Implementation and Testing of Sigma-Point Kalman Filters in Simulink for Nonlinear Estimation

Imanol Iriarte Arrese

Bachelor-Thesis – 15. August 2016

Betreuer: M.Sc. Bastian Ritter

TECHNISCHE UNIVERSITÄT DARMSTADT

REGELUNGSTECHNIK UND MECHATRONIK rtm

Scope of Work

Sigma-Point Kalman Filters are commonly used for state and parameter estimation purposes in various physical domains. In contrast to the classical Kalman Filter or the Luenberger Observer these filters are also applicable to general nonlinear systems, but, as all Kalman Filters, assume process and measurement noise to be Gaussian and white. One of those domains is the control of wind turbines. The implementation in Simulink allows using these filters in a complex simulation environment as part of advanced control systems.

Begin:	01. April 2016
End:	15. August 2016
Presentation:	30. August 2016

Prof. Dr.-Ing. Ulrich Konigorski

M.Sc. Bastian Ritter

Technische Universität Darmstadt
Institut für Automatisierungstechnik und Mechatronik
Fachgebiet Regelungstechnik und Mechatronik
Prof. Dr.-Ing. Ulrich Konigorski

Landgraf-Georg-Straße 4
64283 Darmstadt
Telefon 06151/16-4167
www.rtm.tu-darmstadt.de

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 15. August 2016

Imanol Iriarte Arrese

Abstract

This thesis discusses the implementation of Sigma-Point Kalman Filters (SPKF) for state estimation of nonlinear wind turbine systems. First, a theoretical review of nonlinear Kalman filtering is given. Then the different ways of implementing the algorithms and testing them in SIMULINK are discussed and eventually the developed algorithms are explained and illustrative results from nonlinear simulations are presented.

This work confirms that the linear Kalman Filter can be efficiently extended to nonlinear systems by means of Sigma-Point Kalman Filters such as the Unscented Kalman Filter and the Central Difference Kalman Filter. It is also shown that the performance of the square-root implementations available for SPKF is as accurate as that of the original ones, even if they are more computationally efficient algorithms.

Keywords: Nonlinear Filtering, Unscented Kalman Filter, Square Root Central Difference Kalman Filter, Wind Turbine Control, MATLAB/SIMULINK, CSparse/CXSparse

Contents

Symbols and Abbreviations

Model

Symbol	Description	Dimensions
N	Number of states	—
M	Number of inputs	—
r	Number of outputs	—
$F(\cdot)$	State equation of the nonlinear model	
$H(\cdot)$	Output equation of the nonlinear model	
$\boldsymbol{x}_k$	State vector at time step k	$N \times 1$
$\boldsymbol{u}_k$	Input vector at time step k	$M \times 1$
$\boldsymbol{y}_k$	Input vector at time step k	$r \times 1$
$\boldsymbol{w}_k$	Process noise vector at time step k	$N \times 1$
v_k	Measurement noise vector at time step k	$r \times 1$
$\mathbf{A}$	State-state matrix for discrete time linear model	$N \times N$
$\mathbf{B}$	State-input matrix for discrete time linear model	$N \times M$
$\mathbf{C}$	Output-state matrix for discrete time linear model	$r \times N$
$\mathbf{D}$	Output-input matrix for discrete time linear model	$r \times M$
$\bar{\boldsymbol{x}}_{\mathrm{OP}}$	Operating point of the linearized model	$N \times 1$

Symbol	Description	Unit
M_{g}	Generator torque	Nm
β_i	Blade i pitch angle	rad
$\ddot{x}_{\mathrm{T}}$	Nacelle fore-aft acceleration	m/s^2
$\ddot{y}_{\mathrm{T}}$	Nacelle side-side acceleration	m/s^2
n_{g}	Measured generator speed	rpm
φ	Rotor-Azimuth angle	°
$\dot{x}_{\mathrm{T}}$	Nacelle fore-aft velocity	m/s
$\dot{y}_{\mathrm{T}}$	Nacelle side-side velocity	m/s
$\dot{\varphi}_{\mathrm{g}}$	Generator angular speed	rad/s
$\Delta\dot{\varphi}$	Drive-train angular speed	rad/s
x_{T}	Nacelle fore-aft position	m
y_{T}	Nacelle side-side position	m
φ_{g}	Generator azimuth angle	rad
$\Delta\varphi$	Drive-train torsion	rad
v_{w}	Hub-height wind speed	rad/s

Filtering Algorithms

Symbol	Description	Dimensions
$\mathbf{Q}$	Process noise covariance matrix	$N \times N$
$\mathbf{R}$	Measurement noise covariance matrix	$r \times r$
$\hat{\boldsymbol{x}}_k^-$	*A priori* estimate of state vector at time step k	$N \times 1$
$\hat{\boldsymbol{x}}_k^+$	*A posteriori* estimate of state vector at time step k	$N \times 1$
$\hat{\boldsymbol{y}}_k$	Estimated Output at time step k	$r \times 1$
$\mathbf{P}_{\boldsymbol{x}_k}^-$	Error covariance matrix of *A priori* estimated state vector at time step k	$N \times N$
$\mathbf{P}_{\boldsymbol{x}_k}^+$	Error covariance matrix of *A posteriori* estimated state vector at time step k	$N \times N$
$\mathbf{P}_{\boldsymbol{y}_k}$	Error covariance matrix of the estimated output vector at time step k	$r \times r$
$\mathbf{P}_{\boldsymbol{x}_k\boldsymbol{y}_k}$	Cross covariance matrix, states-outputs at time step k	$N \times r$
$\mathbf{S}_{\boldsymbol{x}_k}^-$	Upper Cholesky Factor of Covariance matrix of $\boldsymbol{x}_k$ at time step k	$N \times N$
$\mathbf{S}_{\boldsymbol{y}_k}$	Upper Cholesky Factor of Covariance matrix of $\boldsymbol{y}_k$ at time step k	$r \times r$
$\boldsymbol{\mathcal{K}}_k$	Kalman Gain at time step k	$N \times r$
$\boldsymbol{\mathcal{X}}_{k-1}^+$	Sigma-Points representing the random variable $\boldsymbol{x}_{k-1}^+$	$N \times 2N+1$
${}^*\boldsymbol{\mathcal{X}}_k^-$	Sigma-Points representing the random variable $\boldsymbol{x}_k^-$ without process noise information	$N \times 2N+1$
$\boldsymbol{\mathcal{X}}_k^-$	Sigma-Points representing the random variable $\boldsymbol{x}_k^-$ with process noise information	$N \times 2N+1$
$\boldsymbol{\mathcal{Y}}_k$	Sigma-Points representing the random variable $\boldsymbol{y}_k$	$r \times 2N+1$
$\mathbf{D}_\mathrm{C}$	Cholesky downdate matrix	
$\boldsymbol{d}_i$	i^{th}-column of Cholesky downdate matrix $\mathbf{D}_\mathrm{C}$	
$(\cdot)_i$	i^{th}-column of a given matrix $(\cdot)$	

Symbol	Description	Default
α	Parameter setting the spread of sigma-points in unscented transformation	$10^{-4} \leq \alpha \alpha \leq 1$
κ	Parameter setting the spread of sigma-points in unscented transformation	$\kappa = 0$
λ	Parameter setting the spread of sigma-points in unscented transformation	—
η	Parameter setting the spread of sigma-points in unscented transformation	—
β	Parameter for covariance weighting in scaled unscented transformation	$\beta = 2$
h	Parameter setting the spread of sigma-points in CDKF	$h = \sqrt{3}$
$\omega_i^{(\mathrm{m})}$	Weighing factor of sigma-point i for estimation of the mean	—
$\omega_i^{(\mathrm{c})}$	Weighing factor of sigma-point i for estimation of the covariance	—

Abbreviations

Contraction	Complete meaning
KF	Kalman Filter
EKF	Extended Kalman Filter
SPKF	Sigma-Point Kalman Filter
UKF	Unscented Kalman Filter
SR-UKF	Square-Root Unscented Kalman Filter
CDKF	Central Difference Kalman Filter
SR-CDKF	Square-Root Central Difference Kalman Filter
CKF	Cubature Kalman Filter
SR-CKF	Square-Root Cubature Kalman Filter
CC	Compressed Column format
pdf	Probability Density Function

1 Introduction

As the wind energy technology advances, more complex control algorithms are required in order to make the turbine work within the desired conditions (guaranteeing stability, reducing vibrations or extracting as much energy from the wind as possible). These algorithms usually precise an accurate knowledge of the system states. However, wind turbines are complex nonlinear systems, with plenty of variables influencing them and it is not always possible to measure all the states through sensors (strain gauges, for instance, are often expensive and error-prone[1]). This work focuses on the implementation of nonlinear observers for the estimation of those states.

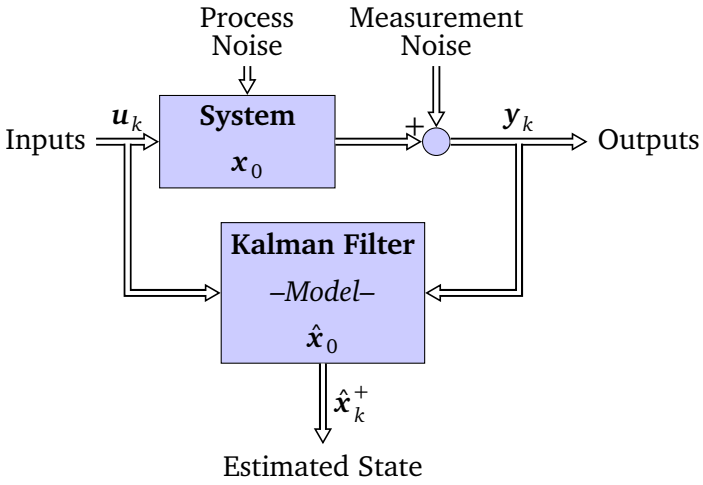

Figure 1.1.: Schematical representation of the Kalman Filter as an observer for systems with uncertainties.

However, the measurements supplied to the sensors for the estimation of the states are usually noisy, and there could be some modeling errors or some unknown dynamics happening inside the plant (see Figure 1.1). It is therefore reasonable to think about the best ways for the observers to deal with this uncertainty. This question was solved in the early 1960s by Rudolf E. Kálmán, when he developed what nowadays is known as *The Kalman Filter* [2].

This algorithm, which in the last 50 years had such different applications as the estimation of trajectories for the *Apollo* program or the prediction of the stock market behavior, can also be employed for the recursive state estimation of a given linear system, out of noisy measurements.

Nevertheless, as wind turbines are highly nonlinear systems, this work will focus on the implementations of variations of the traditional Kalman filtering algorithm for the nonlinear state estimation of wind turbines. Some of these modified algorithms can be considered as part of the class of *Sigma-Point Kalman Filters*.

1.1 Objectives

The main objective is the generation of efficient code for the implementation of the *Unscented Kalman Filter* (UKF), the *Central Difference Kalman Filter* (CDKF) and their *Square-Root* types (SR-UKF, SR-CDKF). This code must be validated within a SIMULINK simulation with a complex model of a wind turbine and should ideally be programmed in a way, which eases the future implementation of such algorithms into microcontrollers, for their real life application.

The specific objectives or working packages in which the project was divided to meet the general goal are the followings:

- Introduction to classical observer theory.
- Literature review for two types of the SPKF algorithms (UKF and CDKF + their SR-types).
- Profound understanding of nonlinear Kalman Filtering techniques.
- Introduction to MATLAB/SIMULINK and setup simulation for a given complex wind turbine model.
- Development of concepts, how to efficiently implement these algorithms in SIMULINK.
- Testing and validation with a complex nonlinear simulation model of a wind turbine.
- Evaluation of the filter performance.
- Documentation and discussion.
- Outlook to further research topics.

1.2 Overview

The report is structured in six chapters. The first one is the present *Introduction*, it is followed by a *Review of the Literature*, in which all the required algorithms are presented and explained. The next chapter, entitled *Implementation Possibilities* discusses the ways in which these algorithms can be implemented for their simulation in SIMULINK. Afterwards, in *Development of the Code*, the generated code is analyzed and explained, getting to the *Results and Discussion*, where the algorithms are tested by means of simulations. Eventually, the *Conclusion* summarizes the entire project, analyzes the achievements and purposes topics for future research. Some appendices are included with a description of the employed linearized model and extracts of the generated code.

2 Review of the Literature

In the following chapter a brief insight into the *State of the Art* is given. The main goal of the chapter is to summarize the existing theoretical tools for the accomplishment of the purposed task. Hence, a brief review on the current theory for state estimation is given, and the Sigma-Point Kalman filtering algorithms are presented.

2.1 Theoretical Introduction to State Estimation

The problem of estimating the state of a system, from which only the input and the output values can be observed has been approached in different ways over the last century. One of those ways is the Kalman Filter, which presents the benefit of being able to deal with noisy measurements and unknown dynamics represented by process noise. However, the Kalman Filter makes the following assumptions[1]:

- The system must be linear and fully-state observable.
- The process and measurement noise must be white and Gaussian and uncorrelated[2].

Even if no system satisfies perfectly these conditions in practice, some of them can approach this behavior, enabling to find very good approximations of their state. Nevertheless, the wind turbine model, whose state must be estimated, is a highly nonlinear model, so the approach of the simple linear Kalman Filter is not suitable.

The following step in the state estimation theory is given by the Extended Kalman Filter (EKF). This was the algorithm of choice over the past four decades when it came to nonlinear state estimation [5]. The underlying idea of this method consists in recursively linearizing the system using as linearization point the previous state estimate. By doing so, the filter adapts itself to the nonlinearities of the plant which leads to acceptable estimations when the system is nearly linear.

The usefulness of this method, however, gets undermined when it is applied to systems with severe nonlinearities. And it presents a couple of drawbacks [6]:

- Linearization around highly nonlinear points can lead to highly unstable filters.
- It is not applicable to systems whose Jacobian cannot be computed.

[1] If the Kalman Filter is applied to a nonlinear system with non-Gaussian noise, it is still the best linear filtering algorithm which exists [3], though sub-optimal

[2] It has constant power density all over the frequency spectrum and its value follows a Gaussian distribution with zero mean

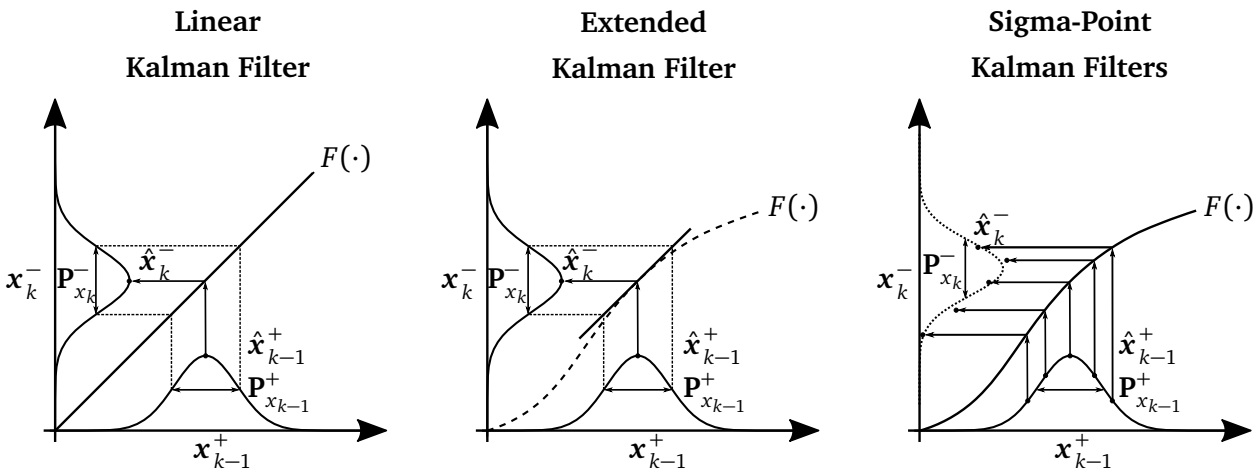

Figure 2.1.: Comparison of the working mechanism of the Kalman Filter, the Extended Kalman Filter and the Sigma-Point Kalman Filters to estimate the mean and covariance of a random variable passed through a function $F(\cdot)$ (adapted from [4]).

These limitations lead to the development of complexer algorithms, usually grouped under the name of *Sigma-Point Kalman Filters* (SPKF).

This family of algorithms deals with the mentioned problems by selecting a set of deterministically chosen points (Sigma Points) representing the whole distribution of the random variable. This set of points can be transformed through the nonlinear function in an easier way than the entire probability density function of the original random variable and the resulting distribution represents an accurate estimation of the value wanted to be estimated [6]. Figure 2.1 shows a comparison between the three different algorithms.

2.2 The Kalman Filter

This algorithm is the basic tool employed for signal estimation tasks since its development in the 1960s. Even if it is restricted to linear models, it is really useful for systems which can be described as linear or for systems that can be linearized around a usual working point. It constitutes the foundation over which all the complexer algorithms are built.

This algorithm relies on the recursive propagation of the mean and the covariance of the states through time [7]. At each time step two estimations are obtained, the *a priori* estimation $(\hat{\boldsymbol{x}}_k^-, \mathbf{P}_{\boldsymbol{x}_k}^-)$, which is obtained by employing the system's state model $F(\cdot)$ and the values of the previous step $(\hat{\boldsymbol{x}}_{k-1}^+, \mathbf{P}_{x_{k-1}}^+)$ and the *a posteriori* estimation, which is done by estimating the output of the system $\hat{\boldsymbol{y}}_k$ by means of the system's measurement model $H(\cdot)$ and comparing it with the measured output of the system $\boldsymbol{y}_k$ to correct the estimation. The Kalman Filter algorithm is therefore constructed of two recursive steps of *Time-update* and *Measurement-update*, which are depicted in Figure 2.2.

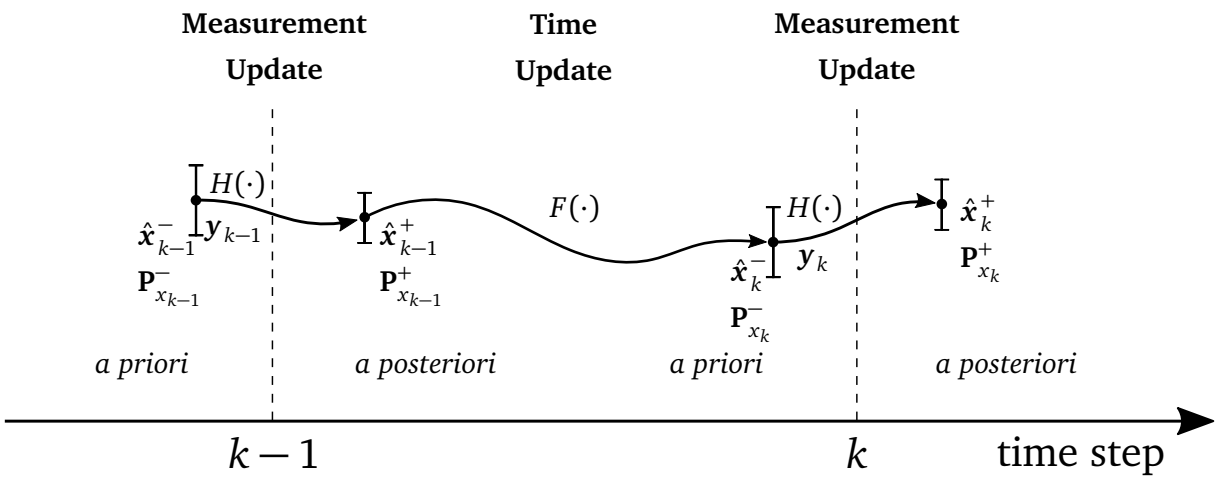

Figure 2.2.: Estimation steps in the Kalman filtering algorithm (adapted from [7]).

In order to understand the Kalman Filter, next discrete time linear system is considered, where both $F(\cdot)$ and $H(\cdot)$ are linear functions[3]:

$$\boldsymbol{x}_k = \mathbf{A}\boldsymbol{x}_{k-1} + \mathbf{B}\boldsymbol{u}_{k-1} + \boldsymbol{w}_{k-1} \tag{2.1}$$

$$\boldsymbol{y}_k = \mathbf{C}\boldsymbol{x}_k + v_k \tag{2.2}$$

Where, for the sake of simplicity, no direct feedthrough between the input $\boldsymbol{u}_k$ and the output $\boldsymbol{y}_k$ is assumed. The elements $\boldsymbol{w}_k$ and v_k represent respectively the process and the measurement noises; which are white, zero-mean, uncorrelated, and have known covariance matrices $\mathbf{Q}$ and $\mathbf{R}$ [7]:

$$\boldsymbol{w}_k \sim \mathcal{N}(0,\mathbf{Q}) \tag{2.3}$$

$$v_k \sim \mathcal{N}(0,\mathbf{R}) \tag{2.4}$$

$$E[\boldsymbol{w}_k \boldsymbol{w}_j^{\mathrm{T}}] = \mathbf{Q}\delta_{k-j} \tag{2.5}$$

$$E[v_k v_j^{\mathrm{T}}] = \mathbf{R}\delta_{k-j} \tag{2.6}$$

$$E[\boldsymbol{w}_k v_j^{\mathrm{T}}] = 0 \tag{2.7}$$

Where δ_{k-j} is the Kronecker-Delta distribution:

$$\delta_{k-j} = \begin{cases} 1 & \text{if } k = j \\ 0 & \text{if } k \neq j \end{cases} \tag{2.8}$$

Initialization

The filter is initialized by setting a value for the mean of the states and their covariance. The covariance matrix is a measure of how confident we are about these initial conditions. Both

[3] The here presented development was mainly adapted from [7]

magnitudes can be fixed based on our previous knowledge of the system (for instance, by means of past simulations):

$$\hat{\boldsymbol{x}}_0^+ = E[\boldsymbol{x}_0] \tag{2.9}$$

$$\mathbf{P}_{x_0}^+ = E[(\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0^+)(\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0^+)^{\mathrm{T}}] \tag{2.10}$$

Time-update

In order to obtain the next *a priori* estimates of the mean $\hat{\boldsymbol{x}}_k^-$ and covariance $\mathbf{P}_{\boldsymbol{x}_k}^-$ out of the initial values (or in general, the estimations at $k-1$) the following procedure must be followed:

$$\hat{\boldsymbol{x}}_k^- = \mathbf{A}\hat{\boldsymbol{x}}_{k-1}^+ + \mathbf{B}\boldsymbol{u}_{k-1} \tag{2.11}$$

$$\mathbf{P}_{\boldsymbol{x}_k}^- = \mathbf{A}\mathbf{P}_{x_{k-1}}^+\mathbf{A}^{\mathrm{T}} + \mathbf{Q} \tag{2.12}$$

The propagation of the mean $\hat{\boldsymbol{x}}_k^-$ is done by applying the dynamical equations of the model, whereas the expression for the propagation of the covariance can be developed from the following reasoning:

$$\mathbf{P}_{\boldsymbol{x}_k}^- = E[(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)^{\mathrm{T}}] \tag{2.13}$$

As the values of the inputs $\boldsymbol{u}_{k-1}$ are the same for the filters than for the system, if we assume no modeling errors,

$$(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-) = \mathbf{A}\boldsymbol{x}_{k-1} + \mathbf{B}\boldsymbol{u}_{k-1} + \boldsymbol{w}_{k-1} - \mathbf{A}\hat{\boldsymbol{x}}_{k-1} - \mathbf{B}\boldsymbol{u}_{k-1} = \mathbf{A}(\boldsymbol{x}_{k-1} - \hat{\boldsymbol{x}}_{k-1}^+) + \boldsymbol{w}_{k-1} \tag{2.14}$$

$$\begin{aligned}(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)^{\mathrm{T}} &= [\mathbf{A}(\boldsymbol{x}_{k-1} - \hat{\boldsymbol{x}}_{k-1}^+) + \boldsymbol{w}_{k-1}][\mathbf{A}(\boldsymbol{x}_{k-1} - \hat{\boldsymbol{x}}_{k-1}^+) + \boldsymbol{w}_{k-1}]^{\mathrm{T}} \\ &= \mathbf{A}(\boldsymbol{x}_{k-1} - \hat{\boldsymbol{x}}_{k-1}^+)(\boldsymbol{x}_{k-1} - \hat{\boldsymbol{x}}_{k-1}^+)^{\mathrm{T}}\mathbf{A}^{\mathrm{T}} + \boldsymbol{w}_{k-1}\boldsymbol{w}_{k-1}^{\mathrm{T}} \\ &\quad + \mathbf{A}(\boldsymbol{x}_{k-1} - \hat{\boldsymbol{x}}_{k-1}^+)\boldsymbol{w}_{k-1}^{\mathrm{T}} + \boldsymbol{w}_{k-1}(\boldsymbol{x}_{k-1} - \hat{\boldsymbol{x}}_{k-1}^+)^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\end{aligned} \tag{2.15}$$

And, as there is no correlation between $(\boldsymbol{x}_{k-1} - \hat{\boldsymbol{x}}_{k-1}^+)$ and $\boldsymbol{w}_{k-1}$[4], we obtain the previously presented result:

$$\mathbf{P}_{\boldsymbol{x}_k}^- = E\left[(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)^{\mathrm{T}}\right] = \mathbf{A}\mathbf{P}_{\boldsymbol{x}_{k-1}}^+\mathbf{A}^{\mathrm{T}} + \mathbf{Q} \tag{2.16}$$

This very same reasoning is applied to the equation of the output, getting an expression for the covariance of $\hat{\boldsymbol{y}}_k$ from the value of $\mathbf{P}_{\boldsymbol{x}_k}^-$:

$$\boldsymbol{y}_k - \hat{\boldsymbol{y}}_k = \mathbf{C}(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-) + v_k \tag{2.17}$$

[4] $(\boldsymbol{x}_{k-1} - \hat{\boldsymbol{x}}_{k-1}^+)$ depend on $\boldsymbol{w}_{k-2}$, but, as $\boldsymbol{w}$ is white noise, there is no correlation between $\boldsymbol{w}_{k-1}$ and $\boldsymbol{w}_{k-2}$ and therefore there is no correlation between $(\boldsymbol{x}_{k-1} - \hat{\boldsymbol{x}}_{k-1}^+)$ and $\boldsymbol{w}_{k-1}$.

$$
\begin{aligned}
\mathbf{P}_{y_k} &= E\left[(\boldsymbol{y}_k - \hat{\boldsymbol{y}}_k)(\boldsymbol{y}_k - \hat{\boldsymbol{y}}_k)^{\mathrm{T}}\right] \\
&= E\left[\mathbf{C}(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)^{\mathrm{T}}\mathbf{C}^{\mathrm{T}} + \boldsymbol{v}_k\boldsymbol{v}_k^{\mathrm{T}} + \mathbf{C}(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)\boldsymbol{v}_k^{\mathrm{T}} + \boldsymbol{v}_k(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)^{\mathrm{T}}\mathbf{C}^{\mathrm{T}}\right] \\
&= E\left[\mathbf{C}(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)^{\mathrm{T}}\mathbf{C}^{\mathrm{T}} + \boldsymbol{v}_k\boldsymbol{v}_k^{\mathrm{T}}\right] \\
&= \mathbf{C}\mathbf{P}_{\boldsymbol{x}_k}^-\mathbf{C}^{\mathrm{T}} + \mathbf{R}
\end{aligned} \tag{2.18}
$$

And eventually, the cross covariance can be computed as

$$
\begin{aligned}
\mathbf{P}_{\boldsymbol{x}_k\boldsymbol{y}_k}^- &= E\left[(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)(\boldsymbol{y}_k - \hat{\boldsymbol{y}}_k)^{\mathrm{T}}\right] \\
&= E\left[(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)^{\mathrm{T}}\mathbf{C}^{\mathrm{T}} + (\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)\boldsymbol{v}_k^{\mathrm{T}}\right] \\
&= E\left[(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)(\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k^-)^{\mathrm{T}}\mathbf{C}^{\mathrm{T}}\right] = \mathbf{P}_{\boldsymbol{x}_k}^-\mathbf{C}^{\mathrm{T}}
\end{aligned} \tag{2.19}
$$

Measurement-update

In this step of the algorithm the *a posteriori* estimates of the mean and covariance $(\hat{\boldsymbol{x}}_k^+, \mathbf{P}_{\boldsymbol{x}_k}^+)$ are inferred out of the *a priori* estimates $(\hat{\boldsymbol{x}}_k^-, \mathbf{P}_{\boldsymbol{x}_k}^-)$ and the current measurement of the system's output $\boldsymbol{y}_k$.

$$
\hat{\boldsymbol{x}}_k^+ = \hat{\boldsymbol{x}}_k^- + \mathcal{K}_k(\boldsymbol{y}_k - \hat{\boldsymbol{y}}_k) = \hat{\boldsymbol{x}}_k^- + \mathcal{K}_k(\boldsymbol{y}_k - \mathbf{C}\hat{\boldsymbol{x}}_k^-) = (\mathbf{I} - \mathcal{K}_k\mathbf{C})\hat{\boldsymbol{x}}_k^- + \mathcal{K}_k\boldsymbol{y}_k \tag{2.20}
$$

$$
\mathbf{P}_{\boldsymbol{x}_k}^+ = \mathbf{P}_{\boldsymbol{x}_k}^- - \mathcal{K}_k\mathbf{C}\mathbf{P}_{\boldsymbol{x}_k}^- = (\mathbf{I} - \mathcal{K}_k\mathbf{C})\mathbf{P}_{\boldsymbol{x}_k}^- \tag{2.21}
$$

Knowing the value of the cross covariance for a linear system (Eq. (2.19)) and that every covariance matrix is symmetric, this last expression can be generalized to [5]:

$$
\mathbf{P}_{\boldsymbol{x}_k}^+ = \mathbf{P}_{\boldsymbol{x}_k}^- - \mathcal{K}_k\mathbf{P}_{\boldsymbol{y}_k}\mathcal{K}_k^{\mathrm{T}} \tag{2.22}
$$

$\mathcal{K}_k$ is known as the *Kalman Gain* and it is computed as follows:

$$
\mathcal{K}_k = \mathbf{P}_{\boldsymbol{x}_k}^-\mathbf{C}^{\mathrm{T}}(\mathbf{C}\mathbf{P}_{\boldsymbol{x}_k}^-\mathbf{C}^{\mathrm{T}} + \mathbf{R})^{-1} = \mathbf{P}_{\boldsymbol{x}_k\boldsymbol{y}_k}\mathbf{P}_{\boldsymbol{y}_k}^{-1} \tag{2.23}
$$

The origin of this expressions can be found in the Recursive Least Squares Estimation (see following section). The Kalman Gain is essentially a matrix which is adapted at every time step for the minimization of the sum of the expected state estimation error (see Eq. (2.30)).

