KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Data assimilation toolbox for Matlab

Wannes Van den Bossche

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
wiskundige ingenieurstechnieken

Promotoren:

Prof. dr. ir. Bart De Moor
Prof. dr. ir. Joos Vandewalle

Assessoren:

Prof. dr. ir. Raf Vandebril
Prof. dr. ir. Giovanni Samaey

Begeleiders:

Dr. ir. Oscar Mauricio Agudelo
Dr. ir. Maarten Breckpot

Academiejaar 2012 – 2013

Preface

To study again at the age of thirty has been one of the most satisfying experiences in my life. Although the road to this moment took a considerable effort and required certain sacrifices, I have never felt a single moment of regret. All the courses offered me enlightening experiences that changed my vision on engineering in numerous ways. That is why I first want to say thank you to all the professors and the assistants whose efforts made these great experiences possible.

This master thesis is the capstone of my two year journey. I would like to start by thanking professor Bart De Moor and professor Joos Vandewalle for making this thesis possible and for their inspiring courses that attracted my interest in the field of this thesis. I also want to thank my co-promotors Oscar Mauricio Agudelo and Maarten Breckpot for their daily guidance of my master thesis. I would like to express special thanks to Oscar Mauricio Agudelo for his enthusiasm towards this thesis and whose suggestions and reviews have led to a better manuscript.

I also would like to thank my assessors professor Raf Vandebril and professor Giovanni Samaey for their time to review my master thesis.

Finally, I want to show my appreciation to my parents, family and friends for their understanding and support during the past two years. And last but most importantly, I want to thank my beloved girlfriend Kim who convinced me to start this journey and whose unconditional support and understanding has been my greatest motivation to successfully finish this wonderful journey.

Wannes Van den Bossche

Contents

Preface i

Abstract iv

Samenvatting v

Inleiding . . . v
Algemene architectuur en werking . . . v
Besluit . . . vii

List of Figures and Tables viii

Glossary ix

1 Introduction 1

2 Data assimilation techniques 5

2.1 State Space models . . . 5
2.2 Recursive Bayesian State Estimation . . . 6
2.3 The Kalman Filter - KF . . . 7
2.4 The Extended Kalman Filter - EKF . . . 9
2.5 Optimal Interpolation - OI . . . 10
2.6 The Unscented Kalman Filter - UKF . . . 11
2.7 The Ensemble Kalman Filter - EnKF . . . 13
2.8 The Ensemble Square Root Filter - EnSRF . . . 15
2.9 The Ensemble Transform Kalman Filter - ETKF . . . 17
2.10 The Deterministic Ensemble Kalman Filter - DEnKF . . . 19
2.11 The Generic Particle Filter - PF-GEN . . . 21
2.12 The Sampling Importance Resampling Particle Filter - PF-SIR . . . 25
2.13 The Auxiliary Sampling Importance Resampling Particle Filter - PF-ASIR . . . 26
2.14 Conclusion . . . 28

3 Data Assimilation Toolbox 31

3.1 Introduction to the toolbox . . . 31
3.2 The general architecture . . . 32
3.2.1 Object-Oriented Programming in Matlab . . . 32
3.2.2 The data assimilation workflow . . . 32
3.2.3 The different classes . . . 35
3.2.4 Error handling . . . 36

3.3 The Gaussian noise models . . . 37
3.3.1 Linear Time Invariant Gaussian noise (nm_gauss_lti) . . . 37
3.3.2 Linear Time Variant Gaussian noise (nm_gauss_ltv) . . . 38
3.3.3 Function handle Gaussian noise (nm_gauss_handle) . . . 39
3.3.4 Gaussian noise model methods . . . 41
3.4 The state space models . . . 42
3.4.1 The discrete-time linear model (ss_DL) . . . 42
3.4.2 The discrete-time nonlinear model with additive noise (ss_DNL_AN) . . . 44
3.4.3 The discrete-time nonlinear model (ss_DNL) . . . 46
3.4.4 State space models methods . . . 47
3.5 The simulation model (sim_D) . . . 50
3.6 The data assimilation techniques . . . 52
3.7 The data assimilation model (dam_D) . . . 56
3.8 Conclusion . . . 58

4 Implementation examples **61**
4.1 The Van der Pol oscillator . . . 61
4.2 The Lorenz equations . . . 64
4.3 The tracking problem . . . 66
4.4 The scalar nonlinear system . . . 69

5 Conclusion **71**

A Installation and folder structure **73**

B Poster **75**

C Accompanying paper **77**

Bibliography **87**

Abstract

Data assimilation is the common name given to the techniques that combine numerical models and measurements in order to obtain an improved estimation of the state of a system. In data assimilation it is assumed that both models and measurements are subject to uncertainties that can be defined as a statistical distribution. The goal of data assimilation is to combine the knowledge of models, measurements and uncertainties to obtain a better estimation than either the measurements or the models alone can provide. The application of this technique arises in many fields such as weather forecasting, oceanography, space weather forecasting and air quality.

Although data assimilation is a highly active research domain, MATLAB does not have an official data assimilation toolbox. This is why the goal of this thesis has been to develop a generic data assimilation toolbox for MATLAB with at least five data assimilation schemes to improve the estimations of any given model as defined by the user. Not only is this initial goal achieved, it has been extended by providing a wider range of data assimilation schemes together with several possibilities to configure noise models and state space models. In addition, several tools to analyze the acquired data assimilation results are incorporated. By applying the MATLAB object-oriented programming approach, the toolbox not only provides the required generic behaviour, but also maintains a structured framework that can be easily extended with new algorithms and classes. Furthermore, the interface of the toolbox is intuitively straightforward since it resembles the ones of official MATLAB toolboxes.

The toolbox currently contains data assimilation techniques that range from the regular Kalman filter up to particles filters. More specifically the data assimilation toolbox is equipped with the following techniques: the Kalman Filter (KF), the Extended Kalman Filter (EKF), the Unscented Kalman Filter (UKF), the Ensemble Kalman Filter (EnKF), the Deterministic Ensemble Kalman Filter (DEnKF), the Ensemble Transform Kalman Filter (ETKF), the Ensemble Square Root Filter (EnSRF), the Optimal Interpolation (OI) technique, the Generic Particle filter (GEN), the Sampling Importance Resampling (SIR) Particle filter and the Auxiliary Sampling Importance Resampling (ASIR) Particle filter.

Samenvatting

Inleiding

Data-assimilatie is een algemene term voor het numerieke proces waarbij metingen optimaal geïntegreerd worden in modellen teneinde een betere schatting te verkrijgen van de werkelijkheid. Bij data-assimilatie wordt verondersteld dat zowel het model als de metingen onderhevig zijn aan storingen die voldoen aan bepaalde statistische verdelingen. Eén van de meest voorkomende toepassingen van data-assimilatie is meteorologie, maar er zijn nog talrijke andere toepassingen zoals oceanografie en het voorspellen van ruimteweer of luchtkwaliteit. Ondanks de toenemende populariteit van data-assimilatie is er geen officiële data-assimilatie toolbox aanwezig in MATLAB.

Deze masterproef stelt een nieuwe data-assimilatie toolbox voor die volledig ontwikkeld is in MATLAB. De toolbox biedt een generiek platform aan dat toelaat om data-assimilatie algoritmen toe te passen op eender welke, door de gebruiker gedefinieerde, functie. Door een object-georiënteerde aanpak biedt de toolbox een transparante en gestructureerde basis waaraan later eenvoudig nieuwe klassen en algoritmen kunnen toegevoegd worden. De toolbox beschikt momenteel over drie verschillende klassen voor het modelleren van Gaussiaanse ruis, drie verschillende klassen voor het modelleren van discrete dynamische systemen en elf verschillende data-assimilatie algoritmen. De geïmplementeerde algoritmen zijn de Kalman Filter (KF), de Extended Kalman Filter (EKF), de Unscented Kalman Filter (UKF), de Ensemble Kalman Filter (EnKF), de Deterministic Ensemble Kalman Filter (DEnKF), de Ensemble Transform Kalman Filter (ETKF), de Ensemble Square Root Filter (EnSRF), de Optimal Interpolation (OI) techniek, de Generic Particle filter (GEN), de Sampling Importance Resampling (SIR) Particle filter en de Auxiliary Sampling Importance Resampling (ASIR) Particle filter.

Algemene architectuur en werking

Zoals reeds vermeld beschikt de toolbox momenteel over drie verschillende toestandsmodellen voor discrete dynamische systemen. Alle modellen bevatten de vectoren $x_k \in \mathbb{R}^n$, $y_k \in \mathbb{R}^m$ and $u_k \in \mathbb{R}^p$ die respectievelijk de toestandsvector, de metingsvector en de ingangsvector van stap k voorstellen. De vectoren $w_k \in \mathbb{R}^n$ en $v_k \in \mathbb{R}^m$

stellen de proces- en metingsruis voor waarvan de stochastische processen $\{w_k\}_{k=0}^{\infty}$ en $\{v_k\}_{k=0}^{\infty}$ afkomstig zijn van een gekende distributie. De drie verschillende modellen zijn:

- Het lineaire model met vergelijkingen

$$\begin{aligned} x_{k+1} &= A_k x_k + B_k u_k + w_k, \\ y_k &= C_k x_k + D_k u_k + v_k, \end{aligned}$$

  waar $A_k \in \mathbb{R}^{n\times n}$, $B_k \in \mathbb{R}^{n\times p}$, $C_k \in \mathbb{R}^{m\times n}$ en $D_k \in \mathbb{R}^{m\times p}$ de systeemmatrices zijn van stap k.

- Het niet-lineaire model met additieve ruis met vergelijkingen

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, k) + w_k, \\ y_k &= h(x_k, u_k, k) + v_k, \end{aligned}$$

  waar zowel de vergelijking van het systeem $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^n$ als de vergelijking van de meting $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^m$ mogelijk niet-lineaire functies zijn.

- Het niet-lineaire model met vergelijkingen

$$x_{k+1} = f(x_k, u_k, w_k, k), \tag{1}$$
$$y_k = h(x_k, u_k, v_k, k), \tag{2}$$

  waar zowel de vergelijking van het systeem $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$ als de vergelijking van de meting $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^m$ mogelijk niet-lineaire functies zijn.

Een typisch data-assimilatie experiment met de toolbox bestaat slechts uit vier stappen. De eerste stap is het definiëren van de verschillende ruismodellen voor zowel de procesruis, de metingsruis als voor de initiële toestand. De toolbox beschikt over drie verschillende klassen voor het modelleren van Gaussiaanse ruis. Het voornaamste verschil tussen deze modellen is de manier waarop het gemiddelde en de covariantie worden gedefinieerd. Een eerste mogelijkheid is met een vector en een matrix wat aanleiding geeft tot een tijdsinvariant ruismodel. Wanneer het gemiddelde en de covariantie gedefinieerd worden door een rij van vectoren en matrices wordt een tijdsvariant ruismodel bekomen. Een laatste mogelijkheid is dat zowel het gemiddelde als de covariantie bepaald worden door functies zoals door de gebruiker opgesteld.

De tweede stap bestaat uit het opstellen van een toestandsmodel. De toolbox beschikt hiervoor over drie klassen die overeenkomen met de reeds eerder geformuleerde toestandsmodellen. De belangrijkste parameters van een toestandsmodel zijn de reeds opgestelde ruismodellen en de systeemvergelijkingen. De systeemvergelijkingen worden opgesteld op twee mogelijke manieren, afhankelijk van het type model. Voor het lineaire model moeten de systeemmatrices vastgelegd worden terwijl voor de

niet-lineaire modellen de functies f en h gedefinieerd moeten worden. Dit gebeurt met door de gebruiker gedefinieerde functies.

De derde stap is afhankelijk van het type experiment. Ofwel is het een experiment op een reëel systeem met echte metingen, ofwel is het een experiment van zuiver academische aard waarbij geen metingen beschikbaar zijn. In het laatste geval beschikt de toolbox over een simulatie-procedure die aan de hand van het toestandsmodel virtuele metingen genereert.

De laatste stap is het toepassen van een data-assimilatie techniek. Eender welke techniek heeft als doel een betere schatting te geven van de toestanden aan de hand van de metingen en het toestandsmodel zoals bekomen in stap twee en drie. Buiten de toestanden kunnen ook de varianties bekomen worden teneinde een indicatie te hebben hoe waarschijnlijk een geschatte toestand is.

De toolbox beschikt ook over verschillende procedures om op een snelle manier de bekomen data te analyseren. Zo is het bijvoorbeeld heel eenvoudig om de verschillende toestanden en hun betrouwbaarheidsintervallen weer te geven in figuren of om hun RMSE waarden te berekenen.

Besluit

Het doel van deze masterproef was initieel een generieke data-assimilatie toolbox in Matlab te ontwikkelen met minstens vijf data-assimilatie technieken voor het verbeteren van schattingen van een door de gebruiker gedefinieerd systeem. Dit initiële doel is volledig bereikt en zelfs uitgebreid naar elf data-assimilatie technieken. Door de object-georiënteerde programmatie van de toolbox is het eveneens zeer eenvoudig om nieuwe technieken, toestandsmodellen en ruismodellen toe te voegen. Ook de reeds aanwezige middelen voor het analyseren van bekomen data zijn eenvoudig uitbreidbaar. De belangrijkste mogelijke verbeteringen zijn het toevoegen van grafische hulpmiddelen. Zo kunnen bijvoorbeeld de bestaande hulpfuncties en demonstraties in de toolbox voorzien worden in html formaat zoals ook door de officiële toolboxen wordt voorzien. Een grafische gebruikersinterface zou zeker ook een mooie toegevoegde waarde betekenen.

List of Figures and Tables

List of Figures

1.1 Using global meteorological observations to obtain an initial state estimation. . . . 2

2.1 Illustration of the basic particle resample algorithm. . . . 22

3.1 Illustration of the workflow of a typical data assimilation experiment. . . . 33
3.2 Illustration of the different classes and their inheritance. . . . 35

4.1 Illustration of a pre-configured simulation plot: true states of the Van der Pol equations. . . . 63
4.2 Illustration of a pre-configured data assimilation plot: true states and analysis states of the Van der Pol equations. . . . 64
4.3 Illustration of a pre-configured data assimilation plot: difference between true states and analysis states of the Lorenz equations. . . . 66
4.4 Three-dimensional illustration of true states and analysis states of the Lorenz equations. . . . 66
4.5 Illustration of the tracking problem: applying radar and triangulation measurements. . . . 68
4.6 Illustration of a pre-configured data assimilation plot: true states, analysis states and the 95% confidence interval of the scalar problem. . . . 70

List of Tables

4.1 The tracking problem. RMSE values of estimated states and true states as obtained from the UKF, ETKF and PF_ASIR for radar and triangulation measurements. . . . 69

Glossary

Variables

a, b, c	Vector variables or scalars if explicitly stated or when clear from context.
A, B, C	Matrix variables or scalars if explicitly stated or when clear from context.
I_a	Identity matrix of size a

Sets

$\mathbb{R}$	The set of real numbers
$\mathbb{R}^{n \times n}$	The set of n-dimensional real vectors
$\mathbb{R}^{n \times m}$	The set of $n \times m$ real matrices
$\{a_1, \ldots, a_n\}$	The set of vectors $a_1, \ldots, a_n$
$\{a_i\}_{i=1}^{n}$	Shorthand notation for the set $\{a_1, \ldots, a_n\}$
$\{a_i\}$	Shorthand notation for a set of well known size

Matrix Operations

A^T	Transpose of matrix A
$A^{(-1)}$	Inverse of matrix A
$A^{(1/2)}$	Square-root of matrix A
$A_{(i,j)}$	Selection of the ith row and jth column of the matrix A
$A_{(i,:)}$	Selection of the ith row and all columns of the matrix A
$A_{(:,j)}$	Selection of all rows and jth column of the matrix A

Symbols

$p(a\|b)$	Probability of a, given b.
$p(a\|b,c)$	Probability of a, given b and c.
$[a_1,\ldots,a_n]$	Matrix containing vectors $a_1,\ldots,a_n$
$\lfloor a \rfloor$	Floor rounding of real number a
$\mathbb{E}[a]$	Expected value of the random vector a
$\hat{a}$	Estimate of the (random) vector a
$\sim (a, A)$	Distributed with mean vector a and covariance matrix A (any distribution)
$\sim \mathcal{N}(a, A)$	Normally distributed with mean vector a and covariance matrix A
$\sim \mathcal{U}(a, b)$	Uniformly distributed in interval $[a, b]$

Fixed symbols

$x \in \mathbb{R}^n$	State vector
$y \in \mathbb{R}^m$	Measurement vector
$u \in \mathbb{R}^p$	Input vector
k	Step number
A, B, C, D	System matrices
f, h	System equations
w, v	Noise vectors
P, Q, R	Covariance matrices

Algorithm specific symbols

$\hat{x}_{k\|k-1}$	Estimation of the vector x at step k, given the measurement from step $k-1$. (Forecast state vector)
$\hat{x}_{k\|k}$	Estimation of the vector x at step k, given the measurement from step k. (Analysis state vector)
$\hat{x}^{(i)}_{k\|k}$	Analysis state vector x that is the ith member of a set of vectors.
$\left[\hat{x}^{(1)}_{k\|k},\ldots,\hat{x}^{(N)}_{k\|k}\right]$	Matrix that contains N analysis state vectors.

List of Abbreviations

CDF	Cumulative Distribution Function
DEnKF	Deterministic Ensemble Kalman Filter
EKF	Extended Kalman Filter
EnKF	Ensemble Kalman Filter
EnSRF	Ensemble square Root Filter
ETKF	Ensemble Transform Kalman Filter
KF	Kalman Filter
LTI	Linear Time Invariant
LTV	Linear Time Variant
MSE	Mean Square error
pdf	Probability Density Function
OI	Optimal Interpolation
PF-ASIR	Auxiliary Sampling Importance Resampling Particle Filter
PF-GEN	Generic Particle Particle Filter
PF-SIR	Sampling Importance Resampling Particle Filter
UKF	Unscented Kalman Filter
RMSE	Root Mean Square error

Chapter 1

Introduction

Data assimilation is the common name given to the techniques that combine numerical models and measurements in order to obtain an improved estimation of the state of a system. In data assimilation it is assumed that both models and measurements are subject to errors or uncertainties that can be defined as a statistical distribution. The goal of data assimilation is to combine the knowledge of models, measurements and uncertainties to obtain a better estimation than either the measurements or the models alone can provide. The application of this technique arises in many fields such as weather forecasting, oceanography, space weather forecasting and air quality.

Consider for example the problem of weather forecasting. To be able to forecast the weather it is imperative to know what the exact state of the current weather is. Furthermore, this exact state is not only required at the location of interest, but over a wide area around this location. The longer the required forecast, the more globally these states must be known. This is why a wide range of measurement instruments are maintained which is known as the world-wide meteorological observing network. This network consists of a variety of measurement sources ranging from surface stations, buoys, radars, airplanes and weather ships to satellites as illustrated in Figure 1.1. Still, the acquired data is insufficient to obtain an accurate initial state. In addition, the available data is always subject to measurement errors. This is why data assimilation techniques are applied on this defective data and several weather models to estimate the most likely initial state and its potential error based. The great improvements in weather forecasts during the last decade are mainly due to improved data assimilation techniques and additional observations [18].

Although data assimilation is a highly active research domain, MATLAB does not have an official data assimilation toolbox. This is why the goal of this thesis has been to develop a generic data assimilation toolbox for MATLAB with at least five data assimilation schemes to improve the estimations of a given model as defined by the user. As will become clear throughout this manuscript, this goal is more than achieved. By applying the MATLAB object-oriented programming approach, the toolbox not only provides the required generic behaviour but also maintains a

Figure 1.1: Using global meteorological observations to obtain an initial state estimation.

structured framework that can be easily extended with new algorithms and classes. The interface of the toolbox is intuitively straightforward since it resembles the ones of official Matlab toolboxes. Furthermore, a total of eleven data assimilation techniques have been implemented. Finally, it is important to notice that the toolbox does not make use of any other official toolboxes to avoid licensing problems.

Chapter 2 presents all the data-assimilation techniques that are implemented in the toolbox. The main advantages and disadvantages of the different algorithms are indicated and a summary of their algorithms is provided. Before describing the algorithms, the reader is familiarized with the different state space representations that are used throughout the different chapters and with the concept of the Bayesian recursive state estimation. Although the Bayesian recursive filter is mainly a conceptual filter, it is still important as it provides an excellent framework to understand other filters. The presentation of the data assimilation algorithms starts with the Kalman Filter (KF). Although this filter dates from 1960 and is only suited for linear systems with a limited amount of states, it is still an often applied technique that deserves its place in the toolbox. Moreover, its simplicity provides a clear picture of the concept behind all Kalman based filters. Next, the Extended Kalman Filter (EKF) is presented which is a nonlinear variant of the Kalman filter. After the EKF, the Optimal Interpolation (OI) technique is summarized which is the first technique that is suited for large-scale systems. The Unscented Kalman Filter (UKF) propagates state points through the nonlinear equations to estimate the probability density function of the state. This filter was inspired by the Ensemble Kalman Filter (EnKF) which is the next illustrated algorithm. A first main difference between the EnKF and the UKF is that the EnKF chooses his state points (ensemble members) at random, while the UKF selects its state points (sigma points) deterministically. The second main difference is that, while the amount of required sigma points is equal to the order of the amount of states, the ensemble size can be selected by the

user. This explains the large amount of applications of the EnKF on large-scale systems. The following sections present three variants of the EnKF: the Ensemble Square Root Filter (EnSRF), the Ensemble Transform Kalman Filter (ETKF) and the Deterministic Ensemble Kalman Filter (DEnKF). The final three sections present the particle filters which also use state points (particles), but without using the Kalman based framework. Instead, they can be considered as brute force implementations of the Bayesian recursive estimator. Although often successful when applied to highly nonlinear systems, they are not suited for large-scale systems since they typically require a large amount of particles. The presented particle filters are: the Generic particle filter (GEN), the Sampling Importance Resampling particle filter (SIR) and the Auxiliary Sampling Importance Resampling particle filter (ASIR).

Chapter 3 can be considered as a user guide to the toolbox. The first sections explain the general architecture of the toolbox and are probably the most important sections to understand how the toolbox is set up. Since the object-oriented approach is used, the toolbox is divided in classes. These classes are selected to have a structure that provides an intuitively straightforward sequence of steps to create a data assimilation experiment. The first group of classes represent the noise models. There are currently three noise model classes for Gaussian noise. The first type of class is used for LTI noise, the second one for LTV noise and the last class can be any type of Gaussian noise since it is configured by means of user defined functions. The second group of classes are the state space models. There are three state space models ranging from linear to fully nonlinear. The parametrization of the state equations consists of matrices or three-dimensional arrays for the linear model and user defined functions for the nonlinear models; the required noise sources are configured using any of the noise classes. Once a state space model is fully configured, any data assimilation technique can be applied. When no measurements are available, a simulation can be used to generate virtual measurements. The data, as obtained from simulation and data assimilation, is automatically stored in advanced data structure classes. These classes allow fast manipulation of the data by means of plots and some basic calculations such as the RMSE.

The last sections of this chapter are dedicated to summarize and explain the main possibilities of all classes and methods. Throughout these sections small examples are provided to aid the user in understanding the different features.

Chapter 4 provides fully illustrated examples on how to apply the data assimilation toolbox. There are four examples included that are often encountered in the literature: the Van der Pol oscillator, the Lorenz equations, a tracking problem and a highly nonlinear scalar system. The first two examples are well known nonlinear systems. The tracking problem consists of determining the location of a linearly modeled airplane with highly nonlinear measurement equations. The scalar system has become, despite it only has one state, a benchmark for nonlinear estimators because of its highly nonlinear state equation in combination with a nonlinear measurement equation.

Chapter 2

Data assimilation techniques

This chapter presents all data assimilation techniques that are implemented in the toolbox. It is not the goal of this chapter to explain every detail of the methods. Instead, only the most important aspects are summarized and the algorithms are briefly explained. As such, it is assumed that the reader already has a good knowledge of state space models, Bayesian statistics and filtering techniques. If not, a good introduction on these topics can be found in [15]. Another good reference for most of the presented Kalman filters is [1].

Before introducing the filters, the first section presents the different types of state space models used throughout the chapter. Next the principle of recursive Bayesian state estimation is summarized. This conceptual filter is important to keep in mind since all filters can be understood in light of this estimator. From then on the Kalman Filter (KF), the Extended Kalman Filter (EKF), the Optimal Interpolation technique (OI), the Unscented Kalman Filter (UKF) and four variants of the ensemble Kalman filter are introduced. The different variants are the traditional Ensemble Kalman Filter (EnKF), the Ensemble Square Root Filter (EnSRF), the Ensemble Transform Kalman Filter (ETKF) and the Deterministic Ensemble Kalman Filter (DEnKF). The last three sections contain particle filters, more specifically the Generic particle filter (GEN), the Sampling Importance Resampling (SIR) particle filter and the Auxiliary Sampling Importance Resampling (ASIR) particle filter.

2.1 State Space models

In this section the discrete-time state space models that are used to illustrate the filter algorithms are summarized. There are three different models and they can be ranked hierarchically from linear at the bottom to completely nonlinear at the top. When ascending in this hierarchy each model is a more general case of the previous one. Hence, if an algorithm is presented for a certain state space model than the algorithm is also suited for the lower ranked models. The higher ranked models are either impossible or suboptimal without proper adjustment of the algorithm. All models contain the vectors $x_k \in \mathbb{R}^n$, $y_k \in \mathbb{R}^m$ and $u_k \in \mathbb{R}^p$ which denote the state vector, the measurement vector and the input vector at time k. The vectors

$w_k \in \mathbb{R}^n$ and $v_k \in \mathbb{R}^m$ represent the process and measurement noise for which the stochastic processes $\{w_k\}_{k=0}^{\infty}$ and $\{v_k\}_{k=0}^{\infty}$ originate from a known type of distribution.

The different state space models are:

- The Discrete Linear model
  The discrete linear model is the least general state space model. Its equations are

$$x_{k+1} = A_k x_k + B_k u_k + w_k, \tag{2.1}$$
$$y_k = C_k x_k + D_k u_k + v_k, \tag{2.2}$$

  where $A_k \in \mathbb{R}^{n\times n}$, $B_k \in \mathbb{R}^{n\times p}$, $C_k \in \mathbb{R}^{m\times n}$ and $D_k \in \mathbb{R}^{m\times p}$ are respectively the state, input, output and feedthrough matrices at time k.

- The Discrete Nonlinear model with Additive Noise
  The discrete nonlinear model with additive noise is the least general nonlinear state space model in the toolbox. Its equations are

$$x_{k+1} = f(x_k, u_k, k) + w_k, \tag{2.3}$$
$$y_k = h(x_k, u_k, k) + v_k, \tag{2.4}$$

  where the system equation $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^m$ are possibly nonlinear functions.

