Aalto University

Dept. of Biomedical Engineering and Computational Science

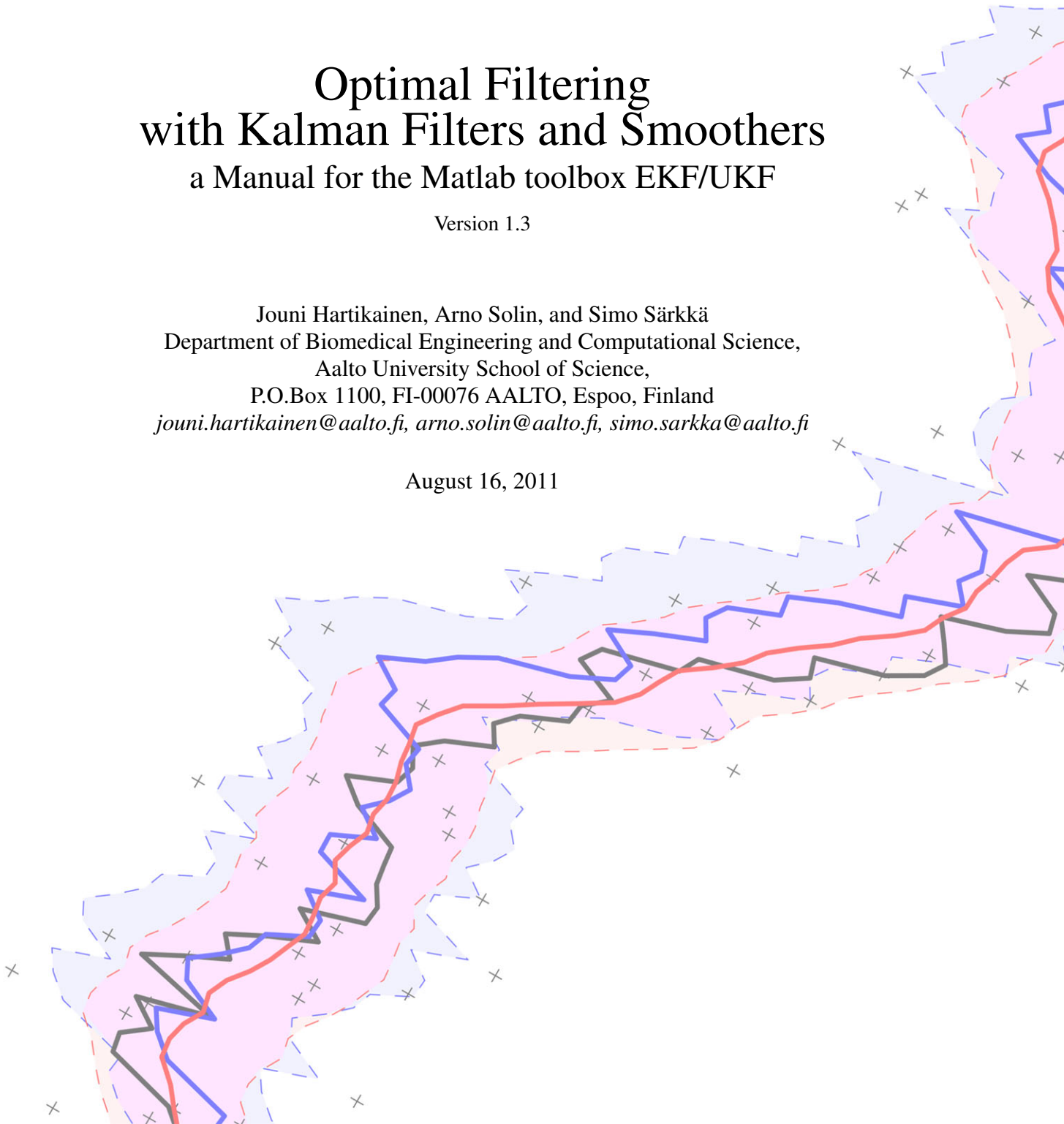
Optimal Filtering with Kalman Filters and Smoothers

a Manual for the Matlab toolbox EKF/UKF

Version 1.3

Jouni Hartikainen, Arno Solin, and Simo Särkkä
Department of Biomedical Engineering and Computational Science,
Aalto University School of Science,
P.O.Box 1100, FI-00076 AALTO, Espoo, Finland
jouni.hartikainen@aalto.fi, arno.solin@aalto.fi, simo.sarkka@aalto.fi

August 16, 2011



Abstract

In this paper we present a documentation for an optimal filtering toolbox for the mathematical software package Matlab. The toolbox features many filtering methods for discrete-time state space models, including the well-known linear Kalman filter and several non-linear extensions to it. These non-linear methods are the extended Kalman filter, the unscented Kalman filter, the Gauss-Hermite Kalman filter and the third-order symmetric cubature Kalman filter. Algorithms for multiple model systems are provided in the form of an Interacting Multiple Model (IMM) filter and it's non-linear extensions, which are based on banks of extended and unscented Kalman filters. Also included in the toolbox are the Rauch-Tung-Striebel and two-filter smoother counter-parts for the filters, which can be used to smooth the previous state estimates, after obtaining new measurements. The usage and function of each method is illustrated with eight demonstration problems.

Contents

Preface

Most of the software provided with this toolbox were originally created by Simo Särkkä while he was doing research on his doctoral thesis (**?**) in the Laboratory of Computational Engineering (LCE) at Helsinki University of Technology (HUT). This document has been written by Jouni Hartikainen at LCE during spring 2007 with a little help from Simo Särkkä. Jouni also checked and commented the software code thoroughly. Many (small) bugs were fixed, and also some new functions were implemented (for example 2nd order EKF and augmented form UKF). Jouni also provided the software code for first three demonstrations, modified the two last ones a bit, and ran all the simulations. In 2010, Arno Solin added the cubature integration based methods to the toolbox.

First author would like to thank Doc. Aki Vehtari for helpful comments during the work and for coming up with the idea of this toolbox in the first place. Prof. Jouko Lampinen also deserves thanks for ultimately making this work possible.

Chapter 1

Introduction

The term optimal filtering refers to methodology used for estimating the *state* of a time varying system, from which we observe indirect noisy measurements. The state refers to the physical state, which can be described by dynamic variables, such as position, velocity and acceleration of a moving object. The noise in the measurements means that there is a certain degree of uncertainty in them. The dynamic system evolves as a function of time, and there is also noise in the dynamics of system, *process noise*, meaning that the dynamic system cannot be modelled entirely deterministically. In this context, the term filtering basically means the process of filtering out the noise in the measurements and providing an optimal estimate for the state given the observed measurements and the assumptions made about the dynamic system.

This toolbox provides basic tools for estimating the state of a linear dynamic system, the Kalman filter, and also two extensions for it, the extended Kalman filter (EKF) and unscented Kalman filter (UKF), both of which can be used for estimating the states of nonlinear dynamic systems. Also the smoother counterparts of the filters are provided. Smoothing in this context means giving an estimate of the state of the system on some time step given all the measurements including ones encountered after that particular time step, in other words, the smoother gives a smoothed estimate for the history of the system's evolved state given all the measurements obtained so far.

This documentation is organized as follows:

- First we briefly introduce the concept of discrete-time state space models. After that we consider linear, discrete-time state space models in more detail and review Kalman filter, which is the basic method for recursively solving the linear state space estimation problems. Also Kalman smoother is introduced. After that the function of Kalman filter and smoother and their usage in this toolbox in demonstrated with one example (CWPA-model).

- Next we move from linear to nonlinear state space models and review the extended Kalman filter (and smoother), which is the classical extension to

Kalman filter for nonlinear estimation. The usage of EKF in this toolbox is illustrated exclusively with one example (Tracking a random sine signal), which also compares the performances of EKF, UKF and their smoother counter-parts.

- After EKF we review unscented Kalman filter (and smoother), which is a newer extension to traditional Kalman filter to cover nonlinear filtering problems. The usage of UKF is illustrated with one example (UNGM-model), which also demonstrates the differences between different nonlinear filtering techniques.
- We extend the concept of sigma-point filtring by studying other non-linear variants of Kalman filters. The Gauss-Hermite Kalman filter (GHKF) and third-order symmetric Cubature Kalman filter (CKF) are presented at this stage.
- To give a more thorough demonstration to the provided methods two more classical nonlinear filtering examples are provided (Bearings Only Tracking and Reentry Vehicle Tracking).
- In chapter **??** we shortly review the concept multiple model systems in general, and in sections **??**.1 and **??**.2 we take a look at linear and non-linear multiple model systems in more detail. We also review the standard method, the Interacting Multiple Model (IMM) filter, for estimating such systems. It's usage and function is demonstrated with three examples.

Details of the toolbox functions can be found on the toolbox web page, or in Matlab by typing `help <function name>`. The mathematical notation used in this document follows the notation used in (**?**).

Chapter 2

Discrete-Time State Space Models: Linear Models

2.1 Discrete-Time State Space Models

In this section we shall consider models where the states are defined in discrete-time models. The models are defined recursively in terms of distributions

$$\begin{aligned} \mathbf{x}_k &\sim p(\mathbf{x}_k \,|\, \mathbf{x}_{k-1}) \\ \mathbf{y}_k &\sim p(\mathbf{y}_k \,|\, \mathbf{x}_k), \end{aligned} \tag{2.1}$$

where

- $\mathbf{x}_k \in \mathbb{R}^n$ is the *state* of the system on the time step k.
- $\mathbf{y}_k \in \mathbb{R}^m$ is the measurement on the time step k.
- $p(\mathbf{x}_k \,|\, \mathbf{x}_{k-1})$ is the dynamic model which charaterizes the dynamic behaviour of the system. Usually the model is a probability density (continous state), but it can also be a counting measure (discrete state), or a combination of them, if the state is both continuous and discrete.
- $p(\mathbf{y}_k \,|\, \mathbf{x}_k)$ is the model for measurements, which describes how the measurements are distributed given the state. This model characterizes how the dynamic model is perceived by the observers.

A system defined this way has the so called *Markov*-property, which means that the state $\mathbf{x}_k$ given $\mathbf{x}_{k-1}$ is independent from the history of states and measurements, which can also be expressed with the following equality:

$$p(\mathbf{x}_k \,|\, \mathbf{x}_{1:k-1}, \mathbf{y}_{1:k-1}) = p(\mathbf{x}_k \,|\, \mathbf{x}_{k-1}). \tag{2.2}$$

The past doesn't depend on the future given the present, which is the same as

$$p(\mathbf{x}_{k-1} \,|\, \mathbf{x}_{k:T}, \mathbf{y}_{k:T}) = p(\mathbf{x}_{k-1} \,|\, \mathbf{x}_k). \tag{2.3}$$

The same applies also to measurements meaning that the measurement $\mathbf{y}_k$ is independent from the histories of measurements and states, which can be expressed with equality

$$p(\mathbf{y}_k \,|\, \mathbf{x}_{1:k}, \mathbf{y}_{1:k-1}) = p(\mathbf{y}_k \,|\, \mathbf{x}_k). \tag{2.4}$$

In actual application problems, we are interested in predicting and estimating dynamic system's state given the measurements obtained so far. In probabilistic terms, we are interested in the predictive distribution for the state at the next time step

$$p(\mathbf{x}_k \,|\, \mathbf{y}_{1:k-1}), \tag{2.5}$$

and in the marginal posterior distribution for the state at the current time step

$$p(\mathbf{x}_k \,|\, \mathbf{y}_{1:k}). \tag{2.6}$$

The formal solutions for these distribution are given by the following recursive Bayesian filtering equations (e.g. **?**):

$$p(\mathbf{x}_k \,|\, \mathbf{y}_{1:k-1}) = \int p(\mathbf{x}_k \,|\, \mathbf{x}_{k-1})\, p(\mathbf{x}_{k-1} \,|\, \mathbf{y}_{1:k-1})\, \mathrm{d}\mathbf{x}_{k-1}. \tag{2.7}$$

and

$$p(\mathbf{x}_k \,|\, \mathbf{y}_{1:k}) = \frac{1}{Z_k} p(\mathbf{y}_k \,|\, \mathbf{x}_k)\, p(\mathbf{x}_k \,|\, \mathbf{y}_{1:k-1}), \tag{2.8}$$

where the normalization constant Z_k is given as

$$Z_k = \int p(\mathbf{y}_k \,|\, \mathbf{x}_k)\, p(\mathbf{x}_k \,|\, \mathbf{y}_{1:k-1})\, \mathrm{d}\mathbf{x}_k. \tag{2.9}$$

In many cases we are also interested in smoothed state estimates of previous time steps given the measurements obtained so far. In other words, we are interested in the marginal posterior distribution

$$p(\mathbf{x}_k \,|\, \mathbf{y}_{1:T}), \tag{2.10}$$

where $T > k$. As with the filtering equations above also in this case we can express the formal solution as a set of recursive Bayesian equations (e.g. **?**):

$$\begin{aligned} p(\mathbf{x}_{k+1} \,|\, \mathbf{y}_{1:k}) &= \int p(\mathbf{x}_{k+1} \,|\, \mathbf{x}_k)\, p(\mathbf{x}_k \,|\, \mathbf{y}_{1:k})\, \mathrm{d}\mathbf{x}_k \\ p(\mathbf{x}_k \,|\, \mathbf{y}_{1:T}) &= p(\mathbf{x}_k \,|\, \mathbf{y}_{1:k}) \int \left[\frac{p(\mathbf{x}_{k+1} \,|\, \mathbf{x}_k)\, p(\mathbf{x}_{k+1} \,|\, \mathbf{y}_{1:T})}{p(\mathbf{x}_{k+1} \,|\, \mathbf{y}_{1:k})} \right] \mathrm{d}\mathbf{x}_{k+1}. \end{aligned} \tag{2.11}$$

2.2 Linear State Space Estimation

The simplest of the state space models considered in this documentation are linear models, which can be expressed with equations of the following form:

$$\begin{aligned} \mathbf{x}_k &= \mathbf{A}_{k-1}\,\mathbf{x}_{k-1} + \mathbf{q}_{k-1} \\ \mathbf{y}_k &= \mathbf{H}_k\,\mathbf{x}_k + \mathbf{r}_k, \end{aligned} \tag{2.12}$$

where

- $\mathbf{x}_k \in \mathbb{R}^n$ is the *state* of the system on the time step k.
- $\mathbf{y}_k \in \mathbb{R}^m$ is the measurement on the time step k.
- $\mathbf{q}_{k-1} \sim \mathrm{N}(\mathbf{0}, \mathbf{Q}_{k-1})$ is the process noise on the time step $k-1$.
- $\mathbf{r}_k \sim \mathrm{N}(\mathbf{0}, \mathbf{R}_k)$ is the measurement noise on the time step k.
- $\mathbf{A}_{k-1}$ is the transition matrix of the dynamic model.
- $\mathbf{H}_k$ is the measurement model matrix.
- The prior distribution for the state is $\mathbf{x}_0 \sim \mathrm{N}(\mathbf{m}_0, \mathbf{P}_0)$, where parameters $\mathbf{m}_0$ and $\mathbf{P}_0$ are set using the information known about the system under the study.

The model can also be equivalently expressed in probabilistic terms with distributions

$$\begin{aligned} p(\mathbf{x}_k \,|\, \mathbf{x}_{k-1}) &= \mathrm{N}(\mathbf{x}_k \,|\, \mathbf{A}_{k-1}\,\mathbf{x}_{k-1}, \mathbf{Q}_{k-1}) \\ p(\mathbf{y}_k \,|\, \mathbf{x}_k) &= \mathrm{N}(\mathbf{y}_k \,|\, \mathbf{H}_k\,\mathbf{x}_k, \mathbf{R}_k). \end{aligned} \tag{2.13}$$

2.2.1 Discretization of Continuous-Time Linear Time-Invariant Systems

Often many linear time-invariant models are described with continous-time state equations of the following form:

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{F}\mathbf{x}(t) + \mathbf{L}\mathbf{w}(t), \tag{2.14}$$

where

- the initial conditions are $\mathbf{x}(0) \sim \mathrm{N}(\mathbf{m}(0), \mathbf{P}(0))$,
- $\mathbf{F}$ and $\mathbf{L}$ are constant matrices, which characterize the behaviour of the model, matrix $\mathbf{Q}_c$.
- $\mathbf{w}(t)$ is a white noise process with a power spectral density $\mathbf{Q}_c$.

To be able to use the Kalman filter defined in the next section the model (14) must be discretized somehow, so that it can be described with a model of the form (12). The solution for the discretized matrices $\mathbf{A}_k$ and $\mathbf{Q}_k$ can be given as (see, e.g., **??**).

$$\mathbf{A}_k = \exp(\mathbf{F}\,\Delta t_k) \tag{2.15}$$

$$\mathbf{Q}_k = \int_0^{\Delta t_k} \exp(\mathbf{F}\,(\Delta t_k - \tau))\,\mathbf{L}\,\mathbf{Q}_c\,\mathbf{L}^T\,\exp(\mathbf{F}\,(\Delta t_k - \tau))^T \mathrm{d}\tau, \tag{2.16}$$

where $\Delta t_k = t_{k+1} - t_k$ is the stepsize of the discretization. In some cases the $\mathbf{Q}_k$ can be calculated analytically, but in cases where it isn't possible, the matrix can still be calculated efficiently using the following matrix fraction decomposition:

$$\begin{pmatrix} \mathbf{C}_k \\ \mathbf{D}_k \end{pmatrix} = \exp\left\{ \begin{pmatrix} \mathbf{F} & \mathbf{L}\,\mathbf{Q}_c\,\mathbf{L}^T \\ \mathbf{0} & -\mathbf{F}^T \end{pmatrix} \Delta t_k \right\} \begin{pmatrix} \mathbf{0} \\ \mathbf{I} \end{pmatrix}. \tag{2.17}$$

The matrix $\mathbf{Q}_k$ is then given as $\mathbf{Q}_k = \mathbf{C}_k \mathbf{D}_k^{-1}$.

In this toolbox the discretization can be done with the function `lti_disc`, which uses the matrix fractional decomposition.

2.2.2 Kalman Filter

The classical Kalman filter was first introduced by Rudolph E. Kalman in his seminal paper (**?**). The purpose of the discrete-time Kalman filter is to provide the closed form recursive solution for estimation of linear discrete-time dynamic systems, which can be described by equations of the form (12).

Kalman filter has two steps: the prediction step, where the next state of the system is predicted given the previous measurements, and the update step, where the current state of the system is estimated given the measurement at that time step. The steps translate to equations as follows (see, e.g., **??**, for derivation):

- *Prediction:*

$$\begin{aligned} \mathbf{m}_k^- &= \mathbf{A}_{k-1}\,\mathbf{m}_{k-1} \\ \mathbf{P}_k^- &= \mathbf{A}_{k-1}\,\mathbf{P}_{k-1}\,\mathbf{A}_{k-1}^T + \mathbf{Q}_{k-1}. \end{aligned} \tag{2.18}$$

- *Update:*

$$\begin{aligned} \mathbf{v}_k &= \mathbf{y}_k - \mathbf{H}_k\,\mathbf{m}_k^- \\ \mathbf{S}_k &= \mathbf{H}_k\,\mathbf{P}_k^-\,\mathbf{H}_k^T + \mathbf{R}_k \\ \mathbf{K}_k &= \mathbf{P}_k^-\,\mathbf{H}_k^T\,\mathbf{S}_k^{-1} \\ \mathbf{m}_k &= \mathbf{m}_k^- + \mathbf{K}_k\,\mathbf{v}_k \\ \mathbf{P}_k &= \mathbf{P}_k^- - \mathbf{K}_k\,\mathbf{S}_k\,\mathbf{K}_k^T, \end{aligned} \tag{2.19}$$

where

- $\mathbf{m}_k^-$ and $\mathbf{P}_k^-$ are the predicted mean and covariance of the state, respectively, on the time step k before seeing the measurement.
- $\mathbf{m}_k$ and $\mathbf{P}_k$ are the estimated mean and covariance of the state, respectively, on time step k after seeing the measurement.
- $\mathbf{v}_k$ is the innovation or the measurement residual on time step k.
- $\mathbf{S}_k$ is the measurement prediction covariance on the time step k.
- $\mathbf{K}_k$ is the filter gain, which tells how much the predictions should be corrected on time step k.

Note that in this case the predicted and estimated state covariances on different time steps do not depend on any measurements, so that they could be calculated off-line before making any measurements provided that the matrices $\mathbf{A}$, $\mathbf{Q}$, $\mathbf{R}$ and $\mathbf{H}$ are known on those particular time steps. Usage for this property, however, is not currently provided explicitly with this toolbox.

It is also possible to predict the state of system as many steps ahead as wanted just by looping the predict step of Kalman filter, but naturally the accuracy of the estimate decreases with every step.

The prediction and update steps can be calculated with functions `kf_predict` and `kf_update`.

2.2.3 Kalman Smoother

The discrete-time Kalman smoother, also known as the Rauch-Tung-Striebel-smoother (RTS), (**???**) can be used for computing the smoothing solution for the model (12) given as distribution

$$p(\mathbf{x}_k \mid \mathbf{y}_{1:T}) = \mathrm{N}(\mathbf{x}_k \mid \mathbf{m}_k^s, \mathbf{P}_k^s). \tag{2.20}$$

The mean and covariance $\mathbf{m}_k^s$ and $\mathbf{P}_k^s$ are calculated with the following equations:

$$\begin{aligned}
\mathbf{m}_{k+1}^- &= \mathbf{A}_k\, \mathbf{m}_k \\
\mathbf{P}_{k+1}^- &= \mathbf{A}_k\, \mathbf{P}_k\, \mathbf{A}_k^T + \mathbf{Q}_k \\
\mathbf{C}_k &= \mathbf{P}_k\, \mathbf{A}_k^T\, [\mathbf{P}_{k+1}^-]^{-1} \\
\mathbf{m}_k^s &= \mathbf{m}_k + \mathbf{C}_k\, [\mathbf{m}_{k+1}^s - \mathbf{m}_{k+1}^-] \\
\mathbf{P}_k^s &= \mathbf{P}_k + \mathbf{C}_k\, [\mathbf{P}_{k+1}^s - \mathbf{P}_{k+1}^-]\, \mathbf{C}_k^T,
\end{aligned} \tag{2.21}$$

where

- $\mathbf{m}_k^s$ and $\mathbf{P}_k^s$ are the smoother estimates for the state mean and state covariance on time step k.

- $\mathbf{m}_k$ and $\mathbf{P}_k$ are the filter estimates for the state mean and state covariance on time step k.
- $\mathbf{m}_{k+1}^-$ and $\mathbf{P}_{k+1}^-$ are the predicted state mean and state covariance on time step $k+1$, which are the same as in the Kalman filter.
- $\mathbf{C}_k$ is the smoother gain on time step k, which tells how much the smooted estimates should be corrected on that particular time step.

The difference between Kalman filter and Kalman smoother is that the recursion in filter moves forward and in smoother backward, as can be seen from the equations above. In smoother the recursion starts from the last time step T with $\mathbf{m}_T^s = \mathbf{m}_T$ and $\mathbf{P}_T^s = \mathbf{P}_T$.

The smoothed estimate for states and covariances using the RTS smoother can be calculated with the function `rts_smooth`.

In addition to RTS smoother it is possible to formulate the smoothing operation as a combination of two optimum filters (**?**), of which the first filter sweeps the data forward going from the first measurement towards the last one, and the second sweeps backwards towards the opposite direction.

It can be shown, that combining the estimates produced by these two filters in a suitable way produces an smoothed estimate for the state, which has lower mean square error than any of these two filters alone (**?**). With linear models the forward-backward smoother gives the same error as the RTS-smoother, but in non-linear cases the error behaves differently in some situations. In this toolbox forward-backward smoothing solution can be calculated with function `tf_smooth`.

2.2.4 Demonstration: 2D CWPA-model

Let's now consider a very simple case, where we track an object moving in two dimensional space with a sensor, which gives measurements of target's position in cartesian coordinates x and y. In addition to position target also has state variables for its velocities and accelerations toward both coordinate axes, $\dot{x}$, $\dot{y}$, $\ddot{x}$ and $\ddot{y}$. In other words, the state of a moving object on time step k can be expressed as a vector

$$\mathbf{x}_k = \begin{pmatrix} x_k & y_k & \dot{x}_k & \dot{y}_k & \ddot{x}_k & \ddot{y}_k \end{pmatrix}^T. \tag{2.22}$$

In continous case the dynamics of the target's motion can be modelled as a linear, time-invariant system

$$\frac{d\mathbf{x}(t)}{dt} = \underbrace{\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}}_{\mathbf{F}} \mathbf{x}(t) + \underbrace{\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}}_{\mathbf{L}} \mathbf{w}(t), \tag{2.23}$$

where $\mathbf{x}(t)$ is the target's state on the time t and $\mathbf{w}(t)$ is a white noise process with power spectral density

$$\mathbf{Q}_c = \begin{pmatrix} q & 0 \\ 0 & q \end{pmatrix} = \begin{pmatrix} 0.2 & 0 \\ 0 & 0.2 \end{pmatrix}. \tag{2.24}$$

As can be seen from the equation the acceleration of the object is perturbed with a white noise process and hence this model has the name continous Wiener process acceleration (CWPA) model. There is also other models similar to this, as for example the continous white noise acceleration (CWNA) model (**?**), where the velocity is perturbed with a white noise process.

The measurement matrix is set to

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}, \tag{2.25}$$

which means that the observe only the position of the moving object. To be able to estimate this system with a discrete-time Kalman filter the differential equation defined above must be discretized somehow to get a discrete-time state equation of the form (12). It turns out, that the matrices $\mathbf{A}$ and $\mathbf{Q}$ can be calculated analytically with equations (15) and (16) to give the following:

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & \Delta\mathrm{t} & 0 & \frac{1}{2}\Delta\mathrm{t}^2 & 0 \\ 0 & 1 & 0 & \Delta\mathrm{t} & 0 & \frac{1}{2}\Delta\mathrm{t}^2 \\ 0 & 0 & 1 & 0 & \Delta\mathrm{t} & 0 \\ 0 & 0 & 0 & 1 & 0 & \Delta\mathrm{t} \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.26}$$

$$\mathbf{Q} = \begin{pmatrix} \frac{1}{20}\Delta\mathrm{t}^5 & 0 & \frac{1}{8}\Delta\mathrm{t}^4 & 0 & \frac{1}{6}\Delta\mathrm{t}^3 & 0 \\ 0 & \frac{1}{20}\Delta\mathrm{t}^5 & 0 & \frac{1}{8}\Delta\mathrm{t}^4 & 0 & \frac{1}{6}\Delta\mathrm{t}^3 \\ \frac{1}{8}\Delta\mathrm{t}_\mathrm{k}\,4 & 0 & \frac{1}{6}\Delta\mathrm{t}^3 & 0 & \frac{1}{2}\Delta\mathrm{t}^2 & 0 \\ 0 & \frac{1}{8}\Delta\mathrm{t}^4 & 0 & \frac{1}{6}\Delta\mathrm{t}^3 & 0 & \frac{1}{2}\Delta\mathrm{t}^2 \\ \frac{1}{6}\Delta\mathrm{t}^3 & 0 & \frac{1}{2}\Delta\mathrm{t}^2 & 0 & \Delta\mathrm{t} & 0 \\ 0 & \frac{1}{6}\Delta\mathrm{t}^3 & 0 & \frac{1}{2}\Delta\mathrm{t}^2 & 0 & \Delta\mathrm{t} \end{pmatrix} q, \tag{2.27}$$

where the stepsize is set to $\Delta\mathrm{t} = 0.5$. These matrices can also calculated using the function `lti_disc` introduced in section 2.1 with the following code line:

```
[A,Q] = lti_disc(F,L,Qc,dt);
```

where matrices `F` and `L` are assumed to contain the matrices from equation (23).

The object starts from origo with zero velocity and acceleration and the process is simulated 50 steps. The variance for the measurements is set to

$$\mathbf{R} = \begin{pmatrix} 10 & 0 \\ 0 & 10 \end{pmatrix}, \tag{2.28}$$

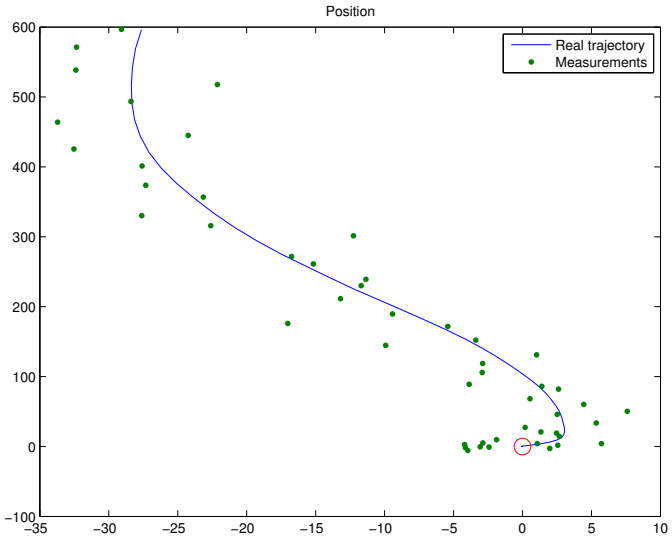


Figure 2.1: The real position of the moving object and the simulated measurements of it using the CWPA model. The circle marks the starting point of the object.

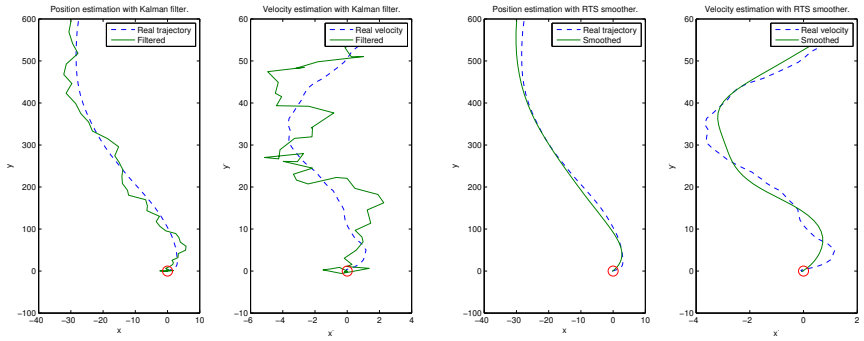
which is relatively high so that the the difference between the filtering and smoothing (described in next section) estimates becomes clearer. The real position of the object and measurements of it are plotted in the figure 1.

The filtering is done with the following code fragment:

```
MM = zeros(size(m,1), size(Y,2));
PP = zeros(size(m,1), size(m,1), size(Y,2));

for i = 1:size(Y,2)
   [m,P] = kf_predict(m,P,A,Q);
   [m,P] = kf_update(m,P,Y(:,i),H,R);
   MM(:,i) = m;
   PP(:,:,i) = P;
end
```

In the first 2 lines the space for state mean and covariance estimates is reserved, and the rest of the code contains the actual filtering loop, where we make the predict and update steps of the Kalman filter. The variables `m` and `P` are assumed to contain the initial guesses for the state mean and covariance before reaching the for-statement. Variable `Y` is assumed to contain the measurements made from the system (See the full source code of the example (`kf_cwpa_demo.m`) provided with the toolbox to see how we generated the measurements by simulating the dynamic system). In the end of each iteration the acquired estimates are stored to matrices `MM` and `PP`, for which we reserved space earlier. The estimates for object's position and velocity with Kalman filter and are plotted in figure 2.



(a) Filter (b) Smoother

Figure 2.2: (a) Estimates for position and velocity of the moving object using the Kalman filter and CWPA model, and (b) Estimate for position and velocity of the moving object using the RTS smoother and CWPA model.

The smoothed estimates for the state mean and covariance can be calculated with the following code line:

```
[SM,SP] = rts_smooth(MM,PP,A,Q);
```

The calculated smoothed estimates for object's position and velocity for the earlier demonstration are plotted in figure 3. As expected the smoother produces more accurate estimates than the filter as it uses all measurements for the estimation each time step. Note that the difference between the smoothed and filtered estimated would be smaller, if the measurements were more accurate, as now the filter performs rather badly due to the great uncertaintity in the measurements. The smoothing results of a forward-backward smoother are not plotted here, as the result are exactly the same as with the RTS smoother.

As one would expect the estimates for object's velocity are clearly less accurate than the estimates for the object's position as the positions are observed directly and the velocities only indirectly. If the velocities were also observed not only the velocity estimates would get more accurate, but also the position ones as well.

Chapter 3

Nonlinear State Space Estimation

In many cases interesting dynamic systems are not linear by nature, so the traditional Kalman filter cannot be applied in estimating the state of such a system. In these kind of systems, both the dynamics and the measurement processes can be nonlinear, or only one them. In this section, we describe two extensions to the traditional Kalman filter, which can be applied for estimating nonlinear dynamical systems by forming Gaussian approximations to the joint distribution of the state $\mathbf{x}$ and measurement $\mathbf{y}$. First we present the Extended Kalman filter (EKF), which is based on Taylor series approximation of the joint distribution, and then the Unscented Kalman filter (UKF), which is respectively based on the unscented transformation of the joint distribution.

3.1 Extended Kalman Filter

3.1.1 Taylor Series Based Approximations

Next we present linear and quadratic approximations for the distribution of variable $\mathbf{y}$, which is generated with a non-linear transformation of a Gaussian random variable $\mathbf{x}$ as follows:

$$\begin{aligned} \mathbf{x} &\sim \mathrm{N}(\mathbf{m}, \mathbf{P}) \\ \mathbf{y} &= \mathbf{g}(\mathbf{x}), \end{aligned} \tag{3.1}$$

where $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$, and $\mathbf{g} : \mathbb{R}^n \mapsto \mathbb{R}^m$ is a general non-linear function. Solving the distribution of $\mathbf{y}$ formally is in general not possible, because it is non-Gaussian for all by linear $\mathbf{g}$, so in practice it must be approximated somehow. The joint distribution of $\mathbf{x}$ and $\mathbf{y}$ can be formed with, for example, linear and quadratic approximations, which we present next. See, for example, **?** for the derivation of these approximations.