When the system has direct feedthrough,

$$
\boldsymbol{x}_k = \mathbf{A}\boldsymbol{x}_{k-1} + \mathbf{B}\boldsymbol{u}_{k-1} + \boldsymbol{w}_{k-1} \tag{2.24}
$$

$$
\boldsymbol{y}_k = \mathbf{C}\boldsymbol{x}_k + \mathbf{D}\boldsymbol{u}_k + \boldsymbol{v}_k \tag{2.25}
$$

the expression of the estimated output can be adapted to $\hat{\boldsymbol{y}}_k = \mathbf{C}\hat{\boldsymbol{x}}_k^- + \mathbf{D}\boldsymbol{u}_k$, leading to the following equation for the *a posteriori* state estimate:

$$
\hat{\boldsymbol{x}}_k^+ = \hat{\boldsymbol{x}}_k^- + \mathcal{K}_k(\boldsymbol{y}_k - \mathbf{C}\hat{\boldsymbol{x}}_k^- - \mathbf{D}\boldsymbol{u}_k) \tag{2.26}
$$

[5] See [3], Appendix A, for a more detailed explanation

In this section, a brief overview of the Recursive Least Squares Estimation is given, based on the deeper development which can be found in [7].

A linear recursive estimator is an algorithm which attempts to estimate a vector $\boldsymbol{x}$ based on the previous estimate $\hat{\boldsymbol{x}}_{k-1}$ and a noisy measurement $\boldsymbol{y}_k$

$$\boldsymbol{y}_k = \mathbf{C}\boldsymbol{x} + \upsilon_k \tag{2.27}$$

$$\hat{\boldsymbol{x}}_k = \hat{\boldsymbol{x}}_{k-1} + \mathcal{K}_k(\boldsymbol{y}_k - \mathbf{C}\hat{\boldsymbol{x}}_{k-1}). \tag{2.28}$$

Where the measurement $\boldsymbol{y}_k$ is a linear combination of the values of the vector $\boldsymbol{x}$ plus the noise υ_k and $\mathcal{K}_k$ represents the estimator gain matrix, to be determined. When computing the expected value of $\boldsymbol{x} - \hat{\boldsymbol{x}}_k$ in this filter, we get to the following result:

$$E[\boldsymbol{x} - \hat{\boldsymbol{x}}_k] = (\mathbf{I} - \mathcal{K}_k\mathbf{C})E[\boldsymbol{x} - \hat{\boldsymbol{x}}_{k-1}] - \mathcal{K}_k E[\upsilon_k] \tag{2.29}$$

Therefore, the estimator has the property of being *unbiased*, because independently of the $\mathcal{K}_k$ chosen, if $E[\boldsymbol{x} - \hat{\boldsymbol{x}}_{k-1}] = 0$ and $E[\upsilon_k] = 0$, $E[\boldsymbol{x} - \hat{\boldsymbol{x}}_k] = 0$ too. In other words, an appropriate set up of the initial value $\hat{\boldsymbol{x}}_0 = E[\boldsymbol{x}]$ leads to $E[\hat{\boldsymbol{x}}_k] = \boldsymbol{x}, \forall k$.

The estimator gain matrix can be chosen according to an optimality criterion; being this a Least Squares Estimator, this criterion is to have a $\mathcal{K}_k$ which minimizes the sum of the variances of the estimation errors at time k:

$$MIN\left\{E\left[(\boldsymbol{x}_1 - \hat{\boldsymbol{x}}_1)^2\right] + ... + E\left[(\boldsymbol{x}_n - \hat{\boldsymbol{x}}_n)^2\right]\right\} \tag{2.30}$$

Operating with this expression leads to the following result:

$$\mathcal{K}_k = \mathbf{P}_{x_{k-1}}\mathbf{C}^{\mathrm{T}}(\mathbf{C}\mathbf{P}_{x_{k-1}}\mathbf{C}^{\mathrm{T}} + \mathbf{R})^{-1} \tag{2.31}$$

$$\hat{\boldsymbol{x}}_k = \hat{\boldsymbol{x}}_{k-1} + \mathcal{K}_k(\boldsymbol{y}_k - \mathbf{C}\hat{\boldsymbol{x}}_{k-1}) \tag{2.32}$$

$$\mathbf{P}_{\boldsymbol{x}_k} = (\mathbf{I} - \mathcal{K}_k\mathbf{C})\mathbf{P}_{x_{k-1}}(\mathbf{I} - \mathcal{K}_k\mathbf{C})^{\mathrm{T}} + \mathcal{K}_k\mathbf{R}\mathcal{K}_k^{\mathrm{T}} \tag{2.33}$$

It is now possible to substitute the expression of $\mathcal{K}_k$ from Eq. (2.31) into the Eq. (2.33), which, after operating, leads to an easier expression for $\mathbf{P}_{\boldsymbol{x}_k}$:

$$\mathbf{P}_{\boldsymbol{x}_k} = (\mathbf{I} - \mathcal{K}_k\mathbf{C})\mathbf{P}_{x_{k-1}} \tag{2.34}$$

These are the equations for the recursive least squares estimator, however the objective of the Kalman Filter is not to estimate a constant vector, but to estimate a vector of time varying signals (the states of the system) so we have to adapt the equations of the recursive least squares estimator to the notation of the Kalman Filter (see Table 2.1).

Eventually getting the before purposed expressions:

$$\mathcal{K}_k = \mathbf{P}^-_{\boldsymbol{x}_k}\mathbf{C}^{\mathrm{T}}(\mathbf{C}\mathbf{P}^-_{\boldsymbol{x}_k}\mathbf{C}^{\mathrm{T}} + \mathbf{R})^{-1} = \mathbf{P}_{\boldsymbol{x}_k\boldsymbol{y}_k}\mathbf{P}^{-1}_{\boldsymbol{y}_k}$$

$$\hat{\boldsymbol{x}}^+_k = \hat{\boldsymbol{x}}^-_k + \mathcal{K}_k(\boldsymbol{y}_k - \hat{\boldsymbol{y}}_k)$$

$$\mathbf{P}^+_{\boldsymbol{x}_k} = \mathbf{P}^-_{\boldsymbol{x}_k} - \mathcal{K}_k\mathbf{C}\mathbf{P}^-_{\boldsymbol{x}_k}$$

Table 2.1.: Change of notation between algorithms

Least Squares Estimator		Kalman Filter
$\boldsymbol{x}$	$\rightarrow$	$\boldsymbol{x}_k$
$\hat{\boldsymbol{x}}_{k-1}$	$\rightarrow$	$\hat{\boldsymbol{x}}_k^-$
$\mathbf{P}_{\boldsymbol{x}_{k-1}}$	$\rightarrow$	$\mathbf{P}_{\boldsymbol{x}_k}^-$
$\hat{\boldsymbol{x}}_k$	$\rightarrow$	$\hat{\boldsymbol{x}}_k^+$
$\mathbf{P}_{\boldsymbol{x}_k}$	$\rightarrow$	$\mathbf{P}_{\boldsymbol{x}_k}^+$

2.3 Sigma-Point Kalman Filters

The Sigma-Point Kalman Filters is the name given by R. van der Merwe et. al. [3] to a family of Kalman Filters employed for nonlinear estimation. These filters select a set of points to capture the whole probabilistic distribution of a random variable and pass these points through the functions $F(\cdot)$ and $H(\cdot)$, which are not restricted to be linear. This process is depicted in Figure 2.3.

All presented algorithms will assume the additive noise condition, which implies that both measurement and process noise do not go through the nonlinear function, but are added to it (see Eq. (2.35) and (2.36)). The sigma-point algorithms are divided in the same steps as the Kalman Filter: *Initialization*, *Time-update* and *Measurement-update*.

In the Time-update, the distribution of the *a priori* state $\hat{\boldsymbol{x}}_k^-$ must be estimated based on the *a posteriori* estimate of the previous step $\hat{\boldsymbol{x}}_{k-1}^+$. In order to do so, a set of sigma points representing the Probability Density Function (pdf) of $\boldsymbol{x}_{k-1}^+$ are computed $\boldsymbol{\mathcal{X}}_{k-1}^+$, and passed through the nonlinear transformation $F(\cdot)$; getting the sigma points representing the PDF of the *a priori* state estimation of next step: ${}^*\boldsymbol{\mathcal{X}}_k^- = F(\boldsymbol{\mathcal{X}}_{k-1}^+, \boldsymbol{u}_{k-1})$. The estimated mean $\hat{\boldsymbol{x}}_k^-$ is computed and a new set of sigma points $\boldsymbol{\mathcal{X}}_k^-$ is generated for representing this variable. This is done to incorporate information about the process noise.

In the Measurement-update, a second nonlinear transformation must be applied. In this case, the starting sigma points are the ones representing the *a priori* state estimation, which, by means of the transformation $H(\cdot)$, turn into the sigma points of the estimated output $\hat{\boldsymbol{y}}_k$: $\boldsymbol{\mathcal{Y}}_k = H(\boldsymbol{\mathcal{X}}_k^-)$. This double nonlinear transformation process is depicted in Figure 2.3.

Once the estimated output is computed, the usual equations for the linear Kalman Filter can be employed. The SPKF are constructed over the same theoretical background as the usual Kalman Filter in which refers to the Kalman Gain; the only difference comes when evaluating the function and estimating the pdf of the next random variable.

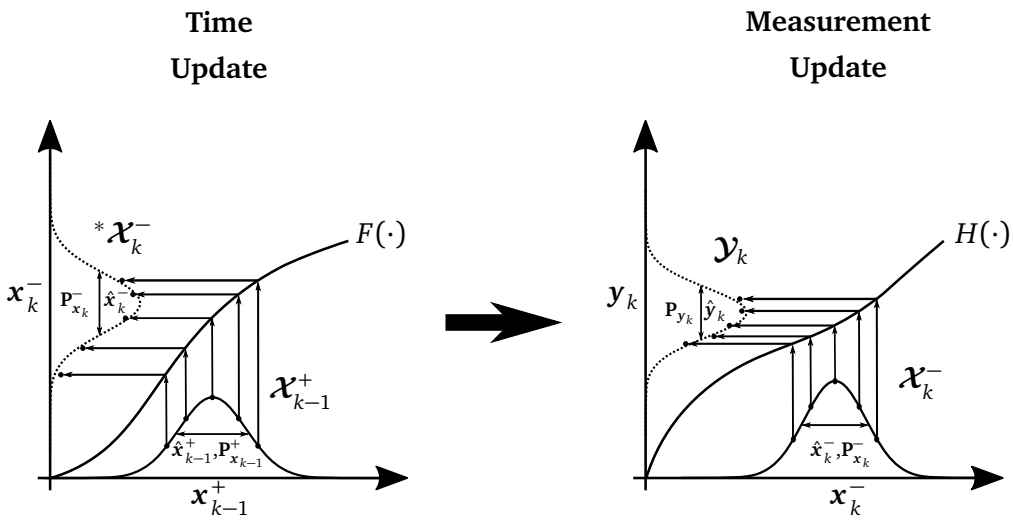

Figure 2.3.: Representation of the nonlinear transformations of the sigma-points for the time and for the measurement update. The algorithms reconstruct a Gaussian distribution from the transformed sigma-points.

In the following pages the Unscented Kalman Filter (UKF) and the Central Difference Kalman Filter (CDKF) will be analyzed, as well as the square-root versions of the algorithms (SR-UKF and SR-CDKF)[6].

There are however more algorithms based on a similar working mechanism, such as The Cubature Kalman Filter (CKF) or The Gauss-Hermite Kalman Filter (GHKF). These algorithms use the so called cubature points [1]. A deeper insight in the mentioned algorithms can be found, for instance, in [8].

2.3.1 The Unscented Kalman Filter

The Unscented Kalman Filter, proposed by Julier et. al. in 1997 [6] is a computationally better approach than the more traditional solutions, such as the Extended Kalman Filter, when it comes to nonlinear state estimation. The basic idea underlying the algorithm is to apply unscented transformations to a set of points (Sigma Points) representing a random variable, to estimate the probability density function of another random variable: the state in the next step (time update), or the output of the system (measurement update).

Given a discrete-time nonlinear system, with purely additive process and measurement noise ($\boldsymbol{w}_k$ and v_k),

$$\boldsymbol{x}_k = F(\boldsymbol{x}_{k-1}, \boldsymbol{u}_{k-1}) + \boldsymbol{w}_{k-1} \tag{2.35}$$

$$\boldsymbol{y}_k = H(\boldsymbol{x}_k, \boldsymbol{u}_k) + v_k \tag{2.36}$$

[6] All presented algorithms were mainly adapted from [3], the notation was however sometimes modified, looking for coherence inside the report and a more straightforward implementation of the algorithms in MATLAB.

the same conditions of uncorrelatedness for the noise that in the linear case will be assumed (see Eq. (2.3) to (2.7)). The algorithm gets once again split into three parts: *Initialization*, *Time-update* and *Measurement-update*.

Initialization

Similary to the linear case, the filter must be initialized by setting a value for the mean of the state and its error covariance:

$$\hat{\boldsymbol{x}}_0^+ = E[\boldsymbol{x}_0] \tag{2.37}$$

$$\mathbf{P}_0^+ = E[(\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0^+)(\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0^+)^{\mathrm{T}}] \tag{2.38}$$

Time-update

The first step is the selection of the sigma-points $\boldsymbol{\mathcal{X}}_{k-1}^+$ representing the *a posteriori* distribution of the previous state estimate $\hat{\boldsymbol{x}}_{k-1}^+$. This can be done in the following way:

$$\boldsymbol{\mathcal{X}}_{k-1}^+ = \left[\; \hat{\boldsymbol{x}}_{k-1}^+, \quad \hat{\mathbf{X}}_{k-1}^+ + \eta\sqrt{\mathbf{P}_{x_{k-1}}^+}, \quad \hat{\mathbf{X}}_{k-1}^+ - \eta\sqrt{\mathbf{P}_{x_{k-1}}^+} \; \right] \tag{2.39}$$

Where $\hat{\mathbf{X}}_{k-1}^+$ is a square matrix $N \times N$ such that

$$\hat{\mathbf{X}}_{k-1}^+ = \left[\hat{\boldsymbol{x}}_{k-1}^+, \ldots, \hat{\boldsymbol{x}}_{k-1}^+\right], \qquad \hat{\mathbf{X}}_{k-1}^+ \in \mathbb{R}^{N\times N} \tag{2.40}$$

and the matrix square-root of the covariance of the state

$$\sqrt{\mathbf{P}_{x_{k-1}}^+} = \mathrm{chol}^{\mathrm{T}}\left(\mathbf{P}_{x_{k-1}}^+\right) = (\mathbf{S}_{x_{k-1}}^+)^{\mathrm{T}} \tag{2.41}$$

is the lower Cholesky factor of $\mathbf{P}_{x_{k-1}}^+$. The selection of the sigma-points is based on the *Scaled Unscented Transformation* [3] and it consists of the mean of the distribution $\hat{\boldsymbol{x}}_{k-1}^+$ and a evenly distributed set surrounding it. The spread of this set depends on the square-root of the covariance matrix and a scaling coefficient

$$\eta = \sqrt{N + \lambda} \tag{2.42}$$

which actually depends on three other parameters: N equals the state dimension and

$$\lambda = \alpha^2(N + \kappa) - N. \tag{2.43}$$

Being α and κ the parameters which can be tuned to determine the spread of the sigma points (usually $10^{-4} \leq \alpha \leq 1$, $\kappa = 0$ [3]). It is therefore a good practice to keep α small in highly non-linear systems, avoiding non-local effects. By setting $\kappa \geq 0$ the covariance matrix is guaranteed to be positive semi-definite, its exact value is not critical though.

Besides, we must compute a set of $2N+1$ scalar weights for the $2N+1$ sigma points. In other words, a coefficient of the form ω_i is assigned to each sigma point $\boldsymbol{\mathcal{X}}_i$ which have to satisfy the condition

$$\sum_{i=0}^{2N-1} \omega_i = 1 \tag{2.44}$$

so that the estimated mean and covariance are not scaled by a factor.

An appropriate choice is the following [3]:

$$\begin{array}{lll} \text{for} & \boldsymbol{\mathcal{X}}_0 \rightarrow & \omega_0^{(m)} = \frac{\lambda}{N+\lambda} \\ \text{for} & \boldsymbol{\mathcal{X}}_i \rightarrow & \omega_i^{(m)} = \frac{1}{2(N+\lambda)} \quad i = 1...2N \end{array} \tag{2.45}$$

However, the weighting on the zero-th sigma-point directly affects the magnitude of the errors in the fourth and higher order terms for symmetric prior distributions [9], so another set of weights is chosen for the weighting of the covariance matrix

$$\begin{array}{lll} \text{for} & \boldsymbol{\mathcal{X}}_0 \rightarrow & \omega_0^{(c)} = \frac{\lambda}{N+\lambda} + (1-\alpha^2+\beta) \\ \text{for} & \boldsymbol{\mathcal{X}}_i \rightarrow & \omega_i^{(c)} = \frac{1}{2(N+\lambda)} \qquad\qquad \text{with } i = 1...2N \end{array} \tag{2.46}$$

where β is a new parameter that allows to minimize higher order errors if there is information about the distribution. For the Gaussian case the optimal choice is $\beta = 2$ [9].

It is now easy to apply the nonlinear transformation $F(\cdot)$ to the selected set, yelding

$${}^*\boldsymbol{\mathcal{X}}_k^- = F(\boldsymbol{\mathcal{X}}_{k-1}^+, \boldsymbol{u}_{k-1}) \tag{2.47}$$

$$\hat{\boldsymbol{x}}_k^- = \sum_{i=0}^{2N-1} \omega_i^{(m)} ({}^*\boldsymbol{\mathcal{X}}_k^-)_i \tag{2.48}$$

$$\mathbf{P}_{x_k}^- = \sum_{i=0}^{2N-1} \omega_i^{(c)} \left(({}^*\boldsymbol{\mathcal{X}}_k^-)_i - \hat{\boldsymbol{x}}_k^- \right) \left(({}^*\boldsymbol{\mathcal{X}}_k^-)_i - \hat{\boldsymbol{x}}_k^- \right)^{\mathrm{T}} + \mathbf{Q} \tag{2.49}$$

being $({}^*\boldsymbol{\mathcal{X}}_k^-)_i$ the i^{th} column of the matrix ${}^*\boldsymbol{\mathcal{X}}_k^-$.

Measurement-update

The nonlinear transformation $H(\cdot)$ must be applied to the set of sigma points, which leads to the estimated output $\hat{\boldsymbol{y}}_k$. It is however useful to modify this set of sigma points, including the information about the process noise present in $\mathbf{Q}$ (and hence in $\mathbf{P}_{x_k}^-$). This can be achieved in several ways [3] and in Section 2.3.2 another possible way of selecting the new set will be discussed. Here, however, the most usual way of extending the set will be presented, the one which will mainly be used along this report and in the implementations of the algorithms.

$$\boldsymbol{\mathcal{X}}_k^- = \left[\begin{array}{ccc} \hat{\boldsymbol{x}}_k^- & \hat{\mathbf{X}}_k^- + \eta\sqrt{\mathbf{P}_{x_k}^-} & \hat{\mathbf{X}}_k^- - \eta\sqrt{\mathbf{P}_{x_k}^-} \end{array} \right] \tag{2.50}$$

Where $\hat{\mathbf{X}}_k^-$ is a square matrix $N \times N$ such that

$$\hat{\mathbf{X}}_k^- = \left[\hat{\boldsymbol{x}}_k^-, \ldots, \hat{\boldsymbol{x}}_k^-\right], \qquad \hat{\mathbf{X}}_k^- \in \mathbb{R}^{N \times N} \tag{2.51}$$

and the matrix square-root of the covariance of the state

$$\sqrt{\mathbf{P}_{x_k}^-} = \mathrm{chol}^{\mathrm{T}}\left(\mathbf{P}_{x_k}^-\right) = (\mathbf{S}_{x_k}^-)^{\mathrm{T}} \tag{2.52}$$

is the lower Cholesky factor of $\mathbf{P}_{x_k}^-$. The transformation is now conducted for all sigma-points (2.50), and the mean and covariance of the new distribution are estimated

$$\boldsymbol{\mathcal{Y}}_k = H(\boldsymbol{\mathcal{X}}_k^-, \boldsymbol{u}_k) \tag{2.53}$$

$$\hat{\boldsymbol{y}}_k = \sum_{i=0}^{2N-1} \omega_i^{(m)} (\boldsymbol{\mathcal{Y}}_k)_i \tag{2.54}$$

$$\mathbf{P}_{y_k} = \sum_{i=0}^{2N-1} \omega_i^{(c)} \left((\boldsymbol{\mathcal{Y}}_k)_i - \hat{\boldsymbol{y}}_k\right)\left((\boldsymbol{\mathcal{Y}}_k)_i - \hat{\boldsymbol{y}}_k\right)^{\mathrm{T}} \tag{2.55}$$

where $(\boldsymbol{\mathcal{Y}}_k)_i$ represents the i^{th} column of $\boldsymbol{\mathcal{Y}}_k$. The cross covariance $\mathbf{P}_{x_k y_k}$ can be computed in the following way:

$$\mathbf{P}_{x_k y_k} = \sum_{i=0}^{2N-1} \omega_i^{(c)} \left((\boldsymbol{\mathcal{X}}_k^-)_i - \hat{\boldsymbol{x}}_k^-\right)\left((\boldsymbol{\mathcal{Y}}_k)_i - \hat{\boldsymbol{y}}_k\right)^{\mathrm{T}} \tag{2.56}$$

Which allows the computation of the Kalman Gain $\boldsymbol{\mathcal{K}}_k$ and, subsequently, weighting the difference between the estimated output and the measured output, the *a posteriori* estimated state and covariance:

$$\boldsymbol{\mathcal{K}}_k = \mathbf{P}_{x_k y_k} \mathbf{P}_{y_k}^{-1} \tag{2.57}$$

$$\hat{\boldsymbol{x}}_k^+ = \hat{\boldsymbol{x}}_k^- + \boldsymbol{\mathcal{K}}_k (\boldsymbol{y}_k - \hat{\boldsymbol{y}}_k^-) \tag{2.58}$$

$$\mathbf{P}_{\boldsymbol{x}_k}^+ = \mathbf{P}_{\boldsymbol{x}_k}^- - \boldsymbol{\mathcal{K}}_k \mathbf{P}_{y_k} \boldsymbol{\mathcal{K}}_k^{\mathrm{T}} \tag{2.59}$$

2.3.2 The Square-Root Unscented Kalman Filter

The Square-Root Unscented Kalman Filter (SR-UKF) is an algorithm developed to improve certain computational properties of the UKF, by simplifying some of its algebraic operations. Besides it has an improved numerical stability and guarantees positive semi-definiteness for the covariance matrices [10].

The computationally costliest operation of the UKF is the computation of the matrix square-root of the covariance for the determination of the sigma points at each time step. This is avoided in the SR-UKF by directly propagating the Cholesky factor $\mathbf{S}$ of the covariance.

In order to better understand the SR-UKF, it is necessary to recall the following linear algebra operations [3]:

- **Cholesky decomposition:** The Cholesky decomposition of a positive definite matrix $\mathbf{P}$, gives an upper triangular matrix $\mathbf{S}$ such that

$$\mathbf{P} = \mathbf{S}^{\mathrm{T}}\mathbf{S}. \tag{2.60}$$

This operation can be understood as the computation of the matrix square-root of $\mathbf{P}$ such that $\mathbf{S}=\mathtt{chol}(\mathbf{P})$. In this report, however, the matrix square-root has been defined as the lower Cholesky factor for the straightforward computation of the sigma-points (see Eq. (2.41)) so, $\sqrt{\mathbf{P}} = \mathbf{S}^{\mathrm{T}}$.