- The Discrete Nonlinear model
  This is the most general model where both measurement and process noise are allowed to be nonlinear. Its equations are

$$x_{k+1} = f(x_k, u_k, w_k, k), \tag{2.5}$$
$$y_k = h(x_k, u_k, v_k, k), \tag{2.6}$$

  where the system equation $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^m$ are possibly nonlinear functions. It must be noted that this model is not often encountered in practice.

2.2 Recursive Bayesian State Estimation

Before presenting the implemented filter algorithms, the Bayesian approach to state estimation is introduced. It is the most general and optimal form of state estimation and as such can be considered as the framework for all subsequent recursive filter algorithms. For a more rigorous description see [15]. The goal of Bayesian state estimation is to recursively construct the probability density function (pdf) of the state given the previous state and a new set of measurements. Since the pdf contains all statistical information it can be considered the optimal solution to the estimation problem. The filter uses two steps which apply the Bayesian theorem: the forecast step and the analysis step.[1] In the forecast step the system model, with known

[1] Other notations can be found in the literature for these two steps such as prediction and update steps or time update and measurement update steps.

process noise, is used to predict the new state pdf from the pdf of the previous state. The analysis step incorporates the current measurement, with known measurement noise, to improve the estimation of the state pdf.

To illustrate the filter more formally, consider the model

$$x_{k+1} = f(x_k, w_k, k), \tag{2.7}$$
$$y_k = h(x_k, v_k, k), \tag{2.8}$$

where the system equation $f : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^m$ are possibly nonlinear functions and $\{w_k\}$ and $\{v_k\}$ are assumed i.i.d. white noise sequences with known pdf's. For ease of notation no inputs are considered in the model. Since (2.7) is a Markov process its equivalent probabilistic description $p(x_k|x_{k-1}, y_{1:k-1})$ is equal to $p(x_k|x_{k-1})$. Similarly, the probabilistic description of (2.8) is the likelihood $p(y_k|x_k)$. Finally, it is assumed that initial condition, known as the prior pdf of the state $p(x_0) \equiv p(x_0|y_0)$, is known. The goal of the Bayesian estimator is to create the posterior pdf $p(x_k|y_{1:k})$ of the state vector. As indicated before this is achieved recursively in two steps:

1. Forecast step
   The posterior pdf, $p(x_{k-1}|y_{1:k-1})$, is known from time $k-1$ and $p(x_k|x_{k-1})$ can be determined from the system equation. As such the prior pdf of the state at time k, $p(x_k|y_{1:k-1})$, can be obtained through the Chapman-Kolmogorov equation
   $$p(x_k|y_{1:k-1}) = \int p(x_k|x_{k-1})p(x_{k-1}|y_{1:k-1})dx_{k-1}. \tag{2.9}$$

2. Analysis step
   The prior pdf is updated using the likelihood $p(y_k|x_k)$ which can be determined from the measurement equation using the Bayes rule
   $$p(x_k|y_{1:k}) = \frac{p(y_k|x_k)p(x_k|y_{1:k-1})}{\int p(y_k|x_k)p(x_k|y_{1:k-1})dx_{k-1}}, \tag{2.10}$$
   where the denominator is a normalizing constant.

The Bayesian estimator should be considered more as a conceptual solution since in most cases an analytical solution is not available. Only when f and h are linear and x_0, $\{w_k\}$ and $\{v_k\}$ are additive, independent and Gaussian a solution exists. This solution is known as the Kalman filter which is the topic of the next section.

2.3 The Kalman Filter - KF

Consider the discrete linear model with system and measurement equations

$$x_{k+1} = A_k x_k + B_k u_k + w_k, \tag{2.11}$$
$$y_k = C_k x_k + D_k u_k + v_k,$$

where $A_k \in \mathbb{R}^{n\times n}$, $B_k \in \mathbb{R}^{n\times p}$, $C_k \in \mathbb{R}^{m\times n}$ and $D_k \in \mathbb{R}^{m\times p}$ are respectively the state, input, output and feedthrough matrices at time k.

Additionally the noise processes $\{w_k\}$ and $\{v_k\}$ satisfy

$$\begin{aligned} w_k &\sim (0, Q_k), \\ v_k &\sim (0, R_k), \\ \mathbb{E}[w_k w_j^T] &= Q_k \delta_{k-j}, \\ \mathbb{E}[v_k v_j^T] &= R_k \delta_{k-j}, \\ \mathbb{E}[v_k w_j] &= 0, \end{aligned} \tag{2.12}$$

where δ_k is the Kronecker delta function for which $\delta_k = 1$ if $k = 0$ and $\delta_k = 0$ otherwise. Hence, the noise processes are white, zero-mean, uncorrelated and have known covariance matrices Q_k and R_k.

Applying the Kalman filter on this model provides the analytical solution to the recursive Bayesian state estimator (see Section 2.2). Therefore, for this model and under the extra assumption of Gaussian noise, the Kalman filter is the optimal filter. Furthermore, it can be shown in a least squares sense that the Kalman filter is the optimal *linear* filter regardless of the statistical nature of the noise, as long as it is zero-mean [3]. That is why, from now on, no assumptions are made on the whiteness or correlations of the noise sources, i.e. the last three equations of (2.12) are omitted for this filter and all subsequent Kalman filters. Still, it is important to keep in mind that when the noise sources are Gaussian, white and uncorrelated the Kalman based filters provide a better estimation. The Kalman filter algorithm is summarized below. For a more detailed introduction and derivation of the algorithm see [15] or [1]. Furthermore, an excellent intuitive example on the basic principle of the Kalman filter can be found in [9].

The Kalman filter algorithm is as follows:

1. Initialization step

$$\begin{aligned} \hat{x}_{0|0} &= \mathbb{E}[x_0] \\ P_{0|0} &= \mathbb{E}[(x_0 - \hat{x}_{0|0})(x_0 - \hat{x}_{0|0})^T] \end{aligned} \tag{2.13}$$

2. For $k = 1, 2, \ldots$

   a) Forecast step

$$\begin{aligned} \hat{x}_{k|k-1} &= A_{k-1}\hat{x}_{k-1|k-1} + B_{k-1}u_{k-1} \\ P_{k|k-1} &= A_{k-1}P_{k-1|k-1}A_{k-1}^T + Q_{k-1} \end{aligned} \tag{2.14}$$

   b) Kalman Gain

$$K_k = P_{k|k-1}C_k^T(C_k P_{k|k-1} C_k^T + R_k)^{-1} \tag{2.15}$$

c) Analysis step

$$\begin{aligned} \hat{x}_{k|k} &= \hat{x}_{k|k-1} + K_k \left(y_k - C_k \hat{x}_{k|k-1} - D_k u_k \right) \\ P_{k|k} &= (I_n - K_k C_k) P_{k|k-1} \end{aligned} \tag{2.16}$$

3. End

2.4 The Extended Kalman Filter - EKF

The Kalman filter offers the optimal solution for linear models, but it has some practical limitations. The real world is nonlinear and many times an approximated linear model will not suffice. Also, for large-scale problems the memory requirements and the calculation times soon become unfeasible. The extended Kalman filter addresses the problem of nonlinearity by linearizing the nonlinear system around the Kalman filter state estimate and subsequently retrieve the Kalman filter state and error covariance estimates from the linearized system. (For more details see [15].

Consider the discrete nonlinear model with

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, w_k, k), \\ y_k &= h(x_k, u_k, v_k, k), \\ w_k &\sim (0, Q_k), \\ v_k &\sim (0, R_k), \end{aligned} \tag{2.17}$$

where the system equation $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^m$ are possibly nonlinear functions.

The extended Kalman filter algorithm can be summarized in the following way:

1. Initialization step

$$\begin{aligned} \hat{x}_{0|0} &= \mathbb{E}[x_0] \\ P_{0|0} &= \mathbb{E}[(x_0 - \hat{x}_{0|0})(x_0 - \hat{x}_{0|0})^T] \end{aligned} \tag{2.18}$$

2. For $k = 1, 2, \ldots$

a) Calculate the following Jacobian matrices:

$$\begin{aligned} A_{k-1} &= \left. \frac{\partial f}{\partial x} \right|_{\hat{x}_{k-1|k-1}, u_{k-1}, 0, k-1} \\ B_{k-1} &= \left. \frac{\partial f}{\partial w} \right|_{\hat{x}_{k-1|k-1}, u_{k-1}, 0, k-1} \end{aligned} \tag{2.19}$$

b) Forecast step

$$\begin{aligned} \hat{x}_{k|k-1} &= f(\hat{x}_{k-1|k-1}, u_{k-1}, 0, k-1) \\ P_{k|k-1} &= A_{k-1} P_{k-1|k-1} A_{k-1}^T + B_{k-1} Q_{k-1} B_{k-1}^T \end{aligned} \tag{2.20}$$

c) Calculate the following Jacobian matrices:

$$\begin{aligned} C_k &= \left.\frac{\partial h}{\partial x}\right|_{\hat{x}_{k|k-1}, u_k, 0, k} \\ D_k &= \left.\frac{\partial h}{\partial v}\right|_{\hat{x}_{k|k-1}, u_k, 0, k} \end{aligned} \tag{2.21}$$

d) Kalman Gain

$$K_k = P_{k|k-1} C_k^T (C_k P_{k|k-1} C_k^T + D_k R_k D_k^T)^{-1} \tag{2.22}$$

e) Analysis step

$$\begin{aligned} \hat{x}_{k|k} &= \hat{x}_{k|k-1} + K_k \left(y_k - h(\hat{x}_{k|k-1}, u_k, 0, k) \right) \\ P_{k|k} &= (I_n - K_k C_k) P_{k|k-1} \end{aligned} \tag{2.23}$$

3. End

2.5 Optimal Interpolation - OI

The optimal interpolation algorithm assumes that the forecast error covariance matrix is constant. Instead of calculating the error covariance by using the dynamical equations, the user determines its value in advance [17]. There are several techniques to determine this error covariance matrix but this is not considered to be the topic of this manuscript. The advantage of this filter is clear: the amount of required calculations is drastically reduced. The disadvantage is of this technique is a reduced accuracy. The OI algorithm here is shown in its most generic form. Of course, in case of a linear time-invariant measurement equation, the calculations reduce a lot since besides the Kalman gain matrix K_k becomes fixed and can be pre-calculated which reveals the true power of the OI filter.

Consider the discrete nonlinear model with

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, w_k, k), \\ y_k &= h(x_k, u_k, v_k, k), \\ v_k &\sim (0, R_k), \end{aligned} \tag{2.24}$$

where the system equation $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}^m$ are possibly nonlinear functions. The measurement noise w_k originates from a zero-mean distribution.

The optimal interpolation algorithm is as follows:

1. Initialization step
$$\hat{x}_{0|0} = \mathbb{E}[x_0] \tag{2.25}$$
and define the error covariance matrix P.

2. For $k = 1, 2, \ldots$

   a) Forecast step
   $$\hat{x}_{k|k-1} = f(\hat{x}_{k-1|k-1}, u_{k-1}, 0, k-1) \tag{2.26}$$

   b) Calculate the following Jacobian matrices:
   $$C_k = \left.\frac{\partial h}{\partial x}\right|_{\hat{x}_{k|k-1}, u_k, 0, k} \tag{2.27}$$
   $$D_k = \left.\frac{\partial h}{\partial v}\right|_{\hat{x}_{k|k-1}, u_k, 0, k}$$

   c) Kalman Gain
   $$K_k = PC_k^T(C_k P C_k^T + D_k R_k D_k^T)^{-1} \tag{2.28}$$

   d) Analysis step
   $$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k\left(y_k - h(\hat{x}_{k|k-1}, u_k, 0, k)\right) \tag{2.29}$$

3. End

2.6 The Unscented Kalman Filter - UKF

The extended Kalman filter is often used in practice due to its simple approach of propagating the mean and error covariance matrix of the states through a linear approximation of the system. Still, it has some clear disadvantages. It requires the Jacobian matrices which can be computationally expensive to calculate or error-prone if provided manually. Furthermore, the linearization is only of order one and is therefore inaccurate when applying to very nonlinear systems [16]. Higher order or iterated extended Kalman filters increase the linearization accuracy but are not commonly used due to their complexity and high computational effort. The unscented Kalman filter addresses the problem of transforming a pdf through a nonlinear function in a completely different way. It applies an unscented transformation which performs the nonlinear transformation on a set of deterministic sigma point vectors whose sample pdf is used to approximate the true pdf. The extra parameter κ is used to reduce higher order errors of the mean and covariance approximation. For a more detailed description of the algorithm the reader is referred to [15].

Consider the discrete nonlinear model with

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, k) + w_k, \\ y_k &= h(x_k, u_k, k) + v_k, \\ w_k &\sim (0, Q_k), \\ v_k &\sim (0, R_k), \end{aligned} \tag{2.30}$$

where the system equation $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^m$ are possibly nonlinear functions.

The unscented Kalman filter algorithm is presented in the following lines:

1. Initialization step

$$\begin{aligned} \hat{x}_{0|0} &= \mathbb{E}[x_0] \\ P_{0|0} &= \mathbb{E}[(x_0 - \hat{x}_{0|0})(x_0 - \hat{x}_{0|0})^T] \end{aligned} \tag{2.31}$$

2. For $k = 1, 2, \ldots$

   a) Forecast step

      i. Calculate $2n + 1$ sigma points

$$\begin{aligned} \hat{x}^{(0)}_{k-1|k-1} &= \hat{x}_{k-1|k-1} && \\ \hat{x}^{(i)}_{k-1|k-1} &= \hat{x}_{k-1|k-1} + \tilde{x}^{(i)} && i = 1, \ldots, 2n \\ \tilde{x}^{(i)} &= \left(\sqrt{(n+\kappa)P_{k-1|k-1}}\right)^T_i && i = 1, \ldots, n \\ \tilde{x}^{(n+i)} &= -\left(\sqrt{(n+\kappa)P_{k-1|k-1}}\right)^T_i && i = 1, \ldots, n \end{aligned} \tag{2.32}$$

      where $\sqrt{P}^T\sqrt{P} = P$, $(P)_i$ denotes the ith row of P and $\hat{x}^{(i)}_{k-1|k-1}$ denotes the ith sigma point.

      ii. Transform the sigma points

$$\hat{x}^{(i)}_{k|k-1} = f(\hat{x}^{(i)}_{k-1|k-1}, u_{k-1}, k-1) \quad i = 0, \ldots, 2n \tag{2.33}$$

      iii. Estimate the forecast state and error covariance

$$\begin{aligned} W^{(0)} &= \frac{\kappa}{n+\kappa} \\ W^{(i)} &= \frac{1}{2(n+\kappa)} \quad i = 1, \ldots, 2n \\ \hat{x}_{k|k-1} &= \sum_{i=0}^{2n} W^{(i)} \hat{x}^{(i)}_{k|k-1} \\ P_{k|k-1} &= \sum_{i=0}^{2n} W^{(i)} \left(\hat{x}^{(i)}_{k|k-1} - \hat{x}_{k|k-1}\right)\left(\hat{x}^{(i)}_{k|k-1} - \hat{x}_{k|k-1}\right)^T + Q_{k-1} \end{aligned} \tag{2.34}$$

b) Analysis step

i. Calculate $2n+1$ sigma points (If desired can be omitted by reusing the previous sigma points.)

$$\begin{aligned}
\hat{x}^{(0)}_{k|k-1} &= \hat{x}_{k|k-1} && (2.35)\\
\hat{x}^{(i)}_{k|k-1} &= \hat{x}_{k|k-1} + \tilde{x}^{(i)} && i = 1, \dots, 2n\\
\tilde{x}^{(i)} &= \left(\sqrt{(n+\kappa)P_{k|k-1}}\right)^T_i && i = 1, \dots, n\\
\tilde{x}^{(n+i)} &= -\left(\sqrt{(n+\kappa)P_{k|k-1}}\right)^T_i && i = 1, \dots, n
\end{aligned}$$

ii. Calculate the predicted measurements

$$\hat{y}^{(i)}_k = h(\hat{x}^{(i)}_{k|k-1}, u_k, k) \quad i = 0, \dots, 2n \tag{2.36}$$

iii. Calculate the mean and error covariance of the predicted measurements and the cross-covariance

$$\begin{aligned}
\hat{y}_k &= \sum_{i=0}^{2n} W^{(i)} \hat{y}^{(i)}_k && (2.37)\\
P_{y_k} &= \sum_{i=0}^{2n} W^{(i)} \left(\hat{y}^{(i)}_k - \hat{y}_k\right)\left(\hat{y}^{(i)}_k - \hat{y}_k\right)^T + R_k\\
P_{xy_k} &= \sum_{i=0}^{2n} W^{(i)} \left(\hat{x}^{(i)}_{k|k-1} - \hat{x}_{k|k-1}\right)\left(\hat{y}^{(i)}_k - \hat{y}_k\right)^T
\end{aligned}$$

iv. Kalman Gain

$$K_k = P_{xy_k} P^{-1}_{y_k} \tag{2.38}$$

v. Estimate the analysis state and error covariance

$$\begin{aligned}
\hat{x}_{k|k} &= \hat{x}_{k|k-1} + K_k\left(y_k - \hat{y}_k\right) && (2.39)\\
P_{k|k} &= P_{k|k-1} - K_k P_y K_k^T
\end{aligned}$$

3. End

2.7 The Ensemble Kalman Filter - EnKF

The unscented Kalman filter provides, in general, much better results than the extended Kalman filter for nonlinear systems at a lower computational effort. However it still has the disadvantage is that it requires $2n+1$ sigma points which causes it to be too expensive for large-scale systems. The ensemble Kalman filter solves this by allowing an heuristic amount of samples, the ensemble members. Typically the size of an ensemble is chosen much smaller than the amount of states for large-scale systems. Furthermore, the ensemble is not selected deterministically but randomly

(Monte Carlo method). More precisely, it is assumed that the ensemble members are samples from the pdf of the true state [10]. It is important to notice that the algorithm does not explicitly calculate the error covariance matrix $P_{k|k-1} \in \mathbb{R}^{n\times n}$. Instead, the matrices $P_y \in \mathbb{R}^{m\times m}$ and $P_{xy} \in \mathbb{R}^{n\times m}$ are calculated. These represent respectively $P_{k|k-1}C_k^T$ and $C_k P_{k|k-1} C_k^T + R_k$ from the regular Kalman filter (Section 2.3). For a more detailed discussion see [1] or [14].

Consider the discrete nonlinear model with

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, k) + w_k, \\ y_k &= h(x_k, u_k, k) + v_k, \\ w_k &\sim (0, Q_k), \\ v_k &\sim (0, R_k), \end{aligned} \tag{2.40}$$

where the system equation $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^m$ are possibly nonlinear functions.

The ensemble Kalman filter algorithm is as follows:

1. Initialization step.
   Generate a matrix of N initial ensemble members $\left[\hat{x}_{0|0}^{(1)}, \hat{x}_{0|0}^{(2)}, \ldots, \hat{x}_{0|0}^{(N)}\right]$ that properly represent the error statistics of the initial guess $x_{0|0}$ of the state.

2. For $k = 1, 2, \ldots$

   a) Forecast step

      i. Transform the ensemble members

$$\hat{x}_{k|k-1}^{(i)} = f(\hat{x}_{k-1|k-1}^{(i)}, u_{k-1}, k-1) + w_{k-1}^{(i)} \quad i = 1, \ldots, N \tag{2.41}$$

      ii. Estimate the forecast state (ensemble mean)

$$\hat{x}_{k|k-1} = \frac{1}{N} \sum_{i=1}^{N} \hat{x}_{k|k-1}^{(i)}$$

   b) Analysis step

      i. Calculate the predicted measurements

$$\hat{y}_k^{(i)} = h(\hat{x}_{k|k-1}^{(i)}, u_k, k) \quad i = 1, \ldots, N \tag{2.42}$$

ii. Calculate the mean and error covariance of the predicted measurements and the cross-covariance

$$\hat{y}_k = \frac{1}{N}\sum_{i=1}^{N}\hat{y}_k^{(i)} \tag{2.43}$$

$$P_{y_k} = \frac{1}{N-1}\sum_{i=1}^{N}\left(\hat{y}_k^{(i)} - \hat{y}_k\right)\left(\hat{y}_k^{(i)} - \hat{y}_k\right)^T + R_{e_k}$$

$$P_{xy_k} = \frac{1}{N-1}\sum_{i=1}^{N}\left(\hat{x}_{k|k-1}^{(i)} - \hat{x}_{k|k-1}\right)\left(\hat{y}_k^{(i)} - \hat{y}_k\right)^T$$

with $R_{e_k} = \frac{1}{N-1}\sum_{i=1}^{N}\left(v_k^{(i)}\right)\left(v_k^{(i)}\right)^T$ or with $R_{e_k} = R_k$, which is specified by the user.

iii. Kalman Gain

$$K_k = P_{xy_k}P_{y_k}^{-1} \tag{2.44}$$

iv. Estimate the analysis state (ensemble mean)

$$\hat{x}_{k|k}^{(i)} = \hat{x}_{k|k-1}^{(i)} + K_k\left(y_k + v_k^{(i)} - \hat{y}_k^{(i)}\right) \quad i = 1,\ldots,N \tag{2.45}$$

$$\hat{x}_{k|k} = \frac{1}{N}\sum_{i=1}^{N}\hat{x}_{k|k}^{(i)}$$

3. End

2.8 The Ensemble Square Root Filter - EnSRF

As can be seen in equation (2.45), the ensemble Kalman filter adds synthetic perturbations $v_k^{(i)}$ to the measurement y_k. Without these perturbations the ensemble Kalman filter would systematically underestimate the analysis error covariance matrix $P_{k|k}$. However, as shown in many papers, there are two downsides of this solution. First, the perturbations only solve the underestimation problem in a statistical sense for an infinite amount of ensembles. Second, the perturbations introduce additional sampling errors. This makes the ensemble Kalman filter suboptimal, especially when using few ensemble members. The ensemble square root filter uses a different approach which avoids the underestimation of the analysis error covariance matrix by using a different Kalman gain. The traditional Kalman gain K_k is used to calculate the analysis ensemble mean, while a reduced Kalman gain $\tilde{K}_k = \beta_k K_k$ is used to update the deviations from the analysis ensemble mean [5]. The factor β_k originates from the Potter square root measurement-update equation [15]. It has a value between 0 and 1 and is selected to correctly estimate the theoretical analysis error covariance matrix $P_{k|k}$. Please note that the sequential processing algorithm as shown below is only valid when the measurement noise v_k is uncorrelated. This implementation has the large advantage that it avoids computing matrix square roots and as such requires similar computation time than the EnKF while offering

increased accuracy. For more details about the algorithm see [1] or [5].

Consider the discrete nonlinear model with

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, k) + w_k, \\ y_k &= h(x_k, u_k, k) + v_k, \\ w_k &\sim (0, Q_k), \\ v_k &\sim (0, R_k), \\ \mathbb{E}[v_k v_j^T] &= R_k \delta_{k-j}, \end{aligned} \tag{2.46}$$

where the system equation $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^m$ are possibly nonlinear functions.

The ensemble square root filter algorithm is summarized in the following lines:

1. Initialization step.
   Generate a matrix of N initial ensemble members $\left[\hat{x}_{0|0}^{(1)}, \hat{x}_{0|0}^{(2)}, \ldots, \hat{x}_{0|0}^{(N)}\right]$ that properly represent the error statistics of the initial guess $x_{0|0}$ of the state.
2. For $k = 1, 2, \ldots$
   a) Forecast step
      i. Transform the ensemble members
      $$\hat{x}_{k|k-1}^{(i)} = f(\hat{x}_{k-1|k-1}^{(i)}, u_{k-1}, k-1) + w_{k-1}^{(i)} \quad i = 1, \ldots, N \tag{2.47}$$
      ii. Estimate the forecast state
      $$\hat{x}_{k|k-1} = \frac{1}{N} \sum_{i=1}^{N} \hat{x}_{k|k-1}^{(i)}$$
      iii. Compose the square root of $P_{k|k-1}$
      $$S_{k|k-1} = \frac{1}{\sqrt{N-1}} \left[\hat{x}_{k|k-1}^{(1)} - \hat{x}_{k|k-1}, \ldots, \hat{x}_{k|k-1}^{(N)} - \hat{x}_{k|k-1}\right] \tag{2.48}$$
   b) Analysis step
      i. Calculate the predicted measurements
      $$\hat{y}_k^{(i)} = h(\hat{x}_{k|k-1}^{(i)}, u_k, k) \quad i = 1, \ldots, N \tag{2.49}$$
      ii. Calculate the mean and error covariance of the predicted measurements
      $$\begin{aligned} \hat{y}_k &= \frac{1}{N} \sum_{i=1}^{N} \hat{y}_k^{(i)} \\ P_{y_k} &= \frac{1}{N-1} \sum_{i=1}^{N} \left(\hat{y}_k^{(i)} - \hat{y}_k\right) \left(\hat{y}_k^{(i)} - \hat{y}_k\right)^T + R_k \end{aligned} \tag{2.50}$$

iii. Update $S_{k|k-1}$ to $S_{k|k}$ and $\hat{x}_{k|k-1}$ to $\hat{x}_{k|k}$ using sequential processing

- $S^{(0)}_{k|k} = S_{k|k-1}$
- $\hat{x}_{k|k} = \hat{x}_{k|k-1}$
- for $j = 1, 2, \ldots, m$

$$\begin{aligned}
\mathcal{F}_k &= \frac{1}{\sqrt{N-1}}\left[\hat{y}^{(1)}_k - \hat{y}_k, \ldots, \hat{y}^{(N)}_k - \hat{y}_k\right]_{(j,:)} \\
\alpha_k &= \left(\mathcal{F}_k^T \mathcal{F}_k + P_{y_{k_{(j,j)}}}\right)^{-1} \\
K_{k_{(:,j)}} &= \alpha_k S^{(j-1)}_{k|k} \mathcal{F}_k \\
\hat{x}_{k|k} &= \hat{x}_{k|k} + K_{k_{(:,j)}} * \left(y_{k_{(j,1)}} - h\left(\hat{x}_{k|k-1}, u_k, k\right)_{(j,1)}\right) \\
\beta_k &= \frac{1}{1 + \sqrt{\alpha_k + R_{k_{(j,j)}}}} \\
\tilde{K}_{k_{(:,j)}} &= \beta_k K_{k_{(:,j)}} \\
S^{(j)}_{k|k} &= S^{(j-1)}_{k|k} - \tilde{K}_{k_{(:,j)}} * \mathcal{F}_k^T
\end{aligned} \tag{2.51}$$

where (l, m) denotes the lth row and mth column, $(l, :)$ the lth row an all columns and $(:, m)$ the mth column an all rows.

iv. Update the ensembles

$$\left[\hat{x}^{(1)}_{k|k}, \hat{x}^{(2)}_{k|k}, \ldots, \hat{x}^{(N)}_{k|k}\right] = \hat{x}_{k|k} \cdot \mathbf{1}_N + \frac{1}{\sqrt{N-1}} S_{k|k} \tag{2.52}$$

where $\mathbf{1}_N \in \mathbb{R}^{1\times N}$ is a vector of 1s.