3.1.2 Linear Approximation

The linear approximation based Gaussian approximation of the joint distribution of variables $\mathbf{x}$ and $\mathbf{y}$ defined by equations (29) is given as

$$\begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix} \sim \mathrm{N}\left(\begin{pmatrix}\mathbf{m}\\ \boldsymbol{\mu}_L\end{pmatrix}, \begin{pmatrix}\mathbf{P} & \mathbf{C}_L\\ \mathbf{C}_L^T & \mathbf{S}_L\end{pmatrix}\right), \tag{3.2}$$

where

$$\begin{aligned}\boldsymbol{\mu}_L &= \mathbf{g}(\mathbf{m})\\ \mathbf{S}_L &= \mathbf{G}_\mathbf{x}(\mathbf{m})\,\mathbf{P}\,\mathbf{G}_\mathbf{x}^T(\mathbf{m})\\ \mathbf{C}_L &= \mathbf{P}\,\mathbf{G}_\mathbf{x}^T(\mathbf{m}),\end{aligned} \tag{3.3}$$

and $\mathbf{G}_\mathbf{x}(\mathbf{m})$ is the Jacobian matrix of $\mathbf{g}$ with elements

$$[\mathbf{G}_\mathbf{x}(\mathbf{m})]_{j,j'} = \left.\frac{\partial g_j(\mathbf{x})}{\partial x_{j'}}\right|_{\mathbf{x}=\mathbf{m}}. \tag{3.4}$$

3.1.3 Quadratic Approximation

The quadratic approximations retain also the second order terms of the Taylor series expansion of the non-linear function:

$$\begin{pmatrix}\mathbf{x}\\ \mathbf{y}\end{pmatrix} \sim \mathrm{N}\left(\begin{pmatrix}\mathbf{m}\\ \boldsymbol{\mu}_Q\end{pmatrix}, \begin{pmatrix}\mathbf{P} & \mathbf{C}_Q\\ \mathbf{C}_Q^T & \mathbf{S}_Q\end{pmatrix}\right), \tag{3.5}$$

where the parameters are

$$\begin{aligned}\boldsymbol{\mu}_Q &= \mathbf{g}(\mathbf{m}) + \frac{1}{2}\sum_i \mathbf{e}_i\ \mathrm{tr}\left\{\mathbf{G}_{\mathbf{xx}}^{(i)}(\mathbf{m})\,\mathbf{P}\right\}\\ \mathbf{S}_Q &= \mathbf{G}_\mathbf{x}(\mathbf{m})\,\mathbf{P}\,\mathbf{G}_\mathbf{x}^T(\mathbf{m}) + \frac{1}{2}\sum_{i,i'} \mathbf{e}_i\,\mathbf{e}_{i'}^T\ \mathrm{tr}\left\{\mathbf{G}_{\mathbf{xx}}^{(i)}(\mathbf{m})\,\mathbf{P}\,\mathbf{G}_{\mathbf{xx}}^{(i')}(\mathbf{m})\,\mathbf{P}\right\}\\ \mathbf{C}_Q &= \mathbf{P}\,\mathbf{G}_\mathbf{x}^T(\mathbf{m}),\end{aligned} \tag{3.6}$$

$\mathbf{G}_\mathbf{x}(\mathbf{m})$ is the Jacobian matrix (32) and $\mathbf{G}_{\mathbf{xx}}^{(i)}(\mathbf{m})$ is the Hessian matrix of $g_i(\cdot)$ evaluated at $\mathbf{m}$:

$$\left[\mathbf{G}_{\mathbf{xx}}^{(i)}(\mathbf{m})\right]_{j,j'} = \left.\frac{\partial^2 g_i(\mathbf{x})}{\partial x_j\,\partial x_{j'}},\right|_{\mathbf{x}=\mathbf{m}}. \tag{3.7}$$

$\mathbf{e}_i = (0\ \cdots\ 0\ 1\ 0\ \cdots\ 0)^T$ is the unit vector in direction of the coordinate axis i.

3.1.4 Extended Kalman filter

The extended Kalman filter (see, for instance, **?????**) extends the scope of Kalman filter to nonlinear optimal filtering problems by forming a Gaussian approximation to the joint distribution of state $\mathbf{x}$ and measurements $\mathbf{y}$ using a Taylor series based transformation. First and second order extended Kalman filters are presented, which are based on linear and quadratic approximations to the transformation. Higher order filters are also possible, but not presented here.

The filtering model used in the EKF is

$$\begin{aligned} \mathbf{x}_k &= \mathbf{f}(\mathbf{x}_{k-1}, k-1) + \mathbf{q}_{k-1} \\ \mathbf{y}_k &= \mathbf{h}(\mathbf{x}_k, k) + \mathbf{r}_k, \end{aligned} \tag{3.8}$$

where $\mathbf{x}_k \in \mathbb{R}^n$ is the state, $\mathbf{y}_k \in \mathbb{R}^m$ is the measurement, $\mathbf{q}_{k-1} \sim \mathrm{N}(\mathbf{0}, \mathbf{Q}_{k-1})$ is the process noise, $\mathbf{r}_k \sim \mathrm{N}(\mathbf{0}, \mathbf{R}_k)$ is the measurement noise, $\mathbf{f}$ is the (possibly nonlinear) dynamic model function and $\mathbf{h}$ is the (again possibly nonlinear) measurement model function. The first and second order extended Kalman filters approximate the distribution of state $\mathbf{x}_k$ given the observations $\mathbf{y}_{1:k}$ with a Gaussian:

$$p(\mathbf{x}_k \,|\, \mathbf{y}_{1:k}) \approx \mathrm{N}(\mathbf{x}_k \,|\, \mathbf{m}_k, \mathbf{P}_k). \tag{3.9}$$

First Order Extended Kalman Filter

Like Kalman filter, also the extended Kalman filter is separated to two steps. The steps for the first order EKF are as follows:

- *Prediction:*

$$\begin{aligned} \mathbf{m}_k^- &= \mathbf{f}(\mathbf{m}_{k-1}, k-1) \\ \mathbf{P}_k^- &= \mathbf{F}_{\mathbf{x}}(\mathbf{m}_{k-1}, k-1)\, \mathbf{P}_{k-1}\, \mathbf{F}_{\mathbf{x}}^T(\mathbf{m}_{k-1}, k-1) + \mathbf{Q}_{k-1}. \end{aligned} \tag{3.10}$$

- *Update:*

$$\begin{aligned} \mathbf{v}_k &= \mathbf{y}_k - \mathbf{h}(\mathbf{m}_k^-, k) \\ \mathbf{S}_k &= \mathbf{H}_{\mathbf{x}}(\mathbf{m}_k^-, k)\, \mathbf{P}_k^-\, \mathbf{H}_{\mathbf{x}}^T(\mathbf{m}_k^-, k) + \mathbf{R}_k \\ \mathbf{K}_k &= \mathbf{P}_k^-\, \mathbf{H}_{\mathbf{x}}^T(\mathbf{m}_k^-, k)\, \mathbf{S}_k^{-1} \\ \mathbf{m}_k &= \mathbf{m}_k^- + \mathbf{K}_k\, \mathbf{v}_k \\ \mathbf{P}_k &= \mathbf{P}_k^- - \mathbf{K}_k\, \mathbf{S}_k\, \mathbf{K}_k^T, \end{aligned} \tag{3.11}$$

where the matrices $\mathbf{F}_{\mathbf{x}}(\mathbf{m}, k-1)$ and $\mathbf{H}_{\mathbf{x}}(\mathbf{m}, k)$ are the Jacobians of $\mathbf{f}$ and $\mathbf{h}$, with elements

$$[\mathbf{F}_{\mathbf{x}}(\mathbf{m}, k-1)]_{j,j'} = \left.\frac{\partial f_j(\mathbf{x}, k-1)}{\partial x_{j'}}\right|_{\mathbf{x}=\mathbf{m}} \tag{3.12}$$

$$[\mathbf{H}_{\mathbf{x}}(\mathbf{m}, k)]_{j,j'} = \left.\frac{\partial h_j(\mathbf{x}, k)}{\partial x_{j'}}\right|_{\mathbf{x}=\mathbf{m}}. \tag{3.13}$$

Note that the difference between first order EKF and KF is that the matrices $\mathbf{A}_k$ and $\mathbf{H}_k$ in KF are replaced with Jacobian matrices $\mathbf{F}_{\mathbf{x}}(\mathbf{m}_{k-1}, k-1)$ and $\mathbf{H}_{\mathbf{x}}(\mathbf{m}_k^-, k)$ in EKF. Predicted mean $\mathbf{m}_k^-$ and residual of prediction $\mathbf{v}_k$ are also calculated differently in the EKF. In this toolbox the prediction and update steps of the first order EKF can be computed with functions `ekf_predict1` and `ekf_update1`, respectively.

Second Order Extended Kalman Filter

The corresponding steps for the second order EKF are as follows:

- *Prediction:*

$$\begin{aligned}
\mathbf{m}_k^- &= \mathbf{f}(\mathbf{m}_{k-1}, k-1) + \frac{1}{2}\sum_i \mathbf{e}_i \,\mathrm{tr}\left\{\mathbf{F}_{\mathbf{xx}}^{(i)}(\mathbf{m}_{k-1}, k-1)\,\mathbf{P}_{k-1}\right\} \\
\mathbf{P}_k^- &= \mathbf{F}_{\mathbf{x}}(\mathbf{m}_{k-1}, k-1)\,\mathbf{P}_{k-1}\,\mathbf{F}_{\mathbf{x}}^T(\mathbf{m}_{k-1}, k-1) \\
&\quad + \frac{1}{2}\sum_{i,i'} \mathbf{e}_i\,\mathbf{e}_{i'}^T \,\mathrm{tr}\left\{\mathbf{F}_{\mathbf{xx}}^{(i)}(\mathbf{m}_{k-1}, k-1)\mathbf{P}_{k-1}\mathbf{F}_{\mathbf{xx}}^{(i')}(\mathbf{m}_{k-1}, k-1)\mathbf{P}_{k-1}\right\} \\
&\quad + \mathbf{Q}_{k-1}.
\end{aligned} \tag{3.14}$$

- *Update:*

$$\begin{aligned}
\mathbf{v}_k &= \mathbf{y}_k - \mathbf{h}(\mathbf{m}_k^-, k) - \frac{1}{2}\sum_i \mathbf{e}_i \,\mathrm{tr}\left\{\mathbf{H}_{\mathbf{xx}}^{(i)}(\mathbf{m}_k^-, k)\,\mathbf{P}_k^-\right\} \\
\mathbf{S}_k &= \mathbf{H}_{\mathbf{x}}(\mathbf{m}_k^-, k)\,\mathbf{P}_k^-\,\mathbf{H}_{\mathbf{x}}^T(\mathbf{m}_k^-, k) \\
&\quad + \frac{1}{2}\sum_{i,i'} \mathbf{e}_i\,\mathbf{e}_{i'}^T \,\mathrm{tr}\left\{\mathbf{H}_{\mathbf{xx}}^{(i)}(\mathbf{m}_k^-, k)\,\mathbf{P}_k^-\,\mathbf{H}_{\mathbf{xx}}^{(i')}(\mathbf{m}_k^-, k)\,\mathbf{P}_k^-\right\} + \mathbf{R}_k \\
\mathbf{K}_k &= \mathbf{P}_k^-\,\mathbf{H}_{\mathbf{x}}^T(\mathbf{m}_k^-, k)\,\mathbf{S}_k^{-1} \\
\mathbf{m}_k &= \mathbf{m}_k^- + \mathbf{K}_k\,\mathbf{v}_k \\
\mathbf{P}_k &= \mathbf{P}_k^- - \mathbf{K}_k\,\mathbf{S}_k\,\mathbf{K}_k^T,
\end{aligned} \tag{3.15}$$

where matrices $\mathbf{F}_{\mathbf{x}}(\mathbf{m}, k-1)$ and $\mathbf{H}_{\mathbf{x}}(\mathbf{m}, k)$ are Jacobians as in the first order EKF, given by Equations (40) and (41). The matrices $\mathbf{F}_{\mathbf{xx}}^{(i)}(\mathbf{m}, k-1)$ and $\mathbf{H}_{\mathbf{xx}}^{(i)}(\mathbf{m}, k)$ are the Hessian matrices of f_i and h_i:

$$\left[\mathbf{F}_{\mathbf{xx}}^{(i)}(\mathbf{m}, k-1)\right]_{j,j'} = \left.\frac{\partial^2 f_i(\mathbf{x}, k-1)}{\partial x_j\,\partial x_{j'}}\right|_{\mathbf{x}=\mathbf{m}} \tag{3.16}$$

$$\left[\mathbf{H}_{\mathbf{xx}}^{(i)}(\mathbf{m}, k)\right]_{j,j'} = \left.\frac{\partial^2 h_i(\mathbf{x}, k)}{\partial x_j\,\partial x_{j'}}\right|_{\mathbf{x}=\mathbf{m}}, \tag{3.17}$$

$\mathbf{e}_i = (0 \cdots 0\ 1\ 0 \cdots 0)^T$ is a unit vector in direction of the coordinate axis i, that is, it has a 1 at position i and 0 at other positions.

The prediction and update steps of the second order EKF can be computed in this toolbox with functions `ekf_predict2` and `ekf_update2`, respectively. By taking the second order terms into account, however, doesn't quarantee, that the results get any better. Depending on problem they might even get worse, as we shall see in the later examples.

3.1.5 The Limitations of EKF

As discussed in, for example, (**?**) the EKF has a few serious drawbacks, which should be kept in mind when it's used:

1. As we shall see in some of the later demonstrations, the linear and quadratic transformations produces realiable results only when the error propagation can be well approximated by a linear or a quadratic function. If this condition is not met, the performance of the filter can be extremely poor. At worst, its estimates can diverge altogether.

2. The Jacobian matrices (and Hessian matrices with second order filters) need to exist so that the transformation can be applied. However, there are cases, where this isn't true. For example, the system might be jump-linear, in which the parameters can change abruptly (**?**).

3. In many cases the calculation of Jacobian and Hessian matrices can be a very difficult process, and its also prone to human errors (both derivation and programming). These errors are usually very hard to debug, as its hard to see which parts of the system produces the errors by looking at the estimates, especially as usually we don't know which kind of performance we should expect. For example, in the last demonstration (Reentry Vehicle Tracking) the first order derivatives were quite troublesome to calcute, even though the equations themselves were relatively simple. The second order derivatives would have even taken many more times of work.

3.1.6 Extended Kalman smoother

The difference between the first order extended Kalman smoother (**??**) and the traditional Kalman smoother is the same as the difference between first order EKF and KF, that is, matrix $\mathbf{A}_k$ in Kalman smoother is replaced with Jacobian $\mathbf{F}_\mathbf{x}(\mathbf{m}_{k-1}, k-1)$, and $\mathbf{m}_{k+1}^-$ is calculated using the model function $\mathbf{f}$. Thus, the

equations for the extended Kalman smoother can be written as

$$\begin{aligned}
\mathbf{m}_{k+1}^{-} &= \mathbf{f}(\mathbf{m}_k, k) \\
\mathbf{P}_{k+1}^{-} &= \mathbf{F}_{\mathbf{x}}(\mathbf{m}_k, k)\, \mathbf{P}_k\, \mathbf{F}_{\mathbf{x}}^{T}(\mathbf{m}_k, k) + \mathbf{Q}_k \\
\mathbf{C}_k &= \mathbf{P}_k\, \mathbf{F}_{\mathbf{x}}^{T}(\mathbf{m}_k, k)\, [\mathbf{P}_{k+1}^{-}]^{-1} \\
\mathbf{m}_k^s &= \mathbf{m}_k + \mathbf{C}_k\, [\mathbf{m}_{k+1}^s - \mathbf{m}_{k+1}^{-}] \\
\mathbf{P}_k^s &= \mathbf{P}_k + \mathbf{C}_k\, [\mathbf{P}_{k+1}^s - \mathbf{P}_{k+1}^{-}]\, \mathbf{C}_k^T.
\end{aligned} \tag{3.18}$$

First order smoothing solutiong with a RTS type smoother can be computed with function `erts_smooth1`, and with forward-backward type smoother the computation can be done with function `etf_smooth1`.

Higher order smoothers are also possible, but not described here, as they are not currently implemented in this toolbox.

3.2 Demonstration: Tracking a random sine signal

Next we consider a simple, yet practical, example of a nonlinear dynamic system, in which we estimate a random sine signal using the extended Kalman filter. By random we mean that the angular velocity and the amplitude of the signal can vary through time. In this example the nonlinearity in the system is expressed through the measurement model, but it would also be possible to express it with the dynamic model and let the measurement model be linear.

The state vector in this case can be expressed as

$$\mathbf{x}_k = \begin{pmatrix} \theta_k & \omega_k & a_k \end{pmatrix}^T, \tag{3.19}$$

where θ_k is the parameter for the sine function on time step k, ω_k is the angular velocity on time step k and a_k is the amplitude on time step k. The evolution of parameter θ is modelled with a discretized Wiener velocity model, where the velocity is now the angular velocity:

$$\frac{d\theta}{dt} = \omega. \tag{3.20}$$

The values of ω and a are perturbed with one dimensional white noise processes $w_a(t)$ and $w_w(t)$, so the signal isn't totally deterministic:

$$\frac{da}{dt} = w_a(t) \tag{3.21}$$

$$\frac{dw}{dt} = w_w(t). \tag{3.22}$$

Thus, the continous-time dynamic equation can be written as

$$\frac{d\mathbf{x}(t)}{dt} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \mathbf{x}(t) + \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \mathbf{w}(t), \tag{3.23}$$

where the white noise process $\mathbf{w}(t)$ has power spectral density

$$\mathbf{Q}_c = \begin{pmatrix} q_1 & 0 \\ 0 & q_2 \end{pmatrix}. \tag{3.24}$$

Variables q_1 and q_2 describe the strengths of random perturbations of the angular velocity and the amplitude, respectively, which are in this demonstration are set to $q_1 = 0.2$ and $q_2 = 0.1$. By using the equation (15) the discretized form of the dynamic equation can written as

$$\mathbf{x}_k = \begin{pmatrix} 1 & \Delta\mathrm{t} & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \mathbf{x}_{k-1} + \mathbf{q}_{k-1}, \tag{3.25}$$

where $\Delta\mathrm{t}$ is the step size (with value $\Delta\mathrm{t} = 0.01$ in this case), and using the equation (16) the covariance matrix $\mathbf{Q}_{k-1}$ of the discrete Gaussian white noise process $\mathbf{q}_{k-1} \sim N(\mathbf{0}, \mathbf{Q}_{k-1})$ can be easily computed to give

$$\mathbf{Q}_{k-1} = \begin{pmatrix} \frac{1}{3}\Delta\mathrm{t}^3\, q_1 & \frac{1}{2}\Delta\mathrm{t}^2\, q_1 & 0 \\ \frac{1}{2}\Delta\mathrm{t}^2\, q_1 & \Delta\mathrm{t}\, q_1 & 0 \\ 0 & 0 & \Delta\mathrm{t}\, q_2 \end{pmatrix}. \tag{3.26}$$

As stated above, the non-linearity in this case is expressed by the measurement model, that is, we propagate the current state through a non-linear measurement function $\mathbf{h}(\mathbf{x}_k, k)$ to get an actual measurement. Naturally the function in this case is the actual sine function

$$\mathbf{h}(\mathbf{x}_k, k) = a_k \sin(\theta_k). \tag{3.27}$$

With this the measurement model can be written as

$$y_k = \mathbf{h}(\mathbf{x}_k, k) + r_k = a_k \sin(\theta_k) + r_k, \tag{3.28}$$

where r_k is white, univariate Gaussian noise with zero mean and variance $\sigma_r = 1$.

The derivatives of the measurement function with respect to state variables are

$$\begin{aligned} \frac{\partial \mathbf{h}(\mathbf{x}_k, k)}{\partial \theta_k} &= a_k \cos(\theta_k) \\ \frac{\partial \mathbf{h}(\mathbf{x}_k, k)}{\partial \omega_k} &= 0 \\ \frac{\partial \mathbf{h}(\mathbf{x}_k, k)}{\partial a_k} &= \sin(\theta_k), \end{aligned} \tag{3.29}$$

so the Jacobian matrix (actually in this case, a vector, as the measurements are only one dimensional) needed by the EKF can be written as

$$\mathbf{H}_{\mathbf{x}}(\mathbf{m}, k) = \begin{pmatrix} a_k \cos(\theta_k) & 0 & \sin(\theta_k) \end{pmatrix}. \tag{3.30}$$

We also filter the signal with second order EKF, so we need to evaluate the Hessian matrix of the measurement model function. In this case the second order derivatives of $\mathbf{h}$ with respect to all state variables can written as

$$\begin{aligned}
\frac{\partial^2 \mathbf{h}(\mathbf{x}_k, k)}{\partial \theta_k \partial \theta_k} &= -a_k \sin(\theta_k) \\
\frac{\partial^2 \mathbf{h}(\mathbf{x}_k, k)}{\partial \theta_k \partial \omega_k} &= 0 \\
\frac{\partial^2 \mathbf{h}(\mathbf{x}_k, k)}{\partial \theta_k \partial a_k} &= \cos(\theta_k) \\
\frac{\partial^2 \mathbf{h}(\mathbf{x}_k, k)}{\partial \omega_k \partial \omega_k} &= 0 \\
\frac{\partial^2 \mathbf{h}(\mathbf{x}_k, k)}{\partial \omega_k \partial a_k} &= 0 \\
\frac{\partial^2 \mathbf{h}(\mathbf{x}_k, k)}{\partial a_k \partial a_k} &= 0.
\end{aligned} \tag{3.31}$$

With these the Hessian matrix can expressed as

$$\mathbf{H}_{\mathbf{xx}}(\mathbf{m}, k) = \begin{pmatrix} -a_k \sin(\theta_k) & 0 & \cos(\theta_k) \\ 0 & 0 & 0 \\ \cos(\theta_k) & 0 & 0 \end{pmatrix}. \tag{3.32}$$

Note that as the measurements are only one dimensional we need to evaluate only one Hessian, and as the expressions are rather simple the computation of this Hessian is trivial. In case of higher dimensions we would need to evaluate the Hessian for each dimension separately, which could easily result in high amount of dense algebra.

In this demonstration program the measurement function (55) is computed with the following code:

```
function Y = ekf_demo1_h(x,param)
  f = x(1,:);
  a = x(3,:);
  Y = a.*sin(f);
  if size(x,1) == 7 Y = Y + x(7,:); end
```

where the parameter `x` is a vector containing a single state value, or a matrix containing multiple state values. It is also necessary to include the parameter `param`, which contains the other possible parameters for the functions (not present in this case). The last three lines are included for the augmented version of unscented Kalman filter (UKF), which is described later in this document. The Jacobian matrix of the measurement function (eq. (58)) is computed with the following function:

```
function dY = ekf_demo1_dh_dx(x, param)
  f = x(1,:);
  w = x(2,:);
  a = x(3,:);
  dY = [(a.*cos(f))' zeros(size(f,2),1) (sin(f))'];
```

The Hessian matrix of the measurement function (eq. 60) is computed with the following function:

```
function df = ekf_sine_d2h_dx2(x,param)
  f = x(1);
  a = x(3);
  df = zeros(1,3,3);
  df(1,:,:) = [-a*sin(f) 0 cos(f); 0 0 0; cos(f) 0 0];
```

These functions are defined in files `efk_sine_h.m`, `ekf_sine_dh_dx.m` and `ekf_sine_d2h_dx2.m`, respectively. The handles of these functions are saved in the actual demonstration script file (`ekf_sine_demo.m`) with the following code lines:

```
h_func = @ekf_sine_h;
dh_dx_func = @ekf_sine_dh_dx;
d2h_dx2_func = @ekf_sine_d2h_dx2;
```

It is also important to check out that the implementation on calculating the derivatives is done right, as it is, especially with more complex models, easy to make errors in the equations. This can be done with function `der_check`:

```
der_check(h_func, dh_dx_func, 1, [f w a]');
```

The third parameter with value `1` signals that we want to test the derivative of function's first (and in this case the only) dimension. Above we have assumed, that the variable `f` contains the parameter value for the sine function, `w` the angular velocity of the signal and `a` the amplitude of the signal.

After we have discretized the dynamic model and generated the real states and measurements same as in the previous example (the actual code lines are not stated here, see the full source code at end of this document), we can use the EKF to get the filtered estimates for the state means and covariances. The filtering (with first order EKF) is done almost the same as in the previous example:

```
MM = zeros(size(M,1),size(Y,2)); PP =
zeros(size(M,1),size(M,1),size(Y,2));

for k=1:size(Y,2)
  [M,P] = ekf_predict1(M,P,A,Q);
  [M,P] = ekf_update1(M,P,Y(:,k),dh_dx_func,R*eye(1),h_func);
  MM(:,k) = M;
  PP(:,:,k) = P;
end
```

As the model dynamics are in this case linear the prediction step functions exactly the same as in the case of traditional Kalman filter. In update step we pass the handles to the measurement model function and it's derivative function and the variance of measurement noise (parameters 6, 4 and 5, respectively), in addition to other parameters. These functions also have additional parameters, which might be needed in some cases. For example, the dynamic and measurement model functions might have parameters, which are needed when those functions are called. See the full function specifications in chapter **??** for more details about the parameters.

With second order EKF the filtering loop remains almost the same with the exception of update step:

```
MM2 = zeros(size(M,1),size(Y,2));
PP2 = zeros(size(M,1),size(M,1),size(Y,2));

for k=1:size(Y,2)
  [M,P] = ekf_predict1(M,P,A,Q);
  [M,P] = ekf_update2(M,P,Y(:,k),dh_dx_func,...
    d2h_dx2_func,R*eye(1),h_func);
  MM(:,k) = M; PP(:,:,k) = P;
end
```

The smoothing of state estimates using the extended RTS smoother is done sameways as in the previous example:

```
[SM1,SP1] = erts_smooth1(MM,PP,A,Q);
```

With the extended forward-backward smoother the smoothing is done with the following function call:

```
[SM2,SP2] = etf_smooth1(MM,PP,Y,A,Q,[],[],[],...
  dh_dx_func,R*eye(1),h_func);
```

Here we have assigned empty vectors for parameters 6,7 and 8 (inverse prediction, its derivative w.r.t. to noise and its parameters, respectively), because they are not needed in this case.

To visualize the filtered and smoothed signal estimates we must evaluate the measurement model function with every state estimate to project the estimates to the measurement space. This can be done with the built-in Matlab function `feval`:

```
Y_m = feval(h_func, MM);
```

The filtered and smoothed estimates of the signals using the first order EKF, ERTS and ETF are plotted in figures 4, 5 and 6, respectively. The estimates produced by second order EKF are not plotted as they do not differ much from first order ones. As can be seen from the figures both smoothers give clearly better estimates than

the filter. Especially in the beginning of the signal it takes a while for the filter to catch on to right track.

The difference between the smoothers doesn't become clear just by looking these figures. In some cases the forward-backward smoother gives a little better estimates, but it tends to be more sensitive about numerical accuracy and the process and measurement noises. To make a comparison between the performances of different methods we have listed the average of root mean square errors (RMSE) on 100 Monte Carlo simulations with different methods in table 1. In addition to RMSE of each state variable we also provide the estimation error in measurement space, because we might be more interested in estimating the actual value of signal than its components. Usually, however, the primary goal of these methods is to estimate the hidden state variables. The following methods were used:

- EKF1: First order extended Kalman filter.
- ERTS1: First order extended Rauch-Tung-Striebel smoother.
- ETF1: First order extended Forward-Backward smoother.
- EKF2: Second order extended Kalman filter.
- ERTS2: First order extended Rauch-Tung-Striebel smoother applied to second order EKF estimates.
- ETF2: First order extended Forward-Backward smoother applied to second order EKF estimates.
- UKF: unscented Kalman filter.
- URTS: unscented Rauch-Tung-Striebel smoother.

From the errors we can see that with filters EKF2 gives clearly the lowest errors with variables θ and a. Due to this also with smoothers ERTS2 and ETF2 give clearly lower errors than others. On the other hand EKF1 gives the lowest estimation error with variable ω. Furthermore, with filters EKF1 also gives lowest error in measurement space. Each smoother, however, gives approximately the same error in measurement space. It can also be seen, that the UKF functions the worst in this case. This is due to linear and quadratic approximations used in EKF working well with this model. However, with more nonlinear models UKF is often superior over EKF, as we shall see in later sections.

In all, none of the used methods proved to be clearly superior over the others with this model. It is clear, however, that EKF should be preferred over UKF as it gives lower error and is slightly less demanding in terms of computation power. Whether first or second order EKF should be used is ultimately up to the goal of application. If the actual signal value is of interest, which is usually the case, then one should use first order EKF, but second order one might better at predicting new signal values as the variables θ and a are closer to real ones on average.

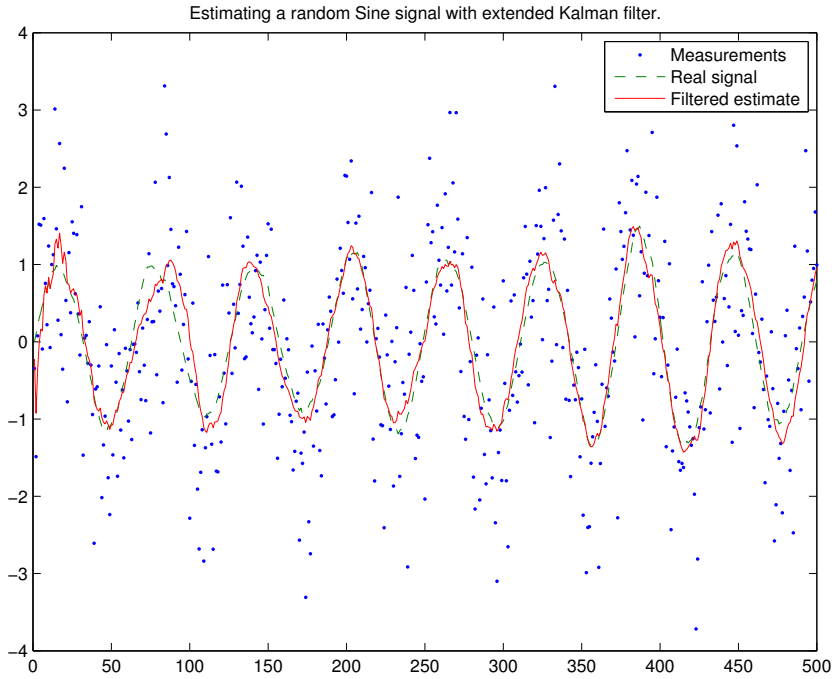


Figure 3.1: Filtered estimate of the sine signal using the first order extended Kalman filter.

Method	*RMSE[θ]*	*RMSE[ω]*	*RMSE[a]*	*RMSE[y]*
EKF1	0.64	0.53	0.40	0.24
ERTS1	0.52	0.31	0.33	0.15
ETF1	0.53	0.31	0.34	0.15
EKF2	0.34	0.54	0.31	0.29
ERTS2	0.24	0.30	0.18	0.15
ETF2	0.24	0.30	0.18	0.15
UKF	0.59	0.56	0.39	0.27
URTS	0.45	0.30	0.30	0.15

Table 3.1: RMSEs of estimating the random sinusoid over 100 Monte Carlo simulations.

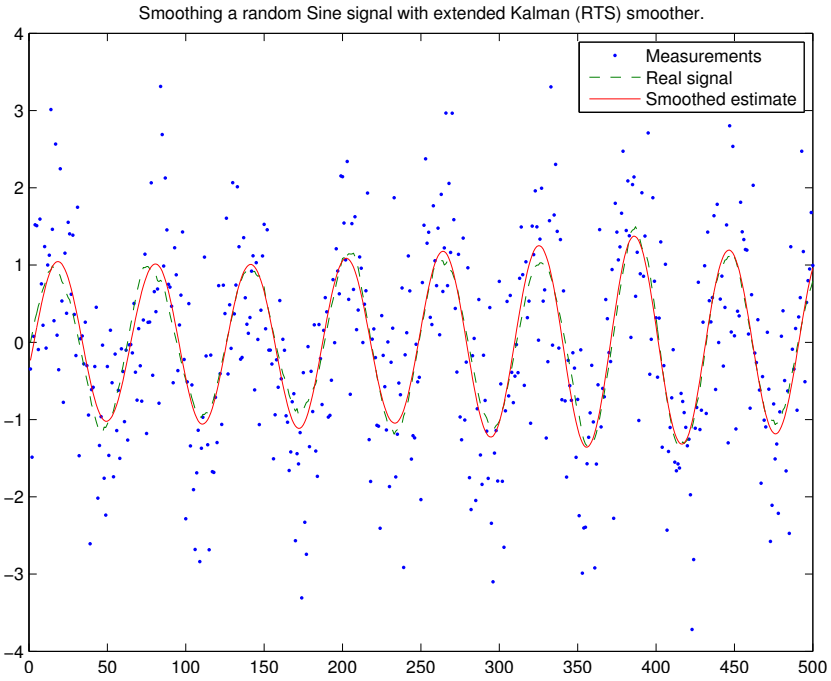


Figure 3.2: Smoothed estimate of the sine signal using the extended Kalman (RTS) smoother.

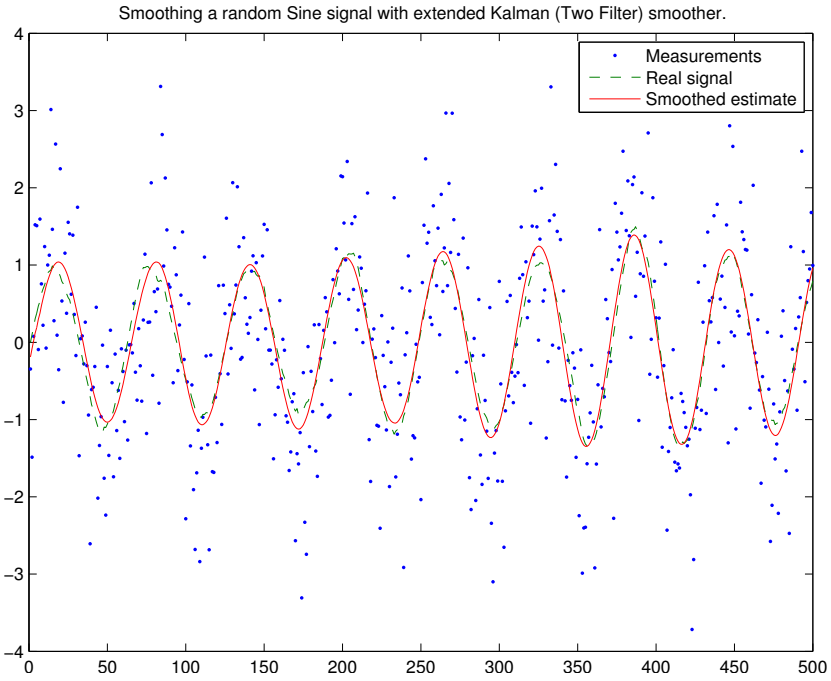


Figure 3.3: Smoothed estimate of the sine signal using a combination of two extended Kalman filters.

3.3 Unscented Kalman Filter

3.3.1 Unscented Transform

Like Taylor series based approximation presented above also the *unscented transform* (UT) (**???**) can be used for forming a Gaussian approximation to the joint distribution of random variables $\mathbf{x}$ and $\mathbf{y}$, which are defined with equations (29). In UT we deterministically choose a fixed number of sigma points, which capture the desired moments (at least mean and covariance) of the original distribution of $\mathbf{x}$ exactly. After that we propagate the sigma points through the non-linear function $\mathbf{g}$ and estimate the moments of the transformed variable from them.