- **QR decomposition:** This is a more efficient decomposition of a matrix $\mathbf{X} \in \mathbb{R}^{L\times N}$, $L \geq N$ in the form

$$\mathbf{X} = \mathbf{QR}, \tag{2.61}$$

where $\mathbf{Q} \in \mathbb{R}^{L\times L}$ is orthonormal and $\mathbf{R} \in \mathbb{R}^{L\times N}$ is upper triangular. If we compute a matrix $\mathbf{P} \in \mathbb{R}^{N\times N}$, such that

$$\mathbf{P} = \mathbf{X}^{\mathrm{T}}\mathbf{X} = \mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{QR} \tag{2.62}$$

We get that, due to orthonormality of matrix $\mathbf{Q}$,

$$\mathbf{Q}^{\mathrm{T}}\mathbf{Q} = \mathbf{I} \tag{2.63}$$

And, if we define $\tilde{\mathbf{R}} \in \mathbb{R}^{N\times N}$ as the upper triangular part of $\mathbf{R}$, we get that $\tilde{\mathbf{R}}$ is the upper Cholesky factor of

$$\mathbf{P} = \mathbf{R}^{\mathrm{T}}\mathbf{R} = \tilde{\mathbf{R}}^{\mathrm{T}}\tilde{\mathbf{R}}. \tag{2.64}$$

The following notation will be employed further on: $\tilde{\mathbf{R}}=\mathtt{qr}(\mathbf{X})$.

- **Cholesky factor update:** If $\mathbf{S}$ is the upper triangular Cholesky factor of $\mathbf{P} = \mathbf{S}^{\mathrm{T}}\mathbf{S}$, i.e. $\mathbf{S}=\mathtt{chol}(\mathbf{P})$ and we define a matrix $\tilde{\mathbf{P}}$, such that[7]

$$\tilde{\mathbf{P}} = \mathbf{P} + \nu \cdot \boldsymbol{x}\boldsymbol{x}^{\mathrm{T}}, \tag{2.65}$$

where $\boldsymbol{x}$ is a column vector and ν is a scalar, the upper triangular Cholesky factor $\tilde{\mathbf{S}}$ of $\tilde{\mathbf{P}} = \tilde{\mathbf{S}}^{\mathrm{T}}\tilde{\mathbf{S}}$ can be computed as follows[8]

$$\tilde{\mathbf{S}} = \mathtt{chol}(\tilde{\mathbf{P}}) = \mathtt{cholupdate}(\mathbf{S}, \sqrt{\nu}\cdot\boldsymbol{x}) \tag{2.66}$$

[7] $\tilde{\mathbf{P}} = \mathbf{P} - \nu \cdot \boldsymbol{x}\boldsymbol{x}^{\mathrm{T}}$ for the Cholesky downdate

[8] $\tilde{\mathbf{S}} = \mathtt{cholupdate}(\mathbf{S}, \sqrt{\nu}\cdot\boldsymbol{x}, '-')$ for the Cholesky downdate

- **Efficient least squares:** The solution of an overdetermined problem $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, $\mathbf{A} \in \mathbb{R}^{L \times N}$, $L > N$ can be approximated by finding an $\boldsymbol{x}$ which minimizes the 2-norm error $||\mathbf{A}\boldsymbol{x} - \boldsymbol{b}||_2$. This $\boldsymbol{x}$ is the least squares solution and can be obtained from the equation

$$(\mathbf{A}^{\mathrm{T}}\mathbf{A})\boldsymbol{x} = \mathbf{A}^{\mathrm{T}}\boldsymbol{b}, \tag{2.67}$$

  provided that the pseudo-inverse $(\mathbf{A}^{\mathrm{T}}\mathbf{A})^{-1}$ exists. An efficient way of solving the system is based on employing the triangular QR decomposition with pivoting, implemented in Matlab's "\" operator. Equivalently in the "/" operator for the right side matrix multiplication in system $\mathbf{X}\mathbf{A}' = \mathbf{B}$, where $\mathbf{X}$ is an unknown matrix $\mathbf{X} \in \mathbb{R}^{M \times N}$, $\mathbf{A}' \in \mathbb{R}^{N \times L}$, $\mathbf{B} \in \mathbb{R}^{M \times L}$, and $L > N$ ($M = 1$ for the vectorial case).

The filtering algorithm is once again divided into three parts of *Initialization*, *Time-update* and *Measurement-update*.

Initialization

The filter gets initialized similarly to the previous ones, but considering the Cholesky factors of the covariance matrix and the noise matrices:

$$\hat{\boldsymbol{x}}_0^+ = E[\boldsymbol{x}_0] \tag{2.68}$$

$$\mathbf{S}_0^+ = \mathtt{chol}\{E[(\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0^+)(\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0^+)^{\mathrm{T}}]\} \tag{2.69}$$

$$\mathbf{S}_{\mathbf{Q}} = \left(\sqrt{\mathbf{Q}}\right)^{\mathrm{T}} = \mathtt{chol}(\mathbf{Q}) \tag{2.70}$$

$$\mathbf{S}_{\mathbf{R}} = \left(\sqrt{\mathbf{R}}\right)^{\mathrm{T}} = \mathtt{chol}(\mathbf{R}) \tag{2.71}$$

Time-update

First, the selection of the sigma points for the *a posteriori* distribution of the state in the previous step is conducted:

$$\boldsymbol{\mathcal{X}}_{k-1}^+ = \left[\begin{array}{ccc} \hat{\boldsymbol{x}}_{k-1}^+ & \hat{\mathbf{X}}_{k-1}^+ + \eta(\mathbf{S}_{x_{k-1}}^+)^{\mathrm{T}} & \hat{\mathbf{X}}_{k-1}^+ - \eta(\mathbf{S}_{x_{k-1}}^+)^{\mathrm{T}} \end{array}\right] \tag{2.72}$$

This set of sigma-points is passed through the nonlinear transformation $F(\cdot)$, computing the *a priori* estimate of the state at time k:

$${}^*\boldsymbol{\mathcal{X}}_k^- = F(\boldsymbol{\mathcal{X}}_{k-1}^+, \boldsymbol{u}_{k-1}) \tag{2.73}$$

$$\hat{\boldsymbol{x}}_k^- = \sum_{i=0}^{2N-1} \omega_i^{(m)} ({}^*\boldsymbol{\mathcal{X}}_k^-)_i \tag{2.74}$$

Where $({}^{*}\boldsymbol{\mathcal{X}}_k^-)_i$ denotes the i^{th} column of the matrix ${}^{*}\boldsymbol{\mathcal{X}}_k^-$. We can now propagate the Cholesky factor of the covariance matrix:

$$\mathbf{S}_{x_k}^- = \mathtt{qr}\left\{\begin{bmatrix} \sqrt{\omega_1^{(c)}} \cdot \left(({}^{*}\boldsymbol{\mathcal{X}}_k^-)_{1:2N-1} - \hat{\mathbf{X}}_{k_{\text{sp}}}^- \right) & \mathbf{S}_{\mathbf{Q}}{}^{\mathrm{T}} \end{bmatrix}^{\mathrm{T}}\right\} \tag{2.75}$$

$$\mathbf{S}_{x_k}^- = \mathtt{cholupdate}\left\{\mathbf{S}_{x_k}^-, \sqrt{\omega_0^{(c)}} \cdot \left(({}^{*}\boldsymbol{\mathcal{X}}_k^-)_0 - \hat{\boldsymbol{x}}_k^- \right)\right\} \tag{2.76}$$

Being here

$$\hat{\mathbf{X}}_{k_{\text{sp}}}^- = \left[\hat{\boldsymbol{x}}_k^-, \ldots, \hat{\boldsymbol{x}}_k^-\right], \qquad \hat{\mathbf{X}}_{k_{\text{sp}}}^- \in \mathbb{R}^{N \times 2N}. \tag{2.77}$$

This is done in two steps, in the first one, a QR decomposition is applied to the transpose of a matrix containing the weighted difference of the 1 to $2N$ sigma-points with the mean and the square-root of the process noise covariance matrix. The weighing factor is set to $\omega_1^{(c)}$ because $\omega_i^{(c)} = \omega_1^{(c)}, i = 1...2N$ (see Eq. (2.46)). Afterwards, a Cholesky factor update is carried out to incorporate the effects of the zero-th sigma-point. This is done because $\omega_0^{(c)}$ might be negative, which would lead to an error if we tried to compute the covariance by just employing the QR decomposition. The obtained matrix is equivalent to the Cholesky factor of the covariance matrix $\mathbf{P}_{x_k}^-$ employed in the UKF.

Proof: *Equivalence between SR-UKF and UKF. Time propagation of covariance.*

$$\mathbf{M} = \begin{bmatrix} \sqrt{\omega_1^{(c)}} \cdot \left(({}^{*}\boldsymbol{\mathcal{X}}_k^-)_{1:2N-1} - \hat{\mathbf{X}}_{k_{\text{sp}'}}^- \right) & \mathbf{S}_{\mathbf{Q}}{}^{\mathrm{T}} \end{bmatrix}^{\mathrm{T}} \tag{2.78}$$

$$\begin{aligned} {}^{*}\mathbf{P}_{x_k}^- &= \left(\mathbf{S}_{x_k}^-\right)^{\mathrm{T}} \mathbf{S}_{x_k}^- = \mathbf{M}^{\mathrm{T}}\mathbf{M} \\ &= \begin{bmatrix} \sqrt{\omega_1^{(c)}} \cdot \left(({}^{*}\boldsymbol{\mathcal{X}}_k^-)_{1:2N-1} - \hat{\mathbf{X}}_{k_{\text{sp}'}}^- \right) & \mathbf{S}_{\mathbf{Q}}{}^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} \sqrt{\omega_1^{(c)}} \cdot \left(({}^{*}\boldsymbol{\mathcal{X}}_k^-)_{1:2N-1} - \hat{\mathbf{X}}_{k_{\text{sp}'}}^- \right)^{\mathrm{T}} \\ \mathbf{S}_{\mathbf{Q}} \end{bmatrix} \\ &= \omega_1^{(c)} \cdot \left(({}^{*}\boldsymbol{\mathcal{X}}_k^-)_{1:2N-1} - \hat{\mathbf{X}}_{k_{\text{sp}'}}^- \right)\left(({}^{*}\boldsymbol{\mathcal{X}}_k^-)_{1:2N-1} - \hat{\mathbf{X}}_{k_{\text{sp}'}}^- \right)^{\mathrm{T}} + \mathbf{Q} \end{aligned} \tag{2.79}$$

$$\begin{aligned} \mathbf{P}_{x_k}^- &= {}^{*}\mathbf{P}_{x_k}^- + \omega_0^{(c)} \cdot \left(({}^{*}\boldsymbol{\mathcal{X}}_k^-)_0 - \hat{\boldsymbol{x}}_k^- \right)\left(({}^{*}\boldsymbol{\mathcal{X}}_k^-)_0 - \hat{\boldsymbol{x}}_k^- \right)^{\mathrm{T}} \\ &= \sum_{i=0}^{2N-1} \omega_i^{(c)} \left(({}^{*}\boldsymbol{\mathcal{X}}_k^-)_i - \hat{\boldsymbol{x}}_k^- \right)\left(({}^{*}\boldsymbol{\mathcal{X}}_k^-)_i - \hat{\boldsymbol{x}}_k^- \right)^{\mathrm{T}} + \mathbf{Q} \end{aligned} \tag{2.80}$$

The here shown reasoning demonstrates that the propagation of the covariance in the square-root formulation of the algorithms is completely equivalent to that of the original algorithm, even if the operations are more efficient computationally.

Measurement-update

At this point, the set of sigma-points ${}^{*}\boldsymbol{\mathcal{X}}_k^-$ can be extended or modified to another set before passing them through the function $H(\cdot)$, like in the previous case. However, two different

ways of proceeding are possible: computing $\boldsymbol{\mathcal{X}}_k^-$, as usual, or extending the existing set to an augmented set of $4N+1$ points, ${}^{(aug)}\boldsymbol{\mathcal{X}}_k^-$. Both transformations enable to incorporate the process noise on the observed sigma-points. The second option uses the previous set of points plus $2N$ new points, keeping more information about the odd-moments [3], but it also requires a higher computational effort. Even if this last option can be used for all analyzed SPKF algorithms, it will only be described here, as most of the implementations employ the first of the options.

- **Option 1:** Redrawing a complete set of sigma points

$$\boldsymbol{\mathcal{X}}_k^- = \begin{bmatrix} \hat{\boldsymbol{x}}_k^- & \hat{\mathbf{X}}_k^- + \eta(\mathbf{S}_{x_k}^-)^{\mathrm{T}} & \hat{\mathbf{X}}_k^- - \eta(\mathbf{S}_{x_k}^-)^{\mathrm{T}} \end{bmatrix} \tag{2.81}$$

It is now possible to compute the estimate of the output

$$\boldsymbol{\mathcal{Y}}_k = H(\boldsymbol{\mathcal{X}}_k^-, \boldsymbol{u}_k) \tag{2.82}$$

$$\hat{\boldsymbol{y}}_k = \sum_{i=0}^{2N-1} \omega_i^{(m)} (\boldsymbol{\mathcal{Y}}_k)_i, \tag{2.83}$$

as well as the Cholesky factor of the measurement covariance and the cross-covariance:

$$\mathbf{S}_{y_k} = \mathrm{qr}\left\{ \begin{bmatrix} \sqrt{\omega_1^{(c)}} \cdot \left((\boldsymbol{\mathcal{Y}}_k)_{1:2N-1} - \hat{\mathbf{Y}}_{k_{\mathrm{sp}}} \right) & \mathbf{S}_{\mathbf{R}}{}^{\mathrm{T}} \end{bmatrix}^{\mathrm{T}} \right\} \tag{2.84}$$

$$\mathbf{S}_{y_k} = \mathtt{cholupdate}\left\{ \mathbf{S}_{y_k}, \sqrt{\omega_0^{(c)}} \cdot \left((\boldsymbol{\mathcal{Y}}_k)_0 - \hat{\boldsymbol{y}}_k \right) \right\} \tag{2.85}$$

$$\mathbf{P}_{x_k y_k} = \sum_{i=0}^{2N-1} \omega_i^{(c)} \left((\boldsymbol{\mathcal{X}}_k^-)_i - \hat{\boldsymbol{x}}_k^- \right) \left((\boldsymbol{\mathcal{Y}}_k^-)_i - \hat{\boldsymbol{y}}_k \right)^T \tag{2.86}$$

With

$$\hat{\mathbf{Y}}_{k_{\mathrm{sp}}} = \left[\hat{\boldsymbol{y}}_k, \ldots, \hat{\boldsymbol{y}}_k \right], \qquad \hat{\mathbf{Y}}_{k_{\mathrm{sp}}} \in \mathbb{R}^{r \times 2N}. \tag{2.87}$$

- **Option 2:** Augmenting the set with $2N$ new points more ($4N+1$ points in total)

$${}^{(aug)}\boldsymbol{\mathcal{X}}_k^- = \begin{bmatrix} {}^*\boldsymbol{\mathcal{X}}_k^- & \left({}^*\boldsymbol{\mathcal{X}}_k^- \right)_0 + \eta \mathbf{S}_Q^{\mathrm{T}} & \left({}^*\boldsymbol{\mathcal{X}}_k^- \right)_0 - \eta \mathbf{S}_Q^{\mathrm{T}} \end{bmatrix} \tag{2.88}$$

In this case, the weights must also be modified to the new size, by setting $N \rightarrow 2N$. The same equations as before apply now, with slight modifications in the indexation:

$${}^{(aug)}\boldsymbol{\mathcal{Y}}_k = H({}^{(aug)}\boldsymbol{\mathcal{X}}_k^-, \boldsymbol{u}_k) \tag{2.89}$$

$$\hat{\boldsymbol{y}}_k = \sum_{i=0}^{4N-1} \omega_i^{(m)} ({}^{(aug)}\boldsymbol{\mathcal{Y}}_k)_i \tag{2.90}$$

$$\mathbf{S}_{y_k} = \mathrm{qr}\left\{ \begin{bmatrix} \sqrt{\omega_1^{(c)}} \cdot \left(({}^{(aug)}\boldsymbol{\mathcal{Y}}_k^-)_{1:4N} - {}^{(aug)}\hat{\mathbf{Y}}_k \right) & \mathbf{S}_{\mathbf{R}}{}^{\mathrm{T}} \end{bmatrix}^{\mathrm{T}} \right\} \tag{2.91}$$

$$\mathbf{S}_{y_k} = \mathtt{cholupdate}\left\{ \mathbf{S}_{y_k}^-, \sqrt{\omega_0^{(c)}} \cdot \left((\boldsymbol{\mathcal{Y}}_k^-)_0 - {}^{(aug)}\hat{\mathbf{Y}}_k \right) \right\} \tag{2.92}$$

$$\mathbf{P}_{x_k y_k} = \sum_{i=0}^{4N-1} \omega_i^{(c)} \left(({}^{(aug)}\boldsymbol{\mathcal{X}}_k^-)_i - \hat{\boldsymbol{x}}_k^- \right) \left(({}^{(aug)}\boldsymbol{\mathcal{Y}}_k^-)_i - \hat{\boldsymbol{y}}_k \right)^{\mathrm{T}} \tag{2.93}$$

Having that

$${}^{(aug)}\hat{\mathbf{Y}}_k = \left[\hat{\boldsymbol{y}}_k, \ldots, \hat{\boldsymbol{y}}_k \right], \qquad \hat{\mathbf{Y}}_k \in \mathbb{R}^{r \times 4N}. \tag{2.94}$$

Eventually, the Kalman Gain is computed, turning it into an efficient least squares problem such that

$$\mathcal{K}_k = \mathbf{P}_{x_k y_k} \mathbf{P}_{y_k}^{-1} = \mathbf{P}_{x_k y_k} \left(\mathbf{S}_{y_k}^{\mathrm{T}} \mathbf{S}_{y_k} \right)^{-1} = \mathbf{P}_{x_k y_k} / \mathbf{S}_{y_k} / \mathbf{S}_{y_k}^{\mathrm{T}}, \tag{2.95}$$

which yields the *a posteriori* state estimate as

$$\hat{\boldsymbol{x}}_k^+ = \hat{\boldsymbol{x}}_k^- + \mathcal{K}_k (\boldsymbol{y}_k - \hat{\boldsymbol{y}}_k^-) \tag{2.96}$$

and finally the Cholesky factor of the covariance of the state as

$$\mathbf{D}_{\mathrm{C}} = \mathcal{K}_k \mathbf{S}_{y_k}^{\mathrm{T}} \tag{2.97}$$

$$\mathbf{S}_{x_k}^+ = \mathtt{cholupdate}\left(\mathbf{S}_{\boldsymbol{x}_k}^-, \boldsymbol{d}_i, '-' \right), \qquad i = 0, \ldots, r-1 \tag{2.98}$$

where $\boldsymbol{d}_i$ is the i^{th} column of $\mathbf{D}_{\mathrm{C}}$ and therefore a sequence of downdates is applied to $\mathbf{S}_{x_k}^-$, being the columns of $\mathbf{D}_{\mathrm{C}}$ the vectors for each downdate. This expression can be shown to be equivalent to that of the non-square-root Kalman filtering algorithms (Eq. (2.22)).

Proof: *Equivalence between SR-UKF and UKF. Measurement propagation of covariance.*

$$\mathbf{P}_{\boldsymbol{x}_k}^+ = \mathbf{P}_{\boldsymbol{x}_k}^- - \mathcal{K}_k \mathbf{P}_{\boldsymbol{y}_k} \mathcal{K}_k^{\mathrm{T}} = \mathbf{P}_{\boldsymbol{x}_k}^- - \mathbf{D}_{\mathrm{C}} \mathbf{D}_{\mathrm{C}}^{\mathrm{T}} = \mathbf{P}_{\boldsymbol{x}_k}^- - \sum_{i=0}^{r-1} \boldsymbol{d}_i \boldsymbol{d}_i^{\mathrm{T}} = \left(\mathbf{S}_{\boldsymbol{x}_k}^+ \right)^{\mathrm{T}} \mathbf{S}_{\boldsymbol{x}_k}^+ \tag{2.99}$$

Hence, according to Eq. (2.66),

$$\begin{aligned} \mathbf{S}_{\boldsymbol{x}_k}^+ &= \mathtt{chol}\left(\mathbf{P}_{\boldsymbol{x}_k}^+ \right) = \mathtt{chol}\left(\mathbf{P}_{\boldsymbol{x}_k}^- - \sum_{i=0}^{r-1} \boldsymbol{d}_i \boldsymbol{d}_i^{\mathrm{T}} \right) \\ &= \mathtt{cholupdate}\left(\mathbf{S}_{\boldsymbol{x}_k}^-, \boldsymbol{d}_i, '-' \right), \qquad i = 0, \ldots, r-1 \end{aligned} \tag{2.100}$$

It is therefore shown that the square-root version of the algorithm propagates the Cholesky factor of the covariance in the same way as the original algorithm propagates the covariance.

2.3.3 The Central Difference Kalman Filter

The Central Difference Kalman Filter (CDKF) is another alternative to the EKF for nonlinear state estimation. In opposition to the UKF, this one is not based on the Unscented Transformation, but on the Sterling's polynomial interpolation formula[9] which entails only one tuning parameter h, compared to the three (α, β, κ) of the previous algorithms [3].

[9] Only the 2nd order approximation will be employed.

Sterling's interpolation

The basic idea underlying Sterling's interpolation of nonlinear functions is based on a Taylor series approximation where the derivatives have been substituted by discrete differences.

The Taylor series expansion around the point $\bar{x}$ of the scalar function $f(x)$ is defined as follows:

$$f(x) = \sum_{n=0}^{\infty} \frac{1}{n!} \frac{d^n f(\bar{x})}{dx^n} (x-\bar{x})^n \approx f(\bar{x}) + \frac{df(\bar{x})}{dx}(x-\bar{x}) + \frac{1}{2!}\frac{d^2 f(\bar{x})}{dx^2}(x-\bar{x})^2 \tag{2.101}$$

Where the elements of order higher than 2 have been neglected.

The derivative of a nonlinear function $f(x)$ can be approximated by a central difference discrete expression, such as

$$\frac{df(\bar{x})}{dx} \approx \frac{f(\bar{x}+h) - f(\bar{x}-h)}{2h} \tag{2.102}$$

$$\frac{d^2 f(\bar{x})}{dx^2} \approx \frac{f(\bar{x}+h) + f(\bar{x}-h) - 2f(\bar{x})}{h^2} \tag{2.103}$$

Where h is the step size, a parameter to be chosen.

This expressions can be extended to vectorial nonlinear functions $\boldsymbol{f}(\boldsymbol{x})$ [3], enabling to compute the distribution of a random variable $\boldsymbol{y}$ such that

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) \tag{2.104}$$

out of the N-dimensional random variable $\boldsymbol{x}$ of known mean $\bar{\boldsymbol{x}}$ and covariance $\mathbf{P}_{\mathbf{x}}$. After operation, the mean of the new distribution happens to be

$$\bar{\boldsymbol{y}} \approx \frac{h^2 - N}{h^2}\boldsymbol{f}(\bar{\boldsymbol{x}}) + \frac{1}{2h^2}\sum_{i=1}^{N-1}\left[\boldsymbol{f}\left(\bar{\boldsymbol{x}} + h(\mathbf{S}_{\boldsymbol{x}}^{\mathrm{T}})_i\right) + \boldsymbol{f}\left(\bar{\boldsymbol{x}} - h(\mathbf{S}_{\boldsymbol{x}}^{\mathrm{T}})_i\right)\right]. \tag{2.105}$$

Where $(\mathbf{S}_{\boldsymbol{x}})_i$ is the i^{th} row of the upper Cholesky factor of the covariance of $\mathbf{P}_{\mathbf{x}}$, such that $\mathbf{P}_{\mathbf{x}} = \mathbf{S}_{\boldsymbol{x}}^{\mathrm{T}}\mathbf{S}_{\boldsymbol{x}}$. The covariance of the new distribution can be approximated by

$$\begin{aligned}\mathbf{P}_{\mathbf{y}} \approx{}& \frac{1}{h^4}\sum_{i=1}^{N-1}\left[\boldsymbol{f}\left(\bar{\boldsymbol{x}} + h(\mathbf{S}_{\boldsymbol{x}}^{\mathrm{T}})_i\right) - \boldsymbol{f}\left(\bar{\boldsymbol{x}} - h(\mathbf{S}_{\boldsymbol{x}}^{\mathrm{T}})_i\right)\right]\cdot\left[\boldsymbol{f}\left(\bar{\boldsymbol{x}} + h(\mathbf{S}_{\boldsymbol{x}}^{\mathrm{T}})_i\right) - \boldsymbol{f}\left(\bar{\boldsymbol{x}} - h(\mathbf{S}_{\boldsymbol{x}}^{\mathrm{T}})_i\right)\right]^{\mathrm{T}} \\ &+ \frac{h^2-1}{4h^4}\sum_{i=1}^{N-1}\left[\boldsymbol{f}\left(\bar{\boldsymbol{x}} + h(\mathbf{S}_{\boldsymbol{x}}^{\mathrm{T}})_i\right) + \boldsymbol{f}\left(\bar{\boldsymbol{x}} - h(\mathbf{S}_{\boldsymbol{x}}^{\mathrm{T}})_i\right) - 2\boldsymbol{f}(\bar{\boldsymbol{x}})\right] \\ &\qquad\cdot\left[\boldsymbol{f}\left(\bar{\boldsymbol{x}} + h(\mathbf{S}_{\boldsymbol{x}}^{\mathrm{T}})_i\right) + \boldsymbol{f}\left(\bar{\boldsymbol{x}} - h(\mathbf{S}_{\boldsymbol{x}}^{\mathrm{T}})_i\right) - 2\boldsymbol{f}(\bar{\boldsymbol{x}})\right]^{\mathrm{T}}\end{aligned} \tag{2.106}$$

and the cross-covariance as follows:

$$\mathbf{P}_{\mathbf{xy}} \approx \frac{1}{2h}\sum_{i=1}^{N-1}(\mathbf{S}_{\boldsymbol{x}}^{\mathrm{T}})_i\left[\boldsymbol{f}\left(\bar{\boldsymbol{x}} + h(\mathbf{S}_{\boldsymbol{x}}^{\mathrm{T}})_i\right) - \boldsymbol{f}\left(\bar{\boldsymbol{x}} - h(\mathbf{S}_{\boldsymbol{x}}^{\mathrm{T}})_i\right)\right]^{\mathrm{T}} \tag{2.107}$$

These expressions of Stirling's Interpolation are now modified to fit into the Sigma-Point filtering scheme, giving a recurrent algorithm with the usual three sections of *Initialization*, *Time-update* and *Measurement-update*.

