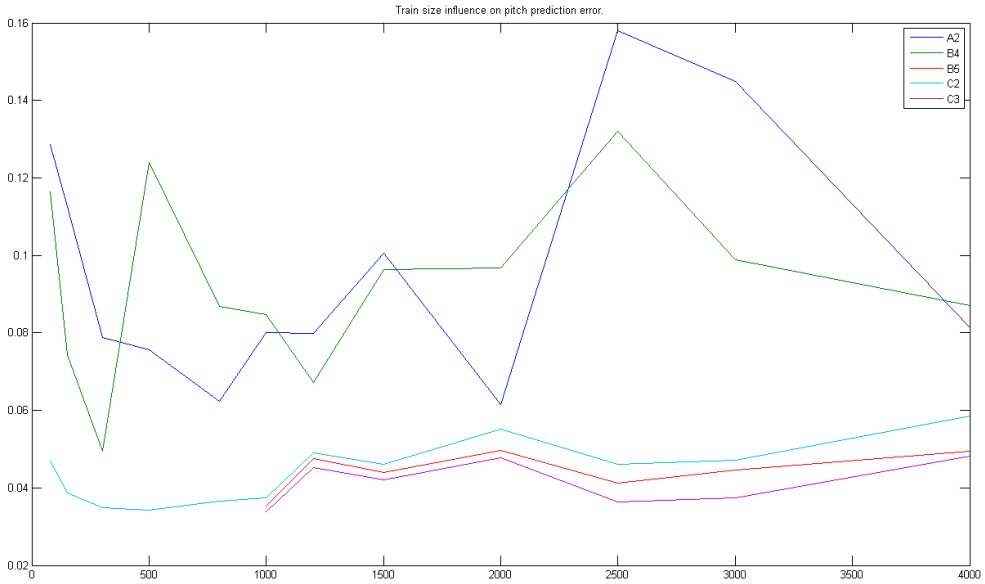

Figure 3.8: Learning curve of various methods. Prediction error of *pitch* depending on train size. Models with penalization terms do not learn with less than 1000 data records.

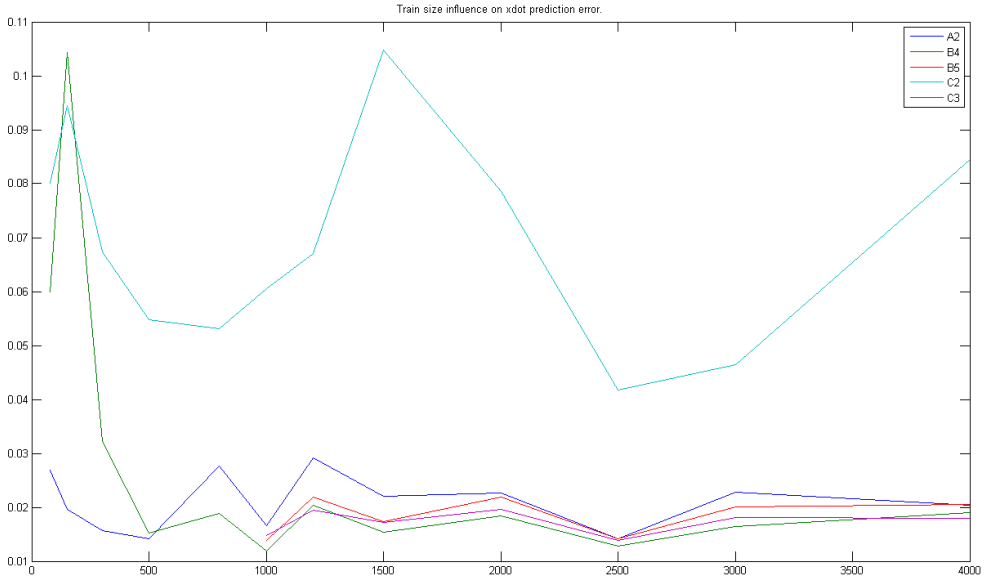

Figure 3.9: Learning curve of various methods. Prediction error of *xdot* depending on train size. Models with penalization terms do not learn with less than 1000 data records.

Validation set distance

We expected differences between various validation sets, mainly that the ones near the training set would have smaller errors. We performed an experiment where different distances from the training set were used (each distance for each method 5 times). It is true for some of the methods that a certain distance from the training set increases errors. Particularly method $C2$, which uses many free parameters and does not use the penalization model is more sensitive to validation set changes(see Figure 3.11). However, most of the methods do not depend on the validation set distance much, so the setting of this parameter could be arbitrary. We used distance 1000 mostly.

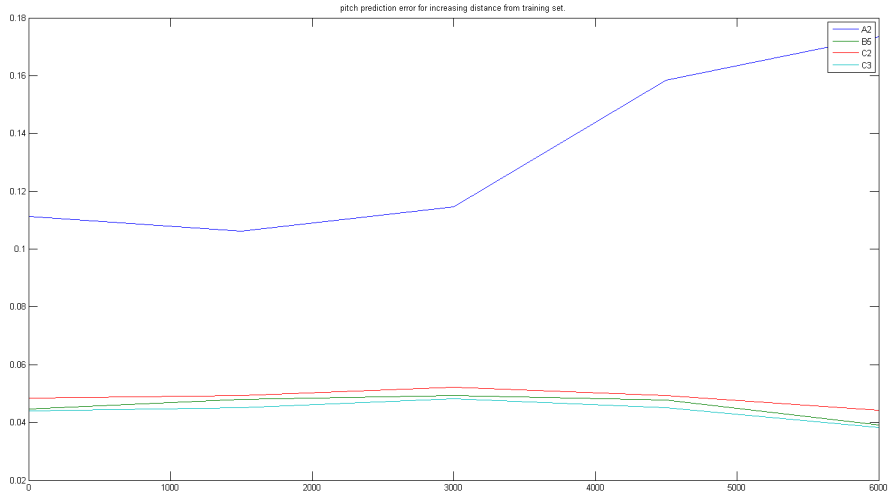

Figure 3.10: Prediction error of *pitch* depending on distance from train set (models $A2$, $B5$, $C2$ and $C3$).

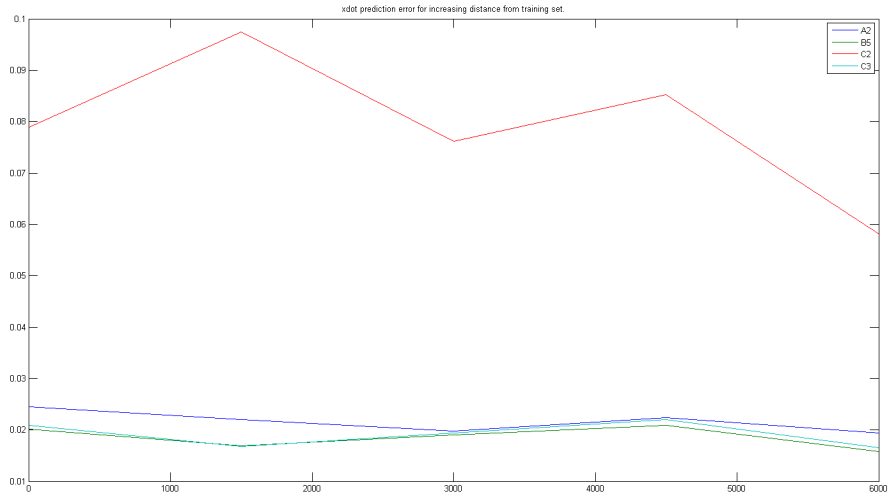

Figure 3.11: Prediction error of *xdot* depending on distance from train set (models $A2$, $B5$, $C2$ and $C3$). Penalization models are more stable.

3.3.3 Experiment A - linear regression learning

In Experiment A we compared the preprocessing steps of learning the basic parameters $(p_0, ..., p_3)$ by linear regression. We were interested in the effect of smoothing the values before applying the regression. The calculated the likelihood and prediction was performed using the original data (smoothed data were used only for learning).

- **A1** - parameters $p_0, ..., p_3$ learnt by linear regression using the measured data

Method	Train LL	Validation LL	*pitch* MSE	*xdot* MSE
A1.	13.51469771	13.51585250	0.1547	0.1081
A2.	13.51469725	13.51585221	**0.0196**	**0.1025**
A3.	13.51469725	13.51585221	0.0275	0.1100
A4.	13.51469775	13.51585253	4.6059	0.1088

Table 3.2: Summary of the Experiment *A*.

- **A2** - parameters $p_0, ..., p_3$ learnt by linear regression with smoothed *xdot*
- **A3** - parameters $p_0, ..., p_3$ learnt by linear regression with smoothed *xdot* and *pitch*
- **A4** - parameters $p_0, ..., p_3$ learnt by robust linear regression[26] using the measured data

For each method except for $A1$, value p_3 was estimated > 1 and we manually set it to 0.997. $A1$ did not learn reasonable values at all. The results suggest to use $A2$ as the baseline method for further experiments and we also use its learnt parameters as the initial values for EM algorithm. This basic method is also sufficient for filtering when all measurements are available. However, its accuracy in prediction can be improved by EM algorithm as shown in further experiments. Robust linear regression performed even worse on *pitch* prediction, because it learnt p_1 to be exactly 1 (probably because the sequences of identical values in data, steps were treated as outliers and assigned a small weight).

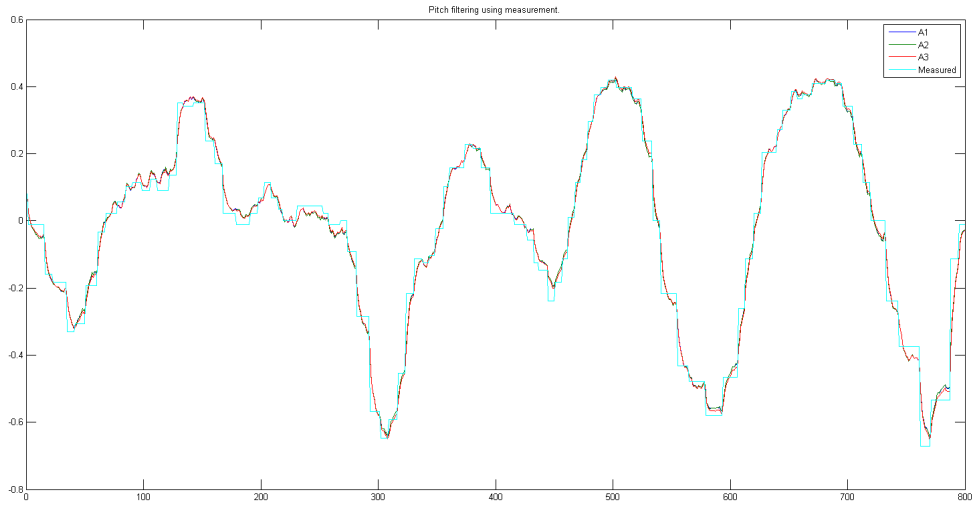

Figure 3.12: Filtering results of *pitch* for methods in Experiment *A*.

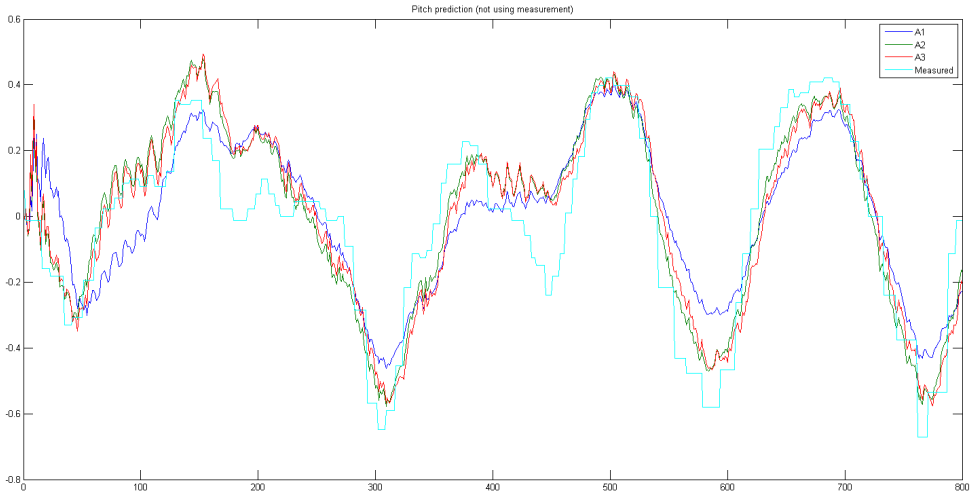

Figure 3.13: Prediction results of *pitch* for methods in Experiment *A*.