The advantage of UT over the Taylor series based approximation is that UT is better at capturing the higher order moments caused by the non-linear transform, as discussed in (**?**). Also the Jacobian and Hessian matrices are not needed, so the estimation procedure is in general easier and less error-prone.

The unscented transform can be used to provide a Gaussian approximation for the joint distribution of variables $\mathbf{x}$ and $\mathbf{y}$ of the form

$$\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} \sim \mathrm{N}\left(\begin{pmatrix} \mathbf{m} \\ \boldsymbol{\mu}_U \end{pmatrix}, \begin{pmatrix} \mathbf{P} & \mathbf{C}_U \\ \mathbf{C}_U^T & \mathbf{S}_U \end{pmatrix} \right). \tag{3.33}$$

The (nonaugmented) transformation is done as follows:

1. Compute the set of $2n + 1$ sigma points from the columns of the matrix $\sqrt{(n+\lambda)\,\mathbf{P}}$:

$$\begin{aligned} \mathbf{x}^{(0)} &= \mathbf{m} \\ \mathbf{x}^{(i)} &= \mathbf{m} + \left[\sqrt{(n+\lambda)\,\mathbf{P}}\right]_i, \quad i = 1, \ldots, n \\ \mathbf{x}^{(i)} &= \mathbf{m} - \left[\sqrt{(n+\lambda)\,\mathbf{P}}\right]_i, \quad i = n+1, \ldots, 2n \end{aligned} \tag{3.34}$$

   and the associated weights:

$$\begin{aligned} W_m^{(0)} &= \lambda/(n+\lambda) \\ W_c^{(0)} &= \lambda/(n+\lambda) + (1 - \alpha^2 + \beta) \\ W_m^{(i)} &= 1/\{2(n+\lambda)\}, \quad i = 1, \ldots, 2n \\ W_c^{(i)} &= 1/\{2(n+\lambda)\}, \quad i = 1, \ldots, 2n. \end{aligned} \tag{3.35}$$

   Parameter λ is a scaling parameter, which is defined as

$$\lambda = \alpha^2\,(n + \kappa) - n. \tag{3.36}$$

   The positive constants α, β and κ are used as parameters of the method.

2. Propagate each of the sigma points through non-linearity as

$$\mathbf{y}^{(i)} = \mathbf{g}(\mathbf{x}^{(i)}), \quad i = 0, \ldots, 2n. \tag{3.37}$$

3. Calculate the mean and covariance estimates for $\mathbf{y}$ as

$$\boldsymbol{\mu}_U \approx \sum_{i=0}^{2n} W_m^{(i)}\, \mathbf{y}^{(i)} \tag{3.38}$$

$$\mathbf{S}_U \approx \sum_{i=0}^{2n} W_c^{(i)}\, (\mathbf{y}^{(i)} - \boldsymbol{\mu}_U)\, (\mathbf{y}^{(i)} - \boldsymbol{\mu}_U)^T. \tag{3.39}$$

4. Estimate the cross-covariance between $\mathbf{x}$ and $\mathbf{y}$ as

$$\mathbf{C}_U \approx \sum_{i=0}^{2n} W_c^{(i)}\, (\mathbf{x}^{(i)} - \mathbf{m})\, (\mathbf{y}^{(i)} - \boldsymbol{\mu}_U)^T. \tag{3.40}$$

The square root of positive definite matrix $\mathbf{P}$ is defined as $\mathbf{A} = \sqrt{\mathbf{P}}$, where

$$\mathbf{P} = \mathbf{A}\mathbf{A}^T. \tag{3.41}$$

To calculate the matrix $\mathbf{A}$ we can use, for example, lower triangular matrix of the Cholesky factorialization, which can be computed with built-in Matlab function `chol`. For convience, we have provided a function (`schol`, which computes the factorialization also for positive semidefinite matrices.

3.3.2 The Matrix Form of UT

The unscented transform described above can be written conviently in matrix form as follows:

$$\mathbf{X} = \begin{bmatrix} \mathbf{m} & \cdots & \mathbf{m} \end{bmatrix} + \sqrt{c} \begin{bmatrix} \mathbf{0} & \sqrt{\mathbf{P}} & -\sqrt{\mathbf{P}} \end{bmatrix} \tag{3.42}$$

$$\mathbf{Y} = \mathbf{g}(\mathbf{X}) \tag{3.43}$$

$$\boldsymbol{\mu}_U = \mathbf{Y}\, \mathbf{w}_m \tag{3.44}$$

$$\mathbf{S}_U = \mathbf{Y}\, \mathbf{W}\, \mathbf{Y}^T \tag{3.45}$$

$$\mathbf{C}_U = \mathbf{X}\, \mathbf{W}\, \mathbf{Y}^T, \tag{3.46}$$

where $\mathbf{X}$ is the matrix of sigma points, function $\mathbf{g}(\cdot)$ is applied to each column of the argument matrix separately, $c = \alpha^2\,(n + \kappa)$, and vector $\mathbf{w}_m$ and matrix $\mathbf{W}$ are defined as follows:

$$\mathbf{w}_m = \begin{bmatrix} W_m^{(0)} & \cdots & W_m^{(2n)} \end{bmatrix}^T \tag{3.47}$$

$$\begin{aligned} \mathbf{W} = {} & \left(\mathbf{I} - \begin{bmatrix} \mathbf{w}_m & \cdots & \mathbf{w}_m \end{bmatrix}\right) \\ & \times \operatorname{diag}(W_c^{(0)} \cdots W_c^{(2n)}) \\ & \times \left(\mathbf{I} - \begin{bmatrix} \mathbf{w}_m & \cdots & \mathbf{w}_m \end{bmatrix}\right)^T. \end{aligned} \tag{3.48}$$

See (Särkkä, 2006) for proof for this.

3.3.3 Unscented Kalman Filter

The *unscented Kalman filter* (UKF) (**???**) makes use of the *unscented transform* described above to give a Gaussian approximation to the filtering solutions of nonlinear optimal filtering problems of form (same as eq. (36), but restated here for convience)

$$\begin{aligned} \mathbf{x}_k &= \mathbf{f}(\mathbf{x}_{k-1}, k-1) + \mathbf{q}_{k-1} \\ \mathbf{y}_k &= \mathbf{h}(\mathbf{x}_k, k) + \mathbf{r}_k, \end{aligned} \tag{3.49}$$

where $\mathbf{x}_k \in \mathbb{R}^n$ is the state, $\mathbf{y}_k \in \mathbb{R}^m$ is the measurement, $\mathbf{q}_{k-1} \sim \mathrm{N}(\mathbf{0}, \mathbf{Q}_{k-1})$ is the Gaussian process noise, and $\mathbf{r}_k \sim \mathrm{N}(\mathbf{0}, \mathbf{R}_k)$ is the Gaussian measurement noise.

Using the matrix form of UT described above the *prediction* and *update* steps of the UKF can computed as follows:

- *Prediction:* Compute the predicted state mean $\mathbf{m}_k^-$ and the predicted covariance $\mathbf{P}_k^-$ as

$$\begin{aligned} \mathbf{X}_{k-1} &= \begin{bmatrix} \mathbf{m}_{k-1} & \cdots & \mathbf{m}_{k-1} \end{bmatrix} + \sqrt{c} \begin{bmatrix} \mathbf{0} & \sqrt{\mathbf{P}_{k-1}} & -\sqrt{\mathbf{P}_{k-1}} \end{bmatrix} \\ \hat{\mathbf{X}}_k &= \mathbf{f}(\mathbf{X}_{k-1}, k-1) \\ \mathbf{m}_k^- &= \hat{\mathbf{X}}_k \, \mathbf{w}_m \\ \mathbf{P}_k^- &= \hat{\mathbf{X}}_k \, \mathbf{W} \, [\hat{\mathbf{X}}_k]^T + \mathbf{Q}_{k-1}. \end{aligned} \tag{3.50}$$

- *Update:* Compute the predicted mean $\boldsymbol{\mu}_k$ and covariance of the measurement $\mathbf{S}_k$, and the cross-covariance of the state and measurement $\mathbf{C}_k$:

$$\begin{aligned} \mathbf{X}_k^- &= \begin{bmatrix} \mathbf{m}_k^- & \cdots & \mathbf{m}_k^- \end{bmatrix} + \sqrt{c} \begin{bmatrix} \mathbf{0} & \sqrt{\mathbf{P}_k^-} & -\sqrt{\mathbf{P}_k^-} \end{bmatrix} \\ \mathbf{Y}_k^- &= \mathbf{h}(\mathbf{X}_k^-, k) \\ \mu_k &= \mathbf{Y}_k^- \, \mathbf{w}_m \\ \mathbf{S}_k &= \mathbf{Y}_k^- \, \mathbf{W} \, [\mathbf{Y}_k^-]^T + \mathbf{R}_k \\ \mathbf{C}_k &= \mathbf{X}_k^- \, \mathbf{W} \, [\mathbf{Y}_k^-]^T. \end{aligned} \tag{3.51}$$

  Then compute the filter gain $\mathbf{K}_k$ and the updated state mean $\mathbf{m}_k$ and covariance $\mathbf{P}_k$:

$$\begin{aligned} \mathbf{K}_k &= \mathbf{C}_k \, \mathbf{S}_k^{-1} \\ \mathbf{m}_k &= \mathbf{m}_k^- + \mathbf{K}_k \, [\mathbf{y}_k - \boldsymbol{\mu}_k] \\ \mathbf{P}_k &= \mathbf{P}_k^- - \mathbf{K}_k \, \mathbf{S}_k \, \mathbf{K}_k^T. \end{aligned} \tag{3.52}$$

The prediction and update steps of the nonaugmented UKF can be computed with functions `ukf_predict1` and `ukf_update1`, respectively.

3.3.4 Augmented UKF

It is possible to modify the UKF procedure described above by forming an *augmented* state variable, which concatenates the state and noise components together, so that the effect of process and measurement noises can be used to better capture the odd-order moment information. This requires that the sigma points generated during the predict step are also used in the update step, so that the effect of noise terms are truly propagated through the nonlinearity (**?**). If, however, we generate new sigma points in the update step the augmented approach give the same results as the nonaugmented, if we had assumed that the noises were additive. If the noises are not additive the augmented version should produce more accurate estimates than the nonaugmented version, even if new sigma points are created during the update step.

The *prediction* and *update* steps of the augmented UKF in matrix form are as follows:

- *Prediction:* Form a matrix of sigma points of the augmented state variable $\tilde{\mathbf{x}}_{k-1} = \begin{bmatrix}\mathbf{x}_{k-1}^T & \mathbf{q}_{k-1}^T & \mathbf{r}_{k-1}^T\end{bmatrix}^T$ as

$$\tilde{\mathbf{X}}_{k-1} = \begin{bmatrix}\tilde{\mathbf{m}}_{k-1} & \cdots & \tilde{\mathbf{m}}_{k-1}\end{bmatrix} + \sqrt{c}\begin{bmatrix}\mathbf{0} & \sqrt{\tilde{\mathbf{P}}_{k-1}} & -\sqrt{\tilde{\mathbf{P}}_{k-1}}\end{bmatrix}, \quad (3.53)$$

  where

$$\tilde{\mathbf{m}}_{k-1} = \begin{bmatrix}\mathbf{m}_{k-1}^T & \mathbf{0} & \mathbf{0}\end{bmatrix}^T \text{ and } \tilde{\mathbf{P}}_{k-1} = \begin{bmatrix}\mathbf{P_{k-1}} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{Q}_{k-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{R}_{k-1}\end{bmatrix}. \quad (3.54)$$

  Then compute the predicted state mean $\mathbf{m}_k^-$ and the predicted covariance $\mathbf{P}_k^-$ as

$$\begin{aligned}\hat{\mathbf{X}}_k &= \mathbf{f}(\mathbf{X^x}_{k-1}, \mathbf{X^q}_{k-1}, k-1) \\ \mathbf{m}_k^- &= \hat{\mathbf{X}}_k\, \mathbf{w}_m \\ \mathbf{P}_k^- &= \hat{\mathbf{X}}_k\, \mathbf{W}\, [\hat{\mathbf{X}}_k]^T,\end{aligned} \quad (3.55)$$

  where we have denoted the components of sigma points which correspond to actual state variables and process noise with matrices $\mathbf{X}_{k-1}^x$ and $\mathbf{X}_{k-1}^q$, respectively. The state transition function $\mathbf{f}$ is also augmented to incorporate the effect of process noise, which is now passed to the function as a second parameter. In additive noise case the process noise is directly added to the state variables, but other forms of noise effect are now also allowed.

- *Update:* Compute the predicted mean $\boldsymbol{\mu}_k$ and covariance of the measurement $\mathbf{S}_k$, and the cross-covariance of the state and measurement $\mathbf{C}_k$:

$$\begin{aligned} \mathbf{Y}_k^- &= \mathbf{h}(\hat{\mathbf{X}}_k, \mathbf{X}_{k-1}^r, k) \\ \mu_k &= \mathbf{Y}_k^- \, \mathbf{w}_m \\ \mathbf{S}_k &= \mathbf{Y}_k^- \, \mathbf{W} \, [\mathbf{Y}_k^-]^T \\ \mathbf{C}_k &= \hat{\mathbf{X}}_k \, \mathbf{W} \, [\mathbf{Y}_k^-]^T, \end{aligned} \tag{3.56}$$

  where we have denoted the component of sigma points corresponding to measurement noise with matrix $\mathbf{X}_{k-1}^r$. Like the state transition function $\mathbf{f}$ also the measurement function $\mathbf{h}$ is now augmented to incorporate the effect of measurement noise, which is passed as a second parameter to the function.

  Then compute the filter gain $\mathbf{K}_k$ and the updated state mean $\mathbf{m}_k$ and covariance $\mathbf{P}_k$:

$$\begin{aligned} \mathbf{K}_k &= \mathbf{C}_k \, \mathbf{S}_k^{-1} \\ \mathbf{m}_k &= \mathbf{m}_k^- + \mathbf{K}_k \, [\mathbf{y}_k - \boldsymbol{\mu}_k] \\ \mathbf{P}_k &= \mathbf{P}_k^- - \mathbf{K}_k \, \mathbf{S}_k \, \mathbf{K}_k^T. \end{aligned} \tag{3.57}$$

Note that nonaugmented form UKF is computationally less demanding than augmented form UKF, because it creates a smaller number of sigma points during the filtering procedure. Thus, the usage of the nonaugmented version should be preferred over the nonaugmented version, if the propagation of noise terms doesn't improve the accuracy of the estimates.

The prediction and update steps of the augmented UKF can be computed with functions `ukf_predict3` and `ukf_update3`, respectively. These functions concatenates the state variables, process and measurements noises to the augmented variables, as was done above.

It is also possible to separately concatenate only the state variables and process noises during prediction step and state variables and measurement noises during update step. Filtering solution based on this formulation can be computed with functions `ukf_predict2` and `ukf_update2`. However, these functions create new sigma points during the update step in addition to ones created during prediction step, and hence the higher moments might not get captured so effectively in cases, where the noise terms are additive.

3.3.5 Unscented Kalman Smoother

The Rauch-Rung-Striebel type smoother using the unscented transformation (**?**) can be used for computing a Gaussian approximation to the smoothing distribution of the step k:

$$p(\mathbf{x}_k|\mathbf{y}_{1:T}) \sim N(\mathbf{x}_k|\mathbf{m}_k^s, \mathbf{P}_k^s), \tag{3.58}$$

as follows (using again the matrix form):

- Form a matrix of sigma points of the augmented state variable $\tilde{\mathbf{x}}_{k-1} = \begin{bmatrix}\mathbf{x}_{k-1}^T & \mathbf{q}_{k-1}^T\end{bmatrix}^T$ as

$$\tilde{\mathbf{X}}_{k-1} = \begin{bmatrix}\tilde{\mathbf{m}}_{k-1} & \cdots & \tilde{\mathbf{m}}_{k-1}\end{bmatrix} + \sqrt{c}\begin{bmatrix}\mathbf{0} & \sqrt{\tilde{\mathbf{P}}_{k-1}} & -\sqrt{\tilde{\mathbf{P}}_{k-1}}\end{bmatrix}, \quad (3.59)$$

  where

$$\tilde{\mathbf{m}}_{k-1} = \begin{bmatrix}\mathbf{m}_{k-1}^T & \mathbf{0}\end{bmatrix}^T \text{ and } \tilde{\mathbf{P}}_{k-1} = \begin{bmatrix}\mathbf{P_{k-1}} & \mathbf{0} \\ \mathbf{0} & \mathbf{Q}_{k-1}\end{bmatrix}. \quad (3.60)$$

- Propagate the sigma points through the dynamic model:

$$\tilde{\mathbf{X}}_{k+1}^- = \mathbf{f}(\tilde{\mathbf{X}}_k^x, \tilde{\mathbf{X}}_k^q, k), \quad (3.61)$$

  where $\tilde{\mathbf{X}}_k^x$ and $\tilde{\mathbf{X}}_k^q$ denotes the parts of sigma points, which correspond to $\mathbf{x}_k$ and $\mathbf{q}_k$, respectively.

- Compute the predicted mean $\mathbf{m}_{k+1}^-$, covariance $\mathbf{P}_{k+1}^-$ and cross-covariance $\mathbf{C}_{k+1}$:

$$\begin{aligned}\mathbf{m}_{k+1}^- &= \tilde{\mathbf{X}}_{k+1}^{-x}\, \mathbf{w}_m \\ \mathbf{P}_{k+1}^- &= \tilde{\mathbf{X}}_{k+1}^{-x}\, \mathbf{W}\, [\tilde{\mathbf{X}}_{k+1}^{-x}]^T \\ \mathbf{C}_{k+1} &= \tilde{\mathbf{X}}_{k+1}^{-x}\, \mathbf{W}\, [\tilde{\mathbf{X}}_k^{x}]^T,\end{aligned} \quad (3.62)$$

  where $\tilde{\mathbf{X}}_{k+1}^{-x}$ denotes the part of propagated sigma points $\tilde{\mathbf{X}}_{k+1}^-$, which corresponds to $\mathbf{x}_k$.

- Compute the smoother gain $\mathbf{D}_k$, the smoothed mean $\mathbf{m}_k^s$ and the covariance $\mathbf{P}_k^s$:

$$\begin{aligned}\mathbf{D}_k &= \mathbf{C}_{k+1}[\mathbf{P}_{k+1}^-]^{-1} \\ \mathbf{m}_k^s &= \mathbf{m}_k + \mathbf{D}_k[\mathbf{m}_{k+1}^s - \mathbf{m}_{k+1}^-] \\ \mathbf{P}_k^s &= \mathbf{P}_k + \mathbf{D}_k[\mathbf{P}_{k+1}^s - \mathbf{P}_{k+1}^-]\mathbf{D}_k^T.\end{aligned} \quad (3.63)$$

The smoothing solution of this augmented type RTS smoother can be computed with function `urts_smooth2`. Also a nonaugmented version of this type smoother has been implemented, and a smoothing solution with that can be computed with function `urts_smooth1`.

3.4 Gauss-Hermite Kalman Filter

3.4.1 Gauss-Hermite Cubature Transformation

To unify many of the filter variants, handling the non-linearities may be brought together to a common formulation. In Gaussian optimal filtering — also called assumed density filtering — the filtering equations follow the assumption that the filtering distributions are indeed Gaussian (**??**).

Using this setting the linear Kalman filter equations can now be adapted to the non-linear state-space model. Desired moments (at least mean and covariance) of the original distribution of $\mathbf{x}$ can be captured exactly by calculating the integrals

$$\begin{aligned} \boldsymbol{\mu}_U &= \int \mathbf{f}(\mathbf{x})\ \mathrm{N}(\mathbf{x} \mid \mathbf{m},\, \mathbf{P})\mathrm{d}\mathbf{x} \\ \mathbf{S}_U &= \int \left(\mathbf{f}(\mathbf{x}) - \boldsymbol{\mu}_U\right)\left(\mathbf{f}(\mathbf{x}) - \boldsymbol{\mu}_U\right)^{\mathsf{T}} \times \mathrm{N}(\mathbf{x} \mid \mathbf{m},\, \mathbf{P})\mathrm{d}\mathbf{x}. \end{aligned} \tag{3.64}$$

These integrals can be evaluated with practically any analytical or numerical integration method. The Gauss–Hermite quadrature rule is a one-dimensional weighted sum approximation method for solving special integrals of the previous form in a Gaussian kernel with an infinite domain. More specifically the Gauss–Hermite quadrature can be applied to integrals of form

$$\int_{-\infty}^{\infty} f(x)\ \exp(-x^2)\mathrm{d}x \approx \sum_{i=1}^{m} w_i\, f(x_i), \tag{3.65}$$

where x_i are the sample points and w_i the associated weights to use for the approximation. The sample points $x_i, i = 1, \ldots, m$, are roots of special orthogonal polynomials, namely the Hermite polynomials. The Hermite polynomial of degree p is denoted with $H_p(x)$ (see **?**, for details). The weights w_i are given by

$$w_i = \frac{2^{p-1} p! \sqrt{\pi}}{p^2 [H_{p-1}(x_i)]^2}. \tag{3.66}$$

The univariate integral approximation needs to be extended to be able to suit the multivariate case. As **?** argue, the most natural approach to grasp a multiple integral is to treat it as a sequence of nested univariate integrals and then use a univariate quadrature rule repeatedly. To extend this one-dimensional integration method to multi-dimensional integrals of form

$$\int_{\mathbb{R}^n} f(\mathbf{x})\ \exp(-\mathbf{x}^{\mathsf{T}}\mathbf{x})\mathrm{d}\mathbf{x} \approx \sum_{i=1}^{m} w_i\, f(\mathbf{x}_i), \tag{3.67}$$

we first simply form the one-dimensional quadrature rule with respect to the first dimension, then with respect to the second dimension and so on (**?**). We get the multidimensional Gauss–Hermite cubature rule by writing

$$\begin{aligned}
&\sum_{i_1} w_{i_1} \int f(x_1^{i_1}, x_2, \ldots, x_n) \exp(-x_2^2 - x_3^2 \ldots - x_n^2) \mathrm{d}x_2 \ldots \mathrm{d}x_n \\
=&\sum_{i_1,i_2} w_{i_1} w_{i_2} \int f(x_1^{i_1}, x_2^{i_2}, \ldots, x_n) \exp(-x_3^2 \ldots - x_n^2) \mathrm{d}x_3 \ldots \mathrm{d}x_n \\
=&\sum_{i_1,i_2,\ldots,i_n} w_{i_1} w_{i_2} \cdots w_{i_n} f(x_1^{i_1}, x_2^{i_2}, \ldots, x_n^{i_n}),
\end{aligned}$$

which is basically what we wanted in Equation (95). This gives us the *product rule* that simply extends the one-dimensional quadrature point set of p points in one dimension to a lattice of p^n cubature points in n dimensions. The weights for these Gauss–Hermite cubature points are calculated by the product of the corresponding one-dimensional weights.

Finally, by making a change of variable $\mathbf{x} = \sqrt{2}\sqrt{\boldsymbol{\Sigma}} + \boldsymbol{\mu}$ we get the Gauss–Hermite weighted sum approximation for a multivariate Gaussian integral, where $\boldsymbol{\mu}$ is the mean and $\boldsymbol{\Sigma}$ is the covariance of the Gaussian. The square root of the covariance matrix, denoted $\sqrt{\boldsymbol{\Sigma}}$, is a matrix such that $\boldsymbol{\Sigma} = \sqrt{\boldsymbol{\Sigma}}\sqrt{\boldsymbol{\Sigma}}^{\mathsf{T}}$.

$$\int_{\mathbb{R}^n} \mathbf{f}(\mathbf{x})\ \mathrm{N}(\mathbf{x} \mid \boldsymbol{\mu},\ \boldsymbol{\Sigma}) \mathrm{d}\mathbf{x} \approx \sum_{i_1,i_2,\ldots,i_n} w_{i_1,i_2,\ldots,i_n} f\left(\sqrt{\boldsymbol{\Sigma}}\, \boldsymbol{\xi}_{i_1,i_2,\ldots,i_n} + \boldsymbol{\mu}\right), \quad (3.68)$$

where the weight $w_{i_1,i_2,\ldots,i_n} = \frac{1}{\pi^{n/2}} w_{i_1} \cdot w_{i_2} \cdots w_{i_n}$ is given by using the one-dimensional weights, and the points are given by the Cartesian product $\boldsymbol{\xi}_{i_1,i_2,\ldots,i_n} = \sqrt{2}\,(x_{i_1}, x_{i_2}, \ldots, x_{i_n})$, where x_i is the ith one-dimensional quadrature point.

The extension of the Gauss–Hermite quadrature rule to an n-dimensional cubature rule by using the product rule lattice approach yields a rather good numerical integration method that is exact for monomials $\prod_{i=1}^{n} x_i^{k_i}$ with $k_i \leq 2p - 1$ (**?**). However, the number of cubature points grows exponentially as the number of dimensions increases. Due to this flaw the rule is not practical in applications with many dimensions. This problem is called the *curse of dimensionality*.

3.4.2 Gauss-Hermite Kalman Filter

The Gauss–Hermite Kalman filter (GHKF) algorithm of degree p is presented below. At time $k = 1, \ldots, T$ assume the posterior density function $p(\mathbf{x}_{k-1} \mid \mathbf{y}_{k-1}) = \mathrm{N}(\mathbf{m}_{k-1|k-1}, \mathbf{P}_{k-1|k-1})$ is known.

Prediction step:

1. Find the roots $x_i, i = 1, \ldots, p$, of the Hermite polynomial $H_p(x)$.

2. Calculate the corresponding weights

$$w_i = \frac{2^{p-1} p!}{p^2 [H_{p-1}(x_i)]^2}.$$

3. Use the product rule to expand the points to a n-dimensional lattice of p^n points $\boldsymbol{\xi}_i, i = 1, \ldots, p^n$, with corresponding weights.

4. Propagate the cubature points. The matrix square root is the lower triangular cholesky factor.

$$\mathbf{X}_{i,k-1|k-1} = \sqrt{2\mathbf{P}_{k-1|k-1}}\boldsymbol{\xi}_i + \mathbf{m}_{k-1|k-1}$$

5. Evaluate the cubature points with the dynamic model function

$$\mathbf{X}^*_{i,k|k-1} = \mathbf{f}(\mathbf{X}_{i,k-1|k-1}).$$

6. Estimate the predicted state mean

$$\mathbf{m}_{k|k-1} = \sum_{i=1}^{p^n} w_i \mathbf{X}^*_{i,k|k-1}.$$

7. Estimate the predicted error covariance

$$\mathbf{P}_{k|k-1} = \sum_{i=1}^{p^n} w_i \mathbf{X}^*_{i,k|k-1}\mathbf{X}^{*\mathsf{T}}_{i,k|k-1} - \mathbf{m}_{k|k-1}\mathbf{m}_{k|k-1}{}^{\mathsf{T}} + \mathbf{Q}_{k-1}.$$

Update step:

1. Repeat steps 1–3 from earlier to get the p^n cubature points and their weights.

2. Propagate the cubature points.

$$\mathbf{X}_{i,k|k-1} = \sqrt{2\mathbf{P}_{k|k-1}}\boldsymbol{\xi}_i + \mathbf{m}_{k|k-1}$$

3. Evaluate the cubature points with the help of the measurement model function

$$\mathbf{Y}_{i,k|k-1} = \mathbf{h}(\mathbf{X}_{i,k|k-1}).$$

4. Estimate the predicted measurement

$$\hat{\mathbf{y}}_{k|k-1} = \sum_{i=1}^{p^n} w_i \mathbf{Y}_{i,k|k-1}.$$

5. Estimate the innovation covariance matrix

$$\mathbf{S}_{k|k-1} = \sum_{i=1}^{p^n} w_i \mathbf{Y}_{i,k|k-1}\mathbf{Y}_{i,k|k-1}{}^{\mathsf{T}} - \hat{\mathbf{y}}_{k|k-1}\hat{\mathbf{y}}_{k|k-1}{}^{\mathsf{T}} + \mathbf{R}_k.$$

6. Estimate the cross-covariance matrix

$$\mathbf{P}_{xy,k|k-1} = \sum_{i=1}^{p^n} w_i \mathbf{X}_{i,k-1|k-1} \mathbf{Y}_{i,k|k-1}{}^{\mathsf{T}} - \mathbf{m}_{k|k-1} \hat{\mathbf{y}}_{k|k-1}{}^{\mathsf{T}}.$$

7. Calculate the Kalman gain term and the smoothed state mean and covariance

$$\begin{aligned} \mathbf{K}_k &= \mathbf{P}_{xy,k|k-1} \mathbf{S}_{k|k-1}^{-1} \\ \mathbf{m}_{k|k} &= \mathbf{m}_{k|k-1} + \mathbf{K}_k (\mathbf{y}_k - \hat{\mathbf{y}}_{k|k-1}) \\ \mathbf{P}_{k|k} &= \mathbf{P}_{k|k-1} - \mathbf{K}_k \mathbf{P}_{yy,k|k-1} \mathbf{K}_k{}^{\mathsf{T}}. \end{aligned}$$

3.4.3 Gauss-Hermite Kalman Smoother

The Gauss–Hermite Rauch–Tung–Striebel smoother (GHRTS) algorithm (**?**) of degree p is presented below. Assume the filtering result mean $\mathbf{m}_{k|k}$ and covariance $\mathbf{P}_{k|k}$ are known together with the smoothing result $p(\mathbf{x}_{k+1} \mid \mathbf{y}_{1:T}) = \mathrm{N}(\mathbf{m}_{k+1|T}, \mathbf{P}_{k+1|T})$.

1. Find the roots $x_i, i = 1, \ldots, p$, of the Hermite polynomial $H_p(x)$.

2. Calculate the corresponding weights

$$w_i = \frac{2^{p-1} p!}{p^2 [H_{p-1}(x_i)]^2}.$$

3. Use the product rule to expand the points to a n-dimensional lattice of p^n points $\boldsymbol{\xi}_i, i = 1, \ldots, p^n$, with corresponding weights.

4. Propagate the cubature points

$$\mathbf{X}_{i,k|k} = \sqrt{2\mathbf{P}_{k|k}} \boldsymbol{\xi}_i + \mathbf{m}_{k|k}.$$

5. Evaluate the cubature points with the dynamic model function

$$\mathbf{X}^*_{i,k+1|k} = \mathbf{f}(\mathbf{X}_{i,k|k}).$$

6. Estimate the predicted state mean

$$\mathbf{m}_{k+1|k} = \sum_{i=1}^{p^n} w_i \mathbf{X}^*_{i,k+1|k}.$$

7. Estimate the predicted error covariance

$$\mathbf{P}_{k+1|k} = \sum_{i=1}^{p^n} w_i \mathbf{X}^*_{i,k+1|k} \mathbf{X}^{*\mathsf{T}}_{i,k+1|k} - \mathbf{m}_{k+1|k} \mathbf{m}_{k+1|k}{}^{\mathsf{T}} + \mathbf{Q}_k.$$

8. Estimate the cross-covariance matrix

$$\mathbf{D}_{k,k+1} = \frac{1}{2n}\sum_{i=1}^{2n}\left(\mathbf{X}_{i,k|k} - \mathbf{m}_{k|k}\right)\left(\mathbf{X}^*_{i,k+1|k} - \mathbf{m}_{k+1|k}\right)^{\mathsf{T}}.$$

9. Calculate the gain term and the smoothed state mean and covariance

$$\begin{aligned}\mathbf{C}_k &= \mathbf{D}_{k,k+1}\mathbf{P}^{-1}_{k+1|k}\\ \mathbf{m}_{k|T} &= \mathbf{m}_{k|k} + \mathbf{C}_k(\mathbf{m}_{k+1|T} - \mathbf{m}_{k+1|k})\\ \mathbf{P}_{k|T} &= \mathbf{P}_{k|k} + \mathbf{C}_k(\mathbf{P}_{k+1|T} - \mathbf{P}_{k+1|k})\mathbf{C}_k{}^{\mathsf{T}}.\end{aligned}$$

3.5 Cubature Kalman Filter

3.5.1 Spherical-Radial Cubature Transformation

As in the Gauss–Hermite cubature rule based Gauss–Hermite transformation the spherical–radial cubature transformation utilizes the assumed density approach. The integrals that need to be solved are the same Gaussian integrals, but the numerical integration method differs from the product rule based Gauss–Hermite method.

The curse of dimensionality causes all product rules to be highly ineffective in integration regions with multiple dimensions. To mitigate this issue, we may seek alternative approaches to solving Gaussian integrals. The non-product rules differ from product based solutions by choosing the evaluation points directly from the domain of integration. That is, the points are not simply duplicated from one dimension to multiple dimensions, but directly chosen from the whole domain.

We constrain our interest to integrals of form

$$\mathrm{I}(\mathbf{f}) = \int_{\mathbb{R}^n} f(\mathbf{x}) \exp(-\mathbf{x}^\mathsf{T}\mathbf{x}) \mathrm{d}\mathbf{x}. \tag{3.69}$$

We make a change of variable from $\mathbf{x} \in \mathbb{R}^n$ to spherical coordinates, which lets us split the integrand into two: a radial integral

$$\mathrm{I}(\mathbf{f}) = \int_0^\infty \mathrm{S}(r)\, r^{n-1} \exp(-r^2) \mathrm{d}r, \tag{3.70}$$

and a spherical integral

$$\mathrm{S}(r) = \int_{S_n} f(r\mathbf{y}) \mathrm{d}\sigma(\mathbf{y}). \tag{3.71}$$

The spherical integral (99) can be seen as a spherical integral with the unit weighting function $w(\mathbf{y}) \equiv 1$. Now the spherical and radial integral may be interpreted separately and computed with the spherical cubature rule and the Gaussian quadrature rule respectively.

In a fully symmetric cubature point set equally weighted points are symmetrically distributed around origin. A point $\mathbf{u}$ is called the *generator* of such a set, if for the components of $\mathbf{u} = \begin{pmatrix} u_1, u_2, \ldots, u_r, 0, \ldots, 0 \end{pmatrix} \in \mathbb{R}^n$, $u_i \geq u_{i+1} > 0$, $i = 1, 2, \ldots, (r-1)$. For example, we may denote $[\mathbf{1}] \in \mathbb{R}^2$ to represent the cubature point set

$$\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \end{pmatrix} \right\},$$

where the generator is $\begin{pmatrix} 1 & 0 \end{pmatrix}^\mathsf{T}$.