Initialization

The filter is initialized by setting a value for the mean and covariance of the states, as in the previous algorithms:

$$\hat{\boldsymbol{x}}_0^+ = E[\boldsymbol{x}_0] \tag{2.108}$$

$$\mathbf{P}_0^+ = E[(\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0^+)(\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0^+)^{\mathrm{T}}] \tag{2.109}$$

Time-update

Afterwards, a set of sigma-points must be selected to represent the given distribution; this time h is the only parameter setting their spread[10]

$$\boldsymbol{\mathcal{X}}_{k-1}^+ = \left[\begin{array}{ccc} \hat{\boldsymbol{x}}_{k-1}^+ & \hat{\mathbf{X}}_{k-1}^+ + h\sqrt{\mathbf{P}_{x_{k-1}}^+} & \hat{\mathbf{X}}_{k-1}^+ - h\sqrt{\mathbf{P}_{x_{k-1}}^+} \end{array} \right] \tag{2.110}$$

The set of sigma-points must be now passed through the nonlinear function $F(\cdot)$:

$${}^*\boldsymbol{\mathcal{X}}_k^- = F(\boldsymbol{\mathcal{X}}_{k-1}^+, \boldsymbol{u}_{k-1}) \tag{2.111}$$

The Equations (2.105) and (2.106) are adapted to compute the mean and covariance of the states' distribution on the next step:

$$\hat{\boldsymbol{x}}_k^- = \sum_{i=0}^{2N-1} \omega_i^{(m)} ({}^*\boldsymbol{\mathcal{X}}_k^-)_i \tag{2.112}$$

$$\begin{aligned} \mathbf{P}_{\boldsymbol{x}_k}^- = \sum_{i=1}^{N-1} \Big\{ & \omega_i^{(c1)} \left[({}^*\boldsymbol{\mathcal{X}}_k^-)_i - ({}^*\boldsymbol{\mathcal{X}}_k^-)_{N+i} \right] \cdot \left[({}^*\boldsymbol{\mathcal{X}}_k^-)_i - ({}^*\boldsymbol{\mathcal{X}}_k^-)_{N+i} \right]^{\mathrm{T}} \\ & + \omega_i^{(c2)} \left[({}^*\boldsymbol{\mathcal{X}}_k^-)_i + ({}^*\boldsymbol{\mathcal{X}}_k^-)_{N+i} - 2\left({}^*\boldsymbol{\mathcal{X}}_k^-\right)_0 \right] \cdot \left[({}^*\boldsymbol{\mathcal{X}}_k^-)_i + ({}^*\boldsymbol{\mathcal{X}}_k^-)_{N+i} - 2\left({}^*\boldsymbol{\mathcal{X}}_k^-\right)_0 \right]^{\mathrm{T}} \Big\} + \mathbf{Q} \end{aligned} \tag{2.113}$$

Where the weights are also adapted to satisfy the equations[11]:

$$\omega_0^{(m)} = \frac{h^2 - N}{h^2} \tag{2.114}$$

$$\omega_i^{(m)} = \frac{1}{2h^2} \qquad i = 1,\ldots,2N \tag{2.115}$$

$$\omega_i^{(c1)} = \frac{1}{4h^2} \qquad i = 1,\ldots,2N \tag{2.116}$$

$$\omega_i^{(c2)} = \frac{h^2 - 1}{4h^4} \qquad i = 1,\ldots,2N \tag{2.117}$$

[10] For Gaussian noise distributions $h = \sqrt{3}$ is the optimal choice [3].

[11] Recall that the equations do not require the definition of $\omega_0^{(c1)}$ and $\omega_0^{(c2)}$ because Eq. (2.106) does not present a summand depending on the mean of the distribution $\boldsymbol{f}(\bar{\boldsymbol{x}})$, the zero-th sigma-point. This avoids the previously mentioned problems with the negative weighting coefficient of the SR-UKF, which forced the use of the Cholesky factor update after the QR-decomposition.

Measurement-update

A new set of sigma points can be computed to represent the distirbution of the *a priori* state estimate $\hat{\boldsymbol{x}}_k^-$, but introducing the effects of the new covariance:

$$\boldsymbol{\mathcal{X}}_k^- = \left[\begin{array}{ccc} \hat{\boldsymbol{x}}_k^- & \hat{\mathbf{X}}_k^- + h\sqrt{\mathbf{P}_{x_k}^-} & \hat{\mathbf{X}}_k^- - h\sqrt{\mathbf{P}_{x_k}^-} \end{array} \right] \tag{2.118}$$

Once again, the mean and covariance of the output distribution can be estimated:

$$\boldsymbol{\mathcal{Y}}_k = H(\boldsymbol{\mathcal{X}}_k^-, \boldsymbol{u}_k) \tag{2.119}$$

$$\hat{\boldsymbol{y}}_k = \sum_{i=0}^{2N-1} \omega_i^{(m)} (\boldsymbol{\mathcal{Y}}_k)_i \tag{2.120}$$

$$\begin{aligned} \mathbf{P}_{\boldsymbol{y}_k}^- = \sum_{i=1}^{N-1} \Big\{ & \omega_i^{(c1)} [(\boldsymbol{\mathcal{Y}}_k)_i - (\boldsymbol{\mathcal{Y}}_k)_{N+i}] \cdot [(\boldsymbol{\mathcal{Y}}_k)_i - (\boldsymbol{\mathcal{Y}}_k)_{N+i}]^{\mathrm{T}} \\ & + \omega_i^{(c2)} \left[(\boldsymbol{\mathcal{Y}}_k)_i + (\boldsymbol{\mathcal{Y}}_k)_{N+i} - 2(\boldsymbol{\mathcal{Y}}_k)_0 \right] \cdot \left[(\boldsymbol{\mathcal{Y}}_k)_i + (\boldsymbol{\mathcal{Y}}_k)_{N+i} - 2(\boldsymbol{\mathcal{Y}}_k)_0 \right]^{\mathrm{T}} \Big\} + \mathbf{R} \end{aligned} \tag{2.121}$$

And the Eq. (2.107) can now be adapted for the estimation of the cross-covariance:

$$\mathbf{P}_{\boldsymbol{x}_k \boldsymbol{y}_k} = \sqrt{\omega_1^{(c1)} \mathbf{P}_{\boldsymbol{x}_k}^-} [\boldsymbol{\mathcal{Y}}_{1:N-1} - \boldsymbol{\mathcal{Y}}_{N+1:2N-1}]^{\mathrm{T}} \tag{2.122}$$

The last steps, as usual, are the calculation of the Kalman gain and the corresponding estimations:

$$\boldsymbol{\mathcal{K}}_k = \mathbf{P}_{\boldsymbol{x}_k \boldsymbol{y}_k} \mathbf{P}_{\boldsymbol{y}_k}^{-1} \tag{2.123}$$

$$\hat{\boldsymbol{x}}_k^+ = \hat{\boldsymbol{x}}_k^- + \boldsymbol{\mathcal{K}}_k (\boldsymbol{y}_k - \hat{\boldsymbol{y}}_k^-) \tag{2.124}$$

$$\mathbf{P}_{\boldsymbol{x}_k}^+ = \mathbf{P}_{\boldsymbol{x}_k}^- - \boldsymbol{\mathcal{K}}_k \mathbf{P}_{\boldsymbol{y}_k} \boldsymbol{\mathcal{K}}_k^{\mathrm{T}} \tag{2.125}$$

2.3.4 The Square-Root Central Difference Kalman Filter

As in the case of the UKF, the CDKF can also be modified to make its implementation computationally more efficient, through the Square-Root Central Difference Kalman Filter (SR-CDKF). In order to do so, the costliest operation (the computation of the square-root covariance matrix for every sigma-point set) is avoided by making the whole algorithm work directly with the Cholesky factors of the covariance matrices. Besides, some of the steps are substituted by equivalent algebraic operations such as the Cholesky factor update, the QR-decomposition and the efficient least squares (see Section 2.3.2 for a detailed explanation).

Once again, the algorithm will be divided in three parts: *Initialization*, *Time-update* and *Measurement-update*.

Initialization

The filter gets initialized as the SR-UKF, with the upper Cholesky factor of the covariance matrices

$$\hat{\boldsymbol{x}}_0^+ = E[\boldsymbol{x}_0] \tag{2.126}$$

$$\mathbf{S}_{x_0}^+ = \mathrm{chol}\{E[(\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0^+)(\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0^+)^{\mathrm{T}}]\} \tag{2.127}$$

$$\mathbf{S}_{\mathbf{Q}} = \left(\sqrt{\mathbf{Q}}\right)^{\mathrm{T}} = \mathrm{chol}(\mathbf{Q}) \tag{2.128}$$

$$\mathbf{S}_{\mathbf{R}} = \left(\sqrt{\mathbf{R}}\right)^{\mathrm{T}} = \mathrm{chol}(\mathbf{R}). \tag{2.129}$$

Time-update

There are only slight variations in the first steps with respect to those of the CDKF

$$\hat{\mathbf{X}}_{k-1}^+ = \left[\hat{\boldsymbol{x}}_{k-1}^+, \ldots, \hat{\boldsymbol{x}}_{k-1}^+\right], \qquad \hat{\mathbf{X}}_{k-1}^+ \in \mathbb{R}^{N\times N} \tag{2.130}$$

$$\boldsymbol{\mathcal{X}}_{k-1}^+ = \left[\begin{array}{ccc} \hat{\boldsymbol{x}}_{k-1}^+ & \hat{\mathbf{X}}_{k-1}^+ + h\left(\mathbf{S}_{x_{k-1}}^+\right)^{\mathrm{T}} & \hat{\mathbf{X}}_{k-1}^+ - h\left(\mathbf{S}_{x_{k-1}}^+\right)^{\mathrm{T}} \end{array}\right] \tag{2.131}$$

$${}^*\boldsymbol{\mathcal{X}}_k^- = F(\boldsymbol{\mathcal{X}}_{k-1}^+, \boldsymbol{u}_{k-1}) \tag{2.132}$$

$$\hat{\boldsymbol{x}}_k^- = \sum_{i=0}^{2N-1} \omega_i^{(m)} ({}^*\boldsymbol{\mathcal{X}}_k^-)_i \tag{2.133}$$

Where the definition of $\omega^{(m)}$, $\omega^{(c1)}$ and $\omega^{(c2)}$ is the one given in Eq. (2.114) to (2.117). The computation of the Cholesky factor of the covariance matrix, requires modifying Eq. (2.113) to put it as a QR-decomposition, in which the upper Cholesky factor $\mathbf{S}_{x_k}^-$ is returned

$$\begin{aligned} \mathbf{S}_{x_k}^- = \mathrm{qr}\Bigg\{\Bigg[& \sqrt{\omega_1^{(c1)}}\left(({}^*\boldsymbol{\mathcal{X}}_k^-)_{1:N-1} - ({}^*\boldsymbol{\mathcal{X}}_k^-)_{N+1:2N-1}\right), \\ & \sqrt{\omega_1^{(c2)}}\left(({}^*\boldsymbol{\mathcal{X}}_k^-)_{1:N-1} + ({}^*\boldsymbol{\mathcal{X}}_k^-)_{N+1:2N-1} - 2\left[({}^*\boldsymbol{\mathcal{X}}_k^-)_0 \overset{\times N-1}{\cdots\cdots}\right]\right), \mathbf{S}_{\mathbf{Q}}{}^{\mathrm{T}}\Bigg]^{\mathrm{T}}\Bigg\} \end{aligned} \tag{2.134}$$

This expression requires to concatenate $N-1$ times the zero-th column of the sigma-point matrix.

$$\left[({}^*\boldsymbol{\mathcal{X}}_k^-)_0 \overset{\times N-1}{\cdots\cdots}\right] = \left[({}^*\boldsymbol{\mathcal{X}}_k^-)_0, \ldots, ({}^*\boldsymbol{\mathcal{X}}_k^-)_0\right], \qquad \left[({}^*\boldsymbol{\mathcal{X}}_k^-)_0 \overset{\times N-1}{\cdots\cdots}\right] \in \mathbb{R}^{N\times N-1} \tag{2.135}$$

Measurement-update

Similarly to the CDKF[12],

$$\hat{\mathbf{X}}_k^- = \left[\hat{\boldsymbol{x}}_k^-, \ldots, \hat{\boldsymbol{x}}_k^-\right], \qquad \hat{\mathbf{X}}_k^- \in \mathbb{R}^{N\times N} \tag{2.136}$$

$$\boldsymbol{\mathcal{X}}_k^- = \left[\begin{array}{ccc} \hat{\boldsymbol{x}}_k^- & \hat{\mathbf{X}}_k^- + h\left(\mathbf{S}_{x_k}^-\right)^{\mathrm{T}} & \hat{\mathbf{X}}_k^- - h\left(\mathbf{S}_{x_k}^-\right)^{\mathrm{T}} \end{array}\right] \tag{2.137}$$

$$\boldsymbol{\mathcal{Y}}_k = H(\boldsymbol{\mathcal{X}}_k^-, \boldsymbol{u}_k) \tag{2.138}$$

$$\hat{\boldsymbol{y}}_k = \sum_{i=0}^{2N-1} \omega_i^{(m)} (\boldsymbol{\mathcal{Y}}_k)_i \tag{2.139}$$

$$\begin{aligned} \mathbf{S}_{y_k} = \mathrm{qr}\Bigg\{\Bigg[& \sqrt{\omega_1^{(c1)}}\left((\boldsymbol{\mathcal{Y}}_k)_{1:N-1} - (\boldsymbol{\mathcal{Y}}_k)_{N+1:2N-1}\right), \\ & \sqrt{\omega_1^{(c2)}}\left((\boldsymbol{\mathcal{Y}}_k)_{1:N-1} + (\boldsymbol{\mathcal{Y}}_k)_{N+1:2N-1} - 2\left[(\boldsymbol{\mathcal{Y}}_k)_0 \overset{\times N-1}{\cdots\cdots}\right]\right), \mathbf{S}_{\mathbf{R}}^{\mathrm{T}}\Bigg]^{\mathrm{T}}\Bigg\} \end{aligned} \tag{2.140}$$

With

$$\left[(\boldsymbol{\mathcal{Y}}_k)_0 \overset{\times N-1}{\cdots\cdots}\right] = \left[(\boldsymbol{\mathcal{Y}}_k)_0, \ldots, (\boldsymbol{\mathcal{Y}}_k)_0\right], \qquad \left[(\boldsymbol{\mathcal{Y}}_k)_0 \overset{\times N-1}{\cdots\cdots}\right] \in \mathbb{R}^{r\times N-1} \tag{2.141}$$

$$\mathbf{P}_{x_k y_k} = \sqrt{\omega_1^{(c1)}}\mathbf{S}_{x_k}^{-\mathrm{T}}\left[\boldsymbol{\mathcal{Y}}_{1:N-1} - \boldsymbol{\mathcal{Y}}_{N+1:2N-1}\right]^{\mathrm{T}} \tag{2.142}$$

$$\boldsymbol{\mathcal{K}}_k = \left(\mathbf{P}_{\boldsymbol{x}_k \boldsymbol{y}_k} / \mathbf{S}_{y_k}\right) / \mathbf{S}_{y_k}^{\mathrm{T}} \tag{2.143}$$

$$\hat{\boldsymbol{x}}_k^+ = \hat{\boldsymbol{x}}_k^- + \boldsymbol{\mathcal{K}}_k(\boldsymbol{y}_k - \hat{\boldsymbol{y}}_k^-) \tag{2.144}$$

$$\mathbf{D}_{\mathrm{C}} = \boldsymbol{\mathcal{K}}_k \mathbf{S}_{y_k}^{\mathrm{T}} \tag{2.145}$$

$$\mathbf{S}_{x_k}^+ = \mathtt{cholupdate}\left(\mathbf{S}_{\boldsymbol{x}_k}^-, \boldsymbol{d}_i, '-'\right), \qquad i = 0, \ldots, r-1 \tag{2.146}$$

where $\boldsymbol{d}_i$ is the i^{th} column of $\mathbf{D}_{\mathrm{C}}$ and therefore a sequence of downdates is applied to $\mathbf{S}_{x_k}^-$, being the columns of $\mathbf{D}_{\mathrm{C}}$ the vectors for each downdate.

[12] Eq. (2.140) corrects what, through logic and experimentation, was considered as a typo in the original bibliography, from which the algorithm was extracted ([3], (3.233))

3 Implementation Possibilities

Once the problem is clarified and the theoretical tools required for its solution have been considered, the next step is to analyze the different technical means of practically implementing those solutions. This chapter will explain the available utilities as well as the reasoning followed until the final choice.

The two compared options are writing the algorithms in C code or in MATLAB-Code. In any of the cases the illustrative results are obtained from SIMULINK simulations.

3.1 C code into SIMULINK

The first approach to the problem has been the implementation of the algorithms in C code. The main benefits of this choice with respect to the implementation in MATLAB-Code are that

- a better control of the process is achieved, enabling a higher efficiency and speed of the algorithm, making it appropriate for real-time applications
- it makes it easy embedding the code into microcontrollers through slight variations of the functions.

In other words, the main goal of developing filtering algorithms is their further implementation in real state estimation problems, which requires having a fast and efficient code running in real-time on a microcontroller. This is theoretically easier to achieve by means of C code.

However, as before specified, the aim is to test the different algorithms in SIMULINK, not in a real microcontroller, and we should therefore consider the ways of embedding C code into SIMULINK, most of the ways imply the use of the so called *S-Functions* [11].

3.1.1 S-Functions

The S-Functions (System Functions) are a mechanism for extending the capabilities of SIMULINK; enabling the user to define his own blocks by means of a computer language such as MATLAB, C, C++ or Fortran [12]. Specifically, the S-Functions written in C (also known as C MEX S-Functions) are compiled by an external compiler which generates a MEX file (*Matlab Executable*) that is called during the simulation.

The code of the S-Function from which the MEX file is obtained does not only have to include the C-functions, but it must define all the parameters of the SIMULINK block, as well as the way in which it interacts with the program during the simulation. There are several ways of generating

this code out of an arbitrary C code written algorithm: the *Legacy Code Tool*, the *S-Function Builder* and the *Handwritten S-Functions*.

- **Legacy Code Tool –** The *Legacy Code Tool* is a set of MATLAB commands, which makes the process of creating S-Functions from a given C code easier. This tool receives the header and source files as inputs, as well as some of the SIMULINK block parameters (sample time, name of the S-Function...) and it generates a C MEX S-Function which can be run inside a SIMULINK model[1].

  Even if the *Legacy Code Tool* constitutes the easiest approach for the creation of C MEX S-Functions out of existing C code, it is also the one which fewest features supports [12].

- **S-Function Builder –** The *S-Function Builder* is another tool which eases the writing of S-Functions from existing C code. It is a graphical user interface, in which the headers, code and other required specifications are treated once again as inputs, giving the MEX file as an output[2].

  This procedure requires a better knowledge of the working mechanism of S-Functions and it supports more features than the previous one, it is still limited though. For instance, it can neither build S-functions that have more than one input or output, nor it can handle data types other than double [13].

- **Handwritten S-Functions –** This last option implies writing the S-Function starting from scratch. There are some templates provided by THE MATHWORKS in which the typical structure and commands are shown[3].

  This option is the most flexible one. It enables modifying every single line of the S-Function code, giving a huge control of the program. However, it is obviously the most complicated way too, since it requires a good knowledge of the working mechanism of S-Functions.

In general, it is possible to state that S-Functions are an efficient way of testing an algorithm which was written in C, within a SIMULINK model. The tool required to embed the code into SIMULINK depends on the complexity of the task to accomplish and the knowledge of the programmer.

3.1.2 The CSparse/CXSparse Library

MATLAB is a language specifically designed for easing the programming of high level algorithms but that is not the case for the C language. Even if it has the benefit of being closer to the

[1] For a detailed explanation see `http://de.mathworks.com/help/simulink/sfg/integrating-existing-c-functions-into-simulink-models-with-the-legacy-code-tool.html`

[2] For a detailed explanation see `http://de.mathworks.com/help/simulink/sfg/s-function-builder-dialog-box.html`

[3] See, for instance `http://de.mathworks.com/help/simulink/sfg/templates-for-c-s-functions.html`

hardware (which enables the desired quickness and efficiency) it is not desirable to program all the linear algebra routines starting from scratch. Thus, one should employ existing libraries.

Looking at the characteristics of the algorithms presented in chapter 2, it is possible to see that the most basic operations needed are the manipulation of vectors and matrices (multiplication, addition...) and some more complex operations such as the Cholesky Factorization for the Sigma-Point implementation.

Besides, the matrices employed in those algorithms for the analyzed wind turbine system can be considered *sparse*; meaning that they have a lot of zero elements.

One library satisfying the requirements is the CSparse library developed by Timothy A. Davis [14]. For its use with 64-bit MATLAB the super-set library CXSparse is required which can also operate with complex matrices.

This library which is included in the code by the source file "`cs.c`" and the header file "`cs.h`" defines the sparse matrices as data structures which can be either in *compressed-column* or in *triplet* format. These data structures only store the non-zero elements of the matrix, as well as a pattern indicating their locations. The formats (*compressed-column* and *triplet*) are required for the different commands defined in the library. Each command must receive the matrices in a given form.

3.2 MATLAB Code into SIMULINK

The algorithms can also be tested by directly writing them in MATLAB code. The main advantage of this procedure is the easiness. The MATLAB language is thought for the efficient implementation of the required functions (Cholesky Factor Update,...) and it is also easier to embed MATLAB code in SIMULINK than C code.

Moreover, the MATLAB CODER software enables the generation of C code from MATLAB code under certain restrictions. When they are satisfied, this is a suitable way of implementing the developed and tested algorithms into a microcontroller for real-life use. Even if the control achieved by directly writing C code is lost, a lot of easiness and understandability of the code is gained and the efficient implementation of the algorithms is also guaranteed.

There are several ways of embedding the MATLAB code into SIMULINK. Some of them are the MATLAB *code S-Functions*, *Interpreted Matlab Functions* and the *Embedded Matlab Functions*.

- **MATLAB S-Functions –** These functions are equivalent to the C Code S-Functions. They are written in MATLAB Code though. Their working mechanism is very similar and they allow for multiple inputs, outputs and various data types. They are also available templates, where the different required commands are specified and they are suitable for direct code generation.
- **Interpreted MATLAB Functions –** This tool enables the easy inclusion of MATLAB code into a SIMULINK model. There is a wide set of functions accepted by this block, but the block is

restricted to a single output and a single input of type `double` which can be either real or complex. The interpreted functions enable the call to functions on the current directory of MATLAB. However, the SIMULINK CODER does not support the C code generation from these functions, which is a considerable disadvantage.

- **Embedded MATLAB Functions –** These functions (in versions of the software after R2011a simply known as "MATLAB *Function Block*") have the main benefit of allowing the readable, efficient and compact C/C++ code generation [12]. They however do not accept all the MATLAB functions[4]. Besides, everything must be defined inside the function, as they do not accept loading data from workspace or using functions defined in the search path of MATLAB. Nevertheless, persistent variables are an efficient technique to define data at the initialization and use it while simulating.

3.2.1 The MATLAB EKF/UKF Toolbox

The MATLAB *EKF/UKF Toolbox* is composed by a set of functions developed by Jouni Hartikainen *et al.* for the implementation in MATLAB of Kalman Filtering and Smoothing algorithms such as the EKF, UKF or CKF, among others.

All the different algorithms of the toolbox are designed with a similar structure and the program is usually divided in two main functions: *predict*, equivalent to the *Time-update* step and *update*, equivalent to the *Measurement-update*. A very detailed description can be found in [8].

[4] Just the ones listed in `http://de.mathworks.com/help/simulink/ug/functions-supported-for-code-generation-alphabetical-list.html`

4 Development of the Code

Now that the problem, the theory to solve it and the practical tools available are clear, it is time to analyze the evolution of the project. This chapter aims to describe the procedure followed until the complete fulfillment of the purposed task.

4.1 Kalman Filter in C Code

Due to the above exposed reasons, the first approach to the problem consisted on writing the algorithms in C-Code, using the CSparse (CXSparse) library. The code is brought into SIMULINK through a handwritten S-Function.

At the beginning, a simple discrete-time SISO model was employed to become familiar with the library and the working mechanism of S-Functions. This model was afterwards extended to a two output model, eventually augmenting it to the linearized model of a wind turbine presented in Appendix A.

The first implemented algorithm is the *Linear Kalman Filter* (see Section 2.2), giving an accurate result for the simplified model. The code was therefore extended for every system of N states, M inputs and r outputs[1], as can be seen in its final version in Appendix B, Listing B.1. The performance of the algorithm with the wind turbine model will be analyzed in Chapter 5. The following pages will analyze how the code was implemented and the commands employed.

4.1.1 Structure of the Program: S-Function

The code was constructed based on the templates that THE MATHWORKS make available for the developing of C MEX S-Functions, which are *sfuntmpl_basic.c* and *sfuntmpl_doc.c*, being the first one a basic template and the second one a exhaustive one, with all available routines. The basic behavior of every S-Function during simulation is depicted in Figure 4.1.

The S-Functions are always initialized with a couple of lines defining the name of the function and its type (lines 9, 10):

```
#define S_FUNCTION_NAME  turKFsparse
#define S_FUNCTION_LEVEL 2
```

and they include the `simstruc` library (line 12)

[1] Some slight variations of the code must be done when changing the number of parameters of the model, they are specified as comments

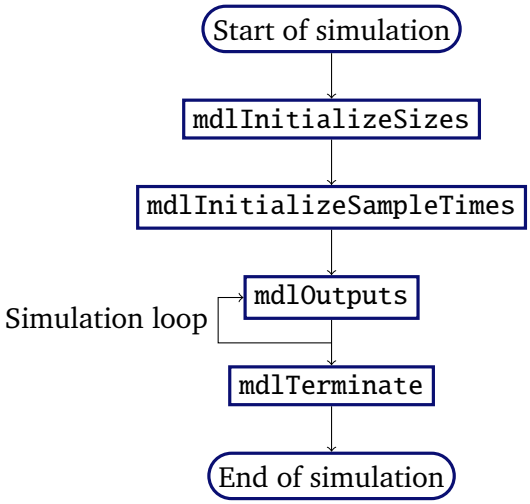

Figure 4.1.: Behavior of basic routines of S-Functions during simulation (adapted from [13]).

```
#include "simstruc.h"
```

which has all the routines for the compilation and appropriate usage of the code. They are structured with the four indispensable commands, that make the skeleton of every C MEX S-Function:

- **mdlInitializeSizes** – Specifies sizes of parameters, number and size of input and output ports of the block and some specifications, such as enabling the block to have direct feedthrough or not. In the here analyzed code, for instance, it is possible to find expressions such as (lines 38 and 39)

  ```
  ssSetInputPortWidth(S, 0, 4);
  ssSetInputPortDirectFeedThrough(S, 0, 1);
  ```

  Which mean that the input in the port number "0" has a size of 4 elements and it enables direct feedthrough; in other words, the current output can depend on the current value of the input.