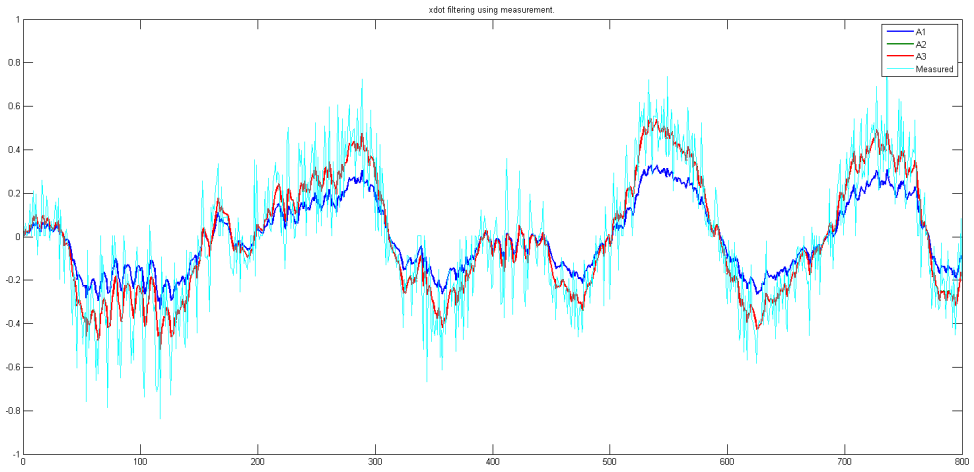

Figure 3.14: Filtering results of *xdot* for methods in Experiment *A*.

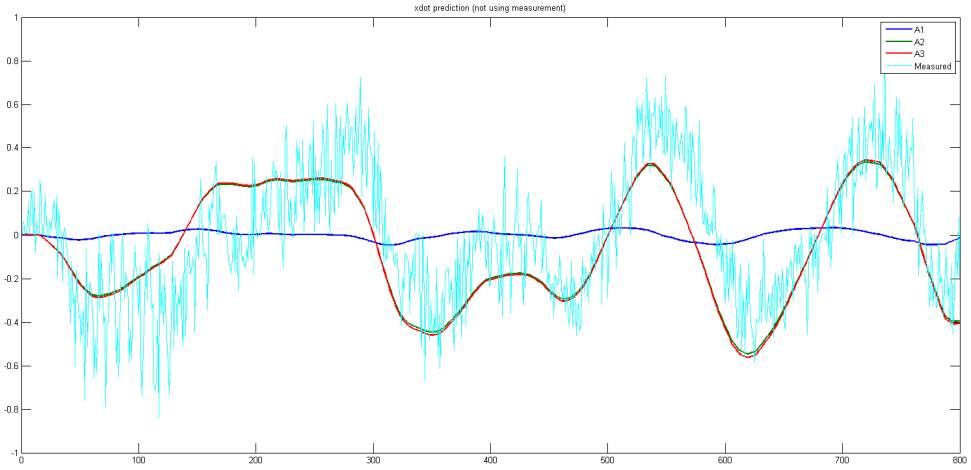

Figure 3.15: Prediction results of *xdot* for methods in Experiment *A*.

3.3.4 Experiment *B* - EM algorithm

The goal of Experiment *B* is to compare various configurations of EM algorithm. We would like to test which parameters are worth learning and for which it is better to keep their values fixed. The first reason is overfitting. Secondly, some of the parameters do not satisfy the Kalman filter assumptions and learning their values leads to wrong results. For instance, *pitch* does not have Gaussian noise and Kalman EM (assuming that it has Gaussian noise) learns a very small variance. Then the filtering results almost copy the measured values. In such cases, likelihood is not an appropriate measure of accuracy. Generally, likelihood is always the highest when all the parameters are free.

Our approach was first to try various configurations and propose new ones according to the performance in prediction of missing measurements. We let the EM algorithm learn all the parameters for various subsequences of the data and then rejected the parameters that have different values for different data. For example, we found out that all other positions in transition matrix A from $p_0, ..., p_3$ are irrelevant, except for $A(1, 2)$, which improves the *xdot* prediction with missing measurements. The best approach, which deals with overfitting automatically, seems to be the model $B5$.

In our model, we treat *elevator* as a measurement instead of control signal and the state is extended by a new variable. The first idea was that the corresponding line of the transition matrix nor variances should not be learnt because we should expect the control signals to be arbitrary and not dependant on the current state. However, trying to learn these parameters increases the performance when used with penalization model (the effect of penalization can be seen in Section 3.3.5 comparing model C2 to C3). We can view the *elevator* state as "effective elevator" and learning its parameters as learning some unknown properties of the system. More variable data would help to evaluate this approach.

- **B1** - EM for parameters $p_0, ..., p_3$ and variance of *pitch* and *xdot*

- **B2** - EM for parameters $p_0, ..., p_3$ and variance of only $xdot$
- **B3** - EM for $p_0, ..., p_3$, $A(1,2)$ and variance of $xdot$
- **B4** - EM for $p_0, ..., p_3$, $A(1,2)$ and row of Q and R corresponding to $xdot$
- **B5** - EM for full matrices A, Q, R with penalization model

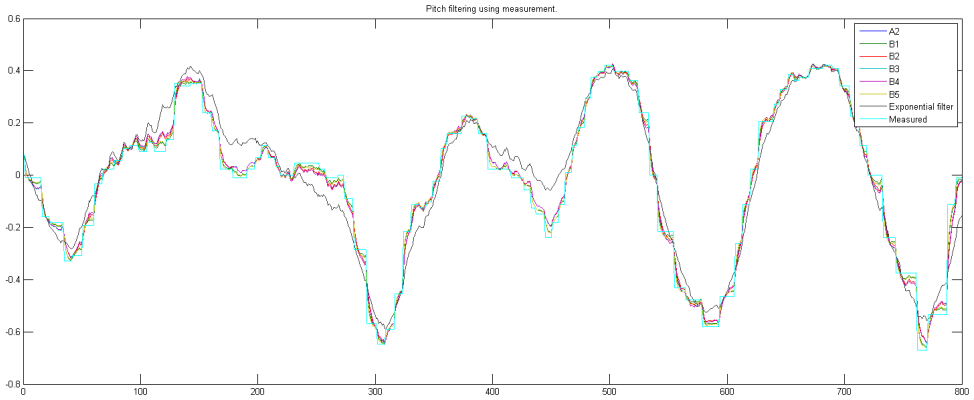

Figure 3.16: Filtering results of *pitch* for methods in Experiment B.

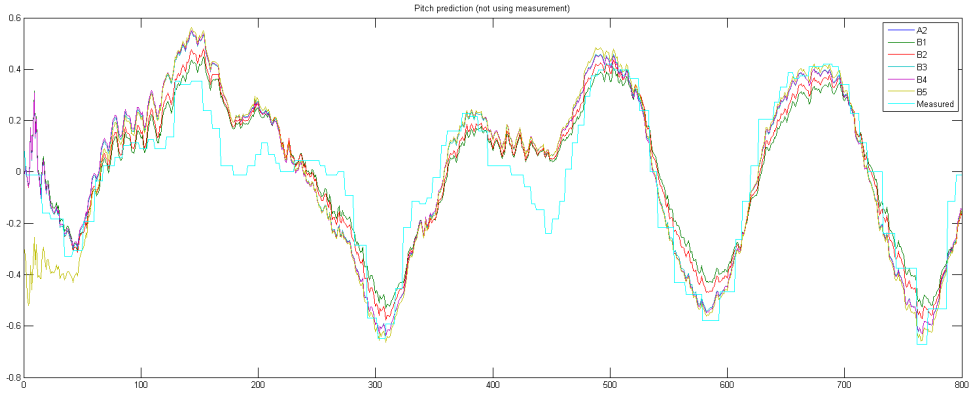

Figure 3.17: Prediction results of *pitch* for methods in Experiment B.

Method	Train LL	Validation LL	*pitch* MSE	*xdot* MSE
A2.	13.515	13.516	0.019633	0.10255
B1.	13.964	13.965	0.02929	0.10355
B2.	13.403	13.404	0.022813	0.1035
B3.	13.390	13.391	**0.016476**	0.095054
B4.	13.375	13.376	0.016617	0.088698
B5.	**14.735**	**14.737**	0.018308	**0.04536**

Table 3.3: Summary of the Experiment B

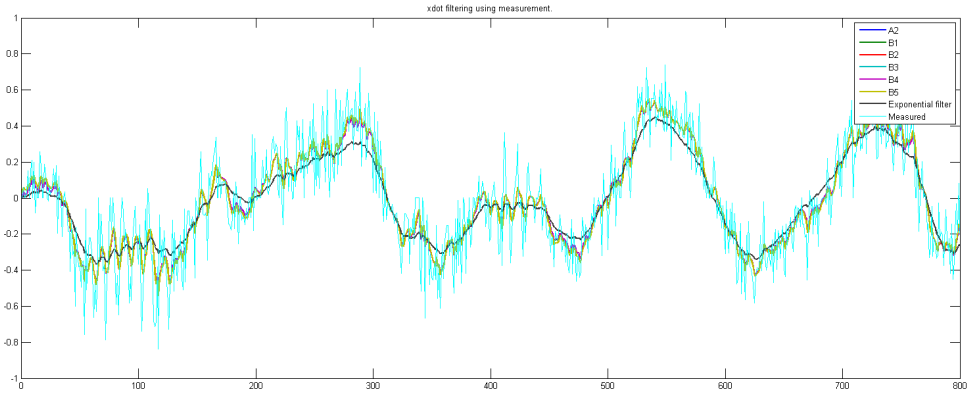

Figure 3.18: Filtering results of *xdot* for methods in Experiment B.

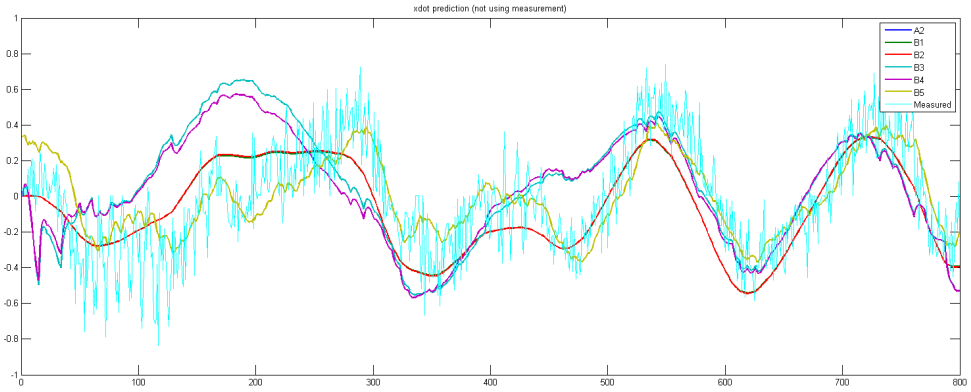

Figure 3.19: Prediction results of *xdot* for methods in Experiment B.

3.3.5 Experiment C - extended model with EM algorithm

In this experiment, we evaluated our extended model described in Section 3.2.2 and transition matrix (3.20). This is the only model where parameters do not have to be adjusted manually to be less than one (this was also true for a few other models on particular data). EM algorithm learning all positions in the transition matrix with penalization model leads to a stable model.