To find the unknowns of a cubature rule of degree d, a set of moment equations have to be solved. This, however, may not be a simple task with increasing dimensions and polynomial degree. To reduce the size of the system of equations or the number of needed cubature points (**?**) use the *invariant theory* proposed

by Sobolev (see **?**). The invariant theory discusses how to simplify the structure of a cubature rule by using the symmetries of the region of integration. The unit hypercube, the unit hypersphere and the unit simplex all contain some symmetry.

Due to invariance theory (**?**) the integral (99) can be approximated by a third-degree spherical cubature rule that gives us the sum

$$\int_{S_n} f(r\mathbf{y})\mathrm{d}\sigma(\mathbf{y}) \approx w \sum_{i=1}^{2n} \mathbf{f}([\mathbf{u}]_i). \tag{3.72}$$

The point set $[\mathbf{u}]$ is invariant under permutations and sign changes, which means that a number of $2n$ cubature points are sufficient to approximate the integral. For the above choice, the monomials $y_1^{d_1} y_2^{d_2} \cdots y_n^{d_n}$, with the sum $\sum_{i=1}^n d_i$ being an odd integer, are integrated exactly.

To make this rule exact for all monomials up to degree three, we have to require the rule to be exact for the even dimensions $\sum_{i=1}^n d_i = \{0, 2\}$. This can be accomplished by solving the unknown parameters for a monomial function of order $n = 0$ and equivalently for a monomial function of order $n = 2$. We consider the two functions $\mathbf{f}(\cdot)$ to be of form $\mathbf{f}(\mathbf{y}) = 1$, and $\mathbf{f}(\mathbf{y}) = y_1^2$. This yields the pair of equations (**?**).

$$\begin{aligned} \mathbf{f}(\mathbf{y}) = 1: \qquad & 2nw = \int_{S_n} \mathrm{d}\sigma(\mathbf{y}) = A_n \\ \mathbf{f}(\mathbf{y}) = y_1^2: \qquad & 2wu^2 = \int_{S_n} y_1^2 \mathrm{d}\sigma(\mathbf{y}) = \frac{1}{n} A_n, \end{aligned}$$

where A_n is the surface area of the n-dimensional unit sphere. Solving these equations yields $u^2 = 1$ and $w = \frac{A_n}{2n}$. Therefore the cubature points can be chosen so that they are located at the intersection of the unit sphere and its axes.

The radial integral defined in Equation (98) can be transformed to a familiar Gauss–Laguerre form (see Abramowitz and Stegun, 1964) by making another change of variable, $t = r^2$, which yields

$$\int_0^\infty \mathrm{S}(r)\, r^{n-1} \exp(-r^2)\mathrm{d}r = \frac{1}{2} \int_0^\infty \tilde{\mathrm{S}}(t)\, t^{\frac{n}{2}-1} \exp(-t)\mathrm{d}t = \sum_{i=1}^m w_i\, \tilde{\mathrm{S}}(t_i), \tag{3.73}$$

where t_i is the ith root of *Laguerre polynomial* $L_m(t)$ and the weights w_i are given by (**?**)

$$w_i = \frac{t_i}{(m+1)^2 (L_{m+1}(t_i))^2}.$$

A first-degree Gauss–Laguerre rule is exact for $\tilde{\mathrm{S}}(t) = \{1, t\}$ (or equivalently $\mathrm{S}(r) = \{1, r^2\}$). Due to the properties of the spherical cubature rule presented

earlier, the combined spherical–radial rule vanishes by symmetry for all odd degree polynomials. Hence, to have the spherical–radial rule to be exact for all polynomials up to degree three in $\mathbf{x} \in \mathbb{R}^n$ it is sufficient to use the first degree Gauss–Laguerre rule of the form (**?**)

$$\int_0^\infty \tilde{\mathrm{S}}_\mathrm{i}(t)\, t^{\frac{n}{2}-1} \exp(-t)\mathrm{d}t = w_1\, \tilde{\mathrm{S}}_\mathrm{i}(t_1), \quad i = \{0, 1\},$$

where $\tilde{\mathrm{S}}_0(t) = 1$ and $\tilde{\mathrm{S}}_1(t) = t$. The corresponding moment equations show that the first-degree Gauss–Laguerre approximation is constructed using the point $t_1 = \frac{n}{2}$ and the weight $w_1 = \Gamma(\frac{n}{2})$, where $\Gamma(\cdot)$ is the Gamma function. The final radial form approximation can be written using Equation (101) in the form

$$\int_0^\infty \mathrm{S}(r)\, r^{n-1} \exp(-r^2)\mathrm{d}r \approx \frac{1}{2}\,\Gamma\left(\frac{n}{2}\right) \mathrm{S}\left(\sqrt{\frac{n}{2}}\right). \tag{3.74}$$

Now we have an approximation for the spherical integral in Equation (100), where the third-degree rule is acquired by the cubature point set $[\mathbf{1}]$ and weight $\frac{A_n}{2n}$. Here the surface area A_n of the $n-1$ hypersphere equals $2\frac{\pi^{n/2}}{\Gamma(n/2)}$, where $\Gamma(\cdot)$ is the Gamma function. By applying the results derived for the spherical and radial integral, we may combine Equations (100) and (102) to construct a third-degree cubature approximation for (97), which yields the elegant solution

$$\mathrm{I}(\mathbf{f}) \approx \frac{\sqrt{\pi^n}}{2n} \sum_{i=1}^{2n} \mathbf{f}\left(\sqrt{\frac{n}{2}}[\mathbf{1}]_i\right).$$

By making a change of variable we get the third-degree spherical–radial cubature rule for an arbitrary integral that is of form *non-linear function* $\times$ *Gaussian*. It can be written as

$$\int_{\mathbb{R}^n} \mathbf{f}(\mathbf{x})\ \mathrm{N}(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})\mathrm{d}\mathbf{x} \approx \sum_{i=1}^{2n} w_i \mathbf{f}\left(\sqrt{\boldsymbol{\Sigma}}\,\boldsymbol{\xi}_i + \boldsymbol{\mu}\right),$$

where the cubature points are $\boldsymbol{\xi}_i = \sqrt{n}[\mathbf{1}]_i$, the corresponding (equal) weights $w_i = \frac{1}{2n}$ and the points $[\mathbf{1}]_i$ from the intersections between the Cartesian axes and the n-dimensional unit hypersphere.

Note that the spherical–radial cubature transform coincide with the result of the unscented transform when the unscented transform is done with parameter values $\alpha = \pm 1$, $\beta = 0$ and $\kappa = 0$.

3.5.2 Spherical-Radial Cubature Kalman Filter

The cubature Kalman filter (CKF) algorithm is presented below. At time $k = 1, \ldots, T$ assume the posterior density function $p(\mathbf{x}_{k-1} \mid \mathbf{y}_{k-1}) = \mathrm{N}(\mathbf{m}_{k-1|k-1}, \mathbf{P}_{k-1|k-1})$ is known.

Prediction step:

1. Draw cubature points $\boldsymbol{\xi}_i, i = 1, \ldots, 2n$ from the intersections of the n-dimensional unit sphere and the Cartesian axes. Scale them by $\sqrt{n}$. That is
$$\boldsymbol{\xi}_i = \begin{cases} \sqrt{n}\, e_i & , \, i = 1, \ldots, n \\ -\sqrt{n}\, e_{i-n} & , \, i = n+1, \ldots, 2n \end{cases}$$

2. Propagate the cubature points. The matrix square root is the lower triangular cholesky factor.
$$\mathbf{X}_{i,k-1|k-1} = \sqrt{\mathbf{P}_{k-1|k-1}}\boldsymbol{\xi}_i + \mathbf{m}_{k-1|k-1}$$

3. Evaluate the cubature points with the dynamic model function
$$\mathbf{X}^*_{i,k|k-1} = \mathbf{f}(\mathbf{X}_{i,k-1|k-1}).$$

4. Estimate the predicted state mean
$$\mathbf{m}_{k|k-1} = \frac{1}{2n}\sum_{i=1}^{2n} \mathbf{X}^*_{i,k|k-1}.$$

5. Estimate the predicted error covariance
$$\mathbf{P}_{k|k-1} = \frac{1}{2n}\sum_{i=1}^{2n} \mathbf{X}^*_{i,k|k-1}\mathbf{X}^{*\mathsf{T}}_{i,k|k-1} - \mathbf{m}_{k|k-1}\mathbf{m}_{k|k-1}{}^{\mathsf{T}} + \mathbf{Q}_{k-1}.$$

Update step:

1. Draw cubature points $\boldsymbol{\xi}_i, i = 1, \ldots, 2n$ from the intersections of the n-dimensional unit sphere and the Cartesian axes. Scale them by $\sqrt{n}$.

2. Propagate the cubature points.
$$\mathbf{X}_{i,k|k-1} = \sqrt{\mathbf{P}_{k|k-1}}\boldsymbol{\xi}_i + \mathbf{m}_{k|k-1}$$

3. Evaluate the cubature points with the help of the measurement model function
$$\mathbf{Y}_{i,k|k-1} = \mathbf{h}(\mathbf{X}_{i,k|k-1}).$$

4. Estimate the predicted measurement
$$\hat{\mathbf{y}}_{k|k-1} = \frac{1}{2n}\sum_{i=1}^{2n} \mathbf{Y}_{i,k|k-1}.$$

5. Estimate the innovation covariance matrix

$$\mathbf{S}_{k|k-1} = \frac{1}{2n}\sum_{i=1}^{2n} \mathbf{Y}_{i,k|k-1}\mathbf{Y}_{i,k|k-1}^{\mathsf{T}} - \hat{\mathbf{y}}_{k|k-1}\hat{\mathbf{y}}_{k|k-1}^{\mathsf{T}} + \mathbf{R}_k.$$

6. Estimate the cross-covariance matrix

$$\mathbf{P}_{xy,k|k-1} = \frac{1}{2n}\sum_{i=1}^{2n} \mathbf{X}_{i,k-1|k-1}\mathbf{Y}_{i,k|k-1}^{\mathsf{T}} - \mathbf{m}_{k|k-1}\hat{\mathbf{y}}_{k|k-1}^{\mathsf{T}}.$$

7. Calculate the Kalman gain term and the smoothed state mean and covariance

$$\begin{aligned}
\mathbf{K}_k &= \mathbf{P}_{xy,k|k-1}\mathbf{S}_{k|k-1}^{-1} \\
\mathbf{m}_{k|k} &= \mathbf{m}_{k|k-1} + \mathbf{K}_k(\mathbf{y}_k - \hat{\mathbf{y}}_k \\
\mathbf{P}_{k|k} &= \mathbf{P}_{k|k-1} - \mathbf{K}_k\mathbf{P}_{yy,k|k-1}\mathbf{K}_k^{\mathsf{T}}.
\end{aligned}$$

3.5.3 Spherical-Radial Cubature Kalman Smoother

The cubature Rauch–Tung–Striebel smoother (CRTS) algorithm (see **?**) is presented below. Assume the filtering result mean $\mathbf{m}_{k|k}$ and covariance $\mathbf{P}_{k|k}$ are known together with the smoothing result $p(\mathbf{x}_{k+1} \mid \mathbf{y}_{1:T}) = \mathrm{N}(\mathbf{m}_{k+1|T}, \mathbf{P}_{k+1|T})$.

1. Draw cubature points $\boldsymbol{\xi}_i, i = 1, \ldots, 2n$ from the intersections of the n-dimensional unit sphere and the Cartesian axes. Scale them by $\sqrt{n}$. That is

$$\boldsymbol{\xi}_i = \begin{cases} \sqrt{n}\, e_i & , \, i = 1, \ldots, n \\ -\sqrt{n}\, e_{i-n} & , \, i = n+1, \ldots, 2n \end{cases}$$

2. Propagate the cubature points

$$\mathbf{X}_{i,k|k} = \sqrt{\mathbf{P}_{k|k}}\boldsymbol{\xi}_i + \mathbf{m}_{k|k}.$$

3. Evaluate the cubature points with the dynamic model function

$$\mathbf{X}^*_{i,k+1|k} = \mathbf{f}(\mathbf{X}_{i,k|k}).$$

4. Estimate the predicted state mean

$$\mathbf{m}_{k+1|k} = \frac{1}{2n}\sum_{i=1}^{2n} \mathbf{X}^*_{i,k+1|k}.$$

5. Estimate the predicted error covariance

$$\mathbf{P}_{k+1|k} = \frac{1}{2n}\sum_{i=1}^{2n} \mathbf{X}^*_{i,k+1|k}\mathbf{X}^{*\mathsf{T}}_{i,k+1|k} - \mathbf{m}_{k+1|k}\mathbf{m}_{k+1|k}^{\mathsf{T}} + \mathbf{Q}_k.$$

6. Estimate the cross-covariance matrix

$$\mathbf{D}_{k,k+1} = \frac{1}{2n}\sum_{i=1}^{2n}\left(\mathbf{X}_{i,k|k} - \mathbf{m}_{k|k}\right)\left(\mathbf{X}^*_{i,k+1|k} - \mathbf{m}_{k+1|k}\right)^{\mathsf{T}}.$$

7. Calculate the gain term and the smoothed state mean and covariance

$$\begin{aligned}\mathbf{C}_k &= \mathbf{D}_{k,k+1}\mathbf{P}^{-1}_{k+1|k}\\ \mathbf{m}_{k|T} &= \mathbf{m}_{k|k} + \mathbf{C}_k(\mathbf{m}_{k+1|T} - \mathbf{m}_{k+1|k})\\ \mathbf{P}_{k|T} &= \mathbf{P}_{k|k} + \mathbf{C}_k(\mathbf{P}_{k+1|T} - \mathbf{P}_{k+1|k})\mathbf{C}_k{}^{\mathsf{T}}.\end{aligned}$$

3.6 Demonstration: UNGM-model

To illustrate some of the advantages of UKF over EKF and augmented form UKF over non-augmented lets now consider an example, in which we estimate a model called Univariate Nonstationary Growth Model (UNGM), which is previously used as benchmark, for example, in **?** and **?**. What makes this model particularly interesting in this case is that its highly nonlinear and bimodal, so it is really challenging for traditional filtering techniques. We also show how in this case the augmented version of UKF gives better performance than the nonaugmented version. Additionally the Cubature (CKF) and Gauss–Hermite Kalman filter (GHKF) results are also provided for comparison.

The dynamic state space model for UNGM can be written as

$$x_n = \alpha x_{n-1} + \beta\frac{x_{n-1}}{1+x^2_{n-1}} + \gamma\cos(1.2(n-1)) + u_n \tag{3.75}$$

$$y_n = \frac{x^2_n}{20} + v_n, n = 1,\ldots,N \tag{3.76}$$

where $u_n \sim N(0,\sigma^2_u)$ and $v_n \sim N(0,\sigma^2_u)$. In this example we have set the parameters to $\sigma^2_u = 1$, $\sigma^2_v = 1$, $x_0 = 0.1$, $\alpha = 0.5$, $\beta = 25$, $\gamma = 8$, and $N = 500$. The cosine term in the state transition equation simulates the effect of time-varying noise.

In this demonstration the state transition is computed with the following function:

```
function x_n = ungm_f(x,param)
  n = param(1);
  x_n = 0.5*x(1,:) + 25*x(1,:)./(1+x(1,:).*x(1,:)) + 8*cos(1.2*(n-1));
  if size(x,1) > 1 x_n = x_n + x(2,:); end
```

where the input parameter `x` contains the state on the previous time step. The current time step index n needed by the cosine term is passed in the input parameter `param`. The last three lines in the function adds the process noise to the state

component, if the augmented version of the UKF is used. Note that in augmented UKF the state, process noise and measurement noise terms are all concatenated together to the augmented variable, but in URTS the measurement noise term is left out. That is why we must make sure that the functions we declare are compatible with all cases (nonaugmented, augmented with and without measurement noise). In this case we check whether the state variable has second component (process noise) or not.

Similarly, the measurement model function is declared as

```
function y_n = ungm_h(x_n,param)
  y_n = x_n(1,:).*x_n(1,:) ./ 20;
  if size(x_n,1) == 3 y_n = y_n + x_n(3,:); end
```

The filtering loop for augmented UKF is as follows:

```
  for k = 1:size(Y,2)
    [M,P,X_s,w] = ukf_predict3(M,P,f_func,u_n,v_n,k);
    [M,P] = ukf_update3(M,P,Y(:,k),h_func,v_n,X_s,w,[]);
    MM_UKF2(:,k) = M;
    PP_UKF2(:,:,k) = P;
  end
```

The biggest difference in this in relation to other filters is that now the predict step returns the sigma points (variable `X_s`) and their weigths (variable `w`), which must be passed as parameters to update function.

To compare the EKF and UKF to other possible filtering techniques we have also used a bootstrap filtering approach (**?**), which belongs to class of *Sequential Monte Carlo* (SMC) methods (also known as *particle filters*). Basically the idea in SMC methods that is during each time step they draw a set of weighted particles from some appropriate importance distribution and after that the moments (that is, mean and covariance) of the function of interest (e.g. dynamic function in state space models) are estimated approximately from drawn samples. The weights of the particles are adjusted so that they are approximations to the relative posterior probabilities of the particles. Usually also a some kind of *resampling* scheme is used to avoid the problem of degenerate particles, that is, particles with near zero weights are removed and those with large weights are duplicated. In this example we used a *stratified resampling* algorithm (**?**), which is optimal in terms of variance. In bootstrap filtering the dynamic model $p(\mathbf{x}_k|\mathbf{x}_{k-1})$ is used as importance distribution, so its implementation is really easy. However, due to this a large number of particles might be needed for the filter to be effective. In this case 1000 particles were drawn on each step. The implementation of the bootstrap filter is commented out in the actual demonstration script (`ungm_demo.m`), because the used resampling function (`resampstr.m`) was originally provided in MCMCstuff toolbox (**?**), which can be found at http://www.lce.hut.fi/research/mm/mcmcstuff/.

In figure 7 we have plotted the 100 first samples of the signal as well as the estimates produced by EKF, augmented form UKF and bootstrap filter. The bi-

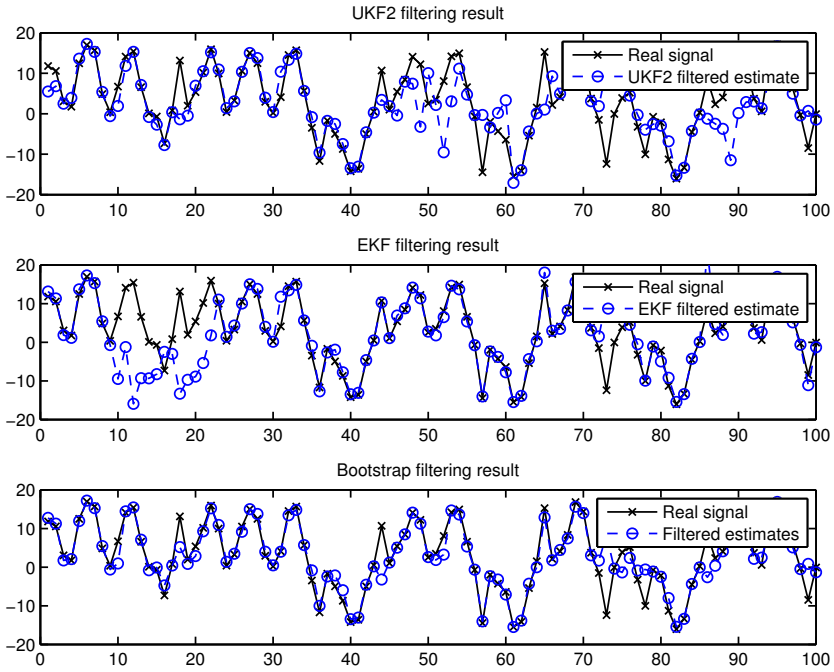


Figure 3.4: First 100 samples of filtering results of EKF, augmented form UKF and bootstrap filter for UNGM-model.

modality is easy to see from the figure. For example, during samples $10 - 25$ UKF is able to estimate the correct mode while the EKF estimates it wrong. Likewise, during steps $45 - 55$ and $85 - 95$ UKF has troubles in following the correct mode while EKF is more right. Bootstrap filter on the other hand tracks the correct mode on almost ever time step, although also it produces notable errors.

In figure 8 we have plotted the absolute errors and 3σ confidence intervals of the previous figures filtering results. It can be seen that the EKF is overoptimistic in many cases while UKF and boostrap filter are better at telling when their results are unreliable. Also the lower error of bootstrap filter can be seen from the figure. The bimodality is also easy to notice on those samples, which were mentioned above.

The make a comparison between nonaugmented and augmented UKF we have plotted 100 first samples of their filtering results in figure 9. Results are very surprising (although same as in **?**). The reason why nonaugmented UKF gave so bad results is not clear. However, the better performance of augmented form UKF can be explained by the fact, that the process noise is taken into account more effectively when the sigma points are propagated through nonlinearity. In this case it seems to be very crucial, as the model is highly nonlinear and multi-modal.

Lastly in figure 10 we have plotted the mean square errors of each tested methods of 100 Monte Carlo runs. Average of those errors are listed in table 2. Here is a discussion for the results:

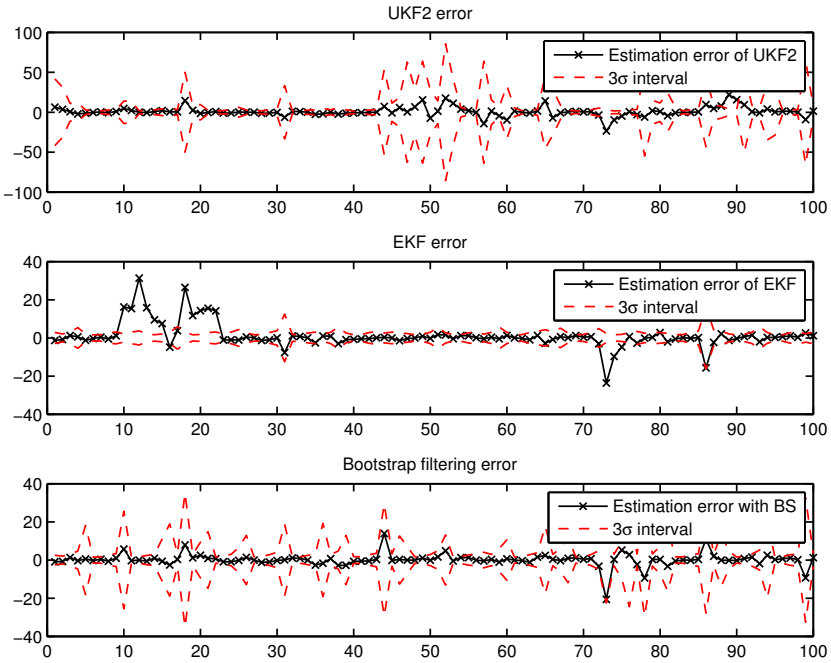


Figure 3.5: Absolute errors of and 3σ confidence intervals of EKF, augmented form UKF and bootstrap in 100 first samples.

- It is surprising that the nonaugmented UKF seems to be better than EKF, while in above figures we have shown, that the nonaugmented UKF gives very bad results. Reason for this is simple: the variance of the actual signal is approximately 100, which means that by simply guessing zero we get better performance than with EKF, on average. The estimates of nonaugmented UKF didn't variate much on average, so they were better than those of EKF, which on the other hand variated greatly and gave huge errors in some cases. Because of this neither of the methods should be used for this problem, but if one has to choose between the two, that would be EKF, because in some cases it still gave (more or less) right answers, whereas UKF were practically always wrong.

- The second order EKF were also tested, but that diverged almost instantly, so it were left out from comparison.

- Augmented form UKF gave clearly the best performance from the tested Kalman filters. As discussed above, this is most likely due to the fact that the process noise terms are propagated through the nonlinearity, and hence odd-order moment information is used to obtain more accurate estimates. The usage of RTS smoother seemed to improve the estimates in general, but oddly in some cases it made the estimates worse. This is most likely due to

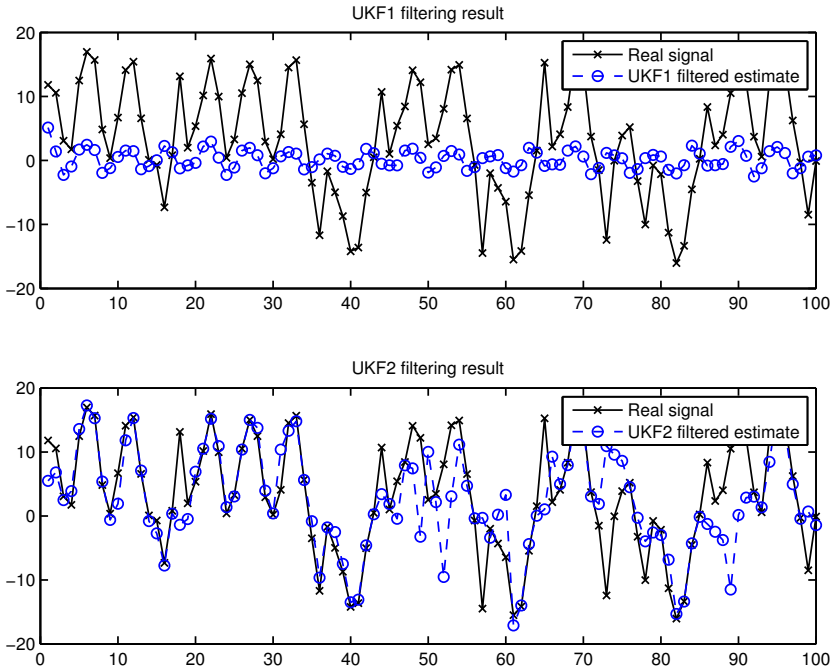


Figure 3.6: Filtering results of nonaugmented UKF (UKF1) and augmented UKF (UKF2) of 100 first samples.

Method	*MSE[x]*
UKF1	87.9
URTS1	69.09
UKF2	63.7
URTS2	57.7
EKF	125.9
ERTS	92.2
BS	10.2
GHKF	40.9
GHRTS	31.6
CKF	72.3
CRTS	71.4

Table 3.2: MSEs of estimating the UNGM model over 100 Monte Carlo simulations.

the multi-modality of the filtering problem.

- The Gauss–Hermite method performed rather well in both filtering and smoothing. This was mostly due to the degree of approximation as the rule entailed 10 sigma points.
- The cubature Kalman filter gave results close to the UKF variants, which is to no surprise as the filter uses similar sigma points.
- Bootstrap filtering solution was clearly superior over all other tested methods. The results had been even better, if greater amount of particles had been used.

The reason why Kalman filters didn't work that well in this case is because Gaussian approximations do not in general apply well for multi-modal cases. Thus, a particle filtering solution should be preferred over Kalman filters in such cases. However, usually the particle filters need a fairly large amount of particles to be effective, so they are generally more demanding in terms of computational power than Kalman filters, which can be a limiting factor in real world applications. The errors, even with bootstrap filter, were also relatively large, so one must be careful when using the estimates in, for example, making financial decisions. In practice this means that one has to monitor the filter's covariance estimate, and trust the state estimates and predictions only when the covariance estimates are low enough, but even then there is a change, that the filter's estimate is completely wrong.

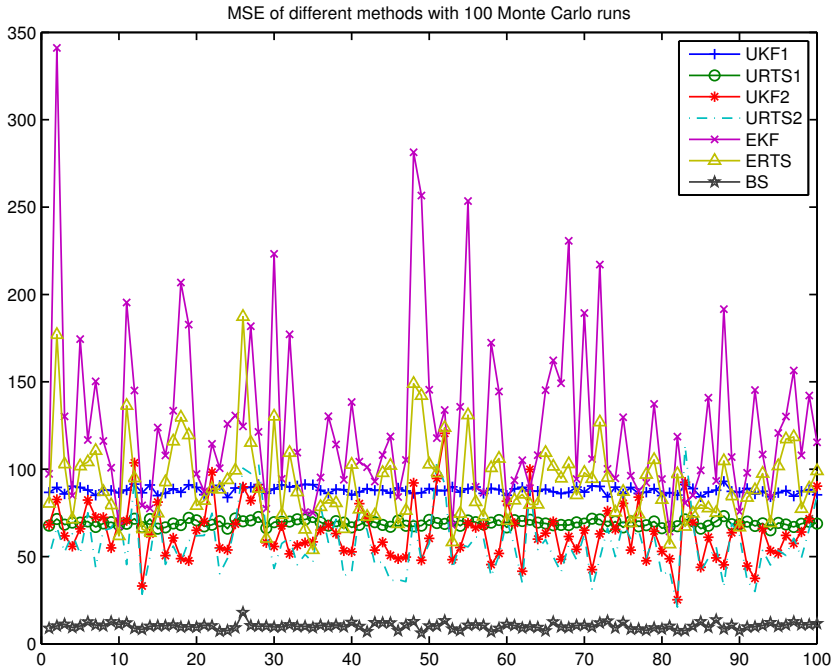


Figure 3.7: MSEs of different methods in 100 Monte Carlo runs.

3.7 Demonstration: Bearings Only Tracking

Next we review a classical filtering application (see, e.g., **?**), in which we track a moving object with sensors, which measure only the bearings (or angles) of the object with respect positions of the sensors. There is a one moving target in the scene and two angular sensors for tracking it. Solving this problem is important, because often more general multiple target tracking problems can be partitioned into sub-problems, in which single targets are tracked separately at a time (**?**).

The state of the target at time step k consists of the position in two dimensional cartesian coordinates x_k and y_k and the velocity toward those coordinate axes, $\dot{x}_k$ and $\dot{y}_k$. Thus, the state vector can be expressed as

$$\mathbf{x}_k = \begin{pmatrix} x_k & y_k & \dot{x}_k & \dot{y}_k \end{pmatrix}^T. \tag{3.77}$$

The dynamics of the target is modelled as a linear, discretized Wiener velocity model (**?**)

$$\mathbf{x}_k = \begin{pmatrix} 1 & 0 & \Delta\mathrm{t} & 0 \\ 0 & 1 & 0 & \Delta\mathrm{t} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{k-1} \\ y_{k-1} \\ \dot{x}_{k-1} \\ \dot{y}_{k-1} \end{pmatrix} + \mathbf{q}_{k-1}, \tag{3.78}$$

where $\mathbf{q}_{k-1}$ is Gaussian process noise with zero mean and covariance

$$E[\mathbf{q}_{k-1}\mathbf{q}_{k-1}^T] = \begin{pmatrix} \frac{1}{3}\Delta\mathrm{t}^3 & 0 & \frac{1}{2}\Delta\mathrm{t}^2 & 0 \\ 0 & \frac{1}{3}\Delta\mathrm{t}^3 & 0 & \frac{1}{2}\Delta\mathrm{t}^2 \\ \frac{1}{2}\Delta\mathrm{t}^2 & 0 & \Delta\mathrm{t} & 0 \\ 0 & \frac{1}{2}\Delta\mathrm{t}^2 & 0 & \Delta\mathrm{t} \end{pmatrix} q, \tag{3.79}$$

where q is the spectral density of the noise, which is set to $q = 0.1$ in the simulations. The measurement model for sensor i is defined as

$$\theta_k^i = \arctan\left(\frac{y_k - s_y^i}{x_k - s_x^i}\right) + r_k^i, \tag{3.80}$$

where (s_x^i, s_y^i) is the position of sensor i and $r_k^i \sim N(0, \sigma^2)$, with $\sigma = 0.05$ radians. In figure 11 we have plotted a one realization of measurements in radians obtained from both sensors. The sensors are placed to $(s_x^1, s_y^1) = (-1, -2)$ and $(s_x^2, s_y^2) = (1, 1)$.

The derivatives of the measurement model, which are needed by EKF, can be

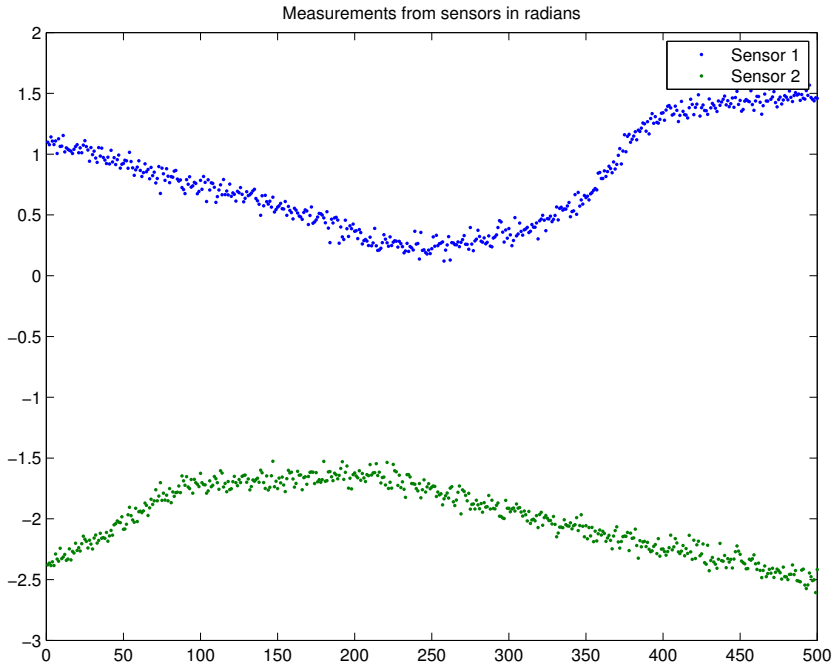


Figure 3.8: Measurements from sensors (in radians) in bearings only tracking problem .

computed as

$$\begin{aligned}
\frac{\partial \mathbf{h}^i(\mathbf{x}_k)}{\partial x_k} &= \frac{-(y_k - s_y^i)}{(x_k - s_x^i)^2 + (y_k - s_y^i)^2} \\
\frac{\partial \mathbf{h}^i(\mathbf{x}_k)}{\partial y_k} &= \frac{(x_k - s_x^i)}{(x_k - s_x^i)^2 + (y_k - s_y^i)^2} \\
\frac{\partial \mathbf{h}^i(\mathbf{x}_k)}{\partial \dot{x}_k} &= 0 \\
\frac{\partial \mathbf{h}^i(\mathbf{x}_k)}{\partial \dot{y}_k} &= 0, i = 1, 2.
\end{aligned} \tag{3.81}$$

With these the Jacobian can written as

$$\mathbf{H}_{\mathbf{x}}(\mathbf{x}_k, k) = \begin{pmatrix} \frac{(x_k - s_x^1)}{(x_k - s_x^1)^2 + (y_k - s_y^1)^2} & \frac{-(y_k - s_y^1)}{(x_k - s_x^1)^2 + (y_k - s_y^1)^2} & 0 & 0 \\ \frac{(x_k - s_x^2)}{(x_k - s_x^2)^2 + (y_k - s_y^2)^2} & \frac{-(y_k - s_y^2)}{(x_k - s_x^2)^2 + (y_k - s_y^2)^2} & 0 & 0 \end{pmatrix}. \tag{3.82}$$

The non-zero second order derivatives of the measurement function are also rela-

tively easy to compute in this model:

$$\begin{aligned}
\frac{\partial^2 \mathbf{h}^i(\mathbf{x}_k)}{\partial x_k \partial x_k} &= \frac{-2(x_k - s_x^i)}{((x_k - s_x^i)^2 + (y_k - s_y^i)^2)^2} \\
\frac{\partial^2 \mathbf{h}^i(\mathbf{x}_k)}{\partial x_k \partial y_k} &= \frac{(y_k - s_y^i)^2 - (x_k - s_x^i)^2}{((x_k - s_x^i)^2 + (y_k - s_y^i)^2)^2} \\
\frac{\partial^2 \mathbf{h}^i(\mathbf{x}_k)}{\partial y_k \partial y_k} &= \frac{-2(y_k - s_y^i)}{((x_k - s_x^i)^2 + (y_k - s_y^i)^2)^2}.
\end{aligned} \tag{3.83}$$

Thus, the Hessian matrices can be written as

$$\mathbf{H}_{\mathbf{xx}}^i(\mathbf{x}_k, k) = \begin{pmatrix}
\frac{-2(x_k - s_x^i)}{((x_k - s_x^i)^2 + (y_k - s_y^i)^2)^2} & \frac{(y_k - s_y^i)^2 - (x_k - s_x^i)^2}{((x_k - s_x^i)^2 + (y_k - s_y^i)^2)^2} & 0 & 0 \\
\frac{(y_k - s_y^i)^2 - (x_k - s_x^i)^2}{((x_k - s_x^i)^2 + (y_k - s_y^i)^2)^2} & \frac{-2(y_k - s_y^i)}{((x_k - s_x^i)^2 + (y_k - s_y^i)^2)^2} & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{pmatrix}, i = 1, 2. \tag{3.84}$$

We do not list the program code for the measurement function and it's derivatives here as they are straightforward to implement, if the previous examples have been read.