End

2.9 The Ensemble Transform Kalman Filter - ETKF

The ensemble transform Kalman filter is another type of square root filter that does not add noise perturbations to the measurements. It is particularly interesting since it uses a fast method to calculate the prediction (do not confuse with forecast) error covariance matrix before any actual observations are taken. The algorithm is computationally faster than the ensemble square root filter (EnSRF) of the previous section. The ETKF algorithm that is summarized here, is taken from [1] with a minor adjustment to make the filter unbiased as advised in [12] and [2]. This adjustment is indicated with an $(*)$ in the algorithm.

Consider the discrete nonlinear model with

$$\begin{aligned}
x_{k+1} &= f(x_k, u_k, k) + w_k, \\
y_k &= h(x_k, u_k, k) + v_k, \\
w_k &\sim (0, Q_k), \\
v_k &\sim (0, R_k),
\end{aligned} \tag{2.53}$$

where the system equation $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^m$ are possibly nonlinear functions.

The ensemble transform Kalman filter algorithm can be briefly described as follows:

1. Initialization step.
   Generate a matrix of N initial ensemble members $\left[\hat{x}_{0|0}^{(1)}, \hat{x}_{0|0}^{(2)}, \ldots, \hat{x}_{0|0}^{(N)}\right]$ that properly represent the error statistics of the initial guess $x_{0|0}$ of the state.

2. For $k = 1, 2, \ldots$

   a) Forecast step

      i. Transform the ensemble members

$$\hat{x}_{k|k-1}^{(i)} = f(\hat{x}_{k-1|k-1}^{(i)}, u_{k-1}, k-1) + w_{k-1}^{(i)} \quad i = 1, \ldots, N \tag{2.54}$$

      ii. Estimate the forecast state (ensemble mean)

$$\hat{x}_{k|k-1} = \frac{1}{N} \sum_{i=1}^{N} \hat{x}_{k|k-1}^{(i)}$$

      iii. Compose the square root of $P_{k|k-1}$

$$S_{k|k-1} = \frac{1}{\sqrt{N-1}} \left[\hat{x}_{k|k-1}^{(1)} - \hat{x}_{k|k-1}, \ldots, \hat{x}_{k|k-1}^{(N)} - \hat{x}_{k|k-1}\right] \tag{2.55}$$

   b) Analysis step

      i. Calculate the predicted measurements

$$\hat{y}_k^{(i)} = h(\hat{x}_{k|k-1}^{(i)}, u_k, k) \quad i = 1, \ldots, N \tag{2.56}$$

      ii. Calculate the mean of the predicted measurements

$$\hat{y}_k = \frac{1}{N} \sum_{i=1}^{N} \hat{y}_k^{(i)} \tag{2.57}$$

      iii. Calculate $\mathcal{F}_k$ and $\mathcal{T}_k$

$$\mathcal{F}_k = \frac{1}{\sqrt{N-1}} \left[\hat{y}_k^{(1)} - \hat{y}_k, \ldots, \hat{y}_k^{(N)} - \hat{y}_k\right] L_{R_k}^{-T} \tag{2.58}$$
$$\mathcal{T}_k = \mathcal{F}_k^T \mathcal{F}_k + I_m$$

      where $R_k = L_{R_k} L_{R_k}^T$

      iv. Kalman Gain

$$K_k = S_{k|k-1} \mathcal{F}_k \mathcal{T}_k^{-1} \tag{2.59}$$

v. Estimate the analysis state (ensemble mean)

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \left(L_{R_k}^{-1} y_k - h\left(\hat{x}_{k|k-1}, u_k, k \right) \right) \tag{2.60}$$

vi. Calculate the eigenvalue decomposition of $\mathcal{F}_k \mathcal{F}_k^T$

$$\mathcal{F}_k \mathcal{F}_k^T = V_k \Lambda_k V_k^T \tag{2.61}$$

vii. Calculate $S_{k|k}$

$$S_{k|k} = S_{k|k-1} V_k \left(\Lambda_k + I_N \right)^{-1/2} V_k^T \tag{2.62}$$

(*) V_k^T is added to obtain unbiased filter

viii. Update the ensembles

$$\left[\hat{x}_{k|k}^{(1)}, \hat{x}_{k|k}^{(2)}, \ldots, \hat{x}_{k|k}^{(N)} \right] = \hat{x}_{k|k} \cdot \mathbf{1}_N + \frac{1}{\sqrt{N-1}} S_{k|k} \tag{2.63}$$

where $\mathbf{1}_N \in \mathbb{R}^{1 \times N}$ is a vector of 1s.

3. End

2.10 The Deterministic Ensemble Kalman Filter - DEnKF

This section contains the deterministic ensemble Kalman filter algorithm as described in [13]. Like the square root based algorithms the DEnKF does not use perturbated measurements. The main assumption of the algorithm is that only small corrections are made on the forecast. In this case, the EnKF without perturbated observations reduces the forecast error covariance matrix by an amount that is almost twice as large as needed. By using only half the theoretical Kalman gain in the analysis step to update the ensembles with a readjustment of the mean, the theoretical covariance matrix is almost asymptotically matched. Since, like the square root filters, the error covariance matrix of the analysis step does not depend on particular realizations of the measurements, the filter is called a deterministic filter. The DEnKF provides similar performance as the square root filters while it retains the simplicity of the EnKF.

Consider the discrete nonlinear model with

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, k) + w_k, \\ y_k &= h(x_k, u_k, k) + v_k, \\ w_k &\sim (0, Q_k), \\ v_k &\sim (0, R_k), \end{aligned} \tag{2.64}$$

where the system equation $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^m$ are possibly nonlinear functions.

The deterministic ensemble Kalman filter algorithm is as follows:

1. Initialization step.
   Generate an initial ensemble $\left[\hat{x}_{0|0}^{(1)}, \hat{x}_{0|0}^{(2)}, \ldots, \hat{x}_{0|0}^{(N)}\right]$ that properly represent the error statistics of the initial guess $x_{0|0}$ of the state.

2. For $k = 1, 2, \ldots$

   a) Forecast step

      i. Transform the ensemble members

$$\hat{x}_{k|k-1}^{(i)} = f(\hat{x}_{k-1|k-1}^{(i)}, u_{k-1}, k-1) + w_{k-1}^{(i)} \quad i = 1, \ldots, N \tag{2.65}$$

      ii. Estimate the forecast state (ensemble mean)

$$\hat{x}_{k|k-1} = \frac{1}{N} \sum_{i=1}^{N} \hat{x}_{k|k-1}^{(i)}$$

   b) Analysis step

      i. Calculate the predicted measurements

$$\hat{y}_k^{(i)} = h(\hat{x}_{k|k-1}^{(i)}, u_k, k) \quad i = 1, \ldots, N \tag{2.66}$$

      ii. Calculate the mean and error covariance of the predicted measurements and the cross-covariance

$$\hat{y}_k = \frac{1}{N} \sum_{i=1}^{N} \hat{y}_k^{(i)} \tag{2.67}$$

$$P_{y_k} = \frac{1}{N-1} \sum_{i=1}^{N} \left(\hat{y}_k^{(i)} - \hat{y}_k\right) \left(\hat{y}_k^{(i)} - \hat{y}_k\right)^T + R_k$$

$$P_{xy_k} = \frac{1}{N-1} \sum_{i=1}^{N} \left(\hat{x}_{k|k-1}^{(i)} - \hat{x}_{k|k-1}\right) \left(\hat{y}_k^{(i)} - \hat{y}_k\right)^T$$

      iii. Kalman Gain

$$K_k = P_{xy_k} P_{y_k}^{-1} \tag{2.68}$$

      iv. Estimate the analysis state (ensemble mean)

$$\tilde{x}_{k|k} = 0.5 * K_k \left(y_k - h(\hat{x}_{k|k-1}, u_k, k)\right) \tag{2.69}$$

$$\hat{x}_{k|k}^{(i)} = \hat{x}_{k|k-1}^{(i)} + 0.5 * K_k \left(y_k - \hat{y}_k^{(i)}\right) + \tilde{x}_{k|k} \quad i = 1, \ldots, N \tag{2.70}$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + 2 * \tilde{x}_{k|k}$$

3. End

2.11 The Generic Particle Filter - PF-GEN

Particle filters are a Monte Carlo implementation of the recursive Bayesian filter as illustrated in Section 2.2. The particle filter uses random samples, named particles to approximate the required pdf's. This approximation can be improved by increasing the amount of particles. In case of highly nonlinear and/or non-Gaussian noise sources they often perform better than the Kalman based filters, although typically at a much higher computational cost.

The main principles of the generic particle filter are now roughly explained. For more detailed information on the implemented algorithm see [6]. In the forecast step, the particles are propagated through the system equation to generate a new set of particles. In the analysis step each particle is assigned a weight $q^{(i)}$ by using the measurement equation. These weight are the likelihoods of the measurement for each particle ($p(y_k|x_k)$ of equation (2.9)). Next, the assigned weights are normalized to sum up to one. The particles and weights now represent the posterior pdf ($p(x_k|y_{1:k})$ of equation (2.9) and can be used to estimate whatever desired statistics (most often mean and covariance).

The algorithm as described thus far is in fact the Sequential Importance Sampling (SIS) algorithm. A common problem with the SIS particle filter is however the degeneracy phenomenon. Degeneracy means that the likelihood of most particles becomes close to zero after a few iterations. Its impact can be evaluated using the formula of the estimated effective sample size $\hat{N}_{eff}$. To reduce the effect of degeneracy, the generic particle filter algorithm applies resampling when $\hat{N}_{eff}$ is below some threshold N_T. The general idea of resampling is to remove particles with small weights and to keep particles with large weights.

There are four resampling algorithms taken under consideration. Each of them make use of a basic algorithm which selects the N new particles $\hat{x}_{k|k}^{(j)}$ from the previous N particles $\hat{x}_{k|k-1}^{(i)}$ such that $P\left(\hat{x}_{k|k}^{(j)} = \hat{x}_{k|k-1}^{(i)}\right) = q^{(i)},\ i,j = 1,\ldots,N$. This is implemented in two steps:

1. Generate a random number r that is uniformly distributed on $[0,1]$.
2. Assign the particle $\hat{x}_{k|k}^{(j)}$ the value of the particle $\hat{x}_{k|k-1}^{(i)}$ according to $Q^{(i-1)} < r \leq Q^{(i)}$ where $Q^{(i)} = \sum_{m=1}^{i} q^{(m)}$.

This procedure is illustrated in in Figure 2.1.

Now, the four resampling algorithms are presented. A more detailed study of these algorithms can be found in [4]. An important remark is that after resampling all the weights are set equal, i.e. $q^{(i)} = \frac{1}{N}\ \forall i$.

- Multinomial resampling
  This resampling algorithm is often also referred to as simple resampling. The

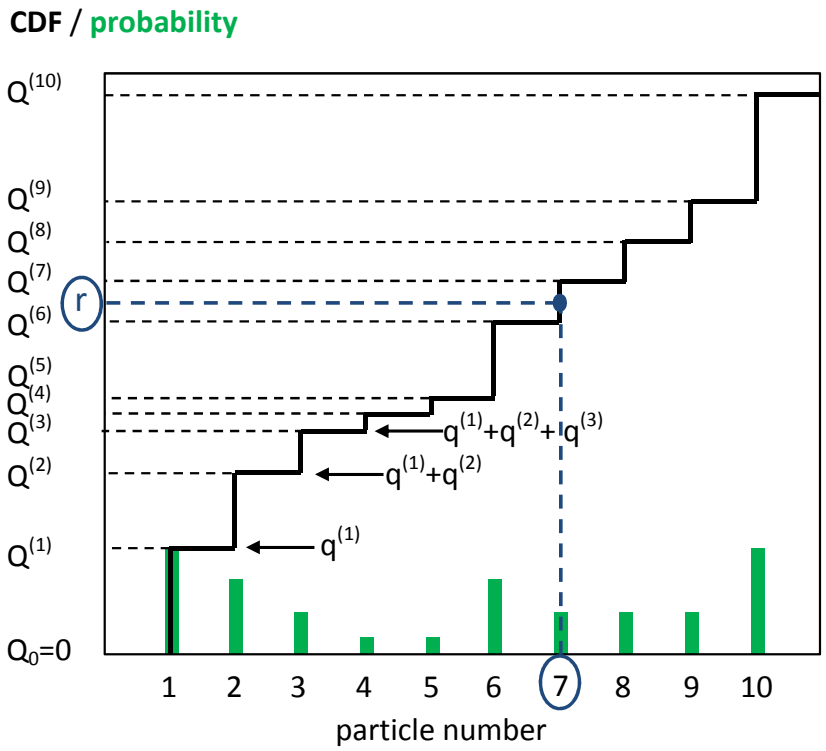


Figure 2.1: Illustration of the basic particle resample algorithm.

main idea is to generate N ordered random sampled uniform numbers r_l

$$r_l = r_{l+1}\tilde{r}_l^{\frac{1}{N}}, \quad r_N = \tilde{r}_N^{\frac{1}{N}}, \text{ with } \tilde{r}_l \sim \mathcal{U}(0,1), \tag{2.71}$$

and then use the basic algorithm to compose the analyzed particles by applying

$$\hat{x}_{k|k}^{(j)} = \hat{x}_{k|k-1}^{(i)}, \text{ with } i \text{ according to } Q^{(i-1)} < r_l \leq Q^{(i)}. \tag{2.72}$$

- Stratisfied resampling
  This algorithm uses a different way to generate the N ordered random numbers. It uses a comb of equally spaced numbers with an interval of $1/N$ and adds N random sampled numbers to it from the uniform distribution $\mathcal{U}(0, 1/N)$. Hence, N ordered random sampled uniform numbers are generated

$$r_l = \frac{(l-1) + \tilde{r}_l}{N}, \text{ with } \tilde{r}_l \sim \mathcal{U}(0,1), \tag{2.73}$$

and the basic algorithm is used to compose the analyzed particles by setting

$$\hat{x}_{k|k}^{(j)} = \hat{x}_{k|k-1}^{(i)}, \text{ with } i \text{ according to } Q^{(i-1)} < r_l \leq Q^{(i)}. \tag{2.74}$$

- Systematic resampling
  The systematic resampling is very similar to the stratisfied resampling, but has the advantage that it only uses one random sample. It also generates the comb of equally spaced numbers but translates this with an offset taken from the uniform distribution $\mathcal{U}(0, 1/N)$. As such, the ordered numbers are generated

$$r_l = \frac{(l-1) + \tilde{r}}{N}, \text{ with } \tilde{r} \sim \mathcal{U}(0,1), \tag{2.75}$$

  and the basic algorithm composes the analyzed particles by applying

$$\hat{x}_{k|k}^{(j)} = \hat{x}_{k|k-1}^{(i)}, \text{ with } i \text{ according to } Q^{(i-1)} < r_l \leq Q^{(i)}. \tag{2.76}$$

- Residual resampling
  Residual resampling uses a different approach to resampling. Instead of immediately resampling, it first restricts $n = \sum_i^N \lfloor N q_i \rfloor$ particles and uses one of the previous resampling algorithms to regain the original population size. The two steps of the algorithm are:

  - The restricted particles are selected with a modified basic algorithm that avoids the usage of random samples. First, a set of unity spaced numbers $\tilde{n} = \{1, 2, \ldots, n_l, \ldots, n-1, n\}$ is generated. Then, the particles that need to be restricted are selected by setting $\hat{x}_{k|k}^{(j)} = \hat{x}_{k|k-1}^{(i)}$, with i according to $\tilde{N}^{(i-1)} < \tilde{n}_l \leq \tilde{N}^{(i)}$, where $\tilde{N}^{(i)} = \sum_{m=1}^{i} \left\lfloor N q^{(m)} \right\rfloor$.
  - The $N-n$ remaining particles are obtained by applying one of the previous resampling algorithms using the forecast particles $\hat{x}_{k|k-1}^{(i)}$ and remaining weights $\tilde{q}^{(i)} = \frac{N q^{(i)} - \lfloor N q^{(i)} \rfloor}{\sum_{m=1}^{N} \left(N q^{(m)} - \lfloor N q^{(m)} \rfloor \right)}$.

If a resampling algorithm is required by a particle filter algorithm, it is indicated with the syntax

$$\left[\left\{ \hat{x}_{k|k}^{(i)}, q_k^{(i)} \right\}_{i=1}^{N}, index \right] = resample \left[\left\{ \hat{x}_{k|k-1}^{(i)}, q_k^{(i)} \right\}_{i=1}^{N} \right]$$

where *resample* denotes any of the resample algorithms, $\left\{ \hat{x}_{k|k}^{(i)}, q_k^{(i)} \right\}_{i=1}^{N}$ denotes the entire set of particles and their corresponding weights and *index* is a vector that contains the index of the resampled particles. For example, $index = [1 \;\; 2 \;\; 1 \;\; 2 \;\; 5]$ indicates that the first particle is resampled to locations one and three, the second particle to locations two and four, while the fifth particle remains at the same location; the particles three and four are not retained after resampling. An unused input or output argument of *resample* is indicated by $\sim$.

Now that the generic particle filter is completely described, the algorithm is summarized while considering the discrete nonlinear system

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, k) + w_k, \\ y_k &= h(x_k, u_k, k) + v_k, \end{aligned} \tag{2.77}$$

where the system equation $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^m$ are possibly nonlinear functions, and $\{w_k\}$ and $\{v_k\}$ are independent white noise processes with known pdf's. It is important to note that, unlike the Kalman based filters, only the pdf 's need to be known. As such the particle filter can handle a much wider range distributions.

The generic particle filter algorithm can be summarized as follows:

1. Initialization step.
   Generate an initial set of particles $\left[\hat{x}_{0|0}^{(1)}, \hat{x}_{0|0}^{(2)}, \ldots, \hat{x}_{0|0}^{(N)}\right]$ that properly represent the pdf of the initial state $x_{0|0}$.

2. For $k = 1, 2, \ldots$

   a) Forecast step

      i. Forecast the particles

$$\hat{x}_{k|k-1}^{(i)} = f(\hat{x}_{k-1|k-1}^{(i)}, u_{k-1}, k-1) + w_{k-1}^{(i)} \quad i = 1, \ldots, N \tag{2.78}$$

   b) Analysis step

      i. Assign each particle a weight $q_k^{(i)} \propto p(y_k|\hat{x}_{k|k-1}^{(i)})$

$$q_k^{(i)} = pdf\left(y_k - h(\hat{x}_{k|k-1}^{(i)}, u_k, k)\right) q_{k-1}^{(i)} \quad i = 1, \ldots, N \tag{2.79}$$

      ii. Normalize the weights

$$q_k^{(i)} = \frac{q_k^{(i)}}{\sum_{m=1}^{N} q_k^{(m)}} \tag{2.80}$$

      iii. Calculate the estimated effective sample size

$$\hat{N}_{eff} = \frac{1}{\sum_{i=1}^{N} (q_k^{(i)})^2} \tag{2.81}$$

      iv. Resample in case of degeneration
      If $\hat{N}_{eff} < N_T N$,

$$\left[\left\{\hat{x}_{k|k}^{(i)}, q_k^{(i)}\right\}_{i=1}^{N}, \sim\right] = resample\left[\left\{\hat{x}_{k|k-1}^{(i)}, q_k^{(i)}\right\}_{i=1}^{N}\right], \tag{2.82}$$

      else

$$\hat{x}_{k|k}^{(i)} = \hat{x}_{k|k-1}^{(i)}, \tag{2.83}$$

      end.
      With $N_T \in [0, 1]$ the resample threshold and where $\sim$ indicates an unused return argument.

v. Estimate the analysis state (weighted sum of particles)

$$\hat{x}_{k|k} = \frac{1}{N}\sum_{i=1}^{N} q_k^{(i)} \hat{x}_{k|k}^{(i)}$$

3. End

2.12 The Sampling Importance Resampling Particle Filter - PF-SIR

The SIR filter is almost identical to the generic filter of the previous section. The only difference is that the resampling step is executed every time. The downside of this filter is that resampling every iteration can cause a loss of diversity in the particles. Although almost identical to the generic particle filter, the SIR filter algorithm is also included for the reader's convenience. For more information see [6].

Consider the discrete nonlinear system

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, k) + w_k, \\ y_k &= h(x_k, u_k, k) + v_k, \end{aligned} \tag{2.84}$$

where the system equation $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^m$ are possibly nonlinear functions, and $\{w_k\}$ and $\{v_k\}$ are independent white noise processes with known pdf's.

The sampling importance resampling particle filter algorithm is presented in the following lines:

1. Initialization step.
   Generate an initial set of particles $\left[\hat{x}_{0|0}^{(1)}, \hat{x}_{0|0}^{(2)}, \ldots, \hat{x}_{0|0}^{(N)}\right]$ that properly represent the pdf of the initial state $x_{0|0}$.

2. For $k = 1, 2, \ldots$

   a) Forecast step

      i. Forecast the particles

$$\hat{x}_{k|k-1}^{(i)} = f(\hat{x}_{k-1|k-1}^{(i)}, u_{k-1}, k-1) + w_{k-1}^{(i)} \quad i = 1, \ldots, N \tag{2.85}$$

   b) Analysis step

      i. Assign each particle a weight $q_k^{(i)} = p(y_k|\hat{x}_{k|k-1}^{(i)})$

$$q_k^{(i)} = pdf\left(y_k - h(\hat{x}_{k|k-1}^{(i)}, u_k, k)\right) \quad i = 1, \ldots, N \tag{2.86}$$

ii. Normalize the weights

$$q_k^{(i)} = \frac{q_k^{(i)}}{\sum_{m=1}^{N} q_k^{(m)}} \tag{2.87}$$

iii. Resample

$$\left[\left\{\hat{x}_{k|k}^{(i)}, q_k^{(i)}\right\}_{i=1}^{N}, \sim\right] = resample\left[\left\{\hat{x}_{k|k-1}^{(i)}, q_k^{(i)}\right\}_{i=1}^{N}\right], \tag{2.88}$$

Where $\sim$ indicates an unused return argument.

iv. Estimate the analysis state (weighted sum of particles)

$$\hat{x}_{k|k} = \frac{1}{N}\sum_{i=1}^{N} q_k^{(i)} \hat{x}_{k|k}^{(i)}$$

3. End

2.13 The Auxiliary Sampling Importance Resampling Particle Filter - PF-ASIR

The resampling step of the generic and SIR filters reduce the degeneracy problem. However, another problem known as sample impoverishment is still present. This effect is caused by the fact that particles with high weights are selected many times which leads to a loss of diversity. As a consequence, distributions with long tails are badly approximated. The ASIR filter solves the sample impoverishment by introducing an auxiliary variable $\mu_k^{(i)}$, hence the name auxiliary particle filter. The variable $\mu_k^{(i)}$ is a statistical characterization of x_k based on $\hat{x}_{k|k-1}^{(i)}$. In this case the mean is selected in which case $\mu_k^{(i)} = \mathbb{E}[x_k|\hat{x}_{k|k-1}^{(i)}]$.

To understand the basic principle of the ASIR, recall that the main idea of the SIR is to propagate all particles $\hat{x}_{k|k-1}^{(i)}$ unconditionally, calculate the weights $q_k^{(i)} = p(y_k|\hat{x}_{k|k-1}^{(i)})$ of the obtained particles and finally resample using these weights. The ASIR has two adjustments. First, it does not propagate all particles unconditionally. Instead it uses the available measurement and the auxiliary variable to determine the most likely particles, and forecasts only the more likely particles. Second, it uses the auxiliary variable to adjust the weights of the particles to $q_k^{(i)} = \frac{p(y_k|\hat{x}_{k|k-1}^{(i)})}{p(y_k|\mu_k^{(i)})}$. The denominator of this adjusted weights tends to decrease $q_k^{(i)}$ for highly likely particles and increase $q_k^{(i)}$ for highly unlikely particles. As such the diversity of the particles is improved [15].

When the process noise is small the ASIR often provides a better performance than the SIR. However, when dealing with large process noise the single point

$\mathbb{E}[x_k|\hat{x}^{(i)}_{k|k-1}]$ does not provide a good characterization of $p(x_k|\hat{x}^{(i)}_{k|k-1})$. In this case the ASIR provides a worse result. For more information see [6].

Consider the discrete nonlinear system

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, k) + w_k, \\ y_k &= h(x_k, u_k, k) + v_k, \end{aligned} \tag{2.89}$$

where the system equation $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^m$ are possibly nonlinear functions, and $\{w_k\}$ and $\{v_k\}$ are independent white noise processes with known pdf's.

The auxiliary sampling importance resampling filter algorithm is as follows:

1. Initialization step.
   Generate an initial set of particles $\left[\hat{x}^{(1)}_{0|0}, \hat{x}^{(2)}_{0|0}, \ldots, \hat{x}^{(N)}_{0|0}\right]$ that properly represent the pdf of the initial state $x_{0|0}$.

2. For $k = 1, 2, \ldots$

   a) Forecast step

      i. Calculate $\mu^{(i)}_k$

$$\mu^{(i)}_k = f(\hat{x}^{(i)}_{k-1|k-1}, u_{k-1}, k-1) \quad i = 1, \ldots, N \tag{2.90}$$

      ii. Assign each auxiliary variable a weight

$$q^{(m)}_{\mu_k} = pdf\left(y_k - h(\mu^{(i)}_k, u_k, k) q^{(i)}_{k-1}\right) \quad i = 1, \ldots, N \tag{2.91}$$

      iii. Normalize the weights

$$q^{(i)}_k = \frac{q^{(m)}_{\mu_k}}{\sum_{m=1}^{N} q^{(m)}_{\mu_k}} \tag{2.92}$$

      iv. Calculate the resample index

$$[\sim, \sim, ind] = resample\left[\left\{\hat{x}^{(i)}_{k|k-1}, q^{(i)}_k\right\}_{i=1}^{N}\right], \tag{2.93}$$

      Where $\sim$ indicates an unused return argument.

      v. Forecast the most likely particles

$$\hat{x}^{(i)}_{k|k-1} = f(\left(\hat{x}^{(i)}_{k-1|k-1}\right)_{(:,ind)}, u_{k-1}, k-1) + w^{(i)}_{k-1} \quad i = 1, \ldots, N \tag{2.94}$$

      where $(:, ind)$ selects all particles according to the index vector

   b) Analysis step

i. Assign each particle a weight

$$q_k^{(i)} = \frac{pdf\left(y_k - h(\hat{x}_{k|k-1}^{(i)}, u_k, k)\right)}{\left(q_{\mu_k}^{(m)}\right)_{(:,ind)}} \quad i = 1, \ldots, N \tag{2.95}$$

where $(:, ind)$ selects all likelihoods according to the index vector

ii. Normalize the weights

$$q_k^{(i)} = \frac{q_k^{(i)}}{\sum_{m=1}^{N} q_k^{(m)}} \tag{2.96}$$

iii. Estimate the analysis state (mean of particles)

$$\hat{x}_{k|k} = \frac{1}{N} \sum_{i=1}^{N} \hat{x}_{k|k}^{(i)}$$

3. End

2.14 Conclusion

In this chapter the main aspects of the different filters that are implemented in the toolbox have been presented.