- **mdlInitializeSampleTimes** – This routine specifies the sample time of the block, as can be seen in lines 63 and 64,

  ```
  ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
  ssSetOffsetTime(S, 0, 0.0);
  ```

  where the sample time is specified as the same of the simulation with no offset.

- **mdlOutputs** – This function is where the actual algorithm is; the code is repeated each simulation step (see Figure 4.1) and the outputs are computed every time. The main code under this function will be analyzed in more detail in Section 4.1.2 as it has more to do with the algorithm itself, it is however interesting to explain commands such as (line 95)

  ```
  InputRealPtrsType   uk = ssGetInputPortRealSignalPtrs(S,0);
  ```

  which every sample time stores in the variable uk the new value of the input or (line 119)

```
real_T            *A = mxGetPr(PARAM_matrixA);
```

where `PARAM_matrixA` was defined in the preamble (line 17) as

```
#define PARAM_matrixA ssGetSFcnParam(S,0)
```

This code gets run only the first time because the matrices are constant. It enables to get the matrix as a parameter from the SIMULINK block (and hence from the workspace of MATLAB).

- **`mdlTerminate`** – This routine is run at the end of the simulation. It is mandatory to put it for the good operation of the block, it is however totally empty in the code, as it is not required to do anything at the end of the simulation.

Apart from the here depicted general S-Function structure, the Listing B.1 also has a preamble of "`#define`"-s and "`#include`"-s, characteristic of the C coded programs (lines 9 to 23) and there is a self written function at the end of the code (`makesparse`).

4.1.2 Implementation of the Algorithm: CXSparse

Once the skeleton of the function and its behavior during simulation is clear, it is time to focus on the implementation of the equations from Section 2.2.

The CXSparse library is included in line 13 through the header file `cs.h` which defines all the functions from the library and must have a source file (`cs.c`) where the code of those functions is detailed (see Section 4.1.3 for more information on the topic).

At the beginning of the simulation, all the employed sparse matrices and vectors must be defined (lines 73-85). The rest of elements can be defined inside the `if(key){}` environment, defining them only in the first run step of the code.

After inputting the matrices as parameters, the following operation is conducted for all of them:

```
F = cs_spalloc(N, N, N*N, 1, 1);          //allocating space in memory
makesparse(A,F,N,N);                      //turning the matrix sparse
F=cs_compress(F);                         //turning it from triplet to CC
```

where `cs_spalloc` and `cs_compress` are two commands defined on the CXSparse library and `makesparse` is a self written command. The overall translation would be the following:

> *Allocate a sparse $N \times N$ matrix* $\mathbf{F}$ *in triplet format, with a maximum amount of* N^2 *non-zero entries, put all the elements of matrix* $\mathbf{A}$ *into the matrix* $\mathbf{F}$ *and turn it into compressed column format.*

This needs to be done because the `cs_entry` command requires a matrix in triplet format, whereas the following commands require the matrix in compressed column (CC) format.

Afterwards, the Kalman Filter equations (2.11) to (2.23) are implemented using the commands available (Table 4.1).

Table 4.1.: Overview of the employed commands of the CXSparse library (adapted from [14])

Command	Description
C=cs_add(**A**,**B**,α,β)	Adds two sparse CC matrices: $\mathbf{C} = \alpha\mathbf{A} + \beta\mathbf{B}$
C=cs_multiply(**A**,**B**)	Multiplies two sparse CC matrices: $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$
C=cs_transpose(**A**,1)	Transposes a sparse matrix: $\mathbf{C} = \mathbf{A}^{\mathrm{T}}$
cs_lusol(0, **A**, $\boldsymbol{b}$, 1)	Solves $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ using LU factorization stores answer in $\boldsymbol{b}$
cs_gaxpy(**A**,$\boldsymbol{x}$,$\boldsymbol{y}$)	Computes sparse CC matrix times dense column vector: $\mathbf{y} = \mathbf{A}\boldsymbol{x} + \boldsymbol{y}$
cs_spalloc(a,b,c,1,1)	Allocates an a×b matrix with a maximum of c nonzero entries in CC form
cs_spfree(**A**)	Frees a sparse matrix
cs_entry(**A**,i,j,a)	Adds entry "a" in position (i,j) to triplet matrix **A**
cs_compress(**A**)	Converts triplet-form matrix **A** into CC

In order to compute the Kalman gain from Eq. (2.23), the computation of an inverse matrix is required. Since this is not one of the functionalities of the package, it is useful to transform the system into a conventional linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$.

$$\mathcal{K}_k = \mathbf{P}^{-}_{x_k}\mathbf{C}^{\mathrm{T}}(\mathbf{C}\mathbf{P}^{-}_{x_k}\mathbf{C}^{\mathrm{T}} + \mathbf{R})^{-1} = \mathbf{P}_{x_k y_k}\mathbf{P}^{-1}_{y_k} = \left[\left(\mathbf{P}^{-1}_{y_k}\right)^{\mathrm{T}}\mathbf{P}^{\mathrm{T}}_{x_k y_k}\right]^{\mathrm{T}} = \left[\mathbf{S}^{-1}_{aux}\mathbf{B}_{aux}\right]^{\mathrm{T}} \tag{4.1}$$

$$\mathbf{S}_{aux} = \mathbf{C}\left(\mathbf{P}^{-}_{x_k}\right)^{\mathrm{T}}\mathbf{C}^{\mathrm{T}} + \mathbf{R}^{\mathrm{T}} \tag{4.2}$$

$$\mathbf{B}_{aux} = \mathbf{C}(\mathbf{P}^{-}_{x_k})^{\mathrm{T}} \tag{4.3}$$

It is remarkable though that it is not a vector $\boldsymbol{x}$ what we are looking for, but a matrix $\mathcal{K}^{\mathrm{T}}_k \in \mathbb{R}^{r\times N}$, so the solution adopted for solving a system $\mathbf{S}_{aux}\mathcal{K}^{\mathrm{T}}_k = \mathbf{B}_{aux}$ was to split it into N different systems, one per each of the N columns of $\mathcal{K}^{\mathrm{T}}_k$ (lines 197-214).

$$\left.\begin{aligned} \mathbf{S}_{\mathrm{aux}}\boldsymbol{k}_0 &= \boldsymbol{b}_0 \\ \mathbf{S}_{\mathrm{aux}}\boldsymbol{k}_1 &= \boldsymbol{b}_1 \\ \vdots \quad &\quad \vdots \\ \mathbf{S}_{\mathrm{aux}}\boldsymbol{k}_{N-1} &= \boldsymbol{b}_{N-1} \end{aligned}\right\} \mathcal{K}_k = \left[\boldsymbol{k}_0, \quad \boldsymbol{k}_1, \ldots, \boldsymbol{k}_{N-a}\right]^{\mathrm{T}} \tag{4.4}$$

Finally, the estimated state is obtained by (lines 234-236):

```
for (i=0; i<width; i++) {
  *xest1++ =x_est->x[i];                    //Show the result
}
```

The elements inside the cs data structures are accessed through the "->" operator. In this extract of the code, the x vector of the cs data structure x_est is called. The *sparse* data structures are

defined by a set of vectors and constants. Vector `x` contains all the nonzero entries of the matrix in a single column. The position of these elements in the matrix is defined in other vectors of the data structure.

Eventually, the code of the self written function `makesparse` is given, as well as a set of commands required for the appropriate compilation of the code. The `makesparse` function (lines 250 to 259) has the task of transforming the matrices received from MATLAB into sparse *triplet* format matrices. It is defined as a separate routine because it is repeatedly called throughout the code.

```
static void makesparse (double *value_array, cs *sparsematrix, int rows, int →
    ←columns ){
    int j,k,p=0;
    for (k=0; k<columns; ++k){
    for (j=0; j<rows; ++j) {
        cs_entry(sparsematrix, j, k, value_array[p+j]);
    }
    p=p+j;
  }
  p=0;
}
```

4.1.3 Introducing the Code into SIMULINK

The first step for creating a C MEX S-Function from the handwritten C code is to compile the file. Thus, a C compiler must be installed in MATLAB[2], by

```
mex -setup
```

command.

Furthermore, the required library must be created from the source file `cs.c`, which specifies all employed functions.

When the compiler is run it creates object files (`.obj`) from the source files (`.c`). This is way the source file of the library can be compiled into an object file and create a static library (`.lib`) afterwards. After creation of the object files and libraries, the linker searches on the specified libraries for the definition of all the employed functions. Thereby, the code where all the functions of CXSparse are defined is compiled only once, creating a library. This library is used every time that the code of the S-Function is compiled.

The static library is created by putting all the functions together into the `cs.c` source file and running the following commands in that directory from the *Microsoft Windows 7 x64 Debug Build Environment*:

[2] From the ones listed in `http://de.mathworks.com/support/sysreq/files/SystemRequirements-Release2013b_SupportedCompilers.pdf?sec=win64`, e.g. the *Microsoft Windows SDK 7.1*

```
cl -c cs.c
LIB cs.obj
```

Afterwards, once the compiler is set up correctly in MATLAB, the MEX file is created by executing the following command in the MATLAB command window:

```
mex -DNCOMPLEX turKFsparse.c cs.lib ;
```

`turKFsparse.c` is the name of the source file of the S-Function and `-DNCOMPLEX` is a flag of the CXSparse library. It is set for the compilation to avoid compatibility problems with the compiler and MATLAB.

4.1.4 Overview of the Implementation

Once the development process of the C MEX S-Function Linear Kalman Filter is finished, its estimation of the state is satisfactory and comparable to that of the other algorithms (as will be analyzed in Chapter 5). However the following drawbacks are observed:

- The memory efficiency objective of using a sparse matrices library is not fulfilled: As the matrices are input from MATLAB, even the smallest elements of the matrices are stored, instead of leaving free space in the zero elements. Moreover, it has not been possible to guarantee a maximum amount of nonzero elements for varying matrices such as the covariances of the state or the Kalman Gain.
- The quickness objective of using C code is not fulfilled. The lack of efficient algorithms for the computation of inverse matrices implies having to find the Kalman Gain as described in Eq. (4.4); solving N linear systems within a for loop, which can lead to increased computation times for high order systems.
- The complexity of the programming is high: Dealing with a hardware-close language like C and with a library like CXSparse for the implementation of complex algorithms is more complicated than with higher level languages as MATLAB. The developing of the Linear Kalman Filter was not straightforward and it did not require routines such as the Cholesky Factor Update or the QR-Decomposition.

Therefore, the implementation of more complex nonlinear algorithms with MATLAB interpreted functions and MATLAB embedded functions is investigated in the following.

4.2 Sigma-Point Kalman Filters as Interpreted MATLAB Functions

Interpreted MATLAB functions define SIMULINK blocks with just one input and one output where plenty of the MATLAB commands can be used. This tool is employed to test the already existing algorithms of the EKF/UKF toolbox and to develop own efficient algorithms.

The basic structure employed by all the sigma-point filters is depicted in Figure 4.2. The interpreted MATLAB function is the one composed by the *Initialize* routine, where all the initial parameters are given and the input vector is divided into $\boldsymbol{u}_k$ and $\boldsymbol{y}_k$. It calls cyclically the *predict* and *update* functions, each one receiving the required parameters. These functions also call the corresponding *transform* function, where the actual transformation is done, by calling functions *Weights* and *Sigmas* for the computations of Sigma-points and weighing factors.

The values of $\hat{\boldsymbol{x}}_{k-1}^{+}$ and $\mathbf{P}_{k-1}^{+}$ are stored between calls to the function by declaring these variables as `persistent`.

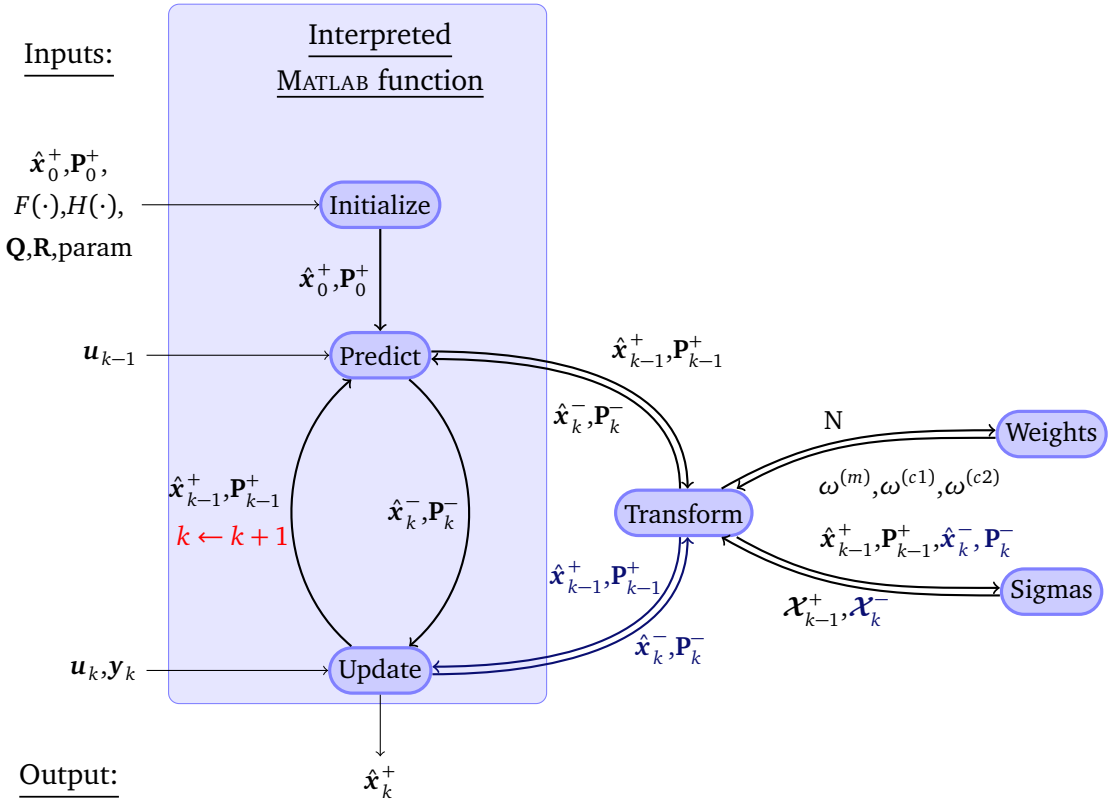

Figure 4.2.: Scheme of the common structure of the filtering algorithms implemented as interpreted MATLAB functions. Dark blue color indicates that paths and variables are employed during the update-step. "param" represents the parameters defining the spread of the sigma-points: α,β,κ for the UKF/SR-UKF and h for the CDKF/SR-CDKF.

The implementation in MATLAB of the already existing routines of the library EKF/UKF required understanding how the functions work. The following filters of the EKF/UKF were implemented successfully:

- Linear Kalman Filter (KF)
- Extended Kalman Filter (EKF)
- Unscented Kalman Filter (UKF)[3]
- Cubature Kalman Filter (CKF)
- Gauss-Hermite Kalman Filter (GHKF)[4]

The understanding achieved was useful to develop own algorithms (SR-UKF, CDKF and SR-CDKF). The next pages analyze the self-developed algorithms, implemented as interpreted MATLAB functions and adopting the nomenclature and structure of the EKF/UKF toolbox algorithms.

Initialize

This is the function that the interpreted MATLAB function block has assigned. The initialization step is done and the further functions (*Predict* and *update*) are called cyclically. The *Initialize* file of the CDKF is shown in Listing 4.1 as an example. When the initialization step is called, the input vector is separated into the $\boldsymbol{u}_k$ and the $\boldsymbol{y}_k$ vectors and, at the first time step of the simulation, all the required data is load from a `.mat` file. These matrices, vectors and scalars are declared as *persistent variables* being available during the whole simulation. They are listed below:

- $\hat{\boldsymbol{x}}_0^+ \rightarrow$ Initial estimate of the state
- $\mathbf{P}_0^+ \rightarrow$ Initial estimate of the covariance of the state
- $\mathbf{Q} \rightarrow$ Covariance matrix of the process noise
- $\mathbf{R} \rightarrow$ Covariance matrix of the measurement noise
- param $\rightarrow$ Set of parameters adjusting the spread of the sigma-points: α, β, κ for the UKF/SR-UKF and h for the CDKF/SR-CDKF.

Afterwards, the *predict* and *update* functions are called cyclically, referencing by means of function handles the dynamics defined in the search path or current directory of MATLAB:

- $F(\cdot) \rightarrow$ State dynamics function in the form of function handle (`@mysystem`)
- $H(\cdot) \rightarrow$ Measurement dynamics function in the form of function handle (`@measurement`)

[3] In the additive noise version (`ukf_predict1` and `ukf_update1`).

[4] This algorithm is in general not supposed to work for $N \geq 4$ [1]. However, when setting the degree of approximation to $p = 1$ some accurate estimations could be achieved.

Listing 4.1: CDKF Interpreted Matlab Function

```
function [ x ] = CDKFmatlabIn( in )
% Interpreted Matlab function for implementation of
% Central Difference Kalman Filter
persistent Q R x_est p_est h
u=in(1:4);
y=in(5:8);
if isempty(Q)
    load('Data.mat');
    Q=Data.Q;
    R=Data.R;
    x_est = Data.X0;
    p_est=Data.Pini;
    h=sqrt(3);
end
[x_prd,p_prd] = cdkf_predict(x_est,p_est,@mysystem,Q,u,h);
[x_est,p_est] = cdkf_update(x_prd,p_prd,y,@measurement,R,u,h);
x=x_est;
end
```

Prediction of Mean and Covariance

When the *predict* function is called, MATLAB searches in the directory or in the path to find the file containing such function. Its main objective is to check that all the variables were initialize correctly and default values are put if necessary. Afterwards, the proper transformation is conducted, by calling the corresponding function and finally the process noise covariance **Q** is added in the not Square Root cases. An example of this function can be seen in Listing 4.2.

Listing 4.2: CDKF Predict

```
function [M,P] = cdkf_predict(M,P,f,Q,f_param,h)
  % Check which arguments there are and apply defaults
  if nargin < 2
    error('Too few arguments');
  end
  if nargin < 3
    f = [];
  end
  if nargin < 4
    Q = [];
  end
  if nargin < 5
    f_param = [];
  end
  if nargin < 6
```

```
    h = [];
end
 if isempty(f)
    f = eye(size(M,1));
end
if isempty(Q)
    Q = zeros(size(M,1));
end
% Do transform and add process noise
[M,P] = cendif_transform(M,P,f,f_param,h);
P = P + Q;
```

Central-Difference Transformation and Unscented Transformation

The *transform* function is where the main part of the algorithm happens, it is called both in the time-update and in the measurement-update[5]. This can be done because of the symmetry that both steps have for the different sigma-point algorithms (see Table 4.2).

The algorithms avoid for loops, in order to make them computationally more efficient. The notation of the EKF/UKF toolbox has been adopted for the new implemented algorithms for the sake of consistency.

Table 4.2.: Equivalence of the steps done in the transformation. The square-root versions of the algorithms compute the Cholesky factor of the covariance **S** instead the covariance **P** (step 5.). The computation of the cross covariance is sometimes done in the *update* function, outside the *transform* one.

Step	**Time-update**	**Measurement-update**
1. Generate a new set of sigma points	$\boldsymbol{\mathcal{X}}_{k-1}^{+}$	$\boldsymbol{\mathcal{X}}_{k}^{-}$
2. Generate a set of weights	$\omega^{(m)}$, $\omega^{(ci)}$	$\omega^{(m)}$, $\omega^{(ci)}$
3. Pass the sigma-points through the function	${}^{*}\boldsymbol{\mathcal{X}}_{k}^{-} = F(\boldsymbol{\mathcal{X}}_{k-1}^{+}, \boldsymbol{u}_{k-1})$	$\boldsymbol{\mathcal{Y}}_{k} = H(\boldsymbol{\mathcal{X}}_{k}^{-}, \boldsymbol{u}_{k})$
4. Compute mean of distribution	$\hat{\boldsymbol{x}}_{k}^{-} = \sum_{i=0}^{2N} \omega_{i}^{(m)} ({}^{*}\boldsymbol{\mathcal{X}}_{k}^{-})_{i}$	$\hat{\boldsymbol{y}}_{k} = \sum_{i=0}^{2N} \omega_{i}^{(m)} (\boldsymbol{\mathcal{Y}}_{k})_{i}$
5. Compute covariance and cross covariance of distribution	$\mathbf{P}_{x_k}^{-}$	$\mathbf{P}_{y_k}$, $\mathbf{P}_{x_k y_k}$

[5] An exception is one of the implementations of the SR-UKF where different sets of sigma points are used for the measurement-update (see Section 2.3.2), requiring a different *transform* function.

Listing 4.3: CDKF transform

```
function [mu,S,X,Y,w] = cendif_transform(M,P,g,g_param,h)
  N=size(M,1);
  if nargin < 4
     g_param = [];
  end
   % Apply defaults
  if nargin < 5
     h = [];
  end
  % Calculate sigma points
  % Bachelor Thesis Eq.: (2.110) and (2.118)
  [WM,WC1,WC2] = cendif_weights(size(M,1),h);
  X = cendif_sigmas(M,P,h);
  w = {WM,WC1,WC2};
  % Propagate through the function
  % Bachelor Thesis Eq.: (2.111) and (2.119)
  if isnumeric(g)
    Y = g*X;
  elseif ischar(g) | strcmp(class(g),'function_handle')
    U=repmat(g_param,1,size(X,2));
    Y = feval(g,X,U);
  else
    U=repmat(g_param,1,size(X,2));
    Y = g(X,U);
  end
       % Bachelor Thesis: Eq. (2.112) and (2.120)
       mu = Y * WM;
       % Bachelor Thesis: Eq. (2.113) and (2.121)
       A= Y(:,2:N+1) - Y(:,N+2:2*N+1);
       B= Y(:,2:N+1) + Y(:,N+2:2*N+1)-repmat(2*Y(:,1),1,N);
       S= WC1(1) * A * A' + WC2(1) * B * B';
  end
```

Computation of sigma-points and their weights

The function `cendif_sigmas` computes the sigma-points of a distribution defined by its mean, its covariance and a parameter h setting the spread of the points. The square-root versions of the algorithms directly receive the Cholesky factor of the covariance and the algorithms based on the unscented transformation (UKF and SR-UKF) get the parameter η instead of h. Listing 4.4 shows the code for the CDKF as an example.

Listing 4.4: CDKF sigmas

```
function X = cendif_sigmas(M,P,h);
% Bachelor Thesis Eq.: (2.110) and (2.118)
A=chol(P)';
X = [zeros(size(M)) A -A];
X = h*X + repmat(M,1,size(X,2));
```

The function `cendif_weights` generates the set of weights to compute the mean and the covariance from the transformed sigma-points. The weights depend on the type of filter:

- **UKF/SR-UKF –** Require $\omega_i^{(m)}$ and $\omega_i^{(c)}$, $i = 0, \ldots, 2N$.
- **CDKF/SR-CDKF –** Require $\omega_i^{(m)}$, $\omega_j^{(c1)}$ and $\omega_j^{(c2)}$, $i = 0, \ldots, 2N$, $j = 1, \ldots, 2N$.

The computation of the CDKF-weights is shown in Listing 4.5 as an example .

Listing 4.5: CDKF weights

```
function [WM,WC1,WC2] = cendif_weights(n,h)
% Check which arguments are there
if nargin < 1
  error('At least dimensionality n required.');
end
if nargin < 2
  h = [];
end
% Apply default values
if isempty(h)
  h = sqrt(3);
end
% Compute the normal weights
WM = zeros(2*n+1,1);
WC1 = zeros(2*n+1,1);
WC2 = zeros(2*n+1,1);
% Bachelor Thesis Eq.: (2.114)-(2.117)
for j=1:2*n+1
  if j==1
    WM(j) = (h^2-n) / h^2;
  else
    WM(j) = 1 / (2 * h^2);
  end
  WC1(j) = 1 / (4 * h^2);
  WC2(j) = ((h^2)-1)/(4*h^4);
end
```

Update of Mean and Covariance

Eventually the *Update* function is called, which presents a lot of similarities with the *Predict* function; they both check the input parameters and call the *transform* function. The main difference is that, at the end of the execution, the last steps of the algorithm are conducted: the cross covariance is computed (when it is not computed in the *Transform* step), the Kalman Gain is computed and *a posterior* estimation of the state and its covariance is done. Listing 4.6 shows the function for the CDKF.

Listing 4.6: CDKF Update

```
function [M,P,K,MU,S] = cdkf_update(M,P,y,H,R,H_param,h)
  % Check that all arguments are there
  if nargin < 5
    error('Too few arguments');
  end
  if nargin < 6
    H_param = [];
  end
  if nargin < 7
    h = [];
  end
  % Do transform and make the update
  [MU,S,~,Y,w] = cendif_transform(M,P,H,H_param,h);
  S=S+R;    %Add measurement noise covariance
  WC1=w{2};
  % Bachelor Thesis Eq.: (2.122)
  C=sqrt(WC1(2))*chol(P)'*(Y(:,2:size(M,1)+1)-Y(:,size(M,1)+2:end))';
  % Bachelor Thesis Eq.: (2.123)
  K = C / S;
  % Bachelor Thesis Eq.: (2.124)
  M = M + K * (y - MU);
  % Bachelor Thesis Eq.: (2.125)
  P = P - K * S * K';
```

It is however interesting to analyze the way in which these last steps of the algorithms are done for the square-root versions of the algorithms, as they rely on functions such as the Cholesky Factor Downdate for the propagation of the error covariance in the measurement update (see Section 2.3.2 for a detailed explanation).

```
  Down = C/S_y;
  K = Down/S_y';
  M = M + K*(y-MU);
  for i=1:size(Down,2)
   S_x=cholupdate(S_x,Down(:,i),'-');
  end
```

The shown extract of the SR-CDKF[6] computes Equations (2.143) to (2.146) (or, equivalently, Equations (2.95) to (2.98) for the SR-UKF). First, the downdate matrix $\mathbf{D}_C$ is computed which is employed to compute the Kalman Gain. Afterwards, the mean is updated with the measurement, and the error covariance is obtained by r consequent Cholesky Factor downdates, using the r columns of $\mathbf{D}_C$ the vectors for each downdate.

4.3 Sigma-Point Kalman Filters as Embedded MATLAB Functions

The embedded functions have the main benefit of eased C code generation directly from SIMULINK. However, a couple of limitations apply (see Section 3.2):

- Model and filter parameters must be defined inside the function.
- Not all the MATLAB commands are supported.

These limitations have two consequences:

- The nonlinear model and the required filter parameters have to be provided within the embedded function.
- Commands such as `feval` and `cholupdate` must be redefined.

Thus, the developed and tested interpreted MATLAB functions have to be modified in order to reach embedded MATLAB functions. Listing 4.7 shows the code of the embedded function CDKF implementation. Both model and filter parameters must be defined inside the function. In this example, the model is a linear system defined by the matrices **A**,**B**,**C** and **D** (lines 5-9). Therefore, the model evaluation (lines 38-41 and 57-59) is that of a linear system. This implementation can easily be modified to define a nonlinear system.