The target starts with state $\mathbf{x}_0 = \begin{pmatrix} 0 & 0 & 1 & 0 \end{pmatrix}$, and in the estimation we set the prior distribution for the state to $\mathbf{x}_0 \sim N(\mathbf{0}, \mathbf{P}_0)$, where

$$\mathbf{P}_0 = \begin{pmatrix}
0.1 & 0 & 0 & 0 \\
0 & 0.1 & 0 & 0 \\
0 & 0 & 10 & 0 \\
0 & 0 & 0 & 10
\end{pmatrix}, \tag{3.85}$$

which basically means that we are fairly certain about the target's origin, but very uncertain about the velocity. In the simulations we also give the target an slightly randomized acceleration, so that it achieves a curved trajectory, which is approximately the same in different simulations. The trajectory and estimates of it can be seen in figures 12, 13 and 14. As can be seen from the figures EKF1 and UKF give almost identical results while the estimates of EKF2 are clearly worse. Especially in the beginning of the trajectory EKF2 has great difficulties in getting on the right track, which is due to the relatively big uncertainty in the starting velocity. After that the estimates are fairly similar.

In table 3 we have listed the root mean square errors (mean of position errors) of all tested methods (same as in random sine signal example on page 25 with the addition of UTF) over 1000 Monte Carlo runs. The numbers prove the previous observations, that the EKF1 and UKF give almost identical performances. Same observations apply also to smoothers. Had the prior distribution for the starting velocity been more accurate the performance difference between EKF2 and other methods would have been smaller, but still noticeable.

Method	*RMSE*
EKF1	0.114
ERTS1	0.054
ETF1	0.054
EKF2	0.202
ERTS2	0.074
ETF2	0.063
UKF	0.113
URTS	0.055
UTF	0.055
GHKF	0.107
GHRTS	0.053
CKF	0.108
CRTS	0.053

Table 3.3: RMSEs of estimating the position in Bearings Only Tracking problem over 1000 Monte Carlo runs. The Gauss–Hermite rule is of degree 3.

These observations also hold for the cubature methods. The Gauss–Hermite Kalman filter and cubature Kalman filter give practically identical results as the unscented filter.

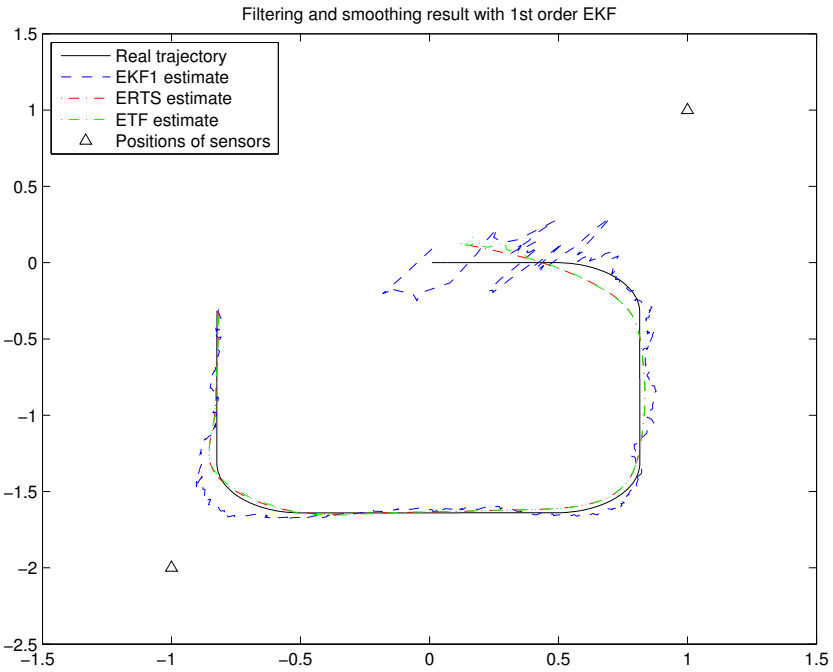


Figure 3.9: Filtering and smoothing results of first order EKF.

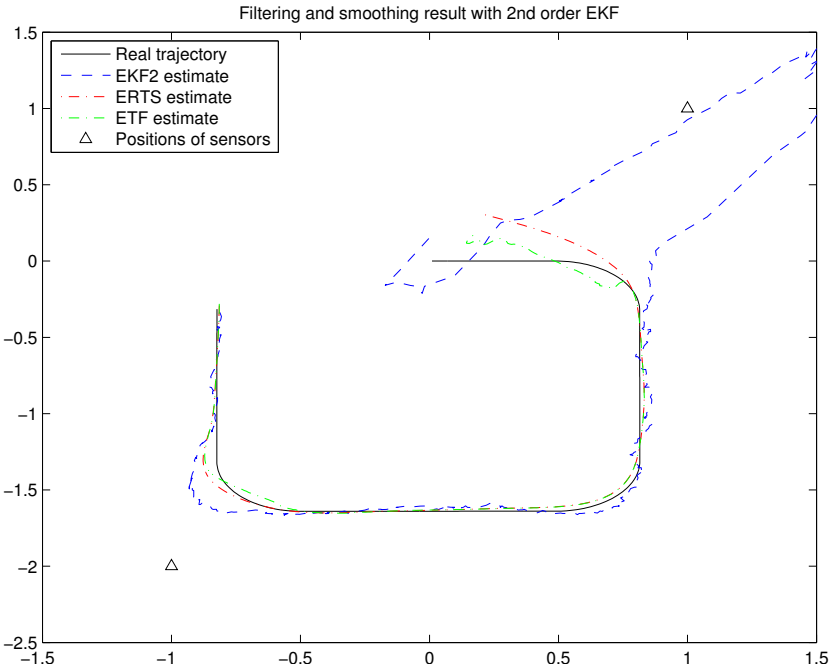


Figure 3.10: Filtering and smoothing results of second order EKF.

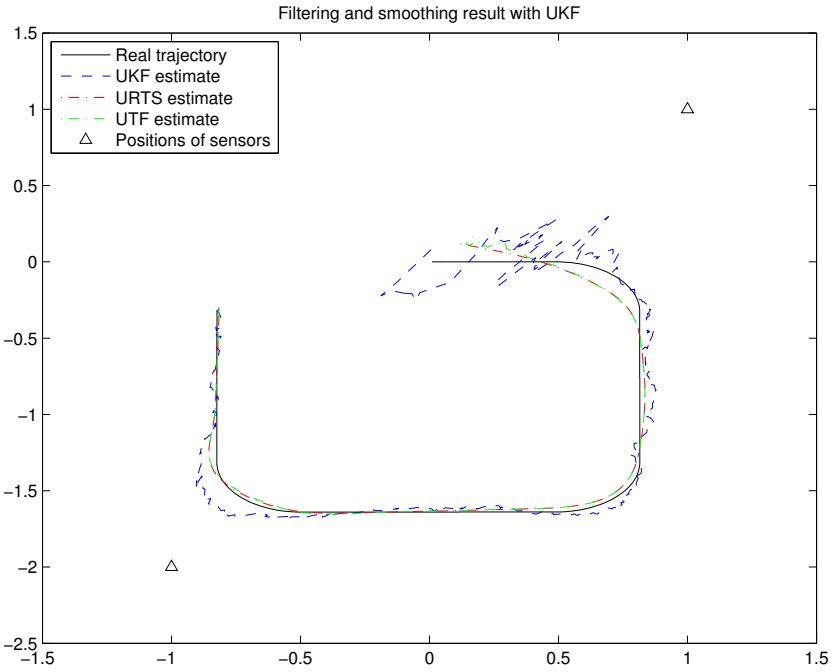


Figure 3.11: Filtering and smoothing results of UKF

3.8 Demonstration: Reentry Vehicle Tracking

Next we review a challenging filtering problem, which was used in **?** to demonstrate the performance of UKF. Later they released few corrections to the model specifications and simulation parameters in **?**.

This example conserns a reentry tracking problem, where radar is used for tracking a space vehicle, which enters the atmosphere at high altitude and very high speed. Figure 15 shows a sample trajectory of the vehicle with respect to earth and radar. The dynamics of the vehicle are affected with three kinds of forces: aerodynamic drag, which is a function of vehicle speed and has highly nonlinear variations in altitude. The second type of force is gravity, which causes the vehicle to accelerate toward the center of the earth. The third type of forces are random buffeting terms. The state space in this model consists of vehicles position (x_1 and x_2), its velocity (x_3 and x_4) and a parameter of its aerodynamic properties (x_5). The dynamics in continuous case are defined as (**?**)

$$\begin{aligned}
\dot{x}_1(t) &= x_3(t) \\
\dot{x}_2(t) &= x_4(t) \\
\dot{x}_3(t) &= D(t)x_3(t) + G(t)x_1(t) + v_1(t) \\
\dot{x}_4(t) &= D(t)x_4(t) + G(t)x_2(t) + v_2(t) \\
\dot{x}_5(t) &= v_3(t),
\end{aligned} \tag{3.86}$$

where $\mathbf{w}(t)$ is the process noise vector, $D(t)$ the drag-related force and $G(t)$ the gravity-related force. The force terms are given by

$$\begin{aligned}
D(k) &= \beta(t) \exp\left\{ \frac{[R_0 - R(t)]}{H_0} \right\} V(t) \\
G(t) &= -\frac{Gm_0}{R^3(t)} \\
\beta(t) &= \beta_0 \exp x_5(t),
\end{aligned} \tag{3.87}$$

where $R(t) = \sqrt{x_1^2(t) + x_2^2(t)}$ is the vehicle's distance from the center of the earth and $V(t) = \sqrt{x_3^2(t) + x_4^2(t)}$ is the speed of the vehicle. The constants in previous definition were set to

$$\begin{aligned}
\beta_0 &= -0.59783 \\
H_0 &= 13.406 \\
Gm_0 &= 3.9860 \times 10^5 \\
R_0 &= 6374.
\end{aligned} \tag{3.88}$$

To keep the implementation simple the continuous-time dynamic equations were

discretized using a simple Euler integration scheme, to give

$$\begin{aligned}
x_1(k+1) &= x_1(k) + \Delta\mathrm{t}\, x_3(k) \\
x_2(k+1) &= x_2(k) + \Delta\mathrm{t}\, x_4(k) \\
x_3(k+1) &= x_3(k) + \Delta\mathrm{t}(D(k)x_3(k) + G(k)x_1(k)) + w_1(k) \\
x_4(k+1) &= x_4(k) + \Delta\mathrm{t}(D(k)x_4(k) + G(k)x_2(k)) + w_2(k) \\
x_5(k+1) &= x_5(k) + w_3(k),
\end{aligned} \tag{3.89}$$

where the step size between time steps was set to $\Delta\mathrm{t} = 0.1s$. Note that this might be too simple approach in real world applications due to high nonlinearities in the dynamics, so more advanced integration scheme (such as Runge-Kutta) might be more preferable. The discrete process noise covariance in the simulations was set to

$$Q(k) = \begin{pmatrix} 2.4064 \times 10^{-5} & 0 & 0 \\ 0 & 2.4064 \times 10^{-5} & 0 \\ 0 & 0 & 10^{-6} \end{pmatrix}. \tag{3.90}$$

The lower right element in the covariance was initially in **?** set to zero, but later in **?** changed to 10^{-6} to increase filter stability.

The non-zero derivatives of the discretized dynamic equations with respect to

state variables are straightforward (although rather technical) to compute:

$$
\begin{aligned}
&\frac{\partial x_1(k+1)}{\partial x_1(k)} = 1 \\
&\frac{\partial x_1(k+1)}{\partial x_3(k)} = \Delta \mathrm{t} \\
&\frac{\partial x_2(k+1)}{\partial x_2(k)} = 1 \\
&\frac{\partial x_2(k+1)}{\partial x_4(k)} = \Delta \mathrm{t} \\
&\frac{\partial x_3(k+1)}{\partial x_1(k)} = \Delta \mathrm{t} * (\frac{\partial D(k)}{\partial x_1(k)} x_3(k) + \frac{\partial G(k)}{\partial x_1(k)} x_1(k) + G(k)) \\
&\frac{\partial x_3(k+1)}{\partial x_2(k)} = \Delta \mathrm{t} * (\frac{\partial D(k)}{\partial x_2(k)} x_3(k) + \frac{\partial G(k)}{\partial x_2(k)} x_1(k)) \\
&\frac{\partial x_3(k+1)}{\partial x_3(k)} = \Delta \mathrm{t} * (\frac{\partial D(k)}{\partial x_3(k)} x_3(k) + D(k)) + 1 \\
&\frac{\partial x_3(k+1)}{\partial x_4(k)} = \Delta \mathrm{t} * (\frac{\partial D(k)}{\partial x_4(k)} x_3(k)) \\
&\frac{\partial x_3(k+1)}{\partial x_4(k)} = \Delta \mathrm{t} * (\frac{\partial D(k)}{\partial x_5(k)} x_3(k)) \\
&\frac{\partial x_4(k+1)}{\partial x_1(k)} = \Delta \mathrm{t} * (\frac{\partial D(k)}{\partial x_1(k)} x_4(k) + \frac{\partial G(k)}{\partial x_1(k)} x_2(k)) \\
&\frac{\partial x_4(k+1)}{\partial x_2(k)} = \Delta \mathrm{t} * (\frac{\partial D(k)}{\partial x_2(k)} x_4(k) + \frac{\partial G(k)}{\partial x_2(k)} x_2(k) + G(k)) \\
&\frac{\partial x_4(k+1)}{\partial x_3(k)} = \Delta \mathrm{t} * (\frac{\partial D(k)}{\partial x_3(k)} x_4(k)) \\
&\frac{\partial x_4(k+1)}{\partial x_4(k)} = \Delta \mathrm{t} * (\frac{\partial D(k)}{\partial x_4(k)} x_4(k) + D(k)) + 1 \\
&\frac{\partial x_4(k+1)}{\partial x_5(k)} = \Delta \mathrm{t} * (\frac{\partial D(k)}{\partial x_5(k)} x_4(k)) \\
&\frac{\partial x_5(k+1)}{\partial x_5(k)} = 1,
\end{aligned}
\tag{3.91}
$$

where the (non-zero) derivatives of the force, position and velocity related terms

with respect to state variables can be computed as

$$\begin{aligned}
\frac{\partial R(k)}{\partial x_1(k)} &= x_1(k)\frac{1}{R(k)} \\
\frac{\partial R(k)}{\partial x_2(k)} &= x_2(k)\frac{1}{R(k)} \\
\frac{\partial V(k)}{\partial x_3(k)} &= x_3(k)\frac{1}{V(k)} \\
\frac{\partial V(k)}{\partial x_4(k)} &= x_4(k)\frac{1}{V(k)} \\
\frac{\partial \beta(k)}{\partial x_5(k)} &= \beta(k)\frac{1}{R(k)} \\
\frac{\partial D(k)}{\partial x_1(k)} &= -\frac{\partial R(k)}{\partial x_1(k)}\frac{1}{H_0} * D(k) \\
\frac{\partial D(k)}{\partial x_2(k)} &= -\frac{\partial R(k)}{\partial x_2(k)}\frac{1}{H_0} * D(k) \\
\frac{\partial D(k)}{\partial x_3(k)} &= \beta(k)\exp\left\{\frac{[R_0 - R(k)]}{H_0}\right\}\frac{\partial V(k)}{\partial x_3} \\
\frac{\partial D(k)}{\partial x_4(k)} &= \beta(k)\exp\left\{\frac{[R_0 - R(k)]}{H_0}\right\}\frac{\partial V(k)}{\partial x_4} \\
\frac{\partial D(k)}{\partial x_5(k)} &= \frac{\partial \beta(k)}{x_5(k)}\exp\left\{\frac{[R_0 - R(k)]}{H_0}\right\}V(k) \\
\frac{\partial G(k)}{\partial x_1(k)} &= \frac{3Gm_0}{(R(k))^4}\frac{\partial R(k)}{\partial x_1(k)} \\
\frac{\partial G(k)}{\partial x_2(k)} &= \frac{3Gm_0}{(R(k))^4}\frac{\partial R(k)}{\partial x_2(k)}.
\end{aligned} \tag{3.92}$$

The prior distribution for the state was set to multivariate Gaussian, with mean and covariance (same as in **?**)

$$\begin{aligned}
\mathbf{m}_0 &= \begin{pmatrix} 6500.4 \\ 349.14 \\ -1.8093 \\ -6.7967 \\ 0 \end{pmatrix} \\
\mathbf{P}_0 &= \begin{pmatrix} 10^{-6} & 0 & 0 & 0 & 0 \\ 0 & 10^{-6} & 0 & 0 & 0 \\ 0 & 0 & 10^{-6} & 0 & 0 \\ 0 & 0 & 0 & 10^{-6} & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.
\end{aligned} \tag{3.93}$$

In the simulations the initial state were drawn from multivariate Gaussian with

mean and covariance

$$\begin{aligned}\mathbf{m}_0 &= \begin{pmatrix} 6500.4 \\ 349.14 \\ -1.8093 \\ -6.7967 \\ 0.6932 \end{pmatrix} \\ \mathbf{P}_0 &= \begin{pmatrix} 10^{-6} & 0 & 0 & 0 & 0 \\ 0 & 10^{-6} & 0 & 0 & 0 \\ 0 & 0 & 10^{-6} & 0 & 0 \\ 0 & 0 & 0 & 10^{-6} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix},\end{aligned} \tag{3.94}$$

that is, vehicle's aerodynamic properties were not known precisely beforehand.

The radar, which is located at $(s_x, s_y) = (R_0, 0)$, is used to measure the range r_k and bearing θ_k in relation to the vehicle on time step k. Thus, the measurement model can be written as

$$\begin{aligned} r_k &= \sqrt{(x_1(k) - s_x)^2 + (x_2(k) - s_y)^2} + q_1(k) \\ \theta_k &= \tan^{-1}\left(\frac{x_2(k) - s_y}{x_1(k) - s_x}\right) + q_2(k), \end{aligned} \tag{3.95}$$

where the measurement noise processes $q_1(k)$ and $q_2(k)$ are Gaussians with zero means and standard deviations $\sigma_r = 10^{-3}$km and $\sigma_\theta = 0.17$mrad, respectively. The derivatives of θ_k with respect to state variables can computed with equations (109). For r_k the derivatives can be written as

$$\begin{aligned} \frac{\partial r_k}{\partial x_1(k)} &= x_1(k)\frac{1}{r_k} \\ \frac{\partial r_k}{\partial x_2(k)} &= x_2(k)\frac{1}{r_k}. \end{aligned} \tag{3.96}$$

In the table 4 we have listed the RMS errors of position estimates with tested methods, which were

- EKF1: first order extended Kalman filter.
- ERTS: first order Rauch-Tung-Striebel smoother.
- UKF: augmented form unscented Kalman filter.
- URTS1: unscented Rauch-Tung-Striebel smoother with non-augmented sigma points.
- URTS2: unscented Rauch-Tung-Striebel smoother with augmented sigma points.

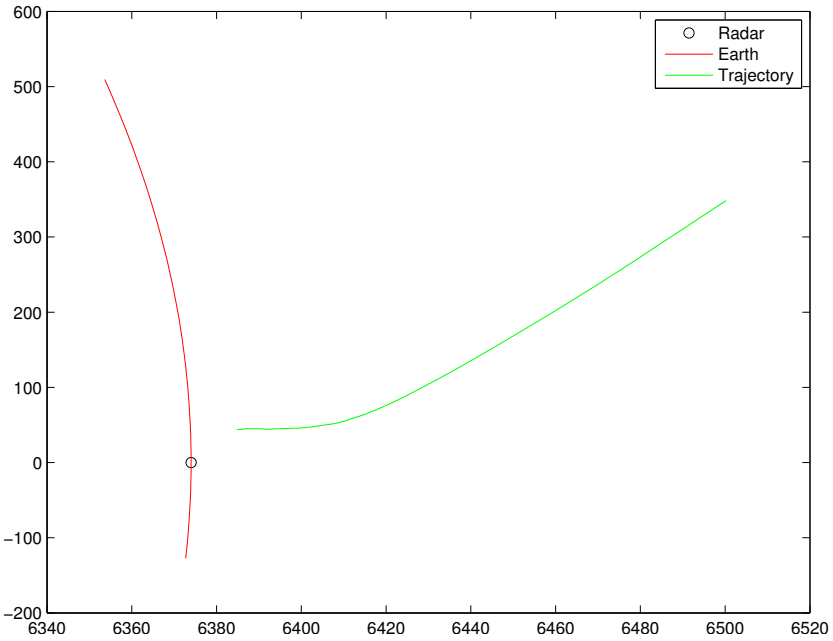


Figure 3.12: Sample vehicle trajectory, earth and position of radar in Reentry Vehicle Tracking problem.

- UTF: unscented Forward-Backward smoother.

Extended Forward-Backward smoother was also tested, but it produced in many cases divergent estimates, so it was left out from comparison. Second order EKF was also left out, because evaluating the Hessians would have taken too much work while considering the fact, that the estimates might have gotten even worse.

From the error estimates we can see, that EKF and UKF — and the other methods as well — give almost identical performances, altough in the article (**?**) UKF was clearly superior over EKF. The reason for this might be the fact that they used numerical approximations (central differences) for calculating the Jacobian in EKF rather than calculating the derivatives in closed form, as was done in this demonstration.

In figure 16 we have plotted the mean square errors and variances in estimating x_1, x_3 and x_5 with EKF and ERTS over 100 Monte Carlo runs. It shows that using smoother always gives better estimates for positions and velocities, but for x_5 the errors are practically the same after ~ 45 seconds. This also shows that both methods are pessimistic in estimating x_5, because variances were bigger than the true errors. Figures for x_2 and x_4 are not shown, because they are very similar to the ones of x_1 and x_3. Also by using UKF and URTS the resulting figures were in practically identical, and therefore left out.

Method	*RMSE*
EKF1	0.0084
ERTS	0.0044
UKF	0.0084
URTS1	0.0044
URTS2	0.0044
UTF	0.0044
GHKF	0.0084
GHRTS	0.0049
CKF	0.0084
CRTS	0.0049

Table 3.4: Average RMSEs of estimating the position in Reentry Vehicle Tracking problem over 100 Monte Carlo runs. The Gauss–Hermite rule is of degree 3.

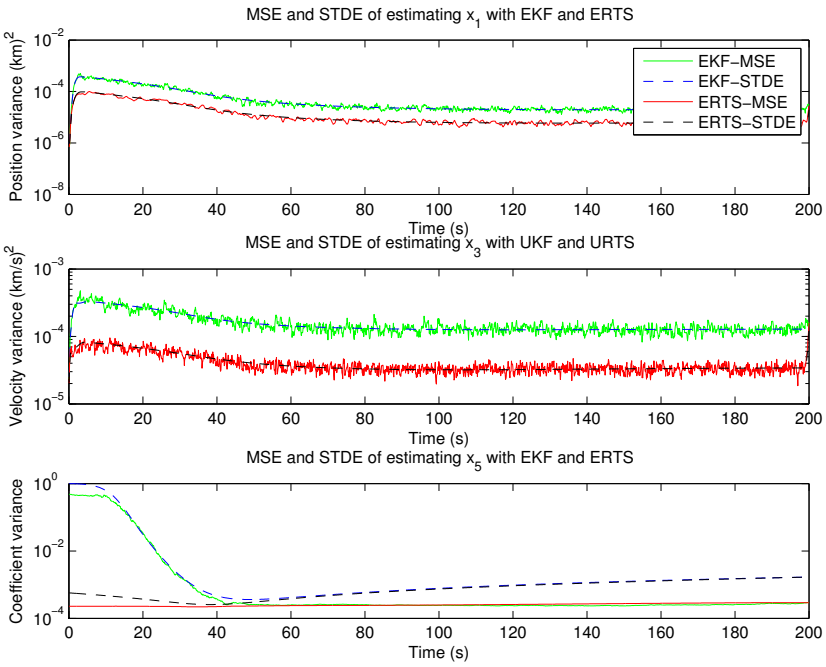


Figure 3.13: MSEs and variances in estimating of x_1, x_3 and x_5 using EKF and ERTS over 100 Monte Carlo runs.

Chapter 4

Multiple Model Systems

In many practical scenarios it is reasonable to assume that the the system's model can change through time somehow. For example, a fighter airplane, which in normal situation flies with stabile flight dynamics, might commence rapid maneuvers when approached by a hostile missile, or a radar can have a different SNR in some regions of space than in others, and so on. Such varying system characteristics are hard to describe with only a one certain model, so in estimation one should somehow take into account the possibility that the system's model might change.

We now consider systems, whose current model is one from a discrete set of n models, which are denoted by $M = \{M^1, \ldots, M^n\}$. We assume that for each model M^j we have some prior probability $\mu_0^j = P\{M_0^j\}$. Also the probabilities of switching from model i to model j in next time step are assumed to be known and denoted by $p_{ij} = P\{M_k^j | M_{k-1}^i\}$. This can be seen as a transition probability matrix of a first order Markov chain characterizing the mode transitions, and hence systems of this type are commonly referred as *Markovian switching systems*. The optimal approach to filtering the states of multiple model system of this type requires running optimal filters for every possible model sequences, that is, for n models n^k optimal filters must be ran to process the kth measurement. Hence, some kind of approximations are needed in practical applications of multiple model systems.

filtering problems is the Generalized Pseudo-Bayesian (GPB) algorithms (In this section we describe the Interacting Multiple Model (IMM) filter (**?**), which is a popular method for estimating systems, whose model changes according to a finite-state, discrete-time Markov chain. IMM filter can also be used in situations, in which the unknown system model structure or it's parameters are estimated from a set of candidate models, and hence it can be also used as a method for model comparison.

As previously we start with linear models, and after that we review the EKF and UKF based nonlinear extensions to the standard IMM-filter through demonstrating filtering problems.

4.1 Linear Systems

We can now modify the equations of linear systems described in (12) to have form

$$\begin{aligned} \mathbf{x}_k &= \mathbf{A}^j_{k-1}\,\mathbf{x}_{k-1} + \mathbf{q}^j_{k-1} \\ \mathbf{y}_k &= \mathbf{H}^j_k\,\mathbf{x}_k + \mathbf{r}^j_k, \end{aligned} \tag{4.1}$$

where now we have denoted by j the model (or mode) which is in effect during the time step $k-1$. Conditioned on the currently active mode we can use the classical Kalman filter (section 2.1.2) for estimating the state of the system on each time step. However, the active mode of the system is not usually known, so we must estimate it also.

4.1.1 Interacting Multiple Model Filter

IMM-filter (**?**) is a computationally efficient and in many cases well performing suboptimal estimation algorithm for Markovian switching systems of type described above. Basically it consists of three major steps: interaction (mixing), filtering and combination. In each time step we obtain the initial conditions for certain model-matched filter by mixing the state estimates produced by all filters from the previous time step under the assumption that this particular model is the right model at current time step. Then we perform standard Kalman filtering for each model, and after that we compute a weighted combination of updated state estimates produced by all the filters yielding a final estimate for the state and covariance of the Gaussian density in that particular time step. The weights are chosen according to the probabilities of the models, which are computed in filtering step of the algorithm.

The equations for each step are as follows:

- *Interaction:*

  The mixing probabilities $\mu_k^{i|j}$ for each model M^i and M^j are calculated as

$$\bar{c}_j = \sum_{i=1}^{n} p_{ij}\mu^i_{k-1}, \tag{4.2}$$

$$\mu_k^{i|j} = \frac{1}{\bar{c}_j} p_{ij}\mu^i_{k-1}, \tag{4.3}$$

  where μ^i_{k-1} is the probability of model M^i in the time step $k-1$ and $\bar{c}_j$ a normalization factor.

Now we can compute the mixed inputs (that is, means and covariances) for each filter as

$$\mathbf{m}_{k-1}^{0j} = \sum_{i=1}^{n} \mu_k^{i|j} \mathbf{m}_{k-1}^{i}, \tag{4.4}$$

$$\mathbf{P}_{k-1}^{0j} = \sum_{i=1}^{n} \mu_k^{i|j} \times \left\{ \mathbf{P}_{k-1}^{i} + \left[\mathbf{m}_{k-1}^{i} - \mathbf{m}_{k-1}^{0j} \right] \left[\mathbf{m}_{k-1}^{i} - \mathbf{m}_{k-1}^{0j} \right]^T \right\} \tag{4.5}$$

where $\mathbf{m}_{k-1}^{i}$ and $\mathbf{P}_{k-1}^{i}$ are the updated mean and covariance for model i at time step $k-1$.

- *Filtering:*

  Now, for each model M^i the filtering is done as

$$\left[\mathbf{m}_k^{-,i}, \mathbf{P}_k^{-,i} \right] = \mathrm{KF}_p(\mathbf{m}_{k-1}^{0j}, \mathbf{P}_{k-1}^{0j}, \mathbf{A}_{k-1}^{i}, \mathbf{Q}_{k-1}^{i}), \tag{4.6}$$

$$\left[\mathbf{m}_k^{i}, \mathbf{P}_k^{i} \right] = \mathrm{KF}_u(\mathbf{m}_k^{-,i}, \mathbf{P}_k^{-,i}, \mathbf{y}_k, \mathbf{H}_k^{i}, \mathbf{R}_k^{i}), \tag{4.7}$$

  where we have denoted the prediction and update steps (equations (18) and (19)) of the standard Kalman filter with $\mathrm{KF}_p(\cdot)$ and $\mathrm{KF}_u(\cdot)$, correspondingly. In addition to mean and covariance we also compute the likelihood of the measurement for each filter as

$$\Lambda_k^i = \mathrm{N}(\mathbf{v}_k^i; 0, \mathbf{S}_k^i), \tag{4.8}$$

  where $\mathbf{v}_k^i$ is the measurement residual and $\mathbf{S}_k^i$ it's covariance for model M^i in the KF update step.

  The probabilities of each model M^i at time step k are calculated as

$$c = \sum_{i=1}^{n} \Lambda_k^i \bar{c}_i, \tag{4.9}$$

$$\mu_k^i = \frac{1}{c} \Lambda_k^i \bar{c}_i, \tag{4.10}$$

  where c is a normalizing factor.

- *Combination:*

  In the final stage of the algorithm the combined estimate for the state mean and covariance are computed as

$$\mathbf{m}_k = \sum_{i=1}^{n} \mu_k^i \mathbf{m}_k^i, \tag{4.11}$$

$$\mathbf{P}_k = \sum_{i=1}^{n} \mu_k^i \times \left\{ \mathbf{P}_k^i \left[\mathbf{m}_k^i - \mathbf{m}_k \right] \left[\mathbf{m}_k^i - \mathbf{m}_k \right]^T \right\}. \tag{4.12}$$

In this toolbox prediction and updates steps of the IMM-filter can be computed with functions `imm_kf_predict` and `imm_kf_update`. For convience we have also provided function `imm_filter`, which combines the two previously mentioned steps.

4.1.2 Interacting Multiple Model Smoother

Likewise in the single model case also it is useful to smooth the state estimates of the IMM filter by using all the obtained measurements. Since the optimal fixed-interval smoothing with n models and N measurements requires running n^N smoothers we must resort to suboptimal approaches. One possibility (**?**) is to combine the estimates of two IMM filters, one running forwards and the other backwards in time. This approach is restricted to systems having invertible state dynamics (i.e. system for which the inverse of matrix $\mathbf{A}^j$ in (125) exists), which is always the case for discretized continuous-time systems.

First we shall review the equations for the IMM-filter running backwards in time, and then the actual smoothing equations combining the estimates of the two filters.

Backward-time IMM-filter

Our aim is now to compute the backward filtering density $p(\mathbf{x}_k|\mathbf{y}_{k:N})$ for each time step, which is expressed as a sum of model conditioned densities:

$$p(\mathbf{x}_k|\mathbf{y}_{k:N}) = \sum_{j=1}^{n} \mu_k^{b,j} p(\mathbf{x}_k^j|\mathbf{y}_{k:N}), \tag{4.13}$$

where $\mu_k^{b,j}$ is the backward-time filtered model probability of M_k^j. In the last time step N this is the same as the forward filter's model probability, that is, $\mu_N^{b,j} = \mu_N^j$. Assuming the model conditioned densities $p(\mathbf{x}_k^j|\mathbf{y}_{k:N})$ are Gaussians the backward density in (137) is a mixture of Gaussians, which is now going to be approximated with a single Gaussian via moment matching.