The Kalman filter provides the optimal state estimation, but only in some restricted cases. The most critical restriction is that it can not be applied to nonlinear systems which makes it unsuitable for a wide range of applications. The extended Kalman filter solves this by a linearization of the system. The most important downside of this approach is that it requires the Jacobians of the system equations which is very expensive and error-prone. Furthermore if often provides poor results in case of highly nonlinear systems. The unscented Kalman filter solves both issues by propagating deterministically selected sigma points through the state equations. Still, the amount of required sigma points has the same order as the amount of states. As such, none of the previously mentioned filters are fitted for large-scale systems. The optimal interpolation technique is suited for large-scale systems since it uses a fixed error covariance matrix which allows pre-calculation of the Kalman gain when the measurement equation is linear.

The ensemble Kalman filter is designed especially to estimate the states of large-scale systems. Although it also propagates samples through the system equations it is possible to select the amount of ensemble members. This allows the user to choose a proper balance between computational load and quality of the state estimation. Also, the members are not selected deterministically, but as random samples from the pdf of the true state. The downside of the ensemble Kalman filter is that it uses perturbated observations to correctly estimate the analysis error covariance matrix. This solutions is however only valid in a statistical sense. The ensemble square root

filter, the ensemble transform Kalman filter and the deterministic ensemble Kalman filter offer a deterministic alternative to the perturbated observations. The ensemble square root filter provides improved accuracy at the same computational effort, but only under the restriction of uncorrelated measurement noise. The ensemble transform Kalman filter is a fast variant of square root filters. The deterministic Kalman filter provides similar performance as square root filters while preserving the convenient structure of the EnKF, although restricted by the assumption of small forecast corrections.

The last class of filters, the particle filters, are not Kalman based filters but brute force implementations of the recursive Bayesian filter. Since typically a very large amount of particles is needed they are not suited for large-scale systems. However, in case of highly nonlinear or non-Gaussian noise or both, they often provide better approximations than the Kalman based filters. The generic particle filter only resamples when the estimated number of effective particles is under a predefined threshold. The SIR filter resamples every iteration which can cause a rapid loss of diversity in the particles. Finally, the auxiliary SIR filter uses an auxiliary variable to prevent sample impoverishment.

Chapter 3

Data Assimilation Toolbox

This chapter provides a user guide to the data assimilation toolbox which is completely developed in the MATLAB programming environment. The first section provides some practical guidelines on how to get acquainted with the toolbox. The second section illustrates why the object-oriented approach is selected and how the different classes work together to achieve a straightforward data assimilation procedure. The way the toolbox deals with error handling is also discussed in this section. The last sections summarize the most important properties and methods of all classes and is accompanied by some illustrative examples.

3.1 Introduction to the toolbox

The best way to master the data assimilation toolbox is to actively use it while progressing through the different sections in this chapter. This is why this first section describes how to start using the toolbox. To install the toolbox please consult Appendix A. Besides the installation, also the main folder structure is listed in this appendix. Although this chapter provides a rather extensive summary of the main features of the toolbox it is by no means intended as a complete reference manual. To gain more in depth information about certain classes or functions please consult their help files. To obtain an overview of the main help files enter `dahelp` in the command line. Another useful source of information are the numerous demonstrations that are included in the toolbox. Each class and their most important methods are interactively explained with these demonstrations. For a complete summary enter `dademos` in the command line. If for some reason the coherence between the different classes becomes unclear, it can be useful to go through Chapter 4 which contains complete implementation examples. These examples clearly show the practical sequence of applying the different classes and their methods.

Notice that this chapter uses a somewhat different notation for some symbols than the more official notation used in Chapter 2. This new notation stems from the MATLAB notation used in the toolbox. To avoid any confusion this new notation is used throughout the entire chapter to indicate that the symbols refer to MATLAB

code. For example, the forecast state estimation $\hat{x}_{k|k-1}$ becomes `xf` and the analysis state estimation $\hat{x}_{k|k}$ is now indicated by `xa`. The notations `Pf` and `Pa` are completely similar. Likewise, the initial state estimation which consists of $\hat{x}_{0|0}$ and $P_{0|0}$ is denoted with `x0` and `P0`.

3.2 The general architecture

3.2.1 Object-Oriented Programming in Matlab

Matlab has two main programming styles: procedural programming and object-oriented programming. Procedural programming uses functions to handle the necessary operations on data. The structure of such a program is often sequential, where a series of functions receive input data and return the modified data. It is safe to state that this is the far most widely used approach in Matlab although the object-oriented approach is also completely supported. While in procedural programming the design is mostly focused on the different steps to complete, the object-oriented programming style is focused on collecting data and functions in objects. Each object manages and modifies its own data and interacts with other objects or functions through the object's interface.

The goal of the toolbox is to provide a generic environment to incorporate any data assimilation technique for any user-defined state space model. The object-oriented programming possibilities of Matlab provide an excellent framework to attain such a generic toolbox. By defining the main objects as classes, a clear program structure is obtained that can be easily expanded with additional filters, state space models, etc. An additional advantage of the object oriented approach is that the structure of the program is automatically guarded against a chaotic growth of functions.

In this manuscript it is assumed that the reader is familiar with procedural programming in Matlab and that the reader has some basic knowledge of object-oriented programming. More detailed information about programming in Matlab can be found in [8] and [7]. Some frequently used object-oriented terms are refreshed for the reader's convenience. At the top level is a class which describes a set of objects with common characteristics. These objects are instances of a class, which contain the actual data values stored in the objects' properties. Methods are functions, defined by its class, that are performed on the class objects. They govern the behavior of an object in a way that is common to all objects of a class. Subclasses (child classes) are classes that are derived from a superclass (parent class). They inherit all the methods and properties from their superclasses.

3.2.2 The data assimilation workflow

The best way to introduce the different classes and their most important methods is to illustrate the typical workflow of a data assimilation experiment. This way it is

immediately clear why some parts of the toolbox are configured as classes and how the different pieces nicely fit together. This section is considered vital to obtain a good understanding of the toolbox. If something is not clear about the subsequent sections it might be a good time to run through this section once more. The typical data assimilation workflow is shown in Figure 3.1. The workflow starts at the top of the schematic and goes in several steps to the bottom.

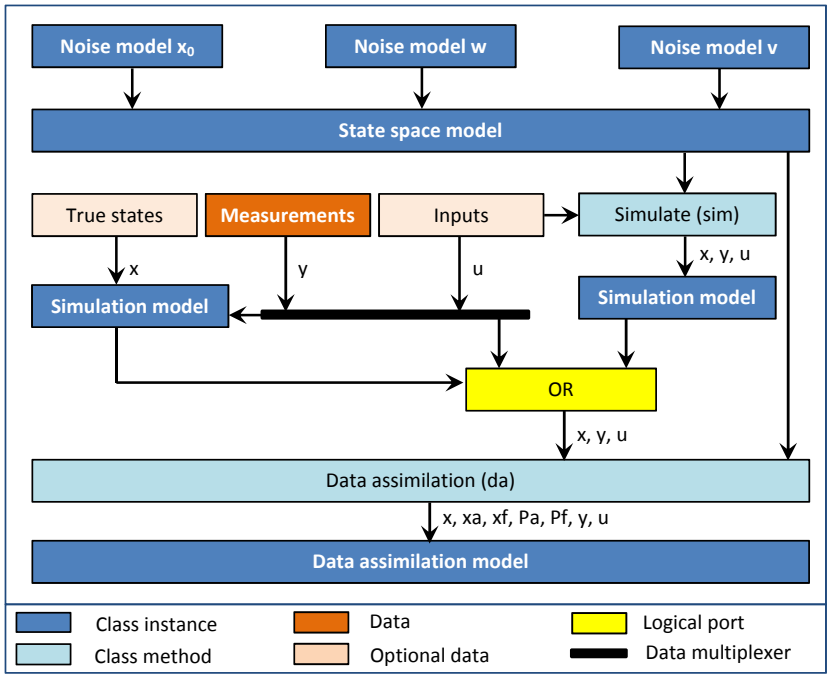


FIGURE 3.1: Illustration of the workflow of a typical data assimilation experiment.

The first step of a data assimilation experiment is defining the noise models. A noise model is a class that represents any kind of probability distribution. At this moment only the Gaussian noise model, as described in Section 3.3, is defined in the toolbox. Extending the toolbox with additional distributions is possible in a straightforward manner. A state space model requires three noise models: the initial state estimation `x0`, the process noise `w` and the measurement noise `v`. While it is clear that the last two are noise models, the initial state estimation might be less straightforward. A more common approach would be to use the initial state estimation and error covariance estimation as separate input arguments. There are however some benefits from combining them into a noise model. First of all, the correctness of the parameters is automatically checked by the noise model. But more importantly, all methods that are defined for a noise model are now instantly available to the initial state. For example, the sample method of this initial state model is used in the simulation method to obtain a simulation model. How these methods work is explained in further detail in the subsequent sections.

The second step is to configure a state space model which is another class of the toolbox. At the moment there are three state space models defined in the toolbox. These are the ones already encountered in the first chapter, namely the discrete-time linear, discrete-time nonlinear with additive noise and discrete-time nonlinear models as defined in equations (2.11), (2.17) and (2.30). Additional parameters that need to be set are the system equations, sampling time, etc. These settings are explained more detailed in Section 3.4.

Once the state space model is properly configured, there are two possible ways to proceed which depend on the nature of the experiment. Either the experiment is based on a real setup where measurements are available or the experiment has a more theoretical nature and no measurements are available. In the first case, the real measurements can be used in the data assimilation method. In the last case the measurements can be obtained by using the simulation method of the discrete-time state space models. The simulation method uses the state space model to simulate both states and measurements. It starts by drawing a sample from the initial state estimate `x0` and than runs sequentially through the system and measurement equations, while adding respectively samples from the process and measurement noise. This method is described more detailed in Section 3.4.4. The possible outputs of the simulation method are the true states `x`, the measurements `y` and the inputs `u` (if applicable). When calling the simulation method, the user can define which variables should be saved in a simulation model. The simulation model class can be regarded as an advanced Matlab structure. The advantage of saving the data in an object is that its methods allow fast post-processing of the retrieved data, e.g. by generating predefined plots. Now, the simulation model can be presented to the data assimilation technique which extracts required data such as measurements, inputs and true states. Note that when measurements and true states are available separately, this data should also be combined in a simulation object.

Now that the model is completely configured and all required data is available, the data can be assimilated. How to configure and apply these filters is thoroughly explained in Section 3.6. Note that all available assimilation techniques have already been summarized theoretically in Chapter 2. The default output of the data assimilation method is the matrix of analysis states `xa`. If desired, also the true states `x` (if available), the forecast states `xf`, the analysis error covariances `Pa`, the forecast error covariances `Pf`, the measurements `y` and the inputs `u` (if applicable) can be requested. It is important to note that only the variances of the covariance matrices are returned by default, since the matrices become huge when dealing with large-scale systems. Only upon specific request of the user, the entire covariance matrices are provided. As in the case of the simulation model, the requested variables are saved in a data assimilation model. The advantage of defining such a class to hold all desired data is again clear. The data assimilation object holds all desired information regarding a data assimilation test. As such interesting plots can be made such as comparison plots of estimated states versus true states, plots of the estimated states with their 95% confidence intervals or RMSE calculations of the different states. Notice that

this is the only reason why the true state `x` is provided as a possible input argument to the data assimilate method. More information about the data assimilation class can be found in Section 3.7.

3.2.3 The different classes

This section present all the classes that are currently defined in the toolbox. Figure 3.2 gives a clear overview of all classes and their inheritance relations. The simulation models only consist of class sim_D and the data assimilation models only consist of class dam_D at the moment. They are of little interest in this section. Please see sections 3.5 and 3.7 for their description. The noise models and state space models are the main focus of this section since the advantages of their inherited structures are explained here.

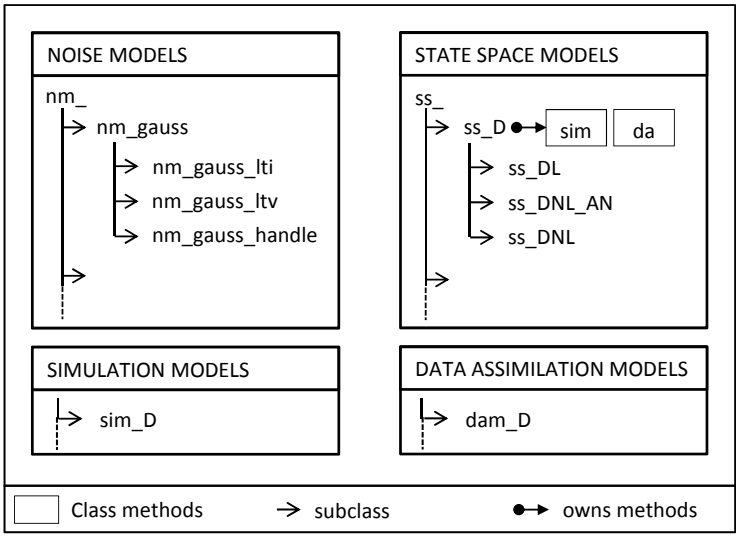


FIGURE 3.2: Illustration of the different classes and their inheritance.

The structure of the noise models starts with the superclass nm_ which is the parent of all noise model classes. This single parent contains only one child at the moment, the Gaussian noise class (nm_gauss) which is sufficient for the majority of the cases. Still, other noise distributions can easily be added under the superclass nm_. For example, a noise class nm_gauss_ht with a heavy tailed Gaussian distribution could be added to investigate its influence on the performance of a filter. The Gaussian noise class in its turn is the parent of three other classes: the linear time-invariant Gaussian noise class (nm_gauss_lti), the linear time-variant Gaussian noise class (nm_gauss_ltv) and the function handle Gaussian noise class (nm_gauss_handle). The LTI variant is the most straightforward, its mean (`mu`) and covariance matrix (`Sigma`) are configured with a vector and a matrix. The LTV variant provides time-variant behaviour by allowing `mu` or `Sigma`, or both, to be defined by a three dimensional array where the third dimension represents different step numbers. The function handle class is the most general variant since for this

class mu and Sigma are defined by user-defined functions. A first advantage of the noise model structure is that a state space object can easily check if an input argument is in fact a noise model. It only needs to validate if the input argument is of type nm_ instead of inquiring all possible noise model classes. Likewise, a filter only needs to validate whether an input argument is of type nm_gauss to confirm that the input is a Gaussian noise model. A second advantage is that all methods of the Gaussian noise models have an identical interface. For example, to draw `n` samples from a Gaussian noise model `nmObj` at step `k` the command `sample(nmObj,n,k)` is used, regardless of the type of noise model. If the noise model is LTI it disregards the argument `k`. It is strongly recommended to keep this transparency when extending the toolbox with new noise classes. More detailed information about the noise models is provided in Section 3.3.

The structure of the state space models is similar to the structure of the noise models. It consists of the superclass ss_ which is the parent of all other state space model classes. This parent currently holds only one child class, namely the discrete-time state space class (ss_D). As indicated in Figure 3.2, the simulation and data assimilation methods belong to this superclass. The ss_D class has three child classes: the discrete-time linear class (ss_DL), discrete-time nonlinear with additive noise class (ss_DNL_AN) and discrete-time nonlinear class (ss_DNL) which correspond to equations (2.11), (2.17) and (2.30). The system matrices of the linear class can be configured using either a matrix or a three dimensional array in a similar fashion as the LTV noise model. The equations of the nonlinear classes are always defined by user-defined functions. A function handle that represents `f` is provided for the system equation and a function handle that represents `h` is used for the measurement equation. Just like the structure of the noise model classes, the structure of the state space model classes have the advantage that it is straightforward to verify if an input argument is a discrete state space model. Also the transparent calling of methods is applicable to these models. For example, to obtain a measurement update of forecast state xf using state space model `ssObj` while applying input `u` and noise `v` at step `k`, the command `eval_htot(ssObj,xf,k,u,v)` is used. This syntax is correct regardless of which specific type the state space model `ssObj` is. Additional features are of course present. For instance when `v` is empty or not supplied, a sample is taken from the distribution of `v`. More detailed information about the state space models is provided in Section 3.4.

3.2.4 Error handling

The toolbox is equipped with an extensive error handling mechanism. Whenever a class is constructed, its input arguments are checked in great detail. Instead of providing a general error, detailed error information is always provided. Each input parameter is checked in several stages and the error indicates exactly what is wrong: an invalid class type, a wrong dimension, an impossible value, etc.

The different data assimilation techniques are equipped with the same kind of

error handling and if necessary they express warnings to the user. The main goal of these warnings is to make the user aware of possible mistakes. For example, when using a Kalman filter and the measurement noise model is not zero-mean, the user is advised that the state estimations will be less good than when using zero mean noise.

3.3 The Gaussian noise models

This section describes the Gaussian noise model classes. First, the properties and construction methods are summarized for each specific type of class. The class methods are presented at the end since all models have the same methods and since their syntax allows to address them identically. As indicated before, additional information can be found in the help files and demos of the toolbox.

3.3.1 Linear Time Invariant Gaussian noise (nm_gauss_lti)

The linear time invariant Gaussian noise model (nm_gauss_lti) is the most simple noise model class. For this reason it is advised to use this noise model instead of the function handle variant to configure LTI noise. It can be configured faster and, more importantly, it has the advantage that some numerically expensive calculations only need to be made once.

The main properties of the nm_gauss_lti class are:

- `mu` - the mean of the distribution which is defined as a column vector.
- `Sigma` - the covariance matrix of the distribution which can be defined in two ways. The first possibility a symmetric matrix, but if the noise is uncorrelated a column vector that contains the variances is also allowed.

The most common ways to construct a nm_gauss_lti class are:

- `obj = nm_gauss_lti(mu,Sigma)` - creates a LTI noise object `obj` with mean `mu` and covariance matrix `Sigma`.
- `obj = nm_gauss_lti(mu)` - creates a LTI noise object `obj` with mean `mu` and unit covariance matrix (identity matrix) `Sigma`.
- `obj = nm_gauss_lti(Sigma)` - creates a LTI noise object `obj` with zero mean and covariance matrix `Sigma`. To avoid any ambiguity with the previous syntax, `Sigma` can not be a column vector in this case.

To consult or adjust a property of an object, the syntax `obj.property` is used. For example: `obj.mu` shows the value of `mu`.

Example 3.3.1. Construction of an object of class nm_gauss_lti

To create a zero mean Gaussian distribution with covariance matrix $F = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$ the following command is used: `obj = nm_gauss_lti([1 2;2 1])`

3.3.2 Linear Time Variant Gaussian noise (nm_gauss_ltv)

The linear time variant Gaussian noise model class (nm_gauss_ltv) is very similar to the nm_gauss_lti class. To introduce the time variant behaviour the matrices can be configured as three-dimensional arrays where the third dimension corresponds to predefined step numbers.

The main properties of the nm_gauss_ltv class are:

- `mu` - the mean of the distribution which can be defined in two ways. When time invariant, `mu` is defined with a column vector as in class nm_gauss_lti . In case of a time variant behaviour, `mu` is configured as a 3D-array of column vectors where each vector corresponds to the step number as defined in the property `kIndex`.
  For example: an array of size $[2 \times 1 \times 3]$ corresponds to a LTV bivariate mean which requires three different step numbers in `kIndex`.

- `Sigma` - the covariance matrix of the distribution which can be defined in two ways. When time invariant, `Sigma` is defined with either a matrix or a column vector (when uncorrelated) as in class nm_gauss_lti. In case of a time variant behaviour, `Sigma` is configured as a 3D-array of either column vectors or matrices where each vector/matrix corresponds to the step number as defined in the property `kIndex`.
  For example: an array of size $[2 \times 2 \times 4]$ corresponds to a LTV bivariate covariance matrix which requires four different step numbers in `kIndex`.

- `kIndex` - vector that contains the step numbers that correspond to each vector or matrix in the 3D-arrays `mu` and/or `Sigma`. When not specified, the default content is set to $[0 \;\; 1 \;\; \ldots \;\; l]$ where l is the third dimension of the 3D-array.
  For example: when `mu` is defined as an array of size $[2 \times 1 \times 3]$ with mean vectors $[1 \;\; 1]^T$, $[2 \;\; 2]^T$ and $[3 \;\; 3]^T$ and `kIndex`=$[0 \;\; 5 \;\; 10]$ then the first mean vector corresponds to step number $k = 0$, the second to step number $k = 5$ and the third to step number $k = 10$.

- `kMethod` - string value that indicates the rounding method to retrieve the correct 2D-array from `mu` or `Sigma` at step k . The following string values are possible:

  - `'low'`: given k, extracts the 2D-array corresponding to the lower bound step number value in `kIndex`. (This is the default setting)

  - `'high'`: given k, extracts the 2D-array corresponding to the higher bound step number value in `kIndex`.
  - `'near'`: given k, extracts the 2D-array corresponding to the nearest bound step number value in `kIndex` .

  For example: Given the two column vectors $[1 \;\; 1]^T$, $[2 \;\; 2]^T$, `kIndex`=$[2 \;\; 4]$ and $k = 2.5$, then `'low'` and `'near'` yield $[1 \;\; 1]^T$ while `'high'` yields $[2 \;\; 2]^T$.

The most common ways to construct a nm_gauss_ltv class are:

- `obj = nm_gauss_ltv(mu,Sigma,kIndex,kMethod)` - creates a LTV noise object `obj` with mean `mu` and covariance matrix `Sigma`. Either `mu` or `Sigma` have to be LTV. If both are LTV their array dimension must match.
- `obj = nm_gauss_ltv(mu,Sigma,kIndex)`, `nm_gauss_ltv(mu,Sigma,kMethod)` or `nm_gauss_ltv(mu,Sigma)` - uses the default values for the omitted arguments.
- `obj = nm_gauss_ltv(mu)` or `nm_gauss_ltv(Sigma)` - creates a LTI configuration for respectively `Sigma` and `mu` as described in section 3.3.1, while using default values for the arguments that are left out. In both cases the provided `mu` and `Sigma` must be 3D-arrays. Also, please remind that `Sigma` can not be provided as an array of column vectors in this case to avoid ambiguity.

Example 3.3.2. Construction of an object of class nm_gauss_ltv

To create a Gaussian distribution with LTV means $[1 \;\; 1]^T$ and $[2 \;\; 2]^T$, LTI covariance matrix $F = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$ and `kMethod='high'` the following commands are used:

```
mu = zeros(2,1,2);mu(:,:,1)=[1;1];mu(:,:,2)=[1;1];
obj = nm_gauss_ltv(mu,[1 2;2 1],[],'high');
```

Note that by leaving an optional argument empty, its default value is assigned.

3.3.3 Function handle Gaussian noise (nm_gauss_handle)

The function handle Gaussian noise model (nm_gauss_handle) is the most general noise model class. To create an object of this class both `mu` and `Sigma` are functions and are provided to the object via function handles.

The main properties of the nm_gauss_handle class are:

- `mu` - function handle to a function that returns the mean of the distribution, e.g. `@funmu`. The function must have the step number k and the sample time `Ts` as input parameters and should return the mean as a column vector.
- `Sigma` - function handle to a function that returns the covariance matrix of the distribution, e.g. `@funsigma`. The function must have the step number k and the sample time `Ts` as input parameters and should either return the entire covariance matrix or the variances as a column vector.
- `Ts` - sample time of the noise model that is provided to the functions. (Default value=1)

The most common ways to construct a nm_gauss_handle class are:

- `obj = nm_gauss_handle(mu,Sigma,Ts)` - creates a function handle noise object `obj` with its mean defined by the function return of `mu` and with the covariance matrix defined by the function return of `Sigma`, while using a sampling time of `Ts`.
- `obj = nm_gauss_handle(mu,Sigma)` - creates a function handle noise object `obj` with its mean defined by the function return of `mu` and with the covariance matrix defined by the function return of `Sigma`, while using a sampling time of `Ts=1`.

Example 3.3.3. Construction of an object of class nm_gauss_handle

To create Gaussian distribution using function handles two functions are configured. The function for the mean returns a fixed zero column vector:

```
function mu = funmu(k,Ts)
    mu=[0;0];
end
```

The second function provides a covariance matrix of which the first variance is slowly increasing with time:

```
function sigma = funsigma(k,Ts)
    sigma=[1+0.01*k*Ts   0
           0             1];
end
```

Now, the noise model object with sample time `Ts=2` is created using the command `obj = nm_gauss_handle(@funmu,@funsigma,2);`

3.3.4 Gaussian noise model methods

All Gaussian noise models have the same methods. Additionally, when applying the full syntax as listed below they are valid for any Gaussian noise model object. The following methods are available:

- `[covmat] = cov(obj,k)` - returns the covariance matrix `covmat` of the noise model `obj` at step `k`.
- `[covmat,isDiag] = cov(obj,k)` - returns the covariance of noise model `obj` at step `k` as originally defined by the user, i.e. as a matrix or a column vector. The parameter `isDiag` contains a code which indicates whether the returned matrix is a diagonal matrix (`isDiag=1`), a column vector (`isDiag=2`) or a regular matrix (`isDiag=0`).
- `[cholmat] = cholcov(obj,k)` - returns the upper triangular square root matrix `covmat` of the covariance matrix of noise model `obj` at step k. The case `[cholmat,isDiag]` is completely similar as explained in method `cov`.
- `varvec = var(obj,k)` - returns the vector of variances `varvec` of noise model `obj` at step `k`.
- `meanvec = mean(obj,k)` - returns the mean vector `meanvec` of noise model `obj` at step `k`.
- `samples = sample(obj,n,k)` - returns `n` samples from the distribution of noise model `obj` at step `k`.
- `densEst = pdf(obj,x,k)` - returns the probability vector `densEst`. Each element of this vector contains the probability of the corresponding column vector in matrix `x`. The probability is calculated with the distribution of noise model `obj` at step `k`.