Method	Train LL	Validation LL	*pitch* MSE	*xdot* MSE
C1.	16.665	16.667	0.024451	0.083317
C2.	**20.172**	**20.174**	0.076379	0.048347
C3.	17.735	17.737	**0.019754**	**0.04366**

Table 3.4: Summary of the Experiment C

- **C1** - extended model, EM for parameters $r_0, ..., r_5$, $A(1,2)$ and covariance matrices except for parameters corresponding to *pitch* and *elevator*
- **C2** - extended model, EM for all A, R, Q except for *pitch* variance
- **C3** - extended model, EM for all A, R, Q except for *pitch* variance with penalization model

We can see that despite $C2$ has the greatest likelihood, $C3$ (which is the same plus using penalization model) generalizes better and has more accurate predictions.

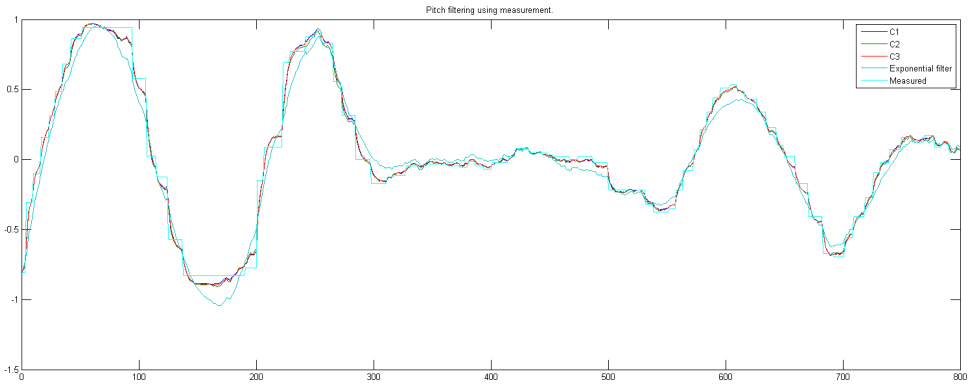

Figure 3.20: Filtering results of *pitch* for methods in Experiment C.

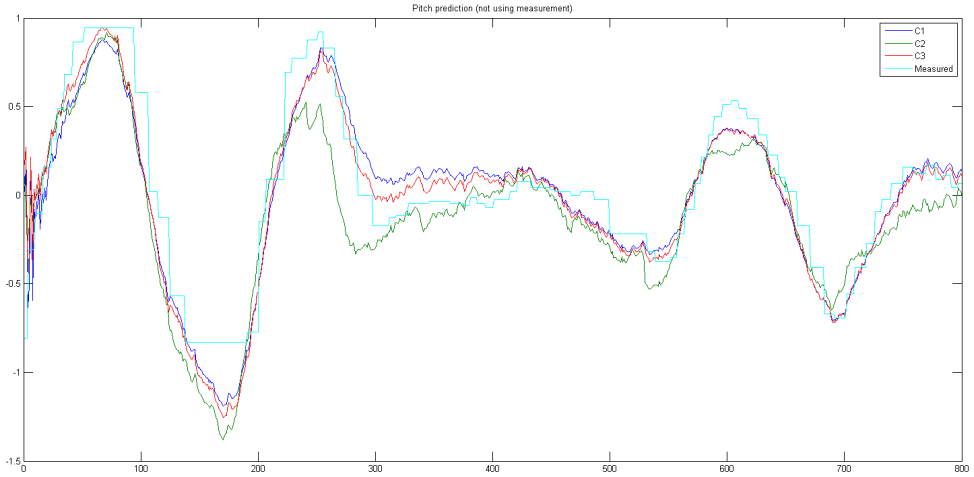

Figure 3.21: Prediction results of *pitch* for methods in Experiment C.

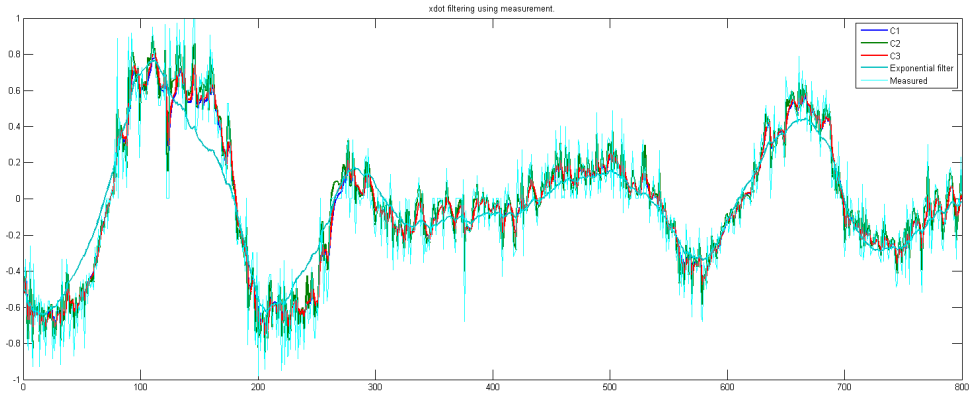

Figure 3.22: Filtering results of *xdot* for methods in Experiment C.

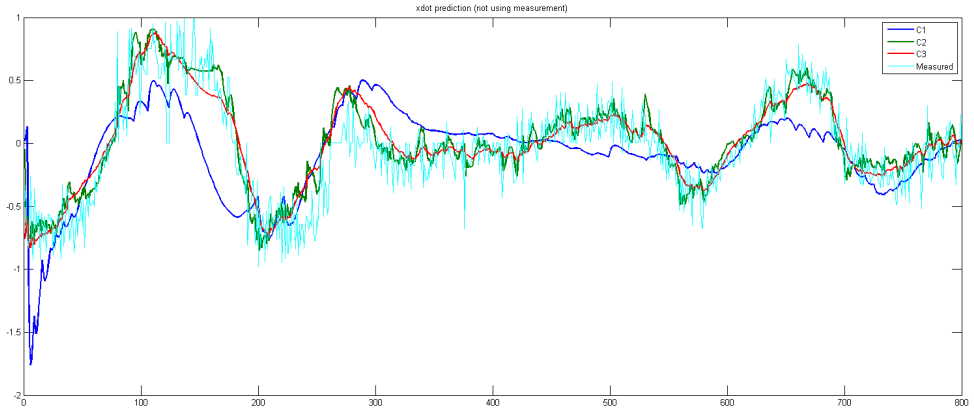

Figure 3.23: Prediction results of *xdot* for methods in Experiment C.

3.3.6 Experiment D - Particle filter

We used the same transition matrix, transition and measurement variances as learnt by $B5$ and applied for Particle filter with 1000 particles. The results of the Particle filter were almost the same, but slightly worse. The difference is not only in approximating the posterior by particles, but our Particle filter does not sample nor weights using full covariance matrices Q, R, just their diagonals.

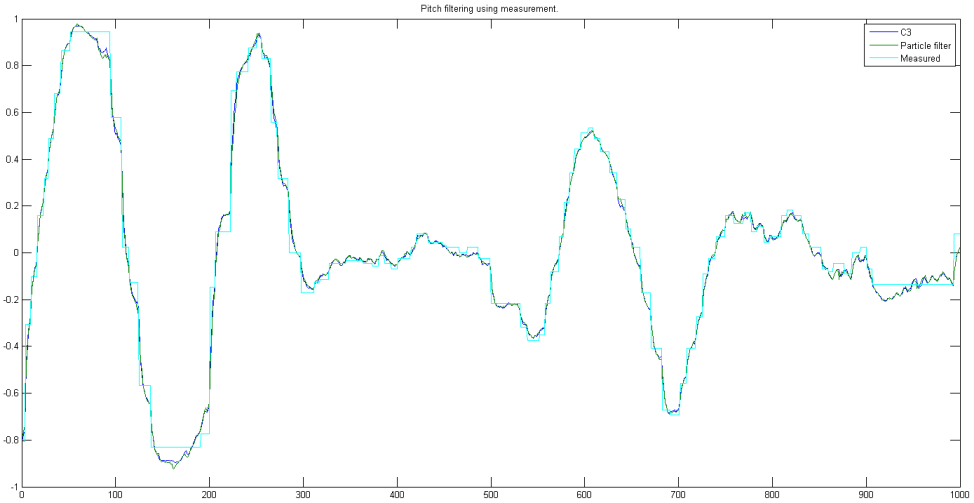

Figure 3.24: Particle filter compared to $C3$ (using same model) in *pitch* filtering.

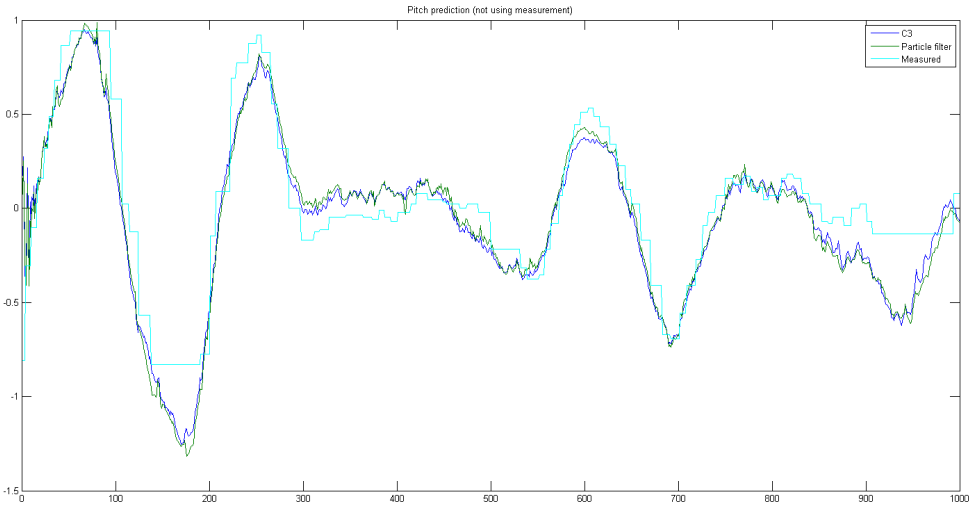

Figure 3.25: Particle filter compared to $C3$ (using same model) in *pitch* prediction.

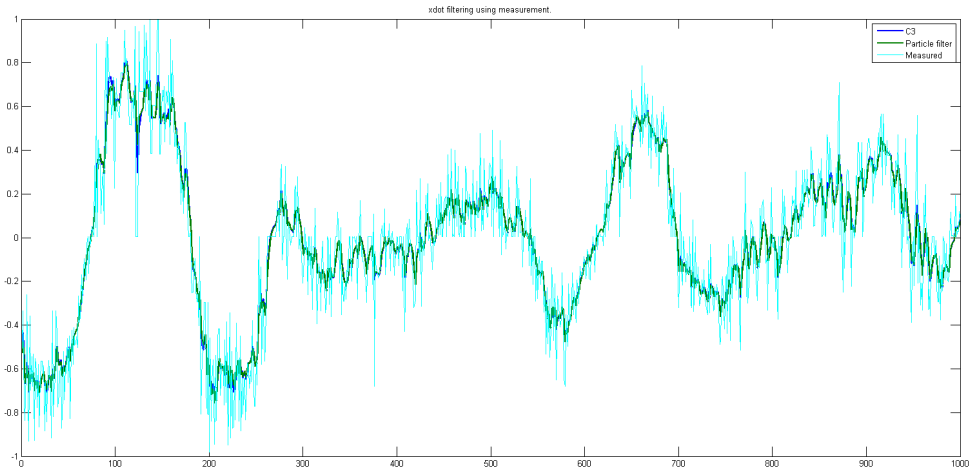

Figure 3.26: Particle filter compared to $C3$ (using same model) in *xdot* filtering.