The model conditioned backward-filtering densities can be expressed as

$$p(\mathbf{x}_k^j|\mathbf{y}_{k:N}) = \frac{1}{c} p(\mathbf{y}_{k:N}|\mathbf{x}_k^j) p(\mathbf{x}_k^j|\mathbf{y}_{k+1:N}), \tag{4.14}$$

where c is some normalizing constant, $p(\mathbf{y}_{k:N}|\mathbf{x}_k^j)$ the model-conditioned measurement likelihood and $p(\mathbf{x}_k^j|\mathbf{y}_{k+1:N})$ is the model-conditioned density of the state given the future measurements. The latter density is expressed as

$$p(\mathbf{x}_k^j|\mathbf{y}_{k+1:N}) = \sum_{i=1}^{n} \mu_{k+1}^{b,i|j} p(\mathbf{x}_k^j|M_{k+1}^i, \mathbf{y}_{k+1:N}), \tag{4.15}$$

where $\mu_{k+1}^{b,i|j}$ is the conditional model probability computed as

$$\mu_{k+1}^{b,i|j} = P\{M_{k+1}^i | M_k^j, \mathbf{y}_{k+1:N}\} \tag{4.16}$$

$$= \frac{1}{a_j} p_{ij}^{b,k} \mu_{k+1}^{b,i}, \tag{4.17}$$

where a_j is a normalization constant given by

$$a_j = \sum_{i=1}^{n} p_{ij}^{b,k} \mu_{k+1}^{b,i}. \tag{4.18}$$

The backward-time transition probabilities of switching from model M_{k+1}^i to model M_k^j in (141) and (142) are defined as $p_{ij}^{b,k} = P\{M_k^j | M_{k+1}^i\}$. The prior model probabilities can be computed off-line recursively for each time step k as

$$P\{M_k^j\} = \sum_{i=1}^{n} P\{M_k^j | M_{k-1}^i\} P\{M_{k-1}^i\} \tag{4.19}$$

$$= \sum_{i=1}^{n} p_{ij} P\{M_{k-1}^i\} \tag{4.20}$$

and using these we can compute $p_{ij}^{b,k}$ as

$$p_{ij}^{b,k} = \frac{1}{b_i} p_{ji} P\{M_k^j\}, \tag{4.21}$$

where b_i is the normalizing constant

$$b_i = \sum_{j=1}^{n} p_{ji} P\{M_k^j\}. \tag{4.22}$$

The density $p(\mathbf{x}_k^j | M_{k+1}^i, \mathbf{y}_{k+1:N}^N)$ is now approximated with a Gaussian $N(\mathbf{x}_k | \mathbf{m}_{k|k+1}^{b,i}, \mathbf{P}_{k|k+1}^{b,-(i)})$, where the mean and covariance are given by the Kalman filter prediction step using the inverse of the state transition matrix:

$$\left[\hat{\mathbf{m}}_k^{b,i}, \hat{\mathbf{P}}_k^{b,i}\right] = \text{KF}_p(\mathbf{m}_{k+1}^{b,i}, \mathbf{P}_{k+1}^{b,i}, (\mathbf{A}_{k+1}^i)^{-1}, \mathbf{Q}_{k+1}^i). \tag{4.23}$$

The density $p(\mathbf{x}_k^j | \mathbf{y}_{k+1:N})$ in (139) is a mixture of Gaussians, and it's now approximated with a single Gaussian as

$$p(\mathbf{x}_k^j | \mathbf{y}_{k+1:N}) = N(\mathbf{x}_k^j | \hat{\mathbf{m}}_k^{b,0j}, \hat{\mathbf{P}}_k^{b,0j}), \tag{4.24}$$

where the mixed predicted mean and covariance are given as

$$\hat{\mathbf{m}}_k^{b,0j} = \sum_{i=1}^{n} \mu_{k+1}^{b,i|j} \hat{\mathbf{m}}_k^{b,i} \tag{4.25}$$

$$\hat{\mathbf{P}}_k^{b,0j} = \sum_{i=1}^{n} \mu_{k+1}^{b,i|j} \cdot \left[\hat{\mathbf{P}}_k^{b,i} + \left(\hat{\mathbf{m}}_k^{b,i} - \hat{\mathbf{m}}_k^{b,0j}\right)\left(\hat{\mathbf{m}}_k^{b,i} - \hat{\mathbf{m}}_k^{b,0j}\right)^T\right] \tag{4.26}$$

Now, the filtered density $p(\mathbf{x}_k^j|\mathbf{y}_{k:N})$ is a Gaussian $N(\mathbf{x}_k|\mathbf{m}_k^{b,j}, \mathbf{P}_k^{b,j})$, and solving it's mean and covariance corresponds to running the Kalman filter update step as follows:

$$\left[\mathbf{m}_k^{b,j}, \mathbf{P}_k^{b,j}\right] = \mathrm{KF}_u(\hat{\mathbf{m}}_k^{b,0j}, \hat{\mathbf{P}}_k^{b,0j}, \mathbf{y}_k, \mathbf{H}_k^j, \mathbf{R}_k^j). \tag{4.27}$$

The measurement likelihoods for each model are computed as

$$\Lambda_k^{b,i} = \mathrm{N}(\mathbf{v}_k^{b,i}; 0, \mathbf{S}_k^{b,i}), \tag{4.28}$$

where $\mathbf{v}_k^{b,i}$ is the measurement residual and $\mathbf{S}_k^{b,i}$ it's covariance for model M^i in the KF update step. With these we can update the model probabilities for time step k as

$$\mu_k^{b,j} = \frac{1}{a} a_j \Lambda_k^{b,i}, \tag{4.29}$$

where a is a normalizing constant

$$a = \sum_{j=1}^{m} a_j \Lambda_k^{b,i}. \tag{4.30}$$

Finally, we can form the Gaussian approximation to overall backward filtered distribution as

$$p(\mathbf{x}_k|\mathbf{y}_{k:N}) = N(\mathbf{x}_k|\mathbf{m}_k^b, \mathbf{P}_k^b), \tag{4.31}$$

where the mean and covariance are mixed as

$$\mathbf{m}_k^b = \sum_{j=1}^{n} \mu_k^{b,j} \mathbf{m}_k^{b,j} \tag{4.32}$$

$$\mathbf{P}_k^b = \sum_{j=1}^{n} \mu_k^{b,j} \left[\mathbf{P}_k^{b,j} + \left(\mathbf{m}_k^{b,j} - \mathbf{m}_k^b\right)\left(\mathbf{m}_k^{b,j} - \mathbf{m}_k^b\right)^T\right]. \tag{4.33}$$

4.1.3 Two-filter based fixed-interval IMM-Smoother

We can now proceed to evaluate the fixed-interval smoothing distribution

$$p(\mathbf{x}_k|\mathbf{y}_{1:N}) = \sum_{j=1}^{n} \mu_k^{s,j} p(\mathbf{x}_k^j|\mathbf{y}_{1:N}), \tag{4.34}$$

where the smoothed model probabilities are computed as

$$\mu_k^{s,j} = P\{M_k^j|\mathbf{y}_{1:N}\} \tag{4.35}$$

$$= \frac{1}{d} d_j \mu_k^j, \tag{4.36}$$

where μ_k^j is the forward-time filtered model probability, d_j the density $d_j = p(\mathbf{y}_{k+1:N}|M_k^j, \mathbf{y}_{1:k})$ (which is to be evaluated later) and d the normalization constant given by

$$d = \sum_{j=1}^{n} d_j \mu_k^j. \tag{4.37}$$

The model-conditioned smoothing distributions $p(\mathbf{x}_k^j|\mathbf{y}_{1:N})$ in (158) are expressed as a mixtures of Gaussians

$$p(\mathbf{x}_k^j|\mathbf{y}_{1:N}) = \sum_{i=1}^{n} \mu_{k+1}^{s,i|j} p(\mathbf{x}^i|M_{k+1}^j, \mathbf{y}_{1:n}), \tag{4.38}$$

where the conditional probability $\mu_{k+1}^{s,i|j}$ is given by

$$\begin{aligned} \mu_{k+1}^{s,i|j} &= P\{M_{k+1}^i|M_k^j, \mathbf{y}_{1:n}\} && (4.39)\\ &= \frac{1}{d_j} p_{ji} \Lambda_k^{ji} && (4.40) \end{aligned}$$

and the likelihood Λ_k^{ji} by

$$\Lambda_k^{ji} = p(\mathbf{y}_{k+1:N}|M_k^j, M_{k+1}^i, \mathbf{y}_{1:k}). \tag{4.41}$$

We approximate this now as

$$\Lambda_k^{ji} \approx p(\hat{\mathbf{x}}_k^{b,i}|M_k^j, M_{k+1}^i, \mathbf{x}_k^j), \tag{4.42}$$

which means that the future measurements $\mathbf{y}_{k+1:N}$ are replaced with the n model-conditioned backward-time one-step predicted means and covariances $\{\hat{\mathbf{m}}_k^{b,i}, \hat{\mathbf{P}}_k^{b,i}\}_{r=1}^n$, and $\mathbf{y}_{1:k}$ will be replaced by the n model-conditioned forward-time filtered means and covariances $\{\mathbf{m}_k^i|\mathbf{P}_k^i\}_{r=1}^n$. It follows then that the likelihoods can be evaluated as

$$\Lambda_k^{ji} = N(\Delta_k^{ji}|0, D_k^{ji}), \tag{4.43}$$

where

$$\begin{aligned} \Delta_k^{ji} &= \hat{\mathbf{m}}_k^{b,i} - \mathbf{m}_k^j && (4.44)\\ D_k^{ji} &= \hat{\mathbf{P}}_k^{b,i} + \mathbf{P}_k^j. && (4.45) \end{aligned}$$

The terms d_j can now be computed as

$$d_j = \sum_{i=1}^{n} p_{ji} \Lambda_k^{ji}. \tag{4.46}$$

The smoothing distribution $p(\mathbf{x}_k^j|M_{k+1}^i, \mathbf{y}_{1:N})$ of the state matched to the models M_k^j and M_{k+1}^i over two successive sampling periods can be expressed as

$$p(\mathbf{x}_k^j|M_{k+1}^i, \mathbf{y}_{1:N}) = \frac{1}{c} p(\mathbf{y}_{k+1:N}|M_{k+1}^i, \mathbf{x}_k) p(\mathbf{x}_k^j|\mathbf{y}_{1:k}), \tag{4.47}$$

where $p(\mathbf{y}_{k+1:N}|M_{k+1}^i, \mathbf{x}_k)$ is the forward-time model-conditioned filtering distribution, $p(\mathbf{x}_k^j|\mathbf{y}_{1:k})$ the backward-time one-step predictive distribution and c a normalizing constant. Thus, the smoothing distribution can be expressed as

$$\begin{aligned} p(\mathbf{x}_k^j|M_{k+1}^i, \mathbf{y}_{1:N}) &\propto N(\mathbf{x}_k|\hat{\mathbf{m}}_k^{b,i}, \hat{\mathbf{P}}_k^{b,i}) \cdot N(\mathbf{x}_k|\mathbf{m}_k^i, \mathbf{P}_k^i) && (4.48) \\ &= N(\mathbf{x}_k|\mathbf{m}_k^{s,ji}, \mathbf{P}_k^{s,ji}), && (4.49) \end{aligned}$$

where

$$\begin{aligned} \mathbf{m}_k^{s,ji} &= \mathbf{P}_k^{s,ji}\left[\left(\mathbf{P}_k^i\right)^{-1}\mathbf{m}_k^i + \left(\hat{\mathbf{P}}_k^{b,i}\right)^{-1}\hat{\mathbf{m}}_k^{b,i}\right] && (4.50) \\ \mathbf{P}_k^{s,ji} &= \left[\left(\mathbf{P}_k^i\right)^{-1} + \left(\hat{\mathbf{P}}_k^{b,i}\right)^{-1}\right]^{-1}. && (4.51) \end{aligned}$$

The model-conditioned smoothing distributions $p(\mathbf{x}_k^j|\mathbf{y}_{1:N})$, which were expressed as mixtures of Gaussians in (162), are now approximated by a single Gaussians via moment matching to yield

$$p(\mathbf{x}_k^j|\mathbf{y}_{1:N}) \approx N(\mathbf{x}_k^j|\mathbf{m}_k^{s,j}, \mathbf{P}_k^{s,j}), \tag{4.52}$$

where

$$\begin{aligned} \mathbf{m}_k^{s,j} &= \sum_{i=1}^n \mu_{k+1}^{s,i|j}\mathbf{m}_k^{s,ji} && (4.53) \\ \mathbf{P}_k^{s,j} &= \sum_{i=1}^n \mu_{k+1}^{s,i|j} \cdot \left[\mathbf{P}_k^{s,ij} + \left(\mathbf{m}_k^{s,ij} - \mathbf{m}_k^{s,j}\right)\left(\mathbf{m}_k^{s,ij} - \mathbf{m}_k^{s,j}\right)^T\right]. && (4.54) \end{aligned}$$

With these we can match the moments of the overall smoothing distribution to give a single Gaussian approximation

$$p(\mathbf{x}_k|\mathbf{y}_{1:N}) \approx N(\mathbf{x}_k|\mathbf{m}_k^s, \mathbf{P}_k^s), \tag{4.55}$$

where

$$\begin{aligned} \mathbf{m}_k^s &= \sum_{j=1}^n \mu_k^{s,j}\mathbf{m}_k^{s,ji} && (4.56) \\ \mathbf{P}_k^s &= \sum_{j=1}^n \mu_k^{s,j} \cdot \left[\mathbf{P}_k^{s,j} + \left(\mathbf{m}_k^{s,j} - \mathbf{m}_k^s\right)\left(\mathbf{m}_k^{s,j} - \mathbf{m}_k^s\right)^T\right]. && (4.57) \end{aligned}$$

These smoothing equations can be computed with function `imm_smooth`.

4.1.4 Demonstration: Tracking a Target with Simple Manouvers

A moving object with simple manouvers can be modeled by a Markovian switching system, which describes the normal movement dynamics with a simple Wiener process velocity model (see section 2.2.9) with low process noise value, and the manouvers with a Wiener process acceleration model (see section 2.1.4) with high process noise value. In the following we shall refer the former as model 1 and the latter as the model 2, which could actually also be a velocity model, but we use acceleration model instead to demonstrate how models with different structure are used in this toolbox.

The variance of process noise for model 1 is set to

$$\mathbf{Q}_c^1 = \begin{pmatrix} q_1 & 0 \\ 0 & q_1 \end{pmatrix} = \begin{pmatrix} 0.01 & 0 \\ 0 & 0.01 \end{pmatrix} \tag{4.58}$$

and for model 2 to

$$\mathbf{Q}_c^2 = \begin{pmatrix} q_2 & 0 \\ 0 & q_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{4.59}$$

In both cases the measurement model is the same as in the section 2.1.4 (that is, we observe the position of the object directly) with the exception that the variance of the measurements is now set to

$$\mathbf{R} = \begin{pmatrix} 0.1 & 0 \\ 0 & 0.1 \end{pmatrix}. \tag{4.60}$$

The time step size is set to $\Delta t = 0.1$. The true starting state of the system is set to

$$\mathbf{x}_0 = \begin{bmatrix} 0 & 0 & 0 & -1 & 0 & 0 \end{bmatrix}, \tag{4.61}$$

which means that the object starts to move from origo with velocity -1 along the y-axis. The model transition probability matrix is set to

$$\mathbf{\Phi} = \begin{pmatrix} 0.98 & 0.02 \\ 0.02 & 0.98 \end{pmatrix}, \tag{4.62}$$

which means that both models have equal probability of shifting to another model during each sampling period. The prior model probabilities are set to

$$\mu_0 = \begin{bmatrix} 0.9 & 0.1 \end{bmatrix}. \tag{4.63}$$

In software code the filtering loop of IMM filter can be done as follows:

```
for i = 1:size(Y,2)
  [x_p,P_p,c_j] = imm_predict(x_ip,P_ip,mu_ip,p_ij,ind,dims,A,Q);
  [x_ip,P_ip,mu_ip,m,P] = imm_update(x_p,P_p,c_j, ...
    ind,dims,Y(:,i),H,R);
  MM(:,i) = m;
  PP(:,:,i) = P;
  MU(:,i) = mu_ip';
  MM_i(:,i) = x_ip';
  PP_i(:,i) = P_ip';
end
```

The variables x_ip and P_ip contain the updated state mean and covariance for both models as cell arrays. The cell arrays are used because the size of the state variable can vary between different models. For example, in this example the models have 4 and 6 state components. Similarly, the variables x_p and P_p contain the predicted state means and covariances. Likewise, the model parameters (variables A,Q,H and R) are all cell arrays containing the corresponding model parameters for each model. The variable mu_ip contains the probability estimates of each model as a regular array, which is initialized before the filtering loop with the prior probabilities of models (eq. (194)). The variable p_ij is the model transition probability matrix (193) as a regular Matlab matrix. The vector c_j contains the normalizing constants computed during the prediction step (eq. (127)), which are needed during the update step in eqs. (134) and (134). The variable ind is a cell array containing vectors, which map the state variables of each model to state variables of the original system under the study. For example, for model 1 the vector is [1 2 3 4]' and for model 2 [1 2 3 4 5 6]'. The purpose of this might seem a bit unclear for systems like these, but for more complex models we must know how the state variables are connected as they are not always in the same order and some components might be missing. The variable dims contains the number of dimensions in the original system under the study, which in this case is 6. The last five lines of code store the estimation results for each time step.

The smoothing can be done with the function call

```
[SM,SP,SM_i,SP_i,MU_S] = imm_smooth(MM,PP,MM_i,PP_i,MU,p_ij, ...
  mu_0j,ind,dims,A,Q,R,H,Y);
```

where the variable mu_0j contains the prior model probabilities. The overall smoothed state estimates are returned in variables SM and SP, and the model-conditioned smoothed estimates in variables SM_i and SP_i. The variable MU_S contains the calculated smoothed model probabilities.

The system is simulated 200 time steps, and the active model during the steps $1-50$, $71-120$ and $151-200$ is set to model 1 and during the steps $51-70$ and $121-150$ to model 2. The purpose of forcing the model transitions instead of simulating them randomly is to produce two manouvers, which can be used to demonstrate the properties of IMM-filter. It also reflects the fact that in real problems we do not know the model transition probability matrix accurately. The figures 17, 18 and 19 show the true trajectory of the object and the measurements made of it. The two manouvers can be clearly seen. Figures also show the filtering and smoothing results produced by Kalman filter and (RTS) smoother using the models 1 and 2 separately (figures 17 and 18, correspodingly) as well as the estimates produced by the IMM filter and smoother (figure 19). In figure 20 we have plotted the filtered and smoothed estimates of the probabilities of model 1 in each time step. It can be seen that it takes some time for the filter to respond to model transitions. As one can expect, smoothing reduces this lag as well as giving substantially better overall performance.

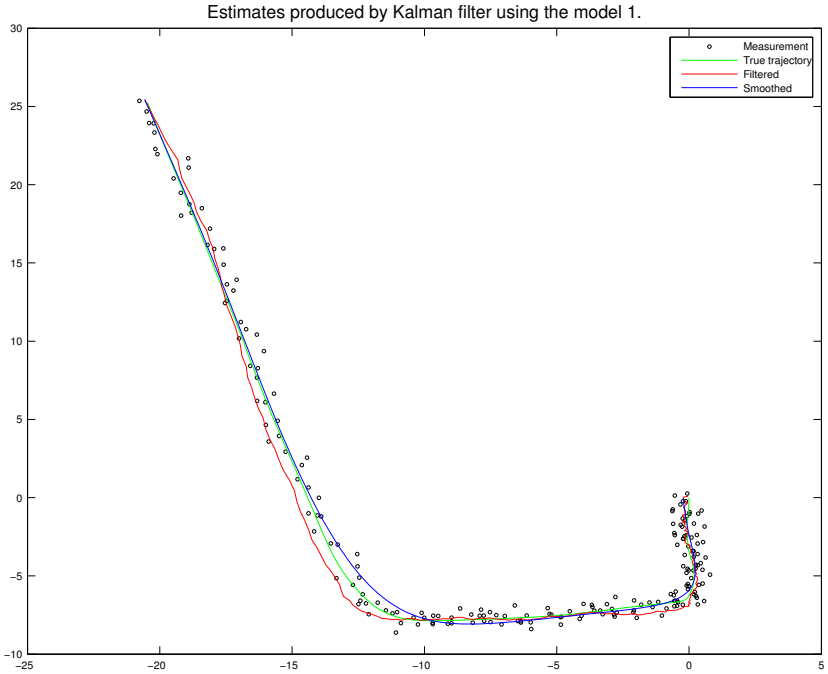


Figure 4.1: Tracking results of Kalman filter and smoother with model 1 in Tracking a Target with Simple Manouvers example.

The illustrate the performance differences between different models we have listed the MSE errors of position estimates for each model in the table 5. From these it can be seen that the estimates of Kalman filter with model 1 are clearly much worse than with model 2 or with the IMM filter. On the otherhand, the difference between the KF with model 2 and the IMM filter is smaller, but still significant in the favor of IMM. This is most likely due to fact that model 2 is more flexible than model 1, so model 2 is better in the regions of model 1 than model 1 in the regions of model 2. The bad performance of the model 1 can be seen especially during the manouvers. The IMM filter combines the properties of both models in a suitable way, so it gives the best overall performance. The effect of smoothing to tracking accuracy is similar with all models.

4.2 Nonlinear Systems

The non-linear versions of IMM filter and smoother reviewed in previous section can be obtained simply by replacing the Kalman filter prediction and update steps (in eqs. (131), (131), (147) and (151)) by their extended Kalmam filter or unscented Kalman filter counterparts, which were reviewed in sections 2.2.2 and 2.2.6. These algorithms are commonly referred as IMM-EKF and IMM-UKF. Naturally this approach introduces some error to estimates of already suboptimal IMM, but can

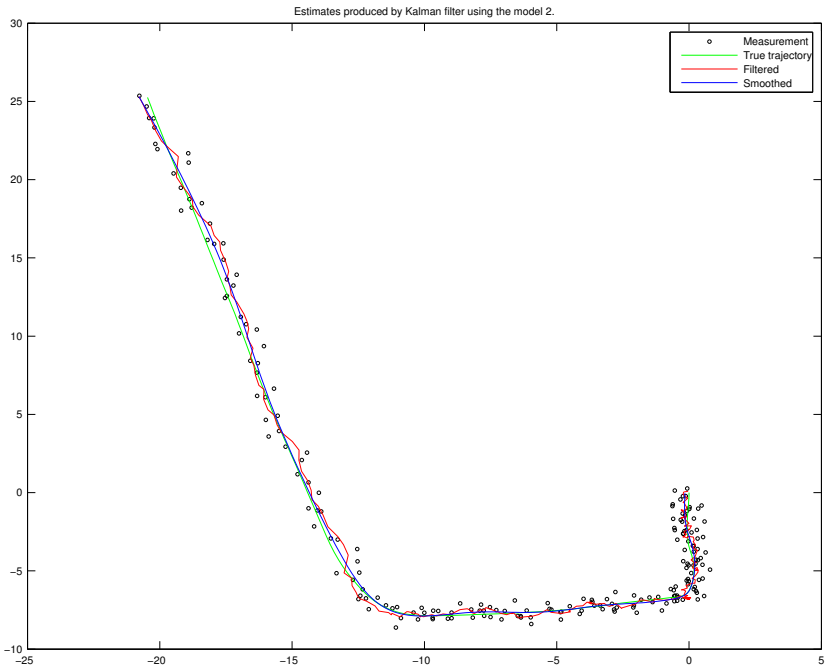


Figure 4.2: Tracking results of Kalman filter and smoother with model 2 in Tracking a Target with Simple Manouvers example.

Method	*MSE*
KF1	0.1554
KS1	0.0314
KF2	0.0317
KS2	0.0071
IMM	0.0229
IMMS	0.0057

Table 4.1: Average MSEs of estimating the position in Tracking a Target with Simple Manouvers example over 1000 Monte Carlo runs.

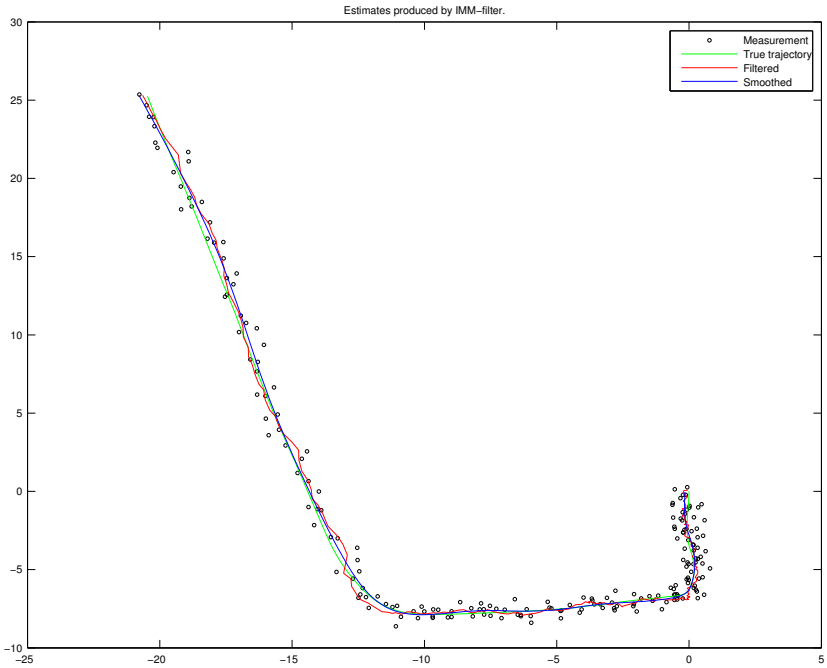


Figure 4.3: Tracking results of IMM filter and smoother in Tracking a Target with Simple Manouvers example.

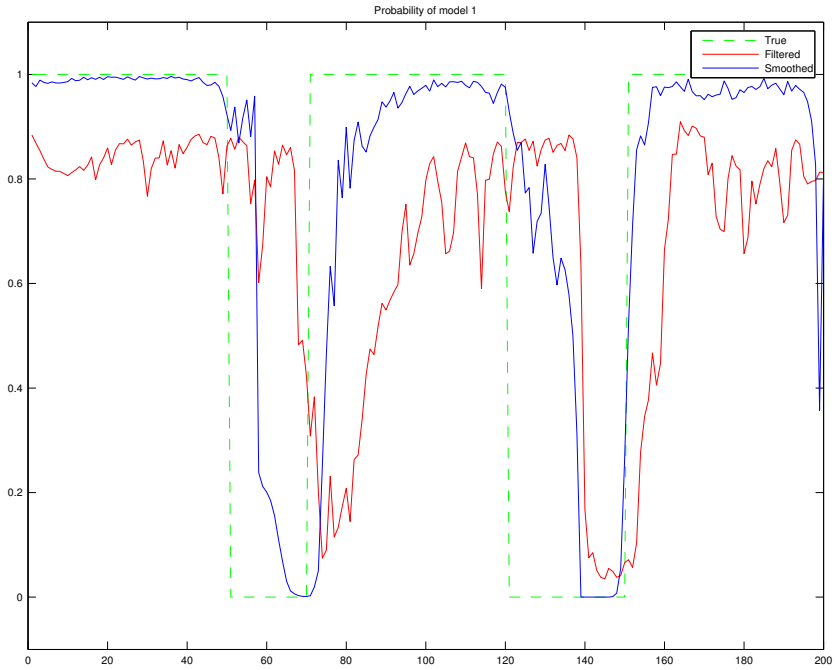


Figure 4.4: Filtered and smoothed probability of model 1 in each time step in Tracking a Target with Simple Manouvers example.

still provide sufficient accuracy with suitable models. filter based IMM filter for estimating more models, but that approach is not currently

The prediction and update steps of IMM-EKF is provided by functions `eimm_predict` and `eimm_update`, and smoothing can be done with function `eimm_smooth`. The corresponding functions for IMM-UKF are `uimm_predict`, `uimm_update` and `uimm_smooth`.

4.2.1 Demonstration: Coordinated Turn Model

A common way of modeling a turning object is to use the *coordinated turn model* (see, e.g., Bar-Shalom et al., 2001). The idea is to augment the state vector with a turning rate parameter ω, which is to be estimated along with the other system parameters, which in this example are the position and the velocity of the target. Thus, the joint system vector can be expressed as

$$\mathbf{x}_k = \begin{pmatrix} x_k & y_k & \dot{x}_k & \dot{y}_k & \omega_k \end{pmatrix}^T. \tag{4.64}$$

The dynamic model for the coordinated turns is

$$\mathbf{x}_{k+1} = \underbrace{\begin{pmatrix} 1 & 0 & \frac{\sin(\omega_k \Delta \mathrm{t})}{\omega_k} & \frac{\cos(\omega_k \Delta \mathrm{t})-1}{\omega_k} & 0 \\ 0 & 1 & \frac{1-\cos(\omega_k \Delta \mathrm{t})}{\omega_k} & \frac{\sin(\omega_k \Delta \mathrm{t})}{\omega_k} & 0 \\ 0 & 0 & \cos(\omega_k \Delta \mathrm{t}) & -\sin(\omega_k \Delta \mathrm{t}) & 0 \\ 0 & 0 & \sin(\omega_k \Delta \mathrm{t}) & \cos(\omega_k \Delta \mathrm{t}) & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}}_{\mathbf{F}_k} \mathbf{x}_k + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} v_k, \tag{4.65}$$

where $v_k \sim N(0, \sigma_\omega^2)$ is univariate white Gaussian process noise for the turn rate parameter. This model is, despite the matrix form, non-linear. Equivalently, the dynamic model can be expressed as a set of equations

$$\begin{aligned} x_{k+1} &= x_k + \frac{\sin(\omega_k \Delta \mathrm{t})}{\omega_k}\dot{x}_k + \frac{\cos(\omega_k \Delta \mathrm{t}) - \Delta \mathrm{t}}{\omega_k}\dot{x}_k \\ y_{k+1} &= y_k + \frac{1 - \cos(\omega_k \Delta \mathrm{t})}{\omega_k}\dot{x}_k + \frac{\sin(\omega_k \Delta \mathrm{t})}{\omega_k}\dot{y}_k \\ \dot{x}_{k+1} &= \cos(\omega_k \Delta \mathrm{t})\dot{x}_k - \sin(\omega_k \Delta \mathrm{t})\dot{y}_k \\ \dot{y}_{k+1} &= \sin(\omega_k \Delta \mathrm{t})\dot{x}_k + \cos(\omega_k \Delta \mathrm{t})\dot{y}_k \\ \omega_{k+1} &= \omega_k + v_k \end{aligned}. \tag{4.66}$$

The source code of the turning model can found in the m-file `f_turn.m`. The inverse dynamic model can be found in the file `f_turn_inv.m`, which basically calculates the dynamics using the inverse $\mathbf{F}_k$ in eq. (189) as it's transition matrix.

To use EKF in estimating the turning rate ω_k the Jacobian of the dynamic model must be computed, which is given as

$$\mathbf{F}_{\mathbf{x}}(\mathbf{m}, k) = \begin{pmatrix} 1 & 0 & \frac{\sin(\omega_k \Delta \mathrm{t})}{\omega_k} & \frac{\cos(\omega_k \Delta \mathrm{t})-1}{\omega_k} & \frac{\partial x_{k+1}}{\partial \omega_k} \\ 0 & 1 & \frac{1-\cos(\omega_k \Delta \mathrm{t})}{\omega_k} & \frac{\sin(\omega_k \Delta \mathrm{t})}{\omega_k} & \frac{\partial y_{k+1}}{\partial \omega_k} \\ 0 & 0 & \cos(\omega_k \Delta \mathrm{t}) & -\sin(\omega_k \Delta \mathrm{t}) & \frac{\partial \dot{x}_{k+1}}{\partial \omega_k} \\ 0 & 0 & \sin(\omega_k \Delta \mathrm{t}) & \cos(\omega_k \Delta \mathrm{t}) & \frac{\partial \dot{y}_{k+1}}{\partial \omega_k} \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \tag{4.67}$$

where the partial derivatives w.r.t to turning rate are

$$\begin{aligned} \frac{\partial x_{k+1}}{\partial \omega_k} &= \frac{\omega \Delta \mathrm{t} \cos(\omega_k \Delta \mathrm{t}) - \sin(\omega_k \Delta \mathrm{t})}{\omega_k^2} \dot{x}_k - \frac{\omega_k \Delta \mathrm{t} \sin(\omega_k \Delta \mathrm{t}) + \cos(\omega_k \Delta \mathrm{t}) - 1}{\omega_k^2} \dot{y}_k \\ \frac{\partial y_{k+1}}{\partial \omega_k} &= \frac{\omega_k \Delta \mathrm{t} \sin(\omega_k \Delta \mathrm{t}) + \cos(\omega_k \Delta \mathrm{t}) - 1}{\omega_k^2} \dot{x}_k - \frac{\omega_k \Delta \mathrm{t} \cos(\omega_k \Delta \mathrm{t}) - \sin(\omega_k \Delta \mathrm{t})}{\omega_k^2} \dot{y}_k \\ \frac{\partial \dot{x}_{k+1}}{\partial \omega_k} &= -\Delta \mathrm{t} \sin(\omega_k \Delta \mathrm{t}) \dot{x}_k - \Delta \mathrm{t} \cos(\omega_k \Delta \mathrm{t}) \dot{y}_k \\ \frac{\partial \dot{y}_{k+1}}{\partial \omega_k} &= -\Delta \mathrm{t} \cos(\omega_k \Delta \mathrm{t}) \dot{x}_k - \Delta \mathrm{t} \sin(\omega_k \Delta \mathrm{t}) \dot{y}_k. \end{aligned} \tag{4.68}$$

The source code for calculating the Jacobian is located in the m-file `f_turn_dx.m`.

Like in previous demonstration we simulate the system 200 time steps with step size $\Delta \mathrm{t} = 0.1$. The movement of the object is produced as follows:

- Object starts from origo with velocity $(\dot{x}, \dot{y}) = (1, 0)$.
- At 4s object starts to turn left with rate $\omega = 1$.
- At 9s object stops turning and moves straight for 2 seconds with a constant total velocity of one.
- At 11s objects starts to turn right with rate $\omega = -1$.
- At 16s object stops turning and moves straight for 4 seconds with the same velocity.

The resulting trajectory is plotted in figure 21. The figure also shows the measurements, which were made with the same model as in the previous example, that is, we observe the position of the object directly with the additive noise, whose variance in this case is set to $\sigma_r^2 = 0.05$.

In the estimation we use the following models:

1. Standard Wiener process velocity model (see, for example, section 2.2.9) with process noise variance $q_1 = 0.05$, whose purpose is to model the relatively slow turns. This model is linear, so we can use the standard Kalman filter for estimation.