Example 3.3.4. Making optimal use of the syntax of the toolbox.

Consider a piece of code that wants to retrieve the covariance matrix from a configured noise model `nmObj`. A first attempt to retrieve the covariance matrix of `nmObj` would be to consult the property using the syntax

```
covmat=nmObj.Sigma;
```

However, in most cases this does not return the desired covariance matrix. When `nmObj` is of class nm_gauss_handle a function handle is obtained, in case of a LTV noise model a 3D-array might be obtained and even when the noise model is LTI a column vector might be retrieved.
This obvious mistake is used to stress the usage of methods to retrieve data from an

object. In this case the method `cov` can be used as follows:

```
covmat=cov(nmObj,k);
```

Note that, when the intention is to swiftly check the covariance matrix of a class nm_gauss_lti, a shorter notation can be applied, namely '`covmat=cov(nmObj);`'.

3.4 The state space models

This section describes the discrete-time state space model classes. First, the properties and construction methods are summarized for each specific type of class. The methods are presented in the final section since the majority of the state space models have the same methods and since their syntax allows to address them identically. As indicated before, additional information can be obtained from the help files and demos inside the toolbox.

3.4.1 The discrete-time linear model (ss_DL)

The discrete-time linear state space model class (ss_DL) is the least general model. When defining the system matrices as regular matrices it is a time invariant model. To obtain time variant behaviour, one or multiple system matrices are defined as three-dimensional arrays in analogy with the LTV noise model. Although already presented in Chapter 2, the equations are shown again for the reader's convenience:

$$x_{k+1} = A_k x_k + B_k u_k + w_k,$$
$$y_k = C_k x_k + D_k u_k + v_k.$$

The main properties of the ss_DL class are:

- `A` - the state matrix.
- `B` - the input matrix.
- `C` - the output matrix.
- `D` - the feedthrough matrix. (When not used can be set equal to the scalar 0)
- `x0` - initial state noise model. If the input parameter is a predefined noise model, it can be of any class. If a Gaussian noise model is required, a cell array containing the input arguments of a Gaussian noise class can be used. When the configuration of Gaussian the noise model only requires one parameter, it can be entered without cell structure.
- `w` - process noise model. (See `x0` for configuration)
- `v` - measurement noise model. (See `x0` for configuration)

- `k0` - initial step number. (Default=0)
- `Ts` - sample time. (Default=1)
- `TimeUnit` - string that represents the unit of the time variable. `TimeUnit` can take the values: `'nanoseconds'`, `'microseconds'`, `'milliseconds'`, `'seconds'`, `'minutes'`, `'hours'`, `'days'`, `'weeks'`, `'months'`, `'years'`. (Default= `'seconds'`)
- `kIndex` - vector that contains the step numbers that correspond to each vector or matrix when one or more system matrices are configured as 3D-arrays. For more information see Section 3.3.2
- `kMethod` - string value that indicates the rounding method to retrieve the correct 2D-array from a 3D-array at step k. The following string values are possible: `'low'`, `'high'` and `'near'`. For more information see Section 3.3.2

The most common ways to construct a ss_DL class are:

- `obj = ss_DL(A,B,C,D,x0,w,v,k0,Ts,TimeUnit,kIndex,kMethod)` - creates a discrete-time linear state space object `obj`. When an empty object is inserted for `k0`, `Ts`, `TimeUnit`, `kIndex` or `kMethod` or if one of these arguments is omitted, their default value is used. Note that it is only allowed to leave out an input argument from right to left. For example: `kIndex` can only be left out when `kMethod` is also left out.
- `obj = ss_DL(A,B,C,D,x0,w,v)` - minimal construction to create a discrete-time linear state space object `obj` which uses the default values for all omitted arguments.

Example 3.4.1. Construction of an object of class ss_DL

Let us assume that the system matrices `A`, `B`, `C` and the initial state noise object `x0` are already configured. To create a linear state space model where `w` and `v` are zero mean Gaussian noise models with covariance matrices

$$Q = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}, \quad R = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

and sample time `Ts=2` the following command can be used:

```
obj = ss_DL(A,B,C,0,x0,{[0;0],[1 2;2 1]},[2 0;0 2],[],2);
```

Since `D=0`, the toolbox resizes it to its proper size. This trick only applies to `D` since it is often set to zero. Note that the measurement noise `v` is configured with a shorter syntax than the process noise `w`. Finally, notice that by using an empty matrix for `k0` its default value is used.

3.4.2 The discrete-time nonlinear model with additive noise (ss_DNL_AN)

The discrete-time nonlinear state space model with additive noise (ss_DNL_AN) is the least general nonlinear state space class that currently resides in the toolbox. Nevertheless it is the most widely used representation of a nonlinear system. To achieve nonlinear behaviour, the state equation and the measurement equation are each equipped with a user-defined function that is presented to the model with a function handle. This method of configuring the equations and the possibility of adding Jacobian functions are the main differences with respect to the linear ss_DL class. Although already presented in Chapter 2, the equations of the ss_DNL_AN model are indicated again for the reader's convenience:

$$x_{k+1} = f(x_k, u_k, k) + w_k,$$
$$y_k = h(x_k, u_k, k) + v_k.$$

The main properties of the ss_DNL_AN class are:

- `f` - function handle for the function f that returns an update value of the state equation without its noise, e.g. `@funf`. The syntax of the function file should be `newval = funf(x,k,u,~,Ts)`. Note that $\sim$ indicates the absence of the process noise w.
- `h` - function handle for the function h that returns an update value of the measurement equation without its noise, e.g. `@funh`. The syntax of the function file should be `newval = funh(x,k,u,~,Ts)`. Note that $\sim$ indicates the absence of the measurement noise v.
- `fJacX` - function handle for the function that returns the Jacobian of f with respect to x, e.g. `@funfjacx`. The syntax of the function file should be `newval = funfjacx(x,k,u,~,Ts)`. Note that $\sim$ indicates the absence of the process noise w. If the function handle is not provided or set to an empty matrix, the Jacobian is estimated numerically.
- `hJacX` - function handle for the function that returns the Jacobian of h with respect to x, e.g. `@funhjacx`. The syntax of the function file should be `newval = funhjacx(x,k,u,~,Ts)`. Note that $\sim$ indicates the absence of the measurement noise v. If the function handle is not provided or set to an empty matrix, the Jacobian is estimated numerically.
- `x0` - initial state noise model. See Section 3.4.1 for its configuration.
- `w` - process noise model. See Section 3.4.1 for its configuration.
- `v` - measurement noise model. See Section 3.4.1 for its configuration.
- `k0` - initial step number. (Default=0)
- `Ts` - sample time. (Default=1)

- `TimeUnit` - string that represents the unit of the time variable. See Section 3.4.1 for the possible values. (Default= `'seconds'`)

The most common ways to construct a ss_DNL_AN class are:

- `obj = ss_DNL_AN(f,h,x0,w,v,k0,Ts,TimeUnit,fJacX,hJacX)` - creates a discrete-time nonlinear state space object `obj` with additive noise. When an empty object is inserted for `k0`, `Ts`, `TimeUnit` or if any of these arguments is omitted, their default value is used. When `fJacX` or `hJacX` are not defined the Jacobians are numerically estimated when requested upon. Note that it is only allowed to leave out an input argument from right to left. For example: `fJacX` can only be left out when `hJacX` is also left out.

- `obj = ss_DNL_AN(f,h,x0,w,v)` - minimal construction to create a discrete-time nonlinear state space object `obj` with additive noise. The default values are used for the arguments that are left out.

Example 3.4.2. Construction of an object of class ss_DNL_AN

Assume that the process, measurement and initial state noise objects are already configured as variables `w`, `v` and `x0`. To create a nonlinear state space model with additive noise the function handles for `f` and `h` are the other required arguments. The function f is defined as

```
function x1 = funf(x,k,u,~,Ts)
    x1(1)= 2*x(1) + x(2)*k + u;
    x1(2)= 3*x(1);
end
```

while the function h is defined as

```
function y1 = funh(x,k,u,~,Ts)
    C = [1 0;0 1];
    y1= C * x;
end
```

The construction of the ss_DNL_AN object is now as follows:

```
obj = ss_DNL_AN(@funf,@funh,x0,w,v);
```

All omitted parameters receive their default values. Since no function handles for the Jacobians are provided, the Jacobians are approximated numerically when requested.

3.4.3 The discrete-time nonlinear model (ss_DNL)

The discrete-time nonlinear state space model (ss_DNL) is the most general state space class. The nonlinear behaviour of the state equation and the measurement equation are fully defined by user-defined functions. These are presented to the model by means of function handles. In addition to the properties of the ss_DNL_AN model, the ss_DNL model also contains the Jacobians with respect to the noise variables. Recall that the equations of the ss_DNL model are:

$$x_{k+1} = f(x_k, u_k, w_k, k),$$
$$y_k = h(x_k, u_k, v_k, k).$$

The main properties of the ss_DNL class are:

- `f` - function handle for the function f that returns an update value of the state equation, e.g. `@funf`. The syntax of the function file should be `newval = funf(x,k,u,w,Ts)`.
- `h` - function handle for the function h that returns an update value of the measurement equation, e.g. `@funh`. The syntax of the function file should be `newval = funh(x,k,u,v,Ts)`.
- `fJacX` - function handle for the function that returns the Jacobian of f with respect to x, e.g. `@funfjacx`. The syntax of the function file should be `newval = funfjacx(x,k,u,w,Ts)`. If the function handle is not provided or set to an empty matrix, the Jacobian is estimated numerically.
- `hJacX` - function handle for the function that returns the Jacobian of h with respect to x, e.g. `@funhjacx`. The syntax of the function file should be `newval = funhjacx(x,k,u,v,Ts)`. If the function handle is not provided or set to an empty matrix, the Jacobian is estimated numerically.
- `fJacW` - function handle for the function that returns the Jacobian of f with respect to w, e.g. `@funfjacw`. The syntax of the function file should be `newval = funfjacw(x,k,u,w,Ts)`. If the function handle is not provided or set to an empty matrix, the Jacobian is estimated numerically.
- `hJacV` - function handle for the function that returns the Jacobian of h with respect to v, e.g. `@funhjacv`. The syntax of the function file should be `newval = funhjacx(x,k,u,v,Ts)`. If the function handle is not provided or set to an empty matrix, the Jacobian is estimated numerically.
- `x0` - initial state noise model. See Section 3.4.1 for its configuration.
- `w` - process noise model. See Section 3.4.1 for its configuration.
- `v` - measurement noise model. See Section 3.4.1 for its configuration.
- `k0` - initial step number. (Default=0)

- `Ts` - sample time. (Default=1)
- `TimeUnit` - string that represents the unit of the time variable. See Section 3.4.1 for the possible values. (Default= `'seconds'`)

The most common ways to construct a ss_DNL class are:

- `obj = ss_DNL(f,h,x0,w,v,k0,Ts,TimeUnit,fJacX,fJacW,hJacX,hJacV)` - creates a discrete-time nonlinear state space object `obj`. When an empty object is inserted for `k0`, `Ts`, `TimeUnit` or if one of these arguments is omitted, their default value is used. When `fJacX`, `hJacX`, `fJacW` or `hJacV` are not defined the Jacobians are numerically estimated. Note that it is only allowed to leave out an input argument from right to left. For example: `hJacX` can only be left out when `hJacV` is also left out.
- `obj = ss_DNL(f,h,x0,w,v)` - minimal construction to create a discrete-time nonlinear state space object `obj`. The default values are used for the arguments that are left out.

Example 3.4.3. Construction of an object of class ss_DNL

Assume that the process, measurement and initial state noise objects are already configured as variables `w`, `v` and `x0` and the functions behind the function handles for `f` and `h` are already created. Now, a nonlinear state space model is constructed as follows:

```
obj = ss_DNL(@funf,@funh,x0,w,v);
```

Later, it is decided to aid the model by adding a function to calculate the Jacobian of f with respect to x. The function is defined as follows

```
function fJacX = funfjacx(x,k,u,w,Ts)
    fJacX = [2  k
             3  0];
end
```

To add the handle to the ss_DNL object the following command is entered

```
obj.fJacX = @funfjacx;
```

3.4.4 State space models methods

Not all discrete-time state space models have the same methods. The methods that are common to all state space classes are summarized first:

- `value=eval_ftot(obj,x,k,u,w)` - returns the new state x_{k+1} of the state space model `obj` by using the state equation with parameters: the current state vector `x`, the input vector `u` , the noise vector `w` and the step value `k`. It is sufficient to use `w=0` to indicate the zero noise vector. When setting `w=[]`, a sample of the process noise model that resides in `obj` is used as value for `w` to update the state.

- `value=eval_htot(obj,x,k,u,v)` - returns the measurement y of the state space model `obj` by using the measurement equation with parameters: the current state vector `x`, the input vector `u` , the noise vector `v` and the step value `k`. It is sufficient to use `v=0` to indicate the zero noise vector. When setting `v=[]`, a sample of the measurement noise model that resides in `obj` is used as value for `v` to update the state.

- `value=eval_ftotJacX(obj,x,k,u,w)` - returns the Jacobian of the state equation of the state space model `obj` with respect to x with parameters: the current state vector `x`, the input vector `u` , the noise vector `w` and the step value `k`. It is sufficient to use `w=0` to indicate the zero noise vector. When setting `w=[]`, a sample of the process noise model that resides in `obj` is used as value for `w`.

- `value=eval_htotJacX(obj,x,k,u,v)` - returns the Jacobian of the measurement equation of the state space model `obj` with respect to x with parameters: the current state vector `x`, the input vector `u` , the noise vector `v` and the step value `k`. It is sufficient to use `v=0` to indicate the zero noise vector. When setting `v=[]`, a sample of the measurement noise model that resides in `obj` is used as value for `v`.

- `value=eval_ftotJacW(obj,x,k,u,w)` - returns the Jacobian of the state equation of the state space model `obj` with respect to w with parameters: the current state vector `x`, the input vector `u` , the noise vector `w` and the step value `k`. It is sufficient to use `w=0` to indicate the zero noise vector. When setting `w=[]`, a sample of the process noise model that resides in `obj` is used as value for `w`.

- `value=eval_htotJacV(obj,x,k,u,v)` - returns the Jacobian of the measurement equation of the state space model `obj` with respect to v with parameters: the current state vector `x`, the input vector `u` , the noise vector `v` and the step value `k`. It is sufficient to use `v=0` to indicate the zero noise vector. When setting `v=[]`, a sample of the measurement noise model that resides in `obj` is used as value for `v`.

- `simObj=sim(obj,samples,u,conf,x0,noise)` - the simulation method simulates the states x and measurements y of the discrete-time state space model `obj` by iterating through the state and measurement equations while applying the inputs u. Its output argument is the simulation object `simObj` of class sim_D. For more information about class sim_D see Section 3.5. The different input arguments of the simulation method are:

  - `samples`: indicates how many measurement or state samples should be acquired. (Default=1)
  - `u`: a matrix of inputs where each column vector represents an input (Default = `[]`).
  - `conf`: string that indicates which variables must be saved in the simulation object `simObj`. Possible string values contain `'x'`, `'u'` and `'y'`. Note that the variable `y` is always saved regardless of the configuration. (Default setting = `'x u y'`) For example: when `conf='x y'`, the simulation object contains the true states `x` and the measurements `y`.
  - `x0`: initial state column vector. (Default=`[]`) When empty, a sample is taken from the initial state noise model `x0` that resides in `obj`.
  - `noise`: option that allows to specify the noise behaviour during simulation. Possible values are `noise=0` for no noise, `noise=1` to only simulate the process noise, `noise=2` to only add measurement noise and `noise=3` to simulate both process and measurement noise. (Default=3)

- `daObj=da(obj,alg,...)` - performs data assimilation on the discrete-time state space model `obj` using the algorithm `alg`. More information about the data assimilation method is provided in Section 3.6, which is completely devoted to this topic.

The following methods are only applicable to the ss_DL class:

- `value=eval_A(obj,k)` - returns the value of system matrix A from the state space model `obj` at step `k`.
- `value=eval_B(obj,k)` - returns the value of system matrix B from the state space model `obj` at step `k`.
- `value=eval_C(obj,k)` - returns the value of system matrix C from the state space model `obj` at step `k`.
- `value=eval_D(obj,k)` - returns the value of system matrix D from the state space model `obj` at step `k`.

The following method is only applicable to the nonlinear classes ss_DNL and ss_DNL_AN:

- `value=checkConsist(obj,uSize, ySize)` - checks the consistency of the configuration of state space model `obj`, i.e. the provided input size `uSize` and measurement size `ySize` parameters are used to check if the user-defined functions are properly responding.

Note that this method is not required for the ss_DL class since this class can validate the consistency of the object whenever a system matrix is adjusted. The nonlinear classes, on the other hand, do not know when the functions have been changed by

the user.

Example 3.4.4. The simulation method

Consider a configured state space object `ssObj`. The most common ways to use the simulation method are:

`simObj=sim(ssObj,50)` - returns a simulation object with 50 simulated measurements and true states.

`simObj=sim(ssObj,50,u)` - returns a simulation object with the provided 50 inputs and 50 simulated measurements and true states.

`simObj=sim(ssObj,50,[],'',[1;2])` - returns a simulation object with 50 simulated measurements and true states. The initial state vector used to start the simulation is $[1 \quad 2]^T$.

3.5 The simulation model (sim_D)

The discrete-time simulator model (sim_D) is used as a container for the data output of the simulation method. Saving the simulation data in this advanced structure allows to internally check the consistency of the data and to define methods that show the data in a more user friendly way. As explained in Section 3.4.4, the data that needs to be saved in the simulation model is specified in the simulation method property `conf`. An important remark is that once the simulation model is constructed, it is impossible to change its content. This is to assure that the data structure remains consistent. Of course, the data can be viewed or extracted whenever desired.

The main properties of the sim_D class are:

- `y` - the simulated measurements which are stored in a matrix where each column is a measurement. Since this property is the main reason to perform a simulation, it must be non-empty.
- `x` - the simulated states which can be regarded as the true states. The property `x` is a matrix of simulated states where each column is a state. This property can be left empty (`[]`) if desired.
- `u` - the used inputs. These are stored in a matrix where each column represents an input vector. This property can be left empty (`[]`) if desired.

- `k` - the array of step numbers. Each element contains the step number corresponding to each column of `x`, `u` and `y`. For example: the second element of `k` corresponds to the step number of column two. (Can not be empty.)
- `Ts` - the sample time. (Default=1)
- `TimeUnit` - string that represents the unit of the time variable. `TimeUnit` can take the values: `'nanoseconds'`, `'microseconds'`, `'milliseconds'`, `'seconds'`, `'minutes'`, `'hours'`, `'days'`, `'weeks'`, `'months'`, `'years'`. (Default= `'seconds'`)

The most common ways to construct a sim_D class are:

- `obj = sim_D(x,u,y,k,Ts,TimeUnit)` - creates a discrete-time simulation object `obj`. When an empty object is inserted for `Ts` or `TimeUnit` or if one of these arguments is omitted, their default value is used. Note that it is only allowed to leave out an input argument from right to left. For example: `Ts` can only be left out when `TimeUnit` is also left out.
- `obj = sim_D(x,u,y,k)` - minimal construction to create a discrete-time simulation object `obj`. The default values are used for the arguments that are left out.

The main method of the sim_D class is:

- `plot(obj,conf,nrs,varargin)` or `plot(obj,conf,nrs1,nrs2,varargin)` - overloaded plot method that allows fast creation of plots using predefined layouts. The input arguments of the plot method are:
  - `obj`: a simulation object of class sim_D.
  - `conf`: string that defines the variable(s) to plot, the desired scale of the x-axis and the type of plot. Possible values are `'u * **'`, `'x * **'`, `'y * **'` and `'xy *'`, where
    - `'*'` can be either `'k'` to plot in function of the step index or `'t'` to plot in function of time and
    - `'**'` can be either `'1'` to create a single plot which contains all variables or `'2'` to create a subplot per variable.

    Note that `'1'` can not be chosen in combination with `'xy'` since this option only supports subplots. The default value for `'*'` is `'k'` and the default value for `'**'` is `'2'`.
  - `nrs`: vector that contains the variable numbers that need to be plotted. A maximum of 15 variables can be plotted. For example: `nrs=[1:3,6]` plots variable numbers 1 to 3 and 6.
  - `nrs1` and `nrs2`: when `conf` contains `'xy'`, two vectors need to be supplied where the first one indicates the variable numbers for the states and the second one indicates the variable numbers for the measurements. The size of both vectors need to be equal.

– `varargin`: additional arguments as in regular plot function. When these are specified, the predefined settings such as colors, titles and the legend are not included any more. For example:
`plot(obj,'y',[5:9],'linewidth',3)` plots using in line width 3.

Of course, additional methods can easily be defined if desired.

Example 3.5.1. Making plots of a simulation object.

Consider a configured simulation object `simObj` as obtained through a simulation. Some common ways to use the plot method are:

`plot(simObj,'x',(1:4))` - plots the first four states of `simObj` in four different subplots using the step index values for the x-axis scale.

`plot(simObj,'x t',(1:4))` - plots the first four states of `simObj` in four different subplots using the time values to scale the x-axis.

`plot(simObj,'x 1',(1:4))` - plots the first four states of `simObj` in four different subplots using the step index values for the x-axis scale.

`plot(simObj,'xy',(3:6),(1:4))` - plots the states three to six of `simObj` together with corresponding measurements one to four in four different subplots using the step index values for the x-axis scale.

For examples of illustrated plots, please check Chapter 4 which contains several examples.

3.6 The data assimilation techniques

This section describes the syntax of the different data assimilation methods. Note that these methods are actually part of the state space models methods as described in Section 3.4.4. They deserve to be summarized in a separate section because they are the objective of this toolbox. There are two main syntax possibilities to apply a data assimilation method.

The first syntax is: `daObj=da(ssObj,alg,...)`
This syntax performs the data assimilation algorithm as specified in the string value `alg` on state space object `ssObj` and saves the outputs of the algorithm in the data assimilation object `daObj`. This last object serves, very much like a simulation object, as a container to store the requested data (see Section 3.7). The string value in `alg` that specifies the algorithm can be 'KF' for Kalman Filter, 'EKF' for extended

Kalman filter, 'UKF' for unscented Kalman filter, 'EnKF' for ensemble Kalman filter, 'DEnKF' for deterministic ensemble Kalman filter, 'EnSRF' for ensemble square root filter, 'ETKF' for ensemble transform Kalman filter, 'OI' for optimal interpolation, 'PF_GEN' for the generic particle filter, 'PF_SIR' for the SIR particle filter and 'PF_ASIR' for the auxiliary SIR particle filter.

The second syntax is: `daObj='alg'(ssObj,...)`
Now, 'alg' is not a variable but is a placeholder for the algorithm strings like KF, EKF, etc. Hence, to perform a data assimilation with the Kalman filter the syntax becomes `daObj=KF(ssObj,...)`.

The choice of syntax is really a matter of taste, there are no advantages or disadvantages in using a particular syntax. From now on, the different algorithms are explained using the first syntax, the extension to the second syntax is considered trivial. The syntax of all assimilation methods consists of two parts: a first part that is valid for all algorithms and a a second part that is algorithm specific. The first part is explained now; the second part is explained for each algorithm separately. Any algorithm method can be configured in the following ways:

1. `daObj=da(ssObj,alg,simmodel,outconf,optionscell)` - performs the data assimilation with a simulation object. Its different arguments are:

   - `ssObj`: a state space object.
   - `alg`: a string that contains the requested algorithm.
   - `simmodel`: a simulation model object. The measurements and possible inputs that reside in this object are used to assimilate the data. If specified in `outconf`, the true states are extracted to save in the data assimilation object. The amount of measurements is used to determine the amount of required iterations in the assimilation sequence.
   - `outconf`: a string that specifies which variables should be saved in the data assimilation object. Possible string values contain `'xf'`, `'Pf'`, `'Pa'`, `'x'`, `'y'`, `'u'` and `'cov'`. The string `'cov'` indicates that the full covariance matrix should be saved instead of the variance vectors. The default setting is to only save the variances. Note that `'xa'` is always saved as it is the main goal of any assimilation. For example: `'Pa y'` saves the analysis states `xa`, the diagonal elements of analysis error covariance matrix `Pa` and the measurements `y`.
   - `optionscell`: cell array of algorithm specific settings.

   The parameters `outconf` and `optionscell` can be left out if desired, but the order of the parameters must remain.

2. `daObj=da(ssObj,alg,y,u,outconf,optionscell)` - performs the data assimilation with real measurements. In the rare case that true states are available, the previous syntax should be used. The different arguments are:

- `ssObj`: a state space object.
- `alg`: a string that contains the requested algorithm.
- `y`: a matrix of measurements where each column vector is a measurement. The amount of measurements is used to determine the amount of required iterations in the assimilation sequence. Note that `y` should start from `y0`, i.e. the measurement corresponding to `x0`, the initial state at step `k0` as specified in `ssObj`.
- `u`: a matrix of inputs where each column vector is an input.
- `outconf` a string that specifies which variables should be saved in the data assimilation object. Check the previous simulation object syntax for more information.
- `optionscell`: cell array of algorithm specific settings.

The parameters `u`, `outconf` and `optionscell` can be omitted if desired, but the order of the parameters must remain. For example:
using the syntax `daObj = da(ssObj,alg,y,outconf)` is allowed, but
`daObj = da(ssObj,alg,outconf,y)` is not allowed.

Now that the generic part of the `da` method is clear, the different algorithms and their specific settings in `optionscell` can be summarized. Note that, unless specified otherwise, `optionscell` can be left out entirely which causes the algorithms to use the default values. Also, in case of multiple arguments, one or more arguments can be left out but only from right to left. The algorithm specific settings of `optionscell` are:

- The Kalman filter (`alg='KF'`) - the KF has no specific settings.
- The extended Kalman filter (`alg='EKF'`) - the EKF has no specific settings.
- The optimal interpolation technique (`alg='OI'`) - has the specific setting `optionscell = {lti}`. The parameter `lti=1` indicates that the measurement equation of `ssObj` is linear and time invariant. If so, the algorithm calculates the Kalman gain only once. The default setting is `lti=0`. For more information see Section 2.5.
- The unscented Kalman filter (`alg='UKF'`) - has the specific setting `optionscell = {kappa,sptwice}`. The parameter `kappa` is a scalar that can be used to reduce the higher order errors of the mean and covariance approximations. Its default value is zero. The parameter `sptwice` indicates whether the sigma points of the forecast step need to be reused in the analysis step (`sptwice=0`) or need to be recalculated in the analysis step (`sptwice=1`). Its default value is zero. For more information see Section 2.6.
- The ensemble Kalman filter (`alg='EnKF'`) - has the specific setting `optionscell = {nEns}` or `optionscell={initEns}`. The parameter `nEns` indicates the number of ensembles to use when randomly sampling them from the initial state

noise model of `ssObj`. The parameter `initEns` contains a user-defined initial ensemble. When neither of these parameters are provided, the algorithm sets `nEns` equal to the amount of states of `ssObj` with a minimum of 5 and a maximum of 50.