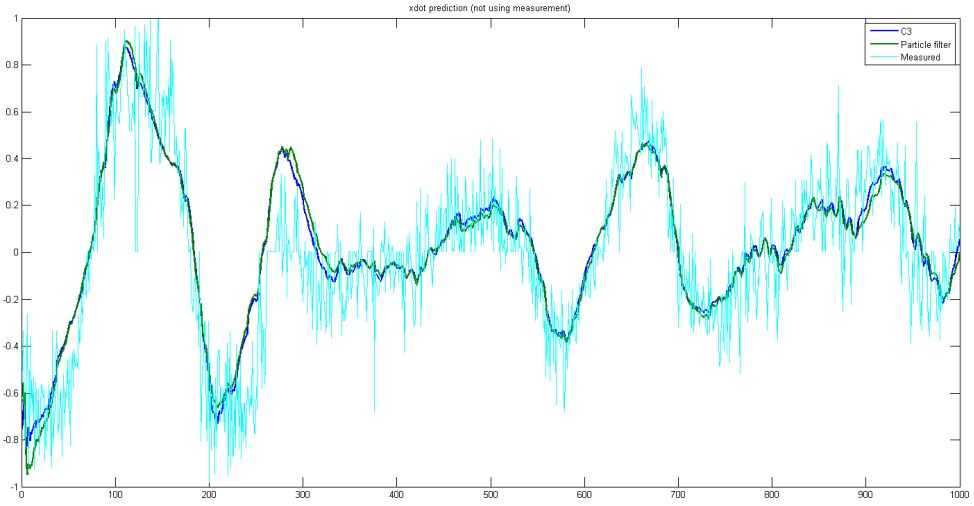

Figure 3.27: Particle filter compared to $C3$ (using same model) in *xdot* prediction.

3.3.7 Final evaluation

In the previous experiments we learnt the models and tuned them not to be overfitted using validation sets. All the data (including the validation sets) used so far was from the flight number 3. Now, we test the best models on separate datasets, i.e., flights 1, 2 and 4. The models reached similar results as on the validation sets. The various flights were not very different, the performance was already well estimated by validation sets in flight 3. The results are summarized in Table 3.5.

Method	pitch 1	pitch 2	pitch 4	xdot 1	xdot 2	xdot 4
A2.	0.0193	0.0359	0.0185	0.1862	0.667	0.5099
B4.	**0.0173**	**0.0326**	0.0176	0.1172	0.107	0.1352
B5.	0.0174	0.0332	**0.0176**	**0.0569**	**0.0614**	0.0403
PF	0.0212	0.0350	0.0202	0.0572	0.0625	0.0409
C3	0.0182	0.0353	0.0186	0.0570	0.0619	**0.0401**

Table 3.5: Prediction error results of flights 1, 2 and 4 for the best methods including Particle filter (PF).

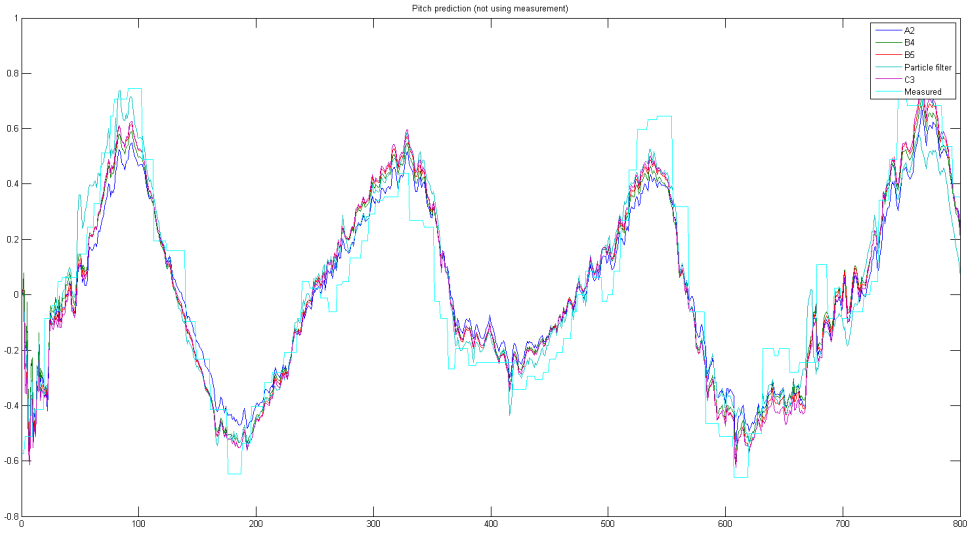

Figure 3.28: Prediction of *pitch* on unseen data from flight 1.

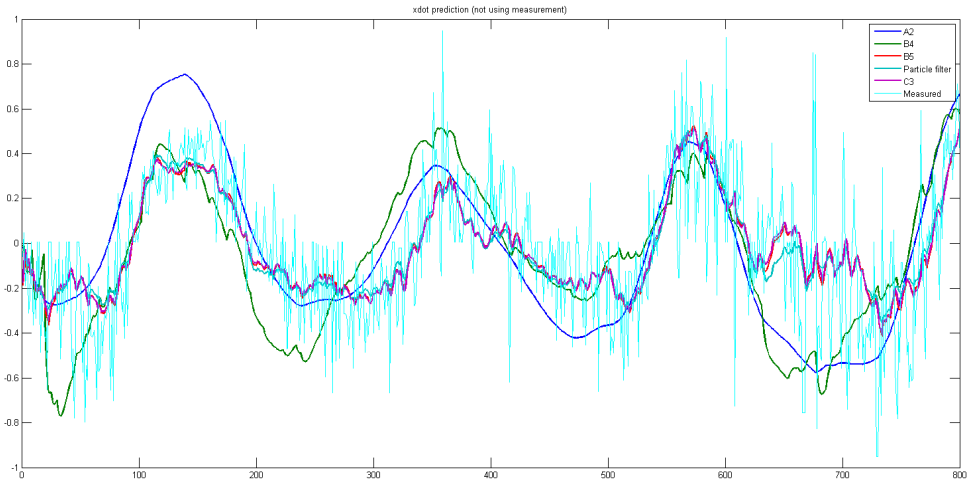

Figure 3.29: Prediction of *xdot* on the unseen data from flight 1.

Method	*pitch* error	*xdot* error
$C3$ scaled data	0.0177	0.0433
$C3$ original data	2.648	0.0564

Table 3.6: Comparison of the prediction results using scaled and original data.

3.3.8 Other experiments

Scaling

To show the effect of scaling the data on EM algorithm, we used the same learning method as $C3$ with its initial variances adjusted to fit the original data. We compared the performance of $B5$ with all data scaled to $[-1, 1]$ and the modified $C3$ on the original data. In original data, *pitch* and *elevator* are in range $[-10, 10]$ and *xdot* in range $[-1, 1]$, which makes a significant difference in the likelihood function (it is sensitive to covariance matrices). As EM algorithm is a gradient algorithm, it was misled by *xdot* variable. The comparison is summarized in Table 3.6.

Estimating κ parameters

To estimate the tradeoff constants κ, κ_2 and κ_3 we tried various values and compared the prediction performance on the validation sets. We started with 0.2 and continued with twice as large at each time step while the performance was improving. Then we performed a binary search between the last two values until a reasonable precision and convergence of the performance. We estimated $\kappa = \kappa_2 = 80$ and $\kappa_3 = 0$ (it did not improve at all).

Measurement offset

We designed a model with nonzero means of the measurement noises. The transition matrix is extended by a new state variable that always remains one (the corresponding positions were fixed for EM algorithm). The measurement matrix is extended by a column of measurement noise means. The learnt values were not significant and were different for various parts of the data, so we decided to simplify the model and use zero measurement noises.

Adding random noise to particles

In Particle filter, we generated 5% of the particles at each step randomly (with various distributions) in order to improve diversity. However, this modification degraded the performance, so we use the standard SIR Particle filter.

Mixture of Gaussians model

Adding a mixture of nonlinear functions to the transition model is too complex and we were not able to adjust the parameters in a reasonable time. The first problem is to spread the means of Gaussians across the state space. As the state space is at least 3 dimensional, it was too complex (especially for *MATLAB* implementation) to find a solution with 4 Gaussians per dimension. Moreover,

it is still necessary to tune their covariances and it is still very likely that such a model would not generalize well.

Higher order Markovian system

We tried to learn a model that would predict the next state from k previous observations (e.g. $k = 20$). We also tried to introduce nonlinearities to these predictions using a feed-forward neural network. However, this model did not maintain the current state distribution and performed poorly. This suggests that transforming a dynamic problem into a static one with a sliding window is not appropriate.

3.4 Conclusion

Best models description

The best models were shown to be $B5$ and $C3$. Both are learnt by EM algorithm with penalization model ($\kappa = \kappa_2 = 80$) on first 1000 data records of flight 3. Both have all positions in transition and covariances matrices free to learn, except for $C3$ having *pitch* variance fixed. $B5$ has state vector $(pitch, xdot, elevator)$ and transition matrix

$$A = \begin{bmatrix} 0.979574 & -0.025133 & 0.028671 \\ 0.029254 & 0.994377 & -0.008402 \\ 0.209937 & -0.869643 & 0.091070 \end{bmatrix}. \tag{3.19}$$

Model $C3$ has state vector $(pitch, xdot, xdotdot, elevator)$ and transition matrix

$$A = \begin{bmatrix} 0.976291 & -0.017484 & 0.037701 & 0.020246 \\ 0.011445 & 0.980655 & -0.007368 & 0.330526 \\ 0.197686 & -0.830580 & 0.133771 & 0.023545 \\ 0.047391 & 0.034580 & -0.000572 & 0.115297 \end{bmatrix}. \tag{3.20}$$

Covariance matrices of $B5$ are

$$Q = \begin{bmatrix} 0.001067 & 0.000042 & 0.000003 \\ 0.000042 & 0.001196 & 0.000005 \\ 0.000003 & 0.000005 & 0.176147 \end{bmatrix}, \tag{3.21}$$

$$R = \begin{bmatrix} 0.003341 & -0.000077 & -0.000101 \\ -0.000077 & 0.018085 & 0.000766 \\ -0.000101 & 0.000766 & 0.011445 \end{bmatrix}. \tag{3.22}$$

Covariance matrices of $C3$ are

$$Q = \begin{bmatrix} 0.001000 & 0.000052 & -0.000018 & 0.000014 \\ 0.000052 & 0.001336 & 0.000065 & -0.000244 \\ -0.000018 & 0.000065 & 0.170937 & -0.000040 \\ 0.000014 & -0.000244 & -0.000040 & 0.001730 \end{bmatrix}, \tag{3.23}$$

$$R = \begin{bmatrix} 0.010000 & -0.000179 & -0.000095 \\ -0.000179 & 0.016130 & 0.002276 \\ -0.000095 & 0.002276 & 0.004952 \end{bmatrix}. \tag{3.24}$$

Both models use identity as the measurement matrix H.

Summary

The basic filtering task with all measurements available is solved by Kalman filter. Its performance is better than using exponential filter, mainly because exponential filter slowly accommodates to new values. From the theoretical point of view, Kalman filter is more suitable as it describes the dynamic system more properly. It outputs sufficient results also with parameters basically learnt by linear regression, because the task is relatively easy with measurement correction at each time step.

The parameters can be better tuned using EM algorithm. This improves the model more significantly when prediction of missing measurements is required. However, EM algorithm should be used carefully and only for the parameters that satisfy its assumptions (mainly the Gaussian noise), or adapt a penalization model to avoid overfitting.

Using Particle filter is possible, too. However, it does not improve the model unless we know certain nonlinearities or non-Gaussian densities of the model.

4. Other datasets

4.1 Vicon data

We received data from an experiment called Vicon. It was performed using a special system for tracking objects. Three drones were flying in a room and a set of external cameras were recording. The system can provide very accurate positions of each drone for each time step.

Moreover, each drone has its own camera to detect other drones. This camera system can detect black circle on a white background. Each drone carries this sign hung on itself. The detection work as follows. It starts scanning the camera image from the position of previously detected object. As soon as it finds the object, it calculates its distance estimation based on its size and relative angle. It outputs a new record with the distance estimation and x and y coordinates of the object on the camera picture. This record is then joined with the drone's real position obtained by the Vicon system.

We have a file consisting of such records for each drone. The main data fields of a record were the following:

- **vicon_time** - system time in ms
- **relative_dist_cam** - estimated distance of an object seen by a drone's camera
- **x_coord_cam** - x coordinate where the object was seen by a drone's camera
- **y_coord_cam** - y coordinate where the object was seen by a drone's camera
- **x_vicon** - x coordinate where Vicon detected the drone
- **y_vicon** - y coordinate where Vicon detected the drone
- **z_vicon** - z coordinate where Vicon detected the drone.