Listing 4.7: Embedded CDKF

```
function [x,yh] = embCDKF(u,y)
persistent A B C D Q R Px_est x_est u_prev h WM WC1 WC2 N
%% STEP 0 INITIALIZATION
if isempty(A)
%%0A DEFINITION OF THE REQUIRED MATRICES FOR THE MODEL
A = [...];
B = [...];
C = [...];
D = [...];
%%0B DEFINITION OF THE REQUIRED PARAMETERS FOR THE FILTER
N=size(A,2);
Q=diag([1e-3 1e-3 1e-6 1e-6 1e-3 1e-3 1e-2 1e-6 1e-1]);
R=diag([1e-1 1e-2 5e2 5e2]);
Px_est=diag([1e-2 1e-3 1e1 1e-3 1e-2 1e-2 1e1 1e-3 1e1]);
x_est=zeros(9,1);
h=sqrt(3);
```

[6] It is the same code as the one employed in the SR-UKF

```
% Bachelor Thesis Eq.: (2.114)-(2.117)
WM = zeros(2*N+1,1);
WC1 = zeros(2*N+1,1);
WC2 = zeros(2*N+1,1);
for j=1:2*N+1
  if j==1
    WM(j) = (h^2-N) / h^2;
  else
    WM(j) = 1 / (2 * h^2);
  end
  WC1(j) = 1 / (4 * h^2);
  WC2(j) = ((h^2)-1)/(4*h^4);
end
u_prev = u;
end
%% STEP 1 TIME UPDATE
%%1A SIGMA-POINTS
% Bachelor Thesis Eq.: (2.110)
S=chol(Px_est)';
X = [zeros(size(x_est)) S -S];
X = h*X + repmat(x_est,1,size(X,2));
%%1B EVALUATION OF STATE EQUATION
U_PREV=repmat(u_prev,1,size(X,2));
% Bachelor Thesis Eq.: (2.111)
Y=A*X+B*U_PREV;
%%1C EVALUATION OF PREDICTED MEAN AND COVARIANCE
% Bachelor Thesis Eq.: (2.112)
x_prd = Y * WM;
% Bachelor Thesis Eq.: (2.113)
aux1= Y(:,2:N+1) - Y(:,N+2:2*N+1);
aux2= Y(:,2:N+1) + Y(:,N+2:2*N+1)-repmat(2*Y(:,1),1,N);
Px_prd= WC1(1) * (aux1 * aux1') + WC2(1) * (aux2 * aux2');
Px_prd = Px_prd + Q;
%% STEP2 MEASUREMENT UPDATE
%%2A SIGMA-POINTS
% Bachelor Thesis Eq.: (2.118)
S=chol(Px_prd)';
X = [zeros(size(x_prd)) S -S];
X = h*X + repmat(x_prd,1,size(X,2));
%%2B EVALUATION OF MEASUREMENT EQUATION
U=repmat(u,1,size(X,2));
% Bachelor Thesis Eq.: (2.119)
Y=C*X+D*U;
%%2C EVALUATION OF PREDICTED MEAN AND COVARIANCE
% Bachelor Thesis Eq.: (2.120)
y_est = Y * WM;
% Bachelor Thesis Eq.: (2.121)
aux1 = Y(:,2:N+1) - Y(:,N+2:2*N+1);
```

```
aux2 = Y(:,2:N+1) + Y(:,N+2:2*N+1)-repmat(2*Y(:,1),1,N);
Py = WC1(1) * (aux1 * aux1') + WC2(1) * (aux2 * aux2');
Py= Py + R;
% Bachelor Thesis Eq.: (2.122)
Pxy=sqrt(WC1(2))*S*(Y(:,2:N+1)-Y(:,N+2:end))';
%%2D KALMAN GAIN
% Bachelor Thesis Eq.: (2.123)
K = Pxy / Py;
% Bachelor Thesis Eq.: (2.124)
x_est = x_prd + K * (y - y_est);
% Bachelor Thesis Eq.: (2.125)
Px_est = Px_prd - K * Py * K';
%%2E OUTPUT
u_prev = u;
yh=y_est;
x=x_est;
end
```

Moreover, in the Square Root versions of the algorithms (see Listing B.2 and Listing B.3), the Cholesky Factor Update function must be implemented. Therefore, according to the theory explained in Section 2.3.2, we get that

$$\tilde{\mathbf{S}} = \text{chol}(\tilde{\mathbf{P}}) = \text{cholupdate}(\mathbf{S}, \sqrt{\nu} \cdot \boldsymbol{x}) = \text{chol}(\mathbf{S}^{\mathrm{T}}\mathbf{S} + \nu \cdot \boldsymbol{x}\boldsymbol{x}^{\mathrm{T}}), \tag{4.5}$$

which in MATLAB code turns

```
Sx_prd=cholupdate(Sx_prd, sqrt(WC(1))*(Y(:,1)-x_prd));
```

into

```
Sx_prd=chol(Sx_prd'*Sx_prd+WC(1)*(Y(:,1)-x_prd)*(Y(:,1)-x_prd)');
```

Equivalently, the propagation of the Cholesky factor of the covariance in the measurement update requires r consequitive Cholesky downdates. This is adapted in an efficient way for C code generation.

```
% Cholesky Downdate for generation of C code
% Bachelor Thesis Eq.: (2.146)
Sx_est=chol(Sx_prd'*Sx_prd - Down*Down');
```

5 Results and Discussion

In this chapter the developed algorithms are tested under different conditions in SIMULINK. First, the developed interpreted MATLAB functions will be tested with the linearized model of a wind turbine. The performance of the filters is compared to that of the filters of the EKF/UKF toolbox, and the performance of the C code Linear Kalman Filter is verified too. Secondly, the developed embedded MATLAB functions are tested with the nonlinear model of a wind turbine, and their performance will be compared to that of the CKF of the EKF/UKF toolbox.

5.1 Optimal Choice of the Filter Parameters

In this section a brief insight in the choice of the filter parameters is given. The selection of the parameters defining the spread of the sigma-points[1] has already been discussed in Section 2.3. It is however interesting to think about the choice of the initial conditions $\hat{\boldsymbol{x}}_0^+$ and $\mathbf{P}_{x_0}^+$ and the noise covariance matrices $\mathbf{Q}$ and $\mathbf{R}$.

In the general case, there is always some uncertainty in the choice of the initial conditions of the filters. Using an observer usually implies that the states can not directly be measured. Therefore, $\hat{\boldsymbol{x}}_0^+$ must be chosen trying to estimate the value of $\boldsymbol{x}_0$ by means of a prior knowledge of the system (for instance, by means of simulations). The error covariance matrix $\mathbf{P}_{x_k}$ is a measure of how far the estimated state and the real state are. Hence, the initial covariance $\mathbf{P}_{x_0}^+$ is a measure of the confidence on the initial conditions given to the filters

$$\mathbf{P}_{x_0}^+ = E[(\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0^+)(\boldsymbol{x}_0 - \hat{\boldsymbol{x}}_0^+)^{\mathrm{T}}]. \tag{5.1}$$

When the difference between the initial conditions of the plant and the filter is high, the initial covariance should also be high, so that the filter trusts more the new measurement than the initial state estimate. However, a compromise is observed: when the covariance is too high, the filters trusts the output too much and a lot of noise gets into the estimated signal.

$$\begin{aligned}\mathcal{K}_k &= \mathbf{P}_{\boldsymbol{x}_k}^- \mathbf{C}^{\mathrm{T}} \mathbf{P}_{\boldsymbol{y}_k}^{-1} = \mathbf{P}_{\boldsymbol{x}_k}^- \mathbf{C}^{\mathrm{T}} \left(\mathbf{C}\mathbf{P}_{\boldsymbol{x}_k}^- \mathbf{C}^{\mathrm{T}} + \mathbf{R}\right)^{-1} = \mathbf{C}^{-1} + \mathbf{P}_{x_k}^- \mathbf{C}^{\mathrm{T}} \mathbf{R}^{-1} \\ &= \mathbf{C}^{-1} + \left(\mathbf{A}\mathbf{P}_{\boldsymbol{x}_{k-1}}^+ \mathbf{A}^{\mathrm{T}} + \mathbf{Q}\right) \mathbf{C}^{\mathrm{T}} \mathbf{R}^{-1}\end{aligned} \tag{5.2}$$

Eq. (5.2) shows that the Kalman gain is proportional to the state error covariance, which confirms that high error covariances lead to a higher confidence in the output than in the previous state estimate and vice versa.

$$\hat{\boldsymbol{x}}_k^+ = \hat{\boldsymbol{x}}_k^- + \mathcal{K}_k(\boldsymbol{y}_k - \hat{\boldsymbol{y}}_k) \tag{5.3}$$

[1] α, β and κ for the UKF and SR-UKF and h for the CDKF and SR-CDKF.

An appropriate setting of the initial covariance leads to better initial dynamics of the filters and a faster reduction of the initial estimation error.

A similar reasoning applies to the choice of the noise matrices **Q** and **R**. **Q** is a measure of the spread of the process noise

$$\mathbf{Q} = E[\boldsymbol{w}_k \boldsymbol{w}_k^{\mathrm{T}}] \tag{5.4}$$

and the state error covariance is proportional to it

$$\mathbf{P}_{\boldsymbol{x}_k}^{-} = \mathbf{A}\mathbf{P}_{\boldsymbol{x}_{k-1}}^{+}\mathbf{A}^{\mathrm{T}} + \mathbf{Q} \tag{5.5}$$

Therefore, high process noises lead to high Kalman Gains (see Eq. (5.2)) and lower confidence in the previous estimates than in the measurements. On the contrary, **R** is a measure of the spread of the measurement noise

$$\mathbf{R} = E[\upsilon_k \upsilon_k^{\mathrm{T}}]. \tag{5.6}$$

Thus, the measurement error covariance is proportional to it

$$\mathbf{P}_{\boldsymbol{y}_k} = \mathbf{C}\mathbf{P}_{\boldsymbol{x}_k}^{-}\mathbf{C}^{\mathrm{T}} + \mathbf{R} \tag{5.7}$$

this leads to an inverse proportional relationship with the Kalman Gain (see Eq. (5.2)). Hence, high measurement noises (high values of **R**) will lead to low Kalman Gains and a higher trust on the model and the previous estimate than on the output, filtering the noise.

In a real implementation, the values of these matrices should correspond with the noise happening in the system. This can be done by modeling these noisy processes or by optimizing the parameters of the matrices through tests.

5.2 Comparison of the Filters with a Linearized Model

In order to test the performance of the implemented filters, the states of the wind turbine and their estimation will be plotted and the Root Mean Square Error (RMSE) of the filter will be computed for each state, as a measure of its estimating accuracy.

$$RMSE_i = \sqrt{\frac{\sum_{k=1}^{T}\left(\hat{x}_{k_i} - x_{k_i}\right)^2}{T}} \qquad i = 1, \ldots, N \tag{5.8}$$

where T represents the total number of steps k in the simulation.

This measure of the error is given in the same unit as the original magnitude. In order to better appreciate the performance of the filters, the *Relative RMSE* was defined having as a reference the MATLAB based Kalman Filter[2].

$$\text{Relative } RMSE_i = \frac{RMSE_i^{GivenFilter}}{RMSE_i^{MatlabKF}} \qquad i = 1, \ldots, N \tag{5.9}$$

The model employed for these simulations is the one described in Appendix A. The filters use the actual model, estimating the wind speed as a state. However, the simulation is done with the modified model, where the wind speed is controlled as an input. This enables to test the reaction of the filters to different wind speed fields.

All the tests are done under some basic conditions of

- **Wrong initial conditions –** The initial conditions of the plant are set to the operating point, whereas the initial conditions of the filter are set to zero for all the states

$$\boldsymbol{x}_0 = \bar{\boldsymbol{x}}_{\text{OP}} \tag{5.10}$$

$$\hat{\boldsymbol{x}}_0^+ = \mathbf{0}. \tag{5.11}$$

  For the reasons exposed in the previous section, the initial covariance in the following simulations has been tuned to

$$\mathbf{P}_0^+ = \texttt{diag([1e-2 1e-3 1e1 1e-3 1e-2 1e-2 1e1 1e-3 1e1])}. \tag{5.12}$$

  minimizing the initial estimation error without damaging the dynamics.

- **Constant input vector $\boldsymbol{u}$–** The system is simulated in open loop, being the inputs set to their values in the operating point. However, different wind fields are applied to the system by means of the modified input vector $\tilde{\boldsymbol{u}}$, where the wind speed can be controlled (see Appendix A for more details).

Some different scenarios will be considered in the simulation by increasing the noise, introducing modeling errors and introducing turbulent wind fields; testing the robustness of the algorithms.

5.2.1 First scenario: Step Wind-Field

The first studied scenario consists on introducing some steps on the wind velocity and analyzing how the filters track the system. As already mentioned, the filters had no knowledge of the initial value of the states. Moreover, some process and measurement noise was introduced to the system with the following covariance matrices:

$$\mathbf{Q} = \texttt{diag([1e-3 1e-3 1e-6 1e-6 1e-3 1e-3 1e-2 1e-6 1e-1])} \tag{5.13}$$

$$\mathbf{R} = \texttt{diag([1e-1 1e-2 5e2 5e2])} \tag{5.14}$$

[2] For the nonlinear case, the CDKF is used as a reference.

Figure 5.2 shows the outputs of the system (vector $\boldsymbol{y}_k$) generated with such a setup and the predicted output $\hat{\boldsymbol{y}}_k$. The tracking of the filters is shown in Figure 5.3; just some of the filters were plotted because they where representative of the behavior of all of them, as can be inferred from Figure 5.1. This Figure shows how the C code Kalman Filter has the same behavior as the MATLAB code Kalman filter, confirming its correct implementation. Besides, the UKF and the CDKF have the same performance as their square-root counterparts. Both perform quite similarly to the CKF. The EKF shows higher variations compared to the other ones.Table 5.1 presents the RMSE for all the states and all the analyzed filters.

As can be seen, the algorithms react satisfactorily, reducing the estimation error due to the unknown initial conditions, reacting to the variations of wind during simulation and filtering the noise of the output of the system.

Table 5.1.: RMSE of the estimation of the states by each filter. Scenario: Wind Step field

		KF Matlab	C-Code KF	EKF	CKF	UKF	SR-UKF	CDKF	SR-CDKF
$\dot{x}_\mathrm{T}$	(m/s)	0.19273	0.19273	0.19417	0.19330	0.19330	0.19330	0.19330	0.19330
$\dot{y}_\mathrm{T}$	(m/s)	0.16903	0.16903	0.17047	0.16906	0.16906	0.16906	0.16906	0.16906
$\dot{\varphi}_\mathrm{g}$	(rad/s)	0.01363	0.01363	0.01778	0.01359	0.01359	0.01359	0.01359	0.01359
$\Delta\dot{\varphi}$	(rad/s)	0.02135	0.02135	0.02211	0.02135	0.02135	0.02135	0.02135	0.02135
x_T	(m)	0.09710	0.09710	0.10070	0.09458	0.09458	0.09458	0.09458	0.09458
y_T	(m)	0.02048	0.02048	0.03638	0.02047	0.02047	0.02047	0.02047	0.02047
φ_g	(rad)	0.19434	0.19434	0.21996	0.19319	0.19319	0.19319	0.19319	0.19319
$\Delta\varphi$	(rad)	0.00189	0.00189	0.00207	0.00188	0.00188	0.00188	0.00188	0.00188
v_w	(m/s)	2.29748	2.29748	2.24681	2.21142	2.21142	2.21142	2.21142	2.21142

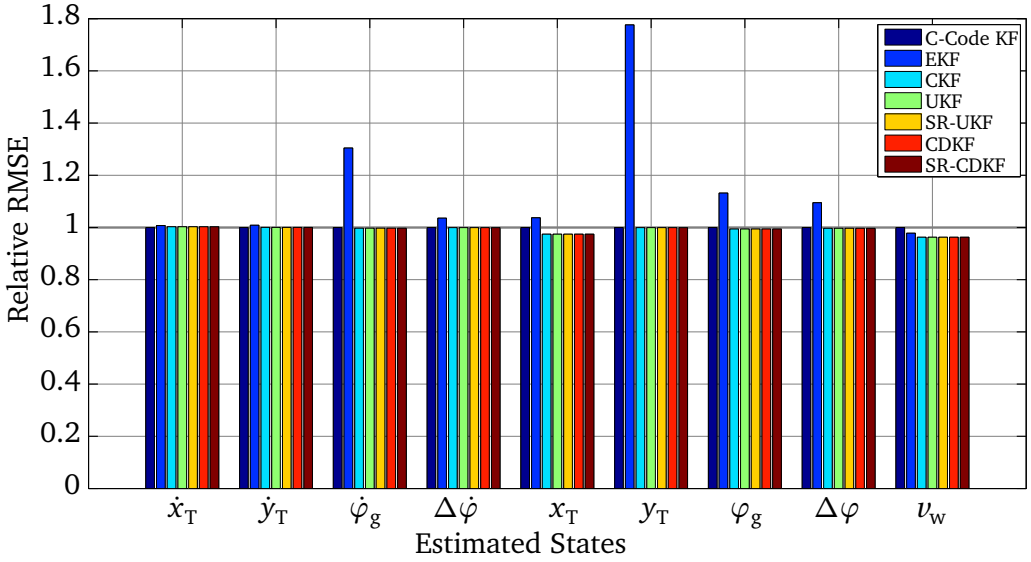

Figure 5.1.: Representation of the RMSE of the estimated states with respect to the RMSE of the MATLAB Kalman Filter. Simulated scenario: Step Wind-Field.

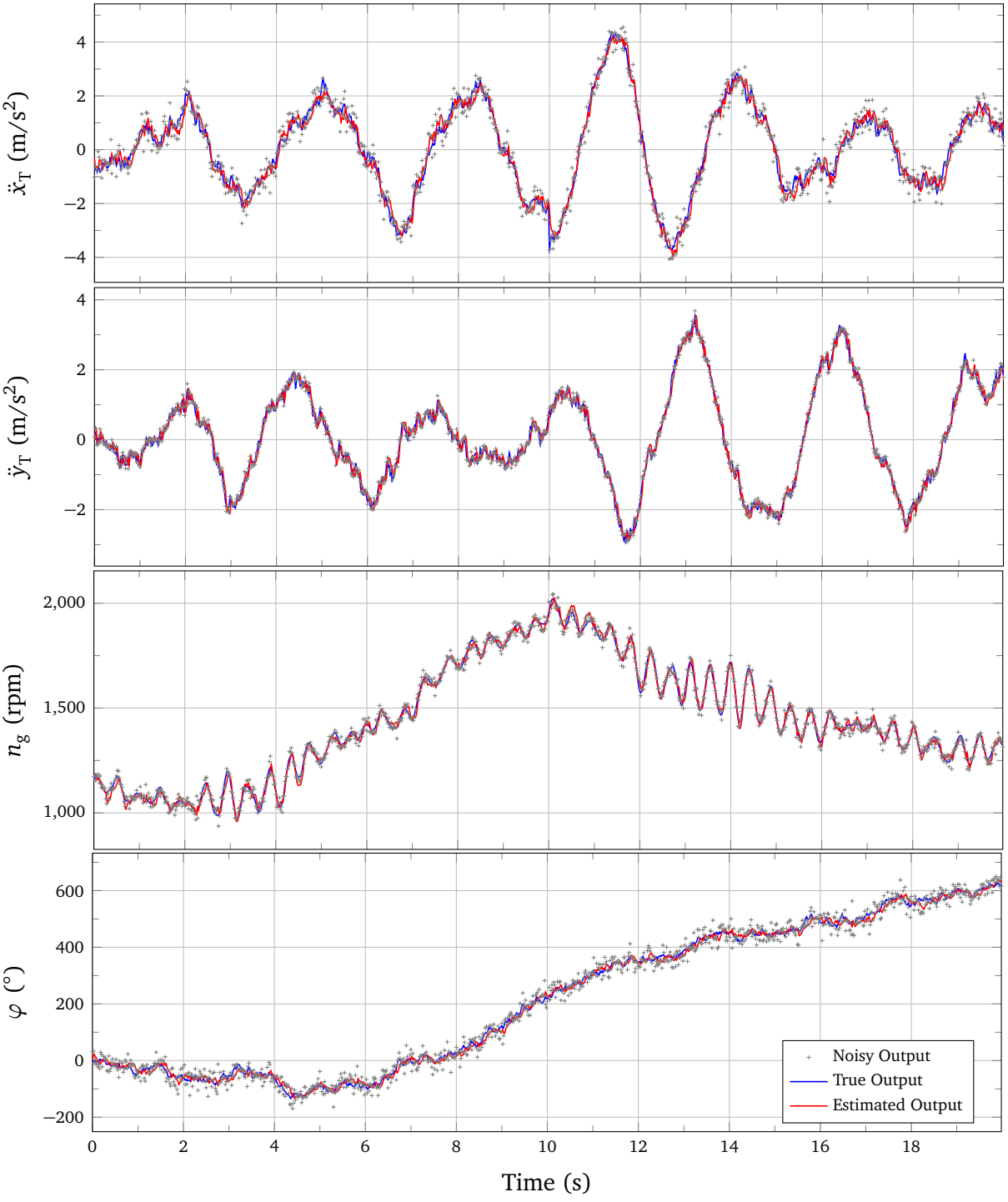

Figure 5.2.: Outputs of the plant, their noisy measure input to the filters and the estimation done by the SR-CDKF. Scenario: Step Wind-Field

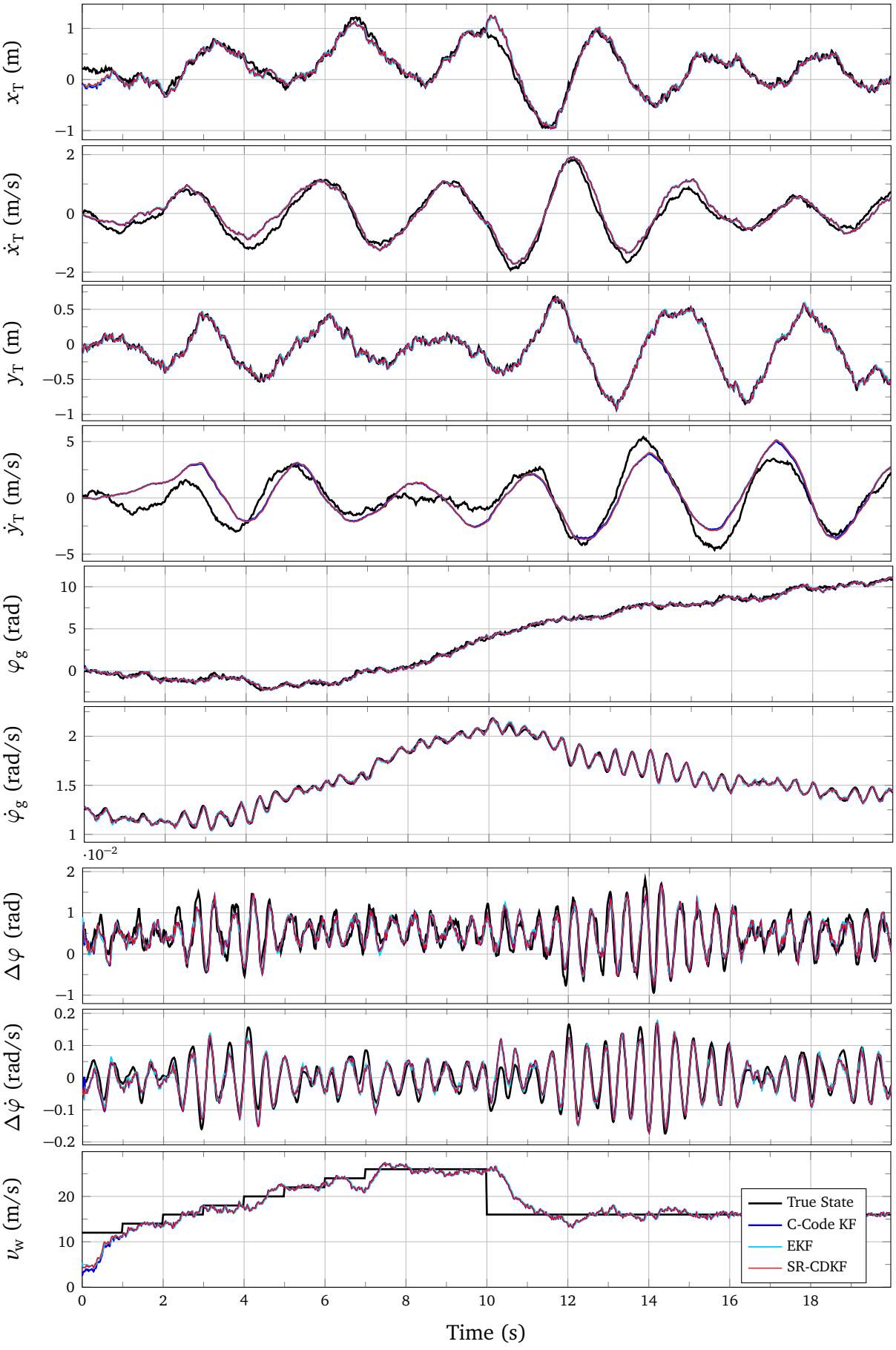

Figure 5.3.: States of the model and their estimation. Scenario: Step Wind-Field.

5.2.2 Second scenario: Modeling Errors

The next simulated scenario attempted to make the filters work under more realistic conditions. Even if the uncertainty about the initial value is maintained, modeling errors were introduced by changing some parameters of the **A** matrix of the filters e.g., the damping factor of the nacelle was set to zero in both the x and y axis. This is done in the continuous case by

$$\mathbf{A}_c(1,1) = 0 \qquad \mathbf{A}_c(2,2) = 0. \tag{5.15}$$

And, due to the following relationship for a sampled system [15],

$$\mathbf{A} = e^{\mathbf{A}_c T} \approx \mathbf{I} + \mathbf{A}_c T \tag{5.16}$$

the required changes in the discrete case are

$$\mathbf{A}_c(1,1) = 1 \qquad \mathbf{A}_c(2,2) = 1. \tag{5.17}$$

The modeling errors change the eigenvalues of the model filter compared to those of the plant; it is therefore a task of the Kalman Filter to adapt the eigenvalues of the error dynamics, to fit those of the plant. This can be done, because the error dynamics depend on the Kalman Gain [16] and it adapts to the plant by means of the received measurements. The eigenvalues of $\mathbf{A} - \mathcal{K}_k\mathbf{C}$ rule the error dynamics. These are dependent on $\mathcal{K}_k$, which varies every step, following a criterion of minimization of the error (as explained on Section 2.2). Thus, these dynamics get recursively adapted to the ones of the plant and the filters learn on each step about the real system, trying to make a better estimation.

Moreover, both the process and measurement noise were increased by employing the following covariance matrices:

$$\mathbf{Q} = \texttt{diag([1e-2 1e-2 1e-6 1e-5 1e-2 1e-2 1e-1 1e-5 1e0])} \tag{5.18}$$

$$\mathbf{R} = \texttt{diag([1e2 1e2 1e4 1e4])} \tag{5.19}$$

This makes the algorithms work with more uncertainty, which enables to test their robustness. The outputs of the plant introduced to the filters can be seen in Figure 5.5 and the achieved tracking of the states in Figure 5.6. The RMSE of the different algorithms are shown in Table 5.2, which is graphically represented in Figure 5.4, by means of the relative RMSE.