- The deterministic ensemble Kalman filter (`alg='DEnKF'`) - has the specific setting `optionscell = {nEns}` or `optionscell={initEns}`. See the EnKF for more information.
- The ensemble transform Kalman filter (`alg='ETKF'`) - has the specific setting `optionscell = {nEns}` or `optionscell={initEns}`. See the EnKF for more information.
- The ensemble square root filter (`alg='EnSRF'`) - has the specific setting `optionscell = {nEns}` or `optionscell={initEns}`. See the EnKF for more information.
- The generic particle filter (`alg='PF_GEN'`) - has the specific setting `optionscell = {nPar,Nt,resampler}` or `optionscell = {initPar,Nt,resampler}`. The different parameters are:
  - `nPar`: indicates the number of particles to use. With this setting the particles are randomly sampled from the initial state noise model of `ssObj`. When not specified `nPar` is set equal to 100.
  - `initPar`: contains user-defined initial particles. When not specified `nPar` is set equal to 100.
  - `Nt`: the effective sample size threshold for resampling which is expressed in a scale from 0 to 1× `nPar`. Resampling occurs when $N_{eff} <$ (`Nt` × `nPar`). The default value is 1.
  - `resampler`: string that specifies the resample algorithm. Possible values are `'multin'` (multinomial), `'residual'`, `'systematic'` or `'stratisfied'`. In case if `'residual'` a second resample algorithm must be specified in `optionscell`. The default value (also for the second resample algorithm) is `'stratisfied'`.

  For more information see Section 2.11.
- The sampling importance resampling particle filter (`alg='PF_SIR'`) - has the specific setting `optionscell = {nPar,resampler}` or `optionscell = {initPar,resampler}`. The explanation of the different parameters can be found in the previous algorithm PF_GEN.
- The auxiliary sampling importance resampling particle filter (`alg='PF_ASIR'`) - has the specific setting `optionscell = {nPar,resampler}` or `optionscell = {initPar,resampler}`. The explanation of the different parameters can be found in the algorithm PF_GEN.

Example 3.6.1. The data assimilation method

Consider a configured state space object `ssObj`. Some common ways to use the data assimilation method are:

`damObj=da(ssObj,'KF',simObj,'Pa')` - retrieves measurements from simulation object `simObj` to perform the Kalman filter algorithm on `ssObj` in order to create a data assimilation object that contains the analysis states and the variances of the analysis covariance matrices.

`damObj=da(ssObj,'EnKF',y,{50})` - uses the measurements `y` to perform the ensemble Kalman filter algorithm on `ssObj` with 50 ensemble members in order to create a data assimilation object that contains the analysis states.

`damObj=da(ssObj,'PF_GEN',y,{60,0.6,'residual','systematic'})` - uses the measurements `y` to perform the generic particle filter algorithm on `ssObj` with 60 particles. The resample threshold is set to $0.6 \times 60 = 36$ and the resample method `residual` uses `'systematic'` to resample the particles. Finally, a data assimilation object that contains the analysis states is created.

3.7 The data assimilation model (dam_D)

The discrete-time data assimilation model class (dam_D) is very similar to the simulation model (sim_D) discussed earlier. As indicated in the previous section the data that needs to be saved in the assimilation model is specified in the data assimilation method property `conf`. As in case of the (sim_D) class, it is impossible to alter the contents of a data assimilation object once it is constructed to ensure data consistency.

The main properties of the dam_D class are:

- `alg` - string that indicates the algorithm used to generate the data object. Possible values are `'KF'`, `'EKF'`, etc.
- `x` - the simulated states that can be regarded as the true states. This is a matrix of simulated states where each column is a state. This property can be left empty (`[]`) if desired.
- `xf` - the forecast states. This is a matrix of forecast states where each column is a forecast state. This property can be left empty (`[]`) if desired.
- `xa` - the analysis states. This is a matrix of analysis states where each column is a analysis state. This property can be left empty (`[]`) if desired.

- `Pf` - the (co)variance forecast array. This is either a 3D array of covariance matrices or a matrix variance vectors. This property can be left empty (`[]`) if desired.
- `Pa` - the (co)variance analysis array. This is either a 3D array of covariance matrices or a matrix variance vectors. This property can be left empty (`[]`) if desired.
- `u` - the used inputs. These are stored in a matrix where each column represents an input vector. This property can be left empty (`[]`) if desired.
- `y` - the measurements. These are stored in a matrix where each column is a measurement. This property can be left empty (`[]`) if desired.
- `k` - the array of step numbers. Each element contains the step number corresponding to each column of `x`, `u`, `y`, etc. (Can not be empty.)
- `Ts` - the sample time. (Default=1)
- `TimeUnit` - string that represents the unit of the time variable. `TimeUnit` can take the values: `'nanoseconds'`, `'microseconds'`, `'milliseconds'`, `'seconds'`, `'minutes'`, `'hours'`, `'days'`, `'weeks'`, `'months'`, `'years'`.
  (Default= `'seconds'`)
- `covFull` - wariable that indicates whether the full covariance matrices are saved (`covFull=1`) or the only the variances (`covFull=0`). Is a required input argument if Pf or Pa are non-empty.

The most common ways to construct a dam_D class are:

- `obj = dam_D(alg,x,xf,xa,Pf,Pa,u,y,k,covFull,Ts,TimeUnit)` - creates a discrete-time data assimilation object `obj`. When an empty object is inserted for `Ts`, `TimeUnit` or `covFull`, or if one of these arguments is omitted, their default value is used. Note that it is only allowed to leave out an input argument from right to left.
- `obj = dam_D(alg,x,xf,xa,Pf,Pa,u,y,k,covFull)` - minimal construction to create a discrete-time data assimilation object `obj` if `Pa` or `Pf` are provided. The default values are used for the arguments that are left out.
- `obj = dam_D(alg,x,xf,xa,[],[],u,y,k)` - minimal construction to create a discrete-time data assimilation object `obj`. The default values are used for the arguments that are left out.

The main methods of the dam_D class are:

- `plot(obj,conf,nrs,varargin)` or `plot(obj,conf,nrs1,nrs2,varargin)` - overloaded plot method that allows fast creation of plots using predefined layouts. The input arguments of the plot method are similar to the ones

described in Section 3.5. The only difference is that the parameter `conf` has the extra possible string values `'xf'`, `'xa'`, `'Pf'`, `'Pa'`, `'xaPa'`, `'xaxPa'`, `'xfPf'`, `'xax'`, `'xaxy'` and `'xa-x'`. In case of `'Pf'` and `'Pa'` only the variances are plotted. The string values `'xaPa'`, `'xaxPa'` and `'xfPf'` plot the states with their 95% confidence interval. String value `'xax'` compares the true states with the assimilated states and `'xaxy'` adds the measurements. The string value `'xa-x'` plots the difference between the true state and the assimilated state. The values `'xax'`, `'xaxy'`, `'xaPa'`, `'xaxPa'` and `'xfPf'` are only possible in combination with the subplot plot style.

- `val=rmse(obj,nrs)` - returns a column vector `val` that holds the RMSE values of the true states `x` and assimilated analysis states `xa` that reside in the dam_D object `obj`. The array `nrs` contains the state numbers that have to be taken into account and needs to be specified.

- `val=mse(obj,nrs)` - is completely similar to `val=rmse(obj,nrs)` but calculates the MSE values.

Of course, additional methods can be defined if desired.

Example 3.7.1. Examples of using the dam_D class methods.

Consider a configured data assimilation object `damObj` as obtained through a simulation. Some common ways to use the plot and RMSE methods are:

`plot(damObj,'xax',(1:4))` - plots the first four true states and analysis states of `damObj` in four different subplots using the step index values for the x-axis scale.

`plot(damObj,'xaPa t',(1:4))` - plots the first four states of `damObj` with their 95% confidence intervals in four different subplots using the time values to scale the x-axis.

`value=rmse(damObj,(2:5))` - returns the column vector `value` that contains the RMSE of first three true states and analysis states that reside in `damObj` object.

For examples of illustrated plots, please see Chapter 4 which contains several examples.

3.8 Conclusion

This chapter has provided a complete overview of the data assimilation toolbox. The first section advised the user to actively use the toolbox while progressing through the chapter. The second section clarified the main architecture of the toolbox which obtains its framework from the object-oriented possibilities of Matlab. The classes

have been selected with great care which results in a transparent structure with clear boundaries between the classes. The interaction between the classes are provided by intuitively straightforward methods. This results in a very user friendly environment where a data assimilation technique can be performed with a minimum amount of user code. While remaining user friendly, the toolbox still has an extensive range of possibilities to define noise models, state space models, simulations and data assimilation techniques.

A typical data assimilation starts with defining the Gaussian noise models. There are three different possibilities to define these. A first possibility is to define a linear time invariant model, a second one is to use a linear time variant model and the last one is to define the model with user-defined functions. As such, all possible variants of Gaussian noise sources are covered by the toolbox.

The second step is to define a state space model. There are currently three such models defined in the toolbox which cover the vast majority of possible state space representations. The noise sources inside the state space models are defined using the noise models of the first step. The system equations of a linear state space model are defined by matrices: regular matrices for time invariant behaviour and three-dimensional arrays for time variant behaviour. The nonlinear variants receive their system equations by means of user-defined functions.

To final required parameter to start with the data assimilation are the measurements. Either the user possesses real measurements or the user uses the simulation tools to obtain virtual measurements. Now, only a single instruction is needed to perform any data assimilation technique on any configured state space model. In its most straightforward setup only two parameters are required: the desired algorithm and the measurements.

The data assimilation results are automatically saved in a data assimilation object. This object is fitted with some often required methods to check the performance of an algorithm. With only a single line of code, several state estimations can be plotted together with their uncertainties, they can also be compared with other variables such as the measurements or the true states. If the true states are available, the RMSE or MSE of any desired state estimation(s) can also be obtained with a single line of code.

All these possibilities and many more have been explained in great detail throughout the last sections which summarized the main properties, constructors and methods of all classes. In addition, the most important features have been illustrated with small examples of MATLAB code.

Chapter 4

Implementation examples

This chapter presents the usage of the toolbox by illustrating the sequence of required command lines to apply a data assimilation technique. A total of four nonlinear systems, that are often encountered in the literature, are taken under consideration. The first system is the well known Van der Pol oscillator. The second system is defined by the Lorenz equations which are famous for their chaotic solutions that resemble a butterfly. The third system, which is referred to as the tracking problem, consists of an airplane that is tracked by either a radar or by two observers. These kind of tracking problems are known for their highly nonlinear measurement equations. The last system consists of a highly nonlinear scalar state equation and a nonlinear measurement equation. Although scalar, this system has become a benchmark for nonlinear estimators due to its complex nonlinearity.

4.1 The Van der Pol oscillator

The Van der Pol oscillator is a widely used example in the literature. It is highly nonlinear and, depending on the direction of time, it can exhibit both stable and unstable limit cycles. A first-order Euler discretization of the equations as presented in [14] yields:

$$x_{k+1} = f(x_k) \tag{4.1}$$

with

$$\begin{aligned} x_k &= \begin{bmatrix} x_{1,k} & x_{2,k} \end{bmatrix}^T, \\ f(x_k) &= \begin{bmatrix} x_{1,k} + T_s x_{2,k} \\ x_{2,k} + T_s(\alpha(1 - x_{1,k}^2)x_{2,k} - x_{1,k}) \end{bmatrix}. \end{aligned} \tag{4.2}$$

The sampling time T_s is set equal to 0.1 seconds and the scalar parameter α, which indicates the nonlinearity and the strength of the damping, is set equal to 0.2 in this experiment. Furthermore, the Van der Pol oscillator is driven by w_k, i.e.

$$x_{k+1} = f(x_k) + w_k \tag{4.3}$$

where $w_k \in \mathbb{R}^2$ is zero-mean Gaussian noise with covariance matrix

$$Q = \begin{bmatrix} 10^{-2} & 0 \\ 0 & 10^{-2} \end{bmatrix}. \tag{4.4}$$

The linear measurement equation is

$$y_k = x_{1,k} + v_k \tag{4.5}$$

where the measurement noise $v_k \in \mathbb{R}$ is zero-mean Gaussian noise with $R = 10^{-2}$. The initial state estimation is $x_0 = [0 \;\; 5]^T$ and the initial error covariance matrix is $P_0 = \begin{bmatrix} 5 & 0 \\ 0 & 5 \end{bmatrix}$.

The typical procedure of performing a data assimilation technique starts by defining the noise models. This is done as follows:

```
>> mu_x0=[0 5]';sigma_x0=diag([5, 5]);            %initial state
>> x0={mu_x0,sigma_x0};
>> mu_w=[0 0]';sigma_w=diag([10e-2 10e-2]);       %process noise
>> w={mu_w,sigma_w};
>> mu_v=0;sigma_v=10e-2;                          %meas. noise
>> v={mu_v,sigma_v};
```

Before creating the state space model, the sample time, the initial step number and the unit of time are defined

```
>> k0 = 0;               %intial step number
>> Ts=0.1;               %sample time
>> TimeUnit='seconds';   %unit of time
```

and the system equations are defined in user-defined functions. For example, the function for the system equation is configured as follows:

```
function x1 = VDP_f(x,k,u,~,Ts)
   alpha=0.2;

   x1(1,1) = x(1) + Ts * x(2);
   x1(2,1) = x(2) + Ts * ( (alpha * (1 - x(1)^2) * x(2) ) - x(1));
end
```

Since the process noise of the Van der Pol equation is additive, the ss_DNL_AN class is selected to create the state space model which is done as follows:

```
>> ssObj=ss_DNL_AN(@VDP_f,@VDP_h,x0,w,v,k0,Ts,TimeUnit,...
                                          @VDP_fjacx,@VDP_hjacx);
```

This command provides the state space model `ssObj` that is simulated over 500 iterations, starting from the initial state `x0init`. In addition, the configuration is set to save the true states and measurements in a simulation object. Hence, the required command lines are:

```
>> x0init=[1.4 0]';samples=500; conf='x y';u=[];
>> simObj=sim(ssObj,samples,u,conf,x0init);
```

To validate the simulation results, the plot command can be applied on the obtained simulation object `ssObj` as follows:

```
>> plot(simObj,'x',(1:2));
```

This provides a pre-configured plot of both true states as illustrated in Figure 4.1. The EKF data assimilation method is now applied while saving the true states and

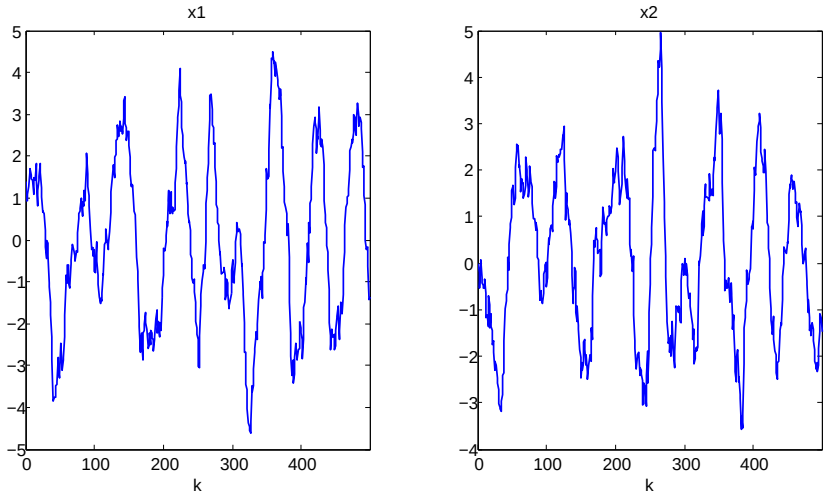


FIGURE 4.1: Illustration of a pre-configured simulation plot: true states of the Van der Pol equations.

measurements in the data assimilation object `damEKF`:

```
>> damEKF=da(ssObj,'EKF',simObj,'x y');
```

Note that the analysis states `xa` are not entered in the configuration since they are always saved. In correspondence to the simulation object, the results can be checked by applying the plot command as follows

```
>> plot(damEKF,'xax',(1:2));
```

which provides a pre-configured plot of the true states and analysis states as illustrated in Figure 4.2. Note how the analysis state of $x_{1,k}$ provides a better estimation than the analysis state of $x_{2,k}$ due to the measurements.

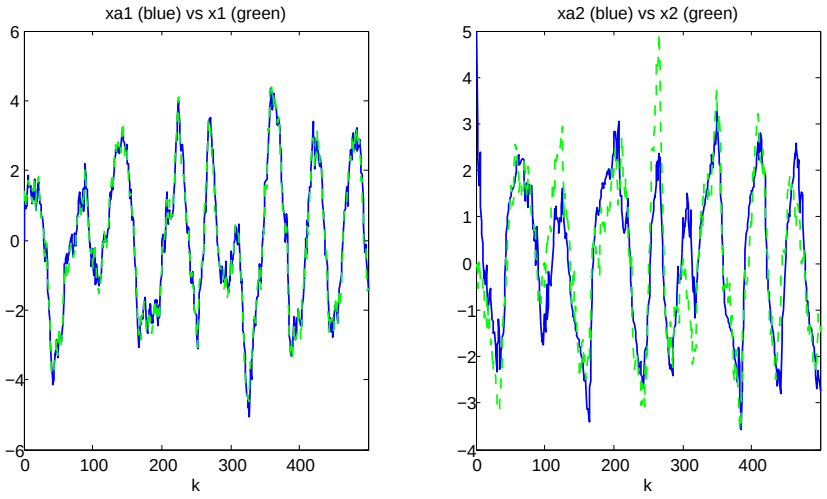


Figure 4.2: Illustration of a pre-configured data assimilation plot: true states and analysis states of the Van der Pol equations.

4.2 The Lorenz equations

The Lorenz equations are a simplified mathematical model for atmospheric convection defined by three ordinary differential equations:

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x), \\ \frac{dy}{dt} &= x(\rho - z) - y, \\ \frac{dz}{dt} &= xy - \beta z, \end{aligned} \tag{4.6}$$

where σ, ρ and β are the system parameters which are set to $\sigma = 10$, $\rho = 28$ and $\beta = 8/3$. With these settings the system exhibits the typical chaotic behaviour as observed by Lorenz.

To implement this continuous-time model into the discrete-time toolbox, the model is integrated between two consecutive steps using the Runge-Kutta *ode*45 solver of Matlab as follows:

```
function x1 = L3_f(x,k,u,~,Ts)

    %L3f: contains Lorenz equations
    [TOUT,x1] = ode45(@L3f,[k*Ts  (k+1)*Ts],x);
    x1=x1(end,:)';

end
```

As such, the system equations can be defined as:

$$\begin{aligned} x_{k+1} &= f(x_k) + w_k \\ y_k &= x_{1,k} + v_k \end{aligned} \tag{4.7}$$

with $x_k = [x_{1,k} \;\; x_{2,k} \;\; x_{3,k}]^T$, $f(x_k)$ the integrated Lorenz model and where $w_k \in \mathbb{R}^3$ and $v_k \in \mathbb{R}$ are zero-mean Gaussian noise with respective covariance matrices

$$Q = \begin{bmatrix} 10^{-3} & 0 & 0 \\ 0 & 10^{-3} & 0 \\ 0 & 0 & 10^{-3} \end{bmatrix} \quad \text{and} \quad R = \begin{bmatrix} 2 \end{bmatrix}. \tag{4.8}$$

Furthermore, the initial state estimation is defined by the zero vector $x_0 = [-5.8 \;\; -5.7 \;\; 20.5]^T$ and the unity error covariance matrix $P_0 = I_3$.

The data assimilation procedure starts by defining the noise models

```
>> x0=[-5.8;-5.7;20.5];            %initial state
>> w={diag([1e-3  1e-3  1e-3])};   %process noise
>> v={0,2};                        %meas. noise
```

after which a ss_DNL_AN object is created with a sample time of 0.01 seconds as follows:

```
>> ssObj=ss_DNL_AN(@L3_f,@L3_h,x0,w,v,[],0.01);
```

The obtained state space model `ssObj` is now simulated over 2000 iterations, starting from the initial state $[-6 \;\; -6 \;\; 20]^T$ while saving the true states and measurements as follows:

```
>> simObj=sim(ssObj,2000,[],'x y',[-6;-6;20]);
```

The EnKF algorithm is applied with 40 ensemble members using the simulating measurements from `simObj`, while saving the true states and measurements in the data assimilation object `damEnKF`:

```
>> damEnKF=da(ssObj,'EnKF',simObj,'x y',{40});
```

The overloaded plot command is applied on the data assimilation object using the syntax

```
>> plot(damDEnKF,'xa-x 1 t',(1:3));
```

which delivers a single plot of the difference between the true states and analysis states in function of time as illustrated in Figure 4.3. Notice that the errors of the unmeasured states $x_{2,k}$ and $x_{3,k}$ are only slightly larger than those of the measured state $x_{1,k}$. Figure 4.4 illustrates the obtained analysis states and true states in a three-dimensional plot. Hence, the well known butterfly appears.

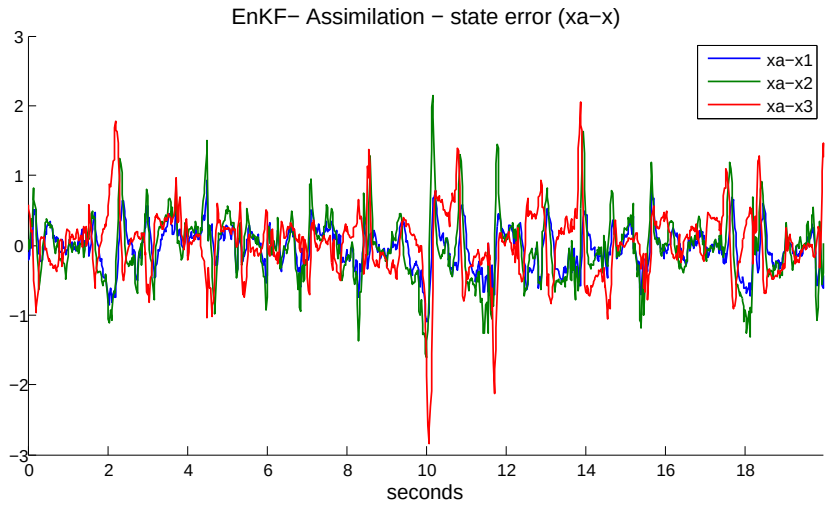


Figure 4.3: Illustration of a pre-configured data assimilation plot: difference between true states and analysis states of the Lorenz equations.

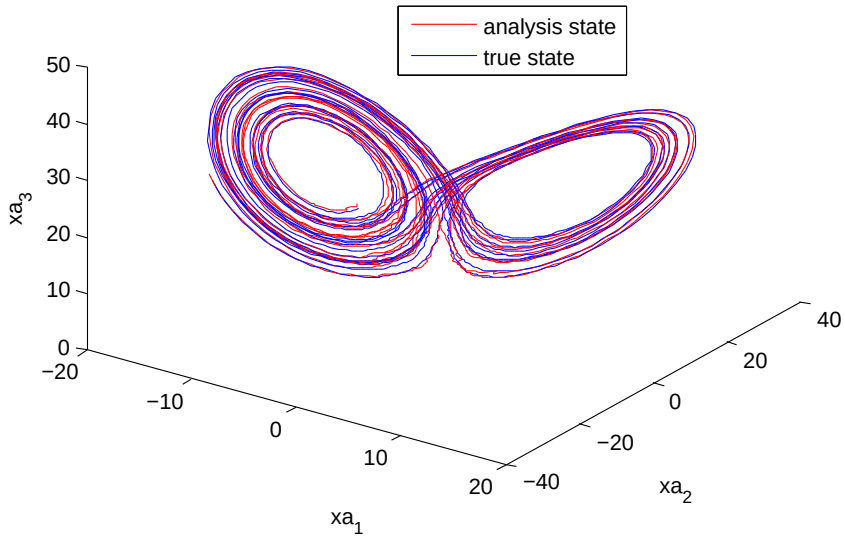


Figure 4.4: Three-dimensional illustration of true states and analysis states of the Lorenz equations.

4.3 The tracking problem

The tracking problem as described in [11] presents the problem of estimating the true position of an airplane. The state equation of the airplane movement is:

$$x_{k+1} = f(x_k) + w_k \tag{4.9}$$

with

$$x_k = [x_{1,k} \quad x_{2,k} \quad x_{3,k} \quad x_{4,k}]^T, \tag{4.10}$$
$$f(x_k) = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} x_k,$$

and where $w_k \in \mathbb{R}^4$ is zero-mean Gaussian noise with covariance matrix

$$Q = \begin{bmatrix} 10^{-7} & 0 & 0 & 0 \\ 0 & 10^{-7} & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0.5 \end{bmatrix}. \tag{4.11}$$

The different states $x_{1,k}$, $x_{2,k}$, $x_{3,k}$ and $x_{4,k}$ represent respectively the horizontal and vertical position, and the horizontal and vertical velocity. The initial state estimation consists of $x_0 = [-200 \quad 200 \quad 4 \quad 0]^T$ and $P_0 = I_4$.