2. A combination of Wiener process velocity model and a coordinated turn model described above. The variance of the process noise for the velocity model is set to $q_2 = 0.01$ and for the turning rate parameter in the turning model to $q_\omega = 0.15$. The estimation is now done with both the EKF and UKF based IMM filters as the turning model is non-linear. However, as the measurement model is linear, we can still use the standard IMM filter update step instead of a non-linear one. In both cases the model transition probability matrix is set to

$$\mathbf{\Phi} = \begin{pmatrix} 0.9 & 0.1 \\ 0.1 & 0.9 \end{pmatrix}, \tag{4.69}$$

and the prior model probabilities are

$$\mu_0 = \begin{bmatrix} 0.9 & 0.1 \end{bmatrix}. \tag{4.70}$$

model (in case of $\omega = 0$)

In software code the prediction step of IMM-EKF can be done with the function call

```
[x_p,P_p,c_j] = eimm_predict(x_ip,P_ip,mu_ip,p_ij,ind, ...
  dims,A,a_func,a_param,Q);
```

which is almost the same as the linear IMM-filter prediction step with the exception that we must now also pass the function handles of the dynamic model and it's Jacobian as well as their possible parameters to the prediction step function. In above these are the variables `A`, `a_func` and `a_param` which are cell arrays containing the needed values for each model. If some of the used models are linear (as is the case now) their elements of the cell arrays `a_func` and `a_param` are empty, and `A` contains the dynamic model matrices instead of Jacobians. The prediction step function of IMM-UKF has exactly the same interface.

The EKF based IMM smoothing can be done with a function call

```
[SMI,SPI,SMI_i,SPI_i,MU_S] = eimm_smooth(MM,PP,MM_i,PP_i,MU,p_ij, ...
  mu_0j,ind,dims,A,ia_func, ...
  a_param,Q,R,H,h_func,h_param,Y);
```

where we pass the function handles of inverse dynamic models (variable `ia_func`) for the backward time IMM filters as well as Jacobians of the dynamic models (`A`) and their parameters (`a_param`), which all are cell arrays as in the filter described above. Sameway we also pass the handles to the measurement model functions (`h_func`) and their Jacobians (`H`) as well as their parameters (`h_param`). UKF based IMM smoothing is done exactly the same.

In figure 22 we have plotted the filtered and smoothed estimates produced all the tested estimation methods. It can be seen that the estimates produced by IMM-EKF and IMM-UKF are very close on each other while IMM-UKF still seems to

Method	*MSE*
KF	0.0253
KS	0.0052
EIMM1	0.0179
EIMMS1	0.0039
UIMM1	0.0155
UIMMS1	0.0036

Table 4.2: Average MSEs of estimating the position in Tracking Object with Simple Manouvers example over 100 Monte Carlo runs.

be a little closer to the right trajectory. The estimates of standard KF differ more from the estimates of IMM-EKF and IMM-UKF and seem to have more problems during the later parts of turns. Smoothing improves all the estimates, but in the case of KF the estimates are clearly oversmoothed.

The estimates for probability of model 2 produced by IMM-EKF and IMM-UKF are plotted in figure 23. The filtered estimates seem to be very close on each other whereas the smoothed estimates have little more variation, but neither one of them don't seem to be clearly better. The reason why both methods give about 50% probability of model 2 when the actual model is 1 can be explained by the fact that the coordinated turn model actually contains the velocity model as it's special case when $\omega = 0$. The figure 23 shows the estimates of the turn rate parameter with both IMM filters and smoothers, and it can be seen that smoothing improves the estimates tremendously. It is also easy to see that in some parts IMM-UKF gives a little better performance than IMM-EKF.

In table 6 we have listed the average MSEs of position estimates produced by the tested methods. It can be seen, that the estimates produced by IMM-EKF and IMM-UKF using the combination of a velocity and a coordinated turn model are clearly better than the ones produced by a standard Kalman filter using the velocity model alone. The performance difference between IMM-EKF and IMM-UKF is also clearly noticable in the favor of IMM-UKF, which is also possible to see in figure 22. The effect of smoothing to estimation accuracy is similar in all cases.

4.2.2 Demonstration: Bearings Only Tracking of a Manouvering Target

We now extend the previous demonstration by replacing the linear measurement model with a non-linear bearings only measurement model, which were reviewed in section 2.2.9. The estimation problem is now harder as we must use a non-linear version of IMM filter update step in addition to the prediction step, which was used in the previous demonstration.

The trajectory of the object is exactly the same as in the previous example. The sensors producing the angle measurements are located in $(s_x^1, s_y^1) = (-0.5, 3.5)$,

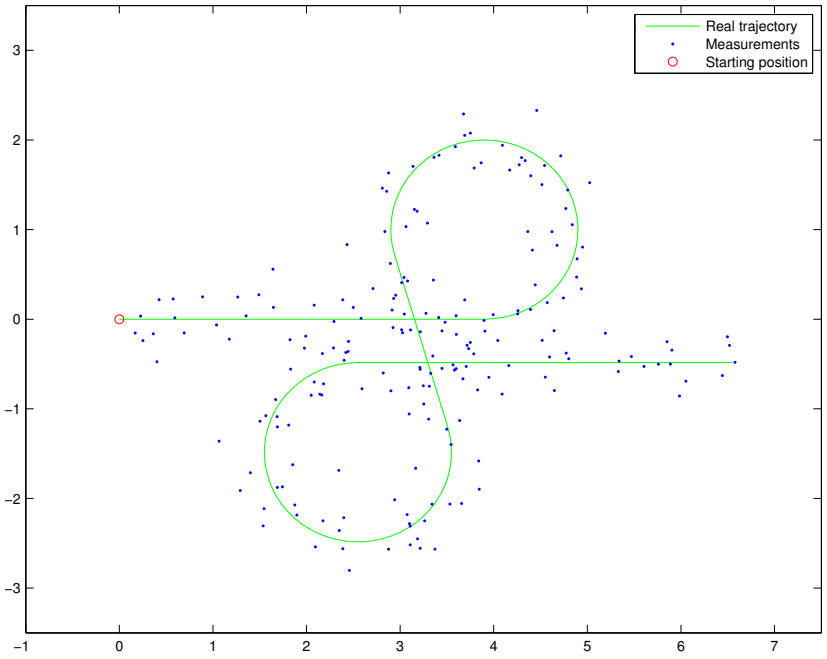


Figure 4.5: Object's trajectory and a sample of measurements in the Coordinated turn model demonstration.

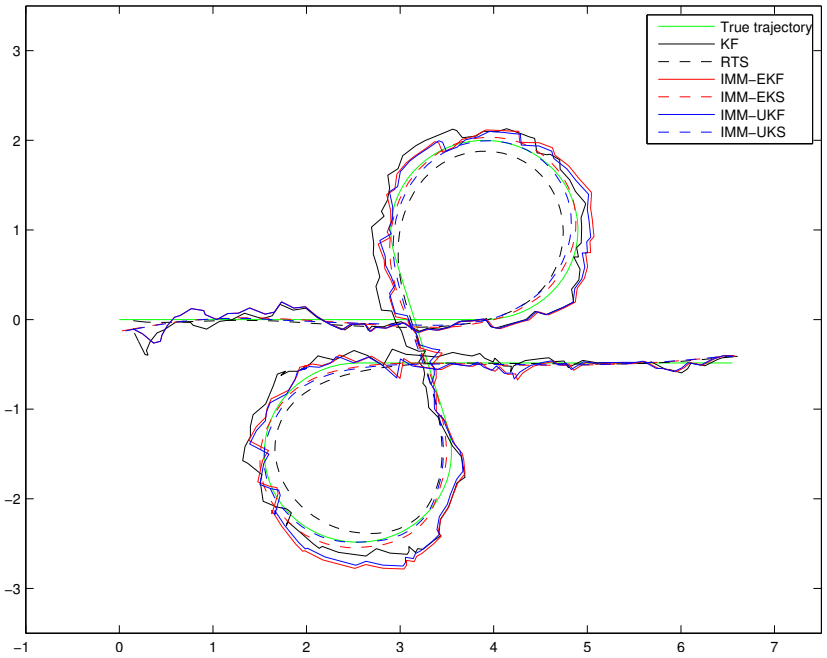


Figure 4.6: Position estimates in the Coordinated turn model demonstration.

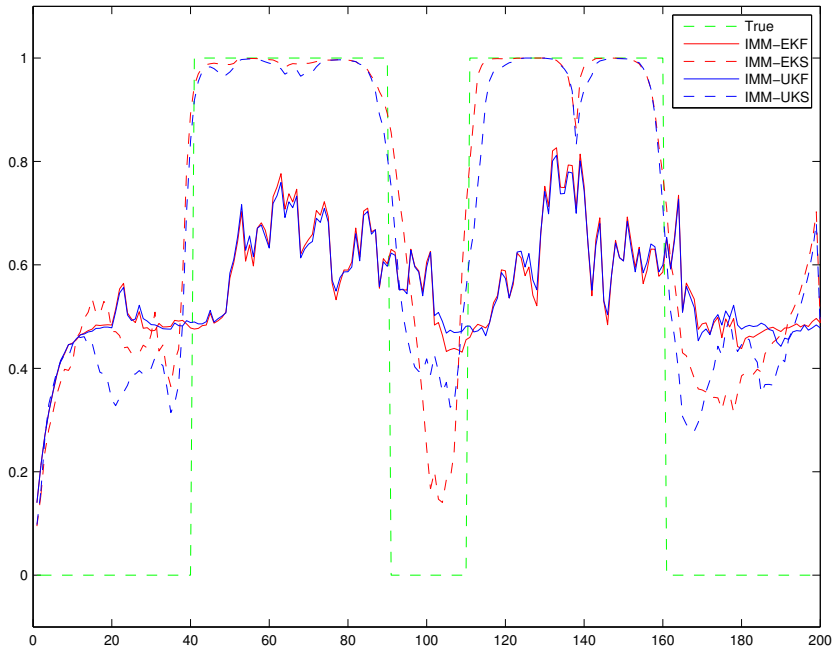


Figure 4.7: Estimates for model 2's probability in Coordinated turn model demonstration.

Figure 4.8: Estimates of the turn rate parameter ω_k in the Coordinated turn model demonstration.

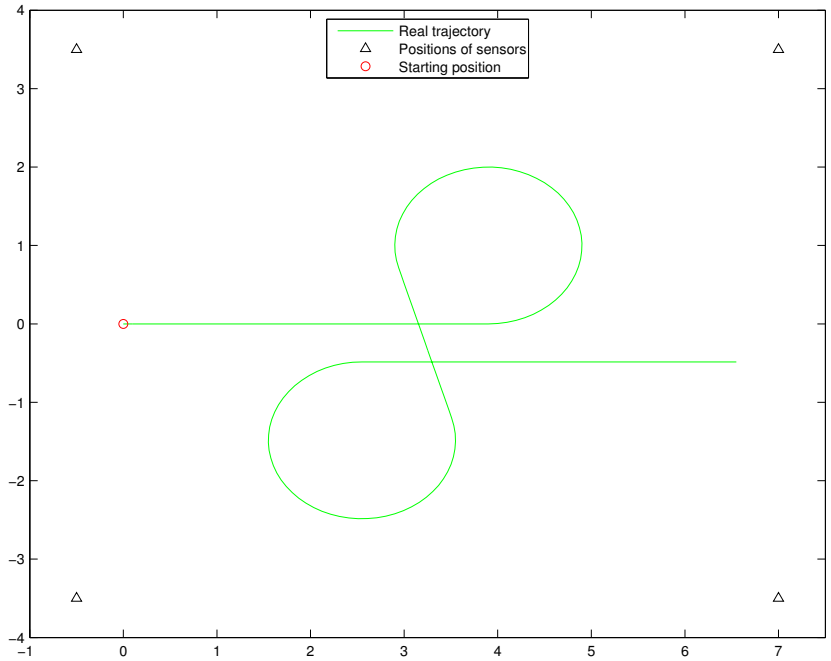


Figure 4.9: Object's trajectory and positions of the sensors in Bearings Only Tracking of a Manouvering Target demonstration.

$(s_x^2, s_y^2) = (-0.5, 3.5)$, $(s_x^3, s_y^3) = (7, -3.5)$ and $(s_x^4, s_y^4) = (7, 3.5)$. The trajectory and the sensor positions are shown in figure 25. The standard deviation of the measurements is set to $\sigma = 0.1$ radians, which is relatively high.

The function call of IMM-EKF update step is of form

```
[x_ip,P_ip,mu_ip,m,P] = eimm_update(x_p,P_p,c_j,ind,dims, ...
  Y(:,i),H,h_func,R,h_param);
```

which differs from the standard IMM filter update with the additional parameters `h` and `h_param`, which contain the handles to measurement model functions and their parameters, respectively. Also, the parameter `H` now contains the handles to functions calculating the Jacobian's of the measurement functions. In IMM-UKF the update function is specified similarly.

The position of the object is estimated with the following methods:

- EKF and EKS: Extended Kalman filter and (RTS) smoother using the same Wiener process velocity model as in the previous demonstration in the case standard Kalman filter.
- UKF and UKS: Unscented Kalman filter and (RTS) smoother using the same model as the EKF.

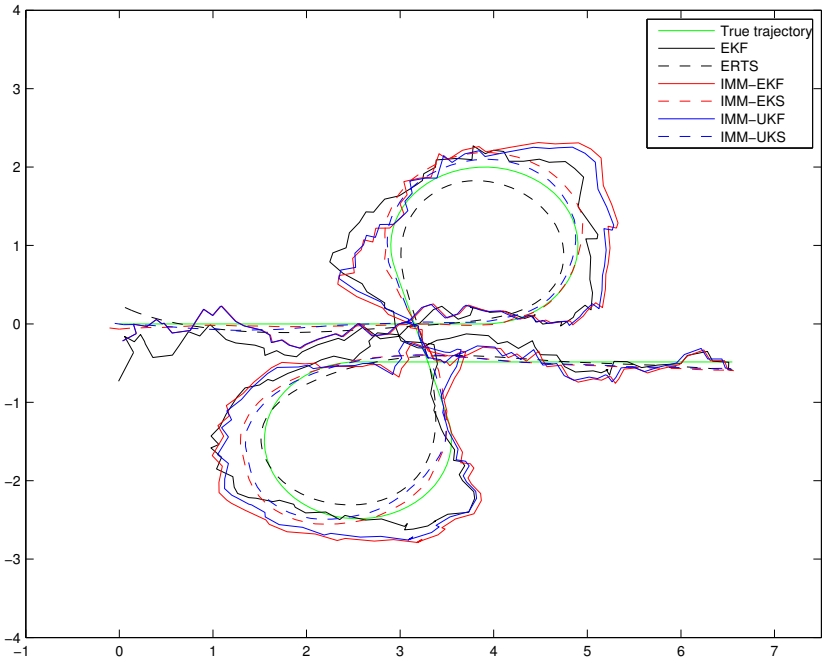


Figure 4.10: Filtered and smoothed estimates of object's position using all the tested methods in Bearings Only Tracking of a Manouvering Target demonstration.

- IMM-EKF and IMM-EKS: EKF based IMM filter and smoother using the same combination of Wiener process velocity model and a coordinated turn model as was used in the previous demonstration in the case of IMM-EKF and IMM-UKF.
- IMM-UKF and IMM-UKS: UKF based IMM filter and smoother using the same models as IMM-EKF.

A sample of trajectory estimates are plotted in figure 26. The estimates are clearly more inaccurate than the ones in the previous section. In figure 27 we have plotted the estimates of model 2's probability for IMM-EKF and IMM-UKF. The figure look very similar to the one in the previous demonstration, despite the non-linear and more noisy measurements. Also the turn rate estimates, which are plotted in figure 28, are very similar to the ones in the previous section with exception that now the difference between the smoothed estimates of IMM-EKF and IMM-UKF is bigger.

In table 7 we have listed the average mean square errors of position estimates over 100 Monte Carlo runs. It can be observed that the estimates of EKF and UKF are identical in practice, which is to be expected from Bearings Only Tracking demonstration. The difference between IMM-UKF and IMM-EKF has grown in the favor of IMM-UKF, whereas the accuracy of IMM-EKF is now more close to

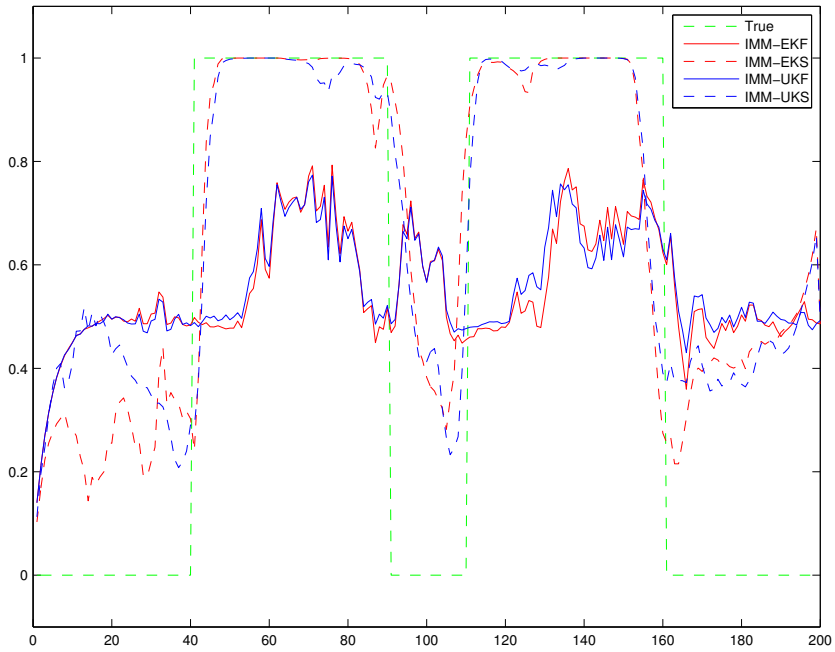


Figure 4.11: Estimates for model 2's probability in Bearings Only Tracking of a Manouvering Target demonstration.

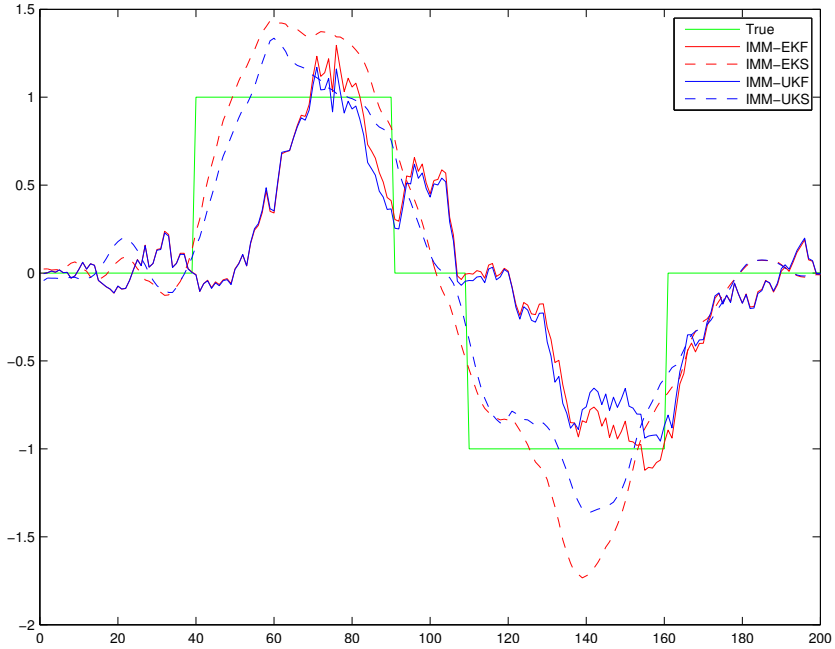


Figure 4.12: Estimates for the turn rate parameter ω_k in Bearings Only Tracking of a Manouvering Target demonstration

Method	*MSE*
EKF	0.0606
ERTS	0.0145
UKF	0.0609
URTS	0.0144
IMM-EKF	0.0544
IMM-EKS	0.0094
IMM-UKF	0.0441
IMM-UKS	0.0089

Table 4.3: Average MSEs of estimating the position in Bearings Only Tracking of a Manouvering Target example over 100 Monte Carlo runs.

the ones of EKF and UKF. On the other hand the smoothed estimates of IMM-UKF and IMM-EKF are still very close to one another, and are considerably better than the smoothed estimates of EKF and UKF.

It should be noted that the performance of each tested method could be tuned by optimizing their parameters (e.g. variance of process noise of dynamic models, values of model transition matrix in IMM etc.) more carefully, so the performance differences could change radically. Still, it is clear that IMM filter does actually work also with (atleast some) non-linear dynamic and measurement models, and should be considered as a standard estimation method for multiple model systems. Also, one should prefer IMM-UKF over IMM-EKF as the performance is clearly (atleast in these cases) better, and the implementation is easier, as we have seen in the previous examples.

Chapter 5

Functions in the Toolbox

5.1 Discrete-time State Space Estimation

5.1.1 Linear Kalman Filter

kf_predict

kf_predict Perform Kalman Filter prediction step. The model is		
Syntax:	`[X,P] = KF_PREDICT(X,P,A,Q,B,U)`	
Input:	`X`	Nx1 mean state estimate of previous step
	`P`	NxN state covariance of previous step
	`A`	Transition matrix of discrete model (optional, default identity)
	`Q`	Process noise of discrete model (optional, default zero)
	`B`	Input effect matrix (optional, default identity)
	`U`	Constant input (optional, default empty)
Output:	`X`	Predicted state mean
	`P`	Predicted state covariance

kf_update

kf_update		
Kalman filter measurement update step. Kalman Filter model is		
Syntax:	`[X,P,K,IM,IS,LH] = KF_UPDATE(X,P,Y,H,R)`	
Input:	`X`	Nx1 mean state estimate after prediction step
	`P`	NxN state covariance after prediction step
	`Y`	Dx1 measurement vector.
	`H`	Measurement matrix.
	`R`	Measurement noise covariance.
Output:	`X`	Updated state mean
	`P`	Updated state covariance
	`K`	Computed Kalman gain
	`IM`	Mean of predictive distribution of Y
	`IS`	Covariance or predictive mean of Y
	`LH`	Predictive probability (likelihood) of measurement.

kf_lhood

kf_lhood		
Calculate likelihood of measurement in Kalman filter. If and X and P define the parameters of predictive distribution (e.g. from KF_PREDICT)		
Syntax:	`LH = KF_LHOOD(X,P,Y,H,R)`	
Input:	`X`	Nx1 state mean
	`P`	NxN state covariance
	`Y`	Dx1 measurement vector.
	`H`	Measurement matrix.
	`R`	Measurement noise covariance.
Output:	`LH`	Likelihood of measurement.

kf_loop

kf_loop		
Calculates state estimates for a set measurements using the Kalman filter. This function is for convience, as it basically consists only of a space reservation for the estimates and of a for-loop which calls the predict and update steps of the KF for each time step in the measurements. See also: KF_PREDICT, KF_UPDATE		
Syntax:	`[MM,PP] = KF_LOOP(X,P,H,R,Y,A,Q)`	
Input:	`X`	Nx1 initial estimate for the state mean
	`P`	NxN initial estimate for the state covariance
	`H`	DxN measurement matrix
	`R`	DxD measurement noise covariance
	`Y`	DxM matrix containing all the measurements.
	`A`	Transition matrix of the discrete model (optional, default identity)
	`Q`	Process noise of the discrete model (optional, default zero)
	`MM`	Filtered state mean sequence
	`MM`	Filtered state mean sequence
Output:	`MM`	Filtered state mean sequence
	`PP`	Filtered state covariance sequence

rts_smooth

rts_smooth		
Rauch-Tung-Striebel smoother algorithm. Calculate "smoothed" sequence from given Kalman filter output sequence by conditioning all steps to all measurements.		
Syntax:	`[M,P,S] = RTS_SMOOTH(M,P,A,Q)`	
Input:	`M`	NxK matrix of K mean estimates from Kalman filter
	`P`	NxNxK matrix of K state covariances from Kalman Filter
	`A`	NxN state transition matrix or NxNxK matrix of K state transition matrices for each step.
	`Q`	NxN process noise covariance matrix or NxNxK matrix of K state process noise covariance matrices for each step.
Output:	`M`	Smoothed state mean sequence
	`P`	Smoothed state covariance sequence
	`D`	Smoother gain sequence

tf_smooth

tf_smooth		
Two filter linear smoother algorithm. Calculate "smoothed" sequence from given Kalman filter output sequence by conditioning all steps to all measurements.		
Syntax:	`[M,P] = TF_SMOOTH(M,P,Y,A,Q,H,R, [use_inf])`	
Input:	`M`	NxK matrix of K mean estimates from Kalman filter
	`P`	NxNxK matrix of K state covariances from Kalman Filter
	`Y`	Sequence of K measurement as DxK matrix
	`A`	NxN state transition matrix.
	`Q`	NxN process noise covariance matrix.
	`H`	DxN Measurement matrix.
	`R`	DxD Measurement noise covariance.
	`use_inf`	If information filter should be used (default 1)
Output:	`M`	Smoothed state mean sequence
	`P`	Smoothed state covariance sequence

5.1.2 Extended Kalman Filter

ekf_predict1

ekf_predict1		
Perform Extended Kalman Filter prediction step.		
Syntax:	`[M,P] = EKF_PREDICT1(M,P, [A,Q,a,W,param])`	
Input:	`M`	Nx1 mean state estimate of previous step
	`P`	NxN state covariance of previous step
	`A`	Derivative of a() with respect to state as matrix, inline function, function handle or name of function in form A(x,param) (optional, default eye())
	`Q`	Process noise of discrete model (optional, default zero)
	`a`	Mean prediction E[a(x[k-1],q=0)] as vector, inline function, function handle or name of function in form a(x,param) (optional, default A(x)*X)
	`W`	Derivative of a() with respect to noise q as matrix, inline function, function handle or name of function in form W(x,param) (optional, default identity)
	`param`	Parameters of a (optional, default empty)
Output:	`M`	Updated state mean
	`P`	Updated state covariance

ekf_predict2

ekf_predict2		
Perform Extended Kalman Filter prediction step.		
Syntax:	`[M,P] = EKF_PREDICT2(M,P, [A,F,Q,a,W,param])`	
Input:	`M`	Nx1 mean state estimate of previous step
	`P`	NxN state covariance of previous step
	`A`	Derivative of a() with respect to state as matrix, inline function, function handle or name of function in form A(x,param) (optional, default identity)
	`F`	NxNxN Hessian matrix of the state transition function w.r.t. state variables as matrix, inline function, function handle or name of function in form F(x,param) (optional, default identity)
	`Q`	Process noise of discrete model (optional, default zero)
	`a`	Mean prediction E[a(x[k-1],q=0)] as vector, inline function, function handle or name of function in form a(x,param) (optional, default A(x)*X)
	`W`	Derivative of a() with respect to noise q as matrix, inline function, function handle or name of function in form W(x,k-1,param) (optional, default identity)
	`param`	Parameters of a (optional, default empty)
Output:	`M`	Updated state mean
	`P`	Updated state covariance

ekf_update1

ekf_update1		
Extended Kalman Filter measurement update step. EKF model is		
Syntax:	`[M,P,K,MU,S,LH] = EKF_UPDATE1(M,P,Y,H,R, [h,V,param])`	
Input:	`M`	Nx1 mean state estimate after prediction step
	`P`	NxN state covariance after prediction step
	`Y`	Dx1 measurement vector.
	`H`	Derivative of h() with respect to state as matrix, inline function, function handle or name of function in form H(x,param)
	`R`	Measurement noise covariance.
	`h`	Mean prediction (innovation) as vector, inline function, function handle or name of function in form h(x,param). (optional, default H(x)*X)
	`V`	Derivative of h() with respect to noise as matrix, inline function, function handle or name of function in form V(x,param). (optional, default identity)
	`param`	Parameters of h (optional, default empty)
Output:	`M`	Updated state mean
	`P`	Updated state covariance
	`K`	Computed Kalman gain
	`MU`	Predictive mean of Y
	`S`	Predictive covariance of Y
	`LH`	Predictive probability (likelihood) of measurement.

ekf_update2

ekf_update2		
Extended Kalman Filter measurement update step. EKF model is		
Syntax:	`[M,P,K,MU,S,LH] = EKF_UPDATE2(M,P,Y,H,H_xx,R, [h,V,param])`	
Input:	`M`	Nx1 mean state estimate after prediction step
	`P`	NxN state covariance after prediction step
	`Y`	Dx1 measurement vector.
	`H`	Derivative of h() with respect to state as matrix, inline function, function handle or name of function in form H(x,param)
	`H_xx`	DxNxN Hessian of h() with respect to state as matrix, inline function, function handle or name of function in form H_xx(x,param)
	`R`	Measurement noise covariance.
	`h`	Mean prediction (measurement model) as vector, inline function, function handle or name of function in form h(x,param). (optional, default H(x)*X)
	`V`	Derivative of h() with respect to noise as matrix, inline function, function handle or name of function in form V(x,param). (optional, default identity)
	`param`	Parameters of h (optional, default empty)
Output:	`M`	Updated state mean
	`P`	Updated state covariance
	`K`	Computed Kalman gain
	`MU`	Predictive mean of Y
	`S`	Predictive covariance Y
	`LH`	Predictive probability (likelihood) of measurement.

erts_smooth1

erts_smooth1 Extended Rauch-Tung-Striebel smoother algorithm. Calculate "smoothed" sequence from given Kalman filter output sequence by conditioning all steps to all measurements.		
Syntax:	`[M,P,D] = ERTS_SMOOTH1(M,P,A,Q, [a,W,param,same_p])`	
Input:	`M`	NxK matrix of K mean estimates from Unscented Kalman filter
	`P`	NxNxK matrix of K state covariances from Unscented Kalman Filter
	`A`	Derivative of a() with respect to state as matrix, inline function, function handle or name of function in form A(x,param) (optional, default eye())
	`Q`	Process noise of discrete model (optional, default zero)
	`a`	Mean prediction E[a(x[k-1],q=0)] as vector, inline function, function handle or name of function in form a(x,param) (optional, default A(x)*X)
	`W`	Derivative of a() with respect to noise q as matrix, inline function, function handle or name of function in form W(x,param) (optional, default identity)
	`param`	Parameters of a. Parameters should be a single cell array, vector or a matrix containing the same parameters for each step or if different parameters are used on each step they must be a cell array of the format { param_1, param_2, ...}, where param_x contains the parameters for step x as a cell array, a vector or a matrix. (optional, default empty)
	`same_p`	1 if the same parameters should be used on every time step (optional, default 1)
Output:	`K`	Smoothed state mean sequence
	`P`	Smoothed state covariance sequence
	`D`	Smoother gain sequence

etf_smooth1

etf_smooth1 Two filter nonlinear smoother algorithm. Calculate "smoothed" sequence from given extended Kalman filter output sequence by conditioning all steps to all measurements.		
Syntax:	`[M,P] = ETF_SMOOTH1(M,P,Y,A,Q,ia,W,aparam, H,R,h,V,hparam,same_p_a,same_p_h)`	
Input:	`M`	NxK matrix of K mean estimates from Kalman filter
	`P`	NxNxK matrix of K state covariances from Kalman Filter
	`Y`	Measurement vector
	`A`	Derivative of a() with respect to state as matrix, inline function, function handle or name of function in form A(x,param) (optional, default eye())
	`Q`	Process noise of discrete model (optional, default zero)
	`ia`	Inverse prediction function as vector, inline function, function handle or name of function in form ia(x,param) (optional, default inv(A(x))*X)
	`W`	Derivative of a() with respect to noise q as matrix, inline function, function handle or name of function in form W(x,param) (optional, default identity)
	`aparam`	Parameters of a. Parameters should be a single cell array, vector or a matrix containing the same parameters for each step or if different parameters are used on each step they must be a cell array of the format { param_1, param_2, ...}, where param_x contains the parameters for step x as a cell array, a vector or a matrix. (optional, default empty)
	`H`	Derivative of h() with respect to state as matrix, inline function, function handle or name of function in form H(x,param)
	`R`	Measurement noise covariance.
	`h`	Mean prediction (measurement model) as vector, inline function, function handle or name of function in form h(x,param). (optional, default H(x)*X)
	`V`	Derivative of h() with respect to noise as matrix, inline function, function handle or name of function in form V(x,param). (optional, default identity)
	`hparam`	Parameters of h. See the description of aparam for the format of parameters. (optional, default aparam)
	`same_p_a`	If 1 uses the same parameters on every time step for a (optional, default 1)
	`same_p_h`	If 1 uses the same parameters on every time step for h (optional, default 1)
Output:	`M`	Smoothed state mean sequence
	`P`	Smoothed state covariance sequence

5.1.3 Unscented Kalman filter

ukf_predict1

ukf_predict1 Perform additive form Unscented Kalman Filter prediction step.		
Syntax:	`[M,P] = UKF_PREDICT1(M,P,` `[a,Q,param,alpha,beta,kappa,mat])`	
Input:	`M`	Nx1 mean state estimate of previous step
	`P`	NxN state covariance of previous step
	`a`	Dynamic model function as a matrix A defining linear function a(x) = A*x, inline function, function handle or name of function in form a(x,param) (optional, default eye())
	`Q`	Process noise of discrete model (optional, default zero)
	`param`	Parameters of a (optional, default empty)
	`alpha`	Transformation parameter (optional)
	`beta`	Transformation parameter (optional)
	`kappa`	Transformation parameter (optional)
	`mat`	If 1 uses matrix form (optional, default 0)
Output:	`M`	Updated state mean
	`P`	Updated state covariance

ukf_predict2

ukf_predict2 Perform augmented form Unscented Kalman Filter prediction step for model		
Syntax:	`[M,P] = UKF_PREDICT2(M,P,a,Q,` `[param,alpha,beta,kappa])`	
Input:	`M`	Nx1 mean state estimate of previous step
	`P`	NxN state covariance of previous step
	`a`	Dynamic model function as inline function, function handle or name of function in form a([x;w],param)
	`Q`	Non-singular covariance of process noise w
	`param`	Parameters of a (optional, default empty)
	`alpha`	Transformation parameter (optional)
	`beta`	Transformation parameter (optional)
	`kappa`	Transformation parameter (optional)
	`mat`	If 1 uses matrix form (optional, default 0)
Output:	`M`	Updated state mean
	`P`	Updated state covariance

ukf_predict3

ukf_predict3		
Perform augmented form Unscented Kalman Filter prediction step for model		
Syntax:	`[M,P,X,w] = UKF_PREDICT3(M,P,a,Q,R, [param,alpha,beta,kappa])`	
Input:	`M`	Nx1 mean state estimate of previous step
	`P`	NxN state covariance of previous step
	`a`	Dynamic model function as inline function, function handle or name of function in form a([x;w],param)
	`Q`	Non-singular covariance of process noise w
	`R`	Measurement covariance.
	`param`	Parameters of a (optional, default empty)
	`alpha`	Transformation parameter (optional)
	`beta`	Transformation parameter (optional)
	`kappa`	Transformation parameter (optional)
	`mat`	If 1 uses matrix form (optional, default 0)
Output:	`M`	Updated state mean
	`P`	Updated state covariance
	`X`	Sigma points of x
	`w`	Weights as cell array {mean-weights,cov-weights,c}

ukf_update1

ukf_update1		
Perform additive form Discrete Unscented Kalman Filter (UKF) measurement update step. Assumes additive measurement noise.		
Syntax:	`[M,P,K,MU,S,LH] = UKF_UPDATE1(M,P,Y,h,R,param, alpha,beta,kappa,mat)`	
Input:	`M`	Mean state estimate after prediction step
	`P`	State covariance after prediction step
	`Y`	Measurement vector.
	`h`	Measurement model function as a matrix H defining linear function h(x) = H*x, inline function, function handle or name of function in form h(x,param)
	`R`	Measurement covariance.
	`param`	Parameters of a (optional, default empty)
	`alpha`	Transformation parameter (optional)
	`beta`	Transformation parameter (optional)
	`kappa`	Transformation parameter (optional)
	`mat`	If 1 uses matrix form (optional, default 0)
Output:	`M`	Updated state mean
	`P`	Updated state covariance
	`K`	Computed Kalman gain
	`MU`	Predictive mean of Y
	`S`	Predictive covariance Y
	`LH`	Predictive probability (likelihood) of measurement.