We do not know which drone was detected by the camera. However, we can estimate that by joining the files. For each drone we create a joint log by joining each its record with the closest (in time) records of both other drones. Let the drones be called 1, 2 and 3. In a joint log, the other drones are denoted A and B, A is the one with the smaller name (for instance in joint log of drone 3, drone 1 is A and drone 2 is B). We compute the real distance of A and B from the current drone (based on vicon coordinates). We get a joint record like this:

- **vicon_time**
- **vicon_time_A**
- **vicon_time_B**
- **dist_cam**
- **dist_A**

- **dist_B** .

First, we checked if joining the data on time is valid. We calculated time match errors, that is the differences between *vicon_time*, *vicon_time_A* and *vicon_time_B* for each record. On average, the time match error is 100ms, but most of them are around 10ms. Therefore, we can consider this matching as valid.

Now, we can plot the data for each drone.

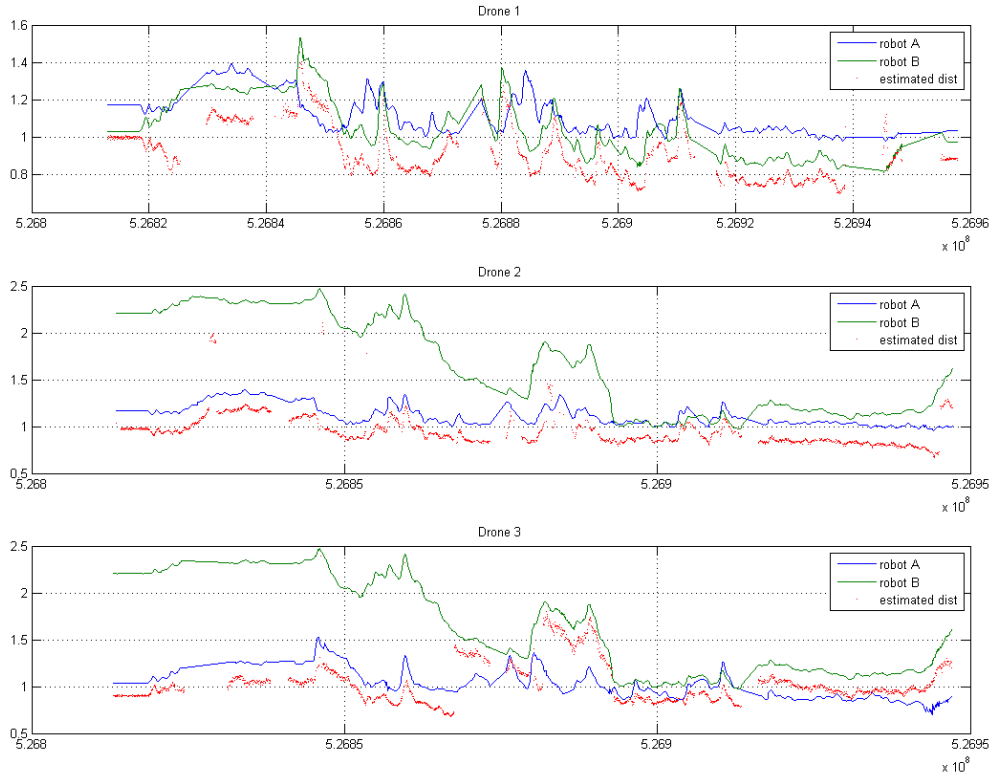

Figure 4.1: Real distances of drones and the estimates in time.

We can see that the detection module underestimates the distances by a constant plus a small random noise. To plot a histogram of measurement errors, we have to find out which drone was most probably seen. Let err_A be the distance error if the drone sees the drone A. That is $dist\_cam - dist\_A$. Similarly for seeing drone B. Then we define the distance error as the one with smaller absolute value from err_A and err_B. Note that the error is signed as we want to know also if the measurements are underestimated or overestimated. This definition of the distance error is not perfect, because we cannot be sure which drone was actually seen. However, it should be sufficient enough to design the measurement model. The distribution of the real distances is a Gaussian centered in a point with a fixed offset from the measured value.

We also tried to learn a model that would predict the error from camera coordinates (for instance, objects seen on the edges of the camera view have less accurate distance calculations). However, there was not enough data to

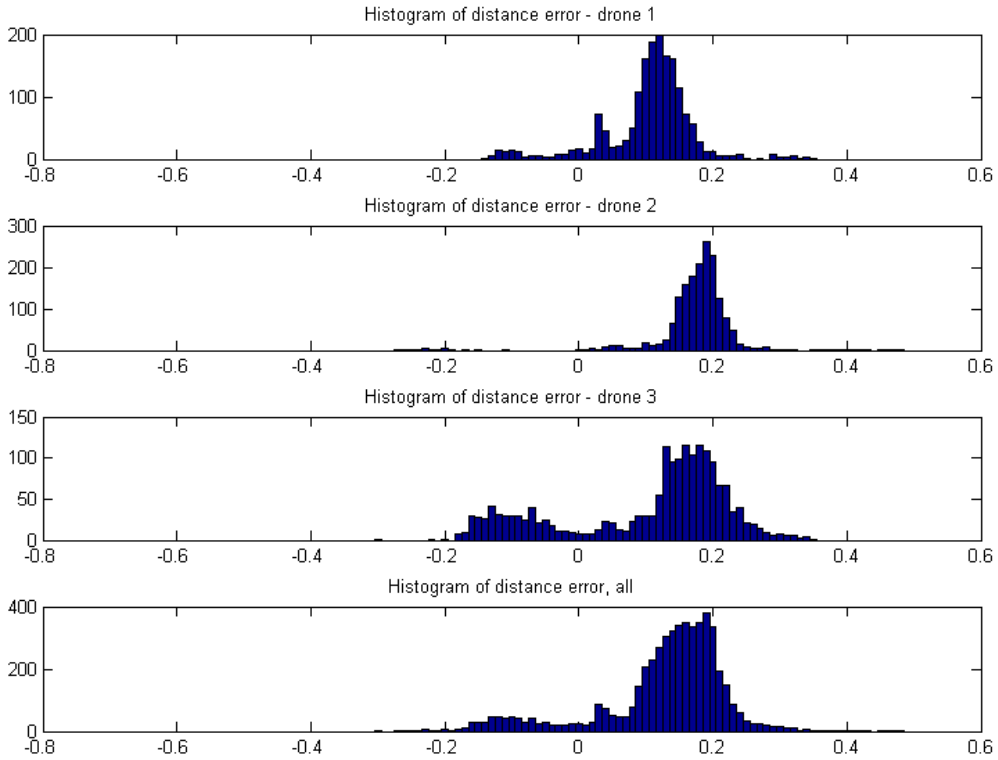

Figure 4.2: Histograms of distance errors for each drone.

distinguish various regions of the camera image. Almost all the records were from the center of the image.

4.2 Ground robots

For testing our implementation of Particle filter on a different type of robot, we used data from a ground robot. Our task was to filter the robot's position using its control signals, GPS and compass measurements. The robot was moving in a town with a known map of streets. Using that, true location was provided for us. We could learn the parameters of the model from that and we used it as a measure of accuracy of our filter.

4.2.1 Problem description

The state of the robot is described by its position and orientation. The position is measured in meters relative to a certain point (0, 0) in the town and the orientation is in radians. Control signals consist of *alpha*, *dist* and *beta*. A single control signal means that the robot is requested to rotate with angle *alpha*, then move forward *dist* meters and then rotate with angle *beta*. Based on this we have the following (nonlinear) transition model:

$$
\begin{aligned}
x_{t+1} &= x_t + cos(angle_t + alpha_t)dist_t \\
y_{t+1} &= y_t + sin(angle_t + alpha_t)dist_t \\
angle_{t+1} &= y_t + alpha_t + beta_t.
\end{aligned} \tag{4.1}
$$

The robot receives two kinds of measurements, GPS and compass. The measurements are asynchronous, they do not come together, nor together with the control signals. There is approximately 10 compass measurements and 10 control signals for a single GPS measurement. Moreover, it takes some time to initialize the GPS sensor, therefore there is no GPS data at the beginning (first 5%).

Compass shows the orientation of the robot (corresponds to the state variable *angle*) in radians. GPS shows the latitude and longitude that can be easily converted to our metric system during the preprocessing step[1].

The data also contains the expected accuracy for each measurement of both GPS and compass, but the analysis showed that these values are useless.

4.2.2 Data analysis

When true state values are provided, we do not need to use EM algorithm as its estimation phases can never produce better estimate than we already have. However, we should try to analyze the data and perform a single maximization step based on that.

GPS

We plot the real position and GPS measurements first as a planar image to see the trajectory in 2D (Figure 4.3). Then we plot the distance error in time (Figure 4.3). GPS shows wrong data in the middle part. For the rest of the time, it provides a reasonable estimate of the position. 4.3 also shows that expected GPS accuracy does not correspond to the real error. Finally, we generate a histogram of GPS error (Figure 4.3). If we ignore the cases where the GPS is absolutely wrong, the error can be approximated by a normal distribution.

Our first approach was to set the variance large enough so that the particles with the correct position would have non-zero probability in case of GPS being wrong. However, setting the variance low (5m) as it really is in case of GPS working correctly leads to better performance. This approach works, because if the GPS is working, it provides a better estimation and if it stops working, all the particles have zero weights and are sampled just according to the control signals. This assumes that the first sequence of GPS measurements is correct and that the sequence of wrong measurements is not too long (control signals are sufficient to lead the particles without correction). These properties hold in our data.

Compass

Compass values are given in radians. Some of the values from compass sensor are greater than 2π so we have to compute the modulo value. We also have to be careful with comparing angles near 0 (0.1 is close to 6.2 in radians).

[1]For the map projection details see [9]. In MATLAB implementation, we used a function from [10]

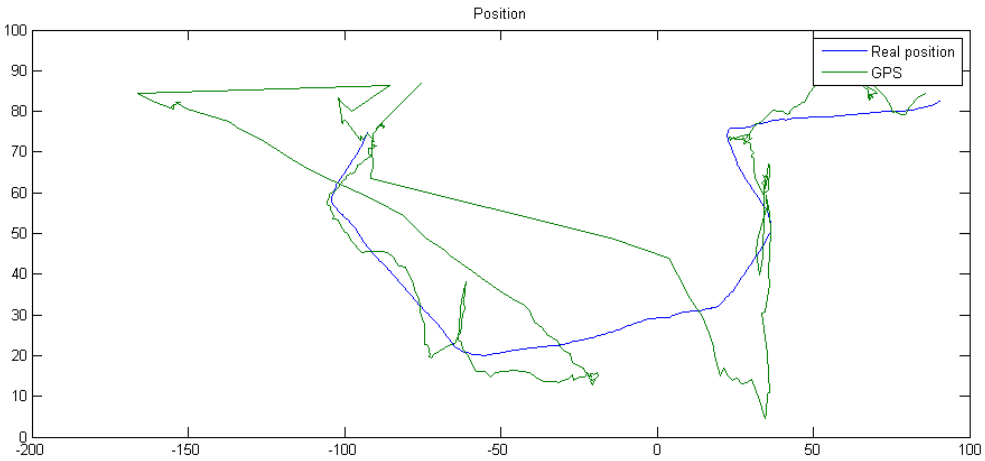

Figure 4.3: Real trajectory in 2D with measurements of GPS. The beginning of the trajectory is not plotted as there is no GPS measurement available.

By plotting the real angles and compass values, we can propose the measurement model for compass. Figures 4.6 and 4.7 show that there are 5 separate regions of compass values, each with a different offset from the real value. Based on this, we represent the measurement model with 5 normal distributions. For example, measurements for real angles between 4.5 and 6 radians are distributed normally with mean $angle - 0.25$. Histogram suggests low variances about 0.15, but higher variances (e.g 1) work better in practice.

Identifying such regions may seem as overfitting. However, there is a recurring region in our data, time 1500 to 2500 and then 3500 to 4500, and both of the sequences have the same characteristics.