There are two variants of the measurement equation which are illustrated in Figure 4.5. The first equation models the distance $d = y_{r_{1,k}}$ and angle $\theta = y_{r_{2,k}}$ as obtained from a radar, while the second one models the two distances $d_1 = y_{t_{1,k}}$ and $d_2 = y_{t_{2,k}}$ as obtained from two different observers through triangulation. Hence, radar measurement equation is

$$y_{r_k} = \begin{bmatrix} y_{r_{1,k}} \\ y_{r_{2,k}} \end{bmatrix} = \begin{bmatrix} \sqrt{x_{1,k}^2 + x_{2,k}^2} \\ \arctan(x_{2,k}/x_{1,k}) \end{bmatrix} + v_{r_k}, \tag{4.12}$$

while the triangulation measurement equation is

$$y_{t_k} = \begin{bmatrix} y_{t_{1,k}} \\ y_{t_{2,k}} \end{bmatrix} = \begin{bmatrix} \sqrt{(x_{1,k} - p_{1x})^2 + (x_{2,k} - p_{1y})^2} \\ \sqrt{(x_{1,k} - p_{2x})^2 + (x_{2,k} - p_{2y})^2} \end{bmatrix} + v_{t_k}, \tag{4.13}$$

where $p_{1x} = -300$, $p_{1y} = 0$, $p_{2x} = 300$ and $p_{2y} = 0$ are the coordinates of the two observers. The radar is assumed to be located at position $(0, 0)$. The measurement noise sources $v_{r_k}, v_{t_k} \in \mathbb{R}^2$ are zero-mean and Gaussian with respective covariance matrices

$$R_r = \begin{bmatrix} 200 & 0 \\ 0 & 0.003 \end{bmatrix} \quad \text{and} \quad R_t = \begin{bmatrix} 200 & 0 \\ 0 & 200 \end{bmatrix}. \tag{4.14}$$

In this example, the performances of the UKF, the ETKF and the PF_ASIR are compared for both the radar and triangulation measurements by means of the RMSE values of the true and estimated states. After creating the state and measurement functions, the noise models are defined:

```
>> x0={[-200;200;4;0]};      w={diag([10e-7;10e-7;0.5;0.5])};
>> vR={diag([200;0.003])};  vT={diag([200;200])};
```

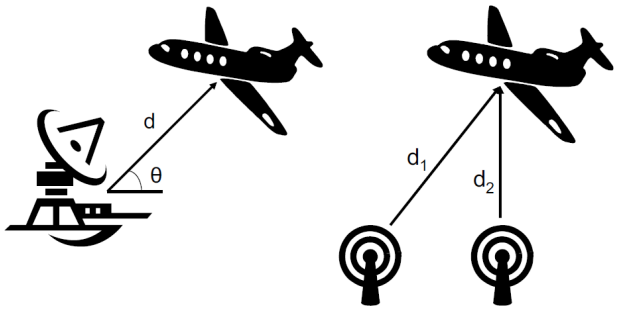


Figure 4.5: Illustration of the tracking problem: applying radar and triangulation measurements.

Next, two state space models are created which differ in measurement function and measurement noise model:

```
>>k0 = 0; Ts=0.01;
>>ssObjR=ss_DNL_AN(@TRACK_f,@TRACK_R_h,x0,w,vR,k0,Ts);
>>ssObjT=ss_DNL_AN(@TRACK_f,@TRACK_T_h,x0,w,vT,k0,Ts);
```

Both models are simulated, with an initial state that is equal to the estimated initial state, in order to obtain 50 measurements:

```
>>x0init=[-200;200;4;0];samples=50; conf='x y';u=[];
>>simObjR=sim(ssObjR,samples,u,conf,x0init);
>>simObjT=sim(ssObjT,samples,u,conf,x0init);
```

Now, the RMSE values of the true states and the analysis states of the different algorithms are calculated. To avoid misleading conclusions, the results are averaged over five experiments. The required code for the radar measurement becomes:

```
its=5;rmseArray=zeros(4,3);
for i=1:its
    damUKF=da(ssObjR,'UKF',simObjR,'x y');
    damETKF=da(ssObjR,'ETKF',simObjR,'x y',{50});
    damPFASIR=da(ssObjR,'PF_ASIR',simObjR,'x y',{200});
    rmseArray(:,1)=rmseArray(:,1)+rmse(damUKF,(1:4));
    rmseArray(:,2)=rmseArray(:,2)+rmse(damETKF,(1:4));
    rmseArray(:,3)=rmseArray(:,3)+rmse(damPFASIR,(1:4));
end
rmseArray=rmseArray./its;
```

Notice that the ETKF and PF_ASIR are configured with respectively 50 ensemble members and 200 particles. Next, the same procedure is applied using the triangulation models, i.e. with objects `ssObjT`, `simObjT`. The obtained RMSE values of both models are summarized in table 4.1. Notice how the UKF and the ETKF provide better estimations then the PF_ASIR for this experiment.

	RADAR			TRIANGULATION		
State	UKF	ETKF	PF_ASIR	UKF	ETKF	PF_ASIR
1	2.4905	2.4885	3.3594	6.4709	6.4395	7.7609
2	7.5299	7.9214	9.0680	11.188	11.231	9.9894
3	0.7018	0.7205	0.7665	2.1839	2.2532	2.4159
4	1.1816	1.2932	1.3933	1.2048	1.2548	1.2289

TABLE 4.1: The tracking problem. RMSE values of estimated states and true states as obtained from the UKF, ETKF and PF_ASIR for radar and triangulation measurements.

4.4 The scalar nonlinear system

The scalar nonlinear system is given by the following equations:

$$x_{k+1} = \frac{1}{2}x_k + \frac{25x_{k-1}}{1+x_{k-1}^2} + 8\cos[1.2(k-1)] + w_k \tag{4.15}$$
$$y_k = \frac{1}{20}x_k^2 + v_k$$

where w_k and v_k are zero-mean Gaussian noise, both with variances equal to 1. The initial state estimation is $x_0 = 0.1$ and the initial error covariance is $P_0 = 2$.

The state of this system is estimated using the Generic Particle filter (PF_GEN). After implementing the user-defined functions, the noise models are defined as

```
>> x0={0.1,2};w={0,1};v={0,1};
```

Then, the state space model is defined by

```
>> ssObj=ss_DNL_AN(@NLSC_f,@demo_NLSC_h,x0,w,v);
```

This short syntax automatically assumes a sampling time of one second and an initial step number `k0=0`. Subsequently the state space model is simulated to obtain 40 measurements as follows:

```
x0init=0.1;samples=40; conf='x y';u=[];
simObj=sim(ssObj,samples,u,conf,x0init);
```

Notice that the initial simulation state is set equal to the initial state estimation. The Generic particle filter with 50 particles is now applied on the state space model and the measurements as acquired from the simulation model

```
damASIR=da(ssObj,'PF_ASIR',simObj,'x y Pa',{50});
```

To check the accuracy of the state estimation, a plot of the true state and the analysis state with its 95% confidence interval is made using the following command:

```
plot(damASIR,'xaxPa',1);
```

The corresponding plot is shown in Figure 4.6. Notice that the true state is always located between the confidence intervals.

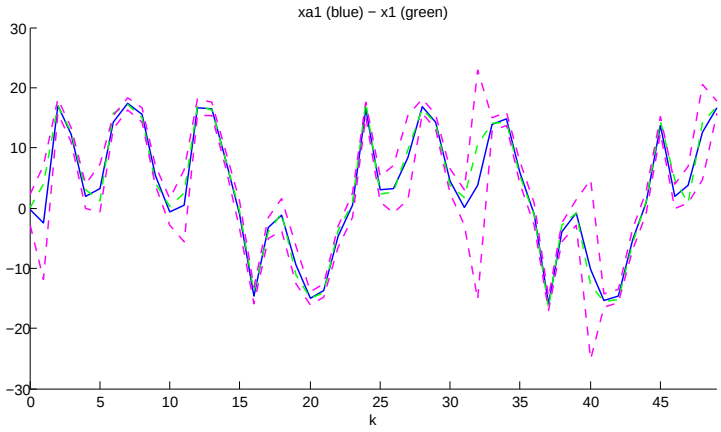


Figure 4.6: Illustration of a pre-configured data assimilation plot: true states, analysis states and the 95% confidence interval of the scalar problem.

Chapter 5

Conclusion

The goal of this thesis was to develop a generic data assimilation toolbox for MATLAB with at least five data assimilation schemes for improving the estimations of a given model defined by the user. This initial goal is clearly achieved. Even more, the goal has been extended by providing eleven data assimilation schemes that range from the regular Kalman filter to the particles filters. More specifically the data assimilation toolbox contains the following data assimilation techniques: the Kalman Filter (KF), the Extended Kalman Filter (EKF), the Unscented Kalman Filter (UKF), the Ensemble Kalman Filter (EnKF), the Deterministic Ensemble Kalman Filter (DEnKF), the Ensemble Transform Kalman Filter (ETKF), the Ensemble Square Root Filter (EnSRF), the Optimal Interpolation (OI) technique, the Generic Particle filter (GEN), the Sampling Importance Resampling (SIR) Particle filter and the Auxiliary Sampling Importance Resampling (ASIR) Particle filter.

The toolbox contains three classes to model Gaussian noise. The first class is solely used to configure linear time-invariant Gaussian noise, the second one is intended for time-variant Gaussian noise and the third one allows the configuration of any user-defined Gaussian noise. The state space models can be defined using any of the three predefined state space classes in the toolbox. The first class is suited to define linear state space models with a LTI or LTV behaviour. The second class is indented for nonlinear state space models with additive noise, while the last one can handle a completely nonlinear state space model. In addition, the acquired simulation and data assimilation data are carefully stored in simulation and data assimilation classes which are equipped with several tools to analyze their data content.

By applying the MATLAB object-oriented programming approach, the toolbox not only provides the required generic behaviour, but also maintains a structured framework that can be easily extended with new algorithms and classes. Despite the extensive framework and its numerous possibilities, the execution of a data assimilation algorithm is still straightforward and is typically performed in a few command lines.

Besides the requirement of a generic toolbox, the performance of the toolbox is implicitly an important concern. Therefore, the methods of the state space models

and the noise models have been developed with great care towards performance. Also the data assimilation algorithms have been implemented with a proper balance between performance, memory usage and readability. Still, it must be noted that the MATLAB object-oriented approach is typically slower than its procedural variant of programming, mainly because of the calling of properties and methods. This is however not a major concern since in the case of data assimilation virtually all of the computational time is typically spent on model evaluation.

A toolbox is by nature never finished and expansions will most surely be made. Adding new data assimilation techniques or extending an existing algorithm are some trivial possibilities. Apart from the algorithms complexity, the implementation of a new algorithm is a relatively easy expansion. This is mainly due to the fact that all noise model classes, as well as all state space model classes, are equipped with methods that have an identical syntax. As such, a developer does not need to generate an extensive list of conditional statements for all possible class cases. A second possible expansion is to add state space model classes. An interesting class would consist of a state space model with nonlinear state equation and linear measurement equation. This system is often encountered in practice and could be defined as class ss_DNL_DL_AN. The advantage of adding such a class is mainly inspired by user convenience. It is clear that defining one or two system matrices requires less configuration time than making a user-defined function. Also, if an algorithm requires the calculation of Jacobian matrices, the user will not need to provide these for the measurement equation. Adding a state-space model would typically take a few hours since this is only a matter of combining existing pieces of code. A third possible expansion is to add non-Gaussian noise models. Finally, the available tools for simulation and especially data assimilation models are easily extended.

Of course there is always room for improvements. Possible improvements are mainly related to the presentation of the toolbox. For example, besides the typical help files, MATLAB offers the possibility to publish parts of code into 'html' files that can be included in the MATLAB help section. As such, it would be possible to search for specific terms which is very helpful for an initial user. The toolbox demos could be published in a similar fashion, after which they can be added to the official MATLAB demos. Adding a graphical user interface (GUI) would also create an added value to the user experience. Although most toolboxes are typically used with the command line, a GUI is often more appealing to a new user.

Appendix A

Installation and folder structure

To install the data assimilation toolbox only two steps are required. First place the main folder 'DA_Toolbox$y$_z' in your favorite MATLAB directory (y and z denote the major and minor revision). Next, open the main folder and enter *init* or *init*(0) on the command line. Calling *init* adds and saves all required paths to the MATLAB search path. If you want to add the paths but don't want to save them for future sessions use *init*(0).

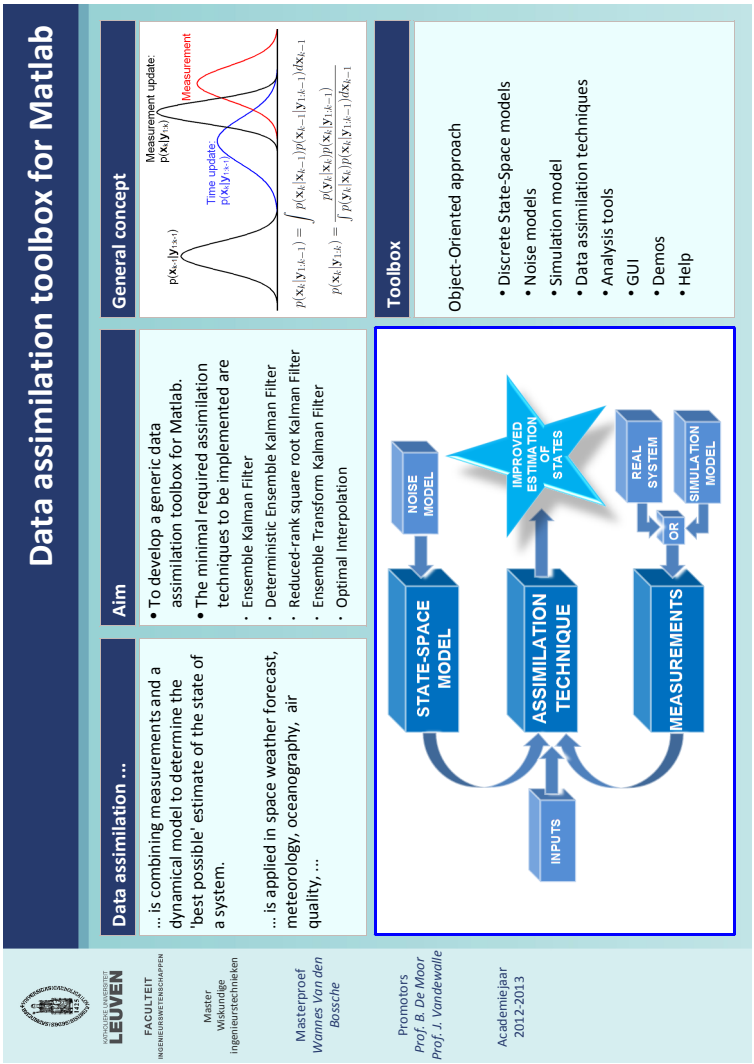
The main folder structure is now summarized:

- da: contains the all the classes, i.e. the noise models, state space models, simulation models and data assimilation models.
- dademos: contains all the demo files.
- daguis: contains all files related to the GUI.
- dahelp: contains all files related to help instructions such as published code.
- daobsolete: contains obsolete files of a previous revision.
- daresource: contains additional resources that are not code related such as icons and images.
- datemplates: contains templates to help creating user defined functions that are required as handles in some classes, i.e. nm_gauss_handle, ss_DNL and ss_DNL_AN
- dautils: contains utility files. These files are typically functions which have a general nature (not tailored for the toolbox).

Note that some folders and subfolders are currently empty. They are just defined to aid in future developments of the toolbox.

Appendix B

Poster



Appendix C

Accompanying paper

Data assimilation toolbox for Matlab

Wannes Van den Bossche

***Abstract*—Although data assimilation is an active field of research, there is currently no official Matlab data assimilation toolbox. This paper presents a new Matlab toolbox that is fitted for this purpose. The toolbox provides a generic platform for data assimilation techniques where any technique can be applied on any user defined model. To aid the user in building their specific model equations three state space representations are available which are designed to facilitate the configuration of any linear or nonlinear model. Also, three different classes of Gaussian noise allow a fast configuration of any type of Gaussian noise source. In addition, the toolbox contains several tools that allow a fast analysis of the acquired data assimilation records. At this moment the toolbox has eleven data assimilation techniques at its disposal. These techniques are the Kalman Filter (KF), the Extended Kalman Filter (EKF), the Unscented Kalman Filter (UKF), the Ensemble Kalman Filter (EnKF), the Deterministic Ensemble Kalman Filter (DEnKF), the Ensemble Transform Kalman Filter (ETKF), the Ensemble Square Root Filter (EnSRF), the Optimal Interpolation (OI) technique, the Generic Particle filter (GEN), the Sampling Importance Resampling (SIR) Particle filter and the Auxiliary Sampling Importance Resampling (ASIR) Particle filter.**

I. Introduction

THIS paper introduces a MATLAB toolbox for data assimilation. Although data assimilation is an active field of research, there is currently no official Matlab data assimilation toolbox. This new toolbox provides a generic platform for data assimilation techniques where any technique can be applied on any user defined model. By applying the MATLAB object-oriented programming approach, the toolbox not only provides a generic behaviour, but also maintains a structured framework that can be easily extended with new algorithms and classes.

The first section of this paper provides a short summary of the different algorithms that are implemented in the toolbox. The second section describes the general architecture of the toolbox. Since the toolbox is developed using the object-oriented approach, this section mainly identifies and describes the different classes and methods of the toolbox. The third and last section illustrates how to apply the toolbox on a typical example, the Lorenz equations.

II. The data assimilation techniques

There are eleven data assimilation techniques available in the toolbox. The first technique is the well known Kalman filter (KF) which is the optimal filter in case of a linear state space model with white, zero-mean and uncorrelated Gaussian noise sources. The KF has the disadvantages that it can only be applied on linear models and that it is not suited for large-scale systems. The Extended Kalman Filter (EKF) addresses the problem of nonlinearity by linearizing the nonlinear system around the Kalman filter state estimates. The EKF is often applied due to its simplicity. Still, it has some clear disadvantages. It is not suited for large-scale systems and the required Jacobian matrices can be computationally expensive to calculate or they are error-prone if provided manually. Furthermore, the linearization is only of order one and is therefore inaccurate when applying to very nonlinear systems [1]. The Unscented Kalman Filter (UKF) avoids using the Jacobians by propagating deterministically selected sigma points through the state equations whose sample pdf is used to approximate the true pdf[2]. Still, the amount of required sigma points has the same order as the amount of states. As such, the UKF is not suitable for large-scale systems. The Optimal Interpolation technique (OI) is suited for large-scale systems assuming a fixed error covariance matrix.

The Ensemble Kalman Filter (EnKF) [3] uses a similar approach as the UKF, but it is designed specifically to estimate the states of large-scale systems. Unlike the UKF, the EnKF allows an heuristic amount of samples, called the ensemble members. This allows the user to choose a proper balance between computational load and quality of the state estimation. The size of an ensemble is typically chosen much smaller than the amount of states in case of large-scale systems. Furthermore, the ensemble is not selected deterministically but randomly (Monte Carlo method). More precisely, it is assumed that the ensemble members are samples from the pdf of the true state [4]. A downside of the ensemble Kalman filter is that it uses perturbated observations to correctly estimate the analysis error covariance matrix. However, as shown in many papers, there are two downsides of this solution. First, the perturbations only solve the underestimation problem in a statistical sense for an infinite amount of ensembles. Second, the perturbations introduce additional sampling errors. This makes the ensemble Kalman filter suboptimal, especially when using few ensemble members. The Ensemble Square Root Filter (EnSRF), the Ensemble Transform Kalman Filter (ETKF) and the Deterministic Ensemble Kalman Filter (DEnKF) offer a deterministic alternative to the perturbated observations. The EnSRF provides improved accuracy at the same computational effort, but only under the restriction of uncorrelated measurement noise [5]. The ETKF is a fast variant of square root filters. The DEnKF provides similar performance as square root filters while preserving the convenient structure of the EnKF, although restricted by the assumptions of small forecast corrections [6].

The last class of filters, the particle filters, are not Kalman based filters but can be considered as a brute force Monte Carlo implementation of the recursive Bayesian filter. The

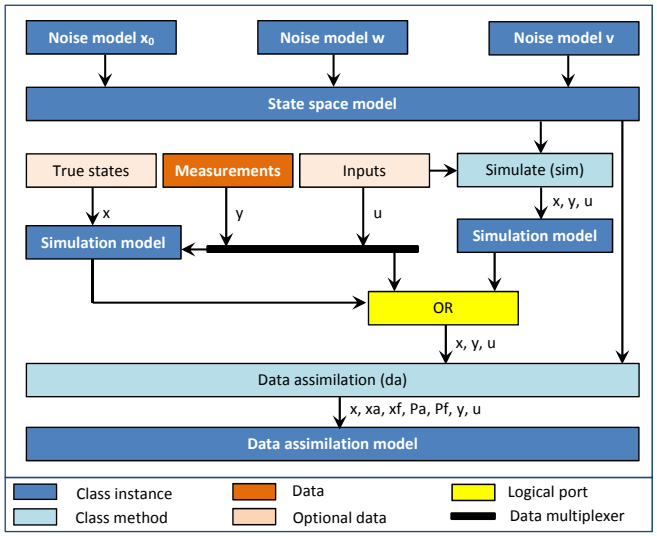


Fig. 1. Simulation Results.

particle filter uses random samples, called particles, to approximate the required pdf's. In case of highly nonlinear and/or non-Gaussian noise sources they often perform better than the Kalman based filters, although typically at a much higher computational cost. Moreover, since a large amount of particles is needed in general, they are not suited for large-scale systems. The Generic particle filter (PF_GEN) as implemented in the toolbox is described in [7]. It is a variant of the Sequential Importance Sampling (SIS) algorithm and solves the degeneracy phenomenon by applying a resampling algorithm whenever the estimated effective sample size falls below a user-defined threshold. The Sequential Importance Resampling particle filter (PF_SIR) is another variant that resamples every iteration. The downside of this filter is that resampling every iteration can cause a loss of diversity in the particles. Finally, the Auxiliary Sequential Importance Resampling particle filter (PF_ASIR) uses an auxiliary variable to prevent sample impoverishment.

III. Toolbox Architecture

The goal of the toolbox is to provide a generic environment to incorporate any data assimilation technique for any user defined state space model. The object-oriented programming possibilities of MATLAB provide an excellent framework to attain such a generic toolbox. By defining the main objects as classes, a clear program structure is obtained that can be easily expanded. The main classes and the most important methods are illustrated in Figure 1 which shows the typical workflow of a data assimilation experiment. This workflow starts at the top of the schematic and goes in several steps to the bottom. There are typically four steps needed to perform a data assimilation experiment. First the noise models are configured, then a state space model is configured, after which the measurements are simulated (when unavailable) and finally a data assimilation technique is applied.

A. The noise model

The first type of classes in the toolbox are the noise models which represent any kind of probability distribution. At this moment only the Gaussian noise model is defined in the toolbox. Extending the toolbox with additional distributions is possible in a straightforward manner. A Gaussian noise model can be configured with one of the following three classes: the linear time-invariant (LTI) Gaussian noise class, the linear time-variant (LTV) Gaussian noise class and the function handle Gaussian noise class. The LTI variant is the most straightforward, its mean and covariance are configured with a vector and a matrix. The LTV variant provides time-variant behaviour by allowing the mean or covariance, or both, to be defined by a three dimensional array where the third dimension represents different step numbers. The function handle class is the most general variant since for this class the mean and covariance are defined by user-defined functions.

B. The state space model

The second type of classes are the discrete-time state space models. There are currently three classes that can be used to construct three different types of state space models. These models can be ranked hierarchically from linear at the bottom to completely nonlinear at the top. When ascending in this hierarchy each model is a more general case of the previous one. All models contain the vectors $x_k \in \mathbb{R}^n$, $y_k \in \mathbb{R}^m$ and $u_k \in \mathbb{R}^p$ which denote the state vector, the measurement vector and the input vector at time k. The vectors $w_k \in \mathbb{R}^n$ and $v_k \in \mathbb{R}^m$ represent the process and measurement noise for which the stochastic processes $\{w_k\}_{k=0}^{\infty}$ and $\{v_k\}_{k=0}^{\infty}$ originate from a known type of distribution. The different state space models that can be configured in the toolbox are now summarized.

- *The discrete-time linear state space model* is the least general state space model. Its equations are

$$\begin{aligned} x_{k+1} &= A_k x_k + B_k u_k + w_k, \\ y_k &= C_k x_k + D_k u_k + v_k, \end{aligned} \tag{1}$$

  where $A_k \in \mathbb{R}^{n\times n}$, $B_k \in \mathbb{R}^{n\times p}$, $C_k \in \mathbb{R}^{m\times n}$ and $D_k \in \mathbb{R}^{m\times p}$ are respectively the state, input, output and feedthrough matrices at time k.
- *The discrete-time nonlinear state space model with additive noise* is the least general nonlinear state space model in the toolbox. Its equations are

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, k) + w_k, \\ y_k &= h(x_k, u_k, k) + v_k, \end{aligned} \tag{2}$$

  where the system equation $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R} \to \mathbb{R}^m$ are possibly nonlinear functions.
- *The discrete-time nonlinear state space model* is the most general model where both measurement and process noise are allowed to be nonlinear. Its equations are

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, w_k, k), \\ y_k &= h(x_k, u_k, v_k, k), \end{aligned} \tag{3}$$

where the system equation $f : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^n \times \mathbb{R} \rightarrow \mathbb{R}^n$ and the measurement equation $h : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^m \times \mathbb{R} \rightarrow \mathbb{R}^m$ are possibly nonlinear functions. It must be noted that this model is not often encountered in practice.

The main parameters to configure a state space model are the noise models, the system equations and the sampling time. In case of a linear state space model, the system equations are configured with either regular matrices to obtain LTI behaviour or with 3D-arrays to obtain a LTV behaviour. The system equation f and the measurement equation h of the nonlinear state space models are configured with user-defined functions. All state space models require three noise models: the initial state estimation x_0, the process noise w and the measurement noise v.

C. The measurements

Once the state space model is properly configured, there are two possible ways to proceed which depend on the nature of the experiment. Either the experiment is based on a real setup where measurements are available or the experiment has a more theoretical nature and no measurements are available. In the first case, the real measurements can be used in the data assimilation method. In the last case the measurements can be obtained by using the simulation method which uses the configured state space model and inputs (if applicable) to simulate both states and measurements. It starts by drawing a sample from the initial state estimate x_0 and then runs sequentially through the system and measurement equations, while adding respectively samples from the process and measurement noise. The user defines which outputs of the simulation method, i.e. the true states x, the measurements y and the inputs u, should be saved in a simulation model. The simulation model class is an advanced MATLAB structure that allows fast post-processing of the retrieved data, e.g. by providing predefined plots.

D. The data assimilation

Any data assimilation technique can now be applied on the configured state space model and available measurements. The default output of the data assimilation method is the analysis states x_a. If desired, also the true states x (if available), the forecast states x_f, the analysis error covariances P_a, the forecast error covariances P_f, the measurements y and the inputs u (if applicable) can be requested. Note that only the variances of the covariance matrices are returned by default, since the matrices become huge when dealing with large-scale systems. The requested variables are saved in a data assimilation model which contains all desired information regarding a data assimilation test. This class contains several methods to analyze the obtained data, such as comparison plots of estimated states versus true states, plots of the estimated states with their 95% confidence intervals or RMSE calculations of the different states.

IV. THE TOOLBOX COMMAND LINE SYNTAX

The most important command line syntaxes to create and manipulate the main objects are now briefly explained. As already indicated, the main objects are the Gaussian noise models, the discrete-time state space models and the simulation and data assimilation models.

A. The Gaussian noise models

There are three possible classes available to create a Gaussian noise object: the nm_gauss_lti, nm_gauss_ltv and nm_gauss_handle classes.