The wind field employed for the testing was no longer a step wind-field, but a turbulent wind field. This kind of field was simulated by introducing a noisy signal of mean $\bar{v}_w = 16\,\mathrm{m/s}$. It is not Gaussian noise, as the usually employed noises; the wind speed is correlated. The measured variance of the signal is $q_{v_w} = 3.16\,\mathrm{m}^2/\mathrm{s}^2$.

This simulation shows the high performance of the algorithms filtering the noise of the estimated output. Moreover, the filters are shown to be robust to model uncertainties, adapting the damping of the predicted states to that of the system. The variations of the EKF with respect to the other algorithms gets reduced, but the SPKF algorithms show more difficulties than the linear Kalman Filters for the tracking of x_T and $\dot{x}_T$ under these conditions.

The filters make an accurate estimation of the turbulent wind. The wind speed shows very clearly the effect of wrong initial conditions, since the filter requires approximately 2 seconds until the estimated state reaches the surroundings of the real state.

Table 5.2.: RMSE of the estimation of the states by each filter. Scenario: Modeling Errors.

		KF Matlab	C-Code KF	EKF	CKF	UKF	SR-UKF	CDKF	SR-CDKF
$\dot{x}_T$	(m/s)	0.89179	0.89179	0.98011	0.98142	0.98142	0.98142	0.98142	0.98142
$\dot{y}_T$	(m/s)	1.09546	1.09546	1.11861	1.11780	1.11780	1.11780	1.11780	1.11780
$\dot{\varphi}_g$	(rad/s)	0.05340	0.05340	0.06474	0.05327	0.05327	0.05327	0.05327	0.05327
$\Delta\dot{\varphi}$	(rad/s)	0.06281	0.06281	0.06909	0.06314	0.06314	0.06314	0.06314	0.06314
x_T	(m)	0.49898	0.49898	0.52949	0.51826	0.51826	0.51826	0.51826	0.51826
y_T	(m)	0.49760	0.49760	0.51086	0.50353	0.50353	0.50353	0.50353	0.50353
φ_g	(rad)	0.74734	0.74734	0.81327	0.74321	0.74321	0.74321	0.74321	0.74321
$\Delta\varphi$	(rad)	0.00623	0.00623	0.00655	0.00621	0.00621	0.00621	0.00621	0.00621
v_w	(m/s)	3.61515	3.61515	3.60554	3.55792	3.55792	3.55792	3.55792	3.55792

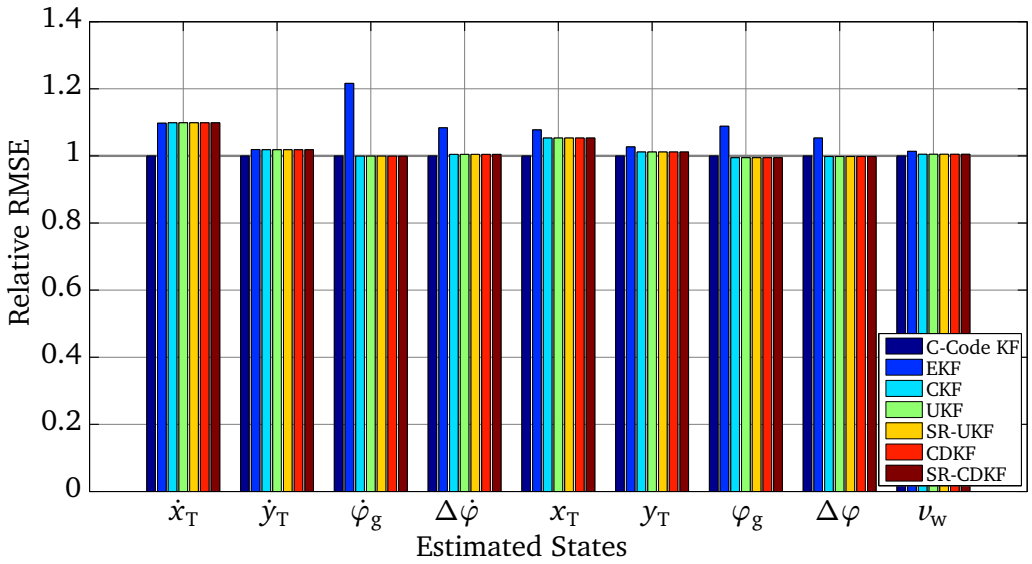

Figure 5.4.: Representation of the RMSE of the estimated states with respect to the RMSE of the MATLAB Kalman Filter. Scenario: Modeling Errors.

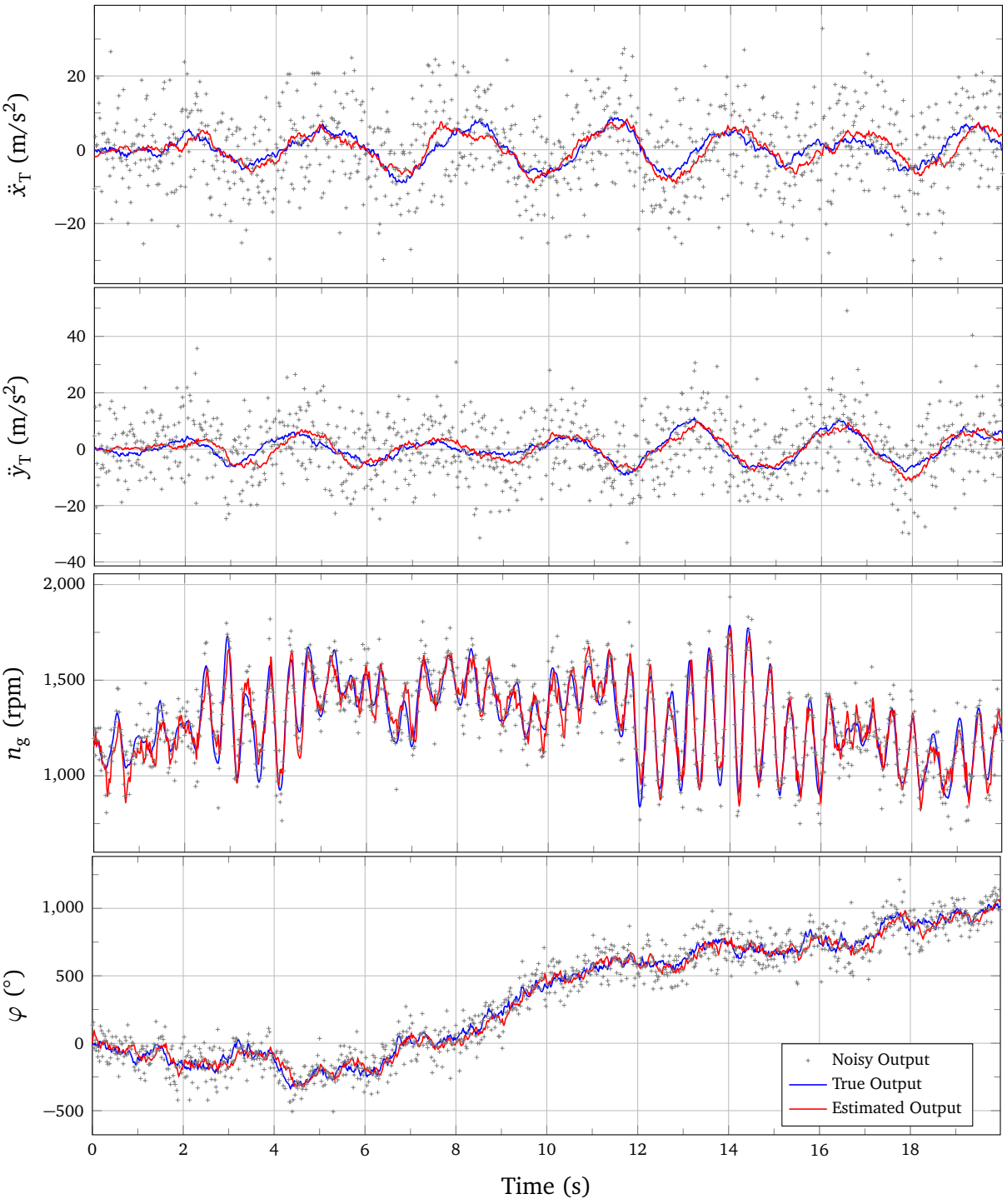

Figure 5.5.: Outputs of the plant, their noisy measure input to the filters and the estimation done by the SR-CDKF.Scenario: Modeling Errors

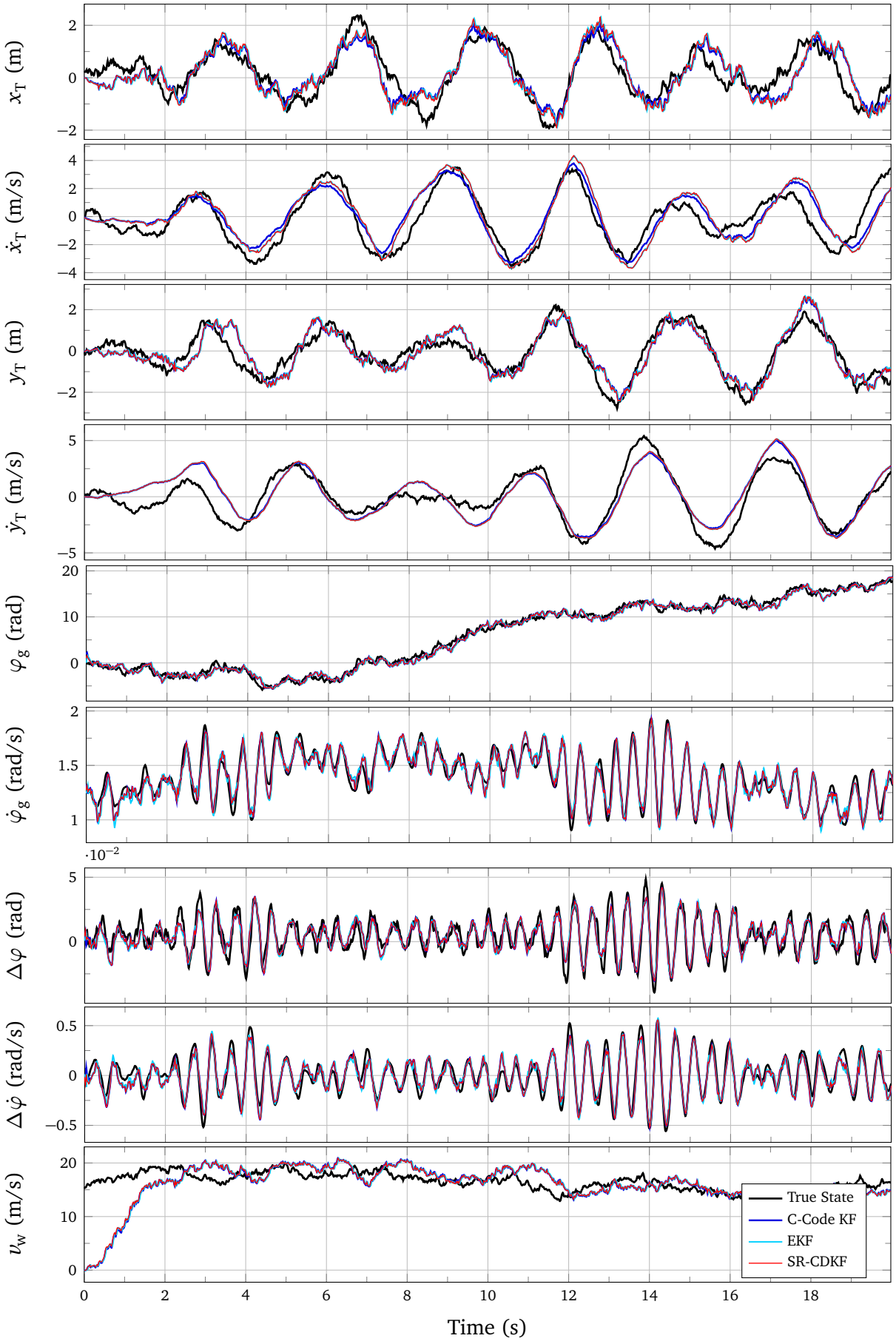

Figure 5.6.: States of the model and their estimation. Scenario: Modeling Errors

5.3 Nonlinear State Estimation

Finally, the sigma-point filtering algorithms are applied to the nonlinear model of a wind turbine[3] using the embedded MATLAB functions. A simulation is presented where the implemented SR-UKF, CDKF and SR-CDKF are compared to the CKF of the EKF/UKF toolbox. The model employed for the simulation has 4 inputs, 7 outputs and 16 states but here only the ones analyzed for the linear case are compared, for symmetry reasons.

In order to get the presented result, the filter matrices $\mathbf{Q}$ and $\mathbf{R}$ are tuned to

$$\mathbf{Q} = \texttt{diag([1e-2, 1e-2, 1e-2, 1e-6, 20, 20, 20, 1e-2,...} \tag{5.20}$$

$$\texttt{1e-1, 1e-1, 1e-6, 1, 1, 1, 1e-4, 10])} \tag{5.21}$$

$$\mathbf{R} = \texttt{diag([1e-2, 3e-3, 1e-4, 1e-5, 1e4, 1e4, 1e4])} \tag{5.22}$$

Besides, some measurement noise is generated with the $\mathbf{R}$ matrix. The input to the system is a turbulent wind field.

The simulation shows a high accuracy filtering the noise of the estimated output and tracking the states. Even if the model is nonlinear, the RMSE of the different algorithms (shown in Table 5.3) can be compared to that of the linear model. The algorithms are tested as MATLAB embedded functions, it is therefore verified that the modifications done to the code enabling the direct C code generation do not have a negative inpact in the performance of the filters. Indeed, the SR-CDKF shows the same performance as the CDKF (see Figure 5.7), and a very similar performance to the SR-UKF. The CKF however presents a higher variation.

Table 5.3.: RMSE of the estimation of the states by each filter. Scenario: Nonlinear Estimation.

		CKF	SR-UKF	CDKF	SR-CDKF
$\dot{x}_\mathrm{T}$	(m/s)	0.03103	0.03378	0.03407	0.03407
$\dot{y}_\mathrm{T}$	(m/s)	0.00451	0.00474	0.00470	0.00470
$\dot{\varphi}_\mathrm{g}$	(rad/s)	0.08035	0.07038	0.07037	0.07037
$\Delta\dot{\varphi}$	(rad/s)	0.00585	0.00629	0.00627	0.00627
x_T	(m)	0.02610	0.02554	0.02533	0.02533
y_T	(m)	0.00367	0.00459	0.00430	0.00430
φ_g	(rad)	0.04366	0.04366	0.04366	0.04366
$\Delta\varphi$	(rad)	0.00042	0.00045	0.00045	0.00045
v_w	(m/s)	1.29774	1.27736	1.27460	1.27460

[3] The model is an extension of the one described in [1].

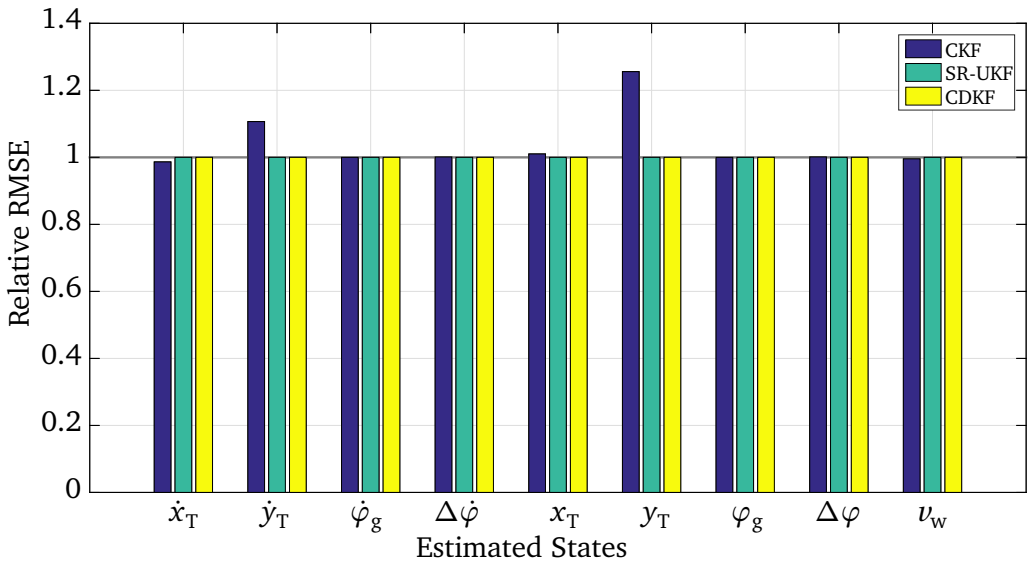

Figure 5.7.: Representation of the RMSE of the estimated states with respect to the RMSE of the CDKF. Scenario: Nonlinear Estimation.

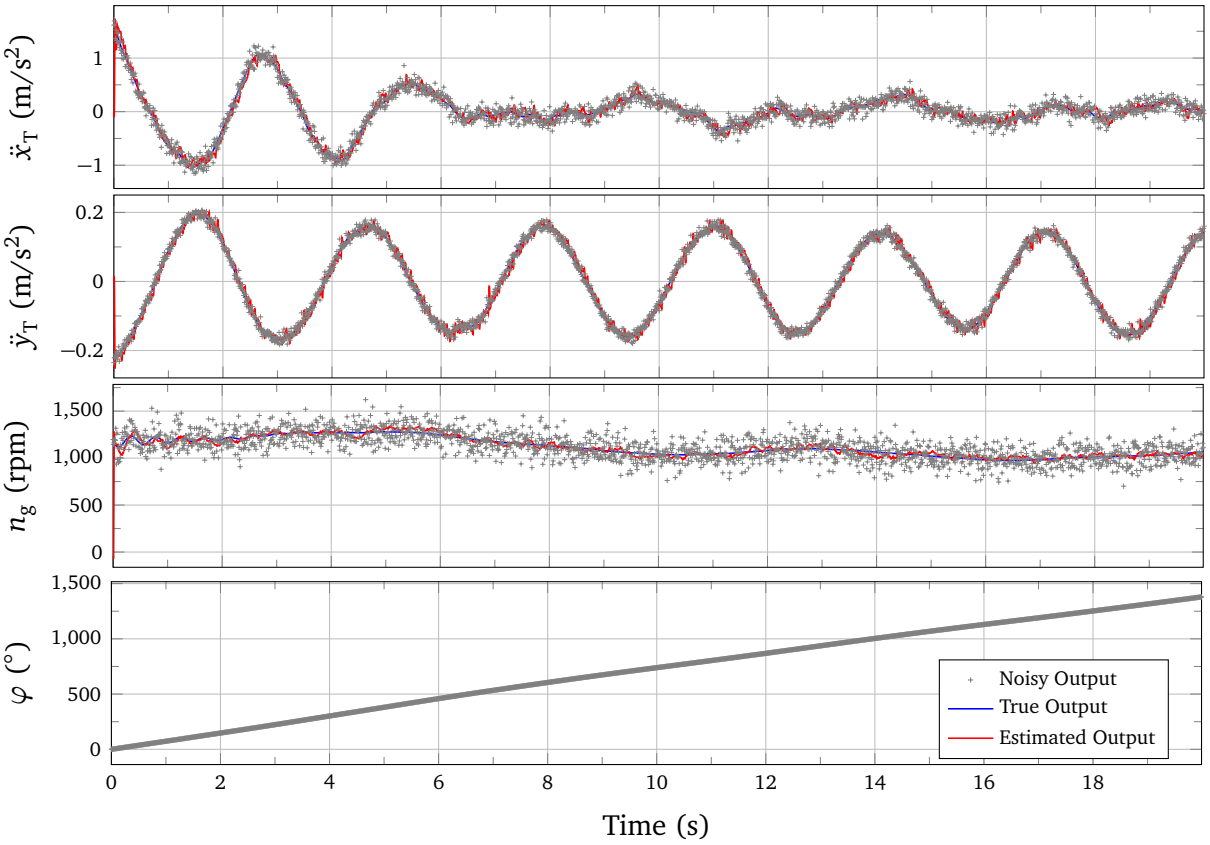

Figure 5.8.: Outputs of the plant, the estimation done by the SR-UKF and their noisy measure, input to the filters.Scenario: Nonlinear Estimation

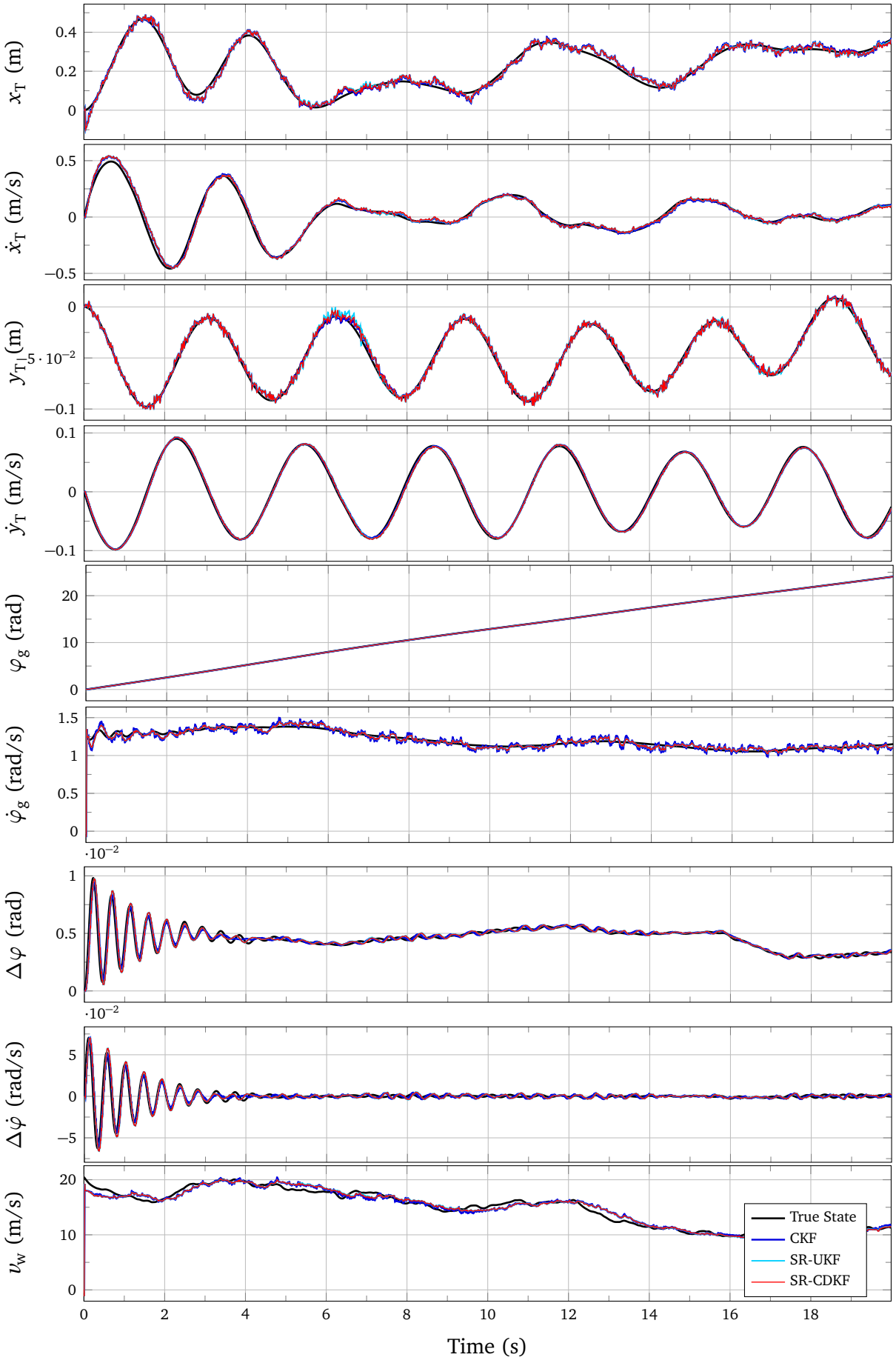

Figure 5.9.: States of the model and their estimation. Scenario: Nonlinear Estimation

6 Conclusion

This chapter summarizes what has been done and what can be done from now on.

6.1 Summary

This thesis constitutes an approach to the topic of nonlinear state estimation. After getting familiar with some basic concepts of stochastic observers and Kalman filtering, the work focuses on Sigma-Point Kalman Filters, analyzing in detail the Unscented Kalman Filter, the Central Difference Kalman Filter and the square-root versions of both algorithms. Once the existing theory is explained, the different ways of implementing the algorithms are discussed. These ways involve the simulation in SIMULINK of the algorithms, either as C code or as MATLAB code. Afterwards, the process of development of the code is explained, giving an detailed analysis of the different implementations.

Eventually, those implementations are tested within SIMULINK. First, all the algorithms implemented as interpreted MATLAB functions are simulated with the linearized model of a wind turbine. This allows comparing the performance and robustness of the different algorithms by varying the noise conditions and introducing model errors. Secondly, the developed nonlinear algorithms are tested as embedded MATLAB functions with the nonlinear model of a wind turbine. These simulations confirm that the square-root version of the algorithms are equivalent to the original algorithms but with better computational properties. Moreover, they show that the algorithms can be adapted for its use as MATLAB embedded functions, which enables the direct efficient C code generation.

6.2 Accomplished Goals

The main objective of this thesis has been the generation and validation of Sigma-Point Kalman Filters for nonlinear state estimation and this objective has been met. The achievements of this thesis are several implementations of the UKF, CDKF and their square-root versions (see Table 6.1), tested in SIMULINK with a complex nonlinear wind turbine model. The nonlinear algorithms are written in MATLAB code and they have successfully been tested as interpreted and embedded MATLAB functions. This eases the task of hardware-implementing the algorithms for their use in real nonlinear state estimation problems.

Furthermore, a fully operative linear Kalman Filter has been written in C code, for its use in linear state estimation problems or for the further development of nonlinear C code algorithms.

Table 6.1.: Summary of the implemented algorithms. "✓" means that the algorithm has been fully implemented, "✗" that an existing implementation (from the EKF/UKF toolbox) has been tested and "-" means that no implementation has been made.

	C-Code S-Function	Interpreted Matlab Function	Embedded Matlab Function
KF	✓	✗	-
EKF	-	✗	-
UKF	-	✗	-
SR-UKF	-	✓	✓
CDKF	-	✓	✓
SR-CDKF	-	✓	✓
CKF	-	✗	✗
SR-CKF	-	-	-
GHKF	-	✗	-

A comparison with some existing filtering algorithms has also been done, as can be seen in Table 6.1.