Representing the compass error distribution by a single gaussian requires either a fixed averaged offset (0.5 radian) or higher variance (2 or 3 radians).

Transition model

The deterministic form of the transition model is described by equations (4.1). There are two main method how to introduce uncertainty to this model. We can either take the deterministic values of the new position and add a random noise to each of them (x, y and angle), or add the noise to the control signals and calculate the new position using these noised signals. For both of the method, we can calculate the appropriate noise distributions using the real values. The first method is more straight-forward but less plausible.

Using the first approach, we compare the real positions to the positions calculated by our deterministic model. Figure 4.8 shows the error of the model. The error is not Gaussian. The error is very close to zero in significantly more cases. We propose a natural hypothesis that the error is dependant on the control signals. The greater the distance to move or angle to rotate, the greater is the expected error. The simplest approach is to handle the control signals with zero distance or zero rotation separately and not to include any noise in these cases. More generally, we try to learn the expected error given the control signals. We use a linear regression to fit the parameters. The error plots suggest that the

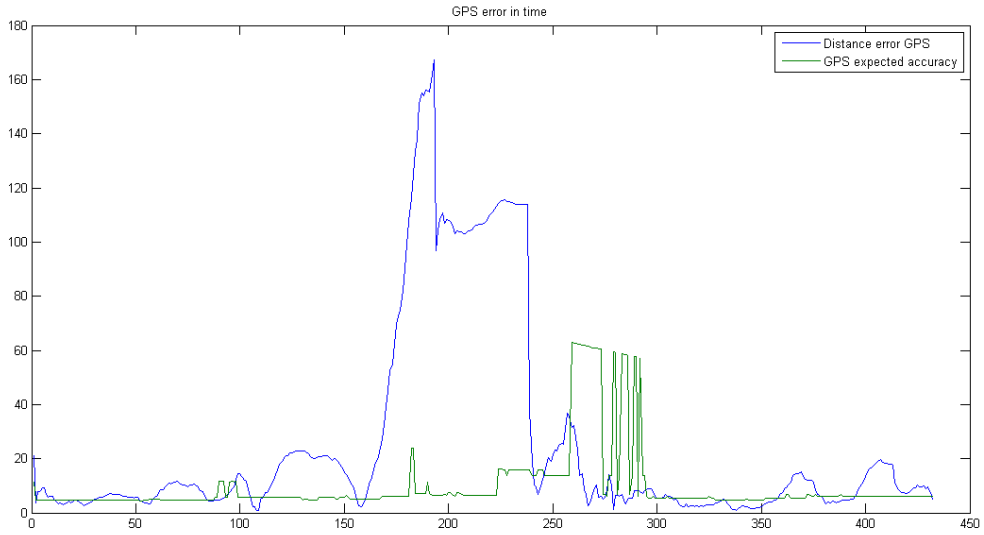

Figure 4.4: GPS distance error in time with its expected accuracy (given as radius of expected position). We see it does not correlate with the real error.

error may grow quadratically with the distance, therefore we added the distance square to the predictors. Considering also the current angle did not prove useful. Finally, we found the parameters that predict the absolute value of the error of x and y coordinate given $dist$ and $dist^2$ and the error of angle given $|alpha + beta|$. We intend to use these predicted values as the variances in the transition model. However, for better performance of the particle filter, multiplying the predicted values by some hand-tuned constants is necessary.

In conclusion, each transition step computes the deterministic values of the new position according to (4.1) and adds a random noise distributed normally with zero mean and variance calculated based on the current control signals.

4.2.3 Asynchronous measurements

In the standard filtering task, we assumed we receive control signals and all measurements at once and in a fixed sample rate. In a more general and more realistic setting, the measurements are received asynchronously. That is, for a given time there is a single message from one of the sensors or a control signal. In our data we assume that no control signal is missing and the robot moves only according to the control signals. There is no timestamp in either measurements or control signals.

We used a modification of particle filtering algorithm that follows a simple set of rules. Each input record can be either control signal or a measurement. If it is a control signal, we perform a transition step for each particle and output our state estimation. If it is a measurement, we perform a measurement step. That is, we assign a weight to each particle according to the model of the corresponding sensor (GPS or compass) and resample.

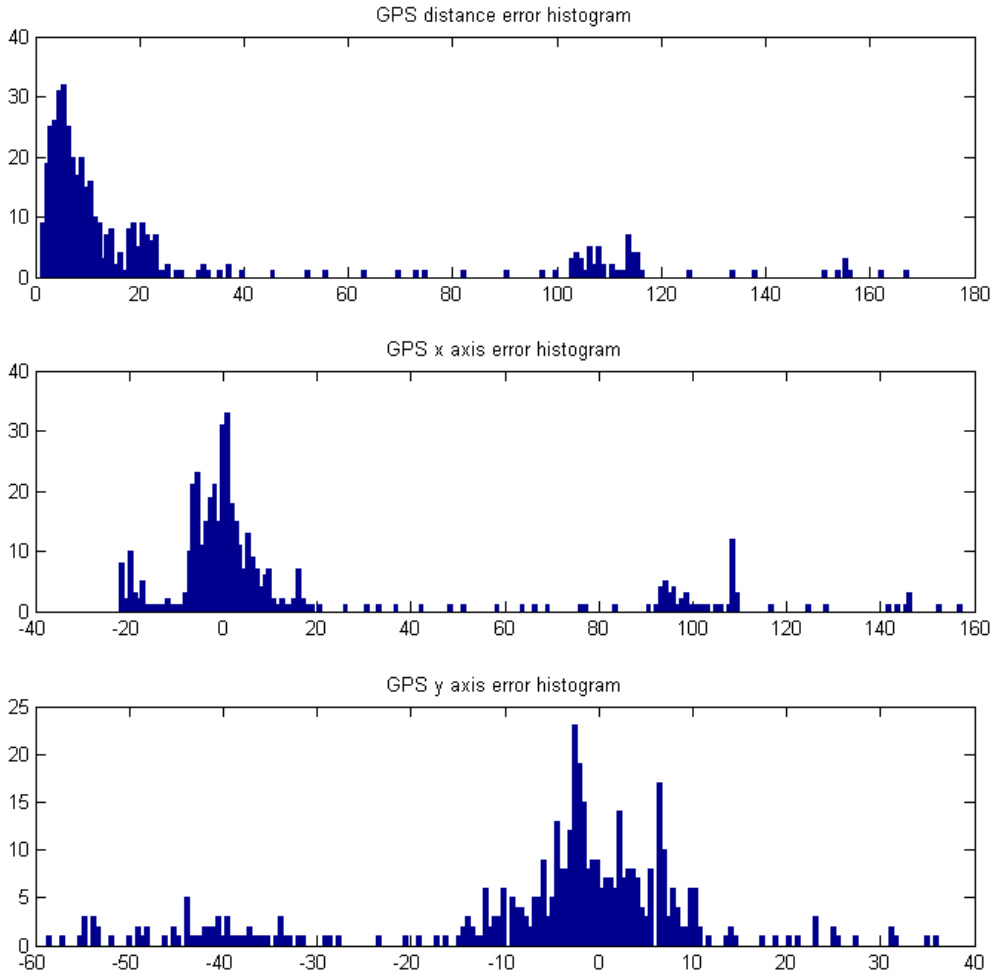

Figure 4.5: Histograms of the GPS error.

4.2.4 Results

Figure 4.9 shows the best performance of our particle filter. Compared to the real positions, the filter has mean error of approximately 6 meters. It also depends on the initialization. We use two types of initialization. The first (on the figure) initializes the particle randomly a few meters around the real start position. The second initializes the particles into a greater area, a rectangle covering all future GPS measurements. That is 250 times 100 meters.

In conclusion, there are a few things that improved the performance the most. First, setting the GPS variance low instead of setting it as a variance that includes absolutely wrong measurements. Second, realizing that the orientation values have to be compared in angle distance sense. Third, analyzing the compass error and finding the best offset. Fourth, identifying the transition error dependance on the control signals.

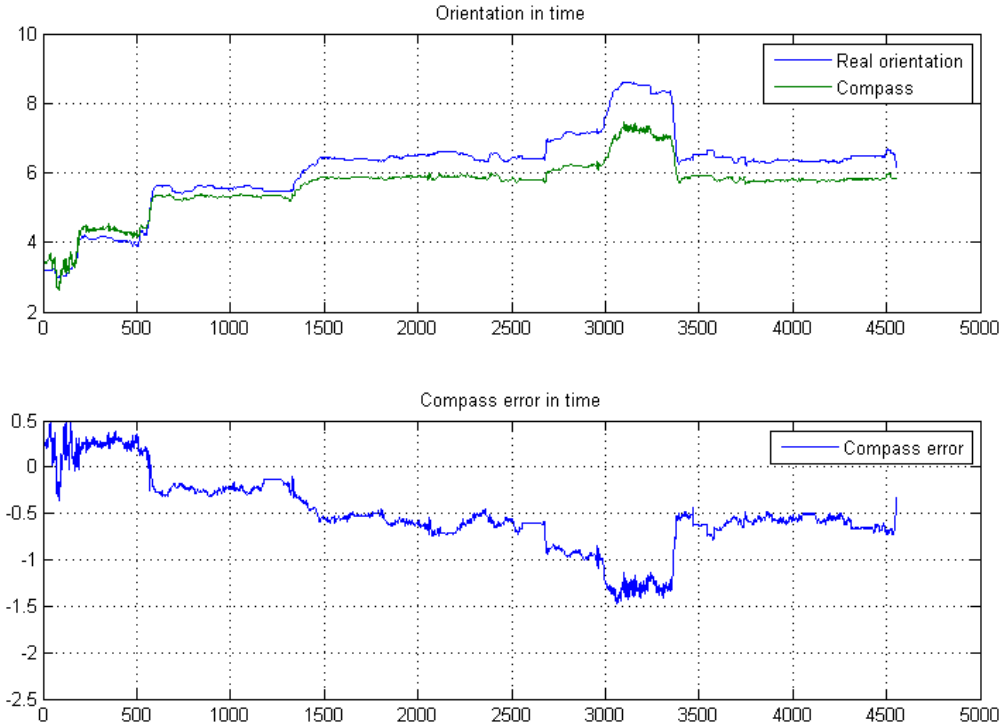

Figure 4.6: Compass values and real orientation in time. The values are in radians, values between 0 and 2 were moved to $[2\pi, 2\pi + 2]$ to improve the plot readability.

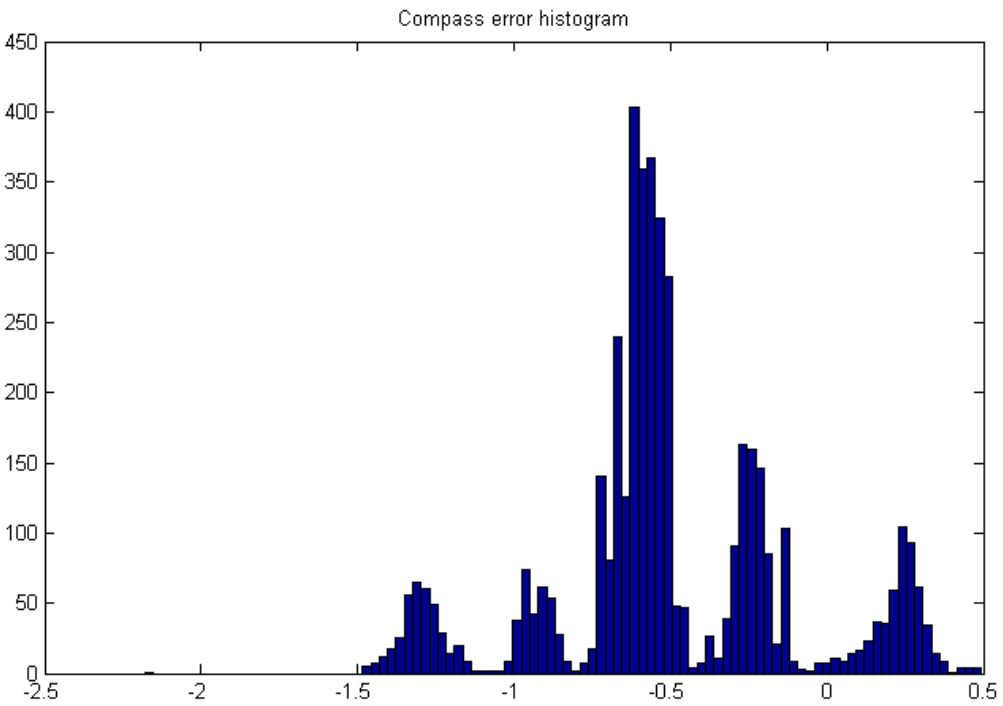

Figure 4.7: Histogram of compass error. The error is signed, positive error means greater measurement than real value.