1) Linear Time Invariant Gaussian noise (nm_gauss_lti): The most general way of constructing a nm_gauss_lti object is `obj = nm_gauss_lti(mu,Sigma)`. The input arguments of this constructor are:

- `mu` - the mean of the distribution which is defined as a column vector.
- `Sigma` - the covariance matrix of the distribution which can be defined in two ways. The first possibility a symmetric matrix, but if the noise is uncorrelated, a column vector that contains the variances is also allowed.

2) Linear Time Variant Gaussian noise (nm_gauss_ltv): The most general way of constructing a nm_gauss_ltv object is `obj = nm_gauss_ltv(mu,Sigma,kIndex,kMethod)`. The input arguments of this constructor are:

- `mu` - the mean of the distribution which can be defined in two ways. When time invariant, `mu` is defined with a column vector as in class nm_gauss_lti . In case of a time variant behaviour, `mu` is configured as a 3D-array of column vectors where each vector corresponds to the step number as defined in the property `kIndex`.
- `Sigma` - the covariance matrix of the distribution which can be defined in two ways. When time invariant, `Sigma` is defined with either a matrix or a column vector (when uncorrelated) as in class nm_gauss_lti. In case of a time variant behaviour, `Sigma` is configured as a 3D-array of either column vectors or matrices where each vector/matrix corresponds to the step number as defined in the property `kIndex`.
- `kIndex` - vector that contains the step numbers that correspond to each vector or matrix in the 3D-arrays `mu` and/or `Sigma`. When not specified, the default content is set to $[0 \;\; 1 \;\; \ldots \;\; l]$ where l is the third dimension of the 3D-array.
- `kMethod` - string value that indicates the rounding method to retrieve the correct 2D-array from `mu` or `Sigma` at step k . The following string values are possible:
  - `'low'`: given k, extracts the 2D-array corresponding to the lower bound step number value in `kIndex`. (This is the default setting)
  - `'high'`: given k, extracts the 2D-array corresponding to the higher bound step number value in `kIndex`.
  - `'near'`: given k, extracts the 2D-array corresponding to the nearest bound step number value in `kIndex` .

3) Function handle Gaussian noise (nm_gauss_handle): The most general way of constructing a nm_gauss_handle ob-

ject is `obj = nm_gauss_handle(mu,Sigma,Ts)`. The input arguments of this constructor are:

- `mu` - function handle to a function that returns the mean of the distribution, e.g. `@funmu`. The function must have the step number k and the sample time `Ts` as input parameters and should return the mean as a column vector.
- `Sigma` - function handle to a function that returns the covariance matrix of the distribution, e.g. `@funsigma`. The function must have the step number k and the sample time `Ts` as input parameters and should either return the entire covariance matrix or the variances as a column vector.
- `Ts` - sample time of the noise model that is provided to the functions. (Default value=1)

B. The discrete-time state space models

There are three possible classes available to create a discrete-time state space object: the ss_DL, ss_DNL_AN and ss_DNL classes.

1) The discrete-time linear model (ss_DL): This class represents the model with system equations (1). The most general way of constructing a ss_DL object is `obj = ss_DL(A,B,C,D,x0,w,v,k0,Ts,TimeUnit, kIndex,kMethod)`. The input arguments of this constructor are:

- `A`, `B`, `C`, `D` - the system matrices which can be either regular matrices or 3D-arrays.
- `x0` - initial state noise model.
- `w` - process noise model.
- `v` - measurement noise model.
- `k0` - initial step number. (Default=0)
- `Ts` - sample time. (Default=1)
- `TimeUnit` - string that represents the unit of the time variable. (Default= `'seconds'`)
- `kIndex` - vector that contains the step numbers that correspond to each vector or matrix when one or more system matrices are configured as 3D-arrays. For more information see Section IV-A2
- `kMethod` - string value that indicates the rounding method to retrieve the correct 2D-array from a 3D-array at step k. For more information see Section IV-A2

2) The discrete-time nonlinear model with additive noise (ss_DNL_AN): This class represents the model with system equations (2). The most general way of constructing a ss_DL_AN object is `obj = ss_DNL_AN(f,h,x0,w,v,k0,Ts,TimeUnit, fJacX,hJacX)`. The input arguments of this constructor are:

- `f`, `h` - function handles for the functions f and h that return an update value of their respective equations without noise, e.g. `@fun`. The syntax of the function files should be `newval = fun(x,k,u,~,Ts)`.
- `fJacX`, `hJacX` - function handles for the functions that respectively return the Jacobians of f and h with respect to x. The syntax of the function files should be `newval = funjac(x,k,u,~,Ts)`. If a function handle is not provided or set to an empty matrix, its Jacobian is estimated numerically.
- `x0`, `w`, `v`, `k0`, `Ts`, `TimeUnit` - see ss_DL.

3) The discrete-time nonlinear model (ss_DNL): This class represents the model with system equations (3). The most general way of constructing a ss_DNL object is `obj = ss_DNL(f,h,x0,w,v,k0,Ts,TimeUnit, fJacX,fJacW,hJacX,hJacV)`. The input arguments of this constructor are:

- `f`, `h` - function handles for the functions that return an update value of respectively the state and measurement equations, e.g. `@fun`. The syntax of the function files should be `newval = fun(x,k,u,nv,Ts)` where the noise vector `nv` represents either `w` or `v`.
- `fJacX`, `hJacX` - function handles for the functions that respectively return the Jacobians of f and h with respect to x. The syntax of the function files should be `newval = funjac(x,k,u,nv,Ts)` where the noise vector `nv` represents either `w` or `v`. If a function handle is not provided or set to an empty matrix, its Jacobian is estimated numerically.
- `fJacW`, `hJacV` - function handles for the functions that respectively return the Jacobians of f and h with respect to w and v. The syntax of the function files should be `newval = funjac(x,k,u,nv,Ts)` where the noise vector `nv` represents either `w` or `v`. If a function handle is not provided or set to an empty matrix, its Jacobian is estimated numerically.
- `x0`, `w`, `v`, `k0`, `Ts`, `TimeUnit` - see ss_DL.

C. The simulation method and model

The simulation method can be applied on any state space object `obj`. In its most general form the syntax is `simObj=sim(obj,samples,u,conf,x0,noise)` which simulates the states `x` and measurements `y` of the discrete-time state space model `obj` by iterating through the state and measurement equations while applying the inputs `u`. Its output argument is the simulation object `simObj` of class sim_D which acts as a container for the requested data output. The different input arguments of the simulation method are:

- `samples`: indicates how many measurement or state samples should be acquired. (Default=1)
- `u`: a matrix of inputs where each column vector represents an input (Default = `[]`).
- `conf`: string that indicates which variables must be saved in the simulation object `simObj`. Possible string values contain `'x'`, `'u'` and `'y'`. Note that the variable `y` is always saved regardless of the configuration. (Default setting = `'x u y'`)
- `x0`: initial state column vector. (Default=`[]`) When empty, a sample is taken from the initial state noise model `x0` that resides in `obj`.
- `noise`: option that allows to specify the noise behaviour during simulation. Possible values are `noise=0` for no noise, `noise=1` to only simulate the process noise, `noise=2` to only add measurement noise and `noise=3`

to simulate both process and measurement noise. (Default=3)

The data of the obtained simulation object `simObj` can be plotted with a pre-configured plot command with syntax `plot(simObj,conf,nrs,varargin)` or `plot(simObj,conf,nrs1,nrs2,varargin)`. The input arguments of the plot method are:

- `simObj`: a simulation object of class sim_D.
- `conf`: string that defines the variable(s) to plot, the desired scale of the x-axis and the type of plot. Possible values are `'u * **'`, `'x * **'`, `'y * **'` and `'xy *'`, where
  - `'*'` can be either `'k'` to plot in function of the step index or `'t'` to plot in function of time and
  - `'**'` can be either `'1'` to create a single plot which contains all variables or `'2'` to create a subplot per variable.
- `nrs`: vector that contains the variable numbers that need to be plotted. A maximum of 15 variables can be plotted.
- `nrs1` and `nrs2`: when `conf` contains `'xy'` two vectors need to be supplied, where the first one indicates the variable numbers for the states and the second one indicates the variable numbers for the measurements. The size of both vectors need to be equal.
- `varargin`: additional arguments as in regular plot function. When these are specified, the predefined settings such as colors, titles and the legend are not included any more.

D. The data assimilation method and model

Any algorithm method can be configured in the following ways:

1) `daObj=da(ssObj,alg,simmodel,outconf, optionscell)` - performs the data assimilation with a simulation object.
2) `daObj=da(ssObj,alg,y,u,outconf, optionscell)` - performs the data assimilation with real measurements.

All algorithms return a data assimilation model `daObj` which acts as a container for all requested data. The different input arguments of the algorithms are:

- `ssObj`: a state space object.
- `alg`: a string that contains the requested algorithm. Possible string values are: 'KF', 'EKF', 'UKF', 'EnKF', 'DEnKF', 'EnSRF', 'ETKF', 'OI', 'PF_GEN', 'PF_SIR' and 'PF_ASIR'.
- `simmodel`: a simulation model object.
- `y`: a matrix of measurements where each column vector is a measurement.
- `u`: a matrix of inputs where each column vector is an input.
- `outconf`: a string that specifies which variables should be saved in the data assimilation object. Possible string values contain `'xf'`, `'Pf'`, `'Pa'`, `'x'`, `'y'`, `'u'` and `'cov'`. The string `'cov'` indicates that the full covariance matrix should be saved instead of the vectors of variances.
- `optionscell`: cell array of algorithm specific settings. The complete list of settings is too extensive to include in this paper. Typical settings include the amount of ensemble members or the amount particles among others.

The data of the obtained data assimilation object `daObj` can be analyzed with several methods:

- `plot(daObj,conf,nrs,varargin)` or `plot(daObj,conf,nrs1,nrs2,varargin)` - overloaded plot method that allows fast creation of plots using predefined layouts. The input arguments of the plot method are similar to the ones as described for the simulation object. The main difference is that the parameter `conf` has the extra possible string values `'xf'`, `'xa'`, `'Pf'`, `'Pa'`, `'xaPa'`, `'xaxPa'`, `'xfPf'`, `'xax'`, `'xaxy'` and `'xa-x'`.
- `val=rmse(daObj,nrs)` - returns a column vector `val` that holds the RMSE values of the true states `x` and assimilated analysis states `xa` that reside in the dam_D object `obj`. The array `nrs` contains the state numbers that have to be taken into account.
- `val=mse(daObj,nrs)` - is completely similar to `val=rmse(daObj,nrs)` but calculates the MSE values.

V. Illustrated example

This section illustrates how to apply the data assimilation toolbox on a typical system. The system that is taken under consideration consists of the Lorenz equations. The Lorenz equations are a simplified mathematical model for atmospheric convection that became famous for their chaotic solutions that resemble a butterfly. They are defined by three ordinary differential equations:

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y-x), \\ \frac{dy}{dt} &= x(\rho-z)-y, \\ \frac{dz}{dt} &= xy-\beta z, \end{aligned} \tag{4}$$

where σ, ρ and β are the system parameters which are set to $\sigma = 10$, $\rho = 28$ and $\beta = 8/3$. With these settings the system exhibits the typical chaotic behaviour as observed by Lorenz. To implement this continuous-time model into the discrete-time toolbox, the user-defined function that contains the state equation integrates the Lorenz equations between two consecutive steps using the Runge-Kutta $ode45$ solver of MATLAB. Hence, the user defined function is set up as follows:

```
function x1 = L3_f(x,k,u,~,Ts)

 %L3f: contains Lorenz equations
 [TOUT,x1] = ode45(@L3f,[k*Ts  (k+1)*Ts],x);
 x1=x1(end,:)';

end
```

As such, the system equations are now defined as follows:

$$\begin{aligned} x_{k+1} &= f(x_k) + w_k \\ y_k &= x_{1,k} + v_k \end{aligned} \tag{5}$$

with $x_k = [x_{1,k} \;\; x_{2,k} \;\; x_{3,k}]^T$, $f(x_k)$ the integrated Lorenz model and where $w_k \in \mathbb{R}^3$ and $v_k \in \mathbb{R}$ are zero-mean Gaussian noise with respective covariance matrices

$$Q = \begin{bmatrix} 10^{-3} & 0 & 0 \\ 0 & 10^{-3} & 0 \\ 0 & 0 & 10^{-3} \end{bmatrix} \quad \text{and} \quad R = \begin{bmatrix} 2 \end{bmatrix}. \qquad (6)$$

Furthermore, the initial state estimation is defined by the zero vector $x_0 = [-5.8 \;\; -5.7 \;\; 20.5]^T$ and the unity error covariance matrix $P_0 = I_3$.
The typical procedure of performing a data assimilation technique starts by defining the noise models. This is done as follows:

```
% Config. initial state
>> x0=[-5.8;-5.7;20.5];
% Config. process noise
>> w={diag([1e-3  1e-3  1e-3])};
% Config. meas. noise
>> v={0,2};
```

Before creating the state space model, the sample time, the initial step number and the unit of time are defined as follows:

```
>> k0 = 0;              %intial step number
>> Ts=0.01;             %sample time
>> TimeUnit='seconds'; %unit of time
```

Since the process noise of the Lorenz equations is additive, the ss_DNL_AN class is selected to create the state space model. First, the user-defined functions $L3\_f$ and $L3\_h$ for respectively the state and measurement equation are configured. Please remind that only the first variable is measured. Now the state space object is constructed as follows:

```
>> ssObj=ss_DNL_AN(@L3_f,@L3_h,x0,w,v,...
                                k0,Ts,TimeUnit);
```

The obtained state space model `ssObj` is now simulated over 2000 iterations, starting from the initial state $[-6 \;\; -6 \;\; 20]^T$ while saving the true states and measurements in the simulation object `simObj` as follows:

```
>> x0init=[-6;-6;20];
>> samples=2000;
>> conf='x y';
>> u=[];
>> simObj=sim(ssObj,samples,u,conf,...
                                    x0init);
```

Now that the state space model and the measurements are available, any nonlinear filter can be applied. In this example The EnKF algorithm is applied with 40 ensemble members, while saving the true states and measurements in the data assimilation object `damEnKF`:

```
>> damEnKF=da(ssObj,'EnKF',simObj,...
                              'x y',{40});
```

Now, the obtained results can be analyzed with several preconfigured tools. A first possibility is to apply the plot command on the data assimilation object using the syntax

```
>> plot(damDEnKF,'xax',(1:3));
```

which delivers three subplots where each subplot indicates the true states and analysis states as illustrated in Figure 2. It is clear that all states are very well estimated in which case it is often more useful to use the following command

```
>> plot(damDEnKF,'xa-x 1 t',(1:3));
```

which delivers a single plot of the difference between the true states and analysis states in function of time as illustrated in Figure 3. Notice that the errors of the unmeasured states $x_{2,k}$ and $x_{3,k}$ are only slightly larger than those of the measured state $x_{1,k}$. It is also possible to obtain the RMSE values of true states and analysis states with the command

```
rmse(damDEnKF,(1:3));
```

which returns the column vector $[0.2992 \;\; 0.4761 \;\; 0.4896]^T$, where each element is a RMSE value of the corresponding state. Since this example contains only three states it is possible to plot a three-dimensional representation of the true and estimated states using the `plot3` command of MATLAB. The result, the typical butterfly pattern, is shown in Figure 4.

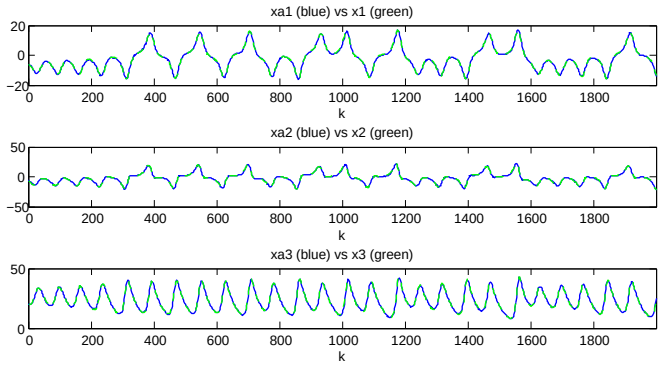


Fig. 2. Illustration of a pre-configured data assimilation plot: true states and analysis states of the Lorenz equations.

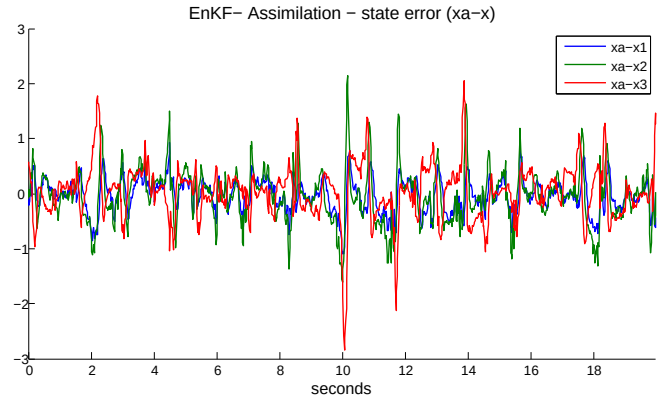


Fig. 3. Illustration of a pre-configured data assimilation plot: difference between true states and analysis states of the Lorenz equations.

VI. CONCLUSION

This paper has presented the MATLAB data assimilation toolbox that provides a generic platform to perform any data

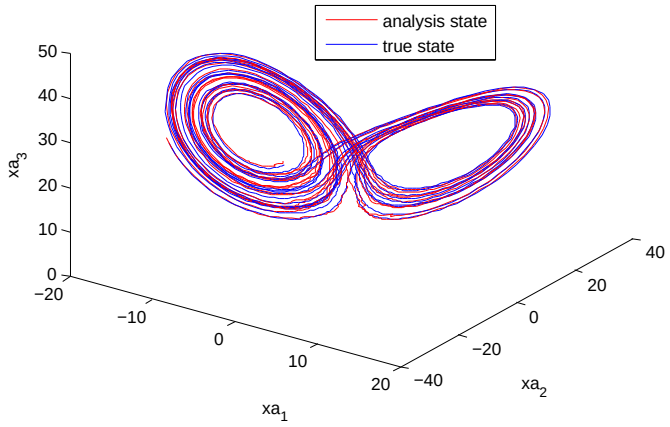


Fig. 4. Three-dimensional illustration of true states and analysis states of the Lorenz equations.

assimilation technique on any user defined function. The toolbox currently holds eleven data assimilation schemes that range from the regular Kalman filter to the particles filters. The solid framework of the toolbox is obtained by using the object-oriented possibilities of MATLAB. By selecting the classes with great care, the toolbox has received an intuitive and transparent structure that can be easily extended with new noise models, state space models and algorithms.

The typical workflow of implementing a data assimilation experiment can be summarized in four steps. The first step is to define the noise models for the initial state, the process noise and the measurement noise. In the second step the state space model is constructed whose equations are defined with matrices or user defined functions and the noise models of the first step. In the third step the measurements are either available or they are generated using the simulation method. In the fourth and final step the state space model and the measurements are provided to a data assimilation technique to obtain improved estimated states.

ACKNOWLEDGMENT

I would like to thank professor Bart De Moor and professor Joos Vandewalle for making this toolbox possible. I would like to express special thanks to Oscar Mauricio Agudelo for his enthusiasm towards this toolbox and whose suggestions and reviews have led to a better paper.

REFERENCES

[1] Simon J. Julier and Jeffrey K. Uhlmann, "Unscented filtering and nonlinear estimation," *Proceedings of the IEEE*, vol. 92, no. 3, pp. 401 – 422, 2004.

[2] D. Simon, *Optimal State Estimation: Kalman, H_∞, and Nonlinear Approaches*. Wiley-Interscience, 2006, iSBN-13: 978-0-471-70858-2.

[3] S. Gillijns, O. Barrero, J. Chandrasekar, B. De Moor, D. S. Bernstein, and A. Ridley, "What is the ensemble kalman filter and how well does it work?" *American Control Conference*, pp. 4448 – 4453, 2006.

[4] O.M. Agudelo, O. Barrero, P. Viaene, and B. De Moor, "Assimilation of ozone measurements in the air quality model aurora by using the ensemble kalman filter," *Proc. of 50th IEEE conference on Decision and Control and European Control Conference*, pp. 4430–4435, 2011.

[5] O. Barrero, "Data assimilation in magnetohydrodynamics systems using kalman filtering," Ph.D. dissertation, Katholieke Universiteit Leuven, 2005.

[6] Pavel Sakov. and Peter R. Oke, "A deterministic formulation of the ensemble kalman filter: an alternative to ensemble square root filters," *Tellus*, vol. 60, no. 2, pp. 361–371, 2008.

[7] M. Sanjeev Arulampalam, Simon Maskell, Neil Gordon, and Tim Clapp, "A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking," *IEEE Transactions on Signal Processing*, vol. 50, no. 2, pp. 174 – 188, 2002.

Bibliography

[1] O. Barrero. *Data Assimilation in Magnetohydrodynamics Systems Using Kalman Filtering*. PhD thesis, Katholieke Universiteit Leuven, 2005. URL: http://homes.esat.kuleuven.be/~bdmdotbe/newer/documents/doc_080328_10.26.pdf.

[2] David M. Livings, Sarah L. Dance, and Nancy K. Nichols. Unbiased ensemble square root filters. *Physica D: Nonlinear Phenomena*, 237(8):1021 – 1028, 2008.

[3] S. Gillijns. *Kalman Filtering Techniques for System Inversion and Data Assimilation*. PhD thesis, Katholieke Universiteit Leuven, 2007. URL: http://homes.esat.kuleuven.be/~bdmdotbe/newer/documents/doc_080325_15.09.pdf.

[4] J. D. Hol. Resampling in particle filters. URL: http://liu.diva-portal.org/smash/get/diva2:19698/FULLTEXT01, 2004. Internship report.

[5] Jeffrey S. Whitaker and Thomas M. Hamill. Ensemble data assimilation without perturbed observations. *Monthly Weather Review*, 130:1913–1924, 2002.

[6] M. Sanjeev Arulampalam, Simon Maskell, Neil Gordon, and Tim Clapp. A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174 – 188, 2002.

[7] MathWorks. *MATLAB Object-Oriented Programming*, r2012b edition. URL: http://www.mathworks.nl/help/pdf_doc/matlab/matlab_oop.pdf.

[8] MathWorks. *MATLAB Programming Fundamentals*, r2012b edition. URL: http://www.mathworks.nl/help/pdf_doc/matlab/matlab_prog.pdf.

[9] P. S. Maybeck. *Stochastic models, estimation and control*. Academic Press, 1979. ISBN: 0-12-480701-1.

[10] O.M. Agudelo, O. Barrero, P. Viaene, and B. De Moor. Assimilation of ozone measurements in the air quality model aurora by using the ensemble kalman filter. *Proc. of 50th IEEE conference on Decision and Control and European Control Conference*, pages 4430–4435, 2011.

[11] F. Orderud. Comparison of kalman filter estimation approaches for state space models with nonlinear measurements, 2005.

[12] Pavel Sakov and Peter R. Oke. Implications of the form of the ensemble transformation in the ensemble square root filters. *Monthly Weather Review*, 136:1042–1053, 2007.

[13] Pavel Sakov. and Peter R. Oke. A deterministic formulation of the ensemble kalman filter: an alternative to ensemble square root filters. *Tellus*, 60(2):361–371, 2008.

[14] S. Gillijns, O. Barrero, J. Chandrasekar, B. De Moor, D. S. Bernstein, and A. Ridley. What is the ensemble kalman filter and how well does it work? *American Control Conference*, pages 4448 – 4453, 2006. URL: ftp://ftp.esat.kuleuven.be/sista/gillijns/reports/TR-05-58.pdf.

[15] D. Simon. *Optimal State Estimation: Kalman, H_∞, and Nonlinear Approaches*. Wiley-Interscience, 2006. ISBN-13: 978-0-471-70858-2.

[16] Simon J. Julier and Jeffrey K. Uhlmann. Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, 92(3):401 – 422, 2004.

[17] Tombette, M., Mallet, V., and Sportisse, B. Pm_{10} data assimilation over europe with the optimal interpolation method. *Atmospheric Chemistry and Physics*, 9(1):57–70, 2009.

[18] D. Wetterdienst. Data assimilation. URL: http://www.dwd.de/.

Fiche masterproef

Student: Wannes Van den Bossche

Titel: Data assimilation toolbox for Matlab

Nederlandse titel: Data-assimilatie toolbox voor Matlab

UDC: 621.3

Korte inhoud:

Data-assimilatie is een algemene term voor het numerieke proces waarbij metingen optimaal geïntegreerd worden in modellen teneinde een betere schatting te verkrijgen van de werkelijkheid. Bij data-assimilatie wordt verondersteld dat zowel het model als de metingen onderhevig zijn aan storingen die voldoen aan bepaalde statistische verdelingen. Eén van de meest voorkomende toepassingen van data-assimilatie is meteorologie, maar er zijn nog talrijke andere toepassingen zoals oceanografie en het voorspellen van ruimteweer of luchtkwaliteit. Ondanks de toenemende populariteit van data-assimilatie is er geen officiële data-assimilatie toolbox aanwezig in Matlab. Deze masterproef stelt een nieuwe data-assimilatie toolbox voor die volledig ontwikkeld is in Matlab. De toolbox biedt een generiek platform aan die toelaat om data-assimilatie algoritmen toe te passen op eender welke, door de gebruiker gedefinieerde, functie. Door een object-georiënteerde aanpak biedt de toolbox een transparante en gestructureerde basis waaraan later eenvoudig nieuwe klassen en algoritmen kunnen toegevoegd worden. De toolbox beschikt momenteel over drie verschillende klassen voor het modelleren van Gaussiaanse ruis, drie verschillende klassen voor het modelleren van dynamische systemen en elf verschillende data-assimilatie algoritmen. De geïmplementeerde algoritmen zijn de Kalman Filter (KF), de Extended Kalman Filter (EKF), de Unscented Kalman Filter (UKF), de Ensemble Kalman Filter (EnKF), de Deterministic Ensemble Kalman Filter (DEnKF), de Ensemble Transform Kalman Filter (ETKF), de Ensemble Square Root Filter (EnSRF), de Optimal Interpolation (OI) techniek, de Generic Particle filter (GEN), de Sampling Importance Resampling (SIR) Particle filter en de Auxiliary Sampling Importance Resampling (ASIR) Particle filter.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: wiskundige ingenieurstechnieken

Promotoren: Prof. dr. ir. Bart De Moor
Prof. dr. ir. Joos Vandewalle

Assessoren: Prof. dr. ir. Raf Vandebril
Prof. dr. ir. Giovanni Samaey

Begeleiders: Dr. ir. Oscar Mauricio Agudelo
Dr. ir. Maarten Breckpot