6.3 Outlook to Further Research

The presented work opens new paths to further research, such as

- Generating C code with MATLAB CODER, introducing it into microcontrollers and testing it with a real system.
- Generalizing the algorithms for the non additive noise case.
- Developing other nonlinear filtering algorithms.
- Continuing with the direct efficient C code implementation of the algorithms (for instance, trying other libraries).
- Modeling the process and measurement noise of a real wind turbine and better tuning the matrices $\mathbf{Q}$ and $\mathbf{R}$.
- Optimizing the selection of the $\hat{\boldsymbol{x}}_0^+$ and the $\mathbf{P}_0^+$ for a given system.
- Modifying the algorithms for parameter or dual estimation.

A Wind Turbine Model with Nine States

The model employed for the testing of the filters is a linearized version of the model presented in [1]. The dimensions of the model are the following:

- Number of states, $N = 9$
- Number of inputs, $M = 4$
- Number of outputs, $r = 4$

It is a linear time invariant discrete model in the form

$$\boldsymbol{x}_k = \mathbf{A}\boldsymbol{x}_{k-1} + \mathbf{B}\boldsymbol{u}_{k-1} \tag{A.1}$$

$$\boldsymbol{y}_k = \mathbf{C}\boldsymbol{x}_k + \mathbf{D}\boldsymbol{u}_k \tag{A.2}$$

The nomenclature of the different variables employed in the model can be seen in Table A.1.

Table A.1.: Nomenclature of inputs, outputs and states of the wind turbine model

$\boldsymbol{u}$	Unit	Inputs	$\boldsymbol{x}$	Unit	States
M_g	(Nm)	generator torque	$\dot{x}_\mathrm{T}$	(m/s)	nacelle fore-aft velocity
β_1	(rad)	blade 1 pitch angle	$\dot{y}_\mathrm{T}$	(m/s)	nacelle side-side velocity
β_2	(rad)	blade 2 pitch angle	$\dot{\varphi}_\mathrm{g}$	(rad/s)	generator angular speed
β_3	(rad)	blade 3 pitch angle	$\Delta\dot{\varphi}$	(rad/s)	drive-train angular speed
$\boldsymbol{y}$	Unit	Outputs	x_T	(m)	nacelle fore-aft position
$\ddot{x}_\mathrm{T}$	(m/s^2)	nacelle fore-aft acceleration	y_T	(m)	nacelle side-side position
$\ddot{y}_\mathrm{T}$	(m/s^2)	nacelle side-side acceleration	φ_g	(rad)	generator azimuth angle
n_g	(rpm)	measured generator speed	$\Delta\varphi$	(rad)	drive-train torsion
φ	(°)	azimuth angle	v_w	(rad/s)	hub-height wind speed

The rank of the observability matrix of the system is 9, which proves that the model is fully observable. It is therefore suitable for the testing of Kalman Filtering algorithms which indeed are stochastic observers.

This is the model employed by the filtering algorithms and the wind speed v_w is considered as a state of the system to be estimated. This is the general case, as the wind is an input to the system which can not be controlled. However, it is useful for simulation purposes to modify the model in order to observe the reaction of the plant and the filters to a given wind field.

The model is the partitioned as follows:

$$\begin{pmatrix} \tilde{\boldsymbol{x}} \\ v_{\mathrm{w}} \end{pmatrix}_k = \begin{pmatrix} \tilde{\mathbf{A}} & \boldsymbol{a}_{v_w} \\ \mathbf{0} & 1 \end{pmatrix} \begin{pmatrix} \tilde{\boldsymbol{x}} \\ v_{\mathrm{w}} \end{pmatrix}_{k-1} + \begin{pmatrix} \tilde{\mathbf{B}}' \\ \mathbf{0} \end{pmatrix} \boldsymbol{u}_{k-1} \tag{A.3}$$

$$\boldsymbol{y}_k = \begin{pmatrix} \tilde{\mathbf{C}} & \boldsymbol{c}_{v_w} \end{pmatrix} \boldsymbol{x}_k + \mathbf{D}\boldsymbol{u}_k \tag{A.4}$$

Which implies the following relationship

$$v_{\mathrm{w}_k} = v_{\mathrm{w}_{k-1}} \tag{A.5}$$

The wind disturbance is modeled by a constant to be estimated. The system can therefore be transformed to the following equivalent system, where the wind speed is an input to the system:

$$\tilde{\boldsymbol{x}}_k = \tilde{\mathbf{A}}\tilde{\boldsymbol{x}}_{k-1} + \tilde{\mathbf{B}}\tilde{\boldsymbol{u}}_{k-1} \tag{A.6}$$

$$\boldsymbol{y}_k = \tilde{\mathbf{C}}\tilde{\boldsymbol{x}}_k + \tilde{\mathbf{D}}\tilde{\boldsymbol{u}}_k \tag{A.7}$$

Where

$$\begin{aligned} \tilde{\mathbf{A}} &= \mathbf{A}(1:8,1:8) & \tilde{\mathbf{B}} &= \begin{pmatrix} \tilde{\mathbf{B}}' & \boldsymbol{a}_{v_{\mathrm{w}}} \end{pmatrix} \\ \tilde{\mathbf{C}} &= \mathbf{C}(:,1:8) & \tilde{\mathbf{D}} &= \begin{pmatrix} \mathbf{D} & \boldsymbol{c}_{v_{\mathrm{w}}} \end{pmatrix} \\ \boldsymbol{a}_{v_w} &= \mathbf{A}(1:8,9) & \boldsymbol{c}_{v_w} &= \mathbf{C}(:,9) \\ \tilde{\boldsymbol{x}} &= \boldsymbol{x}(1:8) & \tilde{\boldsymbol{u}} &= \begin{pmatrix} \boldsymbol{u} \\ v_{\mathrm{w}} \end{pmatrix} \end{aligned} \tag{A.8}$$

The model was linearized with respect to an operating point $\boldsymbol{x}_{\mathrm{OP}}$, such that

$$\boldsymbol{x}_{\mathrm{OP}} = [0.000, 0.000, 1.260, 0.000, 0.203, -0.050, 0.025, 0.005, 16.000]^{\mathrm{T}} \tag{A.9}$$

B Source Code

Listing B.1: C-Code Kalman Filter

```
/*
 * File : turKFsparse.c
 * Abstract:
 *       A Linear Kalman Filter implemented in C, for its use as a
 *       S-function in Simulink
 *
 */

#define S_FUNCTION_NAME  turKFsparse
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"
# include "cs.h"

static void makesparse (real_T *value_array, cs *sparsematrix, int rows, int →
    ←columns );

#define PARAM_matrixA ssGetSFcnParam(S,0)
#define PARAM_matrixB ssGetSFcnParam(S,1) //in order to input matrices as →
    ←parameters
#define PARAM_matrixC ssGetSFcnParam(S,2)
#define PARAM_matrixQ ssGetSFcnParam(S,3)
#define PARAM_matrixR ssGetSFcnParam(S,4)
#define PARAM_vectorXini ssGetSFcnParam(S,5)
#define PARAM_matrixPini ssGetSFcnParam(S,6)

/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *   Setup sizes of the various vectors.
 */
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 7);
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return; /* Parameter mismatch will be reported by Simulink */
    }

    // Setting the widths of the input ports
    if (!ssSetNumInputPorts(S, 2)) return;
    ssSetInputPortWidth(S, 0, 4);
```

```
    ssSetInputPortDirectFeedThrough(S, 0, 1);
    ssSetInputPortWidth(S, 1, 4);
    ssSetInputPortDirectFeedThrough(S, 1, 1);

    if (!ssSetNumOutputPorts(S,1)) return;
    ssSetOutputPortWidth(S, 0, 9);

    ssSetNumSampleTimes(S, 1);

    /* specify the sim state compliance to be same as a built-in block */
    ssSetSimStateCompliance(S, USE_DEFAULT_SIM_STATE);

    ssSetOptions(S,
                 SS_OPTION_WORKS_WITH_CODE_REUSE |
                 SS_OPTION_EXCEPTION_FREE_CODE |
                 SS_OPTION_USE_TLC_WITH_ACCELERATOR);
}

/* Function: mdlInitializeSampleTimes =========================================
 * Abstract:
 *    Specifiy that we inherit our sample time from the driving block.
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
    ssSetModelReferenceSampleTimeDefaultInheritance(S);
}

  //Definition of elements required in the function

   int key=1;

  //Definition of sparse elements
  cs *F;
  cs *G;
  cs *H;
  cs *P_prd;
  cs *P_est;
  cs *x_prd;
  cs *x_est;
  cs *Q;
  cs *R;
  cs *Saux;
  cs *Baux;
  cs *klm_gain;
  cs *auxvecsparse;
```

```
/* Function: mdlOutputs =======================================================
 * Abstract:
 *   The outputs of the function are computed here
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    //input the variables that change every sample time
    int_T               i;
    InputRealPtrsType   uk = ssGetInputPortRealSignalPtrs(S,0);
    InputRealPtrsType   yk = ssGetInputPortRealSignalPtrs(S,1);
    real_T              *xest1 = ssGetOutputPortRealSignal(S,0);
    int_T               width = ssGetOutputPortWidth(S,0);

// Initaialize the filter
//---------------------------------------------------------------------------
    //Puting the inputs of the block in form of vectors for easier computations
    // This should be modified for diferent numbers of inputs/outputs/states
    double y_vec[4]={*yk[0],*yk[1],*yk[2],*yk[3]};
    double u_vec[4]={*uk[0],*uk[1],*uk[2],*uk[3]};
    double b[9];
    double auxvec[9];

    int j,l;
    int p=0,k=0,lim=0;
    // To make the algorithm more general, it automatically detects the number →
        ←of inputs/outputs/states (but they can also be defined directly)
    int N = (int)mxGetNumberOfElements(PARAM_vectorXini);//9; //Number of states
    int M = ssGetInputPortWidth(S,0);//4; //Number of inputs
    int r = ssGetInputPortWidth(S,1);//4; //Number of outputs

    if (key){

    //Input matrices as parameters
    real_T              *A = mxGetPr(PARAM_matrixA);
    real_T              *B = mxGetPr(PARAM_matrixB);
    real_T              *C = mxGetPr(PARAM_matrixC);
    real_T              *Qin = mxGetPr(PARAM_matrixQ);
    real_T              *Rin = mxGetPr(PARAM_matrixR);
    real_T              *P_ini = mxGetPr(PARAM_matrixPini);
    real_T              *x_ini = mxGetPr(PARAM_vectorXini);

    F = cs_spalloc(N, N, N*N, 1, 1);            //allocating space in memory
    makesparse(A,F,N,N);                        //turning the matrix sparse
    F=cs_compress(F);                           //turning it from triplet to CC

    G = cs_spalloc(N, M, N*N, 1, 1);
    makesparse(B,G,N,M);
    G=cs_compress(G);
```

```
    H = cs_spalloc(r, N, r*N, 1, 1);
    makesparse(C,H,r,N);
    H=cs_compress(H);

    Q = cs_spalloc(N, N, N*N, 1, 1);
    makesparse(Qin,Q,N,N);
    Q=cs_compress(Q);

    R = cs_spalloc(r, r, r*r, 1, 1);
    makesparse(Rin,R,r,r);
    R=cs_compress(R);

    x_est = cs_spalloc(N, 1, N, 1, 1);
    makesparse(x_ini,x_est,N,1);
    x_est=cs_compress(x_est);

    P_est = cs_spalloc(N, N, N*N, 1, 1);
    makesparse(P_ini,P_est,N,N);
    P_est=cs_compress(P_est);

    x_prd = cs_spalloc(N, 1, N, 1, 1);
    P_prd = cs_spalloc(N, N, N*N, 1, 1);

    Saux = cs_spalloc(r, r, r*r, 1, 1);
    Baux = cs_spalloc(r, N, r*r, 1, 1);

    klm_gain = cs_spalloc(N, r, N*r, 1, 1);

    key=0;                                          //so that it only gets →
        ←initialized at the beginning
    }

//Time-update
//------------------------------------------------------------------------

    // Predicted state and covariance

    for(j=0; j<(sizeof(auxvec)/sizeof(auxvec[0]));++j){
    auxvec[j]=0 ;                                   //Reset auxiliary vector
    }

    cs_gaxpy(G, u_vec, auxvec);                     //auxvec=G*u_vec+auxvec
    cs_spfree(auxvecsparse);                        //reset sparse auxiliary →
        ←vector
    auxvecsparse = cs_spalloc(N, 1, N, 1, 1);       //allocate sparse auxiliary →
        ←vector
```

```
    makesparse(auxvec,auxvecsparse,N,1);            //turn auxiliary vector into→
        ← sparse
    auxvecsparse=cs_compress(auxvecsparse);         //turn form triplet into →
        ←compressed column, to operate

    //x_prd = F * x_est+ G * u;
    x_prd=cs_add(cs_multiply(F,x_est),auxvecsparse,1,1);

    //p_prd = F * p_est * F' + Q;
    P_prd=cs_add(cs_multiply(cs_multiply(F,P_est),cs_transpose(F,1)),Q,1,1);

//Measurement-update
//-----------------------------------------------------------------------
    // Saux = H * p_prd' * H' + R;
    Saux=cs_add(cs_multiply(H,cs_multiply(cs_transpose(P_prd,1),cs_transpose(H,1)→
        ←)),cs_transpose(R,1),1,1);

    // Baux = H * p_prd';
    Baux=cs_multiply(H,cs_transpose(P_prd,1));

    //code for computing Kalman Gain: Kg=Saux\Baux

    cs_spfree(klm_gain);                            //reset Kalman Gain
    klm_gain = cs_spalloc(r, N, N*r, 1, 1);         //Allocate Kalman Gain
    for (p=0; p<N; ++p){

        for(j=0; j<r; ++j){
        b[j]=Baux->x[j+k];                          //Make column vectors from →
            ←Baux
        }

        cs_lusol (0, Saux, b, 1);                   //solve Saux*x=b; answer in →
            ←b

        for(j=0; j<r; ++j){
        cs_entry(klm_gain, j, p, b[j]);             //Store answer in the →
            ←corresponding column of klm_gain
        }
    k=k+r;
    }
    k=0;
    klm_gain=cs_compress(klm_gain);                 //to compressed column →
        ←format, to operate
    klm_gain=cs_transpose(klm_gain,1);              //Transpose klm_gain


    //code for computing auxvecsparse= Klm_gain*y
```

```
    for(j=0; j<(sizeof(auxvec)/sizeof(auxvec[0]));++j){
    auxvec[j]=0 ;                                    //reset auxvec
    }
    cs_gaxpy(klm_gain, y_vec, auxvec);
    cs_spfree(auxvecsparse);
    auxvecsparse = cs_spalloc(N, 1, N, 1, 1);
    makesparse(auxvec,auxvecsparse,N,1);
    auxvecsparse=cs_compress(auxvecsparse);

    //x_est = x_prd + klm_gain * (y - H * x_prd) = x_prd + Klm_gain*y - klm_gain→
        ←*H*x_prd;
    x_est=cs_add(x_prd,cs_add(auxvecsparse,cs_multiply(klm_gain,cs_multiply(H,→
        ←x_prd)),1,-1),1,1);

    //p_est = p_prd - klm_gain * H * p_prd;
    P_est=cs_add(P_prd,cs_multiply(klm_gain,cs_multiply(H,P_prd)),1,-1);

    for (i=0; i<width; i++) {
      *xest1++ =x_est->x[i];                         //Show the result
    }
}

/* Function: mdlTerminate =====================================================
 * Abstract:
 *    No termination needed, but we are required to have this routine.
 */
static void mdlTerminate(SimStruct *S){}

/* Function: makesparse =======================================================
 * Abstract:
 *    This function stores the values of the matrix introduced as a parameter to→
     ← the
 *    Sfunction into a triplet like sparse matrix
 */
 static void makesparse (double *value_array, cs *sparsematrix, int rows, int →
     ←columns ){
    int j,k,p=0;
    for (k=0; k<columns; ++k){
    for (j=0; j<rows; ++j) {
        cs_entry(sparsematrix, j, k, value_array[p+j]);
    }
    p=p+j;
   }
  p=0;
 }

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
```

```
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

Listing B.2: Embedded SRCDKF

```
function [x,yh] = embSRCDKF(u,y)
persistent A B C D srQ srR Sx_est x_est u_prev h WM WC1 WC2 N
%% Initialization
if isempty(A)
%%0A DEFINITION OF THE REQUIRED MATRICES FOR THE MODEL
A = [...];
B = [...];
C = [...];
D = [...];
%%0B DEFINITION OF THE REQUIRED PARAMETERS FOR THE FILTER
N=size(A,2);
Q=diag([1e-3 1e-3 1e-6 1e-6 1e-3 1e-3 1e-2 1e-6 1e-1]);
srQ=chol(Q);
R=diag([1e-1 1e-2 5e2 5e2]);
srR=chol(R);
Px_est=diag([1e-2 1e-3 1e1 1e-3 1e-2 1e-2 1e1 1e-3 1e1]);
Sx_est=chol(Px_est);
x_est=zeros(16,1);
h=sqrt(3);
% Bachelor Thesis Eq.: (2.114)-(2.117)
WM = zeros(2*N+1,1);
WC1 = zeros(2*N+1,1);
WC2 = zeros(2*N+1,1);
for j=1:2*N+1
  if j==1
    WM(j) = (h^2-N) / h^2;
  else
    WM(j) = 1 / (2 * h^2);
  end
  WC1(j) = 1 / (4 * h^2);
  WC2(j) = ((h^2)-1)/(4*h^4);
end
u_prev = u;
end
%% STEP 1 TIME UPDATE
%%1A SIGMA-POINTS
% Bachelor Thesis Eq.: (2.131)
X = [zeros(size(x_est)) Sx_est' -Sx_est'];
X = h*X + repmat(x_est,1,size(X,2));
%%1B EVALUATION OF STATE EQUATION
U_PREV=repmat(u_prev,1,size(X,2));
% Bachelor Thesis Eq.: (2.132)
```

```
Y=A*X+B*U_PREV;
%%1C EVALUATION OF PREDICTED MEAN AND COVARIANCE
% Bachelor Thesis Eq.: (2.133)
x_prd = Y * WM;
% Bachelor Thesis Eq.: (2.134)
aux1= Y(:,2:N+1) - Y(:,N+2:2*N+1);
aux2= Y(:,2:N+1) + Y(:,N+2:2*N+1)-repmat(2*Y(:,1),1,N);
[~,Sx_prd]=qr([sqrt(WC1(2))* aux1 ,sqrt(WC2(2))*aux2,srQ']',0);
%% STEP2 MEASUREMENT UPDATE
%%2A SIGMA-POINTS
% Bachelor Thesis Eq.: (2.137)
X = [zeros(size(x_prd)) Sx_prd' -Sx_prd'];
X = h*X + repmat(x_prd,1,size(X,2));
%%2B EVALUATION OF MEASUREMENT EQUATION
U=repmat(u,1,size(X,2));
% Bachelor Thesis Eq.: (2.138)
Y=C*X+D*U;
%%2C EVALUATION OF PREDICTED MEAN AND COVARIANCE
% Bachelor Thesis Eq.: (2.139)
y_est = Y * WM;
aux1 = Y(:,2:N+1) - Y(:,N+2:2*N+1);
aux2 = Y(:,2:N+1) + Y(:,N+2:2*N+1)-repmat(2*Y(:,1),1,N);
% Bachelor Thesis Eq.: (2.140)
[~,Sy]=qr([sqrt(WC1(2))* aux1 ,sqrt(WC2(2))*aux2,srR']',0);
% Bachelor Thesis Eq.: (2.142)
Pxy=sqrt(WC1(2))*Sx_prd'*(Y(:,2:N+1)-Y(:,N+2:end))';
%%2D KALMAN GAIN
% Bachelor Thesis Eq.: (2.145)
Down = Pxy/Sy;
% Bachelor Thesis Eq.: (2.143)
K = Down/Sy';
% Bachelor Thesis Eq.: (2.144)
x_est = x_prd + K * (y - y_est);
% Cholesky Downdate for generation of C code
% Bachelor Thesis Eq.: (2.146)
Sx_est=chol(Sx_prd'*Sx_prd - Down*Down');
%%2E OUTPUT
u_prev = u;
yh=y_est;
x=x_est;
end
```

Listing B.3: Embedded SRUKF

```
function [x,yh] = embSRUKF(u,y)
persistent A B C D srQ srR Sx_est x_est u_prev alpha beta kappa eta WM WC N
%% Initialization
if isempty(A)
%%0A DEFINITION OF THE REQUIRED MATRICES FOR THE MODEL
A = [...];
B = [...];
C = [...];
D = [...];
%%0B DEFINITION OF THE REQUIRED PARAMETERS FOR THE FILTER
N=size(A,2);
Q=diag([1e-3 1e-3 1e-6 1e-6 1e-3 1e-3 1e-2 1e-6 1e-1]);
srQ=chol(Q);
R=diag([1e-1 1e-2 5e2 5e2]);
srR=chol(R);
Px_est=diag([1e-2 1e-3 1e1 1e-3 1e-2 1e-2 1e1 1e-3 1e1]);
Sx_est=chol(Px_est);
x_est=zeros(16,1);
alpha=0.1;
beta=2;
kappa=0;
lambda = alpha^2 * (N + kappa) - N;
% Bachelor Thesis Eq.: (2.45)-(2.46)
WM = zeros(2*N+1,1);
WC = zeros(2*N+1,1);
for j=1:2*N+1
  if j==1
    WM(j) = lambda / (N + lambda);
    WC(j) = lambda / (N + lambda) + (1 - alpha^2 + beta);
  else
    WM(j) = 1 / (2 * (N + lambda));
    WC(j) = WM(j);
  end
end
% Bachelor Thesis Eq.: (2.42)
eta = sqrt(N + lambda);
u_prev = u;
end
%% STEP 1 TIME UPDATE
%%1A SIGMA-POINTS
% Bachelor Thesis Eq.: (2.72)
X = [zeros(size(x_est)) Sx_est' -Sx_est'];
X = eta*X + repmat(x_est,1,size(X,2));
%%1B EVALUATION OF STATE EQUATION
U_PREV=repmat(u_prev,1,size(X,2));
% Bachelor Thesis Eq.: (2.73)
```

```
47 Y=A*X+B*U_PREV;
   %%1C EVALUATION OF PREDICTED MEAN AND COVARIANCE
   % Bachelor Thesis Eq.: (2.74)
   x_prd = Y * WM;
   % Bachelor Thesis Eq.: (2.75)
52 [~,Sx_prd]=qr([sqrt(WC(2))*(Y(:,2:end)-repmat(x_prd,[1,size(Y(:,2:end),2)])),srQ→
       ←']',0);
   % Bachelor Thesis Eq.: (2.76)
   Sx_prd=chol(Sx_prd'*Sx_prd+WC(1)*(Y(:,1)-x_prd)*(Y(:,1)-x_prd)');
   %% STEP2 MEASUREMENT UPDATE
   %%2A SIGMA-POINTS
57 % Bachelor Thesis Eq.: (2.81)
   X = [zeros(size(x_prd)) Sx_prd' -Sx_prd'];
   X = eta*X + repmat(x_prd,1,size(X,2));
   %%2B EVALUATION OF MEASUREMENT EQUATION
   U=repmat(u,1,size(X,2));
62 % Bachelor Thesis Eq.: (2.82)
   Y=C*X+D*U;
   %%2C EVALUATION OF PREDICTED MEAN AND COVARIANCE
   % Bachelor Thesis Eq.: (2.83)
   y_est = Y * WM;
67 % Bachelor Thesis Eq.: (2.84)
   [~,Sy]=qr([sqrt(WC(2))*(Y(:,2:end)-repmat(y_est,[1,size(Y(:,2:end),2)])),srR→
       ←']',0);
   % Bachelor Thesis Eq.: (2.85)
   Sy=chol(Sy'*Sy+WC(1)*(Y(:,1)-y_est)*(Y(:,1)-y_est)');
   % Bachelor Thesis Eq.: (2.86)
72 Pxy = WC(1) * (X(:,1)-x_prd) * (Y(:,1) - y_est)' + WC(2) * (X(:,2:end)-repmat(→
       ←x_prd,1,size(X,2)-1)) * (Y(:,2:end) - repmat(y_est,1,size(X,2)-1))';
   %%2D KALMAN GAIN
   % Bachelor Thesis Eq.: (2.97)
   Down = Pxy/Sy;
   % Bachelor Thesis Eq.: (2.95)
77 K = Down/Sy';
   % Bachelor Thesis Eq.: (2.96)
   x_est = x_prd + K * (y - y_est);
   % Cholesky Downdate for C code generation
   % Bachelor Thesis Eq.: (2.98)
82 Sx_est=chol(Sx_prd'*Sx_prd - Down*Down');
   %%2E OUTPUT
   u_prev = u;
   yh=y_est;
   x=x_est;
87 end
```

Bibliography

[1] Bastian Ritter, Axel Schild, Matthias Feldt and Ulrich Konigorski: *The design of nonlinear observers for wind turbine dynamic state and parameter estimation*. Journal of Physics: Conference Series, IOP Publishing, (accepted for publication), 2016.

[2] Kalman, Rudolph Emil: *A new approach to linear filtering and prediction problems*. Journal of basic Engineering 82.1, 1960.

[3] Van Der Merwe, Rudolph: *Sigma-point Kalman filters for probabilistic inference in dynamic state-space models*. PhD thesis, Oregon Health & Science University, 2004.

[4] Orderud, Fredrik: *Comparison of Kalman Filter Estimation Approaches for State Space Models with Nonlinear Measurements*. Proc. of Scandinavian Conference on Simulation and Modeling, 2005.

[5] Van Der Merwe, Rudolph and Eric A. Wan: *The Square-Root Unscented Kalman Filter for State and Parameter Estimation*. IEEE International Conference on. Vol. 6., 2001.

[6] Julier, Simon J. and Jeffrey K. Uhlmann: *A New Extension of the Kalman Filter to Nonlinear Systems*. AeroSense'97. International Society for Optics and Photonics, 1997.

[7] Simon, Dan: *Optimal state estimation: Kalman, H infinity, and nonlinear approaches*. John Wiley & Sons, 2006.

[8] Hartikainen, Jouni, Arno Solin and Simo Särkkä: *Optimal filtering with Kalman filters and smoothers*. Department of Biomedica Engineering and Computational Sciences, Aalto University School of Science, 2011.

[9] Julier, Simon J.: *The Scaled Unscented Tansformation*. American Control Conference, 2002.

[10] Van Der Merwe, Rudolph and Eric A. Wan: *Efficient Derivative-Free Kalman Filters for Online Learning*. European Symposium on Artificial Neural Networks, 2001.

[11] Campa, Giampiero: *What is the best way to implement my algorithm in* Simulink*?* In *The MathWorks*, 2014.

[12] *www.mathworks.com*.

[13] The MathWorks: *Simulink – Writing S-Functions*, 2002.

[14] Davis, Timothy A.: *Direct Methods for Sparse Linear Systems*. Vol. 2. Siam, 2006.

[15] Konigorski, Prof. Dr.-Ing. U.: *Digitale Regelungssysteme I und II, Skriptum*. Technische Universität Darmstadt, 2016.

[16] Adamy, Jürgen: *Systemdynamik und Regelungstechnik II*. Shaker, 2007.