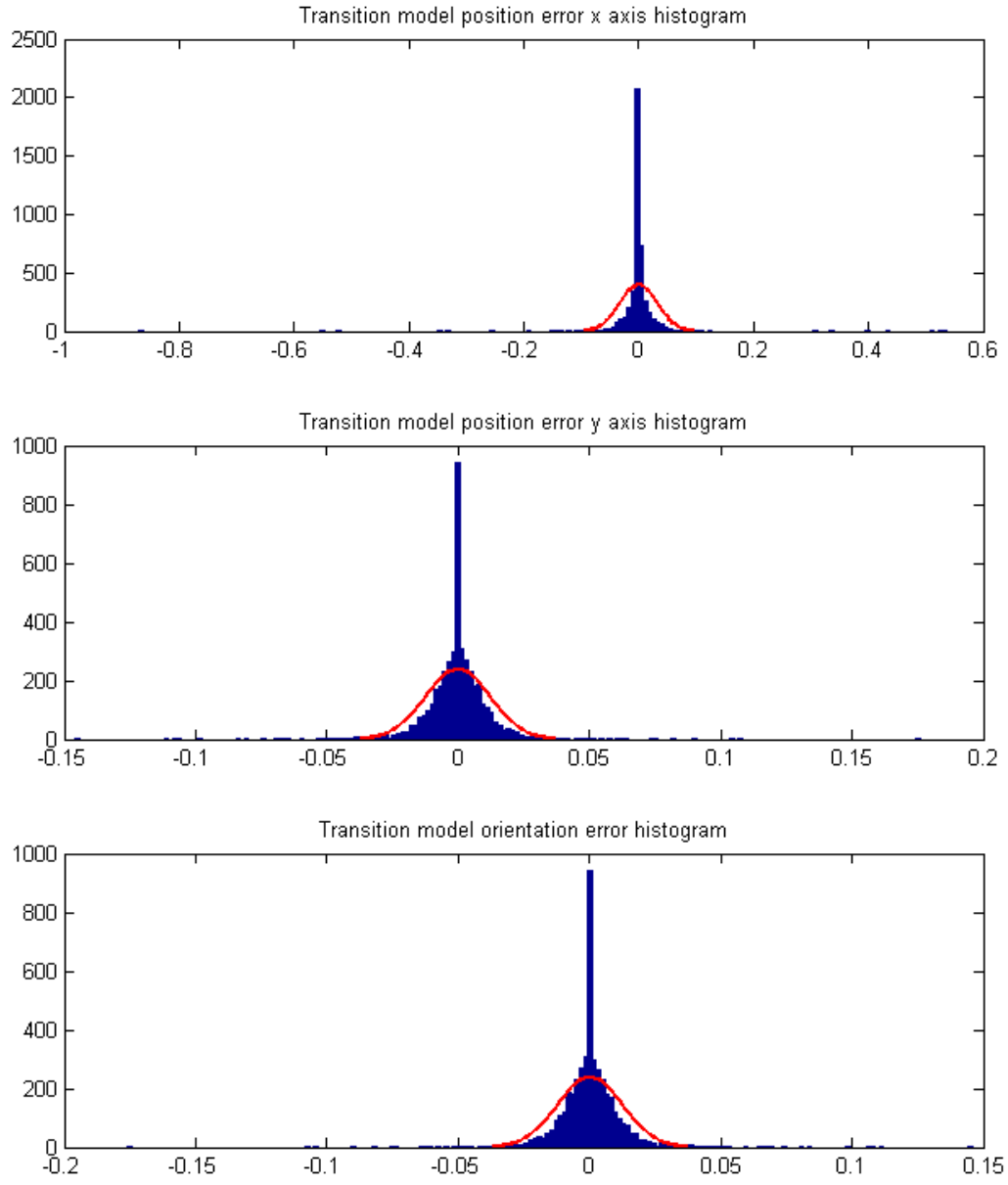

Figure 4.8: Histograms of the transition model error.

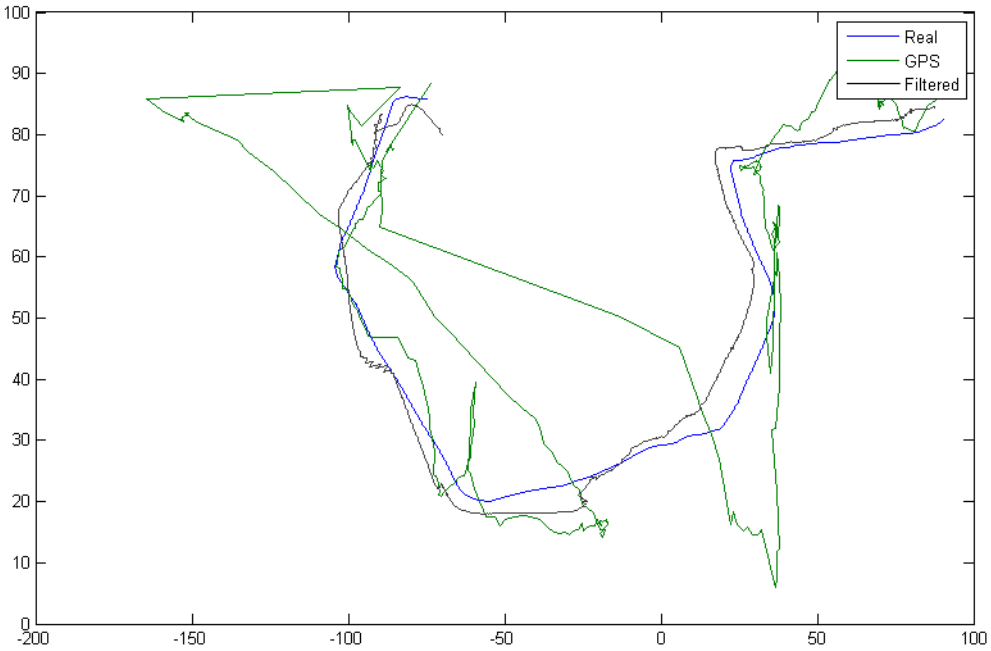

Figure 4.9: The best filtering performance of our methods, 2D trajectory.

5. Implementation

We implemented all the methods using *MATLAB*. In the following sections, we describe the implementation of Kalman filter and smoother, Particle filter and smoother, EM algorithm and evaluation methods. The last section summarizes the scripts that are prepared for testing these methods.

5.1 Kalman filter

Function `kalman_filter(Z, A, H, Q, R, P, x, include_backward)` serves as Kalman filter if `include_backward` parameter is set to `false` and as Kalman smoother if the parameter is `true`. `Z` is a matrix containing all the measurements, each time in a single column. `A, H, Q, R` are standard Kalman filter parameters (transition and measurement matrices and transition and measurement error covariance matrices). `P` is the prior covariance and `x` is the prior state (column) vector.

The function returns the following list: `[X_apr, X_post, X_smooth, P_post, P_smooth, L_smooth]`. The first 3 output variables are matrices of prior, posterior and smoothed (respectively) state estimates for each time step. `P_post` and `P_smooth` are the posterior and smoothed covariances for each time step (3 dimensional matrices with the last dimension of time). `L_smooth` contains matrices L for each time step, as described in Kalman smoother algorithm (Section 2.2.3). All the output parameters are required as inputs for EM algorithm. If not using this function within EM algorithm, `X_post` is the filtering output and `X_smooth` is the smoother output and the rest of the output values can be ignored.

Script `run_task_I_kalman` demonstrates the usage of Kalman filter. The script loads the data from a flight, prepares them into the matrix Z, creates some default parameters and runs the filter. Then it prints the likelihood and plots the filter results for $pitch, xdot$ and x along with the measurements.

5.2 EM algorithm

EM algorithm is performed by function `em(Z, A, H, Q, R, P, x, max_iter, variable_params, main_ind, kappa, kappa2, kappa3)`.

- The parameters `Z, A, H, Q, R, P, x` are identical to the parameters of `kalman_filter`, where `A, H, Q, R` are used as the initial parameters for EM algorithm.
- `max_iter` controls the maximum number of iterations of the algorithm.
- `variable_params` is a function handle (pointer to a function) that returns 4 lists: `[A_ind,H_ind,Q_ind,R_ind]`. Those are the indexes (to corresponding matrices) of parameters that should be learnt. The rest of parameters in matrices `A, H, Q, R` will remain fixed as given in the input parameters. For instance, we usually set `H_ind` to empty list and `H` to identity matrix.

- `main_ind` are the indexes to matrix `A` describing the parameters to which penalization should not be applied (penalization model is described in Section 3.2.4). Penalization of matrix `A` is implicitly disabled by setting `kappa` to 0.

- `kappa, kappa2, kappa3` are the tradeoff constants for penalization model of `A, Q` and `R`, respectively. Q_{def} and R_{def} are set to be the input parameters `Q, R`.

- The function returns the new learnt parameters `[A,H,Q,R]`.

The algorithm internally calls `kalman_filter` function (used as smoother), helper methods `maximize_A, maximize_H, maximize_Q, maximize_R` to perform the maximization steps and `convergence(A, H, Q, R, A_new, H_new, Q_new, R_new)` that decides whether the algorithm has converged. Convergence is here defined as sum of absolute changes of all parameters is below a certain threshold. We set this value to 0.005, but it may be useful to tune it for other applications.

Function `log_likelihood(X_apr, X_post, X_smooth, Z, P_smooth, A, H, Q, R, L)` computes the expected log likelihood normalized by the size of the data. It is not used by EM algorithm, because it directly uses the maximizing equations and does not need to calculate the actual likelihood value. We can use this function as orientation measure. We can obtain all the input parameters by calling `kalman_filter`.

5.3 Particle filter

Filter and smoother

Particle filter does not constrain the model to be just matrices, but it can be represented by arbitrary functions. We design the code to separate the main algorithm from the functions defining the particular model. The main functions required by `particle_filter` are: (they should be passed as function handles)

- `X_new = transition_model(u, X, params)` - generates a new set of particles given the current particles as rows of `X`, current control signal in row vector `u` and some optional parameters `params` (cell array).

- `W = measurement_model(z, X, params)` - returns a column vector `W` of weights for given particles `X`, row vector of current measurements `z` and some optional parameters `params` (cell array). The weights do not have to be normalized.

- `X = init_particles(z, n, M)` - creates the initial set of `M` particles of size `n` (state vector size). For generating the particles, it may use the first measurement (column) vector.

- `p = transition_probability(u, X, X_new)` - does not need to be defined for filter, but is required for smoother. Calculates the probability of transition from `X` to `X_new` given control signals `u`.

The main function for Particle filtering and smoothing is

```
particle_filter(Z, u, n, M, transition_model,
measurement_model, transition_prob,init_particles, params,
do_smooth).
```

Boolean parameter `do_smooth` enables smoothing. `Z` is a matrix of all measurement (row for each time step), similarly `u` for control signals. `params` is a cell array that is passed to functions `transition_model`, `measurement_model` and `transition_prob`.

It returns the following variables:

- `filtered` - a matrix of filtered state estimates (row for each time step)
- `smoothed` - a matrix of smoothed state estimates (row for each time step) or `NaN` if smoother disabled
- `Xs` - sets of particles for each time step (matrix $M \times n \times T$) or `NaN` if smoother disabled
- `Ws` - filtered weights for each time step (matrix $M \times n \times T$) or `NaN` if smoother disabled
- `Wss` - smoothed weights for each time step (matrix $M \times n \times T$) or `NaN` if smoother disabled

The output variables are required by function `pf_likelihood` that calculates the normalized expected log likelihood.