ukf_update2

ukf_update2		
Perform augmented form Discrete Unscented Kalman Filter (UKF) measurement update step. Assumes additive measurement noise.		
Syntax:	`[M,P,K,MU,IS,LH] = UKF_UPDATE2(M,P,Y,h,R,param, alpha,beta,kappa,mat)`	
Input:	`M`	Mean state estimate after prediction step
	`P`	State covariance after prediction step
	`Y`	Measurement vector.
	`h`	Measurement model function as a matrix H defining linear function h(x) = H*x+r, inline function, function handle or name of function in form h([x;r],param)
	`R`	Measurement covariance.
	`param`	Parameters of a (optional, default empty)
	`alpha`	Transformation parameter (optional)
	`beta`	Transformation parameter (optional)
	`kappa`	Transformation parameter (optional)
	`mat`	If 1 uses matrix form (optional, default 0)
Output:	`M`	Updated state mean
	`P`	Updated state covariance
	`K`	Computed Kalman gain
	`MU`	Predictive mean of Y
	`S`	Predictive covariance Y
	`LH`	Predictive probability (likelihood) of measurement.

ukf_update3

ukf_update3		
Perform augmented form Discrete Unscented Kalman Filter (UKF) measurement update step. Assumes additive measurement noise.		
Syntax:	`[M,P,K,MU,IS,LH] = UKF_UPDATE3(M,P,Y,h,R,X,w,param, alpha,beta,kappa,mat,sigmas)`	
Input:	`M`	Mean state estimate after prediction step
	`P`	State covariance after prediction step
	`Y`	Measurement vector.
	`h`	Measurement model function as a matrix H defining linear function h(x) = H*x+r, inline function, function handle or name of function in form h([x;r],param)
	`R`	Measurement covariance.
	`X`	Sigma points of x
	`w`	Weights as cell array {mean-weights,cov-weights,c}
	`param`	Parameters of a (optional, default empty)
	`alpha`	Transformation parameter (optional)
	`beta`	Transformation parameter (optional)
	`kappa`	Transformation parameter (optional)
	`mat`	If 1 uses matrix form (optional, default 0)
Output:	`M`	Updated state mean
	`P`	Updated state covariance
	`K`	Computed Kalman gain
	`MU`	Predictive mean of Y
	`S`	Predictive covariance Y
	`LH`	Predictive probability (likelihood) of measurement.

urts_smooth1

urts_smooth1		
Syntax:	`[M,P,D] = URTS_SMOOTH1(M,P,a,Q, [param,alpha,beta,kappa,mat,same_p])`	
Input:	`M`	NxK matrix of K mean estimates from Unscented Kalman filter
	`P`	NxNxK matrix of K state covariances from Unscented Kalman Filter
	`a`	Dynamic model function as a matrix A defining linear function a(x) = A*x, inline function, function handle or name of function in form a(x,param) (optional, default eye())
	`Q`	NxN process noise covariance matrix or NxNxK matrix of K state process noise covariance matrices for each step.
	`param`	Parameters of a. Parameters should be a single cell array, vector or a matrix containing the same parameters for each step, or if different parameters are used on each step they must be a cell array of the format { param_1, param_2, ...}, where param_x contains the parameters for step x as a cell array, a vector or a matrix. (optional, default empty)
	`alpha`	Transformation parameter (optional)
	`beta`	Transformation parameter (optional)
	`kappa`	Transformation parameter (optional)
	`mat`	If 1 uses matrix form (optional, default 0)
	`same_p`	If 1 uses the same parameters on every time step (optional, default 1)

urts_smooth2

urts_smooth2 Unscented Rauch-Tung-Striebel smoother algorithm. Calculate "smoothed" sequence from given Kalman filter output sequence by conditioning all steps to all measurements.		
Syntax:	`[M,P,S] = URTS_SMOOTH2(M,P,a,Q, [param,alpha,beta,kappa,mat,same_p])`	
Input:	`M`	NxK matrix of K mean estimates from Unscented Kalman filter
	`P`	NxNxK matrix of K state covariances from Unscented Kalman Filter
	`a`	Dynamic model function as inline function, function handle or name of function in form a([x;w],param)
	`Q`	Non-singular covariance of process noise w
	`param`	Parameters of a. Parameters should be a single cell array, vector or a matrix containing the same parameters for each step, or if different parameters are used on each step they must be a cell array of the format { param_1, param_2, ...}, where param_x contains the parameters for step x as a cell array, a vector or a matrix. (optional, default empty)
	`alpha`	Transformation parameter (optional)
	`beta`	Transformation parameter (optional)
	`kappa`	Transformation parameter (optional)
	`mat`	If 1 uses matrix form (optional, default 0)
	`same_p`	If 1 uses the same parameters on every time step (optional, default 1)
Output:	`K`	Smoothed state mean sequence
	`P`	Smoothed state covariance sequence
	`D`	Smoother gain sequence

utf_smooth1

utf_smooth1
Two filter nonlinear smoother algorithm. Calculate "smoothed" sequence from given extended Kalman filter output sequence by conditioning all steps to all measurements.

Syntax:
```
[M,P] = UTF_SMOOTH1(M,P,Y,
[ia,Q,aparam,h,R,hparam,
alpha,beta,kappa,mat,same_p_a,same_p_h])
```

Input:

M	NxK matrix of K mean estimates from Kalman filter
P	NxNxK matrix of K state covariances from Kalman Filter
Y	Measurement vector
ia	Inverse prediction as a matrix IA defining linear function ia(xw) = IA*xw, inline function, function handle or name of function in form ia(xw,param) (optional, default eye())
Q	Process noise of discrete model (optional, default zero)
aparam	Parameters of a (optional, default empty)
h	Measurement model function as a matrix H defining linear function h(x) = H*x, inline function, function handle or name of function in form h(x,param)
R	Measurement noise covariance.
hparam	Parameters of h (optional, default aparam)
alpha	Transformation parameter (optional)
beta	Transformation parameter (optional)
kappa	Transformation parameter (optional)
mat	If 1 uses matrix form (optional, default 0)
same_p_a	If 1 uses the same parameters on every time step for a (optional, default 1)
same_p_h	If 1 uses the same parameters on every time step for h (optional, default 1)

Output:

M	Smoothed state mean sequence
P	Smoothed state covariance sequence

ut_transform

ut_transform ... For default values of parameters, see UT_WEIGHTS.		
Syntax:	`[mu,S,C,X,Y,w] = UT_TRANSFORM(M,P,g,[param,alpha,beta,kappa,mat],n,X,w)`	
Input:	`M`	Random variable mean (Nx1 column vector)
	`P`	Random variable covariance (NxN pos.def. matrix)
	`g`	Transformation function of the form g(x,param) as matrix, inline function, function name or function reference
	`param`	Parameters of g (optional, default empty)
	`alpha`	Transformation parameter (optional)
	`beta`	Transformation parameter (optional)
	`kappa`	Transformation parameter (optional)
	`mat`	If 1 uses matrix form (optional, default 0)
	`X`	Sigma points of x
	`w`	Weights as cell array {mean-weights,cov-weights,c}
Output:	`mu`	Estimated mean of y
	`S`	Estimated covariance of y
	`C`	Estimated cross-covariance of x and y
	`X`	Sigma points of x
	`Y`	Sigma points of y
	`w`	Weights as cell array {mean-weights,cov-weights,c}

ut_weights

ut_weights Computes unscented transformation weights.		
Syntax:	`[WM,WC,c] = ut_weights(n,alpha,beta,kappa)`	
Input:	`n`	Dimensionality of random variable
	`alpha`	Transformation parameter (optional, default 0.5)
	`beta`	Transformation parameter (optional, default 2)
	`kappa`	Transformation parameter (optional, default 3-n)
Output:	`WM`	Weights for mean calculation
	`WC`	Weights for covariance calculation
	`c`	Scaling constant

ut_mweights

ut_mweights		
Computes matrix form unscented transformation weights.		
Syntax:	`[WM,W,c] = ut_mweights(n,alpha,beta,kappa)`	
Input:	`n`	Dimensionality of random variable
	`alpha`	Transformation parameter (optional, default 0.5)
	`beta`	Transformation parameter (optional, default 2)
	`kappa`	Transformation parameter (optional, default 3-size(X,1))
Output:	`WM`	Weight vector for mean calculation
	`W`	Weight matrix for covariance calculation
	`c`	Scaling constant

ut_sigmas

ut_sigmas		
Generates sigma points and associated weights for Gaussian initial distribution N(M,P). For default values of parameters alpha, beta and kappa see UT_WEIGHTS.		
Syntax:	`X = ut_sigmas(M,P,c);`	
Input:	`M`	Initial state mean (Nx1 column vector)
	`P`	Initial state covariance
	`c`	Parameter returned by UT_WEIGHTS
Output:	`X`	Matrix where 2N+1 sigma points are as columns

5.1.4 Cubature Kalman Filter

ckf_transform

ckf_transform		
Syntax:	`[mu,S,C,SX,W] = CKF_TRANSFORM(M,P,g,param)`	
Input:	`M`	Random variable mean (Nx1 column vector)
	`P`	Random variable covariance (NxN pos.def. matrix)
	`g`	Transformation function of the form g(x,param) as matrix, inline function, function name or function reference
	`param`	Parameters of g (optional, default empty)
Output:	`mu`	Estimated mean of y
	`S`	Estimated covariance of y
	`C`	Estimated cross-covariance of x and y
	`SX`	Sigma points of x
	`W`	Weights as cell array

ckf_update

ckf_update		
Perform additive form spherical-radial cubature Kalman filter (CKF) measurement update step. Assumes additive measurement noise.		
Syntax:	`[M,P,K,MU,S,LH] = CKF_UPDATE(M,P,Y,h,R,param)`	
Input:	`M`	Mean state estimate after prediction step
	`P`	State covariance after prediction step
	`Y`	Measurement vector.
	`h`	Measurement model function as a matrix H defining linear function h(x) = H*x, inline function, function handle or name of function in form h(x,param)
	`R`	Measurement covariance.
	`param`	Parameters of h.
Output:	`M`	Updated state mean
	`P`	Updated state covariance
	`K`	Computed Kalman gain
	`MU`	Predictive mean of Y
	`S`	Predictive covariance Y
	`LH`	Predictive probability (likelihood) of measurement.

crts_smooth

crts_smooth Cubature Rauch-Tung-Striebel smoother algorithm. Calculate "smoothed" sequence from given Kalman filter output sequence by conditioning all steps to all measurements. Uses the spherical- radial cubature rule.		
Syntax:	`[M,P,D] = CKF_SMOOTH(M,P,a,Q, [param,same_p])`	
Input:	`M`	NxK matrix of K mean estimates from Cubature Kalman filter
	`P`	NxNxK matrix of K state covariances from Cubature Kalman Filter
	`a`	Dynamic model function as a matrix A defining linear function a(x) = A*x, inline function, function handle or name of function in form a(x,param) (optional, default eye())
	`Q`	NxN process noise covariance matrix or NxNxK matrix of K state process noise covariance matrices for each step.
	`param`	Parameters of a. Parameters should be a single cell array, vector or a matrix containing the same parameters for each step, or if different parameters are used on each step they must be a cell array of the format { param_1, param_2, ...}, where param_x contains the parameters for step x as a cell array, a vector or a matrix. (optional, default empty)
	`same_p`	If 1 uses the same parameters on every time step (optional, default 1)
Output:	`M`	Smoothed state mean sequence
	`P`	Smoothed state covariance sequence
	`D`	Smoother gain sequence

sphericalradial

sphericalradial		
Apply the spherical-radial cubature rule to integrals of form: int f(x) N(x | m,P) dx		
Syntax:	`[I,x,W,F] = sphericalradial(f,m,P[,param])`	
Input:	`f`	Function f(x,param) as inline, name or reference
	`m`	Mean of the d-dimensional Gaussian distribution
	`P`	Covariance of the Gaussian distribution
	`param`	Parameters for the function (optional)
Output:	`I`	The integral
	`x`	Evaluation points
	`W`	Weights
	`F`	Function values

5.1.5 Gauss-Hermite Kalman Filter

gh_packed_pc

gh_packed_pc		
Packs the integrals that need to be evaluated in nice function form to ease the evaluation. Evaluates P = (f-fm)(f-fm)' and C = (x-m)(f-fm)'.		
Syntax:	`pc = GH_PACKED_PC(x,fmmparam)`	
Input:	`x`	Evaluation point
	`fmmparam`	Array of handles and parameters to form the functions.
Output:	`pc`	Output values

gh_transform

gh_transform		
Syntax:	`[mu,S,C,SX,W] = GH_TRANSFORM(M,P,g,p,param)`	
Input:	`M`	Random variable mean (Nx1 column vector)
	`P`	Random variable covariance (NxN pos.def. matrix)
	`g`	Transformation function of the form g(x,param) as matrix, inline function, function name or function reference
	`p`	Number of points in Gauss-Hermite integration
	`param`	Parameters of g (optional, default empty)
Output:	`mu`	Estimated mean of y
	`S`	Estimated covariance of y
	`C`	Estimated cross-covariance of x and y
	`SX`	Sigma points of x
	`W`	Weights as cell array

ghkf_predict

ghkf_predict Perform additive form Gauss-Hermite Kalman Filter prediction step.		
Syntax:	`[M,P] = GHKF_PREDICT(M,P, [f,Q,param,p])`	
Input:	`M`	Nx1 mean state estimate of previous step
	`P`	NxN state covariance of previous step
	`f`	Dynamic model function as a matrix A defining linear function f(x) = A*x, inline function, function handle or name of function in form f(x,param) (optional, default eye())
	`Q`	Process noise of discrete model (optional, default zero)
	`param`	Parameters of f (optional, default empty)
	`p`	Degree of approximation (number of quadrature points)
Output:	`M`	Updated state mean
	`P`	Updated state covariance

ghkf_update

ghkf_update Perform additive form Gauss-Hermite Kalman filter (GHKF) measurement update step. Assumes additive measurement noise.		
Syntax:	`[M,P,K,MU,S,LH] = GHKF_UPDATE(M,P,Y,h,R,param,p)`	
Input:	`M`	Mean state estimate after prediction step
	`P`	State covariance after prediction step
	`Y`	Measurement vector.
	`h`	Measurement model function as a matrix H defining linear function h(x) = H*x, inline function, function handle or name of function in form h(x,param)
	`R`	Measurement covariance
	`param`	Parameters of h
	`p`	Degree of approximation (number of quadrature points)
Output:	`M`	Updated state mean
	`P`	Updated state covariance
	`K`	Computed Kalman gain
	`MU`	Predictive mean of Y
	`S`	Predictive covariance Y
	`LH`	Predictive probability (likelihood) of measurement.

ghrts_smooth

ghrts_smooth		
Gauss-Hermite Rauch-Tung-Striebel smoother algorithm. Calculate "smoothed" sequence from given Kalman filter output sequence by conditioning all steps to all measurements.		
Syntax:	`[M,P,D] = GHRTS_SMOOTH(M,P,f,Q, [param,p,same_p])`	
Input:	`M`	NxK matrix of K mean estimates from Gauss-Hermite Kalman filter
	`P`	NxNxK matrix of K state covariances from Gauss-Hermite filter
	`f`	Dynamic model function as a matrix A defining linear function f(x) = A*x, inline function, function handle or name of function in form a(x,param) (optional, default eye())
	`Q`	NxN process noise covariance matrix or NxNxK matrix of K state process noise covariance matrices for each step.
	`param`	Parameters of f(.). Parameters should be a single cell array, vector or a matrix containing the same parameters for each step, or if different parameters are used on each step they must be a cell array of the format { param_1, param_2, ...}, where param_x contains the parameters for step x as a cell array, a vector or a matrix. (optional, default empty)
	`p`	Degree on approximation (number of quadrature points)
	`same_p`	If set to '1' uses the same parameters on every time step (optional, default 1)
Output:	`M`	Smoothed state mean sequence
	`P`	Smoothed state covariance sequence
	`D`	Smoother gain sequence

hermitepolynomial

hermitepolynomial		
Forms the Hermite polynomial of order n.		
Syntax:	`p = hermitepolynomial(n)`	
Input:	`n`	Polynomial order
Output:	`p`	Polynomial coefficients (starting from greatest order)

ngausshermi

ngausshermi		
Approximates a Gaussian integral using the Gauss-Hermite method in multiple dimensions: int f(x) N(x | m,P) dx		
Syntax:	`[I,x,W,F] = ngausshermi(f,p,m,P,param)`	
Input:	`f`	Function f(x,param) as inline, name or reference
	`n`	Polynomial order
	`m`	Mean of the d-dimensional Gaussian distribution
	`P`	Covariance of the Gaussian distribution
	`param`	Optional parameters for the function
Output:	`I`	The integral value
	`x`	Evaluation points
	`W`	Weights
	`F`	Function values

5.2 Multiple Model Systems

5.2.1 IMM Models

imm_filter

imm_filter		
IMM filter prediction and update steps. Use this instead of separate prediction and update functions, if you don't need the prediction estimates.		
Syntax:	`[X_i,P_i,MU,X,P] = IMM_FILTER(X_ip,P_ip,MU_ip,p_ij, ind,dims,A,Q,Y,H,R)`	
Input:	`X_ip`	Cell array containing $N\hat{j}$ x 1 mean state estimate vector for each model j after update step of previous time step
	`P_ip`	Cell array containing $N\hat{j}$ x $N\hat{j}$ state covariance matrix for each model j after update step of previous time step
	`MU_ip`	Vector containing the model probabilities at previous time step
	`p_ij`	Model transition matrix
	`ind`	Indices of state components for each model as a cell array
	`dims`	Total number of different state components in the combined system
	`A`	State transition matrices for each model as a cell array.
	`Q`	Process noise matrices for each model as a cell array.
	`Y`	Dx1 measurement vector.
	`H`	Measurement matrices for each model as a cell array.
	`R`	Measurement noise covariances for each model as a cell array.
Output:	`X_p`	Updated state mean for each model as a cell array
	`P_p`	Updated state covariance for each model as a cell array
	`MU`	Model probabilities as vector
	`X`	Combined state mean estimate
	`P`	Combined state covariance estimate

imm_predict

imm_predict		
IMM filter prediction step.		
Syntax:	`[X_p,P_p,c_j,X,P] = IMM_PREDICT(X_ip,P_ip,MU_ip,p_ij,ind,dims,A,Q)`	
Input:	`X_ip`	Cell array containing $N\hat{j}$ x 1 mean state estimate vector for each model j after update step of previous time step
	`P_ip`	Cell array containing $N\hat{j}$ x $N\hat{j}$ state covariance matrix for each model j after update step of previous time step
	`MU_ip`	Vector containing the model probabilities at previous time step
	`p_ij`	Model transition probability matrix
	`ind`	Indexes of state components for each model as a cell array
	`dims`	Total number of different state components in the combined system
	`A`	State transition matrices for each model as a cell array.
	`Q`	Process noise matrices for each model as a cell array.
Output:	`X_p`	Predicted state mean for each model as a cell array
	`P_p`	Predicted state covariance for each model as a cell array
	`c_j`	Normalizing factors for mixing probabilities
	`X`	Combined predicted state mean estimate
	`P`	Combined predicted state covariance estimate

imm_smooth

imm_smooth		
Two filter fixed-interval IMM smoother.		
Syntax:	`[X_S,P_S,X_IS,P_IS,MU_S] = IMM_SMOOTH(MM,PP,MM_i,PP_i, MU,p_ij,mu_0j,ind,dims,A,Q,R,H,Y)`	
Input:	`MM`	NxK matrix containing the means of forward-time IMM-filter on each time step
	`PP`	NxNxK matrix containing the covariances of forward-time IMM-filter on each time step
	`MM_i`	Model-conditional means of forward-time IMM-filter on each time step as a cell array
	`PP_i`	Model-conditional covariances of forward-time IMM-filter on each time step as a cell array
	`MU`	Model probabilities of forward-time IMM-filter on each time step
	`p_ij`	Model transition probability matrix
	`mu_0j`	Prior model probabilities
	`ind`	Indices of state components for each model as a cell array
	`dims`	Total number of different state components in the combined system
	`A`	State transition matrices for each model as a cell array.
	`Q`	Process noise matrices for each model as a cell array.
	`R`	Measurement noise matrices for each model as a cell array.
	`H`	Measurement matrices for each model as a cell array
	`Y`	Measurement sequence
Output:	`X_S`	Smoothed state means for each time step
	`P_S`	Smoothed state covariances for each time step
	`X_IS`	Model-conditioned smoothed state means for each time step
	`P_IS`	Model-conditioned smoothed state covariances for each time step
	`MU_S`	Smoothed model probabilities for each time step

imm_update

imm_update		
IMM filter measurement update step.		
Syntax:	`[X_i,P_i,MU,X,P] = IMM_UPDATE(X_p,P_p,c_j,ind,dims,Y,H,R)`	
Input:	`X_p`	Cell array containing $N\hat{j}$ x 1 mean state estimate vector for each model j after prediction step
	`P_p`	Cell array containing $N\hat{j}$ x $N\hat{j}$ state covariance matrix for each model j after prediction step
	`c_j`	Normalizing factors for mixing probabilities
	`ind`	Indices of state components for each model as a cell array
	`dims`	Total number of different state components in the combined system
	`Y`	Dx1 measurement vector.
	`H`	Measurement matrices for each model as a cell array.
	`R`	Measurement noise covariances for each model as a cell array.
Output:	`X_i`	Updated state mean estimate for each model as a cell array
	`P_i`	Updated state covariance estimate for each model as a cell array
	`MU`	Estimated probabilities of each model
	`X`	Combined state mean estimate
	`P`	Combined state covariance estimate

5.2.2 EIMM Models

eimm_predict

eimm_predict IMM-EKF filter prediction step. If some of the models have linear dynamics standard Kalman filter prediction step is used for those.		
Syntax:	`[X_p,P_p,c_j,X,P] = EIMM_PREDICT(X_ip,P_ip, MU_ip,p_ij,ind,dims,A,a,param,Q)`	
Input:	`X_ip`	Cell array containing $N\hat{j}$ x 1 mean state estimate vector for each model j after update step of previous time step
	`P_ip`	Cell array containing $N\hat{j}$ x $N\hat{j}$ state covariance matrix for each model j after update step of previous time step
	`MU_ip`	Vector containing the model probabilities at previous time step
	`p_ij`	Model transition matrix
	`ind`	Indices of state components for each model as a cell array
	`dims`	Total number of different state components in the combined system
	`A`	Dynamic model matrices for each linear model and Jacobians of each non-linear model's measurement model function as a cell array
	`a`	Function handles of dynamic model functions for each model as a cell array
	`param`	Parameters of a for each model as a cell array
	`Q`	Process noise matrices for each model as a cell array.
Output:	`X_p`	Predicted state mean for each model as a cell array
	`P_p`	Predicted state covariance for each model as a cell array
	`c_j`	Normalizing factors for mixing probabilities
	`X`	Combined predicted state mean estimate
	`P`	Combined predicted state covariance estimate

eimm_smooth

eimm_smooth		
EKF based two-filter fixed-interval IMM smoother.		
Syntax:	`[X_S,P_S,X_IS,P_IS,MU_S] = EIMM_SMOOTH(MM,PP, MM_i,PP_i,MU,p_ij,mu_0j,ind,dims, A,a,a_param,Q,R,H,h,h_param,Y)`	
Input:	`MM`	Means of forward-time IMM-filter on each time step
	`PP`	Covariances of forward-time IMM-filter on each time step
	`MM_i`	Model-conditional means of forward-time IMM-filter on each time step
	`PP_i`	Model-conditional covariances of forward-time IMM-filter on each time step
	`MU`	Model probabilities of forward-time IMM-filter on each time step
	`p_ij`	Model transition probability matrix
	`ind`	Indices of state components for each model as a cell array
	`dims`	Total number of different state components in the combined system
	`A`	Dynamic model matrices for each linear model and Jacobians of each non-linear model's measurement model function as a cell array
	`a`	Cell array containing function handles for dynamic functions for each model having non-linear dynamics
	`a_param`	Parameters of a as a cell array.
	`Q`	Process noise matrices for each model as a cell array.
	`R`	Measurement noise matrices for each model as a cell array.
	`H`	Measurement matrices for each linear model and Jacobians of each non-linear model's measurement model function as a cell array
	`h`	Cell array containing function handles for measurement functions for each model having non-linear measurements
	`h_param`	Parameters of h as a cell array.
	`Y`	Measurement sequence
Output:	`X_S`	Smoothed state means for each time step
	`P_S`	Smoothed state covariances for each time step
	`X_IS`	Model-conditioned smoothed state means for each time step
	`P_IS`	Model-conditioned smoothed state covariances for each time step
	`MU_S`	Smoothed model probabilities for each time step

eimm_update

eimm_update		
IMM-EKF filter measurement update step. If some of the models have linear measurements standard Kalman filter update step is used for those.		
Syntax:	`[X_i,P_i,MU,X,P] = IMM_UPDATE(X_p,P_p,c_j,ind,dims,Y,H,h,R,param)`	
Input:	`X_p`	Cell array containing $\hat{Nj}$ x 1 mean state estimate vector for each model j after prediction step
	`P_p`	Cell array containing $\hat{Nj}$ x $\hat{Nj}$ state covariance matrix for each model j after prediction step
	`c_j`	Normalizing factors for mixing probabilities
	`ind`	Indices of state components for each model as a cell array
	`dims`	Total number of different state components in the combined system
	`Y`	Dx1 measurement vector.
	`H`	Measurement matrices for each linear model and Jacobians of each non-linear model's measurement model function as a cell array
	`h`	Cell array containing function handles for measurement functions for each model having non-linear measurements
	`R`	Measurement noise covariances for each model as a cell array.
	`param`	Parameters of h
Output:	`X_i`	Updated state mean estimate for each model as a cell array
	`P_i`	Updated state covariance estimate for each model as a cell array
	`MU`	Estimated probabilities of each model
	`X`	Combined updated state mean estimate
	`P`	Combined updated covariance estimate

5.2.3 UIMM Models

uimm_predict

uimm_predict		
IMM-UKF filter prediction step. If some of the models have linear dynamics standard Kalman filter prediction step is used for those.		
Syntax:	`[X_p,P_p,c_j,X,P] = UIMM_PREDICT(X_ip,P_ip, MU_ip,p_ij,ind,dims,A,a,param,Q)`	
Input:	`X_ip`	Cell array containing $\hat{Nj}$ x 1 mean state estimate vector for each model j after update step of previous time step
	`P_ip`	Cell array containing $\hat{Nj}$ x $\hat{Nj}$ state covariance matrix for each model j after update step of previous time step
	`MU_ip`	Vector containing the model probabilities at previous time step
	`p_ij`	Model transition matrix
	`ind`	Indices of state components for each model as a cell array
	`dims`	Total number of different state components in the combined system
	`A`	Dynamic model matrices for each linear model as a cell array
	`a`	Dynamic model functions for each non-linear model
	`param`	Parameters of a
	`Q`	Process noise matrices for each model as a cell array.
Output:	`X_p`	Predicted state mean for each model as a cell array
	`P_p`	Predicted state covariance for each model as a cell array
	`c_j`	Normalizing factors for mixing probabilities
	`X`	Combined predicted state mean estimate
	`P`	Combined predicted state covariance estimate

uimm_smooth

uimm_smooth		
UKF based two-filter fixed-interval IMM smoother.		
Syntax:	`[X_S,P_S,X_IS,P_IS,MU_S] = UIMM_SMOOTH(MM,PP, MM_i,PP_i,MU,p_ij,mu_0j,ind,dims,A,a, a_param,Q,R,H,h,h_param,Y)`	
Input:	`MM`	Means of forward-time IMM-filter on each time step
	`PP`	Covariances of forward-time IMM-filter on each time step
	`MM_i`	Model-conditional means of forward-time IMM-filter on each time step
	`PP_i`	Model-conditional covariances of forward-time IMM-filter on each time step
	`MU`	Model probabilities of forward-time IMM-filter on each time step
	`p_ij`	Model transition probability matrix
	`ind`	Indices of state components for each model as a cell array
	`dims`	Total number of different state components in the combined system
	`A`	Dynamic model matrices for each linear model and Jacobians of each non-linear model's measurement model function as a cell array
	`a`	Cell array containing function handles for dynamic functions for each model having non-linear dynamics
	`a_param`	Parameters of a as a cell array.
	`Q`	Process noise matrices for each model as a cell array.
	`R`	Measurement noise matrices for each model as a cell array.
	`H`	Measurement matrices for each linear model and Jacobians of each non-linear model's measurement model function as a cell array
	`h`	Cell array containing function handles for measurement functions for each model having non-linear measurements
	`h_param`	Parameters of h as a cell array.
	`Y`	Measurement sequence
Output:	`X_S`	Smoothed state means for each time step
	`P_S`	Smoothed state covariances for each time step
	`X_IS`	Model-conditioned smoothed state means for each time step
	`P_IS`	Model-conditioned smoothed state covariances for each time step
	`MU_S`	Smoothed model probabilities for each time step

uimm_update

uimm_update		
IMM-UKF filter measurement update step. If some of the models have linear measurements standard Kalman filter update step is used for those.		
Syntax:	`[X_i,P_i,MU,X,P] = IMM_UPDATE(X_p,P_p,c_j,ind,dims,Y,H,R)`	
Input:	`X_p`	Cell array containing $N\hat{j}$ x 1 mean state estimate vector for each model j after prediction step
	`P_p`	Cell array containing $N\hat{j}$ x $N\hat{j}$ state covariance matrix for each model j after prediction step
	`c_j`	Normalizing factors for mixing probabilities
	`ind`	Indices of state components for each model as a cell array
	`dims`	Total number of different state components in the combined system
	`Y`	Dx1 measurement vector.
	`H`	Measurement matrices for each model as a cell array.
	`h`	Measurement mean
	`param`	parameters
	`R`	Measurement noise covariances for each model as a cell array.
Output:	`X_i`	Updated state mean estimate for each model as a cell array
	`P_i`	Updated state covariance estimate for each model as a cell array
	`MU`	Probabilities of each model
	`X`	Combined state mean estimate
	`P`	Combined state covariance estimate

5.3 Other Functions

der_check

der_check		
Evaluates function derivative analytically and using finite differences. If no output arguments are given, issues a warning if these two values differ too much.		
Syntax:	`[D0,D1] = DER_CHECK(F,DF,INDEX, [P1,P2,P3,...])`	
Input:	`F`	Name of actual function or inline function in form F(P1,P2,...)
	`DF`	Derivative value as matrix, name of derivative function or inline function in form DF(P1,P2,...).
	`INDEX`	Index of parameter of interest. DF should Calculate the derivative with recpect to parameter Pn, where n is the index.
Output:	`D0`	Actual derivative
	`D1`	Estimated derivative

lti_disc

lti_disc		
Discretize LTI ODE with Gaussian Noise. The original ODE model is in form		
Syntax:	`[A,Q] = lti_disc(F,L,Qc,dt)`	
Input:	`F`	NxN Feedback matrix
	`L`	NxL Noise effect matrix (optional, default identity)
	`Qc`	LxL Diagonal Spectral Density (optional, default zeros)
	`dt`	Time Step (optional, default 1)
Output:	`A`	Transition matrix
	`Q`	Discrete Process Covariance

lti_int

lti_int	
Integrates LTI differential equation	
Syntax:	`[x,P,A] = lti_int(x,P,F,L,Q,T)`

schol

schol		
Compute lower triangular Cholesky factor L of symmetric positive semidefinite matrix A such that		
Syntax:	`[L,def] = schol(A)`	
Input:	`A`	Symmetric pos.semi.def matrix to be factorized
Output:	`L`	Lower triangular matrix such that A=L*L' if def>=0.
	`def`	Value 1,0,-1 denoting that A was positive definite, positive semidefinite or negative definite, respectively.

gauss_pdf

gauss_pdf		
Calculate values of PDF (Probability Density Function) of multivariate Gaussian distribution		
Syntax:	`[P,E] = GAUSS_PDF(X,M,S)`	
Input:	`X`	Dx1 value or N values as DxN matrix
	`M`	Dx1 mean of distibution or N values as DxN matrix.
	`S`	DxD covariance matrix
Output:	`P`	Probability of X.
	`E`	Negative logarithm of P

gauss_rnd

gauss_rnd		
Draw N samples from multivariate Gaussian distribution		
Syntax:	`X = GAUSS_RND(M,S,N)`	
Input:	`M`	Dx1 mean of distibution or K values as DxK matrix.
	`S`	DxD covariance matrix
	`N`	Number of samples (optional, default 1)
Output:	`X`	Dx(K*N) matrix of samples.

rk4

rk4 Perform one fourth order Runge-Kutta iteration step for differential equation		
Syntax:	`[x,Y] = rk4(f,dt,x, [P1,P2,P3,Y])`	
Input:	`f`	Name of function in form f(x,P(:)) or inline function taking the same parameters. In chained case the function should be f(x,y,P(:)).
	`dt`	Delta time as scalar.
	`x`	Value of x from the previous time step.
	`P1`	Values of parameters of the function at initial time t as a cell array (or single plain value). Defaults to empty array (no parameters).
	`P2`	Values of parameters of the function at time t+dt/2 as a cell array (or single plain value). Defaults to P1 and each empty (or missing) value in the cell array is replaced with the corresponding value in P1.
	`P3`	Values of parameters of the function at time t+dt. Defaults to P2 similarly to above.
	`Y`	Cell array of partial results y1,y2,y3,y4 in the RK algorithm of the second parameter in the interated function. This can be used for chaining the integrators. Defaults to empty.
Output:	`x`	Next value of X
	`Y`	Cell array of partial results in Runge-Kutta algorithm.

resampstr

resampstr	
Syntax:	