Asynchronous filter

Function `async_particle_filter(events, n, M, transition_model, measurement_model, init_particles, params)` implements the asynchronous version of Particle filter. It accepts `events` matrix that contains various types of events as its rows. The first item of an event is a number describing its type, `1` stands for a control signal. In this case, transition step is performed on the current set of particles. Otherwise, measurement step (including resampling) is performed. Measurement model should also distinguish between various kinds of measurements. In our ground robot example, we use `measurement_model_ground` measurement function, which either assigns the weights for GPS or compass measurements (does not require all measurements at each time).

Examples

Script `run_task_I_PF` demonstrates usage of the Particle filter and plots sample results. It uses functions that define the model in a simple and readable way, e.g.

```
X_new(:, xdot_i) = c0 * X(:, xdot_i) + c1 * X(:, pitch_i);
```

On the other hand, model defining functions we use in experiments are more general and expect parameters matrices like Kalman filter in `params` cell array.

Measurement models

To estimate the probability of a measurement z given a particle x, our models use a helper function `gaussian_probability(z, x, sigma, prec)` that works for a single state variable and multiple particles. `z` is the measured value, `x` contains the values of particles (used as the mean of the probability distribution). The function calculates

$$p(z|x) = normcdf(z+prec/2, x, \texttt{sigma}) - normcdf(z-prec/2, x, \texttt{sigma}), \quad (5.1)$$

where $normcdf(z, x, \texttt{sigma})$ is the probability of values less than z for a normal distribution with mean x and variance $\texttt{sigma}^2$. We use various precisions (`prec`) for various sensors.

5.4 Evaluation

The main function for evaluating methods is `evaluate` with the following parameters:

- `data` - data to be used. The method parses the data and splits into train and test sets as described below.
- `train_size` - if using fixed train size method, used as the train size. Forward chaining uses this as the size of one fold.
- `test_size` - test set size.
- `k` - number of iterations. Each iteration uses different train and test set.
- `learners` - a cell array with function handles to learning methods to be compared. The function interface is described below.
- `chaining` - a boolean parameter enabling forward chaining.
- `test_offset` - distance of the start of test set from the end of train set.

Let us denote the size of `learners` by L. The output variables of `evaluate` are as follows:

- `Q_train` - normalized expected likelihood on the train set for all learners and all iterations (matrix $L \times k$)
- `Q_test` - normalized expected likelihood on the test set for all learners and all iterations (matrix $L \times k$)
- `pitch_pred_test` - values of predictions of *pitch* for all learners and all iterations (matrix `test_size` $\times L \times k$)
- `xdot_pred_test` - values of predictions of *xdot* for all learners and all iterations (matrix `test_size` $\times L \times k$)
- `pitch_err` - MSE of predictions of *pitch* for all learners and all iterations (matrix $L \times k$)

- `xdot_err` - MSE of predictions of *xdot* for all learners and all iterations (matrix $L \times k$)
- `taskI_test` - filtered values (with known measurements) of *pitch* and *xdot* on test set for all learners and all iterations (matrix $2 \times$ `test_size` $\times L \times k$)
- `test_data` - data used for tests (matrix `test_size` $\times 12 \times k$, 12 is the number of columns in a single data record).

Each `learner` is expected to implement the following interface. Its only input is the training data. The outputs are

- `params` - a cell array of learnt params that will be passed to filtering algorithm and likelihood evaluator.
- `filter` - function handle for a function that implements appropriate filtering algorithm for the learner. A `filter` receives `params` and test data and outputs predictions for and filtered results *pitch* and *xdot*. Most of the learners share the common function `filter_kalman_B` or `filter_kalman_C` (depending on the model structure), but there is also the exponential filter implemented in function `filter_exponential` and Particle filter in function `filter_pf_all`.
- `evaluator` - function handle for a function that calculates the normalized expected likelihood. Common `evaluator` is `kalman_evaluator_B`, for instance. Particle filter also enables calculating the likelihood, but we usually comment this out and return `NaN` because the computation is very slow. Exponential filter does not have a method for calculating likelihood.

Examples of using the `evaluate` method and implementing a `learner` can be found in experiment scripts listed in Section 5.5.

5.5 List of scripts

1. `run_task_I_kalman`
2. `run_task_I_PF`
3. `run_task_ground_robots`
4. `run_experiment_distance`
5. `run_experiment_longer_validation`
6. `run_experiment_scaling`
7. `run_experiment_A`
8. `run_experiment_B`
9. `run_experiment_C`
10. `run_experiment_D`
11. `run_experiment_final_evaluation`

6. Conclusion

6.1 Summary

Our thesis summarizes the most important theoretical concepts of both filtering and system identification problems in a single coherent text with a sufficient level of detail. We implement Kalman filter, Particle filter and EM algorithm. Using real data example of UAV, we provide general guidelines for designing a probabilistic model and tuning its parameters based on given data.

UAV application

We solve the UAV problem successfully using Kalman filter and a model learnt by EM algorithm. We use the same model for both filtering task and prediction of missing sensor values. For the basic filtering task with all measurements available, it is sufficient to learn only those parameters from transition matrix that describe the underlying physical system. These parameters can be learnt also by linear regression.

To predict missing values of a sensor, more complex models are required. Here, we learn full transition and covariance matrices by EM algorithm. We show that EM algorithm is useful in cases where the true values of hidden states are unknown. We modify this algorithm by adding penalization terms for the model complexity. This avoids overfitting and also deals with non-Gaussian variances, which would otherwise be learnt wrongly. Our experiments show that this model performs well on unseen data.

Filtering noised values of speed can be performed using exponential filter. However, this filter leads to worse results than our Kalman filter, mainly because it slowly accommodates to new values. Moreover, Kalman filter describes the system more properly from the theoretical point of view.

We also applied Particle filter for this problem using a linear model learnt by Kalman EM algorithm. The performance was slightly worse than Kalman filter. Particle filter does not improve the model unless we know certain nonlinearities or non-Gaussian densities of the model.

Ground robot dataset

The ground robot dataset in Section 4.2 shows the importance of Particle filter for nonlinear problems. Here, we also propose a solution for dealing with misleading values of GPS, analyze the compass sensor and implement an asynchronous Particle filter.

6.2 Future work

- All the available data followed a simple pattern of balancing above a given point. The most helpful for improving the models would be to collect more data in different conditions, including outdoor and more complex flights.

- About 0.2% of our data contained missing values. As the portion was small, we simply replaced a missing value by the previous value. A more proper way of handling this should be used. For instance, the correction step in the filtering algorithm should be skipped and these values should not be reflected in the likelihood calculation.

- We learnt a single model that handles the configuration with all measurements available as well as missing measurement. The model could be extended by variables indicating the availability of sensors so that various parameters would be used for different configurations. An example of such work is [29].

- EM algorithm could be also further examined. For instance, its depends on the initialization and converges to local optima. [30] proposes a genetic-based modification of EM algorithm.

- There are several ways to improve the standard SIR Particle filter, briefly mentioned in Section 2.3.

Bibliography

[1] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. Probabilistic Robotics (Intelligent Robotics and Autonomous Agents series). Intelligent robotics and autonomous agents. The MIT Press, August 2005.

[2] Greg Welch, Gary Bishop. An Introduction to the Kalman Filter. University of North Carolina at Chapel Hill, Chapel Hill, NC, 1995.

[3] `http://automation.berkeley.edu/resources/EMAlgorithm.ppt` as of November 2013

[4] M. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking. IEEE Trans. on Signal Processing, 50(2):174–189, February 2002.

[5] Thomas B. Schön, Adrian Wills, and Brett Ninness. System identification of nonlinear state-space models. Automatica 47, 1 (January 2011), 39-49.

[6] Amin Zia, Thiagalingam Kirubarajan, James P. Reilly, Derek Yee, Kumaradevan Punithakumar, Shahram Shirani. An EM Algorithm for Nonlinear State Estimation With Model Uncertainties. IEEE Transactions on Signal Processing 56(3): 921-936 (2008).

[7] Tomáš Báča. Control of relatively localized unmanned helicopters. Bachelor thesis, Czech Technical University in Prague, 2013.

[8] Pavel Zedník. Relative visual localization in swarms of unmanned aerial vehicles. Bachelor thesis, Czech Technical University in Prague, 2012.

[9] `http://en.wikipedia.org/wiki/Mercator_projection` as of February 2014

[10] `http://www.mathworks.com/matlabcentral/fileexchange/30994-mercator-map-projection-and-inverse` as of February 2014

[11] `http://en.wikipedia.org/wiki/Exponential_smoothing` as of September 2013

[12] `http://en.wikipedia.org/wiki/Spline_interpolation` as of March 2014

[13] Y. C. Ho and R. C. K. Lee. A Bayesian approach to problems in stochastic estimation and control. IEEE Trans. Automat. Contr., vol. AC-9, pp. 333–339, 1964.

[14] G.A. Terejanu. Unscented Kalman filter tutorial. Workshop on Large-Scale Quantification of Uncertainty, Sandia National Laboratories, 2009, pp. 1–6.

[15] G.A. Terejanu. Extended Kalman Filter Tutorial. Technical Report; Department of Computer Science and Engineering, University of Buffalo: Buffalo, NY, USA, 2003.

[16] N. Kavitha, S. Vijayachitra. Extended Kalman Filter based State Estimation of Wind Turbine. International Journal of Engineering Trends and Technology (IJETT), Volume 5 Number 4, November 2013.

[17] J. R. Movellan. Discrete Time Kalman Filters and Smoothers. MPLab Tutorials, Univ. California at San Diego, 2011.

[18] G. Kitagawa. Monte Carlo filter and smoother for non-Gaussian nonlinear state space models. J. Comput. Graph. Statist., vol. 5, no. 1, pp. 1–25, 1996.

[19] J. S. Liu and R. Chen. Sequential Monte Carlo methods for dynamical systems. J. Amer. Statist. Assoc., vol. 93, pp.1032 -1044, 1998.

[20] J. Carpenter, P. Clifford, and P. Fearnhead. Improved particle filter for nonlinear problems. Proc. Inst. Elect. Eng., Radar, Sonar, Navig., 1999.

[21] W. R. Gilks and C. Berzuini. Following a moving target—Monte Carlo inference for dynamic Bayesian models. J. R. Statist. Soc. B, vol. 63, pp. 127–146, 2001.

[22] C. Musso, N. Oudjane, and F. LeGland. Improving regularised particle filters. Sequential Monte Carlo Methods in Practice, A. Doucet, J. F. G. de Freitas, and N. J. Gordon, Eds. New York: Springer-Verlag, 2001.

[23] M. Pitt and N. Shephard. Filtering via simulation: Auxiliary particle filters. J. Amer. Statist. Assoc., vol. 94, no. 446, pp. 590–599, 1999.

[24] R. Havangi, M. Nekoui, M. Teshnehlab. A multi swarm particle filter for mobile robot localization. International Journal of Computer Science (7) (2010), pp. 15–22.

[25] R.C. Eberhart, J. Kennedy. A new optimizer using particle swarm theory. Proceedings of the Sixth International Symposium on Micromachine and Human Science, Nagoya, Japan, pp. 39–43, 1995.

[26] `http://www.mathworks.com/help/stats/robustfit.html` as of March 2014

[27] `http://ardrone2.parrot.com/` as of March 2014

[28] `http://stats.stackexchange.com/questions/14099/using-k-fold-cross-validation-for-time-series-model-selection` as of March 2014

[29] F. Caron, M. Davy, E. Duflos and P. Vanheeghe. Particle Filtering for Multisensor Data Fusion With. Switching Observation Models: Application to Land. Vehicle Positioning. IEEE Transtions on signal processing, vol. 55, no. 6, June 2007.

[30] Franz Pernkopf, Djamel Bouchaffra. Genetic-Based EM Algorithm for Learning Gaussian Mixture Models. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 27, no. 8, pp. 1344-1348, August, 2005.

A. Content of the CD

The attached CD contains the following items:

- *MATLAB* source codes
- original data with legend
- PDF file with the thesis.